**Membership constraints**

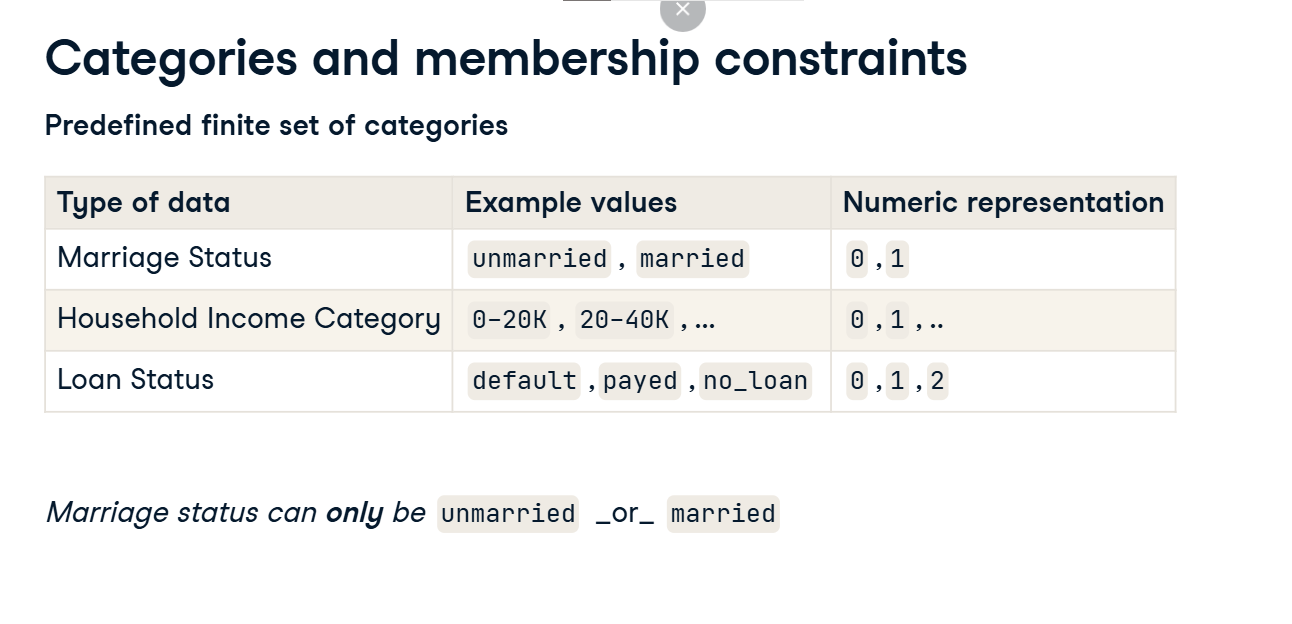
Fantastic work on Chapter 1! You're now equipped to treat more complex, and specific data cleaning problems.

**In this chapter**

In this chapter, we're going to take a look at common data problems with text and categorical data, so let's get started.

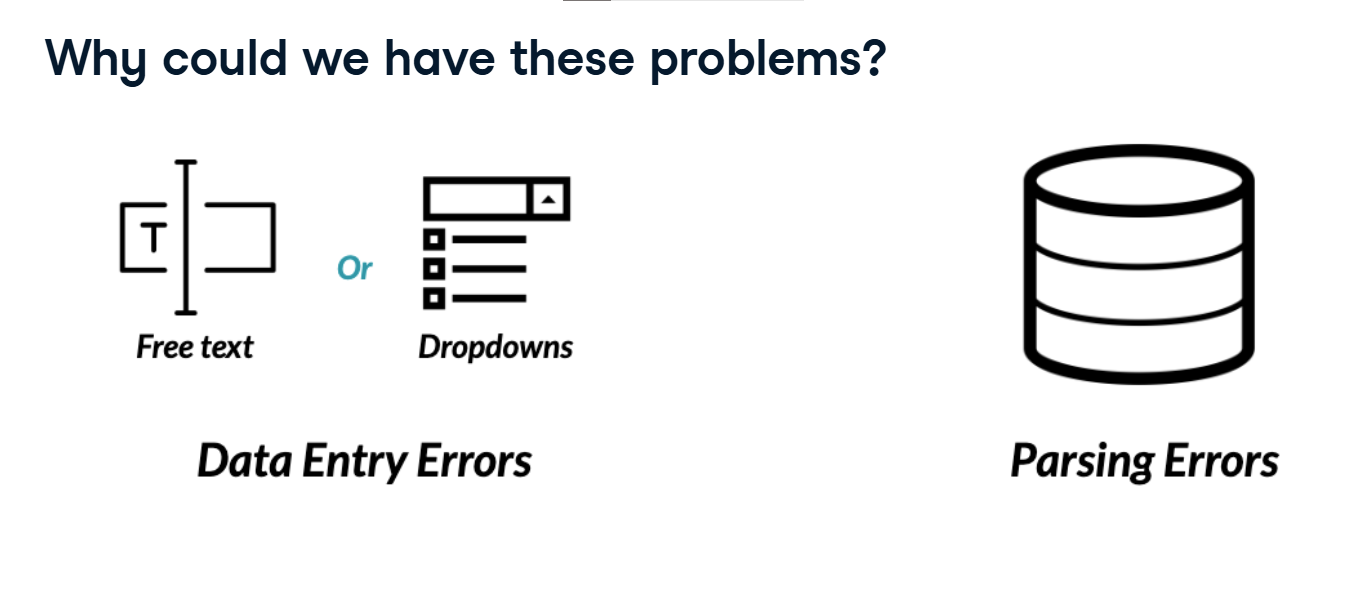
**Categories and membership constraints**

