Ironhack Student Portal

preview.my.ironhack.com/lms/courses/course-

Lesson Goals

- Learn how to rename columns in a data frame.
- Change the order of a data frame's columns.
- Filter records based on conditional statements.
- Create additional categories via binning and conditional statements.
- Learn how to one-hot encode categorical variables.
- Combine data frames via merging and concatenation.
- Melt data from wide format into long format.

Introduction

One of the reasons Pandas has become such a popular tool for data analysts over the last few years is because it makes data transformation and manipulation much faster and easier. In this lesson, we will take a look at how to rename and restructure data as we prepare it to be analyzed.

For this lesson, we will be using the same vehicles data set that we practiced importing and exporting in the Import and Export lesson. Let's go ahead and import the CSV version of the data set and see what it actually looks like.

```
import pandas as pd

data = pd.read_csv('vehicles/vehicles.csv')
data.head()
```

	Make	Model	Year	Engine Displacement	Cylinders	Transmission	Drivetrain	Vehicle Class	Fuel Type	Fuel Barrels/Year	City MPG	Highway MPG	Combined MPG	CO2 Emission Grams/Mile	Fuel Cost/Year
	AM General	DJ Po Vehicle 2WD	1984	2.500	4.000	Automatic 3-spd	2-Wheel Drive	Special Purpose Vehicle 2WD	Regular	19.389	18	17	17	522.765	1950
	AM General	FJ8c Post Office	1984	4.200	6.000	Automatic 3-spd	2-Wheel Drive	Special Purpose Vehicle 2WD	Regular	25.355	13	13	13	683.615	2550
	AM General	Post Office DJ5 2WD	1985	2.500	4.000	Automatic 3-spd	Rear-Wheel Drive	Special Purpose Vehicle 2WD	Regular	20.601	16	17	16	555.438	2100
	AM General	Post Office DJ8 2WD	1985	4.200	6.000	Automatic 3-spd	Rear-Wheel Drive	Special Purpose Vehicle 2WD	Regular	25.355	13	13	13	683.615	2550
	ASC Incorporated	GNX	1987	3.800	6.000	Automatic 4-spd	Rear-Wheel Drive	Midsize Cars	Premium	20.601	14	21	16	555.438	2550

Renaming Columns

Data will often come either without column names or with column names that are not as

intuitive as they could be. When this is the case, we want to assign descriptive names to the columns so that we remember what the values in each column represent. Intuitively naming your columns before diving in and analyzing your data is a good habit to develop.

Pandas provides us with a couple different ways to modify column names. For example, the `columns` method will return a list of all the column names in the data set.

```
data.columns
```

```
Index(['Make', 'Model', 'Year', 'Engine Displacement', 'Cylinders',
       'Transmission', 'Drivetrain', 'Vehicle Class', 'Fuel Type',
       'Fuel Barrels/Year', 'City MPG', 'Highway MPG', 'Combined MPG',
       'CO2 Emission Grams/Mile', 'Fuel Cost/Year'],
      dtype='object')
```

If you want to set the column names for every column in the data set, or change the names of multiple columns, you can just pass the `columns` method a list with the same number of column names as the data has columns, and Pandas will update all the column names. In the example below, we are updating the Make column name to Manufacturer and the Engine Displacement column name to Displacement using this method.

```
data.columns = ['Manufacturer','Model','Year','Displacement',
                'Cylinders','Transmission','Drivetrain',
                'Vehicle Class','Fuel Type','Fuel Barrels/Year',
                'City MPG','Highway MPG','Combined MPG',
                'CO2 Emission Grams/Mile','Fuel Cost/Year']

data.columns
```

```
Index(['Manufacturer', 'Model', 'Year', 'Displacement', 'Cylinders',
       'Transmission', 'Drivetrain', 'Vehicle Class', 'Fuel Type',
       'Fuel Barrels/Year', 'City MPG', 'Highway MPG', 'Combined MPG',
       'CO2 Emission Grams/Mile', 'Fuel Cost/Year'],
      dtype='object')
```

