

# Data Management and Preparation Using R

by Martin Burger

Start Course

Bookmark

Add to Channel

Download Course

Table of contents

Description

**Transcript**

Exercise files

Discussion

Learnin

## Course Overview

### Overview

Hi guys, this is Martin Burger, and I welcome you to my course, Data Management and Preparation Using R. As a data scientist and biostatistician, I know first-hand how important clean and well formatted data is. R is widely used for data analytics. It offers great data management and data preparation tools. In this course, you will learn techniques to solve the most common problems of the data preparation steps. This is basically the first step in the whole data analytics process, which means, you cannot escape this. No matter your industry or analytics approach, you have to prepare your data first. In the course, you will see how to use different import tools to get standard as well as exotic file formats into R. You will learn which object classes are best suited towards your data sets. You will use the tidyverse add on package to clean and format your data, and you will use the data.table package, as well as standard tools in order to filter or query even large data sets. By the end of this course, you will be able to select suitable add on packages and use the best functions for data preparation. I would categorize this course as a beginners plus course. If you're familiar with basic R code, you will be able to fully benefit from this course. Alright guys, I hope you will enjoy this course, I'll see you inside.

# Introduction

## Introduction

Hi guys, this is Martin Burger for Pluralsight. In this module, I will show you where this course is located on the data science landscape. We briefly talk about the main terminology behind this course. I will tell you the packages you will need to follow along, and how to best prepare yourself to get the most out of the course. And I will tell you how the course is structured, and what you can and cannot expect from this course. Basically, this module will give you all the info needed to fully benefit from this course.

## The Data Science Landscape

In this video, I highlight some definitions so that we all talk about the same things. I will also show you where we are on the data science landscape, so that you can orient yourself. First of all, we need to define data management and data preparation. There is no specific definition to these terms, everyone has a slightly different opinion about their meaning. Generally, data management is a much broader term. If you check out data management on Wikipedia, you will learn that there is even a society called DAMA, for data management. For this course, the term data preparation, which is synonymous to data pre-processing, is much more suitable. It is the process of getting the data in such a shape so that further analysis or visualization is possible. It is basically all the things you do between data collection, and data analysis. Let's now take a look at the data management and data preparation landscape. That way we can better understand where we are in the data science world. The whole process starts with collecting data. There are various ways in how this happens. Some are more sophisticated than others. Some data is collected via simple manual data entry, sometimes the people providing data do not even know that they are an active participant in the generation of a data set. If you're visiting a website via a manual click, you just created an entry in a database. Quite likely in several databases actually. The ISP, your internet service provider, will likely keep logs, as well as the site host. This data can now be used for all sorts of analysis. This is also a major reason why data scientists are very keen on data protection. They simply know how and when data is generated, and what it can be used for. Other data is gathered semi automatically via web scraping specific content. Data can even be simulated fully automatically, and of course in science and technology fields, sensors keep logs of measurements, which need to be analyzed. As soon as you have the data collected, the data preparation part starts. This is exactly what this course focuses on. You cannot

escape this step, it is too important. Unfortunately, this topic is widely overlooked, with information being hard to find. Let's take a look at this preparation of pre-processing steps.

Number one, the data import. This might sound trivial, but if you consider all the different data formats out there, you can imagine that this can be confusing. But this is a logical result of the various ways in how data is generated. Different tools generate different formats with different weak spots. People use different programs, partly because of personal preference, partly to save costs on proprietary software. Fortunately, R is able to import and treat almost any data format, however, exotic formats are generally slightly more difficult to import than a standard CSV file. I personally advocate for using CSV whenever possible. If this would be implemented as SOP in corporations, a lot of headache would be spared.

Number two, selecting the object class. Once you have the data set imported, you need to select a proper class. That way you are essentially formatting the data for R. Each class has different traits, which means you can and cannot do certain things with your data in a certain class. A standard data frame might be fine for standard tasks, but there are more advanced classes out there, like the data table. Especially with those huge data sets, nowadays a data frame might not do it anymore.

Number three, getting your data in a tidy form. You need to clean your data. This is in fact a very faceted area. Many things can go wrong. We all know Murphy's Law, anything that can go wrong will eventually go wrong. This is huge implications for you as data scientist. Columns might contain several variables at once, some cells might have different formats, the whole data set needs to be transposed. Some outliers are not plausible, and so on. Generally, a tidy data set has one row for each observation, and one column for each variable. This might sound trivial, but, in your daily work you will find instances where this simpler rule is not followed. Often times you will not even notice that the data set is not tidy in its layout, and is faulty in various ways. We will learn how `tidyr` can help you in getting your data into a clean and tidy format. This part of data pre-processing also contains the subdiscipline's missing data imputation, and outlier detection. These are topics which require more statistical theory in order to cover them deep enough. This is outside the scope of this course, which is a more programming focused one.

Alright, number four, querying and filtering. When you have a huge data set you will likely want to take a look at a certain subset of that data set. You need to filter with the desired parameters. This filtering process is also referred to as querying. Especially `data.table` has proven effective for querying huge data sets, therefore, we will focus on this package in the querying section of this course. After the preparation part, you can start performing various types of analysis. Interestingly, the analytics part is much hyped. Everyone speaks about machine learning, dashboard visualizations, and forecasts, but without clean and proper data, you're quite restricted. Data preparation is overlooked, but it makes the analysis part much easier, or even enables it. For example, in machine learning it might be

required to perform wide too long format conversions since the algorithms need a long data format. Data visualizations, or even dashboards, might require data without missing values. Or it might happen that you only want to work with a subset of data, hence, you need to run a query first. No matter what, it is important to get the preparation part of the whole data science life cycle right, and that is exactly what we focus on in this course.

## Prerequisites and Your Preparation

You might ask yourself if there are any prerequisites for this course, or how to best prepare to get the most out of it. Well, there are two sides to this, the technical tools, and knowledge you should have. As far as the technical side goes, to follow along you need R and RStudio installed. I will run all the demos in our studio, which is by far the most commonly used graphical user interface for R. And this is fully narrated by the way. We use some add-on packages, which I would recommend to install right away, so that you can follow along. `install.packages` is the install function for packages in R. Remember, install once, activate each section you want to use them. So when you see me running the `library` command, you know that this package needs to be activated, since it contains a certain function, or a data set. For the query section, and the data table class in general, we are using the package `data.table`. It's an excellent alternative to `data.frames`, as you will learn. For cleaning and tidying data, we use Hadley Wickham's `tidyr` package, which contains functions for the most common cleaning problems you will encounter. I will also use the `Reshape2` package, which has alternatives to the `tidyr` package. For joins, we use Hadley Wickham's `dplyr` package. This one even contains another alternative to `data.frame`, and some other very useful data management functions. For the diamonds data set in the query section, you will need the very famous data visualization package, `ggplot2`. For the import of more exotic file formats, we are going to use the `foreign` package. Coach and extra data sets are provided as downloadable resources with this course. Now for preparation on the knowledge side, I would recommend Casimir Saternos course, `RStudio: Get Started`, over here on Pluralsight. This will get you up and running with R and RStudio, if you're not yet familiar with the very basics, and if you're totally new to programming. I would categorize this course as a beginners plus course. You just need the very basics of R in order to follow along. If you are good in other languages like Python or Matlab, you should be able to follow anyways. I built this course for all people using R, no matter which type of analysis or visualization you want to create, data preparation and data management is a fundamental part of your work. Therefore, you will certainly benefit from this course.

## Course Expectations

In this video, I will explain what you can and cannot expect from completing this course. And I will relate this to the corresponding course modules. As you already know, this first module is an orientation module. It shows you where we are in the data science world. In general, this course has the structure of the data preparation workflow is outlined in this module. Module 2 will show you how to import standard and exotic formats, and how to select a suitable class. Mainly, we work with `data.frame`, and `data.table`, which have shown to be the two most reliable and versatile classes in R at the moment. For the import demo, I tried to show functions which are as versatile as possible, so that you can easily adjust them. It is far outside the scope of this course to demonstrate the import of each individual file format, like Stata, SPSS, SS, Matlab, JSON, and so on. Luckily, there are functions which are adjusted fairly easily to various formats. I will also demonstrate how to do Copy/Paste imports into R, which is not as trivial as it might sound. In module 3, you can expect to learn how a clean dataset should look like. We use `tidyr`, as well as `reshape2`, to produce tidy and clean datasets, especially why long format conversions, cell splits, and joins will be demonstrated. Of course, there are a million ways in how a dataset is messy, but the problems outlined in this module are the ones you will encounter over and over again. On top of that, I will show you how to perform joins to fuse two data sets together. Those of you familiar to SQL might already know joins. In module 4, we do some queries on a data table, and on standard data frames. Queries allow you to generate a subset out of your dataset based on a specific set of criteria. And the last module will summarize this course and give you guidance on your next steps on how to get more info for further projects. After this course, you should be able to perform the standard data preparation and cleaning steps. And you should also be able to identify the most helpful add-on packages for this sort of task.

