MSACL Data Science 201 Textbook

Patrick Mathias   Shannon Haymond   Randall Julian   Adam Zabell

Contents

Adopting principles of reproducible research

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 reproducibility?

- 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 core practices we will cover in this lesson to help get your code to be more reproducible and reusable:

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

We will cover using a version control system, another practice, in the next lesson.

Adopt a style convention for coding

Style guides

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. We recommend the Tidyverse style guide as the style guide that much of the R community working with Tidyverse tools has converged to. The Tidyverse guide was originally derived from Google's R Style Guide, but since that time Google has updated their style guide to pull from the Tidyverse 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, arguments, and closing parenthesis.

```
# 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"
                              )
```

Packages supporting code style

You're not alone in your efforts to write readable code: there are multiple packages for that. We will not cover them in 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
  - styler can function as an Addin within RStudio (look above your markdown window for addins already installed in your RStudio)
  - You can highlight code, apply styler via the Addins menu, and code will automatically be formatted per the Tidyverse style guid
- formatr allows you to reformat whole files and directories
- lintr checks code and provides output on formatting issues

So, if you have some old scripts you want to make more readable, you can unleash styler or formatr on the file(s) and it will reformat it. Functionality for lintr has been built into more recent versions of RStudio - look at markers to the left of code chunks in the editor window.

Pipes

A common element convention supported by the tidyverse that you may not be familiar with or may not use consistently is 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. This is a nice intuitive example from Andrew Heiss, an educator at Georgia State University:

```
my_morning <- I %>%
  wake_up(time = "8:00") %>%
  get_out_of_bed(side = "correct") %>%
  get_dressed(pants = TRUE, shirt = TRUE) %>%
  leave_house(car = TRUE, bike = FALSE)
```

Contrast that with the same operations but using sequential steps without the pipes:

```
my_morning <- wake_up(I, time = "8:00")
my_morning <- get_out_of_bed(my_morning, side = "correct")
my_morning <- get_dressed(my_morning, pants = TRUE, shirt = TRUE)
my_morning <- leave_house(my_morning, car = TRUE, bike = FALSE)
```

This is readable but contains quite a bit of duplicative code.

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.

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.

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. We are going to use functions available in both base R as well as the fs package, which provides clearer names for functions as well as clearer output for directors and filenames. The fs package should have been installed if you completed the pre-course instructions, and you can load it if needed by running `library(fs)`.

Exercise 1

1. Navigate to "Global Options" under the Tools menu in 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. If you haven't already installed the fs package (from the pre-course instructions), do so now: `install.packages("fs")`. Then load the package with `library(fs)` if you did not already run the set up chunk above.
4. Review the contents of your current folder using `dir_ls()`. (Base equivalent: `list.files()`)
5. Now try to set your working directory using `setwd("test_dir")`. What happened?
6. Create a new test directory using `dir_create("test_dir")`. (Base equivalent: `dir.create("test_dir")`)
7. Review your current directory
8. Set your directory to the test directory you just created
9. 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*
10. Navigate back to one level above the directory you created using `setwd("..")` and list the files
11. Delete the directory you created using the `dir_delete()` function. Learn more about how to use the function by reviewing the documentation: `?dir_delete`. (Base equivalent: `unlink()` + additional arguments)

End Exercise

The functions in the fs package include arguments and capabilities that can be helpful for finding directories or files with names that have a specific pattern. From our project directory, we may want to see the files in a specific folder, without changing the directory of the folder. We can use the path argument in the function:

```r
dir_ls(path = "data")
```

```
## data/2017-01-06_b.csv          data/2017-01-06_p.csv
## data/2017-01-06_s.csv          data/2017-02-06_b.csv
## data/2017-02-06_p.csv          data/2017-02-06_s.csv
## data/2017-03-09_b.csv          data/2017-03-09_p.csv
## data/2017-03-09_s.csv          data/2017-04-08_b.csv
## data/2017-04-08_p.csv          data/2017-04-08_s.csv
## data/2017-05-09_b.csv          data/2017-05-09_p.csv
## data/2017-05-09_s.csv          data/2017-06-08_b.csv
## data/2017-06-08_p.csv          data/2017-06-08_s.csv
## data/2017-07-09_b.csv          data/2017-07-09_p.csv
## data/2017-07-09_s.csv          data/2017-08-09_b.csv
## data/2017-08-09_p.csv          data/2017-08-09_s.csv
## data/2017-09-08_b.csv          data/2017-09-08_p.csv
```

```
## data/2017-09-08_s.csv           data/2017-10-08_b.csv
## data/2017-10-08_p.csv           data/2017-10-08_s.csv
## data/2017-11-08_b.csv           data/2017-11-08_p.csv
## data/2017-11-08_s.csv           data/2017-12-08_b.csv
## data/2017-12-08_p.csv           data/2017-12-08_s.csv
## data/CKD_GFR.csv                data/CKD_data.csv
## data/CKD_stage.csv              data/messy
## data/method_validation_data.xlsx data/monthly_orders_data_set.xlsx
## data/order_details.csv          data/orders_data_set.xlsx
## data/project_data.sqlite
```

Another really handy argument to the `dir_ls` function is glob. This allows you to supply a "wild card" pattern to retrieve records fitting a specific pattern. The syntax is to use an asterisk to indicate any pattern, either at the beginning or end of an expression. For example, we may only want to retrieve the Excel files from our data directory, so we would match to a file extension:

```
dir_ls(path = "data", glob = "*.xlsx")
```

```
## data/method_validation_data.xlsx  data/monthly_orders_data_set.xlsx
## data/orders_data_set.xlsx
```

Or, we may be interested in only the sample csv files that are denoted by "_s":

```
dir_ls(path = "data", glob = "*_s.csv")
```

```
## data/2017-01-06_s.csv data/2017-02-06_s.csv data/2017-03-09_s.csv
## data/2017-04-08_s.csv data/2017-05-09_s.csv data/2017-06-08_s.csv
## data/2017-07-09_s.csv data/2017-08-09_s.csv data/2017-09-08_s.csv
## data/2017-10-08_s.csv data/2017-11-08_s.csv data/2017-12-08_s.csv
```

Note that the asterisk at the beginning of the pattern followed by characters to match against at the end requires that the text pattern be at the very end of the string.

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).

End Exercise

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 "msacl-201-project"
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.

End Exercise

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.

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 (to 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

