Data Science 201 Text

Patrick Mathias, Adam Zabell, and Randall Julian

1/23/2018

Contents

1 Lesson 1: Adopting principles of reproducible research

1.1 What is reproducible research?

In its simplest form, reproducible research is the principle that any research result can be reproduced by anybody. Or, per Wikipedia: "The term reproducible research refers to the idea that the ultimate product of academic research is the paper along with the laboratory notebooks and full computational environment used to produce the results in the paper such as the code, data, etc. that can be used to reproduce the results and create new work based on the research."

Reproducibility can be achieved when the following criteria are met (Marecelino 2016): - All methods are fully reported - All data and files used for the analysis are available - The process of analyzing raw data is well reported and preserved

But I'm not doing research for a publication, so why should I care about reproducible research?

- Someone else may need to run your analysis (or you may want someone else to do the analysis so it's less work for you)
- You may want to improve on that analysis
- You will probably want to run the same exact analysis or a very similar analysis on the same data set or a new data set in the future

"Everything you do, you will probably have to do over again." (Noble 2009)

There are three practices we will cover in this lesson to help get your code to be more reproducible and reusable:

- Develop a standardized but easy-to-use project structure
- Adopt a style convention for coding
- Enforce reproducibility when working with projects and packages

1.2 Develop a standard project structure

In their article "Good enough practices in scientific computing", Wilson et al. highlight useful recommendations for organizing projects (Wilson 2017):

- **Put each project in its own directory, which is named after the project**
- Put text documents associated with the project in the doc directory
- **Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory**
- Put project source code in the src directory
- Put compiled programs in the bin directory
- **Name all files to reflect their content or function**

Because we are focusing on using RMarkdown, notebooks, and less complex types of analyses, we are going to focus on the recommendations in bold in this course. All of these practices are recommended and we encourage everyone to read the original article to better understand motivations behind the recommendations.

1.2.1 Put each project in its own directory, which is named after the project

Putting projects into their own directories helps to ensure that everything you need to run an analysis is in one place. That helps you minimize manual navigation to try and tie everything together (assuming you create the directory as a first step in the project).

What is a project? Wilson et al. suggest dividing projects based on "overlap in data and code files." I tend to think about this question from the perspective of output, so a project is going to be the unit of work that creates an analysis document that will go on to wider consumption. If I am going to create multiple documents from the same data set, that will likely be included in the same project. It gets me to the same place that Wilson et al. suggest, but very often you start a project with a deliverable document in mind and then decide to branch out or not down the road.

Now that we're thinking about creating directories for projects and directory structure in general, let's take the opportunity to review some basic commands and configuration related to directories in R.

Exercise 1

1. Navigate to your **Preferences** for the RStudio application and note the *Default working directory (when not in a project)*
2. Navigate to your Console and get the working directory using `getwd()`
3. Review the contents of your current folder using `list.files()`
4. Now try to set your working directory using `setwd("test_dir")`. What happened?
5. Create a new test directory using `dir.create("test_dir")`
6. Review your current directory
7. Set your directory to the test directory you just created
8. Using the Files window (bottom right in RStudio, click on **Files** tab if on another tab), navigate to the test directory you just created and list the files. *Pro tip: The More menu here has shortcuts to set the currently displayed directory as your working directory and to navigate to the current working directory*
9. Navigate back to one level above the directory you created using `setwd("..")` and list the files
10. Delete the directory you created using the `unlink()` function. Learn more about how to use the function by reviewing the documentation: `?unlink`. Pay special attention to comments about deleting directories.

Optional Exercise (If you do not already have a project directory)

Now that you're warmed up with navigating through directories using R, let's use functionality that's built into RStudio to make our project-oriented lives easier. To enter this brave new world of project directories, let's make a home for our projects. (Alternately, if you already have a directory that's a home for your projects, set your working directory there.) 1. Using the Files navigation window (bottom right, Files tab), navigate to your home directory or any directory you'd like to place your future RStudio projects 2. Create a "Projects" directory 3. Set your directory to the "Projects" directory

```
dir.create("Projects")
setwd("/Projects")
```

Alternately, you can do the above steps within your operating system (eg. on a Mac, open Finder window and create a folder) or if you are comfortable working at the command line, you can make a directory there. In the newest version of RStudio (version 1.1), you have the option of opening up a command line prompt under the Terminal tab (on the left side, next to the Console tab).

Exercise 2

Let's start a new project :

1. Navigate to the **File** menu and select **New Project...** OR Select the **Create a project** button on the global toolbar (2nd from the left)
2. Select **New Directory** option
3. In the Project Type prompt, select **New Project**
4. In the Directory Name prompt under Create New Project, enter "intermediate-R-course"
5. In the Create Project as a Subdirectory of prompt under Create New Project, navigate to the Projects folder you just created (or another directory of your choosing). You can type in the path or hit the **Browse** button to find the directory. Check the option for "Open in a new session" and create your project.

So, what exactly does creating a Project in RStudio do for you? In a nutshell, using these Projects allows you to drop what you're doing, close RStudio, and then open the Project to pick up where you left off. Your data, history, settings, open tabs, etc. will be saved for you automatically.

Does using a RStudio Project allow someone else to pick up your code and just use it? Or let you come back to a Project 1 year later and have everything work magically? Not by itself, but with a few more tricks you will be able to more easily re-run or share your code.

1.2.2 Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory

Before we broke up with Excel, it was standard operating procedure to perform our calculations and data manipulations in the same place that our data lived. This is not necessarily incompatible with reproducibility, if we have very careful workflows and make creative use of macros. However, once you have modified your original input file, it may be non-trivial to review what you actually did to your original raw data (particularly if you did not save it as a separate file). Morever, Excel generally lends itself to non-repeatable manual data manipulation that can take extensive detective work to piece together.

Using R alone will not necessarily save you from these patterns but they take a different form. Instead of clicking around, dragging, and entering formulas, you might find yourself throwing different functions at your data in a different order each time you open up R. While it takes some effort to overwrite your original data file in R, other non-ideal patterns of file management that are common in Excel-land can creep up on you if you're not careful.

One solution to help avoid these issues in maintaining the separation of church and state (if I may use a poor analogy) is to explicitly organize your analysis so that raw data lives in one directory (the *data* directory) and the results of running your R code are placed in another directory (eg. *results* or *output*). You can take this concept a little further and include other directories within your project folder to better organize work such as *figures*, *documents* (for manuscripts), or *processed_data/munge* (if you want to create intermediate data sets). You have a lot of flexibility and there are multiple resources that provide some guidance (Parzakonis 2017), (Muller 2017), (Software Carpentry 2016).

Exercise 3

Let's go ahead and create a minimal project structure:

```r
dir.create("data") # raw data
dir.create("output") # output from analysis
dir.create("cache") # intermediate data (after processing raw data)
dir.create("src") # code goes into this folder
```

This is a bare bones structure that should work for our purposes.

Further exploration/tools for creating projects: There is also a dedicated Project Template package that has a nice "minimal project layout" that can be a good starting point if you want R to do more of the work for you: Project Template. This package duplicates some functionality that the RStudio Project does for you, so you probably want to run it outside of an RStudio Project but it is a good tool to be aware of.

1.2.3 Name all files (and variables) to reflect their content or function

This concept is pretty straightforward: assume someone else will be working with your code and analysis and won't intuitively understand cryptic names. Rather than output such as results.csv, a file name of morphine_precision_results.csv offers more insight. Wilson et al. make the good point that using sequential numbers will come back to bite you as your project evolves: for example, "figure_2.txt" for a manuscript may eventually become "figure_3.txt". We'll get into it in the next section but the final guidance with regards to file names is to using a style convention for file naming to make it easier to read names an manipulate files in R. One common issue is dealing with whitespace in file names: this can be annoying when writing out the file names in scripts so underscores are preferrable. Another issue is the use of capital letters: all lowercase names is easier to write out. As an example, rather than "Opiate Analysis.csv", the preferred name might be "opiate_analysis.csv".

1.3 Adopt a style convention for coding

Reading other people's code can be extremely difficult. Actually, reading your own code is often difficult, particularly if you haven't laid eyes on it long time and are trying to reconstruct what you did. One thing that can help is to adopt certain conventions around how your code looks, and style guides are handy resources to help with this. Google has published an R Style Guide that has been a long-standing resource and nice to refer to, but since we are immersing ourselves in the tidyverse, we will recommend the Tidyverse style guide.

Some highlights: - Use underscores to separate words in a name (see above comments for file names) - Put a space before and after operators (such as `==`, `+`, `<-`), but there are a few exceptions such as `^` or `:` - Use `<-` rather than `=` for assignment - Try to limit code to 80 characters per line & if a function call is too long, separate arguments to use one line each for function, arguements, and closing parenthesis.

```r
# Good
do_something_very_complicated(
  something = "that",
  requires = many,
  arguments = "some of which may be long"
)

# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
                              )
```

While we're talking about style conventions, let's take a little diversion to discuss a common element of code in the tidyverse that you may not be familiar with: the almighty pipe `%>%`. The pipe allows you to chain together functions sequentially so that you can be much more efficient with your code and make it readable. Here is an example (with imaginary functions) adapted from the tidyverse style guide:

```
# one way to represent a hop, scoop, and a bop, without pipes
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)
# another way to represent the same sequence with less code but in a less readable way
foo_foo <- bop(scoop(hop(foo_foo, through = forest), up = field_mice), on = head)

# a hop, scoop, and a bop with the almight pipes
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

Pipes are not compatible with all functions but should work with all of the tidyverse package functions (the magrittr package that defines the pipe is included in the tidyverse). In general, functions expect data as the primary argument and you can think of the pipe as feeding the data to the function. From the perspective of coding style, the most useful suggestion for using pipes is arguably to write the code so that each function is on its own line. The tidyverse style guide section on pipes is pretty helpful.

You're not alone in your efforts to write readable code: there's an app for that! Actually, there are packages for that, and multiple packages at that. We will not discuss in too much depth here but it is good to be aware of them: - styler is a package that allows you to interactively reformat a chunk of code, a file, or a directory - lintr checks code in an automated fashion while you are writing it

So, if you have some old scripts you want to make more readable, you can unleash styler on the file(s) and it will reformat it. Functionality for lintr has been built into more recent versions of RStudio.

1.4 Enforce reproducibility of the directories and packages

1.4.1 Scenario 1: Sharing your project with a colleague

Let's think about a happy time a couple months from now. You've completed this R course, have learned some new tricks, and you have written an analysis of your mass spec data, bundled as a nice project in a directory named "mass_spec_analysis". You're very proud of the analysis you've written and your colleague wants to run the analysis on similar data. You send them your analysis project (the whole directory) and when they run it they immediately get the following error when trying to load the data file with the `read.csv("file.csv")` command: `Error in file(file, "rt") : cannot open the connection` `In addition: Warning message: In file(file, "rt") : cannot open file 'file.csv': No such file or directory`

Hmmm, R can't find the file, even though you set the working directory for your folder using `setwd("/Users/username/path/to/mass_spec_analysis")`.

What is the problem? Setting your working directory is actually the problem here, because it is almost guaranteed that the path to a directory on your computer does not match the path to the directory on another computer. That path may not even work on your own computer a couple years from now!

Fear not, there is a package for that! Let's install the here package to help solve this problem and explore how it works.

Exercise 4

1. Install the package
2. Load the package. What does the output tell you?
3. Run the command `here()`. What does the output tell you?