In this lesson, we'll focus on categorical variables. As discussed early in chapter 1, categorical data represent variables that represent predefined finite set of categories. Examples of this range from marriage status, household income categories, loan status and others. To run machine learning models on categorical data, they are often coded as numbers. Since categorical data represent a predefined set of categories, they can't have values that go beyond these predefined categories.



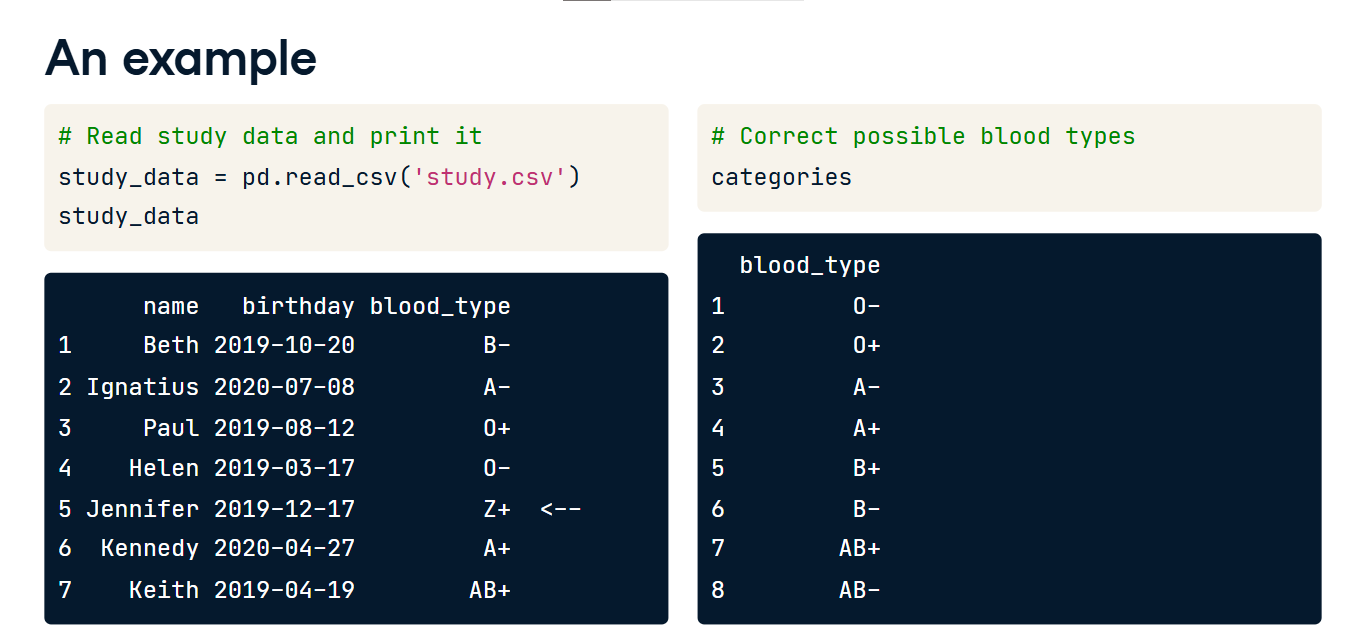
**Why could we have these problems?**

We can have inconsistencies in our categorical data for a variety of reasons. This could be due to data entry issues with free text vs dropdown fields, data parsing errors and other types of errors.