Be sure to work within the RStudio window that contains your "msacl-201-project" project. Refer to the top right of the window and you should see the project name displayed there. Let's go ahead and create a minimal project structure by running the following code within the console:

```
library(fs)
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 future projects you create. Refer to the content below if you decide you want to adopt a standard directory structure for your projects on top of using RStudio Projects.

Keep this project open in a separate window for now. We will revisit it as we learn about version control.

End Exercise

Further exploration/tools for creating projects:

The directory creation code in the above exercise can be packaged into a function that creates the folder structure for you (either within or outside of a project). Software Carpentry has a nice refresher on writing functions: https://swcarpentry.github.io/r-novice-inflammation/02-func-R/.

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.

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".

Enforce reproducibility of the directories and packages

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! The here package is a helpful way to "anchor" your project to a directory without setting your working directory. 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!

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.

Why do we run into this problem? Packages are installed in directories called library paths. Whenever you load a package, R will search for the package in your library paths. You may have multiple library paths - by default there is typically a system library but you may have a user library as well. You can see your library paths with the `libPaths()` function. R will go through the library paths in order, so it may look for user packages before moving to system packages. Ultimately R will default to using the first version of a specific package it finds as it moves through the library paths. If you have a script or notebook that was developed with a different version of package than R defaults to, you can end up with different output than expected or an analysis that does not work. Many packages that are developed by the RStudio team like the tidyverse group of packages are more likely to have significant testing and avoid breaking changes. However, this is not a guarantee.

The generalized way to avoid this issue is to manage packages on a project by project basis: instead of R using the default version of a package it finds, tie specific versions of packages to a project. There are a couple options for accomplishing this.

Option 1: checkpoint Arguably 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.

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()`).

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

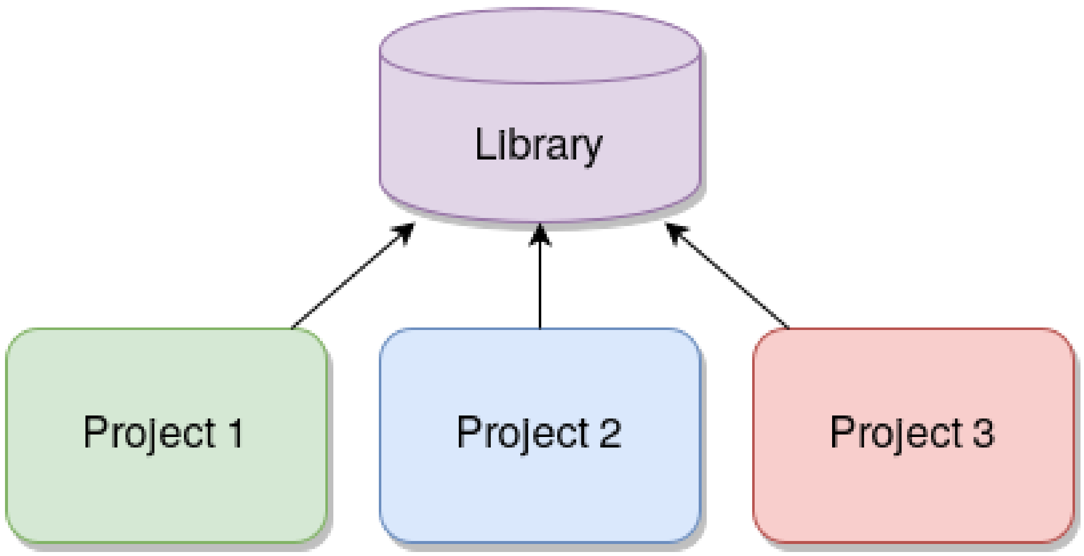


Figure 1: One library rules them all. Source: https://kevinushey-2020-rstudio-conf.netlify.app/slides.html

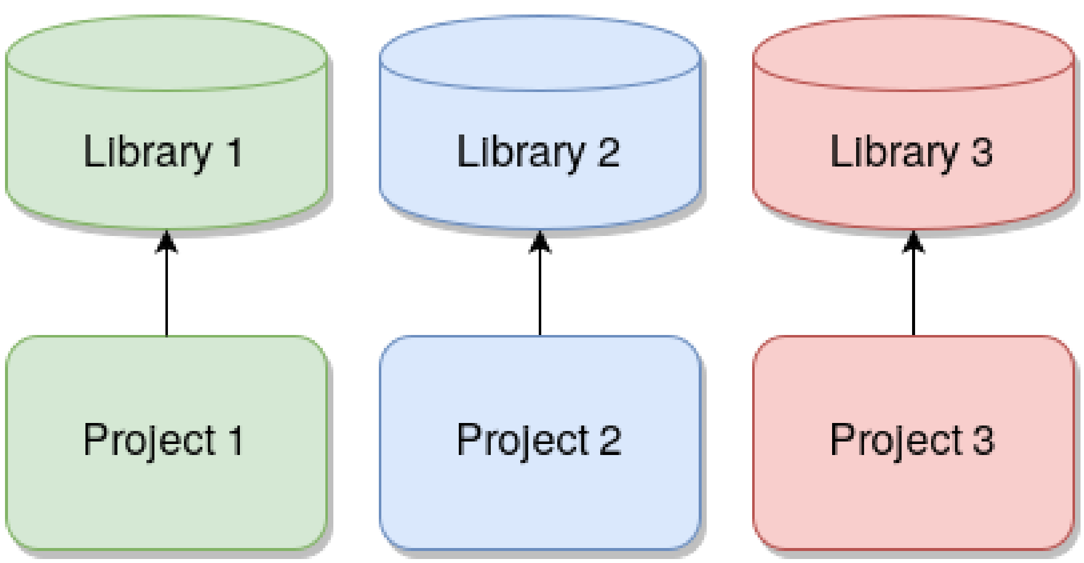


Figure 2: One library per package. Source: https://kevinushey-2020-rstudio-conf.netlify.app/slides.html

Option 2: renv There is another solution to this problem that has tighter integration with RStudio: the renv package. renv allows your to maintain specific package versions on a project by project level. Unlike checkpoint, package versions are not determined by date. Instead you can initiate a project and use renv functions to detect whatever packages you've loaded with their version information and generate a file with that data so that the environment can be replicated easy on another system or by someone else.

- The functionality can be initialized with the `renv::init()` function, which captures the state of your default R libraries for a project-local library which will then be used when future R sessions open the project
- Packages can be updated with most recent versions (if updated at the user or system level) with the `renv::snapshot()` function, which creates a lockfile containing the detailed package info
- The `renv::restore()` function is used to apply updates to packages in a project based on the lockfile data

There is a nice summary of renv here: https://kevinushey-2020-rstudio-conf.netlify.app/slides.html#1.

Note: the renv package was developed as a more stable solution than its predecessor packrat.

Either approach to package management will work - the important point here is to be proactive about how you manage your packages, especially if you know your code will be used over and over again in the future.

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

Version control

Use a version control system

The concept of capturing changes to a document by resaving the file with different names is well-intentioned and lines up with previous concepts of reproducibilty. This can help capture changes you've made in the evolution of a project. The problem with this method is that it is very clunky and, realistically, you will not be able to capture every single change you've made. When writing code, you often do want to capture changes at a higher resolution than when writing a paper or other text document.

The basic functionality of a version control system tracks changes (in addition to who made changes in collaborative settings) and makes it easier to undo changes. But you can go further with version control and implement it as a tool in collaboration workflows because it enables multiple people to work on changes to the same set of files at once.

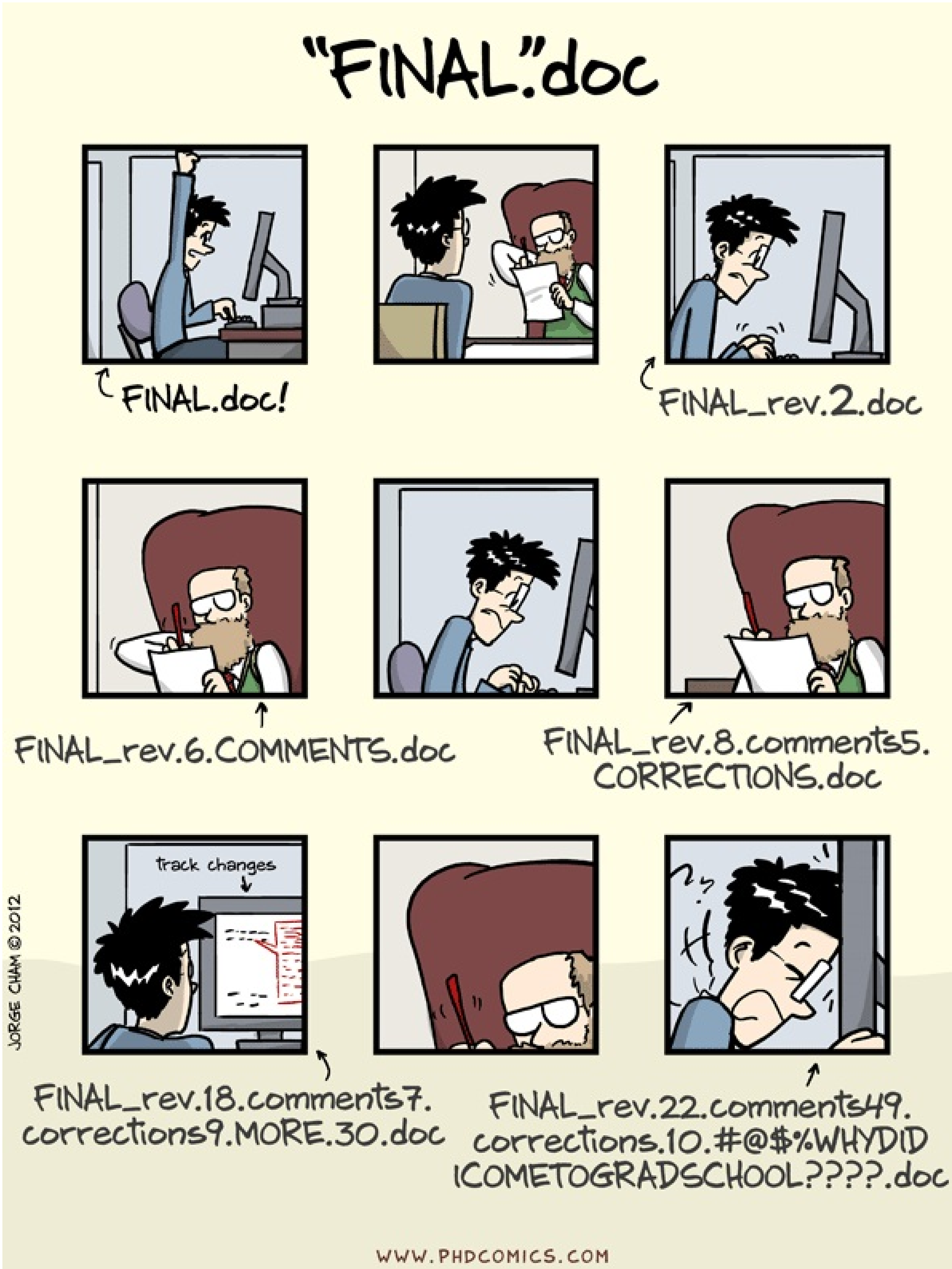


Figure 3: One of many justifications for using version control. Source: phdcomics.com

A brief intro to Git

This section is a high level summary of many concepts explained in Chapter 1 of the Pro Git textbook. There are other great resources to learn about using Git and using Git with RStudio, including http://happygitwithr.com, https://support.rstudio.com/hc/en-us/articles/200532077-Version-Control-with-Git-and-SVN, and http://r-bio.github.io/intro-git-rstudio/.

Git was originally developed as a tool to support the development of Linux (the open source operating system that powers most web servers and many mobile devices). There were a variety of requirements but to meet the needs of a large open source project, the version control system needed to support many contributors working in parallel in a sizable code base.

Git works on the following principles:

- Git works by taking snapshots of a set of files over time
- Most operations are performed on your local machine
- Every change is captured
- Git generally adds data and does not remove it (which means it is hard to lose data)

When working in Git, there are three states that files live in: modified, staged, and committed. A modified file is self explanatory - you have made some change to a file in your project. When the file is staged, you indicate that that modified file will be incorporated into your next snapshot. When the file (or files) is/are committed, you then indicate that the staged file(s) can now be stored. Committing is indicating to Git that you are ready to take the snapshot. This workflow is captured visually below.

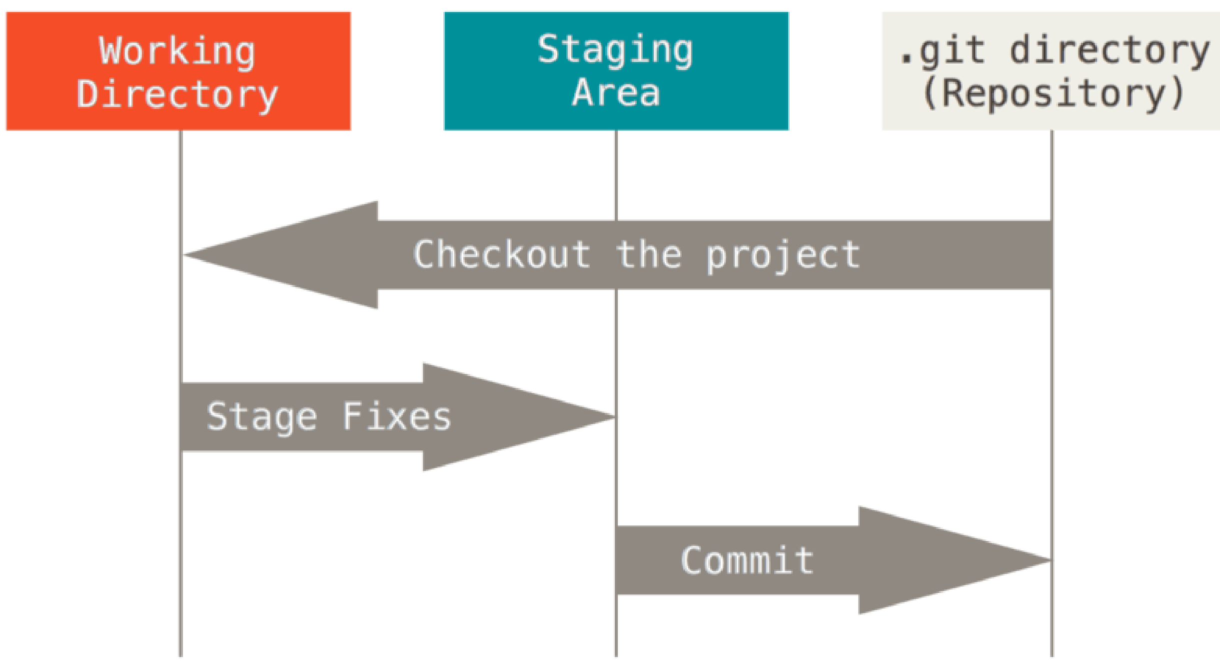


Figure 4: Git basic workflow. Source: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Hands-on with Git

First we need to learn how to interact with Git locally. We can do this easily within a RStudio project.

If you have not set up Git per the pre-course instructions (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git) and signed up for an account on Github.com (https://github.com/join), you will need to do so before you can complete the next exercise.

Exercise 1

If you have not previously set up Git and the interface within RStudio on your system, first follow these steps:

1. First we need to set up our git configuration to include our email address and user name. Open either Terminal (Mac) or Git Bash (Windows) and run the following:
2. git config –global user.name "*your username*"
3. git config –global user.email *your email address*
4. Before we can use the Git interface in RStudio, we need to enable version control in the application. Navigate to "Global Options" under the Tools menu with RStudio and select "Git/SVN" on the lefthand menu. Ensure that the check box for "Enable version control interface for RStudio projects" is checked.

Next we will take our sample project and enable Git within the project to demonstrate the workflow.