## Summary

Let's summarize what we did in this module, and what to expect next. In this module, I told you which steps are part of data preparation. I showed you which packages and programs you will need to follow along and how to best prepare for this course. In the outline, the general course roadmap, and what to expect from this course. In the next module, I will show you how to import different sorts of data, and how to select a suitable class for your datasets.

# Selecting Suitable Classes and Importing Data

## Introduction

Hi guys, this is Martin Burger for Pluralsight. In this module, you will learn how to import your data and how to select a proper class. We'll take a look at standard import methods, which come preinstalled with our studio. We'll take a look at the `fread` function of `data.table`, and we'll learn how to use the package `foreign` for non-CSV file formats. I will also show you alternative classes to `data.frames`. These are `data.table`, and `data.frame`. Learning how to properly import your data is often overlooked. In a scripting language like R, even an otherwise trivial task, like data import, can have some pitfalls. `Data.frames` have some limitations, therefore it is great to have some alternatives at hand, in case you have larger datasets.

## Data Import the Standard Way

Before you can even start cleaning your data, let alone analyze it, you need to get it into your studio first. Like always in R, there are dozens of ways how you can do this. It depends a lot on which type of data you have, and how big the dataset is. Generally, I would recommend to store your data as a CSV, a comma separated values file, and import it as such. That is the way best implemented in our studio, and gets you the fastest results, even for bigger datasets. For small data, or even snippets, you can use the copy and paste way, like I will show you as well. If you have alternative data types like Stata or XLS files, you can use a similar approach than the one I will show you, or you can use the package `foreign`. Let's check out the standard way at first. This is the method best implemented, hence it's easy and fast. As you might have found out, there is an import data icon over here in our studio, which I'm going to press. We have a CSV, so we select that one. Lately our studio also integrated a quick import method for XLS, Stata, SAS, or SPSS, but you will need the package even for that. In this interface, we first need to browse our directory to find the dataset. It is called `Insects`. This method even works without setting a working directory. We have a preview window to see the output. There are a few things we need to be aware of so that the import process works smoothly. Most importantly we have to select the right delimiter, or separator. This is the symbol used to separate the cells. In our case, we have to go for Semicolon, since it's a CSV, with semicolon as separator. This instantly changes the preview, which now makes much more sense. You can also set our NAs, missing data, are

displayed. This is not some sort of advanced imputation logarithm, just some basic cell filling. You can go for empty cells, 0, NAs, and some other methods. Furthermore, over here you can check the header. In this case I'm going to stay with the header since I want the data to be with a proper heading. We can change the file name if we want, or we can change it later on in the console. We simply click ok to get the data imported for our analysis. So now we have a new object in the environment. This one can now be used like a common data. frame, but it also has the classes table, and table\_df. So like I said, this method is fairly easy and works great for CSVs. For small data factors, and other snippets, there is a convenient shortcut function to allow for a copy and paste style insertion. It is the scan function. Let's say we have some numbers in Excel, and we want to copy/paste them into R. Therefore, we got to Excel, copy them over here into our clipboard, and now we can run the scan function in our studio. I am simply getting the new object name, numbers over here, that will contain the data from Excel. If I now execute the scan function, I will have to click in the console, paste the data via Ctrl+V, and simply click Enter. If we now take a look at the new object, we get the data we just pasted. So that is our confirmation that the whole procedure worked. If we do the same thing for string data, like let's say we want to copy this string factor from over here, we do the same thing at the Excel side, but now we need to specify, scan(what = 'character') so that R knows that we want to get strings as an output. Otherwise we would get an error message, since R would expect data of class double. I would say that this sort of thing is a quick and dirty way to get factors and single columns into R. For larger matrices, and data. frame like data, it is not the best way to do it, you would probably have to do this by lists and all sorts of cleaning steps, which is oftentimes not worth the hassle. The scan function is not limited to Excel, each time you want to copy and paste something, this can be used, even on data from your browser.

## Using Fread

Now, the next function I'm going to show you is a secret weapon when it comes to data import. The advantage here is the speed of the function. We are talking about the fread function from the package data. table. Not only is it fast and handles large datasets well, it is also very easy to use. In fact, it appears the default settings are chosen with a lot of experience from the R based ways of importing CSV files. Weak points are eliminated for the most part. The function automatically recognizes the separator of your data. This feature is independent of the file extension. It scans the data and recognizes the separator automatically. Furthermore, strings are not automatically converted to factors, but you can use the strings as factors argument, and set it to true, to get an auto conversion. Furthermore, the headers are displayed automatically if the first line of your data

contains a header. The output you can expect is a data table, with a data frame as backup class. If you want data frame as the sole class of the imported object, you can use the argument `data.table`, and set it to `false`. In order to read in a new file, we'd first set a working directory over here in the Session, and then Set Working Directory, Choose Directory. You set the working directory once in each session. The working directory is the place where R will search for data. It is also the place where it puts datasets you're storing. Let's say for our example we are creating the object `mydata`, which is the data file `Insects`, which is in the working directory. If I now want to import the data, I just have to state `mydata = fread`, and under quotations the full name of the dataset. If we now run it, we can see that the import worked fine. Even the header was recognized to my full satisfaction. So it is a great and easy function for data import. It is even so versatile as to allow copy and paste from the browser, or some other program. I go to Excel and select our dataset. I just copy the table to my clipboard. I then go to RStudio, and I create a new object called `dt`, which is the function `fread`, and now in quotations I paste all the material in. If I run the line you will see that this worked really great and `fread` recognized very well how the table should look like. Now in case the header was not available, or you are not satisfied with the header, we can easily change that. We are going to alter the original code with the same argument, `col.names`. So those are the five size classes, plus the region we have in our dataset. If we run the line again, you will see that we just changed the top of our data, which we'll just copy and paste it from Excel. Again, it's a really versatile function for the import of data into R. By the way, down here I was also checking the class, which confirms that to have data table as the primary class, and data frame as the secondary class.

## Exotic File Formats

In some instances, you might be forced to use a more exotic file format. In this video, I will show you how to use the library `foreign`, to import Minitab, Stata, SPSS, S, Octave, or even SAS files. Make sure to have it on your machine and activate it to follow along. To learn what is inside the package, we will take a look at the Help section. This basically shows us all the functions that are available in that exact package. Nearly all the functions here help us to either load and import external data, or to export and write files like you can see with the last functions down here. Nearly all the import functions of this package are structured in the same way. You just have to state the function, and then you simply state the name of the file, which should be placed in the working directory to get the material imported. This is the same procedure like with the `fread` function. Some of the file types from above can also be imported with the `Hmisc` package. For example, there is a function called `sas.get`, as well as `stata.get`, that will provide similar service



than the foreign package. Let's check out the demo to see how an import with a foreign function works. I prepared a Stata file called `statatest.dta`, which is now in the working directory. This data file has a `dta` ending, therefore I'm using `read.dta`, and in brackets and quotations the name of the file including the ending. So that is `statatest.dta` in this case. Again, this code syntax works for all these read functions over here, so if we run this line, we can now see that we have imported the data successfully. Down here in the console we get the content output, which means we can now work with this dataset, like with a standard `data.frame`. By the way, if you have dates in your data, you could use the `convert.dates` argument, to properly format this. `convert.underscore` will convert your underscores to dots, and `convert.factors` will convert labels to factors in the output. Again, these conversion arguments are the same in all these read functions. Lately our studio integrated a way to easily import SPSS, SAS, Stata, and XLS, via the `import.data` command over here. This method works similarly to the standard CSV method, but you need the `haven` package to integrate that. `Haven` is still in its infancy, whereas `foreign` is a seasoned package which facilitates exotic file import for several years now. If you want you can try it with `import.icon`, and see which method works best for you. At this point, I am again stating that it is preferable to have a CSV file as import files. That means for example if you run SAS or SPSS, try to export your data as a CSV from these source programs. Same goes for Excel files. Nearly all modern data science programs can read and write CSV, which should be the smallest common denominator among all those competing problems.