```r
install.packages("here")
library(here)
here()
```

The here package uses a pretty straightforward syntax to help you point to the file you want. In the example above, where file.csv is a data file in the root directory (I know, not ideal practice per our discussion on project structure above), then you can reference the file using `here("file.csv")`, where `here()` indicates the current directory. So reading the file could be accomplished with `read.csv(here("file.csv"))` and it could be run by any who you share the project with.

The here package couples well with an RStudio Project because there is an algorithm that determines which directory is the top-level directory by looking for specific files - creating an RStudio Project creates an .Rproj file that tells here which is the project top-level directory - if you don't create a Project in RStudio, you can create an empty file named .here in the top-level directory to tell here where to go - there are a variety of other file types the package looks for (including a .git file which is generated if you have a project on Github)

I encourage you to read the following post by Jenny Bryan that includes her strong opinions about setting your working directory: Project-oriented workflow.

Moral of the story: avoid using `setwd()` and complicated paths to your file - use `here()` instead!

1.4.2 Scenario 2: Running your 2018 code in 2019

Now imagine you've written a nice analysis for your mass spec data but let it sit on the shelf for 6 months or a year. In the meantime, you've updated R and your packages multiple times. You rerun your analysis on the same old data set and either (a) one or more lines of code longer works or (b) the output of your analysis is different than the first time you ran it. Very often these problems arise because one or more of the packages you use in your code have been updated since the first time you ran your analysis. Sometimes package updates change the input or output specific functions expect or produce or alter the behavior of packages in unexpected ways. These problems also arise when sharing code with colleagues because different users may have different versions of packages loaded.

Don't worry, there is a package for that! Probably the most lightweight solution to this problem is the checkpoint package. The basic premise behind checkpoint is that it allows you use the package as it existed at a specific date. There is a snapshot for all packages in CRAN (the R package repository) each day, dating back to 2017-09-17. By using checkpoint you can be confident that the version of the package you reference in your code is the same version that anyone else running your code will be using.

Let's run through some basics of using the package.

Exercise 5

1. Install the checkpoint package

```r
install.packages("checkpoint")
```

2. Load the checkpoint package
3. Read the help for the checkpoint function

```r
library(checkpoint)
?checkpoint

# below is sample code but we're not quite ready to run it (yet)
# library(checkpoint)
# checkpoint("2018-01-01", checkpointLocation = tempdir())
# checkpoint function takes the date of the package you want as the primary argument
# and a location for the directory to load the packages into
#
```

```r
# library(dplyr)
# library(lubridate)
```

The behavior of checkpoint makes it complicated to test out in this section: the package is tied to a project and by default searches for every package called within your project (via `library()` or `require()`). However, if you refer to the setup code chunks for this course you will see how checkpoint works in the wild.

The checkpoint package is very helpful in writing reproducible analyses, but there are some limitations/considerations with using it: - retrieving and installing packages adds to the amount of time it takes to run your analysis - package updates over time may fix bugs so changes in output may be more accurate - checkpoint is tied to projects, so alternate structures that don't use projects may not able to utilize the package

In the next lesson we will dive into more detail about how to use checkpoint in the context of an R Markdown document, so this is just a brief introduction to the concept of using checkpoint to make your code reproducible.

1.5 Summary

- Reproducible research is the principle that any research result can be reproduced by anybody
- Practices in reproducible research also offer benefits for to the code author in producing clearer, easier to understand code and being able to easily repeat past work
- Important practices in reproducible research include:
  - Developing a standardized but easy-to-use project structure
  - Adopting a style convention for coding
  - Enforcing reproducibility when working with projects and packages

2 Lesson 2: Getting cozy with R Markdown

2.1 Why integrate your analysis and documentation in one place?

The short answer is that it will be easier for you to understand what you did and easier for anyone else to understand what you did when you analyzed your data. This aligns nicely with the principles of reproducible research and is arguably just as important for any analysis that occurs in a clinical laboratory for operational or test validation purposes. The analysis and the explanation of the analysis live in one place so if you or someone else signs off on the work, what was done is very clear.

The more philosophical answer to this question lies in the principles of literate programming, where code is written to align with the programmer's flow of thinking. This is expected to produce better code because the program is considering and writing out logic while they are writing the code. So the advantages lie in both communication of code to others, and that communication is expected to produce better programming (analysis of data in our case).

There is another advantage of using this framework with the tools we discuss below: the output that you generate from your analysis can be very flexible. You can choose to show others the code you ran for the analysis or you can show them only text, figures, and tables. You can produce a webpage, a pdf, a Word document, or even a set of slides from the same analysis or chunks of code.

2.2 Basics of knitr and rmarkdown

The theme of the course so far is "there's a package for that!" and this of course is no exception. The knitr package and closely related rmarkdown package were built to make it easier for users to generate reports with integrated R code. The package documentation is very detailed but the good news is that RStudio inherently utilizes knitr and rmarkdown to “knit” documents and allows for a simple, streamlined workflow to create these documents.

There are 3 components of a typical R Markdown document:

- header
- text
- code chunks

2.2.1 Header

The header includes metadata about the document that can help populate useful information such as title and author. This information is included in a YAML (originally *Yet Another Markup Language*, now *YAML Ain't Markup Language*) format that is pretty easy to read. For example, the header for this document is:

```
---
title: 'Lesson 2: Getting cozy with R Markdown'
author: "Patrick Mathias"
output: html_document
---
```

The output field dictates the output once the document is knit, and users can add other data such as the date or even parameters for a report.

2.2.2 Text

Text is written in whitespace sections using R Markdown syntax, which is a variant of a simple formatting language called markdown that makes it easy to format text using a plain text syntax. For example, asterisks can be used to *italicize* (`*italicize*`) or **bold** (`**bold**`) text and hyphens can be used to create bullet points: - point 1 - point 2 - point 3

```
- point 1
- point 2
- point 3
```

2.2.3 Code chunks

Interspersed within your text you can integrate “chunks” of R code, and each code chunk can be named. You can supply certain parameters to instruct R what to do with each code chunk. The formatting used to separate a code chunk from text uses a rarely utilized character called the backtick ‘ that typically can be found on the very top left of your keyboard. The formatting for a code chunk includes 3 backticks to open or close a chunk and curly brackets with the opening backticks to supply information about the chunk. Here is the general formatting, including the backticks and the curly braces that indicate the code should be evaluated in R:

~~~
```r
mean(c(10,20,30))
```
~~~

And this is how the code chunk looks by default:

```
mean(c(10,20,30))
```

There are shortcuts for adding chunks rather than typing out backticks: the `Insert` button near the top right of your script window or the `Ctrl+Alt+i/Command+Option+i`(Windows/Mac) shortcut.

In addition code can be integrated within text by using a single backtick to open and close the integrated code, and listing "r" at the beginning of the code (to indicate the language to be evaluated): 20.

2.3 Flexibility in reporting: types of knitr output

Under the hood, the knitting functionality in RStudio takes advantage of a universal document coverter called Pandoc that has considerable flexibility in producing different types of output. The 3 most common output formats are .html, .pdf, and Microsoft Word .docx, but there is additional flexibility in the document formatting. For example, rather than creating a pdf or html file in a typical text report format, you can create slides for a presentation.

There is additional functionality in RStudio that allows you to create an R Notebook, which is a useful variant of an R Markdown document. Traditionally you might put together an R Markdown document, with all its glorious text + code, and then knit the entire document to produce some output. The R Notebook is a special execution mode that allows you to run individual code chunks separately and interactively. This allows you to rapidly interact with your code and see the output without having to run all the code in the entire document. As with inserting a chunk, there are multiple options for running a chunk: the `Run` button near the top right of your script window or the `Ctrl+Shift+Enter/Command+Shift+Enter` (Windows/Mac) shortcut. Within a code chunk, if you just want to run an individual line of code, the `Ctrl+Enter/Command+Enter` (Windows/Mac) shortcut while run only the line your cursor is currently on.

Exercise 1

Let's use the built-in functionality in RStudio to create an R Markdown document.

1. Add a file by selecting the add file button on the top left of your screen
2. Select R Markdown… as the file type
3. Title the document "Sample R Markdown Document" and select OK
4. Put the cursor in the "cars" code chunk (should be the 2nd chunk) and hit `Ctrl+Shift+Enter/Command+Shift+Enter`. What happened?
5. Insert a code chunk under the cars code chunk by using the `Ctrl+Alt+i/Command+Option+i`(Windows/Mac) shortcut
6. Create output for the first lines of the cars data frame using the `head(cars)` command and execute the code chunk

RStudio sets up the document to be run as an R Notebook so you can interactively run chunks separately and immediately view the output.

RStudio also already provides you with an outline of a useful document, including interspersed code chunks. The header is completed based on the data that was entered into the document creation wizard. The first code chunk below the header is a useful practice to adopt: use your first code chunk as a setup chunk to set output options and load packages you will use in the rest of the document. The `knitr::opts_chunk$set(echo = TRUE)` command in the setup chunk tells R to display (or echo) the source code you write in your output document. A detailed list of various options can be found under the R Markdown cheatsheet here: https://www.rstudio.com/resources/cheatsheets/.

Now let's knit this file and create some output.

Exercise 2

1. Click the **Knit** button
2. You are being prompted to save the .Rmd file. Choose the "src" folder of your project and name the file sample_markdown_document
3. RStudio should produce output in .html format and display
4. Click the Open in Browser window and the same output should open in your default internet browser
5. If you find the folder you saved the .Rmd file there should also be a .html file you can open as well

6. Now, instead of hitting the **Knit** button, select the down arrow adjacent to it and click Knit to PDF
7. Repeat the previous step but knit to a Word document

The add file options also allow you to create a presentation in R Markdown. This can be a handy alternative to Powerpoint, especially if you want to share code and/or many figures within a presentation. You can find more information about these presentations and the syntax used to set up slides at the RStudio site on Authoring R Presentations.

2.4 Creating an R Markdown document that integrates best practices in reproducible research

Let's review the components of a project according to the take home points from Lesson 1: - a directory for each project - a folder structure the separates raw data, output, documents, etc. - clear file naming - code written with a style convention - code portability/reproducibility with help from here and checkpoint packages

How can we integrate R Markdown documents within the context of these requirements? We already created a project and associated directories for this course so most of our work is done. Now we are going to add our R Markdown file to the project and set it up so we can be efficient and reproducible.

Exercise 3

1. Close the RStudio window that has your "sample-project-structure" project open (refer to the upper right of your RStudio window to see which active project is open).
2. Find your RStudio window with your "MSACL-intermediate-R-course" project (look in upper right). If you cannot find it, within RStudio, click the arrow to the right of the project name (it will say Project: (None) if you are not in a project) and select your "MSACL-intermediate-R-course" project.
3. From within the course directory ("MSACL-intermediate-R-course"), create a new directory called "coursework" & set your working directory to there
4. Load the here package from within your console and use the `set_here()` command to set your "coursework" directory to be the primary point of reference for your project. (This creates an empty file named ".here", which is the first thing the here package looks for when trying to set its point of reference.)
5. Restart your R session by navigating to the Session menu and selecting Restart R. This is required to reinitialize the here package and point it to the .here file you just created in your "coursework" directory
6. Create the directory structure we discussed in the last lesson, **with the exception of the data directory**
7. Go to your operating system and copy the "class_data" directory you unzipped prior to the course (per the pre-course instructions) into your "coursework" directory
8. At this point we will want to include our individual lesson .Rmd files in the "src" directory. For now, copy your lesson3.Rmd from the lesson3 folder into your "src" directory under coursework.

These steps have set up your directory structure for future lessons. We have pre-made lesson files for future lessons, but it is also helpful to create an independent R Markdown file for any additional code you might want to write outside of the lesson. Plus, we want you to get some experience building your own R Markdown file within your project environment from scratch.

Exercise 4

1. Before we jump into creating our R Markdown file, we need to create a file that will allow us to run checkpoint together with knitr - click the Add File button and select "R Script"
2. Add the following code to your R script (minus the code chunk delimiters)