1. Navigate to your msacl-201-project RStudio window (refer to upper right hand corner to see project name)
2. Note the tabs you see on the upper right quadrant. You should see tabs for Environment, History, and Connections (depending on version of RStudio and any personal window customizations)
3. Navigate to the Tools menu in your menubar and select "Version Control", then "Project Setup…"
4. In the Project Options window that pops up, navigate to the "Git/SVN" menu (if not already there).
5. Select "Git" in the "Version control system" options.
6. A prompt for "Confirm New Git Repository" should pop up. Select "Yes".
7. A "Confirm Restart RStudio" prompt will pop up. Select "Yes".

Capturing changes using the Git window:

1. Create a new R file (one quick way: click shortcut button on top left of window above console and select "R Script").
2. Add a few comment lines (recall that comments are denoted by #) with any content you'd like (e.g. title, author, date).
3. Save the file (can use disk button) to the src folder and give it a name (e.g. sample).
4. Now navigate to the Git window on the top right.
5. There is a list of files and directories (that include files) within the project directory. Click the checkbox under "Staged" for the "src/" folder. The status column changed to include a green A icon. This indicates you are adding the file to the version control system.
6. Hit the "Commit" button (menu options within Git window).
7. A new "RStudio: Review Changes" window will pop up. Highlight the "src/sample.R" file (or whatever you named it) that you added. You will see your code highlighted in green to indicate additions to the code.
8. Type "My first commit" (or anything else you'd like) in the Commit message window on the top right, and hit the "Commit" button under it. A Git commit window will pop up showing some details of the commit you made. Close that window and close the Review Changes window.
9. Go back to your R script and delete at least one line and add another, then save.
10. Navigate back to the Git window, and repeat the steps of checking the box to stage, hitting the Commit button (note the red and green highlights in the Review Changes), writing a commit message, and commiting.

Congratulations! You have learned to stage and commit changes in your local Git repository.

End Exercise

Moving to distributed workflows

So far everything we have done has been on a local repository. A powerful aspect of Git is the ability to maintain a centralized repository outside of your local machine. This can help you synchronize the repo (short for repository) between multiple computers, but more importantly, this facilitates workflows in which multiple people contribute to a project. Imagine our local Git repository has a copy that lives on another system but is publically available for yourself and others to access. That is the function of GitHub, which hosts our course repo.

GitHub is the largest host of Git repositories and hosts open source projects (like this course) for free. GitHub also hosts private repos for a fee, and there are other services such as GitLab and BitBucket that host Git repos but also provide other functionality. GitHub is very popular among academic software projects because most are open source (and therefore free to host) but there is one important factor to consider when using the free GitHub service: content is hosted on their servers so this may not be a good fit for sensitive data (such as health information). Many organizations who write code to analyze sensitive information do not risk committing this information and purchase Git services that allow them to host repositories on their own hardware. *Always be very careful about preventing sensitive information from being available publically when working with version control system (and in general).*

One possible workflow when taking advantage of a distributed Git repository, which we refer to as a "remote" repository, is one which multiple people work from one repo and are continually bringing over copies to their local machines and then committing changes.

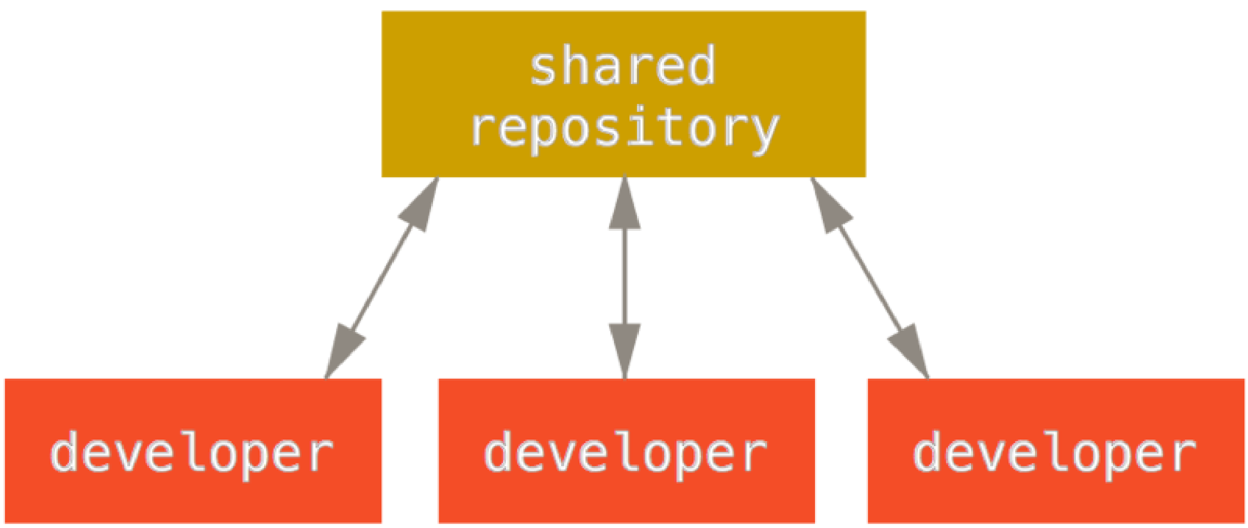


Figure 5: Centralized workflow with Git. Credit: https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

A common workflow in GitHub is one in which there is a single official project repo that contributors create a public clone of, make changes to their own repo, and request that the official repo incorporate changes into the main project ("pull request"). A step-by-step breakdown of the process illustrated below is as follows:

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer a "pull request" asking them to pull changes.
5. The maintainer adds the contributor's repository as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

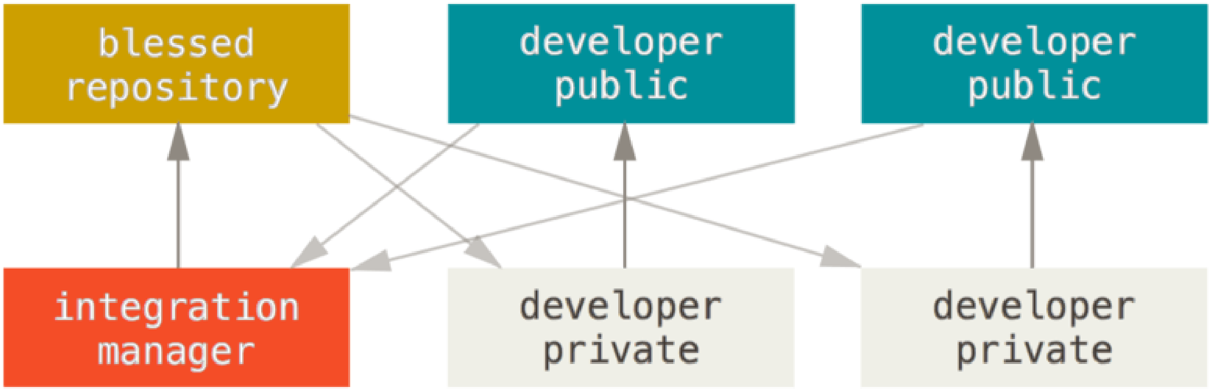


Figure 6: Integration manager workflow with Git. Credit: https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

We have placed the contents of this course into a shared Git repository (central hub) for the class to download and work with. This will help cut down on the scripting you need to run through the exercises for this course and will also give you the chance to work with Git. In the following exercise you will use Git with the existing course repository to move through the typical workflow using RStudio.

Exercise 2

Forking the course repository:

1. Navigate to the course repository at https://github.com/pcmathias/MSACL-intermediate-R-course.
2. Select the "Fork" button at the top right of the repository page. If you are not already signed in, you will be asked to sign in.
3. You should now have the course repository under your account at Github (github.com/*your-user-name*/MSACL-intermediate-R-course).

We will explain why we "forked" the repository in more detail after the exercise.

Opening the repository as a project in RStudio:

1. Under the File menu within the RStudio application, select "New Project".
2. Select "Version Control" in the first Create Project prompt.
3. Select "Git" in the next Create Project from Version Control prompt.
4. Copy and paste the URL for the repository you just forked (github.com/*your-user-name*/MSACL-intermediate-R-course) into the prompt for Repository URL.
5. Select a project name as well as a destination folder for your project (perhaps under a newly created Projects folder?).

Creating a file and using the Git workflow:

1. Let's create a new file within the repository by navigating to "New File" under the File menu and selecting "R Script".
2. Add a title to the first line by inserting a comment (using #) with a title: "# My Commit".
3. Add another comment line: "# Author: *your-user-name*".
4. Add a single line of code, eg. `print("Hello world")`.
5. Save the file in the your repository folder with the following convention: username_commit.R.

6. If not already open, open up the Git window in the top left of the RStudio window (click the Git tab). You should see your new file in that window with two boxes containing yellow question marks. Check the box for the file under Staged and you should see a green box with an "A" under the Status box. This has taken a new file (with a modified status) and staged it.
7. Stage and commit the file per the steps outlined in the previous exercise. Add "My commit" to the Commit message window and hit the "Commit" button below.

That is the general workflow you will use within RStudio. Modify (or add) a file, stage it by checking the box in the Git window, and then commit it. Be sure to include helpful comments when you commit, in case you need to go back to a previous version. All of these changes have happened locally on your machine.

End Exercise

When we first pulled the course repository, we completed the first few steps of this workflow. We took the central version of the course repo and made a local copy on our Github accounts ("forked" the repository). Then we started making local changes and committing them. Now we can work through updating the remote repository.

Exercise 3

These steps are dependent on completing the previous exercise

1. Now that you have committed changes to your local repository, you can update your remote repository on GitHub by "pushing" the local changes to the remote repository. Press the "Push" button (with a green up arrow beside it) to push your changes to remote.
2. You should be prompted for a username and password. Enter your GitHub username and password and you should seen an indication that the push has completed.
