**Categorical pitfalls**

Hello again and welcome to the last chapter on working with categorical data in python. Our first lesson will be on common pitfalls when using categorical data.

**Used cars: the final dataset**

We briefly discussed memory savings in chapter one, but let's revisit it here. To do that, we need to introduce our last dataset. The used cars dataset contains information on over 38,000 used cars including the manufacturer, model, and sale price. This dataset is commonly used to practice building predictive models.

**Huge memory savings**

In chapter 1, we discussed how using a categorical Series can save a lot of memory. This isn't always the case. Consider the manufacturer name column. It has 55 unique entries and is currently stored as an object. If we compare the number of bytes used, using the nbytes attribute, of this column as an object and compare it to this column as a category, we reduce memory usage by almost 90 percent.

1. 1 <https://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html>



**Little memory savings**

However, if we convert a numerical column, or even an object column with lots of different unique values, we will see less memory savings. Consider the odometer value Series. It has over 6,000 unique values. If we convert this to categorical, we still save memory, but this time it's only a 60% memory reduction. This is because the number of bytes needed for a categorical column is proportional to the number of categories.

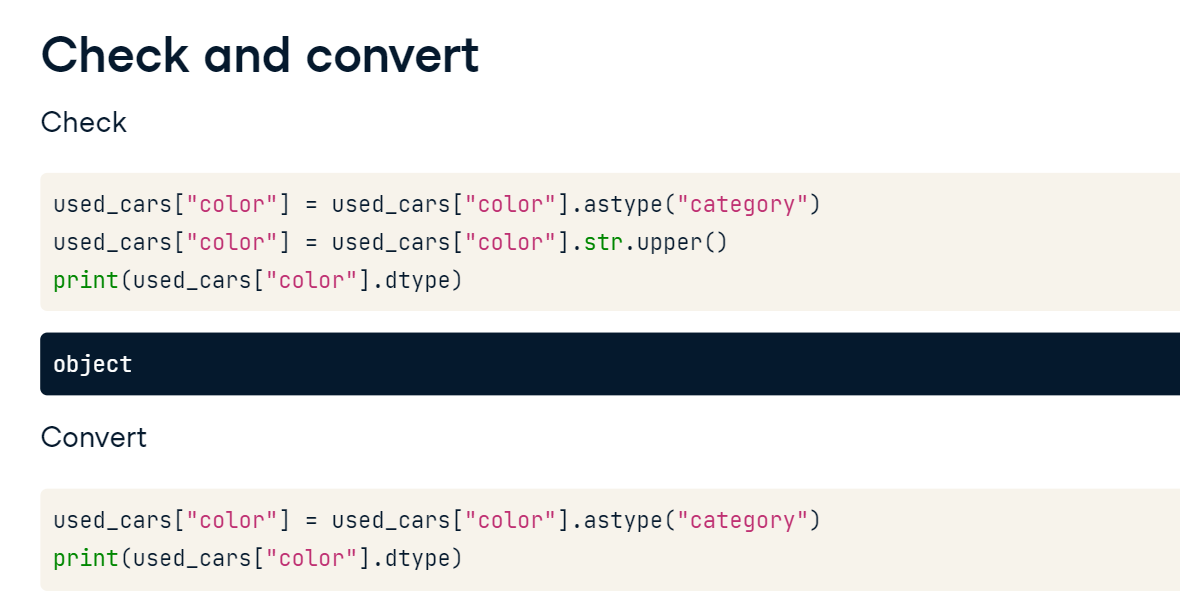


**Using categories can be frustrating**

If you are not carefully thinking through each change you make to your data, you'll likely run into issues. Consider these potentially frustrating challenges: First - the dot-str accessor object and the pandas apply method will always convert the Series back to an object, forcing you to convert it back to a category. Second, the common methods for updating and setting categories discussed in chapter two do not all handle missing categories the same way. And finally, a categorical Series is not a NumPy array. Using NumPy functions on categorical Series usually produces errors. Let's quickly look at how to handle each problem.

**Check and convert**

For the last time in this course, we will check our output and convert it back to a category if necessary! If you make changes to a Series using dot-str or dot-apply, you must convert the Series back to categorical. Always check your columns dtype using dot-dtype, and convert it if necessary using dot-astype and specifying category.