**How do we treat these problems?**

There's a variety of ways we can treat these, with increasingly specific solutions for different types of inconsistencies. Most simply, we can drop the rows with incorrect categories. We can attempt remapping incorrect categories to correct ones, and more. We'll see a variety of ways of dealing with this throughout the chapter and the course, but for now we'll just focus on dropping data.



**An example**

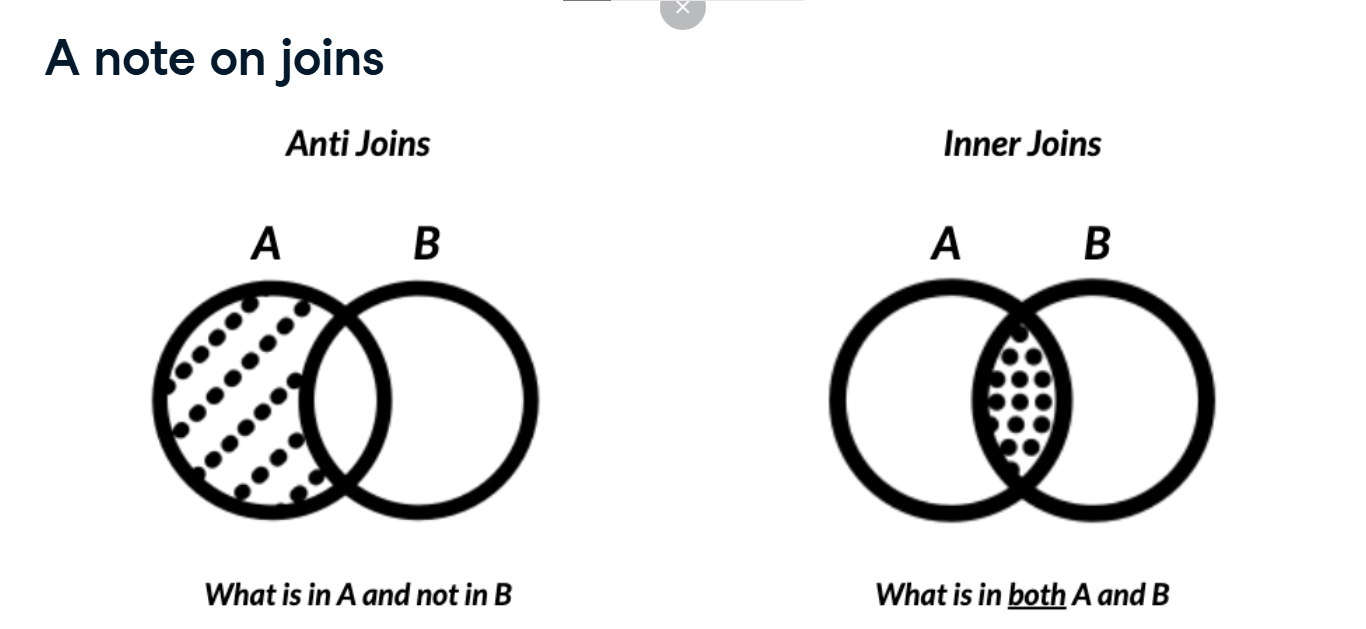
Let's first look at an example. Here's a DataFrame named study\_data containing a list of first names, birth dates, and blood types. Additionally, a DataFrame named categories, containing the correct possible categories for the blood type column has been created as well.

**An example**

Notice the inconsistency here? There's definitely no blood type named Z+. Luckily, the categories DataFrame will help us systematically spot all rows with these inconsistencies. It's always good practice to keep a log of all possible values of your categorical data, as it will make dealing with these types of inconsistencies way easier.

**A note on joins**

Now before moving on to dealing with these inconsistent values, let's have a brief reminder on joins. The two main types of joins we care about here are anti joins and inner joins. We join DataFrames on common columns between them. Anti joins, take in two DataFrames A and B, and return data from one DataFrame that is not contained in another. In this example, we are performing a left anti join of A and B, and are returning the columns of DataFrames A and B for values only found in A of the common column between them being joined on. Inner joins, return only the data that is contained in both DataFrames. For example, an inner join of A and B, would return columns from both DataFrames for values only found in A and B, of the common column between them being joined on.