3. Navigate to your MSACL-intermediate-R-course repository on your web browser (github.com/*your-user-name*/MSACL-intermediate-R-course). You should see the file you've added there.

Now both of your local repo and your remote repo are aligned.

End Exercise

In software projects that have multiple contributors, you need a workflow that allows contributors to work on different pieces of code and contribute changes in a structured fashion, ideally so a single owner of the repository can review and incorporate changes. The common way this is done with open source projects on GitHub is the pull request workflow: a contributor forks a repository, makes some changes in their repo, and then sends a pull request to the original contributor

Optional Exercise

If you would like to try the pull request workflow

1. Navigate to your MSACL repository webpage (under your username in GitHub) and select the "New pull request" button near the top.
2. Under "Compare changes", select the link to "compare across forks".
3. Click the "base fork" button and select "pcmathias/MSACL-intermediate-R-course". Click the "base" button adjacent to the "base fork" button and select "class-contributions".
4. Click the "head fork" button and select your repository, if not already selected.
5. The "Create pull request" button should be available to select now. Click the button and add any comments to close out the pull request process.

On our end, we will get a notification about a pull request and can choose to incorporate the code into the repository.

End Exercise

You can keep your repository synchronized with the original by following steps below.

If you would like to synchronize your MSACL repo with the main course repo in the future

1. Open Terminal within RStudio on the bottom left of the window (tab is adjacent to Console tab).
2. The Terminal window should be set to your MSACL course repo directory. Run `ls` to confirm that you see the course contents. If not, use `cd` to navigate to the right directory.
3. Enter `git remote add upstream https://github.com/pcmathias/MSACL-intermediate-R-course`.
4. Enter `git remote -v` to list the remote repositories. You should see the main course repository listed as upstream.

Now your course repository is linked to the main course repo.

In the future, if you want to retrieve changes to the original course repo: 1. With your working directory set to the project directory, enter `git fetch upstream` (in Terminal console or Git Bash). This pulls any changes from the upstream repo to your local system. 1. Enter `git checkout master` to make sure you are on your master branch (explained more below). 1. Enter `git merge upstream/master` to merge the course repo changes with your local repository.

These instructions were adapted from the following: https://help.github.com/articles/syncing-a-fork.

The Git workflow for keeping changes updated is not as seamless as many modern document editors such as Office 365 or Google Docs, which continuously update changes for you without manual saving. One reason Git does not work that way is that your commits are expected to be strategic and coupled with changes that you may want to roll back. This is important to give you confidence that you do not need to create backup copies of your work, but the trade off is that you have to do extra work to make sure updates are captured. This is especially important when working with a remote repository. We made local changes and pushed those to the remote to update it. But imagine another scenario where you are working on multiple computers and made changes on computer A yesterday but are working on computer B today. If you pushed your changes from computer A to the remote yesterday, you can perform the opposite function on computer B today. You would use the "Pull" button to pull the contents of the remote repository onto your local computer B.

Addditional Git tips and tricks

Using branches When multiple people are working on a repository or you are working on multiple types of changes in a repository, there are other potential workflows besides forking a repository, making changes, and sending a pull request. A branch in Git is essentially another line of development that allows you to work without disrupting the primary line of code development (most often the *master* branch). RStudio provides support to create new branches and change branches - both features are on the top right of the Git window.

So when should you use branches? Arguably the cleanest way to use branches is to couple each branch to a major feature or change in your code. This is particularly helpful if you (and your team) want to work on multiple features at once. You can isolate each feature to branch, test it, and merge the branch (this can be done via similar workflows to the pull request) but also allows parallel development. To take this workflow one step further, GitHub and other Git-based systems allow you to open up "issues" (note the "Issues" tab on a GitHub repo page) that can include feature requests. You can open up a branch, name it for an issue, work on the feature, and then close out the issue when the feature is completed and tested.

Setting up ssh Typing in your password every time you interact with remote repository (eg. in GitHub) can be annoying to do repeatedly. An alternative is to set up SSH. At a high level, this requires setting up a public-private SSH key pair, where the private key lives on your machine (and should not be shared!) and the public key lives in your GitHub profile. There are nice instructions for setting this up from either RStudio or the shell (eg. Terminal tab) at http://happygitwithr.com/ssh-keys.html.

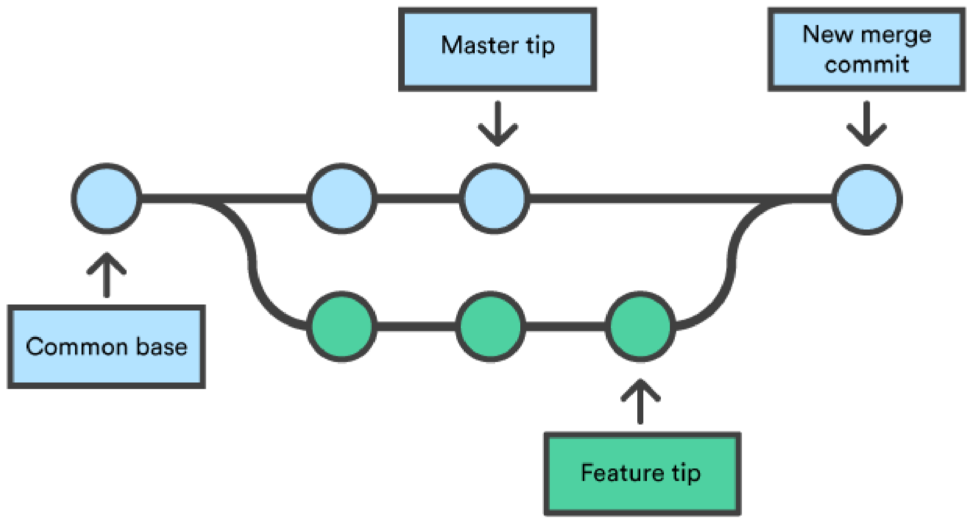


Figure 7: Branching in Git. Credit: https://www.atlassian.com/git/tutorials/using-branches/git-merge

SSH is a useful protocol to know about in general. There is a short tutorial at https://www.hostinger.com/tutorials/ssh-tutorial-how-does-ssh-work that explains many of the concepts. For a general reading resource on cryptography, The Code Book by Simon Singh is highly recommended.

What should and shouldn't go into version control The last thing to consider is a question: should you put everything in your project under version control? Maybe not. Git and similar version control systems typically do not handle raw data files well and the repository site you use may impose file size limits (Github has a 100 MB limit). *Also note that your repository site may be public so storing sensitive data (such as health information) within the repo may be problematic.* The excellent article by Wilson et al. covers these issues in more details and provides nice guidance about what should not go into version control. Practically, the .gitignore can help exclude specific files - it is simply a list of files or groups of files to ignore. As an example, including "*.html" in the .gitignore will exclude html pages from your repository. (The reason for doing this will become more obvious in the next lesson.)

Additional Git resources Finally, there are variety of resources available to learn Git. - The Happy Git and GitHub for the useR online book walks through Git in a lot more detail, with a lot more explanation. - The Pro Git textbook has a lot of detail about a variety of Git topics outside of the context of R and RStudio. - There is also a downloadable Git tutorial that may be helpful to reinforce many of the above concepts: https://github.com/jlord/git-it-electron.

Summary

- Version control helps to track changes to code as you update it, and enables you to jump back to a previous working version of your code if there is a breaking change (as an example)
- The process of staging a file, or indicating it is ready for changes, and commiting files, or explicitly capturing changes, happens locally on the file system your working files are in

- Repositories can be hosted remotely (e.g. on Github) and synced with code on your local filesystem through pulls from the remote to your local machine and pushes from your machine to the remote repository
- Efficient collaboration workflows have evolved using remote repositories and pull requests that allow distributed coders to make changes to local copies of a repostory and request those changes be incorporated into the main repository

Getting cozy with R Markdown

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.

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

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 3: Getting cozy with R Markdown'
output: html_notebook
---
```