## Data Frames

If you would ask 100 advanced R users what the single most important tool within R is, most of them will come up with the `data.frame` object class. This is because nearly all of the data you will either create within R, or load into it, will fit the `data.frame` format. Therefore, it is crucial to understand this type of data. You can imagine a `data.frame` like an Excel sheet. You have a row for each observation, and you have a column for each variable. So it's a very simple structure, everyone can see instantly what your data is about when formatted as a `data.frame`. But it is not only the simplicity that is the core strength of this class, it also enables you to do almost all sorts of analysis, because most of the functions in R allow you to use the `data.frame` as they input data. But there are of course also things you need to be aware of when using this tool. Many users are quite surprised when they find out that variables are in a totally different class as they intended. Let's take a look at how we can set up a simple example, `data.frame`. The underlying command here would be `data.frame`, which tells R that we want our data to be packed into a `data.frame`. Let's create our example `data.frame` with three columns, and we are going to call it

mydata. frame. This example is about 16 members of a sports team. They are either performing three or five pushups, and after those pushups we will do a metabolic measurement on them. Therefore, the first column should be the name of our team member. The second should be the count of pushups to be performed, and the third one is a metabolic measurement after the pushups. So the column name has six names in it. They are under quotations, so that means they should be recognized as strings. Column B contains two counts, which are integers in this case. But here, we would only have two such integers, instead of six, which is basically the number of observations arose. So we will see what R does to this shorter column in a minute. And column measurement contains six random normally distributed numbers, which represent a metabolic measurement after the physical exercise. Rnorm is one of R's functions for random number generation. Especially if you're trying out new tools, this one is very helpful. Over at stack overflow you will find many guys using this feature. Let's now run those lines, and let's create our object mydata. frame, which should be a data. frame. Alright, so down here in the console we now get the output, including the column or variable names in the header. We can see that it has six rows, so the second column of the counts was multiplied in the repetitive order. In R, this is called recycling. R basically solved this problem for us here. It automatically filled up the four empty rows in this column. But, be aware of this because sometimes this might not be desired. Or it could lead you to overlook mistakes or missing values in the data. Now here at the Global Environment, you can click at this arrow to get some more info on the characteristics of the object. We see the classes of each of the variables. The measurements are correctly denoted as numeric, but the counts are also numeric, and not integers, which is what we intended. And on top of that, we have factor for the names instead of character. So we can see that R did not yet create the data. frame to our complete satisfaction. We need to fix those two issues. In order to make sure that strings, like in our first column, are properly recognized as character, we have to use the argument stringsAsFactors, and we need to put it to false, which is what I'm doing down here in this row. We have to do this because this argument is activated per default. Some arguments, like this one, can be seen as a light switch. You turn them on by indicating true, or simply T, which is the default here, or you turn them off with an F for false. If we do not add strings as factors to the code, I would assume we are happy with getting strings as a factor class. Compared to strings, factors have several levels and can be ordered according to those levels. But, string manipulations like gsub are best performed on character class data. Alright, so at last we will be converting the counts to integer, therefore, I'm changing the code of the second column. This makes perfect sense here because counts can only be whole numbers, which is an integer format. Therefore, I'm using s. integer to tell R that this one should be an integer instead of a simple numeric column. So now let's run this code. As you can see, the output in the console

looks the same, but up here in the Global Environment, where we get the information about the classes within our object, we can see that we changed the two classes successfully, and we have our data frame formatted in the exact way we want it to be. We have the names as character, we have the counts as integers, we have the measurements as numeric values. Having the right variable classes assigned is important because operations in R require the correct class. Now in most of the cases you will directly import your dataset as a CSV, but even then you have to be aware about the class of each column. And if required, you need to change them accordingly.

## Alternative Formats

Data frames are the most common type of data you will find in R. Of course, this makes perfect sense because most of the data fits into this object class. Columns for the variables and rows for the observations. That is the simple pattern of the data frame, and of nearly all data. Of course, there might be simple factors or time series data, but those make up for only a minority of cases. R offers several alternatives in how to deal with those datasets. We can see the same thing in R graphing where you can choose between R Base, the standard which comes with R itself, then there is the Lattice package, with plots especially suited for scientific publications, and for some years now we also have ggplot2, which becomes the default way of producing quality plots in R. So those are the three main ways in how you can do the same thing in R, producing great graphs. Of course there are alternatives and add-ons, but those are the dominant ones. So how does this relate to data frames, and data management you might ask. Well, we have the same thing here, we have the function `data.frame` from R Base, which you already know, but we also have a whole different system which does the same thing. It is called `data.table` from the package `data.table`. You can store the same type of data in those data tables, and it offers some advantages to the standard way of a data frame. And there is also the `data_frame`, which is part of the `dplyr` package from Hadley Wickham. You can use it to store the same type of data, and if you have the blind spots from data frames are covered as well. So again, we have three quality ways in how we can do the same thing. We can store our data, we can filter, clean, or subset our data in three different ways. So let's compare these classes. We already talked about the data frame, it's part of our base, which means you do not need to load anything to make use of the class. But for a data table, you need the add-on package `data.table`. A data table is superior to a standard data frame in that it requires less programming. The whole class can be manipulated with less code and function calls, which not only saves time but helps to reduce errors. Furthermore, a data table requires less computing time, which also enables you to handle much larger datasets. And there are other nice side effects like the possibility of ordered joins, or the

great documentation of the package. Note that `data.table` has a second class, which is `data.frame`. That basically means if you're working with your `data.table` and you're using a function from the package which does not recognize the `data.table` format, R will just default to `data.frame`, and will treat the `data.table` as a `data.frame` in this case. So you will get all the upsides of the `data.table` where possible, but no downsides since you will be covered with a backup class, `data.frame`. This feature is also available with the `data_frame`. In fact, the whole class, `data.table`, has its own structure, which is very similar to a SQL structure. In module four of this course, we work extensively with `data.tables`. For filtering or querying `data.tables`, a specific syntax is used which is slightly similar to the `data.table` query syntax. Another feature of the `data.table` is the recycling it will automatically perform on certain columns if the data provided is too short in that specific column. Recycling occurs in a `data.frame` as well, but in a `data_frame`, this feature is eliminated. In order to create a `data.table` in R, you would use the `data.table` function. If we take a look at the output of a `data.table` in R, we can see that the row IDs are clearly separated by those dots. Furthermore, if you would have a string or character column, this column would not automatically be transformed by `data.table`. This is quite a nice feature because this can lead to confusion if you have a factor in your data, and you do not really recognize it as such at the beginning. Unlike the `data.frame`, there are no row names in a `data.table`, just the row IDs at the beginning. By the way, if you want to print a `data.table` and the amount of rows exceeds your machine capacity, only the first and last five rows are printed. For a `data_frame`, with a function `data_frame`, you need columns of equal length. If they are not equally long, you will get an error message telling you which column needs to be fixed. The only exception here would be a column of length one, and this would be recycled. Like with a `data.table`, if you view a large dataset you will display the first few rows, not the whole set, like in the `data.frame`. So the output of a `data_frame` looks like this. Over here in the header, we have information on the column class. We had to provide the column `c`, which was of equal length to column `a` and `b`, and we also got character for `a` and `b` instead of factor. If we compare those three classes we can see that for extended data management tasks, it is beneficial to use an advanced tool like `data.table`. Since `data.tables`, as well as `data_frames` have the stand that `data.frame` is backup, you will not get stuck in case the function you are using does not recognize this class. So keep those classes in mind next time you are dealing with extended datasets and data management tasks.

## Summary

Alright, so let's recap what we did in this module. We were talking about the data import, and that is the first step of any data analytics process. The first thing we took a look at was the import

dataset feature, which comes directly with our studio. This method is fully click supported, which means it is perfectly suitable for the R beginners. We took a brief look at the scan function for a copy/paste approach for small factors and snippets, and then we discussed the fread function. This one is fast and efficient. As you saw, the default settings are cleverly chosen, which means nearly no extra coach. With fread, we had to take care of the working directory, and of course, since this function is in an external package, you have to get the data.table package at first. And we discussed alternative ways in how to get more exotic file formats into your R instance, with the foreign package. We also talked about different classes, namely data.frame, data.table, and data\_frame, which work as a data container in R. These classes allow you to run all sorts of R functions on your data. In the next module, you will now learn how to use tidyr and Reshape2, in order to clean and shape your data.

# Cleaning Data with tidyr

## Introduction

Hi guys, this is Martin Burger for Pluralsight. In this module, we'll discuss the specifics of properly formatted data. You will first learn how a tidy dataset looks, then you will learn how to fix common data formatting problems via four demos. For that, we're going to use three R packages, and some ad-hoc datasets. In this module, you'll also learn about filtering and mutating joins to connect to data frames.