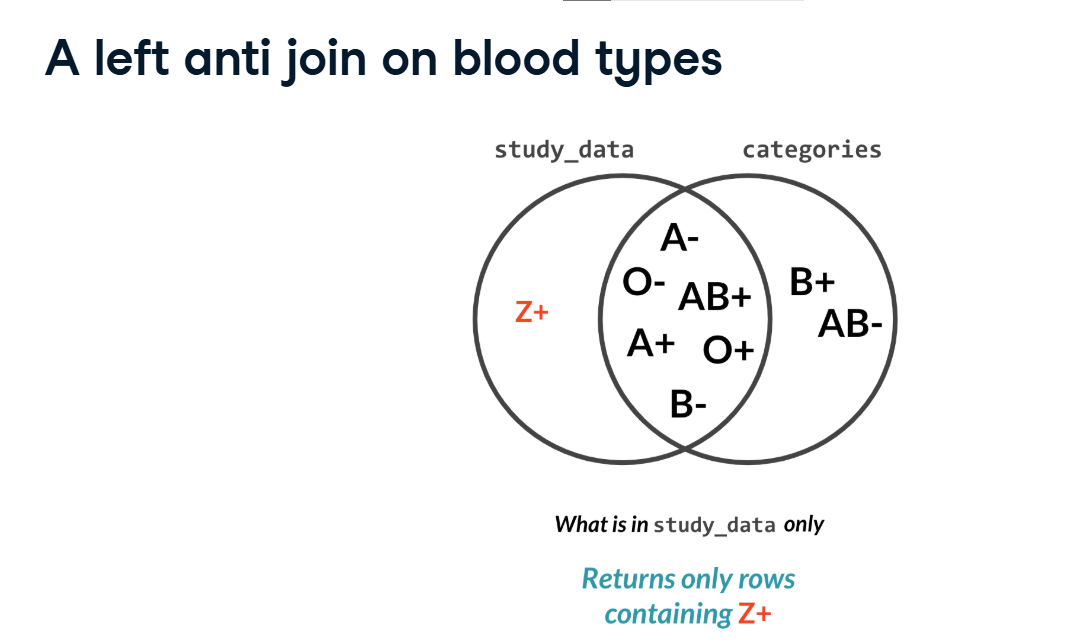


**A left anti join on blood types**

In our example, an left anti join essentially returns all the data in study data with inconsistent blood types,

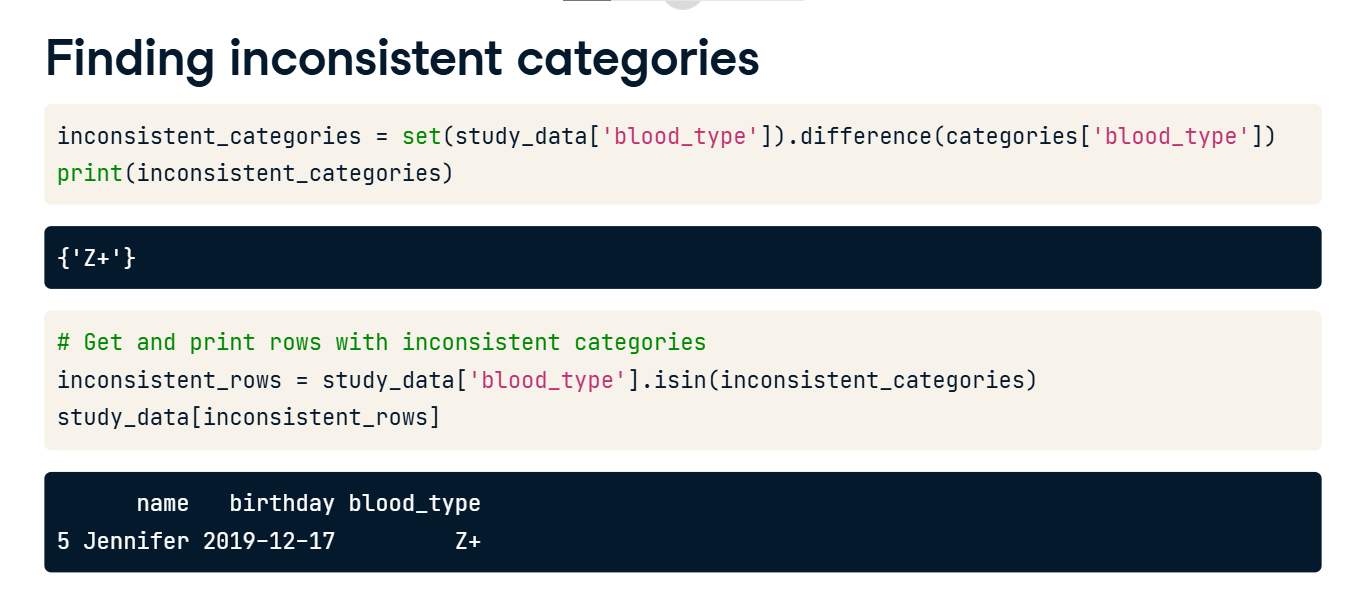
**An inner join on blood types**

and an inner join returns all the rows containing consistent blood types signs.