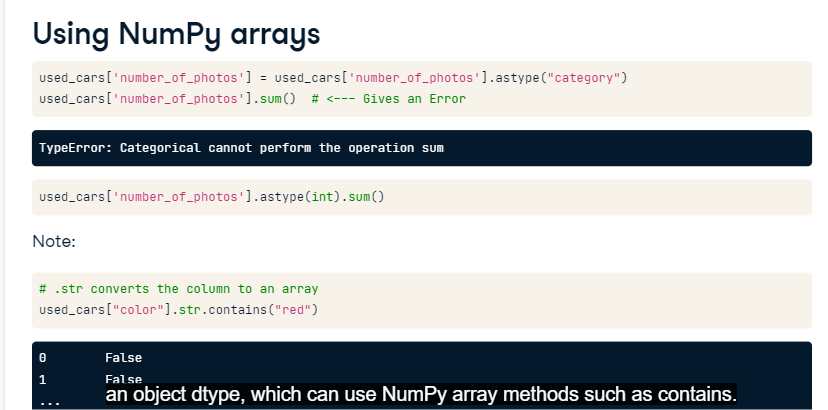
**Look for missing values**

Anytime you are updating categories, whether that is setting, adding, or removing, use value-counts to make sure the changes you made worked as intended. For the color Series, we have set the categories to only black, silver, and blue. Using the value counts method with dropna set to False, we see that over 18,000 entries have become NaN values. If this was not intended, we may need to use a different method for updating the categories.



**Using NumPy arrays**

Although a categorical series is not a NumPy array, it doesn't mean we can't turn it into an array. Humor me while we let the number of photos of a used car be a categorical column. Using NumPys sum function will give a type error, as NumPy doesn't understand the categorical dtype. However, we can quickly convert the series to an integer and us the sum method. This is common when using a categorical column that is an integer, such as the number of stars for a hotel. Note that the dot-str accessor object transforms the Series to an object dtype, which can use NumPy array methods such as contains.



**Label encoding**

Pitfalls aside, let's assume we have prepared the data correctly. In this lesson, we will begin to look at a very useful technique for categorical columns, label encoding.

**What is label encoding?**

Label encoding is a technique that codes categorical values as integers. In Python, these codes often start at 0 and end at n - 1, where n is the number of categories. A -1 code is often used to indicate any missing values. Label encoding is used to save memory and to simplify responses when using survey data. Although the codes created through label encoding can be used in machine learning models, this is not the best encoding method for machine learning. We will cover a better technique in the next lesson.

**Creating codes**

To create an encoding, let's convert the manufacturer name of the used cars dataset to a categorical column. We can get a label encoding by using cat-dot-codes, which will convert the values to integers. If the column is not ordinal, the codes will be assigned in alphabetical order. Here we make a new Series, called manufacturer code, that contains the integer values that correspond to manufacturer names.



**Check output**

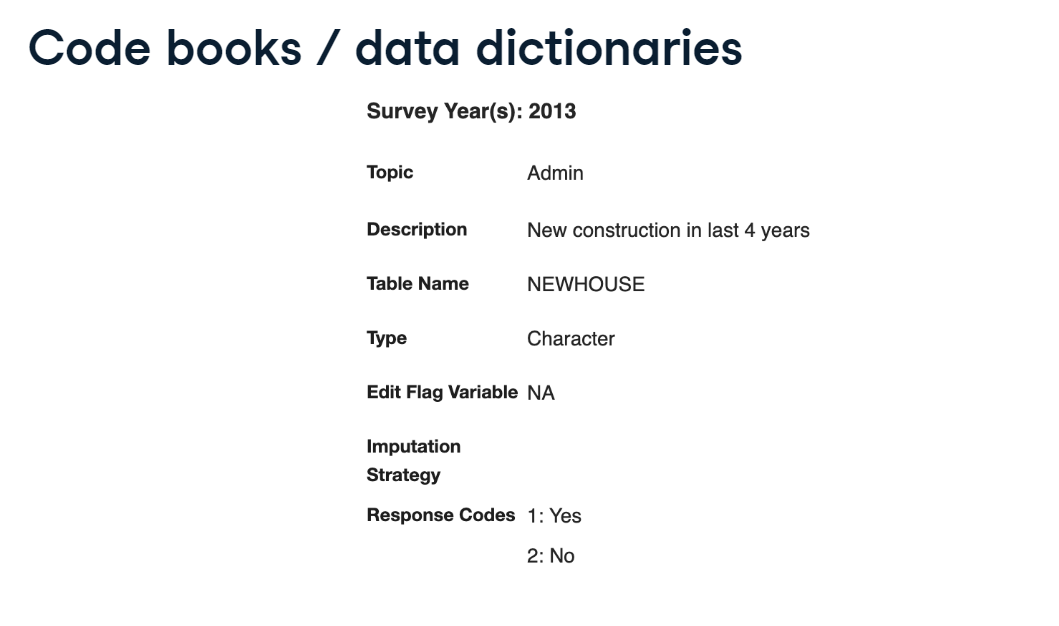
Let's check the output. Subaru is the first manufacturer name in the dataset, but is the 46th name in alphabetical order. It has been assigned a code of 45. Chrysler is the 9th in order, and has been given a code of 8.