```
library("formatR")
library("htmltools")
library("caTools")
library("bitops")
library("base64enc")
library("rprojroot")
library("rmarkdown")
library("evaluate")
library("stringi")
```

3. Save as “knitr-packages.R” in the src folder of your project (explanation of why you did this below)
4. Now add a new R Markdown file, title it “Data Science 201”, and leave as the default .html ouptut selection
5. Go ahead and save the .Rmd by clicking the Save button (disk button) or selecing Save under the File menu: save the file to your src folder within your project. This will be an extra place to write code outside of the lesson files we provide. Name according to the principles we disucssed in lesson 1 (think of it as a scratchpad).
6. In the setup code chunk for the document, load the checkpoint package
7. Add the checkpoint command to load packages from today: `checkpoint("2018-01-21")`
8. Load the following packages in the setup chunk: tidyverse, readxl, lubridate, janitor, lattice, magrittr, modelr, xml2, jsonlite
9. Save the file and then go ahead and run your setup chunk

So why did you have to go through the first 3 steps before starting your R Markdown document? When RStudio knits a document, it calls certain packages behind the scenes to create the document. However, when you are using checkpoint, the package only looks for packages within your project folders and does not see those hidden RStudio folders. So adding knitr-packages.R is a workaround to allow checkpoint to see those packages and knit the document.

The R Markdown document you created will be your scratchpad if you want to experiment with differnt functions or explore data in our dataset outside of the context of the structured lessons.

Now we’re ready to jump into lesson 3!

2.5 Summary

- Integrating code and documentation in one place produces clearer, more reproducible code
- RStudio provides useful built-in functionality for “knitting” documents into a variety of output formats
- R Markdown documents can be integrated within a recommended project structure to create a reproducible analysis

3 Lesson 3: Reading files - beyond the basics

This is a much shorter and less philosophical lesson than the previous lessons but hopefully is very useful when considering how to pull data into R.

3.1 Base functions for reading and writing files

3.1.1 Reading files

R has solid built-in functions for importing data from files with the `read.table()` family of functions. `read.table()` is the generic form that expects a filename (in quotes) at a minimum and, importantly, an indication of the separator character used - it defaults to “” which indicates white space (one or more spaces, tabs, newlines, or carriage returns). The default header parameter for `read.table()` is FALSE, meaning that the function will **not** use the first row to determine column names. Because non-Excel tabular files are generally comma-delimited or tab-delimited with a first row header, `read.csv()` and `read.delim()` are the go-to base file reading functions that include a `header = TRUE` parameter and use comma and tab delimting, respectively, by default.

There are a variety of other useful parameters to consider, including explicitly supplying the column names via the `col.names` parameter (if not defined in header, for example). One related group of parameters to be conscious of with these functions are `stringsAsFactors` and `colClasses`. When R is reading a file, it will convert each column to a specific data type based on the content within that column. The default behavior of R is to convert columns with non-numeric data into a factor, which are a representation of categorical variables. For example, you may want to separate out data by sex (M/F) or between three instruments A, B, and C, and it makes perfect sense to represent these as a factor, so that you can easily stratify the groups during analyses in R, particularly for modeling questions. So, by default, with these base functions `stringsAsFactors = TRUE`, which means that any columns with characters may not have the expected behavior when you analyze the data. In general this may not be a big deal but can cause problems in a couple scenarios: 1. You are expecting a column to be a string to parse the data (using the stringr package for example). Not a huge deal - you can convert to a character 2. There are typos or other data irregularities that cause R to interpret the column as a character and then automatically convert to a factor. If you are not careful and attempt to convert this column back to a numeric type (using `as.numeric()` for example), you can end up coverting the column to a completely different set of numbers! That is because factors are represented as integers within R, and using a function like `as.numeric()` will convert the value to its backend factor integer representation. So `c(20, 4, 32, 5)` could become `c(1, 2, 3, 4)` and you may not realize it.

Problem #2 will come back to haunt you if you are not careful. The brute force defense mechanism is to escape the default behavior: `read.csv("file_name.csv", stringsAsFactors = FALSE)`. This will prevent R from converting any columns with characters into factors. However, you may want some of your columns to be represented as factors. You can modify behavior on a column by column basis. `read.csv("file_name.csv", colClasses = c("character", "factor", "integer")` will set a 3 column csv file to character, factor, and integer data types in that column order.

To be safe, the best practice is arguably to explicitly define column types when you read in a file. It is a little extra work up front but can save you some pain later on.

For the curious, additional information about the history of of stringsAsFactors can be found here.

> Exercise 1
>
> Let’s run through the base reading function with a csv. 1. Use the base `read.csv()` function to read the “2017-01-06.csv” file into a data frame. Recall how to use the `here()` function - it is using the “coursework” directory as the point of reference 1. What is the internal structure of the object? (Recall the `str()` command to quickly view the structure.) 1. What does the data look like? (Recall the `summary()` function to view column types and characteristics about the data.)

```r
base_load <- read.csv(here::here("class_data", "2017-01-06.csv"))
# note that we added the package name to the front of the function
# why? unfortunately lubridate has a deprecated here function
# so we have to tell R which here to use
# str(base_load)
summary(base_load)
```

Repeat the previous steps starting with #2, but include the argument `stringsAsFactors = FALSE` when you read in the data.

```r
base_load_nofactors <- read.csv(here::here("class_data", "2017-01-06.csv"),
                                stringsAsFactors = FALSE)
# str(base_load_nofactors)
summary(base_load_nofactors)
```

For this data set, which fields should be strings and which should be factors?

3.1.2 Writing files

The functions for reading files in base R have equivalents for writing files as well: `write.table()` and `write.csv()`. The first argument in these functions is the data frame or matrix to be written and the second argument is the file name (in quotes).

```r
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

There are a few other important parameters: - `sep` indicates the field separator (“ˆ” for tab) - `row.names` is set to TRUE by default - in general this makes for an ugly output file becuase the first column shows the row number (I almost always set this to FALSE) - `na` indicates the string to use for missing data and is set to R’s standard of “NA” by default - `append` can be set to TRUE if you would like to append your data frame/matrix to an existing file

3.2 Speeding things up with the *readr* package

Base R functions get the job done, but they have some weaknesses: - they are slow for reading large files (slow compared to?) - the automatic conversion of strings to factors by default can be annoying to turn off - output with row names by default can be annoying to turn off

One package in the tidyverse family meant to address these issues is readr. This package provides functions similar to the base R file reading functions, with very similar function names: `read_csv()` (instead of `read.csv()`) or `read_delim()` for example. Tab-delimited files can be read in with `read_tsv()`. These functions are ~10x faster at reading in files than the base R functions and do not automatically convert strings to factors. Readr functions also provide a helpful syntax for explicitly defining column types:

```r
# purely a dummy example, not executable!
imaginary_data_frame <- read_csv(
  "imaginary_file.csv",
  col_types = cols(
    x = col_integer(),
    y = col_character(),
    z = col_datetime()
  )
)
```

Another advantage of these functions is that they actually explicitly tell you how the columns were parsed when you import (as we’ll see in the exercise).

Readr also offers equivalent write functions such as `write_csv()` and `write_tsv()`. There is a variant of `write_csv()` specifically for csv files intended to be read with Excel: `write_excel_csv()`. These functions do not write row names by default.

Exercise 2

Now let's run through using the readr function for a csv: 1. Use the `read_csv()` function to read the "2017-01-06.csv" file into a data frame. 1. What is the internal structure of the object? (Recall the `str()` command to quickly view the structure.) 1. What does the data look like? (Recall the `summary()` function to view column types and characteristics about the data.)

```
readr_load <- read_csv(here::here("class_data", "2017-01-06.csv"))
str(readr_load)
summary(readr_load)
```

Now compare the time required to run the base `read.csv()` function with the readr `read_csv()` function using `system.time()`.

Time to read with base:

```
system.time(base_load <- read.csv(here::here("class_data", "2017-01-06.csv")))
```

Time to read with readr:

```
system.time(readr_load <- read_csv(here::here("class_data", "2017-01-06.csv")))
```

Finally, let's follow some best practices and explicitly define columns with the `col_types` argument. We want to explicitly define compoundName and sampleType as factors. Note that the `col_factor()` expects a definition of the factor levels but you can get around this by supplying a `NULL`. Then run a summary to review the data

```
readr_load_factors <- read_csv(
  here::here("class_data", "2017-01-06.csv"),
  col_types = cols(
    compoundName = col_factor(NULL),
    sampleType = col_factor(NULL)
    )
  )
summary(readr_load_factors)
```

3.3 Dealing with Excel files (gracefully)

You may have broken up with Excel, but unfortunately many of your colleagues have not. You may be using a little Excel on the side. (Don't worry, we don't judge!) So Excel files will continue to be a part of your life. The readxl package makes it easy to read in data from these files and also offers additional useful functionality. As with the other file reading functions, the syntax is pretty straightforward: `read_excel("file_name.xlsx")`. Excel files have an added layer of complexity in that one file may have multiple worksheets, so the `sheet = "worksheet_name"` argument can be added to specify the desired worksheet. Different portions of the spreadsheet can be read using the `range` arugment. For example a subset of rows and columns can be selected via cell coordinates: `read_excel("file_name.xlsx", range = "B1:D6")` or `read_excel("file_name.xlsx, range = cell_cols("A:F")`.

If you are dealing with Excel data that is not a traditional tabular format, the tidyxl package is useful to be aware of. We will not cover it in this course but it is worth reading up on if you ever have to analyze a pivot table or some other product of an Excel analysis.

Exercise 3

You might be able to guess what comes next: we'll read in an Excel file. 1. Use the `read_excel()` function to read the "2017-01-06.xlsx" file into a data frame 1. View a summary of the imported data 1. Now read in only the first 6 columns using the `range` parameter 1. Review the first 6 lines of the imported data

```
readxl_load <- read_excel(here::here("class_data", "2017-01-06.xlsx"))
summary(readxl_load)
readxl_load_subset <- read_excel(here::here("class_data", "2017-01-06.xlsx"), range = cell_cols("A:F"))
head(readxl_load_subset)
```

3.4 Importing dirty data

To close out the discussion on reading files, there is one more useful package to introduce that helps with a variety of data cleaning functions. Since this is R, the package is cleverly and appropriately named janitor. The quick take home in terms of useful functions from this package: - `clean_names()` will reformat column names to conform to the tidyverse style guide: spaces are replaced with underscores & uppercase letters are converted to lowercase - empty rows and columns are removed with `remove_empty_rows()` or `remove_empty_columns()` - `tabyl(variable)` will tabulate into a data frame based on 1-3 variables supplied to it

Let's take these functions for a spin using our data set. We are going to use the development version of the package because there is new, additional functionality. I will chain the commands together with pipes (which we'll discuss in more detail in the next lesson).

First let's review the first few lines of data after cleaning the columns names:

```
install.packages("janitor")
library(janitor)
readr_load_cleaned <- readr_load %>%
  clean_names()
head(readr_load_cleaned)
```

Now we'll do a quick tabulation to count the number of rows:

```
readr_load_cleaned %>% tabyl(compoundname)
```

3.5 Summary

- The base R functions for reading files `read.delim()`, `read.csv()`, etc. are useful tools but it is important to recognize how they handle strings (and the dangers in automatic conversion to factors)
- readr functions such as `read_delim()` or `read_csv()` are faster than base R functions and do not automatically convert strings to factors
- The readxl function `read_excel()` reads Excel files and offers functionality in specifying worksheets or subsets of the spreadsheet
- The janitor package can help with cleaning up irregularly structured input files

4 Lesson 4: Data manipulation in the tidyverse

4.1 A brief diversion to discuss the tidyverse

According to the official tidyverse website, "the tidyverse is an *opinionated* collection of R packages designed for data science." We've gotten a flavor of tidyverse functionality by using the readr packages and will wade deeper into the tidyverse in the next lessons. Because the tidyverse was not a component of the introductory MSACL data science course in previous years, we are going to cover basic functionality of many tidyverse packages throughout the rest of the course. Many of the data manipulation concepts will probably be familiar but the tidyverse offers a consistent interface for functions. Data is consistently the first argument for functions, and that enables compatibility with pipes. The tidyverse includes its own version of a data frame, the tibble, with the primary advantages being nicer printing of output and more predictable behavior with subsetting.

One of the key concepts of the tidyverse philosophy is maintaing "tidy" data. Tidy data is a data structure and a way of thinking about data that not only facilitates using tidyverse packages but more importantly it also provides a convention for organizing data that is amenable to data manipulation. The three criteria for tidy data are: 1. Each variable must have its own column. 2. Each observation must have its own row. 3. Each value must have its own cell.

As an example straight out of the R for Data Science text, consider a data set displaying 4 variables: country, year, population, and cases. One representation might split cases and population on different rows, even though each observation is a country and year:

```
table2
```

Or case and population may be jammed together in one column:

```
table3
```

The tidy representation is:

```
table1
```

Each observation is on one row, and each column represents a variable, with no values being shoved together into a single column.

An advantage of using the tidyverse packages is the relatively robust support documentation around these packages. Stack Overflow is often a go to for troubleshooting but many tidyverse packages have nice vignettes and other online resources to help orient you to how the package functions work. There is a freely available online book, R for Data Science that covers the tidyverse (and more). Cheat Sheets provided by RStudio also provide great quick references for tidyverse and other packages.

You can load the core tidyverse packages by loading tidyverse: `library(tidyverse)`. ggplot2 is probably the most popular tidyverse package and arguably the go to for sophisticated visualizations in R, but inevitably data will need to be manipulated prior to plotting. So the two workhorse packages for many applications are dplyr and tidyr, which we will cover in this lesson.

4.2 Manipulating data with dplyr

The dplyr package provides functions to carve, expand, and collapse a data frame (or tibble).

4.2.1 Carving your data set

Reducing a data set to a subset of columns and/or rows are common operations, particularly on the path to answering a specific set of questions about a data set. If you need to go from a large number of columns (variables) to a smaller set, `select()` allows you to select specific columns by name. If you need only a subset of rows from your data set, `filter()` allows you to pick rows (cases) based on values, ie. you can subset your data based on logic.

Let's take these for a spin using the data we started examining in the last lesson.