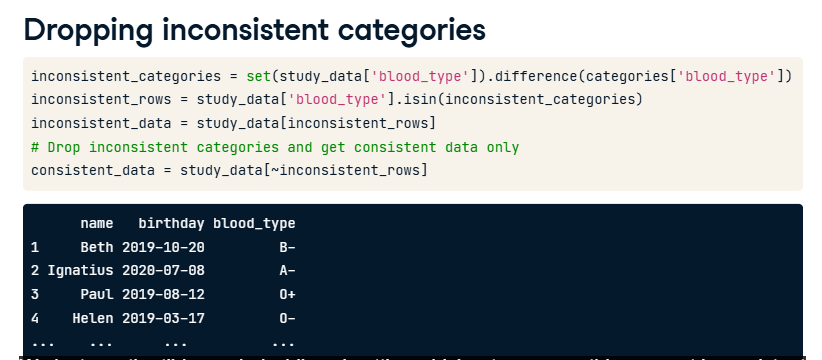
**Finding inconsistent categories**

Now let's see how to do that in Python. We first get all inconsistent categories in the blood\_type column of the study\_data DataFrame. We do that by creating a set out of the blood\_type column which stores its unique values, and use the difference method which takes in as argument the blood\_type column from the categories DataFrame. This returns all the categories in blood\_type that are not in categories. We then find the inconsistent rows by finding all the rows of the blood\_type columns that are equal to inconsistent categories by using the isin method, this returns a series of boolean values that are True for inconsistent rows and False for consistent ones. We then subset the study\_data DataFrame based on these boolean values, and voila we have our inconsistent data.



**Dropping inconsistent categories**

To drop inconsistent rows and keep ones that are only consistent. We just use the tilde symbol while subsetting which returns everything except inconsistent rows.

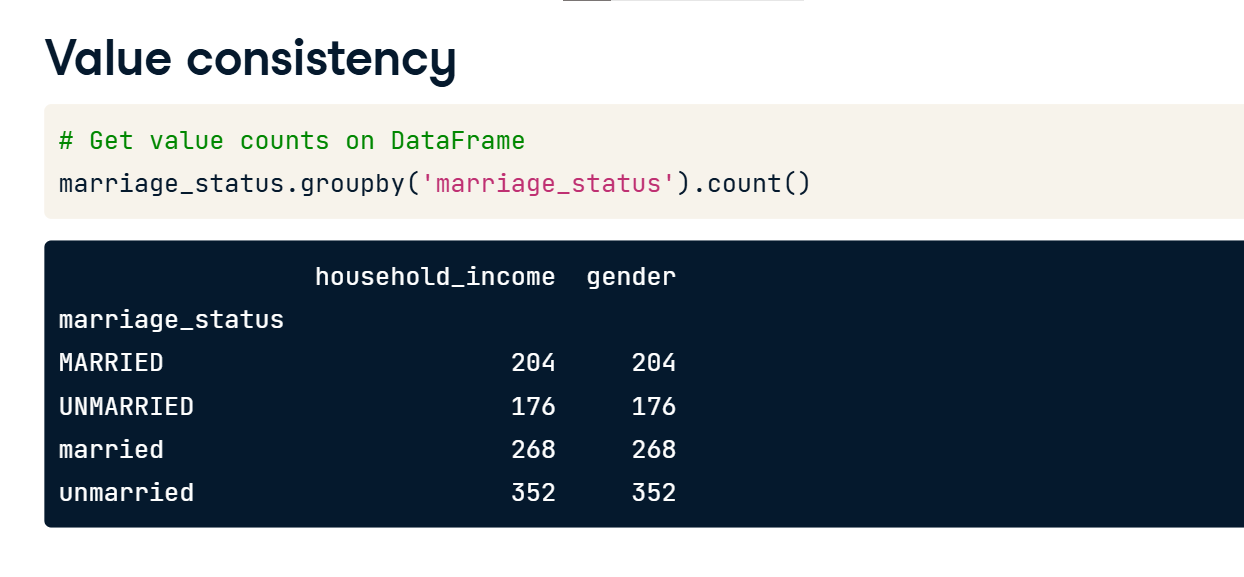


**Categorical variables**

Awesome work on the last lesson. Now let's discuss other types of problems that could affect categorical variables.

