Colorado State University Libraries
CSU Libraries
Training and Instruction
Transcription of Data cleaning using R, 11/2/2017

Collection: Training and Instruction (10217/195518)
Title: Data cleaning using R
Date: 11/2/2017
File Name: FACFLIBR_CaC-DataCleaning_TM_20171102.mp4
Date Transcribed: November 2024
Transcription Platform: Konch AI

BEGIN TRANSCRIPTION

[00:01 - 01:49] Tobin Magle: Hi, and welcome to Coding and Cookies. I'm Tobin Magle, the cyber infrastructure facilitator at Colorado State University. Today, we're going to be discussing how to use the R programming language in RStudio to clean up messy data. This is based on the Data Carpentry OpenRefine curriculum and in our project tutorial. Data cleaning is important because data is rarely perfect when we receive it, and it can take forever to fix by hand. Using a programming language like R to fix common data errors can save you time and make the process less error prone. In this session, we're going to discuss how to use arguments to read.csv to clean up data during import, correcting imported data in ways like recoding, data type conversion, correcting misspellings, removing whitespace and splitting columns, and also how to filter and sort your data. The dataset that we will be using is an ecological study of small animals. It is a tidy dataset, in that each column is a variable and each row is an observation. In this context, each time an animal was observed in the study, a whole new row was added to the spreadsheet. Variables about each observations were recorded like species, sex, date, and location. However, errors in the data, like misspellings, need to be fixed. To set up for this lesson, download the quickstart files link to on this slide. Then open the project in RStudio by double clicking on the .Rproj file. Look at the README file. This file contains informations about the variable names, column classes, and spelling of species names. First we're going to talk about how to use read.csv arguments like header, col.names, colClasses, and na.string to clean up the data during import. First, let's try reading the survey's file without using any of these arguments. I've downloaded the files that were linked to on the slide and unzipped them, and they're in this folder on my desktop.

[01:49 - 03:33] Tobin Magle: So when I open the folder, you can see that we have an R., or .Rproj file. So I'm going to double click on that. And it opens up in our studio window. So make sure that you have RStudio installed on your machine. And we can see that we have a blank script here that just has a bunch of comment notes that scaffolds what we're going to go over in the class. And down here on the lower right, there's a files window that says- lists all the files that are in the working directory that you're in. So you can see that we've got the our project file here, that we've been working going to be working with. We've got the script that's empty. We've got a complete version of the script, if you're interested in that. There's also a data folder that contains a dataset called surveys, no header. And if we go back to the main folder, we've also got a README file. So if you click on the README, here's some good information that you should include about any dataset that you're going to be sharing with others. So, there's a general description at the top. You can think of this as like an abstract. And then we've got sort of a data dictionary where we have the column names, what type of data structure they represent. So either character factor or numeric and then just general description of what they are. We've also got some useful stuff in here, and that they're just vectors of the column names, vectors of the column types, and then a list of correctly spelled species names for the species names column. And we're going to be coming back to this README file later in the lesson. All right. So to get started, we said that we wanted to use read.csv to read in the data without any extra parameters.

[03:33 - 05:22] Tobin Magle: We're only going to use the required parameters. So to figure out which ones are required, we can go down here into the console where our codes actually be running. Insert a read.csv and it will open up a help file for us over here on the right where the files used to be. So if we go over here to read.csv, we can see that file doesn't have a default specified. So that means that it needs something in here. It needs us to specify something. Otherwise, it doesn't know what file to read in. The rest of these have defaults. Things that will go into them. So header is going to equal be equal to true unless you tell it not to. So, we don't need to specify those, but we'll see how you, specifying those can help you clean up your data while you're reading it in. Okay. So, we're going to create a new data frame called surveys [typing] 'eys'. Then we're gonna use the assignment operator to take the output of the function we're going to do next and put it into the data survey's data frame. And the function we're going to be using is read.csv, as I've said before. And then, we're going to specify that we want the file to read in, to be 'data/surveys_no_header.csv'. So this is specifying that if we're looking at the files we're in the working directory here. We want it to go into the data folder, and then inside the data folder we want it to read this file with this name. Okay. So when I hit run. Now we have the surveys dataset loaded. Let's check it out to see what's up. Hmm. What do you think looks wrong with this dataset?