Review the type of data we were working with:

```
samples_jan <- read_csv(
  here::here("class_data", "2017-01-06.csv"),
  col_types = cols(
    compoundName = col_factor(NULL),
    sampleType = col_factor(NULL)
    )
  )
str(samples_jan)
```

Let's say we don't need the last two logical columns and want to get rid of them. We can use `select()` and provide a range of adjacent variables:

```
samples_jan %>%
  select(batchName:expectedConcentration)
```

Or we only care about the first 3 variables plus the concentration:

```
samples_jan %>%
  select(batchName:compoundName, concentration)
```

Now let's carve the data set in the other direction. If we only care about the morphine data, we can use `filter()` to pick those rows based on a logical condition:

```
samples_jan %>%
  filter(compoundName == "morphine")
```

Or maybe we want to examine only the unknown samples with a concentration greater than 0:

```
samples_jan %>%
  filter(sampleType == "unknown", concentration > 0)
```

Note that a comma in the filter state implies a logical AND - condition A and condition B. You could include an OR condition as well using the pipe character | - condition A | condition B.

```
samples_jan %>%
  filter(sampleType == "unknown" | concentration > 0)
```

> Exercise 1
>
> Carve the January data set in both directions. We want sample information (batch, sample, compound) and ion ratio data for only oxycodone measurements in unknown sample types with a concentration > 0. Provide a summary of the data.

```
samples_jan_oxy_ir <- samples_jan %>%
  filter(sampleType == "unknown", concentration > 0) %>%
  select(batchName:compoundName, ionRatio)
summary(samples_jan_oxy_ir)
```

4.2.2 Expanding your data set

Another common data manipulation task is adding or replacing columns that are derived from data in other columns. The `mutate()` function provides a quick and clean way to add additional variables that can include calculations, evaluating some logic, string manipulation, etc. You provide the function with the following argument(s): name of the new column = value. For example, if we continue with our January sample data set that includes concentrations and expected concentrations for standards, we can calculate the ratio of concentration to expected:

```
samples_jan %>%
  filter(sampleType == "standard", expectedConcentration > 0) %>%
  mutate(conc_ratio = concentration/expectedConcentration) %>%
  select(batchName:compoundName, concentration, expectedConcentration, conc_ratio)
```

Notice that we got around the issue of dividing by 0 by filtering for expected concentrations above 0. However, you may want to include these yet don't want R to throw an error. How can you deal with edge cases like this? `mutate()` borrows from SQL (Structured Query Language) and offers a `case_when` syntax for dealing with different cases. The syntax takes some getting used to but this can be helpful when you want to classify or reclassify values based on some criteria. Let's do the same calculation but spell out the case when expected_concentration is 0 and add a small number to numerator and denominator in that case:

```
samples_jan %>%
  filter(sampleType == "standard") %>%
  mutate(
    conc_ratio = case_when(
      expectedConcentration == 0 ~
        (concentration + 0.001)/(expectedConcentration + 0.001),
      TRUE ~ concentration/expectedConcentration
    )
  ) %>%
  select(batchName:compoundName, concentration, expectedConcentration, conc_ratio)
```

Another common operation manipulation is wrangling dates. The lubridate package offers a helpful toolset to quickly parse dates and times. The bread and butter parsing functions are named intuitively based on the order of year, month, date, and time elements. For example, `mdy("1/20/2018")` will convert the string into a date that R can use. There are other useful functions like `month()` and `wday()` that pull out a single element of the date to use for grouping operations, for example. Let's work with a different January data set that has batch data and parse the collection dates in a variety of ways:

```
batch_jan <- read_excel(here::here("class_data", "2017-01-06.xlsx"))
batch_jan_timestamps <- batch_jan %>%
  mutate(
    collect_datetime = ymd_hms(batchCollectedTimestamp),
    collect_month = month(batchCollectedTimestamp),
    collect_day_of_week = wday(batchCollectedTimestamp),
    collect_week = week(batchCollectedTimestamp),
    collect_week_alt = floor_date(collect_datetime, unit = "week")
    # floor_date to use datetime format but group to first day of week
  )
summary(batch_jan_timestamps)
```

You can see from the above example that these functions provide a great deal of flexibility in associating a row with arbitrary time scales. This allows the ability to group items by time and calculate summary data, which we will discuss in the next section.

> Exercise 2
>
> How long an average does it take to review each batch? Using the January batch data in "2017-01-06.xlsx", convert the review start timestamp and review complete timestamp fields into variables with a datetime type, then generate a new field the calculates the duration of the review in minutes. There are multiple approaches to this, but the `difftime()` function may be the most transparent - read the help on this function. The data will need to be collapsed by batch (which I do for you using the `distinct()` function) and display the min, max, median, and mean review times.

```
batch_jan_reviews <- batch_jan %>%
  mutate(review_start_timestamp = ymd_hms(reviewStartTimestamp),
         review_complete_timestamp = ymd_hms(reviewCompleteTimestamp),
         review_duration =
           as.numeric(difftime(reviewCompleteTimestamp,
                               reviewStartTimestamp,
                               units = "mins")
                      )
  )
# note: the output of difftime is a time interval
# convert to numeric to avoid confusion downstream
reviews_jan_grouped <- batch_jan_reviews %>%
           distinct(batchName, review_duration)
min(reviews_jan_grouped$review_duration)
median(reviews_jan_grouped$review_duration)
mean(reviews_jan_grouped$review_duration)
max(reviews_jan_grouped$review_duration)
```

4.2.3 Collapse (summarize) your data set

Carving and expanding your data are helpful but they are relatively simple. Often you will need to do more sophisticated analyses such as calculating statistical measures for multiple subsets of data. Grouping data by a variable using the `group_by()` function is critical tool provided by dplyr and naturally couples with its summary function `summarize()`. By grouping data you can apply a function within individual groups and calculate things like mean or standard deviation. As an example, we may want to look at our January sample data set and look at some statistics for the ion ratios by compound for the unknown sample type with non-zero concentation.

```
samples_jan %>%
  filter(sampleType == "unknown", concentration > 0) %>%
  group_by(compoundName) %>%
  summarize(median_ir = median(ionRatio),
            mean_ir = mean(ionRatio),
            std_dev_ir = sd(ionRatio))
```

We may want to look at this on the batch level, which only requires adding another variable to the `group_by()` function.

```
samples_jan %>%
  filter(sample_type == "unknown", concentration > 0) %>%
  group_by(batchName, compoundName) %>%
  summarize(median_ir = median(ionRatio),
            mean_ir = mean(ionRatio),
            std_dev_ir = sd(ionRatio))
```

Let's revisit our batch dataset with timestamps that we have parsed by time period (eg. month or week) and look at correlation coefficient statistics by instrument, compound, and week:

```
batch_jan_timestamps %>%
  group_by(instrumentName, compoundName, collect_week) %>%
  summarize(median_cor = median(calibrationR2),
            mean_cor = mean(calibrationR2),
            min_cor = min(calibrationR2),
            max_cor = max(calibrationR2))
```

A relatively new package that provides nice grouping functionality based on times is called tibbletime. This package provides similar functionality to the mutating and summarizing we did with times above but has a cleaner syntax for some operations and more functionality.

> Exercise 3
>
> From the January sample dataset, for samples with unknown sample type, what is the minimum, median, mean, and maximum concentration for each compound by batch? What is the mean of the within-batch means by compound?

```
sample_stats_jan <- samples_jan %>%
  filter(sampleType == "unknown") %>%
  group_by(batchName, compoundName) %>%
  summarize(min_conc = min(concentration),
            median_conc = median(concentration),
            mean_conc = mean(concentration),
            max_conc = max(concentration)
            )
head(sample_stats_jan)
sample_means_jan <- sample_stats_jan %>%
  group_by(compoundName) %>%
  summarize(overall_mean = mean(mean_conc))
sample_means_jan
```

4.3 Shaping and tidying data with tidyr

Data in the real world are not always tidy. Consider a variant of the January sample data we've reviewed previously in the "2017-01-06-messy.csv" file.

```
samples_jan_messy <- read_csv(here::here("class_data", "messy",
                                         "2017-01-06-sample-messy.csv"))
head(samples_jan_messy)
```

This certainly isn't impossible to work with, but there are some challenges with not having separate observations on each row. Arguably the biggest challenges revolve around built-in tidyverse functionality, with grouping and plotting as the most prominent issues you might encounter. Luckily the tidyr package can help reshape your data. The `gather()` function can take a list of columns as arguments, the key argument to name the variable you are gathering, and the value argument to name the new column with the values you extract from the old column.

```
samples_jan_tidy <- samples_jan_messy %>%
  gather("morphine", "hydromorphone", "oxymorphone", "codeine", "hydrocodone",
         "oxycodone", key = "compound_name", value = "concentration")
head(samples_jan_tidy)
```

The syntax takes some getting used to, so it's important to remember that you are taking column names and shoving those into rows, so you have name that variable (the key), and you are also putting values across multiple columns into one column, whose variable also needs to be named (the value).

Sometimes other people want your data and they prefer non-tidy data. Sometimes you need messy data for quick visualization purposes. Or sometimes you have data that is actually non-tidy not because multiple observations are on one row, but because a single observation is split up between rows when it could be on one row. It is not too difficult to perform the opposite operation of `gather()` using the `spread()` function. You specify the key, which is the variable than needs to be used to generate multiple new columns, as well as the value, which takes the specifies the variable that will need to populate those new columns. Let's do the opposite on the data set we just gathered:

```
samples_jan_remessy <- samples_jan_tidy %>%
  spread(key = "compound_name", value = "concentration")
head(samples_jan_remessy)
```

There are other useful tidyr functions such as `separate()` and `unite()` to split one column into multiple columns or combine multiple columns into one column, respectively. These are pretty straightforward to pick up so can be an independent exercise if you are intereted.

Exercise 4

The “2017-01-06-batch-messy.csv” file in the messy subdirectory of class_data is related to the “2017-01-06.xlsx” batch file you have worked with before. Unfortunately, it is not set up to have a single observation per row. There are two problems that need to be solved: 1. Each parameter in a batch is represented with a distinct column per compound, but all compounds appear on the same row. Each compound represents a distinct observation, so these should appear on their own rows. 2. There are 3 parameters per obsevation (compound) - calibration slope, intercept, and R^2. However these appear on different lines. All 3 paramters need to appear on the same row. After solving these problems, each row should contain a single compound with all three parameters appearing on that single row. Use `gather()` and `spread()` to tidy this data.

```r
batch_jan_messy <- read_csv(here::here("class_data", "2017-01-06-batch-messy.csv"))
batch_jan_gathered <- batch_jan_messy %>%
  gather("codeine", "hydrocodone", "hydromorphone", "morphine", "oxycodone",
         "oxymorphone", key = "compound_name", value = "value")
batch_jan_tidy <- batch_jan_gathered %>%
  spread(key = "parameter", value = "value")
```

4.4 Summary

- The dplyr package offers a number of useful functions for manipulating data sets
  - `select()` subsets columns by name and `filter()` subset rows by condition
  - `mutate()` adds additional columns, typically with calculations or logic based on other columns
  - `group_by()` and `summarize()` allow grouping by one or more variables and performing calculations within the group
- Manipulating dates and times with the lubridate package can make grouping by time periods easier
- The tidyr package provides functions to tidy and untidy data

5 Lesson 5: Blending data from multiple files and sources

5.1 Joining Relational Data

The database example for this class has three different data.frames (we use data.frame interchangably with the word tibble, even though we usually mean tibble) : one for batch-level information (calibration R^2, instrument name); one for sample-level information (sample type, calculated concentration); and one for peak-level information (quant peak area, modification flag). Accessing the relationships across these three sources – reporting the quant and qual peak area of only the qc samples, for example – requires the tools of relational data. In the tidyverse, these tools are part of the **dplyr** package and involve three ‘families of verbs’ called *mutating joins*, *filtering joins*, and *set operations*, which in turn expect a unique key in order to correctly correlate the data. To begin, read in the batch, sample, and peak data from the month of January. For simplicity, we will reduce size of our working examples to only those rows of data associated with one of two batches.

```r
jan.b <- read_excel(here::here("class_data","2017-01-06.xlsx"))
jan.s <- read_csv(here::here("class_data","2017-01-06.csv"))
jan.p <- read_tsv(here::here("class_data","2017-01-06.txt"))
byBatch <- jan.b[jan.b$batchName=="b802253" | jan.b$batchName=="b252474",]
bySample <- jan.s[jan.s$batchName=="b802253" | jan.s$batchName=="b252474",]
byPeak <- jan.p[jan.p$batchName=="b802253" | jan.p$batchName=="b252474",]
```

Let’s first talk about two simple commands in R which bind data, and the pitfalls of relying too heavily upon them.