**Code books / data dictionaries**

As mentioned at the start of the video, label encoding is often used in surveys. The responses and their corresponding codes are often kept in a code book or a data dictionary. Consider this variable from the American Housing Survey, where a 1 represents YES and a 2 represents NO, for if a house was built in the last four years.

1. 1 https://www.census.gov/data-tools/demo/codebook/ahs/ahsdict.html

****

**Creating a code book**

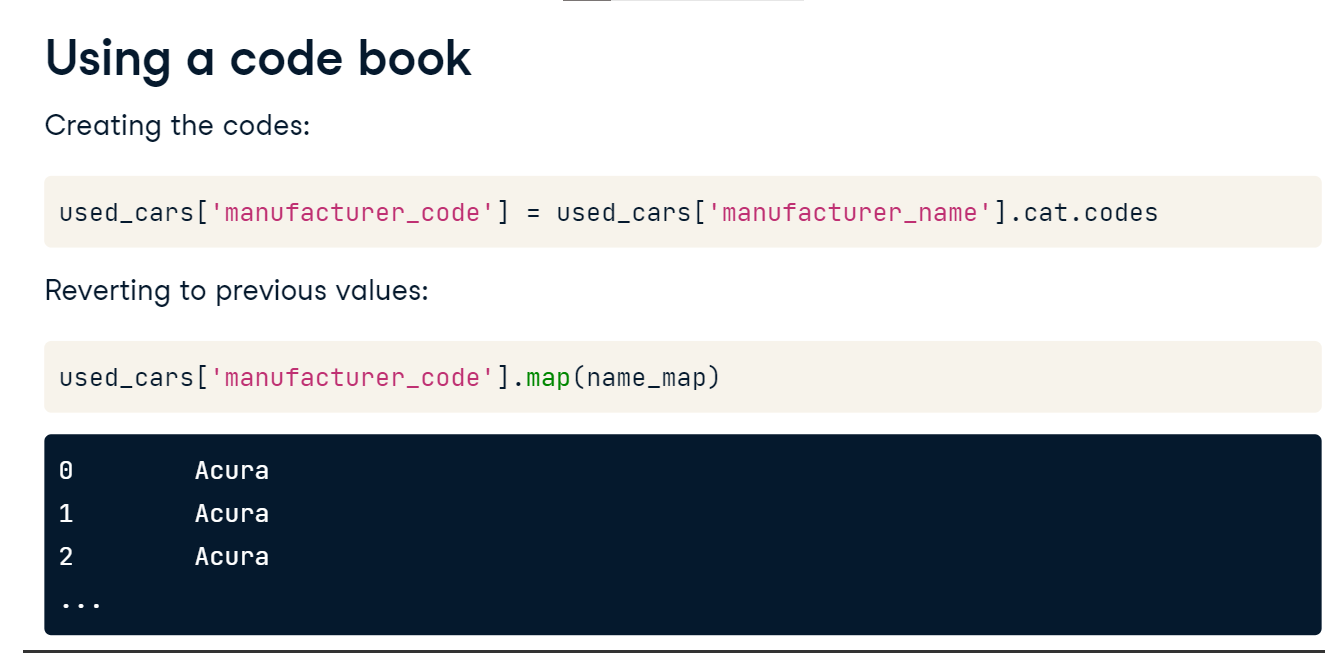
If you do create a label encoding and save the new dataset, you will want to create a map from the new codes to the old values. This can be done by creating an object for the codes and an object for the categories. Python has a built-in function called, zip, that can be used to iterate through the entries of codes and categories one at a time. If we place the zip function inside of a dictionary call, the unique combinations of codes and categories will be added as key-value pairs. Printing the name-map reveals which code maps to which category. As stated earlier, Subaru was the first manufacturer name to show up in the dataset, but is 46th in alphabetical order.