[05:24 - 07:10] Tobin Magle: Well, if we look at the top or the column header for each thing, we've got things that have an X in front of it and then a number, which is kind of weird. I'm not really sure why I would name the files like that, but then when we get to this, the species name column, we can kind of see, oh well, it took the first line of data and made it beat the header. And that's because if we look in the help file, if we look under read.csv, header equals true. So, this function is assuming that the first line in your CSV file is going to be the column headers. So it's pushing them up there. And that's not what we want. But, luckily like I said we can use these, these parameters or arguments to fix this right on the read. To fix this issue, use the header equals false argument inside read.csv. All right. So let's copy this line that we used to read in the file. And we'll paste it in. And then we're going to add a new argument by putting a comma after our first one and then returning over to the next line. And in this case, we're interested in the header argument. And if you look down here and by the default it's true, so it's expecting the top line in your file to be a header. We have kind of a clue here and that it's saved as surveys no header. So instead we'll say header equals false. And if we run this, when we look at the survey's data frame, we can see that the headers are generic numbers. So Variable 1, Variable 2. Instead of shifting this line up. And, yeah. Now we have all of the data actually in the spreadsheet and not in the header column.

[07:10 - 09:07] Tobin Magle: But the next thing that we have to worry about is what to, how to specify the names of these columns. Because v one through v whatever is not very descriptive. To fix this issue, we can use the call names argument. See the README file for the syntax to assign the column names. Let's not reinvent the wheel, and copy and paste what we've already done with reading this file. And then, as we said on the previous slide, we're going to add another argument. But instead of typing out all of the different column names, we're just going to consult the README file. So now you can see why it's, it's kind to give your, your code and your data good metadata for people that are about to use it later. So here's the syntax for how to specify the column headings. You can see that they're in a vector. So it's inside this C function which stands for concatenate. So I'm copying this. And then I'm going to paste it in here. I actually can remove this last comma because there's nothing coming after it. So now we just need to say, tell read.csv which argument we want this to go into. If you look back in the help file, we have col.names up here in read table because all of the arguments from read.table translate into here. It's just read.csv has different defaults than read.table. So if I type col.names in here and then run the script and look at surveys, now we have descriptive column headers which is awesome. So just to review for a second. The first argument file is specifying what file we want to read in. The second argument header is specifying that our, the file that we're reading in does not have a header. And the third argument, col.names, is specifying what the names of the columns of the thing that we read in, because there is no header with the column names in it.

[09:09 - 10:50] Tobin Magle: A less obvious data cleaning issue that can be dealt with upon import is data type. The type of data of a column determines what you can do with it. For example, you can do math and numerical columns. You can use columns with categorical data to create subgroups in your data and filter by subgroup and free text columns are good for human readable notes and text mining. When reading the file, the read.csv function tries to guess the data type based on what's in the column. If the column is all numbers, it becomes either int or num, depending on whether there are any decimal numbers in the column. You can do math in these columns, like calculating the mean. If the column contains any text, it will become either factor or character. By default, read.csv makes text columns into a factor. Factors are stored as integers with text labels, which lets you easily group your data by this column or subset to a particular text factor. For example, the sex column could be stored as a factor as a list of ones and twos with the labels, male and female. If you use the strings as factors equals false argument in read.csv, all the columns will be stored as character variables. Instead of storing the values as integers, they are stored as raw text. One downside to this approach is that the column will take up more room in your computer's memory. This format is better for unstructured notes about your dataset. Because this type is hard, type of text is harder to group into coherent categories. R has a set of functions that helps you determine what data types read.csv assigned to your columns. The str function, which stands for structure, takes a data frame as input and outputs information about the structure of your data frame, such as how many rows and columns there are, the data type for each column, and a preview of the first records in the data set.

[10:51 - 12:33] Tobin Magle: The class function takes any variable as input and tells you what the data type is. The summary function takes data frame and gives you summary statistics of each column. Note how this function treats numeric and categorical data differently in the demo. Let's see how these functions work. All right. So let's see how these functions work. Let's say, what does the structure function first str and then it takes a data frame as input. So you can say surveys. And send this to the console. And you see, we get output to the console that says that surveys itself as a data frame, which we were expecting. And then it lists how many observations are here and then how many variables. Then under this header, it has a row for each column in the dataset. And then the second part says what type of, what type of number it is. So here we've got a bunch of integers up top, some factors where there's some text, some numeric values for latitude and longitude. And seams for hindfoot length and weight, okay? So actually, a shortcut for getting this information is to go up to the Environment tab and click the little blue arrow to the left of surveys, and it opens up a dropdown with this information right here in the same format. Okay. So that structure gives kind of a broad overview. But if you want to kind of narrow it down and only figure out the structure of one thing at a time, you can use the function class. Let's start out by trying it with surveys. So, if you run

this, we have output to the console that says data frame. So, it's essentially the same information that we have right here, in the structure part.