## What Is Tidy Data?

In your daily work, you will receive all sorts of datasets from your colleagues. Unfortunately, these datasets rarely comply with formatting standards. Be patient with them since data management skills are almost never taught properly. Therefore, the first thing you have to do is to check the dataset format, and clean/rearrange it if required. Depending on the data set, this can be done in one step, or it requires multiple steps. Let's take a look at the specifics of an optimal tidy dataset. You want each variable to have its own column, you want each observation to have one row, and you want each experiment, your observational unit form one table or a data frame. Now, in your daily work, you will likely encounter each possible iteration of messy and chaotic data. In this module, we'll look at the most common ones. What do you do if several column names are subcategories of one variable? This is called a wide to long format conversion. What do you do

when multiple variables are stored in one column? And what do you do when one variable is stored in two rows? The solution for that is called a long to wide format conversion. And what do you do when an experiment or observational unit is either split into different tables, data frames, or one table contains data on several observational units? So we are also discussing filtering and mutating joins here, which some of you might already know from SQL. For each of these cases, we check out a demo. Quite often you will encounter problems requiring a total reorganization of a data frame. In these instances, you'll want to take a look at the tidyr package. It contains excellent functions to switch between wide and long data frame formats, and it even contains function to split a column into separate ones. Like most of the packaged of Hadley Wickham, this package is well organized, and has a special logic to it, which requires some time to get used to. But as soon as you're familiar with it, you do not want to miss this tool ever again. In fact, tidyr is not just a package, it offers a reference standard for data frames, which is quite easy to work with in R. There are add-on packages required for this module. You need to install tidyr at first if you do not yet have it on your computer. You also need dplyr for the different joins. And they also use reshape2, which has alternatives available for many tidyr functions.

## Wide to Long Conversion

In this video, you will learn how to convert a wide dataset into a long one. Let's see how a dataset in a wide format looks to better understand what we want to accomplish. So as you can see here, this dataset has these weight classes of 10g each, as column name. The problem here is that these weight classes are subcategories of one variable. That means, it would be wise to combine all these subcategories into one variable, which looks like this. Here we only have three columns, the region name, the weight class, and the count for that class. In the demo, we first import the data, we'll take a look at it, identify and fix any issues of the import process. Then we'll use the gather function for the conversion process. And after that we'll put it into a table df format. I will show you how to rearrange the table, and at last you will also see the alternative function of reshape2, which is called melt. Alright so we activate the tidyr package, and all the packages we need in this module. And I am also getting the dataset insect, which should be available as downloadable material with this course. I'm going to import CSV. When importing, also make sure to activate the header over here. We need this for further work. If you do not activate the header, R might not recognize our columns. And quite importantly, we are going to specify the object name, insect, in lowercases right here, which will make coding easier. Long names with complicated symbols, including file endings, are a prime source for faulty code, therefore make sure to name it elegantly right away. Also make sure to get the correct delimiter, which in this case should be a Semicolon.

So now the data looks like this. It is in wide form. We have the header, which probably needs some fixes. We have the region, where the counts were taken of the insect, and the counts are split into columns according to weight and gram. There are five specimens in Welsh Creek, with 10 grams or less, and so on. This wide form is actually quite common, and you probably have seen it already. The problem however is that the weight classes are subcategories of one variable. Basically, we need to combine those five columns, and put it in only one column. And we are going to call this column weight, since it denotes the weight class in grams. We can probably improve the header a little bit. This is important, since those are the values which will later on be put in a new column. There are several ways in how you can do this character manipulation with, `gsub` comes to mind, but we will take the quick and dirty method here, and just rename the header manually. Therefore, we are overwriting the names of the `insec` object. The first column contains the region where the specimens were collected, and then we have the five weight classes, 10 grams, 20, 30, 40, and so on. If we now take a look at the object, we will see that the header looks much better. Now we can start the conversion to long form. Again, the goal is to gather the five weight classes in one column. And we are going to call this column weight. This one will be our key, and the values will be the counts. For this endeavor, we use the `gather` function from the `tidyr` library. If we check out the function, we can see that we need the data, `insec`, in this case. That is quite straightforward. Then we need a key, which will be the name of the new column consisting of the weight classes. And we'll need the value, which will be counts. We do not want the region column to be part of this, so we are excluding region with `-Region`, as you can see over here. By the way, this is the argument order of this function. If you put the arguments in this form, you do not need to state the argument name. I think it is quite useful for educational purposes, but you are free to use it as you wish. At the beginning, I think it makes sense to state argument names, like this. If we run the code, we get the changed data frame. This is the so called long form. All the weight classes are now put into this second column, and the corresponding counts are over here in the third column. So far so good. That shows you how useful and simple this `gather` function is. By the way, if we accidentally omit the `-Region`, we would get a rather useless result. The whole region `Id` would be missing, and we would get the regions over here at the beginning instead. So always make sure that you exclude the required columns with a minus sign, and the column name. We can now store the data in a convenient table `df` format, and we'll call it `insec.table`. With the `arrange` function we can then change the order within the table. That means we can sort the whole table alphabetically, in accordance to the region names. Now for this conversion process, there is an alternative available called `melt`. It is in both the packages `reshape2`, and `data.table`. For the demo, we are creating the object `insec.melt`, which should look exactly like the one we just created. The function is `melt`, we specify the

data, and now we provide the column ids of the columns we want to collapse. Here it is from 2 to 6, as we can see on the view. We are going to state the variable name, which is weight, and the values are the counts. And that's it. The syntax is quite similar to the gather function, although the arguments have a different name. Per default, the NAs are going to be removed, in case you want to keep them, you can set `na.rm` to true, which is a valid statement for most of the functions we learned in this section. If we check out the result, we can see that it is the same as with the gather function. By the way, this malfunction is available in various forms also for lists or arrays.

## Splitting Cells

Next, we are going to learn what you can do if you have mixed information in each cell of a column. When data is initially recorded, people often try to be as efficient as possible. That means variables will be combined and mixed, which makes it nearly impossible to analyze. Variables need to be split first. If we check out the data frame `doubleinfo`, we can see that the column, `biometrics`, contains a number and a letter in combination. In this case this means the body height in centimeters, and the gender. In this demo, you will learn how to separate such a messy column. Note that for this procedure to work, there needs to be a specific pattern in the data. Luckily `tidyr` has a function called `separate`, which is suitable for this kind of problem. Note that this function is primarily useful if you have values of equal length, like we do over here. There needs to be at least some sort of pattern. Here it is easy since we need to split between position 3 and 4 from the left side, or after the first position starting right-handed. If we take a look at the function, we can see how it is structured. We need to state the data, which is `doubleinfo`, `col` is the column to be split up, and `into` are the two columns to be formed instead of `col`. In this case I'm going to call it `height` and `sex` in this order, since the first three values are height and centimeters, and the last one is sex. And at last the state and number indicating where to split. In this case it is 3. So after the third position from the left, or alternatively, minus one, since it is the first position from the right. Either way would work in this case. It is good to know this in case your data has only a fixed pattern from one side. We execute the `separate` code, and we see a successful split of the `biometrics` column, as is shown in the console. The combined `biometrics` column is now gone, and we have the `height` and the `sex` columns instead. This is clearly an improvement since height can be separately handled as an integer. Again, if you have a combined column with a clear pattern from at least one side, you can use `tidyr` and `separate` for an easy separation.

## Long to Wide Conversion



We already learned how to use `tidyr` to convert from wide to long format. In this video, we use the function `spread` for the opposite, the long to wide format conversion. So this data frame has now a long form, we have the names, we have performance, and counts related to the performance. This could be for example when you have a physical experiment before and after an endurance exercise. Therefore, top and low performance. Both these counts are part of the same observation, therefore, one row is to be preferred. After all, we learned about the rule of one row per observation. It means we want to divide the performance column into two columns - one for the top performance counts, and one for the low performance counts. That is a classic long to wide conversion. So how do we solve this problem? Well, `tidyr` has a function available for that as well, it is called `spread`. And I will also show you the reshape to alternative, which is called the `dcast`. Let's start with a `spread` function. We need to set the data, then we have the key, which in this case is the performance column. That is the column we want to extract the names for the two new columns from. And then we also have the counts, which is the value argument we are going to put into long form. These counts are the number of repetitions of a specific exercise. Top and low counts of each person will now be put into one row instead of two rows. Alright, so as you can see we now converted the data into three rows, one for each name. We have three columns, instead of two. The counts are now split into low and top performance counts. That makes sure we have one column for each variable, and one row for each observation. There is also an alternative available for this feature. We again either need the `reshape2` package, or the `data.table` package, which contains many functions from `reshape2`. The function we are going to use here is called the `dcast`. The structure of the function is similar to the `spread` function, but here we have to use a formula argument. So we specify the data, performance, and we need to form the argument with a tilde. Left side states the columns, which should stay as they are, and the right side of the formula shows the columns which we want to spread. In this case, it is the performance column. The `value.var` argument is the variable, which in this case is called counts. If we run this line, you can clearly see that the result matches the one we previously generated. `Dcast` as well as `spread` are both great tools for long to wide format conversion.