**Using a code book**

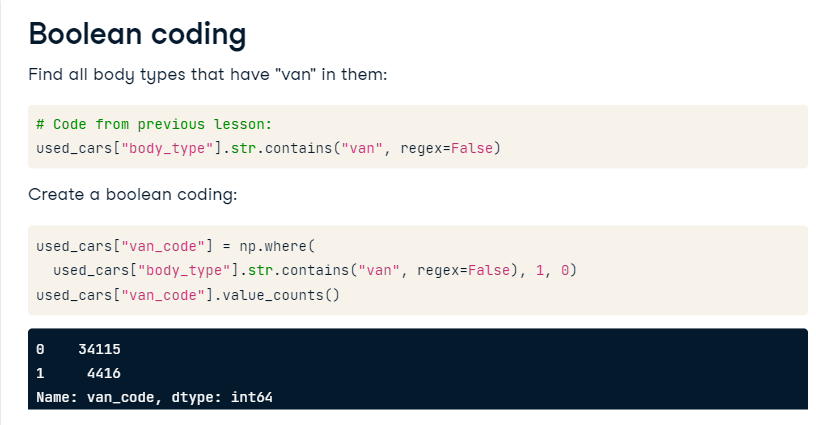
We can then use the name map we just created to convert the codes back to their original categorical values. This happens a lot when using surveys, as responses are often stored as numbers to save on memory. We can convert the column back to the original categories using the dot-map method and specifying our name map. dot-map is similar to dot-replace, and it will replace the Series values based on the keys of the name-map and their corresponding values. dot-map is used in this context because we have a complete mapping. Every single value in the manufacturer code column should have a key in the name-map dictionary.

1. 1 https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.map.html

****

**Boolean coding**

When creating a label encoding for a categorical column, it is common to create a Boolean code that represents a group of categories. For example, say we wanted to create a boolean code for all cars that were vans. We have already seen how to find the cars with a body type that contains the letters v-a-n. We can use the NumPy function, where, to say anytime this statement is true, we want to have a 1 value, and anytime this statement is false we want to have a 0. Looking at the output, only about 4,400 of the 38,000 used cars have van in their body type name.