[12:35 - 14:21] Tobin Magle: But we, we can also do this which is probably more helpful for our purposes here with a single column, instead of the entire data frame. So we type class and then surveys. And then for this example, let's use plot. So if I send this to the console, you can see that plot is an integer. And if we look up here at the structure of the data frame, you can look at the plot column and see that it's also an integer. Okay. So, last thing we're going to do summary. And we'll do it for surveys. And send this to the console. You can scroll up here and see that for things like recordID, month, day, year, these are all numeric. So it's going to give you the min quartiles median, mean and max in each value. Okay. So. And then we go over here for things like species which has text in the column. So all that, the only summary statistic that I can really do for species is to tell you, how many are in each species group. And one of the problems here is that it gets cut off after, let's see 1, 2, 3, 4, 5 groups. So it's just got this big other group that has like 12,000 things in it, which isn't that useful for us. If we look at the species column here, we can see that there's 48 levels. So, there's other ways to extract the full information that we can talk about later. Another thing that we should probably discuss is this locality column. You can see that locality has a bunch of notes in it that don't seem terribly structured, so maybe that shouldn't be a factor. And also the plot column here. The plots are labeled 1 to 24.

[14:21 - 16:05] Tobin Magle: So you can see the max value is 24. But this isn't something that you're ever going to be doing math on. So this might be something that our guests are wrong on. So, we can readdress that later when we talk about which column classes or which columns should be classes, and we read the README file to see what the, the data creators actually suggested. And I think that's pretty much it. Yeah. Let's do an exercise now. First look at the Readme file and look at the column list. Do you agree with what the data creators want the column classes to be? Does this match with the read.csv function guessed when reading in the data frame? All right. So let's go back into the working directory and open up the README file. And we can see this list of column names. So recordID through weight. If you look at these, you can see that recordID is a character. This is because each line in the spreadsheet is going to have a unique number, and it's not really going to save any space or be useful to have this as a factor, because you can't group by things, essentially because there's only one of each. We've also changed month, day and year as factor. This is primarily because we're probably not going to want to do math on these. And for a raw data set, it's not terrible to have them as a factor because you could say, oh, I only want data from a certain year, or I only want data from a certain month across years. However, there are ways that R can use that have or data types that are specifically tied to being able to do cool things with dates and times.

[16:05 - 17:58] Tobin Magle: But we're not, we don't have time to go over that in this lesson. We changed plot to a factor because there's 24 different plots in this study, but it isn't, they're just labels. The plots could easily could have been letters of the alphabet instead of numbers. So you're never going to want to do math on these. These are more of a category. Species data factor. Scientific name stays a factor. Locality becomes character because as we discussed in the last demo, it's pretty much just free text notes. There's not really a good way for R to know how to group them. Latitude, longitude stay numeric. County is a factor. State country also factors. Sex is a factor and then hindfoot length and weight are going to be numeric because they actually do have decimal numbers in there. Okay. So, I think that is pretty much it for discussing why each of these variables has, why the authors decided to put those there. But, if we look at the survey's data frame here on the right, you can see that there are differences here. So we need to do something when we're reading in the dataset to make sure that these, these columns become the right data type. And that will be on the next slide. To specify the class of each variable use the call classes argument. Like the col.names argument. This argument takes a list of strings created by the concatenate function. To the column classes and make them be with the data creators intend them to be, I'm going to come up here and I'm going to copy. Oops. He'll copy this and put it under here under the comment. And then I'm going to add a comma here and return this down so that we can add another parameter here.

[17:59 - 19:47] Tobin Magle: Now I'm going to go into the README file. We can see at the top we have the part we were just looking at, and if we scroll a little bit farther down we can see here the syntax for the colClasses argument. So if I copy this, you can type colClasses. And then put the the order of classes that they intend. And now when we run this, we can see that the surveys dataframe updated a little bit and now month, day and year factors. So is plot. And, let's see. Locality is a character. So it meant. It changed the column classes from what read.csv guessed by default to what the data creators actually intended them to be. The last topic we're going to cover in importing data is missing data. The read.csv function allows you to specify codes for missing data points. The character string 'NA' is the default for indicating missing data. Other common missing data codes are 'negative 999' and blank. You can specify what your code is using the na.strings argument when using read.csv. The strings argument takes a list of missing values indicators. In this case NA and a blank, as indicated in the README file. Let's see how this works with the Readme file. And we can see down here that it says missing data is designated as follows. So for numbers, it's NA and for text, it's a blank. So it makes a little bit of sense to do these separately because there's never going to be a number called NA. But especially since we have two letter species ID, species ID is here. You can see that if you look to this factor, there is a level that is blank. So that just means that they didn't record what species it was when they made the observation.