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.

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

The text sections of your notebook can be organized into sections like a document you would create in software like Microsoft Word. There are multiple levels of headers available and syntax is simple: `# Header 1` on it's own line will create a section of the document. Under that header multiple subsections can be built with a similar syntax - just add more `#` signs to create the next level header: `## Header 2`. For the header to work, it should be on its own line.

Let's practice making modifications to the document in the header and the text.

Exercise 1

Let's use the built-in functionality in RStudio to create an R Notebook and make some modifications.

1. Add a file by selecting the add file button on the top left of your screen
2. Select R Notebook as the file type
3. Find the header. Change the title of the notebook to "My First R Notebook"
4. Add your name as the author by adding another line to the header: `author: "Your Name"`
5. Add a second level heading (##) at the end of the notebook called "My Calculation"

End Exercise

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 within a document by default:

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

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

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). Including "r mean(c(10,20,30))" surrounded by backticks will produce the following output: 20.

Exercise 2

Let's practice working with code chunks with an R Notebook.

1. Within the notebook you created in Exercise 1, find the grey code chunk. Press the green play button on the top right of the chunk. What happened? Alternately, if you cursor is within a code chunk, you can hit `Ctrl+Shift+Enter`/`Command+Shift+Enter` and the code inside the chunk will execute.
2. Insert a code chunk under the cars code chunk by using the `Ctrl+Alt+i`/`Command+Option+i`(Windows/Mac) shortcut. Another option for adding a code chunk is to use the Add Code Chunk button on the top right of the Editor window (green button with a plus sign and a C).
3. Using the new code chunk you created, display the first lines of the cars data frame with the `head(cars)` command and execute the code chunk

End Exercise

A helpful tip: 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/.

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.

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

Exercise 3

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 Word

End Exercise

Documents can be knitted to a pdf format as well, but this requires the installation of a package called tinytex if you don't already have LaTeX (a document preparation language).

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.

Exercise 4

The course repository that your forked and opened as an RStudio project has multiple R Markdown files that contain the course content. If not already open, open up the lesson 3 file: "03 - R Markdown.Rmd".

In addition to the lesson text documents, there are a few folders that each of these documents refer to.

The "assets" folder contains images and other files that can be pulled into your R Markdown document. Let's practice embedding an image into your document. The syntax for incorporating an image is `![text for image caption](folder_name/image_file.ext)`. Practice embedding the "git_basic_workflow.png" diagram from the assets folder in the space below:

Now knit the lesson 3 document to whatever format you'd like and open it.

End Exercise

These steps have set up your directory structure for future lessons. We have pre-made lesson files for future lessons, but it is also may be helpful to create an independent R Markdown file for any additional code you might want to write outside of the lesson.

A word of warning on notebooks

Running chunks in an R Markdown document can be really helpful. Similarly to working in the Console, you can write some code, execute it, and get quick feedback, all while having documentation wrapped around your code. However, there is a problem to running code chunks in notebook mode. The environment can change dynamically if you run different chunks at different times, which means that the same code chunk can produce different answers depending on the sequence you run chunks, or if you do additional work in the Console.

How do you avoid getting the wrong answer? One suggestion is to build a step in to periodically knit the whole document and review the output. Running the entire document should produce consistent results every time. Be aware of this issue and try to knit the document at least before the end of every session with an R Markdown document.

There was a JupyterCon presentation on this topic that captured this issue plus others very nicely. (Jupyter is the Python equivalent of notebooks.) There are some differences between R Markdown (plus RStudio) and Jupyter notebooks, but many of the same issues do apply.

Further reading and resources for R Markdown

Yihui Xie, who developed R Markdown and the knitr package, has written a book dedicated to R Markdown with J.J. Alaire (Founder and CEO of RStudio) and Garrett Grolemund (co-author of R For Data Science): https://bookdown.org/yihui/rmarkdown/. The book is a great resource that covers a variety of topics in addition to traditional R Markdown documents, including notebooks, slide presentations, and dashboards.

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

Reading files - beyond the basics

Reading files into R is often the start of a data analysis, and there are a number of tools to help make data import as efficient as possible.

Bread and butter data import with the *readr* package

Arguably the best "out of the box" package for data import from formatted plain text files is readr, which is one of the packages in the tidyverse. The syntax for function names in this packages is very straightforward: `read_csv()` indicates a read operation on a csv file type. Tab-delimited files can be read in with `read_tsv()`. The most generic file reading function in this package is `read_delim()`, which allows you to indicate the delimiter in the file to separate columns.

A common challenge in importing data is ensuring that the data type for a given column aligns to how you expect to work with the data. The functions in the readr package will scan the first 1000 entries by default and guess the column type based on those entries. This generally helps decrease the amount of effort required to read in data since you don't have to explicitly specify data types for each column. However this behavior does not guarantee the intended outcome for a specific field in your data set. For example, if you are importing a field that you expect to have numerical values but there are some entries with text values in the first 1000 rows, the data type for that field will be set to a character. To help navigate this issue, readr functions also provide a syntax for explicitly defining column types:

```
# 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()
  )
)
```

In addition to the data types in the example, there are a number of other formats supported by the `col_` syntax: logical, double, factor (need to specify levels), date, time, datetime. Another advantage of these functions: on import you will see that they actually explicitly tell you how the columns were parsed when you import (as we'll see in the exercise).

Exercise 1

Now let's run through using the readr function for a csv: 1. Use the `read_csv()` function to read the "2017-01-06_s.csv" file into a data frame. The file is within the "data" folder so you will need to provide a path to that files that includes the folder.

2. What is the internal structure of the object? (Hint: use the `str()` function.)

3. Summarize the data.

4. 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.

End Exercise

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")`. 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")`.