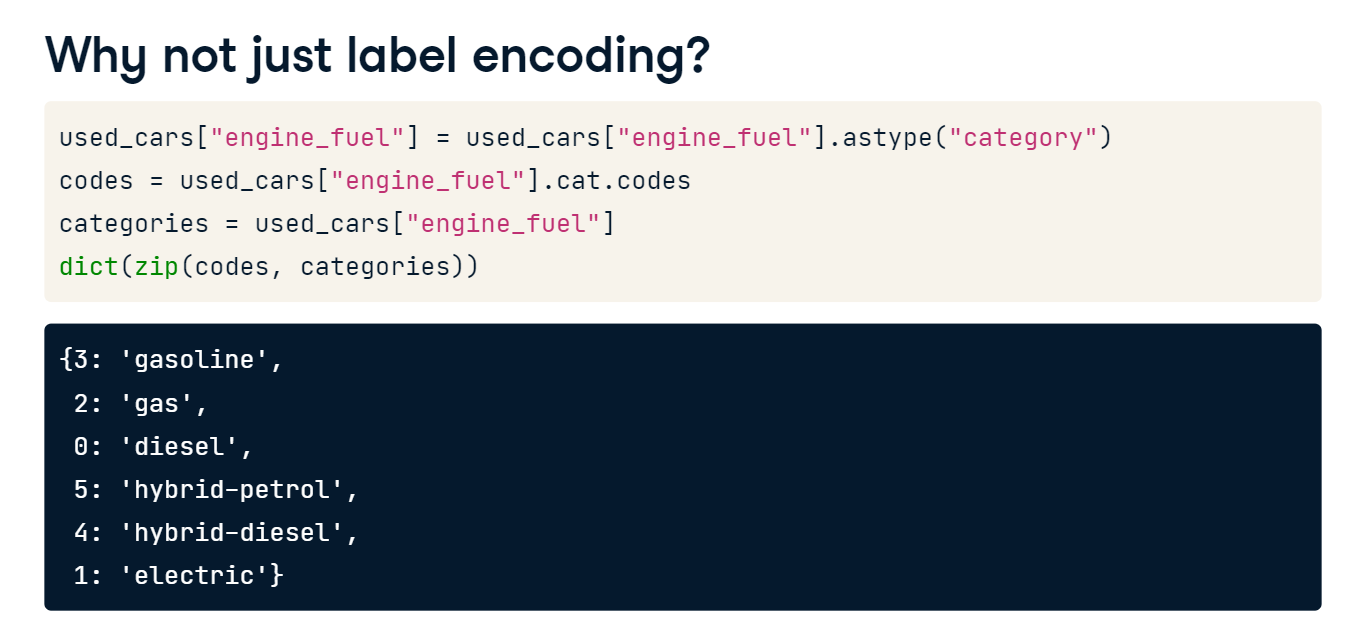


**One-hot encoding**

Hello - and welcome to our final lesson. Here we focus on a short introduction to one-hot encoding.

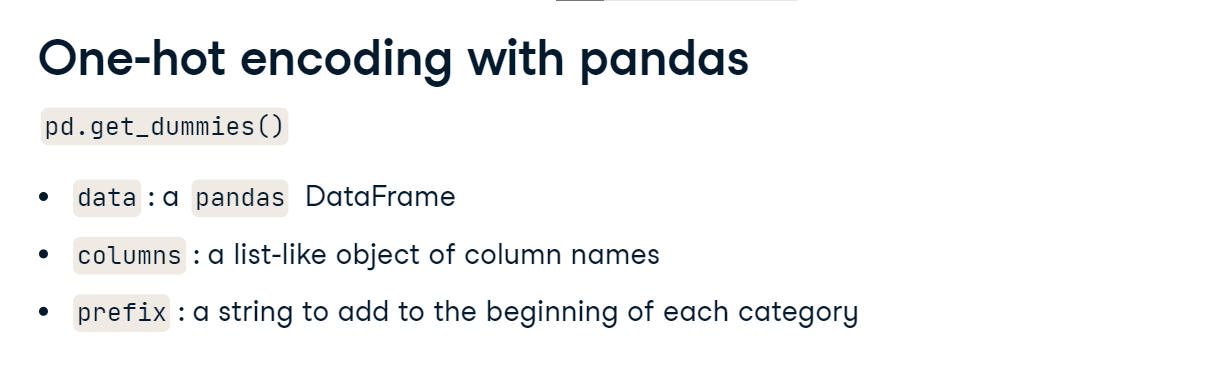
**Why not just label encoding?**

In the last lesson, we learned about label encoding. Consider this mapping that we had used in the last lesson. What do you notice? You might recall that the keys are assigned in alphabetical order, or the order of the category if the column is ordinal. If you try to use a column of these codes in a machine learning model, the algorithm might misinterpret their meaning. Remember, algorithms train on numbers! For example, diesel with a value of 0 might be given less weight than gasoline with a value of 3. We need a better approach.