[19:48 - 21:47] Tobin Magle: All right. So let me go back to data cleaning. Let's copy this. And add another argument. So na strings. And we want that to be 'NA' and a blank. So now when we run this, you can look over here at species and it was 48 levels. Now it's 47 levels and it starts with 'AB' instead of blank. So, now we have all the correct information in here. And this is pretty much all we can do to correct things using the read.csv functions parameters. Now, we're going to move on to talking about data cleaning after import. Such topics that we're going to cover our type conversion, faceting, recoding, removing white space, splitting and combining columns, and clustering to fix spelling errors. We've already seen how to specify what data type you want each column to be while reading in the data. However, you don't have to reload the data to change the data type, if you change your mind. You can do this on the fly using as character, as numeric, and as factor functions. Let's say you want to convert a factor variable to a character variable. Let's use the plot variables in example. To verify what type the variable is, we can use the class function as we learned about earlier. We can also convert from factor to character using the s character function. And then we'll use class scan to verify that it actually changed. Inverting classes. It's always a good practice to make sure that the plot or the variable that you're working with is the class that you think it is already. So to confirm that the plot variable is a factor, we're going to use the class function again and the on surveys and then the plot column. So when you run this, you can see that the plot column is a factor.

[21:48 - 24:01] Tobin Magle: One other way to look at this is to go click on the blue orb, next to the surveys dataset on the left, and then look down here at the plot column. It says it's a factor with 24 levels. So, we're getting what we think we want here to begin with. But let's say we wanted to turn it into a character, instead of a factor. We can do that by doing an assignment operation into it. So, [typing] surveys 'dollar' sign plot. And then we can say [typing] as dot character surveys 'dollar' sign plot. And we have surveys plot twice here because if we just run this part of it without assigning it, it's going to output it to the screen. Instead of into the variable. And then when we look at plot, it's still a factor. But if we run this entire line, we don't get any output to the screen. And if we look at plot, we can see that it's a character here. And that's how you take a factor variable and convert it into character. Now, we can go from character to numeric using the as numeric function. It works well in this case because all of the character character either numbers in it. However, if I can't figure out how to do this case with single character string string to a number, and however, if I can't figure out how to go from what's in the character numeric factor, we're going to look at the plot ID that we're looking at, and we can see that it's character or sorry character to numeric. All right. So, we've confirmed that it is in fact a character. Now we can say plot. Oops. Surveys. Sorry. [typing] surveys plot as numeric. Surveys plot ID. Okay, so when you run this, You can see that plot goes to numeric. And when we look at the dataset, everything looks like it should.

[24:01 - 26:04] Tobin Magle: It's still numbers between 1 and 24. And finally we can go back from numeric to factor using s factor. To numeric factor, we're going to check that plot is numeric like we expected. And that is true. So then we can say [typing] surveys plot as factor surveys plot. Okay. We convert this. Now, we can see that it's a factor of 24 levels. And everything looks pretty much as it should be. Let's do an exercise about type conversion. Year is currently stored as a factor. Try converting the year directly from factor to numeric. Now answer what went wrong? And also, how would you do it so that you get the number of conversions of the label? All right. Let's take a look at exercise two. So it says that the year is currently stored as a factor. What if we want to convert it to a number? So the exercise suggests that we try. We're going to save it in an external variable called year, instead of adding to the data frame, because we know that there's something going to go wrong. So, year is going to be as numeric as surveys year column. So, when we run this, and then we take a look at the year variable that we just created, we don't actually get dates. So if we look at the year column here, the first level should be 1977. And when we look at year, we get six. So, why would that be? Well, this all has to do with the fact that year is stored as numbers with text labels. So, in this case, if we look at the year column here, let's look at actual surveys. So if we look at the year column, the first record is 1982. But it's not actually being stored as 1982.

[26:04 - 28:04] Tobin Magle: It's being stored as a number, with 1982 as label. So what um, what this actually did was it pulled the underlying numbers that it's stored as is, instead of converting the the text labels to a number and then making that the value. Okay. So, that's what went wrong. And how would you do it so that we don't get number, so that we do get the number of conversions of the label instead of the underlying numbers that the factors are stored. So, there's a more computationally efficient way to do this, but I'm going to show you a more intuitive way with character and then to number. So, let's do, I'm just going to copy this. Put it down here so we want and I'll put it into the the data frame since we know we're doing it correctly. All right. So I'm going to take surveys here. And instead of going directly to numeric, I'm going to say I want it to be a character first. Okay, so when I run this, you can see that the year column changed to a character and it's just text 1982, 1982, etc. All right. So then from there, I'm also going to copy this. And from the character, now I'm going to go to a number because R is going to be able to read these character numbers and convert them into regular numbers. So, [typing] as numeric and we'll hit go. Now when we look at year, they're numbers and they're at the actual year numbers that we were looking for, and not the underlying numbers that were labeled in the factor variable. Now, we're going to talk about assessing data quality and factor variables. Factors are integers with text labels. Each unique text label is called level and represents category. The levels function will list all the unique labels in your dataset.