**What type of errors could we have?**

In the last lesson, we saw how categorical data a value membership constraint has, where columns need to have a predefined set of values. However, this is not the only set of problems we may encounter. When cleaning categorical data, some of the problems we may encounter include value inconsistency, the presence of too many categories that could be collapsed into one, and making sure data is of the right type.



**Value consistency**

Let's start with making sure our categorical data is consistent. A common categorical data problem is having values that slightly differ because of capitalization. Not treating this could lead to misleading results when we decide to analyze our data, for example, let's assume we're working with a demographics dataset, and we have a marriage status column with inconsistent capitalization. Here's what counting the number of married people in the marriage\_status Series would look like. Note that the dot-value\_counts() methods works on Series only.



For a DataFrame, we can groupby the column and use the dot-count() method.

To deal with this, we can either capitalize or lowercase the marriage\_status column. This can be done with the str-dot-upper() or dot-lower() functions respectively.



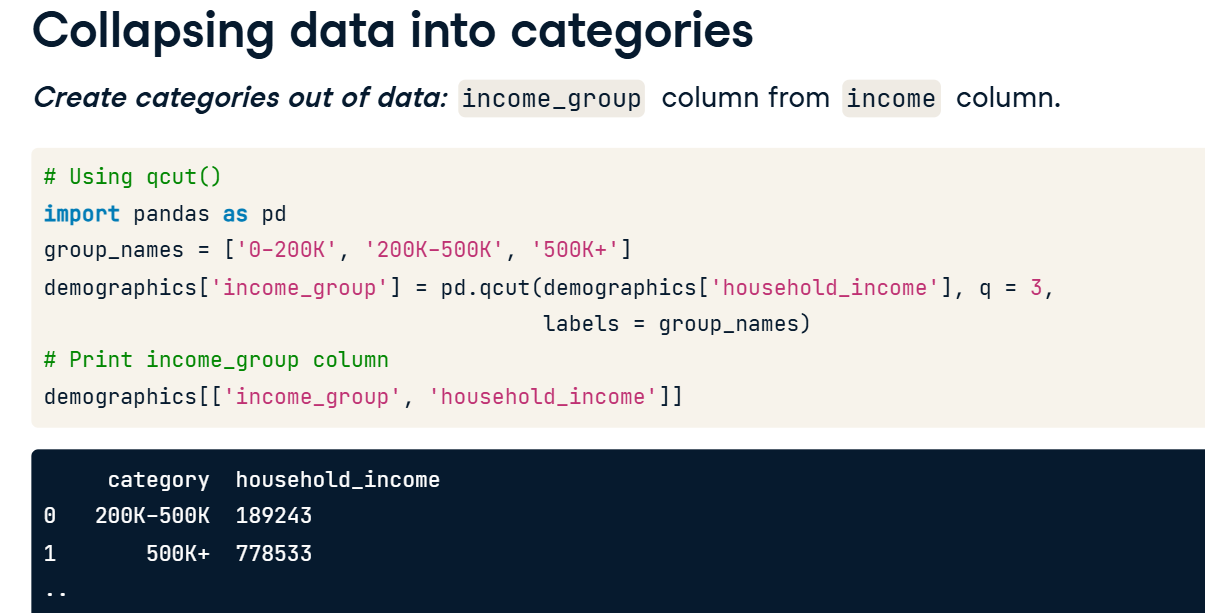
**Value consistency**

Another common problem with categorical values are leading or trailing spaces. For example, imagine the same demographics DataFrame containing values with leading spaces. Here's what the counts of married vs unmarried people would look like. Note that there is a married category with a trailing space on the right, which makes it hard to spot on the output, as opposed to unmarried.