5.2 Blending Data

5.2.1 Simple *rbind* and *cbind*

Sometimes, you need data stored across more than one file. For example, managing the QC deviations across twelve separate months of reports. Rather than hold multiple printouts side-by-side to monitor a trend, you can read each file and knit them together either by row, or by column. Working with tidy data means a 'wide' data.frame where each column is a specific variable, and each row is a specific measurment. If you know that your data sources have the same format, you can safely combine them with an `rbind` to append the second source of data at the end of the first.

```
january <- read_csv(here::here("class_data","2017-01-06.csv"))
as.data.frame(january[187195:187200,])
february <- read_csv(here::here("class_data","2017-02-06.csv"))
as.data.frame(february[1:5,])
twoMonths <- rbind(january,february)
```

Notice the continuation from the last rows of `january` to the first rows of `february` and that the number of rows in the combined data.frame `twoMonths` is the sum of the first two months of sample-level data.

```
twoMonths[187195:187204,]
c(nrow(january), nrow(february), nrow(twoMonths))
```

Because of how R treats columns, as long as the two files have the same number of columns and the same column names, the `rbind` command will correctly associate the data using the column order from the first variable. And if they aren't the same, you get an error that tells you what is wrong.

```
awkwardJanuary <- january[,c(10,1:9)]
missingJanuary <- january[,c(1:9)]
misnamedJanuary <- january
names(misnamedJanuary)[1] <- "bName"
```

> Exercise 1: Attempting an unsuccessful rbind
> Partially complete commands are commented out in the following code chunk. Perform an rbind between February and each of these three alternative January data.frames. Do any of them work? What does the data look like? What error messages do you get?

```
rbind(jan.p, february)
rbind(jan.b, february)
rbind(jan.s, february) # the only January data.frame with matching columns
```

There is an equivalent command called `cbind` which will append columns to a data.frame, but it is far more risky. The expectation within R that each row of information is correctly organized into a set of columns means this command will not check to make sure the order of values are correct between the two terms. Working to retain the correct association is the purpose of using keys as unique identifiers (see below).

```
incomplete_data <- tibble(sampleName="123456",
                          compoundName=c("morphine","hydromorphone",
                                         "codeine","hydrocodone"),
                          concentration=c(34,35,44,45))
additional_columns <- tibble(expectedConcentration=c(20,30,40,40),
                             sampleType="standard")
erronious_columns <- tibble(expectedConcentration=c(40,30,20,40),
                             sampleType="standard")
desired_cbind <- cbind(incomplete_data,additional_columns)
undesired_cbind <- cbind(incomplete_data,erronious_columns)
```

There are also dplyr functions with equivalent function: `bind_rows()` and `bind_cols()`.

There are better ways of adding per-column data. Creating a new named vector directly into the existing data.frame is straightforward and shown below, but a true merge of two distinct data.frames needs more explanation.

```
incomplete_data$batchName <- "batch01"
incomplete_data
```

5.2.2 Primary and foreign keys

A key is the variable in a data.frame – or combination of variables in a data.frame – that uniquely defines every row. In our data, `batchName` is present in each data.frame but always insufficient to define a specific row. In fact, no single column in our data operates as a key. We can build a key by combinging two (or three) columns.

```
byBatch$keyB <- paste(byBatch$batchName, byBatch$compoundName, sep=":")
bySample$keyS <- paste(bySample$sampleName, bySample$compoundName, sep=":")
byPeak$keyP <- paste(byPeak$sampleName, byPeak$compoundName,
                     byPeak$chromatogramName, sep=":")
```

Doing this creates a **primary key**, which is the unique identifier for that data.frame. A **foreign key** by contrast would uniqely identify an item in another table. The *left_join* command (described soon) does it's job correctly but because the compound names for the internal standards are not identical to the analyte compound names, this command incompletely adds the `byBatch` information to `byPeak`. The second command completes the population of `byPeak$keyB` as a foregin key, but only because the row order in `byPeak` follows a very specific format – exactly the problem we were worried about when using *cbind*! There are safer ways of populating this variable which take advantage of set operations (described next).

```
byPeak <- left_join(byPeak, byBatch)
byPeak$keyB[is.na(byPeak$keyB)] <- byPeak$keyB[!is.na(byPeak$keyB)] # dangerous!
```

Exercise 2: Join the batch and sample data using only the batch-specific key
Partially complete commands are commented out in the following code chunk. Since `keyB` is already built for one data.frame, creating this variable in `bySample` is the next step. How would you specify that only this variable which should be used for the join? Notice what that does for all of the variables in the joined data.frame.

```
byBatch$keyB
bySample$keyB <- paste(bySample$batchName, bySample$compoundName, sep=":")
exerciseTwo <- left_join(bySample, byBatch, by="keyB")
```

5.2.3 Set operations *union*, *intersect*, and *setdiff*

These three commands will return a vector which is the unduplicated combination of the two input vectors. `union(A,B)` includes all the values found in both A and B. `intersect(A,B)` returns only those values found in both A and B. `setdiff(A,B)` is order dependent, and returns the values of the first vector which are not also in the second vector.

```
A <- rep(seq(1, 10), 2)
B <- seq(2, 20, 2)
union(A, B)
intersect(A, B)
setdiff(A, B)
setdiff(B, A)
```

These commands are good for checking matches between two vectors, and we can use them to rebuild the `byPeak$keyB` foreign key without the risk of incorrect naming. First, let's show a couple of different ways of resetting `byPeak`. The first is familiar to people using **R** without the tidyverse, and we increase our sophistication in subsequent examples. Any of them work, but the last example is the most tidyverse way of doing things and makes it more readable to your colleagues who are also working in a tidy way.

```r
byPeak <- jan.p[jan.p$batchName=="b802253" | jan.p$batchName=="b252474",] # reset
byPeak$keyP <- paste(byPeak$sampleName, byPeak$compoundName,
                     byPeak$chromatogramName, sep=":")

byPeak <- jan.p %>%
  filter(batchName %in% c("b802253","b252474")) # reset the tidyverse way, but no keyP

byPeak <- unite(filter(jan.p,
                       batchName %in% c("b802253","b252474")),
                keyP,
                sampleName,
                compoundName,
                chromatogramName,
                sep=":",
                remove=FALSE) # reset and add the variable, without pipes

byPeak <- jan.p %>% # all of the above in one tidyverse pipe statement
  filter(batchName %in% c("b802253","b252474")) %>%
  unite(keyP, sampleName, compoundName, chromatogramName, sep=":", remove=FALSE)
```

Now let's construct our `byPeak$keyB` foreign key by creating and then using a new variable called *analyte*, taking advantage of set operations.

```r
allNames <- unique(byPeak$compoundName)
byPeak$analyte <- NA
for (name in allNames[1:6]) {
  compoundPairIdx <- grep(name, allNames)
  theCompound <- intersect(allNames[compoundPairIdx], name)
  theInternalStandard <- setdiff(allNames[compoundPairIdx], name)
  byPeak$analyte[byPeak$compoundName == theInternalStandard] <- theCompound
  byPeak$analyte[byPeak$compoundName == theCompound] <- theCompound
}
```

5.3 Mutating join to add columns

Mutating joins operate in much the same way as the set operations, but on data.frames instead of vectors, and with one critical difference: repeated values are retained. We took advantage of this earlier when using the left_join command, so that the `byBatch$keyB` got repeated for both the Quant and the Qual peak entries in `byPeak`. Having built the `byBatch` primary key, and correctly included it as a foreign key in `byPeak`, correctly joining them into a single data.frame is straightforward.

```r
byPeakWide <- left_join(byPeak,byBatch)
```

There are four kinds of mutating joins, differing in how the rows of the source data.frames are treated. In each case, the matching columns are identified automatically by column name and only one is kept, with row order remaining consistent with the principle (usually the left) source. All non-matching columns are returned, and which rows are returned depends on the type of join. An *inner_join(A,B)* only returns rows from A which have a column match in B. The *full_join(A,B)* returns every row of both A and B, using an NA in those columns which don't have a match. The *left_join(A,B)* returns every row of A, and either the matching value from B or an NA for columns with don't have a match. Finally, the *right_join(A,B)* returns every row of B, keeping the order of B, with either the matching value from columns in A or an NA for columns with no match.

Because these commands can duplicate rows, the potential for breaking things is pretty significant if the key isn't unique. Here are two examples, one where you do – and one where you do not – want that duplication:

```
goodDuplication <- inner_join(
  x = bySample[, c(1:3, 7)],
  y = byBatch[, c(1:6)],
  by = c("batchName", "compoundName")
)
badDuplication <- inner_join(
  x = bySample[, c(1:3, 7)],
  y = byBatch[, c(1:6)],
  by = c("compoundName")
)
```

5.4 Filtering join to check the overlap

We created the `byBatch$keyB` explicitly, but it was effectively present already thanks to the *batchName* and *compoundName* columns. The compound naming scheme in `byPeak` remains problematic since the internal standard isn't identified in `byBatch` or `bySample`, but we fixed this using a new column *analyte*. We could have discovered the problem, and then resolved it, using the semi_join and anti_join commands. The *semi_join(A,B)* returns all rows of A where there is a match from B, but keeps only the columns of A, and does not duplicate a row if there are multiple matches. The *anti_join(A,B)* is the inverse, returning all rows from A where there is no match from B. We still want to create the 'analyte' column for clarity, so one approach would be:

```
byBatch <- jan.b %>% # reset, no keyB
  filter(batchName %in% c("b802253","b252474"))
byPeak <- jan.p %>% # reset, no keyB or keyP
  filter(batchName %in% c("b802253","b252474")) %>%
  mutate(analyte=compoundName)

unique(byPeak$analyte) # notice the similar naming scheme?
byPeak$analyte <- sub("-.*$", "", byPeak$analyte) # notice how we used that similarity?

noMatch <- anti_join(byPeak,byBatch)
noMatch <- anti_join(byPeak,byBatch,by=c("batchName","analyte"="compoundName"))

justMatch <- semi_join(byPeak,byBatch,by=c("batchName","analyte"="compoundName"))
```

Exercise 3: Join the batch and peak data
Start from the reset data.frames built in the prior code chunk, so the `keyB` and `keyP` variables are not present. Partially complete commands are commented out in the following code chunk.

```
byPeak$analyte <- sub("-.*$", "", byPeak$analyte) # ensure the variable has been modified
exerciseThree <- left_join(byBatch, byPeak,by=c("batchName","compoundName"="analyte") )
```

5.5 Summary

- `rbind` and `cbind` add rows (or columns) to an existing data.frame
- `union`, `intersect`, and `setdiff` return a combination of two vectors

- Relational data merges two data.frames on the common columns, called keys
  - A primary key is a unique identifier for every row in a data.frame (the presence of `keyB` in `byBatch`)
  - A foreign key is a unique identifier for another data.frame (the presence of `keyB` in `byPeak`)
- `inner_join`, `full_join`, `left_join`, and `right_join` are mutating joins which add columns
- `semi_join` and `anti_join` are filtering joins which check for overlap

6 Lesson 6: Stronger visualizations with ggplot2 and lattice

6.1 Plotting Data With Default Graphics

Default R comes with several basic plotting commands – `plot` to draw an X,Y graph, `points` to add X,Y points to the current graph, `barplot` to draw vertical or horizontal bars, `boxplot` to draw box-and-whisker plots, `hist` to build and draw a histogram, and many other plot types or plot-specific additions to plots.

The first major drawback to using these plots is that each requires learning a slightly different syntax to decorate the graph. For example, here are three plots basd on the January sample data, showing the ion ratios for all compounds and samples which exhibit and quant and qual peak. The first is a simple series plot, changing the default plot color to blue.

```r
jan.s <- read_csv(here::here("class_data", "2017-01-06.csv"))
jan.s$idx <- c(1:nrow(jan.s))
hasIonRatio <- jan.s$ionRatio > 0
plot(jan.s$ionRatio[which(hasIonRatio)],col='blue')
```

If you want a histogram instead of a sequential series, the function changes but based on how `plot` looked, the coloring results may not be what you expected.

```r
hist(jan.s$ionRatio[which(hasIonRatio)],col='blue')
```

In order to plot the histogram with blue outline, to match the blue open circles of the first plot, you need to specify a different variable.

```r
hist(jan.s$ionRatio[which(hasIonRatio)],border='blue',main='Histogram')
```

The second drawback is that these plots, while drawn quickly, require detailed sort and select mechanisms in order to display complex data on a single graph. Plotting a matrix of graphs (as shown below) is even more difficult and you may spend more time troubleshooting the graph than actually analyzing the data. Here is a simple example which colors the series data by compound.

```r
compounds <- unique(jan.s$compoundName)
for (i in 1:length(compounds)) {
  if (i == 1) {
    plot(
      jan.s$ionRatio[hasIonRatio & jan.s$compoundName == compounds[i]],
      col = i,
      main = "color by compound"
    )
  } else {
    points(
      jan.s$ionRatio[hasIonRatio & jan.s$compoundName == compounds[i]],
      col = i
    )
  }
}
```

6.2 Plotting Data with *lattice*

Working with large datasets, especially data you want to slice by one or more variables, may require moving to another graphing package. Using **lattice** provides a simplified faceting functionality, with syntax more typical of the default graphics package. Instead of plotting every compound on top of each other, here we can show each compound in it's own graph and color by sample type.

```
xyplot(ionRatio ~ idx | compoundName,
       data=jan.s[hasIonRatio,],
       groups=sampleType,
       auto.key=TRUE)
```

Instead of showing the data as a set of individual data points, we can instead show it as a smoothed trace.

```
xyplot(ionRatio ~ idx | compoundName,
       data=jan.s[hasIonRatio,],
       groups=sampleType,
       auto.key=TRUE,
       type=c("l","spline"))
```

The default parameters don't look very smooth when using the default terms, so additional refinement of the spline variables would be required to see something beyond this jagged green line.