[28:04 - 29:53] Tobin Magle: The N levels function will tell you how many levels are in the data set, and the summary function will output the number of records in each level when applied to factor variables. Let's see how these work. Okay. For this example we're going to use the sex column. Um, so I'm going to move it into a new variable called sex. So, we don't mess up the data frame. And so that, I get to type less. Okay, so we moved the contents of the survey column into a variable called sex. You can see that it's a factor with five levels. And to find out what the levels are. So which unique categories do we have in the sex column? I'm going to use the levels function. And down here, we can see that we have female, male, P, R and Z for whatever those means. And if we had a column with a lot more levels in it, we would probably want to know what the count was without having to count manually. So then we can say n levels for a number of levels. Sex. And when we output it to the screen, you can verify that there are still five levels. Okay. So then if we want to know how many levels are in each category, we can use the summary function. So summary. Six. And we've used this before on data frames as a whole. If you did this on the surveys data frame, you would get the same information that we're going to get for the sex column, but then with information for all the other columns. So when we run this, the way that summary works with factor variables is that it has a section for each level here. And then just how many how many of the records fit into each level.

[29:53 - 31:55] Tobin Magle: So you can see that the bulk of them we have under male and female. These are kind of not really high, high prevalence. So maybe they're just an error in data entry. And then we have about 2,500 that where they couldn't record the sex of the animal when they observed it. So those are some useful functions for inspecting factor variables. Let's try an exercise using levels. So, using levels, find out how many years are represented in the census. And then Number 2, which years have the most and least observations? Size 3. The first step is using levels. Find out how many years are represented in the census. And to do this I'm going to overwrite this variable called year and take the data from the data frame. So surveys, sorry. Year. Helps if you have the right columns. Yeah. So when I changed it here, you can see that the year variable now has numbers 1982, et cetera. So, to do this exercise using levels, we first have to convert it to a factor. So I'm going to say [typing] year is as factor year. And now we can see that year is a factor. And the text labels are the same years that we were talking about before. Okay. So, now we can say which years have the most and least observations. Oh, sorry, we missed the first part. So using levels on how many years are represented in the census. And for that we can use the n levels function. And we can see down here that there are 26 years represented in the study. Okay. So down to Step 2 which years have the most and least observations? Well, we showed you earlier that the summary function kind of makes a count of how many things are in each category.

[31:56 - 33:52] Tobin Magle: So we can say summary. Year. And there's a table down here. Oops. Okay. So if we kind of scroll through here, we can see that 1997 had 2,400. And that seems to be the biggest number. So, on the top we have years and on the bottom we have the number of um observations in that year. So it looks like 1997 is the winner for this. As seen in the sex example, sometimes the format we choose to collect our data and is not how we end up wanting to group for analysis. We can use recode, recoding or changing the text label of a level to correct categories and fixed data entry errors. The recode function is defined in the dplyr package. It takes a factor in a list of assignment statements that specify what changes you want made to the factor levels, and it outputs a new factor with the specified levels. Let's see how this works. All right, so let's learn how to use recode to recode the sex factor that we looked at before. To start out, we're going to do a summary function on sex just to make sure that we remember what we're looking at. So, when I send this to the console, we can see that the sex factor variable that we pulled out of the data frame. It has 1, 2, 3, 4, 5, 6 different levels, or actually, I guess it's five levels and then some NA's. Either way we're not really, like it's pretty obvious what 'F' and 'M' are. It's female male, but we don't really have any information about what 'P', 'R' and 'Z' mean. So, maybe for a certain analysis, we would want to remove this category from the data altogether. Or we could lump these together into a category called 'other', just to retain the data, but reduce the number of levels that we're working with.

[33:52 - 35:51] Tobin Magle: So, if we want to use the recode function because it's in the dplyr package, we need to load the dplyr package. So, we're going to say [typing] library dplyr, and if we run this, it loads the package. If you don't already have dplyr installed, you'll probably get an error message here. And in that case, you can go to the tools and install a package. If you want to learn more about that, you can see the actual dplyr lesson in this series, which is the third one in the series. Okay, so now that the the dplyr package is loaded, we can go into the recode function. So, we said that the output is going to be the new factors. We, we want to we want to change the original factor that we're working with. So we'll just have the assignment operator. So the output of the recode function is going to go back into that variable, which is what we were intending. And the first piece of input we want is the factor variable that we want to change, in this case sex. And then we're going to say we want okay, all the things that are labeled 'p', we actually want them to be labeled 'other'. And all the things labeled 'r', we also want them to be labeled 'other'. And then finally the things labeled 'z', we also want to be labeled 'other'. Remember to put the quotes around these. Here. To have this back up. Okay. So, when we run this, and then we do the summary function again. We can see that we only have, we've reduced the number of factors. So 'P', 'R' and 'Z' are gone. Instead we have five records that are labeled 'other'. So, we combine these three levels into one level here.