## Joins in Dplyr

There might be instances where your data is divided into two or more datasets. Let's say in a bank one department is responsible for data on personal savings, and another department is responsible for loans. Each department has its own dataset, but the customer base is largely the same. In such a case, you can use mutating joins to fuse the datasets. However, the data structure and format must be identical. There also needs to be at least one common field. By the way,

splitting tables is also possible with filtering joins as we will learn. You might have heard about the left, right, inner, and outer joins. So we are going into the terrain of database normalization here. This is actually a standard task in SQL processing, but also here in R you might have to perform those tasks. Dplyr has a whole tool set dedicated to this problem. It is called the two table words. Those two table words, we are speaking about mutating and filtering joins, so those dplyr two table words always have a similar structure. You always have the first two arguments, x and y, which are the two tables to work with. A typical dplyr joined function is fairly simple to code as you will see in this demo. Note that we are speaking about operations with two tidy datasets of a strong similarity. Hence it is called two table words. If you have more than two tables, or data frames you want to work with, you would have to look for alternatives like working with lists and stuff like that. Alright so let's say we have two data frames we want to work with at the same time. There is df1, which is a data frame with five observations on three variables. We have names, the corresponding body height, and the specific category. The df2 on the other hand has only four rows and two variables. We again has the names, but here their weight in kilograms is present, instead of the height in centimeters. There is no extra category available. If you compare the names, you will find that two observations, namely Sue and Jim, are available in both data frames. This is crucial to understand as we will see shortly. All right so now we are going to start with joining them together. There are basically four types of mutating joins and two types of filtering joins available in dplyr. For a join, the data sources we use must have common data. Without this data, we cannot join them together. This is often an ID or name column. In our example, this is clearly the name column, which is available in both datasets. This is two observations of the same person, which allows me to demonstrate all four types of mutating joins. So the first join type is the inner join. It pulls data for an observation which is present in both data frames. In our example, we have Sue and Jim matching. When we run inner join on x and y, x in this case is df1, and y is df2, we get this output down here. As expected, there are only two rows, Sue and Jim. All the info available is displayed here in the output. We get height and category from df1, and we also get the weight from df2. So that was the inner join which is the strictest type of join with the smallest subset of info output. The next one is the left join. If a left join is applied, R returns all data from x, but only the matching data from y. If an entry is listed in table A, but it is not in table B, we get NAs for the columns where there is no data available in table B. So let's run the left join. Code for that is simple, left\_join, and then again x and y data frames. So as expected, we get the full info from x, and there is the extra variable weight added now from data frame y. But of course, only for the two matches it was possible to enter this info. The rest gets an NA. A right join does the exact opposite of a left join. If a right join is applied, R pulls all the rows from y. If an entry is also present in x, R will add this info as well. So if we run the code with

`right_join`, we can see that we have all four rows from `y`, and for the two matches, we also get the info and category and length taken from `x`. The rest is again NA. So this is nothing else than the inverse from the left join. If we would run a left join but switching `df1` and `df2`, we would get the same result. The only minor difference would be the fact that variables are arranged in a different order. The fourth type of the mutating joins is called outer join. It returns all rows from both of the supplied data tables. Here the order of the rows matters. First displayed are the results of `x`, and subsequently of `y`. There is no repetitions, if we have a match it will only be displayed once. As we can see, on running `full_join`, we basically get a list of all available data on all observations.

Matches are displayed once, and missing data is an NA. So that one is the least strict type of joins, and will not dismiss any information. Those were the four mutating join types. They mainly effect the variables, or the columns of your datasets. If you have worked with SQL before, it is likely that you already know this. Now, we take a look at the filtering joins. As the name says, they help you to filter one dataset, based on another one. There are two of those filtering joins, the semi and the anti join. If we start with the anti join, and we simply run `anti_join`, which is again a part of `dplyr`, we will see that this one excludes all rows from `x`, which are also present in `y`. Since we know that we have two matching names, there should be three observations left, which we see down here in the console. Note that this filtering join filters on `x`, so extra info from `y` is not important. We need the second data frame just to search for matches, and to exclude those matches from the results. We are now going to compare this one with a semi join. This one is the exact opposite of the anti join. Here we get only the matching rows. Those should be Sue and Jim, which are available in both datasets. And we also get only the info from data frame `x`. After executing this `semi_join` command, we can indeed see that only the matching info is returned in the console. This confirms that anti and semi joins are in fact complimentary to each other. If we add both of them together, we would get the original data frame again. So those were the four mutating joins and the two filtering joins which are a very useful feature from `dplyr`. By the way, you could code things like that in R Base as well. The function in R Base would be the `merge` function.

## Summary

Let's summarize what we did in this module and why we did it. This module had the general theme of cleaning datasets, and preparing them for further analysis. It was all about proper data format. You want each variable to have its own column, you want each observation to have one row, and you want each experiment or observational unit form one table or a data frame. We were mainly using `tidyr`, and it's alternative, `reshape2`. For the joins we used `dplyr`, which is one way in how you can do this in R. We used the `gather` function of `tidyr` for wide to long conversion,

melt is the reshape2 analog to that. We used separate, of tidyr, for cell splitting. This is useful as long as there is a clear pattern in the data. We used the spread of tidyr where dcast of reshape2 for long to wide conversion. And at last, we took a look at the two table words in dplyr for joins. These were the inner, outer, left, and right join for mutating joins, and the anti and semi join as filtering joins. In the next module, we'll learn about data filtering and querying, where the package is dplyr, and data. table.

# Data Filtering and Querying with dplyr and data.table

## Introduction

Hi there, this is Martin Burger for Pluralsight. This module is all about data filtering, or running queries on your datasets. We will start out with some theory on queries and the appropriate packages including the great data. table package. To put a theory into practice, we will do some demos using two different datasets. The first one is a simple ad-hoc dataset, while the other one is the diamonds dataset of ggplot2 with over 50, 000 rows. You will learn step by step how to query on a row and on a column level. I will show you how to group your queries for more detailed outputs. And I will show you how to use keys in data. table to make queries faster.

## What Is A Query?

This module is all about querying your datasets. What do we mean by querying? Well, this is the process of selecting certain parts of the dataset. You extract a specific part of the whole dataset by providing a set of criteria. This is also called data filtering. You can filter by one criteria or by many. If you have a huge dataset, say the phone book of New York, and you're only selecting people with Smith as second name, you are performing a query. Of course, this is a very simple example but what if you want to query for Smith in the 7th district, male, and born between 1975 and 1980? Well, that would be several things to consider in your query. We are lucky that things like that are quite easy in R. Especially since the package, data. table is a great tool for that. On the general level, you can query for rows, or for columns. Going back to our phone book example, if we are interested in all data available for Smith, this would be a row level query. Most of the queries you will perform in your daily work are at that level. Of course, there can be quite some

complexity. The more filter criteria you use, the more complex this whole thing gets. When you have data in a wide format with many columns, it can get necessary to filter for specific columns. Basically, you're extracting columns in this case. This is performed similarly to a row query. If you want to count a specific event in the dataset, this is also done at the column level. Querying gets more interesting the bigger the dataset gets. Bigger datasets are difficult to figure out, hence the need for more or less complex queries. And this is where the `data.table` package comes into play. The creators of this package realized that R has some weak points when it comes to handling and filtering big datasets. They create `data.table` to cover that blind spot in R. It's a very popular and widely known add-on package that is in many ways superior to R Base. You can imagine this package as a separate system within R. A system for data management, data manipulation, data structuring, and most importantly, querying data. The core component of the `data.table` package is the object class, `data.table`. That is basically an advanced form of a data frame. As you might know, the object class mainly used in R is the data frame. Most of the data you will encounter in your work or research, will have a form which fits into a data frame. Of course, there are some exceptions like matrices or even date and time related data, which you would fit into a time series object `ts` or `mts`. So what makes a data frame so special? Well, on one hand it is easy to understand. It has a row for each observation, and a column for each variable. There is a simple pattern looking quite like a matrix, and there are nearly no limits to the number of observations and variables. Now, a very important thing to keep in mind here is the difference between a matrix and a data frame. In a matrix each value, thus each variable or column, has the same class. A data frame is different. Here each column can have a different class, for example numeric, integer, character, Boolean, and so on. And that is what your data will probably look like. Think about sales data with integers, also quantity, numeric, revenue, and character for sales region and shop. With a data frame, you can easily handle that sort of data. Now, since this class, data frame, is part of our base, you likely worked with it. Most of the exercise datasets like `mtcars` are in that exact class. Now, you might ask yourself why we are talking about data frames so much when this whole module is mainly about data tables? Well, the thing here is that you can see the new class, `data.table`, as an advanced type of a data frame. Like in a data frame you can handle matrix like data with different classes with a data table. In fact, the whole class, `data.table`, has its own structure, which is very similar to an SQL structure. If you are already familiar with it. It looks like this - the name of the data table, `i`, `j`, and `by`. Understanding this structure is key to handle this class. You can filter, sort, or order with this understanding. So let's see what those letters actually mean. The `i` stands for the subset from our data table we want to work with. This comes down to row level filtering. `j` is the actual calculation that will be performed in the data subset `i`. This comes down to column level filtering. And this whole calculation will be grouped by `by`, and this

lets you refine the output. Let's compare this with a simple data frame, with name of the data frame `r` and `c`. In a data frame, you insert the info for row level filtering on the left side of the comma. And the info for column level filtering on the right side of the comma. So this set up is analog to a data table, however there is no third parameter by to group with. This makes the data frame a bit more complex to work with.