Finally, the command for drawing a histogram of the results, separated by both compound name and sample type.

```
histogram( ~ ionRatio | compoundName + sampleType,
           data=jan.s[hasIonRatio,])
```

6.2.1 Benchmarks for running time within R

Sometimes there is a need to optimize code for timing as well as readability, and that usually occurs when plotting is involved. Assuming you can't reduce the complexity with a filter (e.g. plotting only one compound, or across a smaller time scale) the choice to display 100k-10m datapoints will simply take the time it takes. Knowing how long to expect a process to complete is good for future users of your code, and requires benchmarking. Here we run into an awkward difficulty within R and Rstudio, where 'execute the command' and 'render the figure' are two different tasks. We'll discuss functions in Lesson 9, but for now just consider it a 'modular unit' of commands we can call which **R** will treat as a single command. When we call *system.time* on this command, it will report the userTime (the **R** session) and systemTime (the OS kernel). Notice how we don't get the returned result of our function unless we wrap it in a *print* statement.

```
system.time(print("Anything seen as a single function can be tested in this way"))

anyUserFunction <- function(someText) {
  hasNumber <- grepl("[:digit:]", someText)
  if (hasNumber) {
    editedText <- "including their pros and cons"
    unusedVariable <- rnorm(1e6)
  } else {
    editedText <- "have you looked ahead yet"
    unusedVariable <- rnorm(5e6)
  }
  return(editedText)
}
system.time(anyUserFunction("we will talk about user functions in lesson 9"))
system.time(print(anyUserFunction("we will talk about user functions in lesson 9")))
```

There are purpose-built packages that can make this measurement easier (the libraries **tictoc** and **microbenchmark** are popular) but the above methodology is usually sufficient.

6.3 Plotting Data With *ggplot2*

To maintain the tidy focus of the tidyverse, the **ggplot2** package keeps the same syntax for all graphing schemes, has arguably prettier default graphs, and a frankly intuitive means for layering/faceting of the underlying data. The main drawback is that plotting from a large data.frame is still measured in minutes. The mock data in this course definitely qualifies as a large dataset, so we recommend that plotting be used judiciously if you're not applying a filter (see below).

Syntax follows the format of {'define the data' {+ 'describe the visualization'}} where each description is called a *geom* and multiple geoms can be stacked together. Definitions for the aesthetic mappings (e.g. plotTerms, color, iconShape, lineType) can be supplied when defining the data and are applied to the subsequent stack of geoms. Any mappings can be overridden within an individual geom.

Our first two examples show the ggplot version of the per component plots previously done with lattice. Notice that defining the data can be done as a variable (here it is `g`) and that definition can be used later for any number of geoms.

```r
g <- jan.s %>%
  filter(ionRatio > 0) %>%
  ggplot(aes(x = idx, y = ionRatio, colour = sampleType))
g + geom_point() + facet_wrap(~compoundName) + scale_x_continuous(labels = NULL)
g + geom_smooth() + facet_wrap(~compoundName)
```

For the histogram, we override the aesthetic because this plot only uses 'one dimension' of the source data.

```r
g +
  geom_histogram(mapping=aes(x=ionRatio,colour=sampleType),inherit.aes=FALSE) +
  facet_wrap(~compoundName)
```

We could easily spend the whole class session on this package, but the above plots showcase the basic syntax. The cheatsheet downloadable from the link at the end of this lesson provides additional examples of what can be done.

Exercise 1: Draw a better histogram
The default histogram paramaters for ggplot will stack the sample types in the same bin, making it difficult to determine if the trend for qc and standard samples is the same as the unknowns. The first plot in this exercise makes adjacent bars, but what does the second plot do?

```r
g <- jan.s %>%
  filter(ionRatio > 0) %>%
  ggplot(aes(x = ionRatio, colour = sampleType, fill = sampleType))
g + geom_histogram(position='dodge', bins=30) + facet_wrap(~compoundName)
g + geom_histogram(aes(y=..density..), bins=30) + facet_grid(sampleType~compoundName)
```

Exercise 2: Plot timing
There is a longstanding community opinion that ggplot "takes longer" than the other two plotting mechanisms, but how much longer is it really? Is the time savings from **lattice** worth learning that syntax?

```r
oneYearSamples <- list.files(here::here("class_data"), pattern = "csv$") %>%
  file.path(here::here("class_data"), .) %>%
  map_dfr(read_csv)
oneYearSamples$idx <- 1:nrow(oneYearSamples)
coreR <- function(oneYearSamples) {
  sampleTypes <- unique(oneYearSamples$sampleType)
  for (i in 4:1) {
    oneType <- which(oneYearSamples$sampleType == sampleTypes[i])
    if (i == 4) {
      plot(oneYearSamples$idx[oneType], oneYearSamples$concentration[oneType], col = i)
    } else {
      points(oneYearSamples$idx[oneType], oneYearSamples$concentration[oneType], col = i)
    }
  }
}
g <- ggplot(oneYearSamples, aes(x = idx, y = concentration, color = sampleType)) + geom_point()
l <- xyplot(
  concentration ~ idx,
  data = oneYearSamples,
  groups = sampleType,
  auto.key = TRUE
)
# system.time(coreR(oneYearSamples))
# dev.off()
# system.time(print(g))
# dev.off()
# system.time(print(l))
```

6.4 Summary

- ggplot2 cheatsheat
- lattice overview
- download a PDF comparison of both packages
- download an older PDF showing the system.time comparison

7 Lesson 7: Modelling the data

7.1 Data Modelling

After enough measurements have been made to consider your data a dataset, the next step is exploring that dataset to find trends and outliers. Most often, this includes a graph as well as a computational model which fits the data to a trendline. Modelling reduces the complexity of individual observations, which is good. However, it can only operate within the limits of the model. The linear model shown below doesn't fit as well as the quadratic model, but deciding whether the quadratic model makes physical sense is not something the model can demonstrate.

```
# data taken from http://www.theanalysisfactor.com/r-tutorial-4/
y <- c(126.6, 101.8, 71.6, 101.6, 68.1, 62.9, 45.5, 41.9, 46.3, 34.1, 38.2, 41.7, 24.7,
       41.5, 36.6, 19.6, 22.8, 29.6, 23.5, 15.3, 13.4, 26.8, 9.8, 18.8, 25.9, 19.3)
x <- c(0, 1, 2, 4, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 24, 25, 26,
       27, 28, 29, 30)
x2 <- x ** 2
raw <- tibble(x, y, x2)
linearModel <- lm(y ~ x, data = raw)
quadraticModel <- lm(y ~ x + x2, data = raw)
plot(x, y)
```

```r
sequenceX <- seq(0, 30, by = 0.1)
modeldata <- data.frame(x = sequenceX, x2 = sequenceX ** 2)
sequenceYL <- predict(linearModel, modeldata)
sequenceYQ <- predict(quadraticModel, modeldata)
lines(sequenceX, sequenceYL, col = "blue", lwd = 2)
lines(sequenceX, sequenceYQ, col = "darkgreen", lwd = 2)
```

> Exercise 1: Rebuild this plot using ggplot
> Partially complete commands are commented out in the following code chunk. Start by looking at the data.frame `modeldata` before and after the *mutate* command, to see what changed. Then, use these new columns as values for the two geoms.

```r
modeldata <- modeldata %>%
  mutate(linear = sequenceYL, quadratic = sequenceYQ)

g <- ggplot(data = raw, aes(x = x, y = y)) + geom_point()
g +
  geom_line(data=modeldata, aes(x=x,y=linear), color='blue') +
  geom_line(data=modeldata, aes(x=x,y=quadratic), color='darkgreen')
```

7.1.1 Parsing dates and times with *lubridate*

The context for most of the data we'll encounter involve a time series, where each batch is collected at a specific date and that order of collection is a necessary component of the analysis. Dates within **R** are interpreted using the IEEE "POSIX" format, but reading data from an exterior source such as a CSV means this information arrives as a string. Functions in the **lubridate** package like `ymd` help to convert strings to POSIX, or combine information from multiple columns of a data.frame into a usable date format.

```r
stringOfDates <- c("2017-12-15", "2017-Dec-16", "2017-12-17", "2017-12-18",
                   "2017-12-19", "2017-12-20", "2017-12-21")
stringOfDates + 7
oneWeek <- ymd(stringOfDates)
oneWeek + 7
tableOfDates <- tibble(
  year = rep(2017, 8),
  month = c(rep(4, 3), rep(5, 5)),
  day = seq(1, 8),
  hour = rep(4, 8),
  minute = c(15, 29, 40, 45, 0, 58, 9, 11)
)
tableOfDates %$% make_date(year, month, day) # uses the magrittr 'not a pipe' operator
tableOfDates %$% make_datetime(year, month, day, hour, minute)
```

It helps to remember R uses "yyyy-mm-dd" as the default order for dates when viewing or exporting.

7.1.2 Symbolic formula notation

The difficulty in using R for scientific modelling is that the language was built for statistical modelling. This is most obvious in the Wilkinson-Rogers notation for formula design. It is a versitile and powerful way to quickly describe the interactions between variables, but lacks an intuitive way to express basic transformations (quadratic, logrithmic, etc)

Formula Notation	Common Understanding
y ~ x	y = *a0* + *a1*x
y ~ x - 1	y = *a1*x
y ~ x + z	y = *a0* + *a1*x + *a2*z
y ~ x * z	y = *a0* + *a1*x + *a2*z + *a3*xz
y ~ x + I(x^2)	y = *a0* + *a1*x + *a2*x^2
y ~ Gauss(amplitude,mu,sigma,x)	fit the data to a formula, defined elsewhere

Wilkinson-Rogers notation comes into it's own when looking for generalized interactions (and their relative importance) between the dependent variable and a collection of independent ones: the formula *y ~ Height * Weight * Age* would solve for eight coefficients, the linear combination for each of the three terms and the intercept. By comparison, the formula *y ~ Height + Weight + Age* would solve four coefficients, ignoring any interaction between Height, Weight, and Age. We'll draw a plot in the next section which will hopefully help this to make more sense.

7.2 Linear modelling

Most of time a linear model, or transforming the data into a linear model, is sufficient to establish the key relationships between variables. Let's start by expanding the `raw` data.frame with additional observations and two more descriptors for the data.

```
newraw <- raw %>%
  mutate(y = raw$y * rnorm(nrow(raw), mean = 1, sd = 0.3)) %>%
  rbind(raw) %>%
  arrange(x)
newraw$instrument <- c(rep("Coarse", 12), rep("Fine", 40))
newraw$date <- c("07-08-2017", "10-08-2017") %>%
  rep(times = 26) %>%
  dmy()
# this could also have been done on a single line without pipes:
# newraw$date <- dmy( rep(c("07-08-2017","10-08-2017"), times=26) )
```

Now we can explore the simple linear model in context of the instrument or the date of collection. Using the `summary` command on each linear model will provide basic statistical results assocaited with that model.

```
fullModel <- lm(y ~ x * date * instrument, data = newraw)
interactingTerms <- lm(y ~ x * instrument, data = newraw)
noninteractingTerms <- lm(y ~ x + instrument, data = newraw)
summary(interactingTerms)
```

We can now bulid the `grid` data.frame for plotting both the data and the models. Breaking down the pipe sequence of commands, we begin with our `newraw` data set and restructure it to work as an evenly spaced grid of points using *data_grid*, and then use *gather_predictions* to attach the prediction (y-value) at each obsevation point (x-value) from two of our linear models.

```
grid <- newraw %>%
  data_grid(x, date, instrument) %>%
  gather_predictions(interactingTerms, noninteractingTerms)
ggplot(newraw, aes(x, y, shape = factor(date))) +
  geom_point() +
  geom_line(data = grid, aes(y = pred, color = model)) +
  coord_cartesian(ylim = range(y)) +
  facet_wrap(~ instrument)
```

Exercise 2: Modelling with categorical variables
Revise `newraw` so that the last 40 observations include a third instrument called “Ultrafine,” and rerun the models. How does this affect the regression and your interpretation?

```
newraw$instrument[13:52] <- rep(c(rep("Fine", 2), rep("Ultrafine", 2)), 10)
interactingTerms <- lm(y ~ x * instrument, data=newraw)
noninteractingTerms <- lm(y ~ x + instrument, data=newraw)
grid <- newraw %>%
  data_grid(x,date,instrument) %>%
  gather_predictions(interactingTerms,noninteractingTerms)
ggplot(newraw, aes(x,y,shape=factor(date))) +
  geom_line(data=grid,aes(y=pred,color=model)) +
  geom_point() +
  coord_cartesian(ylim=range(y)) +
  facet_wrap(~ instrument)
summary(noninteractingTerms)
summary(interactingTerms)
```

7.3 Nonlinear modelling

Although not common when looking for relationships between variables, nonlinear models are a concern for other aspects of data evaluation (e.g. fitting a chromatogram to a Gaussian curve) or assay validation (e.g. instrument response as a function of concentration). Unlike linear modelling, which will find a single solution starting from the data itself, a nonlinear model must iterate from a starting guess for each variable. A well formed guess should converge on the correct value, but there is no general method for finding that guess.