[35:52 - 38:05] Tobin Magle: Okay. And that is how you use the recode function to relabel factor variables. Let's stop and do another quick exercise. So, two of the scientific names have a strange symbol, instead of a space. And I'm not even going to try to name these, but basically there should be a space between the name and species in both cases. And this exercise wants you to use the recode function to fix this error. Take a look at Exercise 4. It says that the error that we're looking for is in the scientific name column. So we're going to be messing with surveys, scientific name and then the recode function. So that's basically what the exercise is supposed to do. So, because we want to essentially change things that are in the same column, we're also going to use surveys scientific name is the first piece of input. And then, I'm instead of risking a spelling error, I'm going to grab this copy. And then we want that first one to be the same thing, except for we want to delete that weird backslash 'xe6' thing and put a space in there. And then, same thing here with the other species. [pauses] Just like we did with the sex example up of. Looks like we might. Nope. I guess we're good. So actually, first, before we do this, let's look at [pauses] the levels of survey scientific name. So these are all the different species that are represented in the dataset. You can see that we have these two species mentioned here. So there's proof positive. And you can also see that it looks like we have a little bit of duplication. And some of it's because we've got extra spaces before here. But we're going to deal with the white space, in a second.

[38:06 - 40:29] Tobin Magle: Okay. So for recoding, we've looked at the levels and we wrote out this, this recode statement. So when I run recode, and then take another look at the levels, you can see that we fixed this, fixed the spelling error. So, the weird symbol in between the genus name and the species signifier is gone, and we have a space in there instead. And that's what they've asked us to do in this exercise. After looking at the levels in the scientific name column, we noticed that this dataset has a common data entry error, adding extra whitespace at the ends of the text. We identified this error using factor levels, and we can use the trim whitespace function to remove whitespace. Trim whitespace takes a character or a factor and returns a character, so be careful to convert your column back into a factor if necessary. Let's see how this works. A little another look at the levels of scientific name. So, to review we're going to use the levels function [typing] surveys and then the scientific name column. And we get this, this list where we can see some whitespace ahead of some of these names and some names that look suspiciously familiar. And just so we can have a more defined metric to it, let's use the n levels function and say [typing] surveys scientific name. And right now we have a total of 27 levels. Here I'm going to move this, so it matches my notes. Okay. So, because we kind of have identified that some of our entries have some whitespace that we need to remove. Let's use the trimws function. So we want surveys scientific name. And then we want to trim surveys scientific name. And then I run that. So in theory, we should have removed the white space by now.

[40:33 - 42:44] Tobin Magle: And let's check to see how many levels we have here. We run it. We get zero. So why do we get zero here? Well, let's look at surveys at the scientific name column. We can say, oh, it's not a factor anymore, it's character. So if you remember on the previous slide, we said that trim whitespace will take a factor or a character, but it always returns a, he returns character not factor. So we simply have to say [typing] surveys scientific name as factor surveys scientific name, and then run it. Now, if we look at scientific name, it's back to being a factors. So we can take this n levels command and put it here and run it. And we can see we have 26 levels where we once had 28. And that drives with the fact that we have two visible things that have wide space of the beginning. Okay. We'll move this down to so it matches the notes. But that's how you can use, uh, trim whitespace to remove whitespace at the end and the levels functions to make sure that you have actually done what you think you're going to be doing. Another common data cleaning error is misspellings. The stringdist package has functions to help identify and fix spelling errors. The most basic function in this package is stringdist, which looks at a pair of strings and quantifies the number of differences between the two. So before we can do anything with stringdist, we have to load the library. When I run this, it gives me a warning that says that it was built under a different version of R, which should turn out okay. But, before we start applying this to our dataset, let's just do a couple of simple stringdist commands with some simple strings to kind of get a sense of how it works.

[42:44 - 44:49] Tobin Magle: So, let's say stringdist and let's say we want to compare the string a, b, c to a, b, c. So, we're comparing the same string to itself. And we forgot a quote mark here. So, if I run this, you can see that the distance is zero because there's no difference between abc and abc. Okay. Let's try basically the same thing, but we'll we'll make an error. So instead of abc and the second one we're going to call it abd. And we run this. You can see that the string distance is one because there's one difference between these two strings. All right. So let's do this. So instead of going abc, we're going to say a cba. And we see here this the the distance is two because b is still in the same place in the middle. And then finally, if we compare two strings that basically don't have anything to do with each other, we get a string distance of three, because all three positions in the string are different. Okay. So that's a general overview of how the string dysfunction works. And we're going to move on a bit later into applying this function to the dataset. Now that we understand a bit about how stringdist works, let's apply it to our data set. So, to apply this to our dataset, let's for simplicity's stake at first, just make a new vector called species names. And we're going to populate that with the data from [typing] surveys scientific name. Okay. So when we run that, we get a new variable. Called sp names, and it has all the text of the names from the species, the scientific names column. Okay, so now let's apply the stringdist function to sp names.