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: `read_excel("file_name.xlsx", sheet = "worksheet_name")`. In case you haven't opened an Excel file manually, there is also a helpful function to list the sheets in a file: `excel_sheets()` takes the path of the file as the argument and returns the list of sheets.

Exercise 2

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 "orders_data_set.xlsx" file into a data frame 2. View a summary of the imported data 3. Now read in only the first 5 columns using the `range` parameter 4. Review the first 6 lines of the imported data

End Exercise

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.

Why not use base functions for reading in 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 delimiting, 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.

Base R functions get the job done, but keep in mind the following weaknesses: - they are slow for reading large files (slow compared to readr, for example) - 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

For these reasons many recommend the readr package functions rather than base reading functions.

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

Writing files

Readr also offers write functions that are analogous to its reading functions, for example `write_csv()` and `write_tsv()`. `write_delim` is the most generic version of the writing functions in readr. There is a variant of `write_csv()` specifically for csv files intended to be read with Excel: `write_excel_csv()`. The primary difference between `write_csv()` and `write_excel_csv()` is that the latter adds a UTF-8 byte order mark, which is a special charater that signals to Excel the UTF-8 encoding of the file. These functions do not write row names by default. 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):

```
write_csv(x, path, na = "NA", append = FALSE, col_names = !append, delim = ",", quote_escape = "double")
```

There are a few other important parameters:

- The default argument for the na parameter is "NA", which means that the output file will contain "NA" in any cell that is empty. This may not be ideal if the target audience is opening the data in Excel but helpful if the data will be imported into R.
- If writing to an existing file, the append argument can be set to "TRUE" to append new rows rather than overwrite the existing file.
- The col_names argument specifies whether column names should appear in the first row - the default is the opposite argument listed for the append: if you are not appending rows, set the col_names argument to TRUE so the first row includes the column names.

Advantages of the readr functions for writing data include:

- similar to readr functions for reading files, writing is generally twice as fast
- by default, row names (actually row numbers) are not printed in the first column

Exercise 3

Preserving raw data without maually manipulating Excel files can be a helpful first step if you are working from a shared file or want to prevent any strange data transformations that may happen from opening and inspecting the file (e.g. timestamps coerced by Excel). One step in your analysis may be to preserve data from an Excel file as a csv. Import the "August" worksheet from the "monthly_orders_data_set.xlsx" file in the data folder, store this in an object called august_orders, and write the imported data to a csv file called "august_orders.csv" within the data folder. Output empty cells instead of NAs when there is missing data.

End Exercise

Importing dirty data

Very often the first set of operations you may want to perform on a data set that's imported is data cleaning. One package that can be very helpful for straightforward data cleaning activities 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 - `tabyl()` will tabulate into a data frame based on 1-3 variables supplied to it - `get_dupes()` returns duplicate records given a set of one or more variables - empty rows and/or columns are removed with `remove_empty()`

Let's take these functions for a spin using our data set. First let's review the first few lines of data after cleaning the column names:

```
library(janitor)
```

```
##
## Attaching package: 'janitor'

## The following objects are masked from 'package:stats':
##
##     chisq.test, fisher.test
```

```
readxl_load <- read_excel("data/orders_data_set.xlsx")
readxl_load_cleaned <- readxl_load %>%
  clean_names()
head(readxl_load_cleaned)
```

```
## # A tibble: 6 x 15
##   order_id patient_id description proc_code order_class_c_d~ lab_status_c
##      <dbl>      <dbl> <chr>       <chr>     <chr>                   <dbl>
## 1    19766     511388 PROTHROMBI~ PRO       Normal                     NA
## 2    88444     511388 BASIC META~ BMP       Normal                     NA
## 3    40477     508061 THYROID ST~ TSH       Normal                      3
## 4    97641     508061 T4, FREE    T4FR      Normal                      3
## 5    99868     505646 COMPREHENS~ COMP      Normal                      3
## 6    31178     505646 GLUCOSE SE~ GLUF      Normal                      3
## # ... with 9 more variables: lab_status_c_descr <chr>, order_status_c <dbl>,
## #   order_status_c_descr <chr>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <chr>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, department <chr>
```

The `tabyl()` function is very helpful for quick summaries to review the data you've loaded. The function expects the data frame as the first argument and then subsequent arguments indicate which variables to tabulate by. Now we'll do a quick tabulation to count the different order classes in this orders data set:

```
order_class_table <- readxl_load_cleaned %>%
  tabyl(order_class_c_descr)
order_class_table
```

```
##  order_class_c_descr     n      percent
##       Clinic Collect  6427 0.1428158749
##             External   401 0.0089107151
##           Historical     5 0.0001111062
##               Normal 36326 0.8072085685
##              On Site  1843 0.0409537354
```

Exercise 4

The orders data set we loaded with readxl contains a data set of laboratory orders. We are interested in understanding the breakdown of the tally of order classes for each specific laboratory test. Use the `tabyl` function to generate a table where the rows are the tests (description variable) and the columns represent the order_class_c_descr. Output the first 10 tests in the table.

End Exercise

Iteration: importing multiple files at once

One of the most compelling reasons to learn how to program is being able to expand your ability to automate or effortlessly repeat common actions and workflows. In most research and clinic lab environments, the data that people deal with day-to-day is not neatly stored in an easy-to-use database. It is often spread out over a series of messy spreadsheets that might be associated with one batch of data, one day of data, one week of data, or some variant. While the best practice for that scenario is probably to build a database to store the data, that requires a good amount of overhead and some expertise. By taking advantage of iteration in R, you can dump similarly formatted files into data frames/tibbles.

The purrr package has a variety of `map()` functions that are well-explained in the iteration chapter of R for Data Science. The `map()` functions take a vector as an input, applies a function to elements of the vector, and returns a vector of identical length to the input vector. There are a number of map functions that correspond to the data type of the output. For example, `map()` returns a list, `map_int()` returns a vector of integers, `map_chr()` returns a character vector, and `map_dfr()` returns a data frame. These are very similar to the `apply()` family of functions but there are some advantages of the purrr functions, including consistent compatibility with pipes and more predictable output data types.

How does this work? Let's take a simple example right out of the R for Data Science text. We'll start with a tibble (tidyverse version of data frame) consisting of 4 variables (a through d) with 10 observations from a normal distribution.

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
df
```

```
## # A tibble: 10 x 4
##          a       b      c       d
##      <dbl>   <dbl>  <dbl>   <dbl>
##  1  0.0585 -0.0193 -0.207  0.332
##  2 -0.603   0.112   0.374  0.280
##  3  1.37    0.974   1.24   0.803
##  4 -1.97    2.77    0.722  2.83
##  5  2.12   -0.112   2.17  -0.890
##  6  1.32    1.01    0.900  0.701
##  7  0.149   0.176  -0.160 -1.80
##  8 -1.98   -0.644   1.47   0.397
##  9  0.631  -0.0685 -0.810 -0.149
## 10  2.85    0.0902  0.287 -0.0957
```

We want to treat each variable as a vector and perform a calculation on each. If we want to take the mean of each and want the output to have a double data type, we use `map_dbl()`:

```
df %>%
  map_dbl(mean)
```

```
##         a         b         c         d
## 0.3944035 0.4295362 0.5989396 0.2407053
```

That is a pretty simple example but it captures the types of operations you can you do by iterating through a data set. For those of you who are familiar with for loops, the map functions can offer similar functionality but are much shorter to write and straight-forward to understand.

Earlier in this lesson we discussed file reading functions, with the recognition that many data analysis tasks rely on flat files for source data. In a laboratory running batched testing such as a mass spectrometry lab, files are often tied to batches and/or dates and named correspondingly. If you want to analyze a set of data over multiple batches, you may find yourself importing data from each individually and stitching together the data using a function like `bind_rows()` (we will discuss this function in a future lesson). The `map()` functions (often `map_dfr()` specifically) can automate this process and save you a lot of time. There are a few prerequisites for this to work, though:

- the underlying file structure must be the same: for spreadsheet-like data, columns must be in the same positions in each with consistent data types
- the files must have the same file extension
- if there are multiple different file types (with different data structures) mixed in one directory, the files must organized and named in a way to associate like data sets with like

In the last lesson we placed our large mass spec data set in the data folder. This consists of a series of monthly data that are grouped into batches, samples, and peaks data, with suffixes of "_b“,”_s“, and”_p", respectively. Let's read all of the sample data into one data frame (technically a tibble). We are going to use the `read_csv()` function since the files are csvs. To use the `map_dfr()` function, we need to supply a vector as input - in this case, a vector of file names. How do generate that input vector?

- First we use `list.files()`, which produces a character vector of names of files in a directory, which is the first argument. The function allows a pattern argument which you can supply with a text string for it to match against - all of the sample files end in "_s.csv".
- Next we pipe that list to `file.path()`, which provides an operating system agnostic way of spitting out a character vector that corresponds to the appropriate file name and path. We started with the names of the files we care about, but we need to append the “data” folder to the beginning of the

names. You'll notice that we used a period as the second argument - this is because by default the pipe feeds the output of the previous step into the first argument. The period is a placeholder to indicate that the output should be fed into a different argument.
- Finally we feed that character to `map_df()`, which takes the `read_csv()` function as its argument. With the map family of functions, there is no need to include the parentheses in the function name if there aren't arguments.