## Data for the Demos

In the next few videos, you will learn how to effectively query and work with a data table. At the same time, I will also show you how things work in a standard data frame. For the demonstration, we use two datasets. The first one is a primitive ad-hoc dataset we generate ourselves, as data. frame and data. table. In the last videos of this module, we will use the diamonds dataset of ggplot2, which provides opportunity to test our new knowledge on a large scale. Let's first create our ad-hoc test objects. We will set up an object as a data frame, and a separate object for a data table, to be able to compare those two classes. Let's call our data frame, `mydf`. It has three columns. `A` is the name column, `b` is a categorical column, and `c` contains some measurements. I'm setting a seed here in order to always get the same numbers for column `c`, which is a randomly generated vector. Basically, this is just a simple dataset like we are using several times throughout this course. The whole thing has seven observations. We are going to use the same data for our data table object. Here we are calling it `mytable`, and it again has three columns on seven observations. It is the same data as before, but the whole thing has the class data table, as you can see with the function called `data. table`. Note that you of course need to have the data. table package on your computer, and activate it. So if you now take a look at the output, we can instantly see that there are some differences. First, there are row IDs with those double dots. That is a clear indicator of a data table. In the environment, or by using `supply class` on all columns of the data, we can also see if the data frame automatically transformed the columns `a` and `b` into vectors, which I did not intend. This was correctly recognized as character by the data table, as we can see again here. And if you compare the classes of both objects, we get the one class data frame for our data. frame, but we get two classes for the data. table, which has data. table as the primary class, and data. frame as the backup class. All right, so those are the basic differences between those two. We will now dive deeper into data. table, and do some comparisons against a data. frame.

## Queries at Row and Column Level

Now let's see how those two objects compare while querying. Therefore, we now start with the querying process. By querying I mean the process of filtering the data in an accordance to the specifications provided. One can query on the basis of nearly any information the data contains. At the beginning, I demonstrate simple row and column level queries in both data. table and data. frame. I will also show you how to obtain counts of observations that are meeting certain criteria. Again, mydf is a data frame, and mytable is a data table. The values in both datasets are identical. A quick note on the syntax. When it comes to querying, always think about using box brackets. You must use those box brackets for querying both data. table and data. frame. So let's say we want the third row, which means the third observation. A standard data. frame would be queried with number of row on the left side, and column number on the right side of the comma. That simply means if we go for box bracket 3, and then comma, leaving the right-side blank, we get their whole information on the third observation. So we are talking about Nora in group B. At this simple level, this looks similar to a data table. We can use the same command, and get the same info as output. But note the difference at the beginning of the output. We have the number 3 at the data. frame which is the row id, which does not come with a data. table. We get simply a 1 instead, which indicates the first line of the data. table output for this data. table query. So this was a simple query at the observational or row level. But what happens if we simply state the number 3 in box brackets, without using a comma? What would be the default result for that query? Well you can see that here for the data. frame we get the third column, which is C, as a data. frame output. So we basically made a query at column level. On the other side, if we do this with a data. table, we get the third observation, so this is about the row level here. Keep that difference in mind, since it reflects the basic structure of a data. table. Where the first position in the box brackets always represents the i, the row level subset of the data. By the way, using a data. frame, we can extract this column data into different ways as we learned right here. The first method, by just using the column ID without a comma, gives us a data. frame with the data. On the other hand, if we indicate the column ID on the right side of a comma, we get this simple vector with the same info. So using a comma here determines the class and structure of the output. Now how do we extract the column data from a data. table? Well, there is the parameter j, so the second parameter of the dt syntax. We can use the column ID to get a data. table with a third column. Or, we insert the name of the column to get a simple vector. In this case it is c, so I'm going to type box brackets, and at the right side of the comma, I'm stating the column id. This gets us the already known output, the desired information of column c. However, we get this whole thing as a simple vector. By the way, there is a trick to get outputs as data tables. This might look trivial at our simple example here, but in case you have to deal with several columns, and you want to perform further work with that output, this might be of great help. Therefore,

you basically have two ways in how to do this. You either state dot, and in round brackets the column name, or you state list, instead of the dot. The dot is actually an abbreviation for list in the data.table system. So you can keep this shortcut in mind since it's going to save some time. And we'll use this one several times when working with data.table. So as you can see in the console, we get the exact same output for both lines. It's the same content, and the format is always a data.table with only one column c in this case. We can actually go a step further, we can do the same query for column c, but in this case, we are not only querying the column, we are also renaming to the column name, c name. This is now displayed down here in the console, where we again get a data table with the info from column c, but we have a new heading for this one in this case. In the next case, we'll use the second parameter j to learn how many rows there are with the column b category of B. Therefore, we are indicating comma, to make sure we are going for j, and after the comma, we state sum, round brackets, and in those brackets we need a logical statement of column b equals B. There are three observations of this category B, so we get a 3 as the output down here in the console. There is an alternative way for this as well. This time we are building it on the parameter i, which indicates that we want all rows with a b value of B. After a comma, we use the upper-case N, after a dot.. N is a special number indicator in data.table. It indicates that you are looking for counts. So each time you use it in your data.table code, you will get the amount of instances that are matching your criteria. In this case, this is of course 3 again. Three observations of category b. A brief reminder at this point, you should already know that in R, if you want the logical operation equal to, you have to use the double equals sign, like I was using in the last two lines. Now, how does the old school data.frame way look for this? Well, there are several ways in how you can get the counts for matches. In this method, I'm getting the length of the output of all b columns, which have a B present. As you can see, things like that are easier to code with the data.table system, especially since there are special indicators integrated. Now, what we did here with the count of Bs, was basically a filtering of rows. You filter a row at the left side of the comma, like we do with this simple code over here. I'm just searching for all rows that have category a in the second column. And I want the output to be a data.table, with the whole info on those rows. Therefore, I'm not going to state anything after the comma. So if we run this line, we get the two rows in the output down here with the full info available. And it is clearly structured as a data.table, since we see those double dots over here. We can do the same thing in an old school way. Subset is a key function of our base and works not only on data.frames, but also on data.tables. If we run subset on mytable, and we are searching for b equals category A, we again get a data.table with those two observations that are in that category. So it is the exact same output as before. We can of course bring it a step further, and we can output only the column A values for those two rows. Therefore, I am adding a new argument to the



subset function. It's the select argument, which indicates the column name, which is column A. So we only get the names in column A as the output. Now, how would this filtering process work with a data frame? Well, we already learned that the left side of the comma contains the info for the rows. So we would have to state column b equals A, to get this result. But here, we would also have to use the dollar sign and a name of the data frame, in order to make clear that we are searching for this info. Of course, we could attach the data frame to the environment in order to omit the specification of the data frame, and the dollar sign, but in either way, it requires an extra step to get access to the data frame. And of course in this case the output is going to be a data frame, as we can see with the row IDs over here without the double dots. It's the first two rows of the original data frame, therefore, we have number one and two stated.