[44:52 - 46:53] Tobin Magle: And so, it's going to basically a stringdist, instead of comparing 1 to 1, it can be vectorized. So you can put an entire vector of strings in here. And then, we're going to compare it just to one of the scientific names, in this case, Amphispiza bilineata. You have no idea how many times I've had to practice that. All right. So now we're going to put that in here and when we run it, you can see that there's varying distances of these, this string. So we know that, if you look in the survey's dataset, or sorry. Let's look at the full dataset. If we look in scientific name and Amphispiza bilineata, the first, or the first two entries, the distance here is zero. So that makes sense. And then it gets larger for a while. Then it looks like we've got a whole spate of these, next to each other. But we do have some differences where there's like ones and twos changes. And that would indicate to me that there's a spelling error instead of a completely different species. And because this is outputting the screen, it won't actually output all 30,000 some records. It's going to stop at some point. So, that's a good general check to, to see that stringdist is doing what we expect it to do. [pauses] So, yeah. Let's look at the levels of species names real quick. So levels. We've already done this a couple times pointed it out. But just to reiterate. We can see, especially if we look at Amphispiza bilineata, we can see that we've got it spelled with an 'e' here instead of an 'i', and then we have bilineatus instead of bilineata.

[46:53 - 48:50] Tobin Magle: Same with Ammospermophilis harrisi. We've got it with one 'i' or two 'i's, so we can tell that we definitely need some spelling correction here. And we're going to go through the steps. First we need a list of the correct spellings. You can find the code to specify the correct spellings in the README file that's accompanying this lesson. README. And I have happened to scroll down to the section, where it says the correct spellings of the species names. So I can copy this. Put it into our R script. Run this code. And now we have a new variable called codes. And it'll become clear in a moment while we call this because it's the vernacular for the function we're going to use next, and it has 20 different spellings, which would indicate to me that since we have, you know, 26 levels, that there's six extra levels that were created in the data set, due to misspellings. Now, we can use a more advanced function, amatch which matches strings to a list of accepted values. amatch accepts a sequence of strings to be cleaned, in this case sp names and a sequence of acceptable values which are called codes. amatch's output is a sequence of numbers that corresponds to the position of the matching value in the list of acceptable values or codes. If the algorithm can't find a match, the output is 'NA'. Let's try amatch in our dataset. Is we need to call the list of codes something or the list of. Yeah. Positions in the code something. So we're going to call it 'i' for index. It'll also be easier to type later in future code. Now, we're gonna use the amatch function. And we're going to say that 'x' is the species names, because that's the column that we want to do the analysis on.

[48:50 - 50:55] Tobin Magle: And then our table is going to be codes. So, amatch is going to go through all of the species names and try to figure out which value in the table of codes matches the closest. And then, it's going to give it a number that tells you where where in codes this letter is found or the closest match is found. So we're going to run this. And now we can see that we have a new variable called 'i'. That's a list of numbers. So, you can see we start out with 3,3,1,1, and then a bunch of NAs, which may or may not turn out to be good later. But as you can see just from this little preview, we're going to need to do some quality control here. As we alluded to in the previous part or the previous demo, the default settings for amatch don't work with every data set, so we need to do some quality control. First, let's make a data frame that puts the original text next to the code assigned by a match. Then we'll see how many of them were signed NA. The goal is to get all of the species assigned to something in codes. Create a data frame that compares the raw text to the assigned code. So, let's see [typing] sp names df, that's a little redundant because, you know, it's going to be a data frame and we want it to be data.frame. So, this is a function that creates a data frame. And you you essentially just put in columns. So, the first column that we have is we're going to call rawtext. And that's just going to be sp names. Because you want to know what it was before a guest, essentially. And then we want the code to be codes sub i. So, with this line of code is going to do is for every for every row in i, it's going to go into the codes data frame and see where the codes are vector, and see which species name is in that position, in the sequence of names.

[50:56 - 52:44] Tobin Magle: It's a little confusing, but hopefully once we run the code we'll you'll see what's happening. So if we run this and then we have a new data frame called species names data frame, and we open it up. You can see here that when the raw text was Amphispiza bilineata, we got matched to the correct spelling. And same with Ammodramus savannarum. But, for the harrisi ones and a lot of the other data, we didn't get any assignments. So, it works for some and not others. And we can fix this by changing the clustering method in the amatch function, which we're going to talk about. The amatch function can use different clustering algorithms to match missing data to predefined codes. To change the clustering algorithm that amatch uses, use the method argument. The default method doesn't always work, it depends on the dataset. For example, some of these algorithms are better for fixing typographical errors, and others are better at fixing phonetic misspellings. For our purposes today, we're going to leave the methods used as black box. If you want more information, see the documentation for the stringdist metrics. All right, so let's try altering the method arguments. Use a different method to cluster these names. Through trial and error, I happen to know that the cosine method works well here. Let's apply it. To change the clustering method, we have to back up here. That's where we have the amatch code. And I'm going to add an argument here. I'm going to say method equals, and if you want to know which, what your choices for method are you can go into amatch documentation.