Below are three nonlinear fits to noisy Gaussian data, each starting from a different initial guess of the parameters. Because the noise is randomly applied, your graph shouldn’t be an exact match to the plot made by anybody else. Not all the fits are going to look great!

```
Gauss <- function(amplitude, mu, sigma, x) {
  amplitude * exp(-0.5 * ((x - mu) / sigma) ^ 2)
}
x <- seq(1, 5, by = 0.1)
yExact <- Gauss(10000, 3, 0.4, x)
yNoisy <- yExact * rnorm(length(yExact), mean = 1, sd = 0.125)
noisyCurve <- tibble(x = x, y = yNoisy)
perfectGuess <- nls(
  y ~ Gauss(a, m, s, x),
  start = list(a = 10000, m = 3, s = 0.4),
  data = noisyCurve
)
okayGuess <- nls(
  y ~ Gauss(a, m, s, x),
  start = list(a = max(yNoisy), m = x[yNoisy == max(yNoisy)], s = 1),
  data = noisyCurve
)
badGuess <- nls(
  y ~ Gauss(a, m, s, x),
  start = list(a = 1, m = 1, s = 1),
  data = noisyCurve
)
forPlot <- noisyCurve %>%
  rename(yNoisy=y) %>%
  cbind(yExact) %>%
  gather(yType, y, -x)
grid <- forPlot %>%
  data_grid(x = seq_range(x, 100), y) %>%
  gather_predictions(perfectGuess, okayGuess)
ggplot(forPlot, aes(x, y)) +
  geom_point(aes(shape = yType)) +
  scale_shape_manual(values = c(3, 20), name = "data source") +
  geom_line(data = grid, aes(y = pred, color = model, linetype = model), size = 1)
summary(perfectGuess)
```

Understanding the reasoning for nonlinear model selection is a critical step before attempting that fit. The risk of non-convergence means the nonlinear model should be avoided, if possible.

7.4 Summary

- **lubridate** functions will convert a text string to a date
- statistical modelling with Wilkinson-Rogers notation builds possible interactions into the analysis
- scientific modelling with Wilkinson-Rogers notation requires careful understanding of terms
- linear models are strongly recommended whenever possible
  - this may require a transformation of the data
  - 'linear' doesn't mean 'straight line' when you draw the plot
- nonlinear models are dependant on a starting guess

8 Lesson 8: Beyond the csv – parsing xml and json files

8.1 Flat and Structured Files

The commands *read_csv*, *read_tsv*, and *read_excel* (discussed in Lesson 5) open files that are often called 'flat' because there is no inherent structure to the delimited rows and columns. Using the first row as a column header is a common convention, but is not required for the function to work properly. In contrast, most databases have a required structure to each element, and these elements sometimes have a nestest structure within structure. The details of database construction are well beyond the scope of this class, but if the source of your data is a database query, the returned results are typically in XML or JSON format. Being able to work with these files in **R** requires some basic parsing techniques.

8.2 The extensible markup language (xml)

When read as a text file, xml looks a lot like html, because html is (by and large) a subset of xml. That's not to say they're interchangable, since html permits some shortcuts which xml does not, and xml isn't restricted to web browser tags. The xml format declares and closes a *key*, with the *value* enclosed between them: `<key>value</key>`. That value can itself be one or more nested key-value statements.

Reading an xml file requires the **xml2** package (or one like it). Consider a short list of album information, where each entry contains the same number of keys, and those keys are always in the same order.

```
cd <- read_xml("https://www.w3schools.com/xml/cd_catalog.xml")
cd.list <- as_list(cd)
cd.rows <- length(cd.list)
cd.names <- names(cd.list[[1]])
cd.df <- data.frame(
  matrix(
    unlist(cd.list),
    nrow = cd.rows,
    byrow = TRUE,
    dimnames = list(NULL, cd.names)
  ),
  stringsAsFactors = FALSE
)
```

8.3 The javascript object notation (json)

A web browser can only send one thing, or retrieve one thing, at a time. The most complex online form has to be reduced to a single string, and then unpacked on the other side. Although that string could be sent as xml, virtually every website operates with json because this format is more lightweight than xml and thus takes less time to transmit. The format looks very different, but operates on the same principle of first naming a thing, and then giving it a value. With json, the key-value statements are separated by a colon, and each statement is separated by a comma: `{"key":value}`. Nested terms are explicitly either an array (ordered, and enclosed in square brackets) or an object (unordered, and enclosed in curly brackets).

Reading a json file requires the **jsonlite** package (or one like it). Consider an enumeartion of named colors and their RGB value.

```
colr <-
  fromJSON("https://raw.githubusercontent.com/corysimmons/colors.json/master/colors.json")
colr.rows <- length(colr)
colr.df <- data.frame(
  matrix(
    unlist(colr),
    nrow = colr.rows,
    byrow = TRUE,
    dimnames = list(NULL, c("R", "G", "B", "T"))
  ),
  stringsAsFactors = FALSE
)
colr.df$name <- names(colr)
```

8.4 Solving problems

The most common problem is that numbers, even if they were stored as numbers, can become text strings during conversion to a data.frame. Jumping ahead to the next lesson, the simplest fix will loop on the column of data that should be numeric.

```
cd.df$PRICE %<>% map_dbl(as.numeric) # also invokes a from-and-back-into pipe notation
```

Exercise 1: Renumbering Redux
Will reading the xml into a tibble (instead of a data.frame) solve this number recognition? First, try it. Then, even if it did work, construct a 'standard pipe' mechanism using `%>%` to change both *PRICE* and *YEAR* to double precision numbers.

```
##The simplest appraoch replaces data.frame with tibble
cd.tibble <- as_tibble( matrix(unlist(cd.list),
                               nrow=cd.rows,
                               byrow=TRUE,
                                dimnames=list(NULL,cd.names)) )
##Assuming that the list elements are always in the same order (not a guarantee) there
## is also a tidyverse approach that avoids building a matrix
cd.tibble <- cd.list %>%
  transpose() %>%
  map_dfr(unlist)
##Neither approach makes the 'PRICE' or 'YEAR' column into a numeric.
# Fixing them in one line
cd.tibble <- cd.tibble %>% mutate(PRICE=as.numeric(PRICE),YEAR=as.numeric(YEAR))
```

There are many more problems that could happen, ultimately because each entry in the xml or json file is an independent object. There may be an element not present in every entry (e.g. not including a price for a CD), or multiple elements within an element (e.g. two artists for a CD), or the elements may not be retrieved in the same order each time (e.g. all Bob Dylan albums return the artist before the title). All are disasterous; the conversion examples shown here using `matrix` will now be wildly offset.

Most problems should be solved on the query-side of the equation, to confirm the database query was built correctly and that the database is populated as expected. But, since "two artists for one CD" is at least possible and might even be common, the retrieved data could simply be less data.frame friendly than you would like. Unless you're very sure of what the returned data must look like, some basic checks now will avoid exhausting effort later.

The following code snippet, for example, will go through each list element of `cd.list` and count how many variables are present, and create a key showing their order. Then, at the end, we look for all the *unique* components of our `numberOfElements` and `namesForElements`. If we have more than one, it's a sign we need to correct our query or adjust our expectations.

```
numberOfElements <- c()
namesForElements <- c()
for (l in cd.list) {
  numberOfElements <- c(numberOfElements, length(l))
  namesForElements <- c(namesForElements, paste(names(l), collapse = ":"))
  for (e in l) {
    valuesInElement <- length(e)
    if (valuesInElement != 1) {
      break
    }
  }
  if (valuesInElement != 1) {
    break
  }
}
paste(unique(numberOfElements), unique(namesForElements), sep = " | ")
```

Exercise 2: Fixing a known misordered data.frame
Add this next line to `cd.list` and rerun the troubleshooting code section. What changed in the output from the *paste* command? How would you fix the list so the columns are correct while still using the *matrix* method described in Exercise 1?

```
cd.list[[27]] <- list(ARTIST=list("Benny Andersson","Tim Rice","Bjorn Ulvaeus"),
                      TITLE=list("Chess"),
                      COUNTRY=list("Sweden"),
                      COMPANY=list("RCA Victor"),
                      PRICE=list("19.95"),
                      YEAR=list("1984"))
##The new output should read
```

```r
## [1] "6 | TITLE:ARTIST:COUNTRY:COMPANY:PRICE:YEAR"
## "6 | ARTIST:TITLE:COUNTRY:COMPANY:PRICE:YEAR"
## which doesn't specify the line causing the problem, but does show there is more than
## one order in the cd.list
##There are two problems with this new entry. The first is how the order doesn't match the
## prior 26 rows in the list, which we can fix in place.
names(cd.list[[27]])
cd.list[[27]] <- cd.list[[27]][names(cd.list[[1]])]
names(cd.list[[27]])
##The second problem with this new entry is that 'ARTIST' is itself a list of three
## entries. One solution is to keep only the first name. Another (shown here) converts
## the list to a single character string, after which the matrix conversion method will
## work as expected.
cd.list[[27]]$ARTIST <- paste(cd.list[[27]]$ARTIST,collapse=", ")
cd.rows <- length(cd.list)
cd.names <- names(cd.list[[1]])
exerciseTwo <- data.frame(matrix(unlist(cd.list),
                                 nrow=cd.rows,
                                 byrow=TRUE,
                                 dimnames=list(NULL,cd.names)),
                          stringsAsFactors=FALSE)
```

Exercise 3: Preventing a misordered data.frame
Propose another way to convert the xml file so it automatically adjusts for inconsistent ordering of the keyword pairs.

```r
cd <- read_xml("https://www.w3schools.com/xml/cd_catalog.xml")
cd.list <- as_list(cd)
forcedNameOrder <- c("YEAR","TITLE","ARTIST","PRICE","COUNTRY","COMPANY")
cd.reorderedList <- map(cd.list,function(x){x[forcedNameOrder]})
exerciseThree <- cd.reorderedList %>%
  transpose() %>%
  map_dfr(unlist)
##This can be done in a single string of pipes
exerciseThree <- read_xml("https://www.w3schools.com/xml/cd_catalog.xml") %>%
  as_list() %>%
  map(function(x){x[forcedNameOrder]}) %>%
  transpose() %>%
  map_dfr(unlist)
```

8.5 Summary

- **xml2** for read/writing xml files
- **jsonlite** for read/writing json files
- troubleshoot the returned query result to catch
  - multiple values where a single was expected
  - the same key names are in every entry
  - the key names are always in the same order (or the conversion code robustly adjusts to handle this)
  - numbers stored as strings
  - strings stored as factors

9 Lesson 9: Scaling with purrr to loop through a vector

9.1 Looping Commands

Reducing duplication in your typed code makes it easier to see the purpose of the code, easier to change the calculation, and easier to remove bugs. The three typical methods in R to iterate with a specific piece of code are functions, for-loops, and vector transformations with the **purrr** package.