If you want to rename just a single column, or just a few columns, you can use the `rename` method and pass a dictionary containing the existing column names and new column names to the columns parameter. Below, we will change the column names we modified in the previous example back to their original column names using the `rename` method.

```
data = data.rename(columns={'Manufacturer':'Make',
                            'Displacement':'Engine Displacement'})

data.columns
```

```
Index(['Make', 'Model', 'Year', 'Engine Displacement', 'Cylinders',
       'Transmission', 'Drivetrain', 'Vehicle Class', 'Fuel Type',
       'Fuel Barrels/Year', 'City MPG', 'Highway MPG', 'Combined MPG',
       'CO2 Emission Grams/Mile', 'Fuel Cost/Year'],
      dtype='object')
```

Changing Column Order

You can also reorder columns in a data frame. To do this, you would create a list containing the data frame's column names in the order you would like them. Then you can just recreate the data frame with the customized ordering as follows.

```
column_order = ['Year','Make','Model','Vehicle Class',
                'Transmission','Drivetrain','Fuel Type',
                'Cylinders','Engine Displacement','Fuel Barrels/Year',
                'City MPG','Highway MPG','Combined MPG',
                'CO2 Emission Grams/Mile','Fuel Cost/Year']

data = data[column_order]
data.head()
```

Year	Make	Model	Vehicle Class	Transmission	Drivetrain	Fuel Type	Cylinders	Engine Displacement	Fuel Barrels/Year	City MPG	Highway MPG	Combined MPG	CO2 Emission Grams/Mile	Fuel Cost/Year
1984	AM General	DJ Po Vehicle 2WD	Special Purpose Vehicle 2WD	Automatic 3-spd	2-Wheel Drive	Regular	4.000	2.500	19.389	18	17	17	522.765	1950
1984	AM General	FJ8c Post Office	Special Purpose Vehicle 2WD	Automatic 3-spd	2-Wheel Drive	Regular	6.000	4.200	25.355	13	13	13	683.615	2550
1985	AM General	Post Office DJ5 2WD	Special Purpose Vehicle 2WD	Automatic 3-spd	Rear-Wheel Drive	Regular	4.000	2.500	20.601	16	17	16	555.438	2100
1985	AM General	Post Office DJ8 2WD	Special Purpose Vehicle 2WD	Automatic 3-spd	Rear-Wheel Drive	Regular	6.000	4.200	25.355	13	13	13	683.615	2550
1987	ASC Incorporated	GNX	Midsize Cars	Automatic 4-spd	Rear-Wheel Drive	Premium	6.000	3.800	20.601	14	21	16	555.438	2550

Filtering Records

When working with data, analysts often need to filter the data based on one or more conditional statements. This is similar to adding a `WHERE` clause to a query in SQL. For example, suppose we needed to filter our data set for all Ford vehicles that had a 6 or more cylinders and a combined MPG of less than 18. We could enter our conditions inside square brackets to subset the data set for just the records that meet the conditions we've specified.

```
filtered = data[(data['Make']=='Ford') &
                (data['Cylinders']>=6) &
                (data['Combined MPG'] < 18)]

filtered.head()
```

	Make	Model	Year	Engine Displacement	Cylinders	Transmission	Drivetrain	Vehicle Class	Fuel Type	Fuel Barrels/Year	City MPG	Highway MPG	Combined MPG	CO2 Emission Grams/Mile	Fuel Cost/Year
1442	Ford	Aerostar Van	1986	2.800	6.000	Automatic 4-spd	Rear-Wheel Drive	Vans	Regular	19.389	15	21	17	522.765	1950
1450	Ford	Aerostar Van	1988	3.000	6.000	Automatic 4-spd	Rear-Wheel Drive	Vans	Regular	19.389	15	20	17	522.765	1950
1452	Ford	Aerostar Van	1989	3.000	6.000	Automatic 4-spd	Rear-Wheel Drive	Vans	Regular	19.389	15	21	17	522.765	1950
1456	Ford	Aerostar Van	1990	4.000	6.000	Automatic 4-spd	Rear-Wheel Drive	Vans	Regular	19.389	15	20	17	522.765	1950
1459	Ford	Aerostar Van	1991	4.000	6.000	Automatic 4-spd	Rear-Wheel Drive	Vans	Regular	19.389	15	20	17	522.765	1950