## Combined Queries

In this demo, we get a step further and combine the first parameter i, and the second parameter j, of a data table. That means we are combining row and column level queries. I will also show you how to transpose a data table. That is basically a swapping of columns. And we will also change the order of the dataset based on a given set of criteria. In the first case, we are trying to get the summation of the values in column c. That is the operation we want to perform on rows three through six. Therefore, we are stating 3, double dots, six at the left side of the comma, indicating the row IDs. And we get the summation of the numeric c values, which is around 5. By the way, if you use the exclamation mark, like over here, it means accept, so we are reading this line like get the summation of column c for all rows except rows three through six, which in this case is 2. 5. So far we learned about the first and the second parameter, but there is even a third one. It's the by parameter. This one helps us to organize the output by a specific criteria. Let's take a look at an example with the by parameter. In this first example, we are assessing the counts. As we already learned, we can do this with the uppercase N shortcut. So we are using this like an appendix. The main point here is that we want the output to be grouped by the three categories of column B, so we are stating comma, we take the left side of the comma, indicating argument by, and we set it to b. And we get the appendix of. N. And if we take a look at the output, we learn that we have two observations in category A, three in category B, and one in category C. We can take this whole thing a step further, and we can indicate that we want the counts of all possible combinations of the names in A, and the categories in C. Therefore, we are stating dot or list, and then B and A, as the column names. Since this is just a small test dataset, we only get ones for the possible combinations in the dataset. But of course, this can be really helpful if you have large datasets of thousands of rows with all sorts of combinations. Now, sometimes it can be helpful to

turn the whole dataset upside down. That is basically transposing it. The transpose function works on nearly all our objects, may it be `data.frames`, `data.tables`, or lists. So it is very versatile. If we use this one on our dataset `mytable`, we get the output down here. The names are now all in the first row, the second row are now all the categories, and the third row other numeric measurements. So we exchanged rows and columns with this function. Another useful function for `data.tables` is the `order` function. It's similar to the `arrange` function from `dplyr`. If you want to order your dataset in a descending order, you can use the minus symbol in front of the column id. Per default, you would have an ascending order set in the function. So if we run `order` on `-b` and `-a`, we get B as the primary order source, and in case of ties in B, it will be ordered descending according to A. So let's take a look at our dataset here. Since we are dealing with strings, its ordered reverse alphabetically. Clearly, we can see the categories of column B to be ordered descending, and the ties, especially for category B, are ordered descending after the first letter of the names. We were using this `order` function as our parameter `i`, therefore, we do not need to state any commas. If you want to permanently change the order of the dataset, you can use the `setorder` function. We are going to use the same order in a descending fashion for B and A. So the function looks like this - `setorder`, and in round brackets the name of the dataset, and then the criteria to be ordered against. If we run the line, take a look at the dataset afterwards. We can indeed see that the order of the data was now set permanently. Always be aware of `data.table` functions, starting with a `set`, because those ones are going to change your dataset for further calculations.

## Converting to `Data.table`

If you have a `data.frame`, or some alternative class, and you want your data to be changed to a `data.table`, it's very easy to convert to the `data.table` class. That means by inserting one short line of code, you can get access to the power of a `data.table`. In this demo, I will show you how to convert to a `data.table`, and how to check which `data.tables` are already active in your environment. In this example, I'm using the `mtcars` dataset, which is part of R Base, so you have it on your machine. `Mtcars` is a `data.frame` per default. Now if I simply use the function `data.table`, it will be changed to this format. Let's call this object `newmt`, and check the class in the same line. And indeed, now we have the primary class `data.table`, as you can see here in the console. So that means if you are currently using a `data.frame` for your data, just a simple line like that, and you have all the advantages of `data.table` available when you process your data. By the way, if you run `tables` with empty brackets, R is going to show you all the `data.tables` you have stored in your environment at that moment. It gives you a quick idea on your current work, in case you lost

the orientation on the objects you already created. In this case we have two data tables already created. It's the mtcars dataset we converted to a dt, and now primary data table we used for all the intro functions. A quick way in how you can see the class of each column, is by checking out the environment over here. You might have to click this icon to open the object and see the class of each column. An alternative way would be a code line like this. We use sapply from the apply family of functions. We are using it on our test table, mytable, and the second argument here is the class function. So that basically means we want the function class to be applied to all components of the mytable object. And if we check the output, it is once again confirmed that the first two columns are indeed characters, while the third one c, is a numeric vector. This is valuable information since you need to know the class of the vector when you write functions for this vector.

## Using Keys in in Data.table

Now if you look at the data frame, you will see that you can have row names as ID. This can be a simple number, but it can also be a name. Think about any registry or medical research data. You can only have one column as row ID, so what do you do when you have a name appearing multiple times in a dataset, or there are several columns containing names? Data table offers a solution for that sort of scenario. It's the key. A key is a sort of super charged row ID, which even helps you to organize and order your data. Keys can have different data types, integer, numeric, character, and so on. And they can be set on multiple columns. The key data does not need to be unique, thus it is possible that the same name appears multiple times in the dataset. So if you set a key, your data gets automatically reordered according to the key column. Let's now set the key in our dataset, and let's see how this whole thing changes our dataset. So this is the original dataset. See the order of column A here. We have Paul first, and Kim is the last observation. We will now set the key to column A, using the setkey function. We have a simple structure to this function. First the name of the dataset, and second the column we want to set the key on. So it's column A in this case. If we run the line, we can now see that the order of the whole dataset has changed. The whole thing is now ordered alphabetically, starting with k as the first letter here, and Sue, S, as the last one. So that is basically in increasing order based on the first letter of the key values, which are characters in this case. Conversely, if you have a data table and you do not yet know where you or your colleagues set the key, just use the key function on the table object, and R will tell you where the key is at. In this case it is off course in column A. The key feature makes querying based on the key column extremely easy. Each time you specify a value from the key in box brackets, R will recognize what you are saying, and it will give you the whole

information, the whole row on that specific entry. So here for example we are searching for the whole information about Kim in the key column. We have two Kim's here, therefore we get those two rows as output. Of course, this only works on the key column. If we are using some info from another column, like here, the category A from the second column, it won't work. We get this kind of useless info down here. This would look totally different if the key was set to column B, then it would work and we would get all the rows containing A as a return. We can insert several values from the key column, as long as we properly concatenate the input. In this case I want to get all the info on Kim and Paul. Therefore, I'm using concatenate, and in brackets I am stating the two strings. So it is the same procedure, just elaborated to two names as we can see down here in the console. All of the info on Kim and Paul is available due to this vector inserted in the function. We can of course also use the second parameter and do some operations on the selected rows. Here I am stating that I want only the column C from all observations with the name Kim. I'm using the alias for list, in order to get the data. table, like we do down here in the console. If we would only state c, like they do in this slightly changed line here, we get the simple vector with the two values of C for the two Kim's in our dataset. It is a minor change in code, but the output has a totally different format. So as you can see, using a key makes querying even easier, especially on the targeted column.

## Filtering Big Data

So far, we learned about the most common functions of data. table, and how to apply them on the small exercise sample. Our data. table only had a few rows. This is perfect to illustrate the power of the package, but you are surely interested in applying this material in real world big data samples. I love R, not just because it is quite versatile, but also because there are so many exercise datasets available. We will now work with the diamonds dataset from ggplot2. To access the data, you have to install the famous data visualization library, ggplot2. Let's take a look at the diamonds dataset. As you can see this whole thing has 54, 000 observations on diamonds. There are 10 variables, one of which is the price. There are also things like coloration, cut, dimensions, clarity, and so on. All of those things can possibly influence the price of the diamonds. Let's put this thing in the data. table, which we are calling newdiam. If we now view this object, newdiam, we will see one of the advantages of this class. Not all of those 54, 000 rows are displayed. We only get the five last and first rows, which will do just fine and does not crash R, or steals precious time. We can also get the summary of this info to get an idea about the limits of values, counts of categories, and so on. All right, so we now have confirmed that this is a data. table, we know a bit about it, so we can now start to query this table. We now are going to solve those eight items

over here. How many diamonds are more expensive than \$10,000? We are going to do this in two different ways. How many diamonds are more expensive than \$10,000 for each color? We are going to create a table with diamonds cut ideal, and the color is either E or J. We order the table by price, we extract a table with price and cut. How many diamonds have combined x, y, z dimensions, the summation of greater 18? We can answer the question, what is the mean price of diamonds with a cut of ideal and premium? What is the number of diamonds with a price over \$10,000, and a cut of premium? You can actually pause this module and try to code the assignment if you want, or you can follow along and see the solution I'm going to present to you. So the first task was to get the count of diamonds more expensive than \$10,000. Here we have two different ways in how we get the number of diamonds greater than \$10,000. The correct answer for that would be 5,222, and both lines give us that output. But how do we code it? Well, the first line is a bit longer and less elegant. It's a standard R Base way, which you might already know. It's the number of rows, therefore `nrow`, of the subset, which is the second function in this nested structure. We are dealing with a subset on the dataset `newdiam`, so that is the first argument in brackets. Then we have the criteria, and price, greater \$10,000, is the second argument. Again, it's a long form compared to the alternative. But it works on nearly all sorts of data in R. It is one of the ancient ways to do this. You could probably have used this method in the early 90s and it would have worked. The other one is a classic `data.table` approach. We use the shortcut for count with `N`, and we are specifying the first parameter `i`, to select the rows. This is as easy and efficient as it gets in a `data.table`, and we get the same result as with the first method. The second task was to get the count of diamonds greater than \$10,000 in price. That is the same as before, but this time we want this whole thing to be categorized by coloration. For each color there should be a count available. For the sake of simplicity we are going to use the previous solution with a `data.table` approach. The only thing we need to do here is to attach the parameter `by`, and we set `by` to `color`. And as you can see, we get this `data.table` down here with the counts for each color type. There are seven colors, hence we get the `data.table` of length seven. For the third question, we have to get a `data.table` that contains all rows with a cut of ideal, and the coloration should be either E or J. We have to use all sorts of operators here, it might seem a bit complicated. So let's see how this is set up. The general idea is to only use the first argument `i` in a `data.table`. That means no comma is required. The cut needs to be ideal. We can state this of course with double quotations. Then, there is an extra criteria, which is an end criteria. Since we have two types of colors that are accepted, we can use the round brackets, we are stating `color equals E`, then `or`, which is the pipe, or horizontal line if you will, and `color equals J`. And if we run this whole thing, we get this `data.table` which indeed only contains ideal cuts in those two colorations. So it worked and we successfully filtered our `data.table` in accordance to