[52:45 - 54:48] Tobin Magle: And where is method? Method is here. And then we can click on stringdist metrics which is related documentation because you can use these same clustering methods in the stringdist function. So we know, I know that cosine works. I just want to make sure we're going to spell it right. It's cosine. And when you're on this object cosine not found haha. You got to put quotes around it. All right. So now we've changed the, the 'i' vector. You can see there's not quite so many NAs here. And now we can create this comparison dataframe again. And when we look at it, you can see that the vast majorities are filled out. Possibly all of them. Not really sure how many NAs are in here. But we can figure this out, by doing some work, you see. Now, we already know that there's less than assigned values in the word the default clustering method. However, it takes too long to go through the entire data set by hand. Let's automate some of the QC using the is.na function. is.na returns a true false list of whether or not the values is at the position as NA, and because true equals one and false equals zero, we can find out the sum and then that'll tell us how many NAs are there. We can also use the double equal sign operator to see if the number of NAs in the recorded data matches the number of NAs in the original dataset. Step is we're going to see if there are actually any unassigned. So, let's start slow. We can do is.na sp_names_df. And then we're going to look at the code column. And when we run this, we see we get a list of true and false.

[54:48 - 57:23] Tobin Magle: A lot of them are not unassigned because it's false. And if they were unassigned, it would be true. But that's a little bit messy. We don't want to go through the whole data set again so we can say some. And yes, we have about 15,000 that are unassigned. But that is not necessarily bad thing because the original data set did have some that were unassigned. So, to figure that out. You can copy this. And then we want to compare that list to some is.na, excuse me surveys scientific name. So essentially we're, we're comparing whether or not they were in NA in the recoded data set versus the original dataset. And when we run this, it's true. So these things are equal. So that means that they both have the same number of NAs. So that's a pretty good indication that we didn't miss any of them that needed to be recoded. So now that we, we know that our data were recorded correctly, we can put the, the vector that we've been working with back into the surveys data set. So if we do surveys scientific name helps you spell surveys correctly, and we want that to be sp names datadrame. And we want to put it back in the code column. So now when we do the number of levels, let's do regular. First, we'll do levels of surveys scientific name. So you can see here that we don't really see as much duplication as we did before. It looks like all the ones that are probably the same thing are grouped together. And if we put n levels in here, we get back 20, which is the same length as the correct spellings here of species names. That we have up here in codes. Okay. And that is how you fix spelling errors in a dataset using clustering algorithms.

[57:23 - 59:33] Tobin Magle: The final common data problem is having multiple variables in one column. We can split columns into multiple columns based on a delimiter using the separate function. Separate is found in the tidyr package. We'll talk more about tidyr in the data wrangling session. As input, separate takes a dataframe, the name of the column you want to split, the separator you want to split by, and the name of the new columns you want to split into. Its output into a new data frame with the columns separated. Let's see how this works. All right. So the first thing that we need to do to use separate is load the tidyr library. So I'm going to do library. T-I-D-Y-R and load that. All right. Now we're free to use a separate function. So I'm going to put the output back into surveys. And then separate. And the first argument is data. And for that we're going to start with the surveys data frame. And then, we pick the column that we want to split and put it in the call argument. And in this case, we want to split scientific name. And then, we pick the separator. And this is what we want to separate by. So in this case, the separator that we want should be the, the space because that's what's in between the genus and species names. And now, we have the argument 'into'. And because we know it's going to split into two columns, we can do genius, or not genius, genus and species. Make sure these are in quotes. And then the the final thing parameter that we're going to use is 'remove'. And this tells whether or not to remove the first column. In this case it's going to be false because we want to be able to compare the output columns to the input column, at least initially.

[59:34 - 01:00:43] Tobin Magle: We could redo this later or remove this later using select from the dplyr package or we could rerun this code again and tell it that we do want remove. So remove would be equal true if we know that it works and we just don't want the column there anymore. All right. So when I run this, you can look at the surveys data frame and we've got scientific name here. And right after it, we've got genus and species and you could run through this and, and verify that it's split correctly on all of these. But the fact that it didn't make more columns or give us an error is a good indication that it split everything into two columns. And that's how you use 'separate'. Thanks for listening. I hope you found this session to be helpful. Please email me at the address on the slide if you need help. Also, check out our data management pages for more information about the services we provide at the Morgan Library. Also, the source material I use for this lesson is linked to on the side, namely the Data Carpentry, R Ecology Lesson, OpenRefine Lesson, and also a Data cleaning reference from the R project.

END TRANSCRIPTION