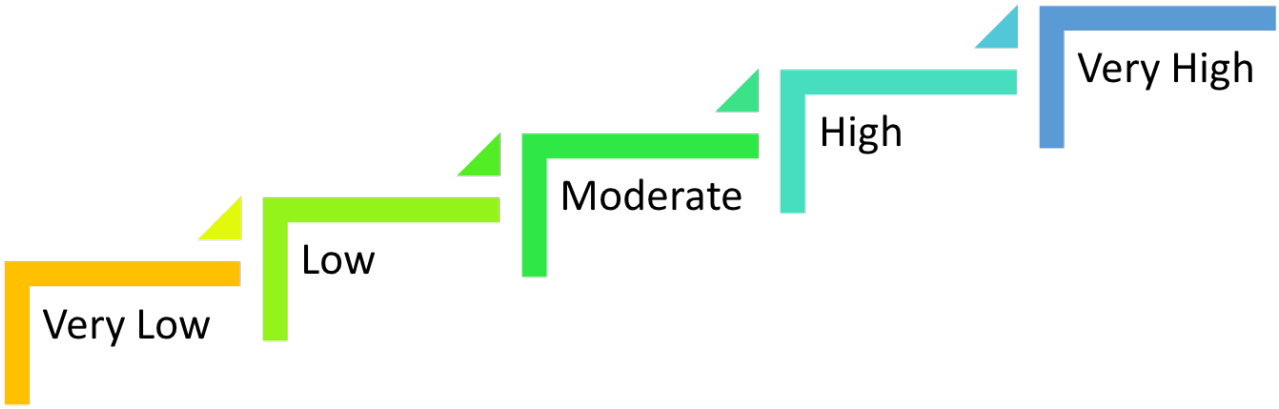
There are a couple of important things to note here. First, when you want to apply multiple conditions, you need to use an "and" operator (&) or an "or" operator (|) between your conditions. The "and" operator will return records where *both* of the conditions surrounding it are true, and the "or" operator will return records where *either* of the conditions surrounding it are true. The second thing to note is that all our conditional statements are enclosed in parentheses. This is easy to forget, but necessary or else your results will be incorrect.

Binning Numeric Variables

When preparing data to be analyzed, one of the things that is useful to do is to create additional categorical variables. Categorical variables allow you to group records in different ways, and each way that you categorize them can provide you with a different perspective when you're conducting your analysis. A common way of creating additional categorical fields is to bin numeric variables in a column based on how relatively high or low they are.

For example, our data set has a Combined MPG variable that tells us how many miles per gallon of gasoline the vehicle can travel - in other words, the vehicle's fuel efficiency. However, it isn't easy to tell whether a vehicle has low or high fuel efficiency just by looking at the numbers. To help with this, we can bin the values in the Combined MPG field into an intuitive number of categories.

For this column, let's choose 5 bins so that we have one middle/moderate bin, two relatively low and high bins outside of that, and then two very low/high bins at the ends.



In Python, let's create a list of labels that will be assigned to our bins.

```
mpg_labels = ['Very Low', 'Low', 'Moderate', 'High', 'Very High']
```

Next, we must determine how we want our data to be binned. There are three main approaches that we can choose from:

- **Equal width bins:** the range for each bin is the same size.
- **Equal frequency bins:** approximately the same number of records in each bin.
- **Custom-sized bins:** the user explicitly defines where they want the cutoff for each bin to be.

If you want equal width bins, you can use the `cut` method in Pandas and pass it the column you want to bin, the number of bins and the list of labels.

```
bins = pd.cut(data['Combined MPG'],5, labels=mpg_labels)
bins.head(10)
```

```
0       Low
1  Very Low
2  Very Low
3  Very Low
4  Very Low
5       Low
6       Low
7       Low
8       Low
9       Low
```

If you want equal frequency bins, you would use the `qcut` method instead with all the same inputs.

```
bins = pd.qcut(data['Combined MPG'],5, labels=mpg_labels)
bins.head(10)
```

```
0        Low
1   Very Low
2   Very Low
3   Very Low
4   Very Low
5       High
6       High
7   Moderate
8       High
9       High
```

Note the difference in results. With equal width binning, there will be some bins that contain more records than others (such as the Low bin). With equal frequency binning, some of those records will be forced into other bins (e.g. the Moderate bin and even the High bin). This is an important consideration when determining how you want to categorize your data.