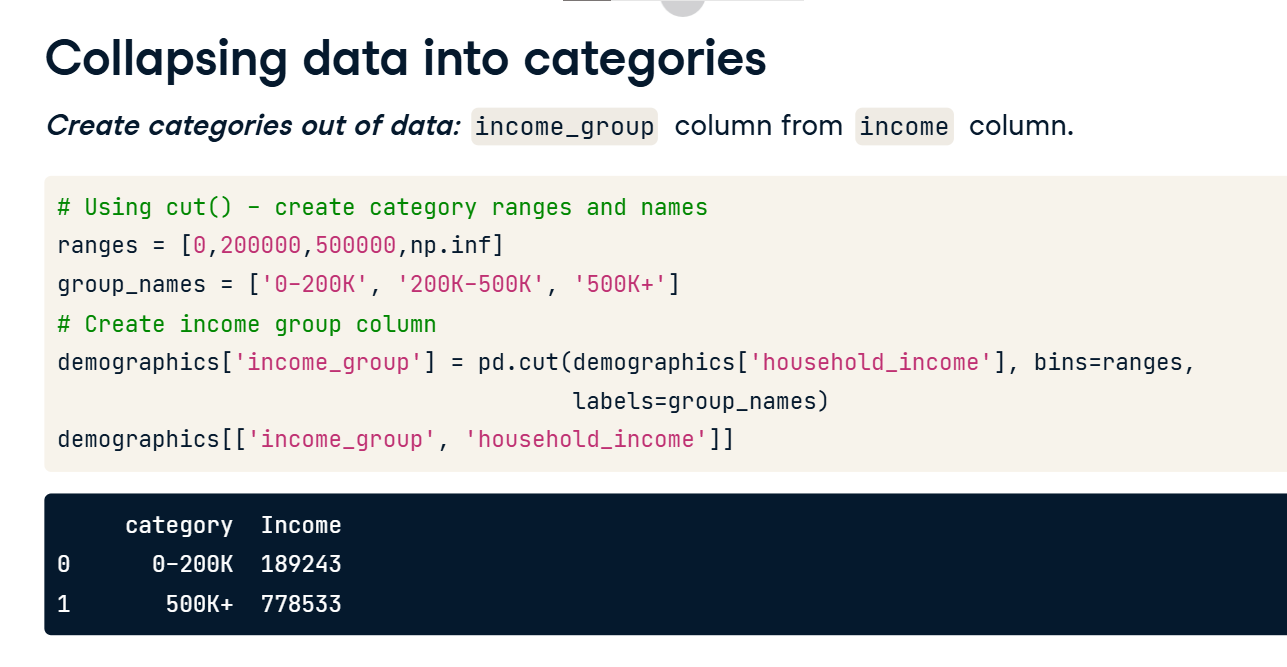
To remove leading spaces, we can use the str-dot-strip() method which when given no input, strips all leading and trailing white spaces.

**Collapsing data into categories**

Sometimes, we may want to create categories out of our data, such as creating household income groups from income data. To create categories out of data, let's use the example of creating an income group column in the demographics DataFrame. We can do this in 2 ways. The first method utilizes the qcut function from pandas, which automatically divides our data based on its distribution into the number of categories we set in the q argument, we created the category names in the group\_names list and fed it to the labels argument, returning the following. Notice that the first row actually misrepresents the actual income of the income group, as we didn't instruct qcut where our ranges actually lie.