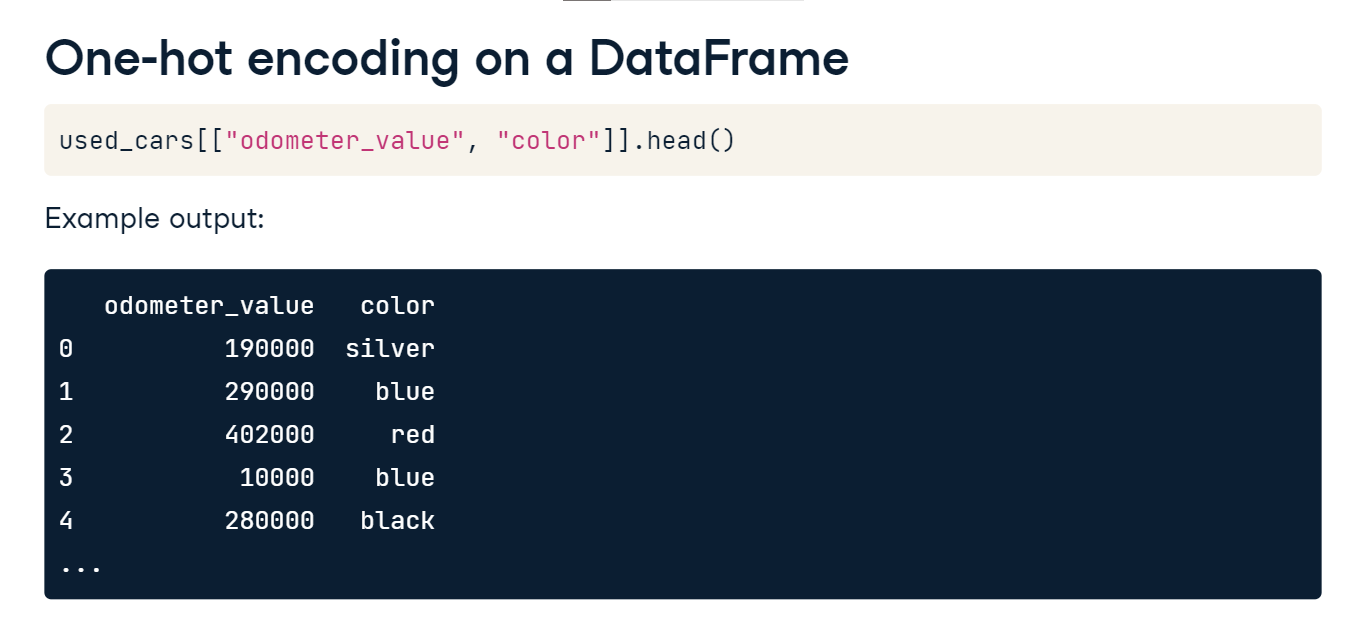
**One-hot encoding with pandas**

In a previous exercise, we created a zero-one column for a single value of a single column. Fortunately, we can do this for all values of a single column, or even all columns at one time using the pandas function get-dummies. One-hot encoding is the process of creating dummy variables, hence the name get-dummies. This function has several inputs. Data is the DataFrame we are using. Columns is a list of the column names we want to encode, and prefix is a string that will be added to the beginning of the new column names. Let's look at a few examples in action.



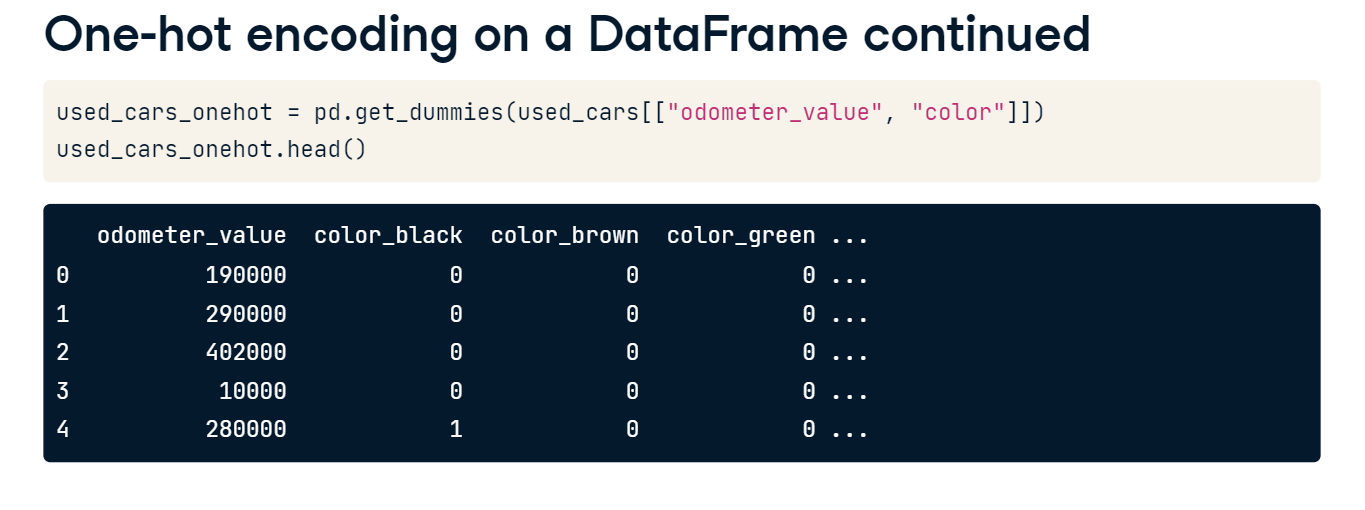
**One-hot encoding on a DataFrame**

Calling pd-dot-get-dummies on a DataFrame will apply one-hot encoding on all object and categorical columns. In this example, the DataFrame has just two columns after we subset it, odometer-value and color. Take a look at their current values.