Finally, if you want custom bin sizes, you can pass a list of bin range values to the `cut` method instead of the number of bins, and it will bin the values for you accordingly.

```
cutoffs = [7,14,21,23,30,40]
bins = pd.cut(data['Combined MPG'],cutoffs, labels=mpg_labels)
bins.head(10)
```

```
0        Low
1   Very Low
2        Low
3   Very Low
4        Low
5   Moderate
6       High
7        Low
8   Moderate
9       High
```

Conditional Categories

Another way to create intuitive additional categories in your data is to create columns based on conditional statements. Earlier in this lesson, we filtered our data based on conditional statements. Here, we will populate the values in a column based on them using the `loc` method.

Our vehicles data set currently has 45 different values in the Transmission field, but one of the key pieces of information embedded in there is whether a vehicle has an automatic or manual transmission. It would be valuable to extract that so that we could

group vehicles by their transmission type. Let's look at how we can create a new TransType column that only contains one of two values for each vehicle: Automatic or Manual.

```
data.loc[data['Transmission'].str.startswith('A'), 'TransType'] = 'Automatic'
data.loc[data['Transmission'].str.startswith('M'), 'TransType'] = 'Manual'
```

We were able to leverage the `str.startswith` method in our conditional statements such that whenever the value in the Transmission field started with an A, we would assign a TransType of Automatic, and when the value started with M, we would assign a TransType of Manual.

One-Hot Encoding Categorical Variables

One-hot encoding is a technique used to expand a single categorical column into as many columns as there are categories. Each column contains a 1 if that record belongs to the category and a 0 if it does not. This is useful for performing analyses where you want to know whether something falls into a specific category or not. It will also be useful when you learn about machine learning, as one-hot encoding makes it easier for some algorithms to interpret and find patterns in categorical data.

To perform one-hot encoding on a column, you can use the Pandas `get_dummies` method and pass it the column you would like to one-hot encode.

```
drivetrain = pd.get_dummies(data['Drivetrain'])
drivetrain.head()
```

	2-Wheel Drive	2-Wheel Drive, Front	4-Wheel Drive	4-Wheel or All-Wheel Drive	All-Wheel Drive	Front-Wheel Drive	Part-time 4-Wheel Drive	Rear-Wheel Drive
0	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0
7	0	0	0	0	0	1	0	0
8	0	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0

Combining Data Frames

Another useful thing to do with data sets is to combine them. Pandas provides us with a few different ways to do this. The first way is by merging. Merging is similar to creating a join in SQL, where you can specify common fields between the two tables and then include information from both in your query. Pandas has a `merge` method that functions in a similar way.

To illustrate, let's create a data frame that has the average Combined MPG for each

Make using the `groupby` method. We will merge that average into our data frame, joining on Make, so that we can see how fuel efficient a vehicle is in comparison to the other vehicles made by the same manufacturer.

```
avg_mpg = data.groupby('Make', as_index=False).agg({'Combined MPG':'mean'})
avg_mpg.columns = ['Make', 'Avg_MPG']

data = pd.merge(data, avg_mpg, on='Make')
data.head(10)
```