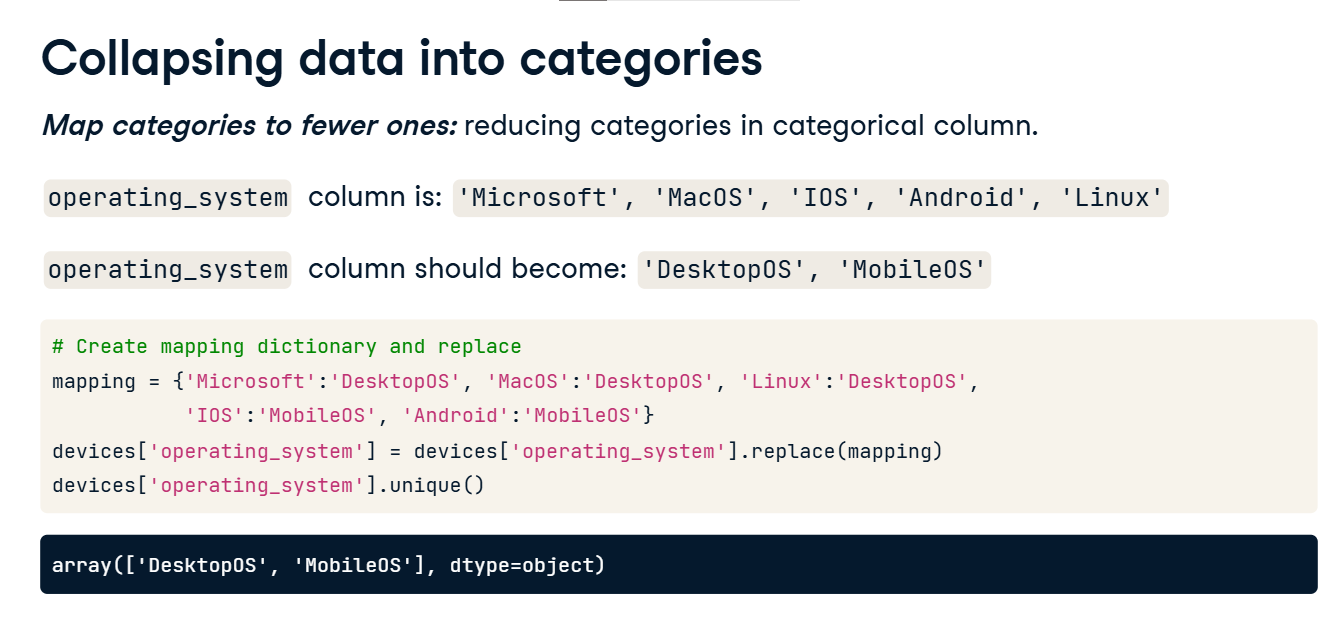


We can do this with the cut function instead, which lets us define category cutoff ranges with the bins argument. It takes in a list of cutoff points for each category, with the final one being infinity represented with np-dot-inf(). From the output, we can see this is much more correct.



**Collapsing data into categories**

Sometimes, we may want to reduce the amount of categories we have in our data. Let's move on to mapping categories to fewer ones. For example, assume we have a column containing the operating system of different devices, and contains these unique values. Say we want to collapse these categories into 2, DesktopOS, and MobileOS. We can do this using the replace method. It takes in a dictionary that maps each existing category to the category name you desire. In this case, this is the mapping dictionary. A quick print of the unique values of operating system shows the mapping has been complete.



**Cleaning text data**

Good job on the previous lesson. In the final lesson of this chapter, we'll talk about text data and regular expressions.

**What is text data?**

Text data is one of the most common types of data types. Examples of it range from names, phone numbers, addresses, emails and more. Common text data problems include handling inconsistencies, making sure text data is of a certain length, typos and others.

**Example**

Let's take a look at the following example. Here's a DataFrame named phones containing the full name and phone numbers of individuals. Both are string columns. Notice the phone number column.

We can see that there are phone number values, that begin with 00 or +. We also see that there is one entry where the phone number is 4 digits, which is non-existent. Furthermore, we can see that there are dashes across the phone number column. If we wanted to feed these phone numbers into an automated call system, or create a report discussing the distribution of users by area code, we couldn't really do so without uniform phone numbers.

**Example**

Ideally, we'd want to the phone number column as such. Where all phone numbers are aligned to begin with 00, where any number below the 10 digit value is replaced with NaN to represent a missing value, and where all dashes have been removed. Let's see how that's done!

**Fixing the phone number column**

Let's first begin by replacing the plus sign with 00, to do this, we use the dot str dot replace method which takes in two values, the string being replaced, which is in this case the plus sign and the string to replace it with which is in this case 00. We can see that the column has been updated.

**Fixing the phone number column**

We use the same exact technique to remove the dashes, by replacing the dash symbol with an empty string.

Now finally we're going to replace all phone numbers below 10 digits to NaN. We can do this by chaining the Phone number column with the dot str dot len method, which returns the string length of each row in the column. We can then use the dot loc method, to index rows where digits is below 10, and replace the value of Phone number with numpy's nan object, which is here imported as np.

**Fixing the phone number column**

We can also write assert statements to test whether the phone number column has a specific length,and whether it contains the symbols we removed. The first assert statement tests that the minimum length of the strings in the phone number column, found through str dot len, is bigger than or equal to 10. In the second assert statement, we use the str dot contains method to test whether the phone number column contains a specific pattern. It returns a series of booleans for that are True for matches and False for non-matches. We set the pattern plus bar pipe minus, the bar pipe here is basically an or statement, so we're trying to find matches for either symbols. We chain it with the any method which returns True if any element in the output of our dot-str-contains is True, and test whether the it returns False.

**But what about more complicated examples?**

But what about more complicated examples? How can we clean a phone number column that looks like this for example? Where phone numbers can contain a range of symbols from plus signs, dashes, parenthesis and maybe more. This is where regular expressions come in. Regular expressions give us the ability to search for any pattern in text data, like only digits for example. They are like control + find in your browser, but way more dynamic and robust.

**Regular expressions in action**

Let's a look at this example. Here we are attempting to only extract digits from the phone number column. To do this, we use the dot str dot replace method with the pattern we want to replace with an empty string. Notice the pattern fed into the method. This is essentially us telling pandas to replace anything that is not a digit with nothing. We won't get into the specifics of regular expressions, and how to construct them, but they are immensely useful for difficult string cleaning tasks, so make sure to check out DataCamp's course library on regular expressions.