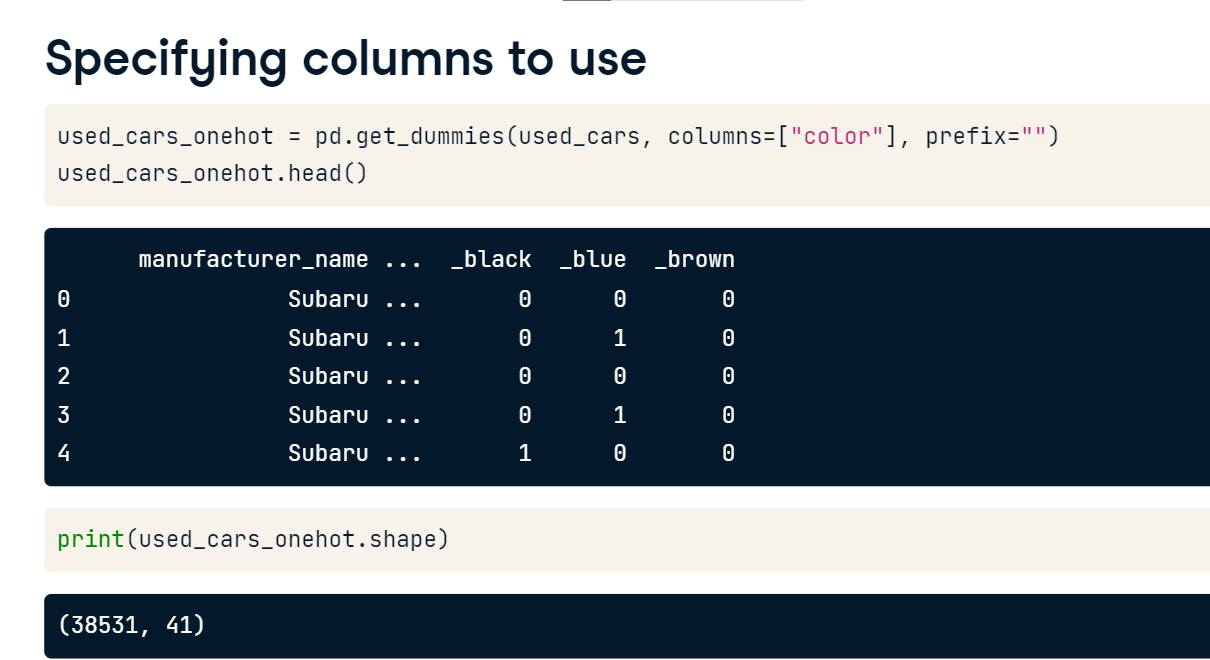
**One-hot encoding on a DataFrame continued**

The syntax is pd-dot-get-dummies with the DataFrame as the first parameter. Used-cars-onehot will now have all object and categorical columns one-hot encoded. Any numeric columns will remain the same. Color had twelve unique color values, so we now have one column per color. A 0 indicates that the car was not that color, while a 1 indicates that the car was that color. Our new DataFrame will have 13 total columns. One for the odometer value and 12 for the new color columns. The original color column is dropped.



**Specifying columns to use**

It may be important to use get-dummies on only a subset of columns, as you may not want to encode all object or categorical variables. In this example, we are one-hot encoding on the color column only. Since we are only doing one column, we have decided to set the prefix to blank. All other columns in the used cars dataset will be left alone, but the color column will be converted to twelve columns, one for each color. Notice the new names for the columns: underscore-black and underscore-blue. In this example, we did not subset the used cars dataset. There are now 41 total columns. 29 from the original dataset, and 12 for the new color columns.



**A few quick notes**

There are a few things to consider when using one-hot encoding. First, if your columns have a lot of unique values, an equal amount of new columns will be created. Training machine learning models on a lot of columns may lead to a problem known as overfitting, something we would like to avoid. Just look what happens when we use the entire used cars dataset. We now have over 1,000 total columns. Secondly, NaN values do not get their own column. This is OK though. If all created columns for a variable are 0, this indicates that the original column was blank. There is no need to have a column for missing values.