Engine Displacement	Cylinders	Transmission	Drivetrain	Vehicle Class	Fuel Type	Fuel Barrels/Year	City MPG	Highway MPG	Combined MPG	CO2 Emission Grams/Mile	Fuel Cost/Year	TransType	Avg_MPG
2.500	4.000	Automatic 3-spd	2-Wheel Drive	Special Purpose Vehicle 2WD	Regular	19.389	18	17	17	522.765	1950	Automatic	14.750
4.200	6.000	Automatic 3-spd	2-Wheel Drive	Special Purpose Vehicle 2WD	Regular	25.355	13	13	13	683.615	2550	Automatic	14.750
2.500	4.000	Automatic 3-spd	Rear-Wheel Drive	Special Purpose Vehicle 2WD	Regular	20.601	16	17	16	555.438	2100	Automatic	14.750
4.200	6.000	Automatic 3-spd	Rear-Wheel Drive	Special Purpose Vehicle 2WD	Regular	25.355	13	13	13	683.615	2550	Automatic	14.750
3.800	6.000	Automatic 4-spd	Rear-Wheel Drive	Midsize Cars	Premium	20.601	14	21	16	555.438	2550	Automatic	16.000
2.200	4.000	Automatic 4-spd	Front-Wheel Drive	Subcompact Cars	Regular	14.982	20	26	22	403.955	1500	Automatic	21.507
2.200	4.000	Manual 5-spd	Front-Wheel Drive	Subcompact Cars	Regular	13.734	22	28	24	370.292	1400	Manual	21.507
3.000	6.000	Automatic 4-spd	Front-Wheel Drive	Subcompact Cars	Regular	16.480	18	26	20	444.350	1650	Automatic	21.507

Another method that Pandas provides for combining data sets is concatenation. The pandas `concat` method lets you attach columns or rows from one data set onto another data set as long as both data sets have the same number of rows (if you are concatenating columns) or columns (if you are concatenating rows). Let's take a look at examples for each of these.

For column concatenation, we can use the one-hot encoded drivetrain data frame we created earlier and add those columns to our vehicles data set. Note that the data frames passed to the `concat` method must be in a list and you set the axis parameter to 1 in order to indicate that you are concatenating columns.

```
data = pd.concat([data, drivetrain], axis=1)
```

To illustrate row concatenation, let's create two new data frames based on conditional filters from our original data frame - one containing only Lexus vehicles and another containing only Audi vehicles. We will then combine them using the `concat` method into a lexus_audi data frame that contains only vehicles manufactured by those two companies.

```
lexus = data[data['Make']=='Lexus']
audi = data[data['Make']=='Audi']

lexus_audi = pd.concat([lexus, audi], axis=0)
```

Again, note that the data frames are passed as a list and that this time the axis is set to 0 to specify that we are concatenating rows.

Melting Data Into Long Format

Our vehicles data set currently has a wide format, where there is a column for each attribute. However, some analytic and visualization tasks will require that the data be in a long format, where there are a few variables that define the entities and then all other attribute information is condensed into two columns: one containing the column/attribute names and another containing the value for that attribute for each entity. Pandas makes it easy to format data this way with the `melt` function. For example, suppose we were going to perform some analysis or visualization task where we needed the Year, Make, and Model to identify the vehicles and then we also needed the City MPG, Highway MPG, and Combined MPG fields for performing various calculations. Below is how we would melt the data into the proper format.

```
melted = pd.melt(data, id_vars=['Year','Make','Model'],
                 value_vars=['City MPG','Highway MPG','Combined MPG'])
melted.head(10)
```

	Year	Make	Model	variable	value
0	1984	AM General	DJ Po Vehicle 2WD	City MPG	18
1	1984	AM General	FJ8c Post Office	City MPG	13
2	1985	AM General	Post Office DJ5 2WD	City MPG	16
3	1985	AM General	Post Office DJ8 2WD	City MPG	13
4	1987	ASC Incorporated	GNX	City MPG	14
5	1997	Acura	2.2CL/3.0CL	City MPG	20
6	1997	Acura	2.2CL/3.0CL	City MPG	22
7	1997	Acura	2.2CL/3.0CL	City MPG	18
8	1998	Acura	2.3CL/3.0CL	City MPG	19
9	1998	Acura	2.3CL/3.0CL	City MPG	21

As you can see, the column names have been stacked into the the *variable* field and their corresponding values have been stacked into the *value* field.

Summary

In this lesson, we learned a variety of ways to manipulate data frames. We started by covering how to change the names of a data frame's columns and the order in which those columns appear. We then learned how to filter records based on conditional logic

and how to add create additional categorical columns that may be useful to us when analyzing and modeling data later. From there, we learned how to combine data from multiple data sets and also how to melt data to make it easier to compute upon.