those two criteria. The fourth item was to order the dataset by price. There are of course several ways in how you can do this. Dplyr has the arrange function that might work here as well, but since we have a data. table, we can simply use the order function on price, and we are done. We get a new data. table down here, which indeed starts with the cheapest diamond, and ends with the most expensive ones. If you would use the minus symbol before price, it will be the other way around in descending order. With the next item, we get a new data. table with only the price and cut variables of our newdiam dataset. That means you have to use the list function in order to get those two variables in the function. The dot is an alias for list, like I'm using here. Since this one deals with the columns, you would also have to use the comma and write the code at the right side after the comma. Keep in mind, column specifications are made at the right side of the comma, with the data. table. The new data. table has only two columns, as you can see in the console. All 54, 000 rows are present, since we did not specify any row filtering at the left side of the comma. With the sixth item, we perform a summation of all three dimensions, x, y, and z. The sum should be greater than 18, and we were interested in the count of diamonds that actually met this criteria. Well, this one is again an operation on the columns, therefore comma, and on the right side we can state that we want the sum, which will basically give us the count of the rows that have x, y, z of greater than 18. And the result in this case is 8, 296. We can also run this in a different way, which is unique to data. table. In this case we are going to use the N count feature again. We are now setting it up with the x, y, z addition at the beginning, and after the comma we are using the count abbreviation. Works well on a data. table object, and gives us the exact same result. The idea on the seventh item is to get the mean price for diamonds with a cut of ideal and premium. So how do we code this one? Well, we are first selecting for the rows with a cut of ideal. Then, the or operator, the pipe, and after the pipe the cut of premium. The second parameter is a list, which contains the function mean on the price, and in this part of the code, I'm also providing a new name to this part of the data. table, I'm calling it meanprice. And at last I make sure we are grouping the cut, which in this case will only be the two cut types, ideal and premium. And indeed, down here in the console, we get the new data. table with the two cut types and the new column called meanprice. We have a meanprice of 3, 457 for the ideal cut, and 4, 584 for the premium cut. With the last item, we are interested in the count of diamonds with a price higher than \$10, 000, and a cut of premium. For counts we were again using the N alias, and we put it at the right side of the comma. We are filtering for a number of rows that have a price greater than \$10, 000 and a cut equal premium, which is of course a character, therefore we are putting it under quotations. If we set it up like this we get a single number, which is the count down here. If we want this whole thing to be a comparison of the counts that match our criteria, and the ones that do not match, we can set it up differently. Here, we are using the search criteria as our last

parameter, the group by parameter. What we are saying here is that we want the counts of all diamonds, which is all rows, therefore the first parameter is empty. And we want the result to be grouped by the ones that meet the criteria, which is price greater \$10, 000, and cut premium, and the ones that simply don't. To get the counts we again use the data. table N abbreviation. We get the same answer as before 1, 807, but here also the extra info on the ones that do not meet the criteria as displayed. As you can see, this whole syntax is a much more nuanced one. A small change can make quite a significant difference in the output.

## Summary

Let's summarize what we did in this module. After we discussed some theory behind queries, and a comparison of data. table with a data. frame, we created our ad-hoc test datasets. At first, we made simple queries on row and column levels. And I showed you how to obtain counts of observations matching certain criteria. Then, we combined row and column level queries, and showed you how to transpose a data. table, and how to permanently rearrange the order of your dataset. We learned how to convert data. frames into data. tables, and how to check which data. tables are already active in your environment. We discussed how to use keys for fast and efficient queries, and at last I showed you how to run various queries on the diamonds big dataset. In the upcoming last module, we will discuss what we learned in the course, and I will show you where to get further information on data management in R.

# Course Recap and Your Next Steps

## Introduction

Hi guys, it's Martin Burger for Pluralsight. In this last module, I will show you some great resources to identify suitable libraries, how to get help, and where to find some practice opportunities. This is crucial info, since you will encounter various problems in your daily work. Knowing where to get help is the first step to solve these issues. At last, we'll also summarize what we did in this course.

## Resources and Next Step

In this video, we'll discuss some helpful resources you can use to get help. I will guide you to an exercise database, and we'll talk about the R package registry. In R, one of the main challenges is to find the right packages with the right functions for a given problem. There are several approaches in how to identify these packages. One would be to consult with the guy on Stack Overflow, and see what peers have to say. Stack Overflow is an online community for all things programming and technology related. If you are developing a new project, this is the best source to get quick and efficient help. People are usually very helpful and provide guidance on either package selection, or specific coding questions. A systematic approach for R package discovery is to check out the R task view. The task view is a collection of packages for a certain topic. There are task views available for a lot of data science related disciplines. Unfortunately, there is no specific task view for data management and data preparation. However, I can recommend the task for you in multivariate statistics. It contains a section on missing data, and imputation libraries. And it also has a link to the mvoutlier package, which is the go-to resource for outlier detection. Basically, this is a great resource for the data cleaning steps we did not cover in this course. Handling of missing data, and outlier detection. A further helpful resource is the R bloggers blog aggregator. Most of the R related blogs have a direct feed into this aggregator. A lot of R topics are discussed and many of them are of good quality. Of course, here on Pluralsight, you have several courses available for many subdisciplines of R. The library is regularly updated, and new courses are added. If you want to practice R, there is the R exercise database over at [r-tutorials.com](https://www.r-tutorials.com). There are blocks of 10 questions available for various topics including data management. And this one also has a very convenient reveal solution feature.

## Course Summary

Before we go through the course summary, I want to point out where you can ask questions and how to best provide feedback. For questions related to this course, you can ask a question in the Pluralsight discussion board. You can also leave a comment in the Pluralsight interface. I would also ask you to leave a rating to this course on Pluralsight. All right, so we started the whole course by taking a look at the data science landscape. We learned that the data preparation part consists of several steps, data import, selecting a proper class, data cleaning and further formatting, and lastly, running queries to get a subset. After the data preparation part is completed, you can continue with the analysis or data visualization. In the second module, I showed you how to import data, and which classes are most suitable for your dataset. Since there are dozens of different file formats, some of them rather exotic, it can be tricky to import all files. We learned several ways in how to import CSVs. I encourage you to use CSV whenever possible. This should



be also communicated to your colleagues so that the whole data analysis process is optimized in your company. We took a look at the `fread` function of `data.table`, which is a really versatile tool to get data into R. And I showed you some ways in how to copy and paste data into R, including the `scan` function. For more exotic file types, I discussed packages like `foreign`. The stand out class in R is the `data.frame`. It is very versatile, well implemented, and well documented. But we now also have the `data.table`, which is a great alternative to the `data.frame`, especially when you have big datasets. In the third module, we used packages `tidyr` and `reshape2`, to clean and reorder data. I showed you how to perform long to wide and wide to long format conversions, as well as cell splitting. These are functions tailored to what's tidy data. One observation per row, one variable per column, and one experiment or observational unit per table. We also took a look at filtering and mutating joins, which are well implemented in the `dplyr` package. In the fourth module, we took an extensive look at queries. We used both `data.frames` and `data.tables` to perform queries. I showed you that the `data.table` is more versatile when it comes to queries. You can go into much more detail with the query outputs. We queried at both the row and column level, and I also showed you how to group the output with the `by` parameter of a `data.table`. We worked with a simple dataset, as well as with a diamonds big dataset, with over 50,000 observations. As you saw, the underlying principles are the same, no matter how big the dataset is. All right guys, this concludes the R data management and data preparation course. I hope you gained useful skills for your daily work. This subject is a crucial one, so you will likely encounter situations where this knowledge will come in handy. Do not forget to add this course to your CV, so that your employer knows that you trained yourself in the latest technology. All the best for your daily work and in your career.

#### Course author



Martin Burger


Martin studied biostatistics and worked for several pharmaceutical companies before he became a data science consultant and author. He published over 15 courses on R, Tableau 9 and other data...

#### Course info

Level

Beginner

Rating  (20)

My rating 

Duration 1h 59m

Released 18 Sep 2017

Share course