9.2 Looping with user functions

Every command in **R** is consistent with the function syntax *command(inputA,inputB,. . .)* and writing your own function is no different. Here, we create two functions that will return a median value across every column, or across every row, of a data.frame.

```
tableOfNumbers <- tibble(
  a = c(53, 1, 51, 23, 28, 12, 87, 0, 47),
  b = floor(100 * runif(9)),
  c = as.integer(100 * rnorm(9)),
  d = rep(5, 9),
  e = seq(1, 9)
)
medianByColumn <- function(tb) {
  recordOfValues <- c()
  for (i in 1:ncol(tb)) {
    byColumn <- unlist(tb[, i])
    recordOfValues %<>% append(median(byColumn))
  }
  return(recordOfValues)
}
medianByRow <- function(tb) {
  recordOfValues <- c()
  for (i in 1:nrow(tb)) {
    byRow <- unlist(tb[i, ])
    recordOfValues %<>% append(median(byRow))
  }
  return(recordOfValues)
}
```

Both of these functions use a for-loop to cycle through the `byColumn` or `byRow` vector of numbers. The logic is basically identical except for the placement of a single comma, which means a single function could be written to handle both tasks. Finally, the `recordOfValues` empty vector needs to be initalized before it can be used, which is slightly confusing. Each of these comments are addressed below.

9.3 Looping with *for* and *while*

These loops are explicitly sequential, showing the input and output of each command in exactly the same way as the primary code you work with. The *for* loop follows a three component format: create the output vector, define the looping sequence, and perform the body of work. In our example functions, that body is two lines long, but it could be as complex as necessary to produce the intended output. When working with long loops (100s to 1000s of iterations), the inefficiency of growing the output vector at each iteration will slow down the code. Instead, the output should be initialized as an empty vector of the expected length. Modifying our `medianByColumn` to resolve this inefficiency:

```r
medianByColumn <- function(tb) {
  recordOfValues <- vector("double", ncol(tb))
  for (i in 1:ncol(tb)) {
    byColumn <- unlist(tb[, i])
    recordOfValues[i] <- median(byColumn)
  }
  return(recordOfValues)
}
```

Exercise 1: Making a two-variable function that ‘switches’
Create a function `medianBy( )` that accepts both the data.frame and a logical value to indicate whether the result should be by row or column.

```r
medianBy <- function(tb,byRow=FALSE) {
  if(byRow) {l <- nrow(tb)} else {l <- ncol(tb)}
  recordOfValues <- vector("double", l)
  for(i in 1:l) {
    if(byRow) {v <- unlist(tb[i,])} else {v <- unlist(tb[,i])}
    recordOfValues[i] <- median(v)
  }
  return(recordOfValues)
}
```

The *while* loop is used when the work being performed should continue until a logical condition occurs. The obvious risk is that the condition never occurs, prompting the need for a failsafe. They are rarely used, but here are two examples.

```r
numberOfFlips <- 0
numberOfHeads <- 0
while (numberOfHeads < 3) { # risky
  if (sample(c("H", "T"), 1) == "H") {
    numberOfHeads <- numberOfHeads + 1
  }
  numberOfFlips <- numberOfFlips + 1
}

countedFlips <- 0
numberOfTails <- 0
while (numberOfTails < 3 | countedFlips == 1000) { # safer
  if (sample(c("H", "T"), 1) == "T") {
    numberOfTails <- numberOfTails + 1
  }
  countedFlips <- countedFlips + 1
}
```

9.4 Looping with *map*

These loops are explicitly vectored, performing the action on every element in that vector. The **purrr** package has five functions which return a vector for each type of output: list, logical, integer, double, and character. It comes with a significant increase in speed and further improved readability, but requires a different seeming workflow. Most notably, for-loops often have bookkeeping variables for the intermediate steps in the body of work, while `map` functions generate the output vector directly and are designed to work well with pipes.

The *map* structure builds on the idea that a solution to “how do I solve for a single set of data” is simpler to visualize. After breaking the problem down to small steps on one set, enclosing those steps in the function will iterate on each column of data in the data.frame to report the result. Revisiting our example functions to use *map_dbl* which will return a real (double precision) number:

```r
medianByColumn <- function(tb) {
  recordOfValues <- map_dbl(tb, median) %>% unname()
  return(recordOfValues)
}
medianByRow <- function(tb) {
  transposedTibble <- as_tibble(t(tb))
  recordOfValues <- map_dbl(transposedTibble, median) %>% unname()
  return(recordOfValues)
}
```

Exercise 2: Improve the `medianBy( )` function to use map looping.

```r
medianBy <- function(tb,byRow=FALSE) {
  if(byRow) {tb <- as_tibble(t(tb))}
  recordOfValues <- map_dbl(tb,median) %>% unname()
  return(recordOfValues)
}
```

9.5 Nested Looping

Having built one loop, a second (or more) iteration may still be necessary to do the work. This is usually done for simple counting variables like `i` and `j` as shown below.

```r
vectorList <- vector("list", 10)
for (i in 1:10) {
  oneVector <- vector("double", 20)
  for (j in 1:20) {
    oneVector[j] <- i * j
  }
  vectorList[[i]] <- oneVector
}
nestingTibble <- vectorList %>%
  set_names(map_chr(1:10, paste0, "column")) %>%
  bind_cols()
```

It can also be done using the map functions. In this example, *map_dfc* returns a data.frame created by column-binding each entry.

```r
betterNestingTibble <- c(1:10) %>%
  map_dfc( ~ tibble(x = . * 1:20) ) %>%
  set_names(map_chr(1:10, paste0, "column"))
```

Deciding when to use a function instead of a for-loop is often an aesthetic choice focused on the readability of the code. The `betterNestingTibble` may not be as clear as code that keeps for-loop(i) and uses the map command to replace the for-loop(j).

9.6 Summary

- `functions( )` capture regularly used segments of code so the main workflow can focus on the result instead of the process
- `for( ) { }` loops retain the serial nature of the workflow and avoid the need to copy/paste the iterations

- `map( )` functions are incredibly fast and powerful, and the preferred looping mechanism in R once the format is understood

10 Lesson 10: Bringing it all together from import to graph to result

10.1 From Import to Graph

These lessons were designed to build and reinforce each other. It's possible to write purely linear code, but **purrr** makes the code easier to read, and remember, and use again. Each csv can be read and examined in isolation, but **rbind** and **dplyr** make it possible to aggregate and discover trends. What follows are the steps to replicate the discovery of one particular problem in the mock data: excessively good R^2 data.

For this lesson, we're fudging slightly by already knowing what we want to start looking for.

```
oneYearBatches <- list.files(here::here("class_data"), pattern = "xlsx$") %>%
  file.path(here::here("class_data"), .) %>%
  map_dfr(read_excel) %>%
  as_tibble() %>%
  type_convert()

ggplot(oneYearBatches, aes(x = calibrationR2, color = compoundName,
                           fill = compoundName)) + geom_histogram(bins = 30)
ggplot(oneYearBatches, aes(x = batchCollectedTimestamp, y = calibrationR2,
                           color = compoundName)) + geom_line()
```

There's something interesting going on with the R^2 values in the month of May, where a large number of them report a value of 1.0 – a perfect fit. Let's focus on that month, and spread out the data so we can clarify whether it's all compounds or just oxymorphone (the magenta color on top).

```
mayPlot <- oneYearBatches %>%
  filter(batchCollectedTimestamp > ymd("2017-04-15"),
         batchCollectedTimestamp < ymd("2017-06-15")) %>%
  ggplot(aes(x = batchCollectedTimestamp, y = calibrationR2, color = compoundName))
mayPlot +
  geom_line() +
  facet_wrap(~ compoundName)
mayPlot +
  geom_point() +
  facet_grid(reviewerName ~ instrumentName)
```

Whatever is going on, it looks like reviewer 'Dave' is the only person it is happening to.

10.2 From Graph to Result

Based on the batch-level data, we can see that 'Dave' – and apparently only Dave – has perfect R^2 values on every batch of data he reviewed throughout the month of May. Digging deeper will require merging information from the batch level with information at the sample (and possibly peak) level.

```
oneYearSamples <- list.files(here::here("class_data"), pattern = "csv$") %>%
  file.path(here::here("class_data"), .) %>%
  map_dfr(read_csv)
davesData <- oneYearSamples %>%
  left_join(select(oneYearBatches, -calibrationSlope, -calibrationIntercept)) %>%
  filter(
    batchCollectedTimestamp > ymd("2017-04-20"),
    batchCollectedTimestamp < ymd("2017-06-10"),
    sampleType == "standard",
    reviewerName == "Dave"
  )
```

The following plots of `davesData` provide compelling evidence for what happened: Dave unselected the middle five calibrators in order to draw a straight line and maximize the R^2 term.

```
davesData %>%
  ggplot(aes(x = batchCollectedTimestamp, y = usedForCurve, color = compoundName)) +
  geom_point() +
  facet_grid(compoundName ~ expectedConcentration) +
  geom_vline(xintercept = as.numeric(as_datetime(c("2017-05-01", "2017-06-01"))),
    linetype = 1,
    colour = "black")

davesData %<>% mutate(
  pctDiff = (concentration - expectedConcentration) / expectedConcentration,
  within20 = abs(pctDiff) <= 0.2
)
davesData %>%
  filter(compoundName == "codeine") %>%
  ggplot(aes(x = batchCollectedTimestamp, y = pctDiff, color = within20)) +
  geom_point() +
  facet_wrap(~ expectedConcentration) +
  ggtitle("Codeine Only") +
  geom_vline(xintercept = as.numeric(as_datetime(c("2017-05-01", "2017-06-01"))),
    linetype = 1,
    colour = "black")
```

The second plot shows that calibrators were dropped regardless of whether they would be within 20% of the expected concentration, suggesting that they were dropped for some other reason. The data does not say why ‘Dave’ did this, but there are a couple of good guesses here which revolve around training.

We intentionally included several other issues within the database, which will require aggregation and plotting to discover.

Exercise : Revealing an ion ratio problem
Ion ratios can be particularly sensitive to instrument conditions, and variability is a significant problem in mass spec based assays which use qualifying ions. With the tools that have been demonstrated in this course, we can look for outlier spikes and stability trends, and separate them out across instruments, or compounds, or sample types. First, plot the ion ratio as a function of instrument name. What trends are most obvious, and what is the reason for each of them? What additional variables would help besides the ones captured in the sample data.frame?

```
# 1 #
oneYearSamples %>%
  left_join(oneYearBatches) %>%
  ggplot(aes(x = batchCollectedTimestamp, y = ionRatio, color = compoundName)) +
  geom_smooth() +
  facet_grid(compoundName ~ instrumentName) # doc is grossly out of step,
# investigate later
# Quants and quals got flipped
```

```r
# 2 #
oneYearSamples %>%
  left_join(oneYearBatches) %>%
  filter(instrumentName != "doc") %>%
  ggplot(aes(x = batchCollectedTimestamp, y = ionRatio, color = compoundName)) +
  geom_smooth() +
  facet_grid(compoundName ~ instrumentName) # grumpy+oxycodone looks least like the others

# 3 #
oneYearSamples %>%
  left_join(oneYearBatches) %>%
  filter(compoundName == "oxycodone" & instrumentName != "doc") %>%
  ggplot(aes(x = batchCollectedTimestamp, y = ionRatio, color = instrumentName)) +
  ggtitle("oxycodone") +
  geom_smooth() # grumpy+oxycodone clearly outlying

# 4 #
oneYearSamples %>%
  left_join(oneYearBatches) %>%
  filter(compoundName == "oxycodone" & instrumentName != "doc" & ionRatio > 0) %>%
  ggplot(aes(x = batchCollectedTimestamp, y = ionRatio, color = instrumentName)) +
  ggtitle("oxycodone") +
  geom_smooth() # ionRatio!=0 makes it even more clear

# 5 #
oneYearPeaks <- list.files(workingDir, pattern = "txt$") %>%
  file.path(workingDir, .) %>%
  map_dfr(read_tsv, col_types = cols())
meanByWeek <- oneYearPeaks %>%
  left_join(oneYearBatches) %>%
  filter(compoundName == "oxycodone" & instrumentName == "grumpy" & peakArea > 0) %>%
  mutate(week = week(batchCollectedTimestamp)) %>%
  group_by(week, chromatogramName) %>%
  summarise(mean = mean(peakArea), sd = sd(peakArea), n = n())
ggplot(meanByWeek, aes(x = week, y = mean, color = chromatogramName)) +
  geom_line() +
  geom_smooth() +
  ggtitle("oxycodone + grumpy") +
  facet_grid(chromatogramName ~ ., scales = "free_y") # quant is constant, qual drops

# 6 #
meanByWeek %>%
  mutate(sd = NULL, n = NULL) %>%
  spread(chromatogramName, mean) %>%
  mutate(ionRatio = quant / qual) %>%
  ggplot(aes(x = week, y = ionRatio)) +
  geom_line() +
  geom_smooth() +
  ggtitle("ionRatio by week for oxycodone on grumpy") # basically recreate step 4
```