```
all_samples <- dir_ls("data", glob = "*_s.csv") %>%
  map_dfr(read_csv) %>%
  clean_names()
```

```
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
```

```
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
```

```
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
```

```
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
```

```
summary(all_samples)
```

```
##   batch_name         sample_name         compound_name         ion_ratio
##  Length:2244840     Length:2244840     Length:2244840     Min.   :0.0000
##  Class :character   Class :character   Class :character   1st Qu.:0.0000
##  Mode  :character   Mode  :character   Mode  :character   Median :0.8165
##                                                            Mean   :0.6564
##                                                            3rd Qu.:1.2452
##                                                            Max.   :2.4332
##     response        concentration     sample_type         expected_concentration
##  Min.   :0.0000   Min.   :  0.00   Length:2244840     Min.   :  0.00
##  1st Qu.:0.0000   1st Qu.:  0.00   Class :character   1st Qu.:  0.00
##  Median :0.2982   Median : 42.55   Mode  :character   Median :  0.00
##  Mean   :0.9658   Mean   :134.46                      Mean   : 35.77
##  3rd Qu.:1.8593   3rd Qu.:261.81                      3rd Qu.:  0.00
##  Max.   :9.2258   Max.   :860.59                      Max.   :500.00
##  used_for_curve  sample_passed
##  Mode :logical   Mode :logical
##  FALSE:1956363   FALSE:57190
##  TRUE :288477    TRUE :2187650
##
##
##
```

Exercise 5

An Excel spreadsheet with multiple sheets containing the same exact data structure is a common pattern of organization that can be painful to work with when parsing data. Luckily map functions can help make the process of importing multiple sheets less painful. There are two additional functions we want to use. We briefly discussed the `excel_sheets()` function in the readxl section - this will return a list of sheets in an Excel file and takes the file path/name as the argument. Once we have a list we'll want to use the `set_names()` function to label each of the data frames we generate from the sheets with the sheet names.

a) Use the `map()` function to create a list of data frames, each containing one of the sheets in the "monthly_orders_data_set.xlsx" file, read the date from each sheet, and store the result in an object called orders_list.

b) Use the `map_df()` function to create a single data frame containing all of the data from the 3 sheets. Use the ".id" argument to add a column indicating which sheet each row came from.

End Exercise

If you weren't already aware of this solution or another for reading in multiple files at once, the purrr package is an extremely handy tool for doing this. Just be aware of the requirements for doing this, and **always check the output**. You do not want to automate a bad or broken process!

Summary

- 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
- The purrr package has useful tools for iterating that can be very powerful when coupled with file reading functions

Data manipulation in the tidyverse

A brief review of tidy data principles & the tidyverse

According to the official tidyverse website, "the tidyverse is an *opinionated* collection of R packages designed for data science." The bottom line is that 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
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

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

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

The tidy representation is:

```
table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

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.

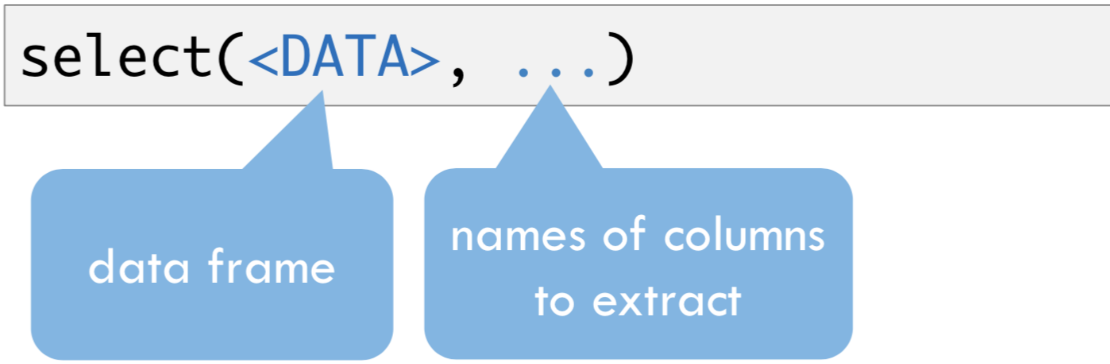
Manipulating data with dplyr

The dplyr package provides functions to carve, expand, and collapse a data frame (or tibble). To complement dplyr, we have also loaded the tidylog package, which provides additional output to clarify exactly what the dplyr package did when you run certain commands.

Select columns

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.

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("data/2017-01-06_s.csv",
  col_types = cols(
    compoundName = col_factor(NULL),
    sampleType = col_factor(NULL)
    )
  ) %>%
  clean_names()
str(samples_jan)
```

```
## tibble [187,200 x 10] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ batch_name            : chr [1:187200] "b802253" "b802253" "b802253" "b802253" ...
##  $ sample_name           : chr [1:187200] "s253001" "s253001" "s253001" "s253001" ...
##  $ compound_name         : Factor w/ 6 levels "morphine","hydromorphone",..: 1 2 3 4 5 6 1 2 3 4 ...
##  $ ion_ratio             : num [1:187200] 0 0 0 0 0 0 0 0 0 0 ...
##  $ response              : num [1:187200] 0 0 0 0 0 0 0 0 0 0 ...
##  $ concentration         : num [1:187200] 0 0 0 0 0 0 0 0 0 0 ...
##  $ sample_type           : Factor w/ 4 levels "blank","standard",..: 1 1 1 1 1 1 2 2 2 2 ...
##  $ expected_concentration: num [1:187200] 0 0 0 0 0 0 0 0 0 0 ...
##  $ used_for_curve        : logi [1:187200] FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ sample_passed         : logi [1:187200] FALSE TRUE TRUE TRUE TRUE TRUE ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   batchName = col_character(),
##   ..   sampleName = col_character(),
##   ..   compoundName = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE),
##   ..   ionRatio = col_double(),
##   ..   response = col_double(),
##   ..   concentration = col_double(),
##   ..   sampleType = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE),
##   ..   expectedConcentration = col_double(),
##   ..   usedForCurve = col_logical(),
##   ..   samplePassed = col_logical()
##   .. )
```

The simplest use of `select()` is to call out specific column names for the new data set:

```r
samples_jan_2columns <- samples_jan %>%
  select(sample_name, concentration)
```

```
## select: dropped 8 variables (batch_name, compound_name, ion_ratio, response, sample_type, ...)
```

```r
head(samples_jan_2columns)
```

```
## # A tibble: 6 x 2
##   sample_name concentration
##   <chr>               <dbl>
## 1 s253001                 0
## 2 s253001                 0
## 3 s253001                 0
## 4 s253001                 0
## 5 s253001                 0
## 6 s253001                 0
```

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:

```r
samples_jan_subset <- samples_jan %>%
  select(batch_name:expected_concentration)
```

```
## select: dropped 2 variables (used_for_curve, sample_passed)
```

```r
head(samples_jan_subset)
```

```
## # A tibble: 6 x 8
##   batch_name sample_name compound_name ion_ratio response concentration
##   <chr>      <chr>       <fct>             <dbl>    <dbl>         <dbl>
## 1 b802253    s253001     morphine              0        0             0
## 2 b802253    s253001     hydromorphone         0        0             0
## 3 b802253    s253001     oxymorphone           0        0             0
## 4 b802253    s253001     codeine               0        0             0
## 5 b802253    s253001     hydrocodone           0        0             0
## 6 b802253    s253001     oxycodone             0        0             0
## # ... with 2 more variables: sample_type <fct>, expected_concentration <dbl>
```

We can accomplish the same selection using `-` to indicate which columns to drop:

```r
samples_jan_subset <- samples_jan %>%
  select(-used_for_curve, -sample_passed)
```

```
## select: dropped 2 variables (used_for_curve, sample_passed)
```

```r
head(samples_jan_subset)
```

```
## # A tibble: 6 x 8
##   batch_name sample_name compound_name ion_ratio response concentration
##   <chr>      <chr>       <fct>             <dbl>    <dbl>         <dbl>
## 1 b802253    s253001     morphine              0        0             0
## 2 b802253    s253001     hydromorphone         0        0             0
## 3 b802253    s253001     oxymorphone           0        0             0
## 4 b802253    s253001     codeine               0        0             0
## 5 b802253    s253001     hydrocodone           0        0             0
## 6 b802253    s253001     oxycodone             0        0             0
## # ... with 2 more variables: sample_type <fct>, expected_concentration <dbl>
```

Or if we only care about the first 3 variables plus the concentration we can combine a range of adjacent columns plus calling other columns explicitly:

```
samples_jan_subset <- samples_jan %>%
  select(batch_name:compound_name, concentration)
```

```
## select: dropped 6 variables (ion_ratio, response, sample_type, expected_concentration, used_for_curve
```

```
head(samples_jan_subset)
```

```
## # A tibble: 6 x 4
##   batch_name sample_name compound_name concentration
##   <chr>      <chr>       <fct>                 <dbl>
## 1 b802253    s253001     morphine                  0
## 2 b802253    s253001     hydromorphone             0
## 3 b802253    s253001     oxymorphone               0
## 4 b802253    s253001     codeine                   0
## 5 b802253    s253001     hydrocodone               0
## 6 b802253    s253001     oxycodone                 0
```

There are also helper functions to select columns meeting certain criteria:

```
samples_jan_vars <- samples_jan %>%
  select(starts_with("sample"))
```

```
## select: dropped 7 variables (batch_name, compound_name, ion_ratio, response, concentration, ...)
```

```
glimpse(samples_jan_vars)
```

```
## Rows: 187,200
## Columns: 3
## $ sample_name   <chr> "s253001", "s253001", "s253001", "s253001", "s253001"...
## $ sample_type   <fct> blank, blank, blank, blank, blank, blank, standard, s...
## $ sample_passed <lgl> FALSE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE...
```

Filter rows

Now let's carve the data set in the other direction. 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.

If we only care about the morphine data, we can use `filter()` to pick those rows based on a logical condition:

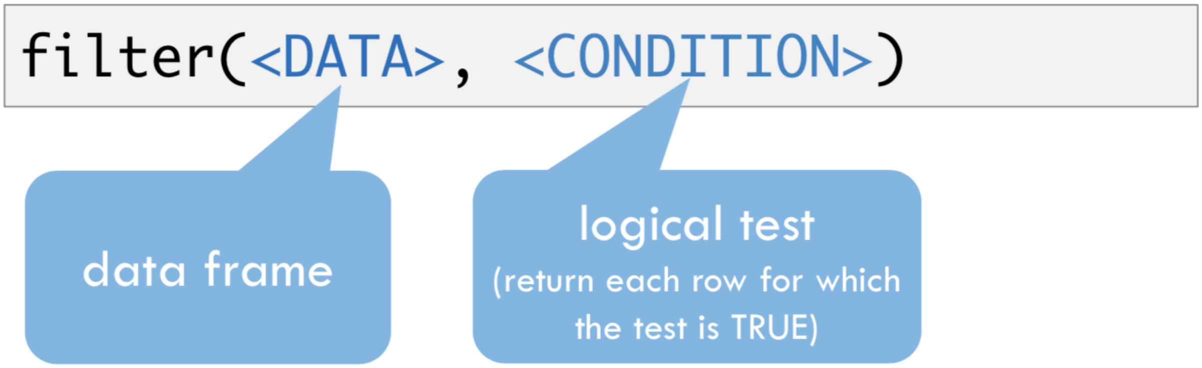


Figure 8: Syntax for `filter()`

```
samples_jan %>%
  filter(compound_name == "morphine") %>% # note the two equal signs (one equal for assignment)
  head()
```

```
## filter: removed 156,000 rows (83%), 31,200 rows remaining
```

```
## # A tibble: 6 x 10
##   batch_name sample_name compound_name ion_ratio response concentration
##   <chr>      <chr>       <fct>             <dbl>    <dbl>         <dbl>
## 1 b802253    s253001     morphine          0        0              0
## 2 b802253    s253002     morphine          0        0              0
## 3 b802253    s253003     morphine          0.735    0.147         19.0
## 4 b802253    s253004     morphine          0.817    0.427         55.1
## 5 b802253    s253005     morphine          0.885    0.769         99.2
## 6 b802253    s253006     morphine          0.714    1.48         191.
## # ... with 4 more variables: sample_type <fct>, expected_concentration <dbl>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

We may be interested in more than one compound, in which case the `%in%` operator can allow us to `filter()` based on a list:

```
samples_jan %>%
  filter(compound_name %in% c("morphine", "hydromorphone")) %>% # note the use of the c() function to c
  head()
```

```
## filter: removed 124,800 rows (67%), 62,400 rows remaining
```

```
## # A tibble: 6 x 10
##   batch_name sample_name compound_name ion_ratio response concentration
##   <chr>      <chr>       <fct>             <dbl>    <dbl>         <dbl>
## 1 b802253    s253001     morphine          0        0              0
## 2 b802253    s253001     hydromorphone     0        0              0
## 3 b802253    s253002     morphine          0        0              0
## 4 b802253    s253002     hydromorphone     0        0              0
## 5 b802253    s253003     morphine          0.735    0.147         19.0
## 6 b802253    s253003     hydromorphone     0.811    0.136         17.7
```

```
## # ... with 4 more variables: sample_type <fct>, expected_concentration <dbl>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

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

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

```
## filter: removed 115,298 rows (62%), 71,902 rows remaining
```

```
## # A tibble: 6 x 10
##   batch_name sample_name compound_name ion_ratio response concentration
##   <chr>      <chr>       <fct>             <dbl>    <dbl>         <dbl>
## 1 b802253    s253010     codeine           0.881     2.48          303.
## 2 b802253    s253011     codeine           0.790     1.94          237.
## 3 b802253    s253011     oxycodone         0.813     4.13          458.
## 4 b802253    s253012     morphine          0.775     2.83          365.
## 5 b802253    s253012     hydromorphone     0.851     1.45          189.
## 6 b802253    s253012     codeine           0.774     3.23          394.
## # ... with 4 more variables: sample_type <fct>, expected_concentration <dbl>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

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(sample_type == "unknown" | concentration > 0) %>%
  head()
```

```
## filter: removed 10,800 rows (6%), 176,400 rows remaining
```

```
## # A tibble: 6 x 10
##   batch_name sample_name compound_name ion_ratio response concentration
##   <chr>      <chr>       <fct>             <dbl>    <dbl>         <dbl>
## 1 b802253    s253003     morphine          0.735    0.147          19.0
## 2 b802253    s253003     hydromorphone     0.811    0.136          17.7
## 3 b802253    s253003     oxymorphone       0.716    0.146          18.3
## 4 b802253    s253003     codeine           0.811    0.179          21.8
## 5 b802253    s253003     hydrocodone       0.767    0.146          22.2
## 6 b802253    s253003     oxycodone         0.841    0.188          20.8
## # ... with 4 more variables: sample_type <fct>, expected_concentration <dbl>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

Exercise 1

Carve the January data set in both directions. Extract 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.

End Exercise

Add new columns with `mutate()`

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(sample_type == "standard", expected_concentration > 0) %>%
  mutate(conc_ratio = concentration/expected_concentration) %>%
  select(batch_name:compound_name, concentration, expected_concentration, conc_ratio) %>%
  head(20)
```

```
## filter: removed 165,600 rows (88%), 21,600 rows remaining
```

```
## mutate: new variable 'conc_ratio' (double) with 21,593 unique values and 0% NA
```

```
## select: dropped 5 variables (ion_ratio, response, sample_type, used_for_curve, sample_passed)
```

```
## # A tibble: 20 x 6
##    batch_name sample_name compound_name concentration expected_concen~
##    <chr>      <chr>       <fct>                 <dbl>            <dbl>
##  1 b802253    s253003     morphine               19.0               20
##  2 b802253    s253003     hydromorphone          17.7               20
##  3 b802253    s253003     oxymorphone            18.3               20
##  4 b802253    s253003     codeine                21.8               20
##  5 b802253    s253003     hydrocodone            22.2               20
##  6 b802253    s253003     oxycodone              20.8               20
##  7 b802253    s253004     morphine               55.1               50
##  8 b802253    s253004     hydromorphone          66.5               50
##  9 b802253    s253004     oxymorphone            64.1               50
## 10 b802253    s253004     codeine                37.3               50
## 11 b802253    s253004     hydrocodone            55.0               50
## 12 b802253    s253004     oxycodone              43.1               50
## 13 b802253    s253005     morphine               99.2              100
## 14 b802253    s253005     hydromorphone          99.1              100
## 15 b802253    s253005     oxymorphone            98.7              100
## 16 b802253    s253005     codeine                90.7              100
## 17 b802253    s253005     hydrocodone            97.0              100
## 18 b802253    s253005     oxycodone             125.               100
## 19 b802253    s253006     morphine              191.               200
## 20 b802253    s253006     hydromorphone         203.               200
## # ... with 1 more variable: conc_ratio <dbl>
```

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(sample_type == "standard") %>%
  mutate(
    conc_ratio = case_when(
      expected_concentration == 0 ~ (concentration + 0.001)/(expected_concentration + 0.001),
      TRUE ~ concentration/expected_concentration
    )
  ) %>%
  select(batch_name:compound_name, concentration, expected_concentration, conc_ratio) %>%
  head(20)
```

```
## filter: removed 162,000 rows (87%), 25,200 rows remaining
```

```
## mutate: new variable 'conc_ratio' (double) with 21,593 unique values and 0% NA
```

```
## select: dropped 5 variables (ion_ratio, response, sample_type, used_for_curve, sample_passed)
```

```
## # A tibble: 20 x 6
##    batch_name sample_name compound_name concentration expected_concen~
##    <chr>      <chr>       <fct>                 <dbl>            <dbl>
##  1 b802253    s253002     morphine                0                  0
##  2 b802253    s253002     hydromorphone           0                  0
##  3 b802253    s253002     oxymorphone             0                  0
##  4 b802253    s253002     codeine                 0                  0
##  5 b802253    s253002     hydrocodone             0                  0
##  6 b802253    s253002     oxycodone               0                  0
##  7 b802253    s253003     morphine               19.0               20
##  8 b802253    s253003     hydromorphone          17.7               20
##  9 b802253    s253003     oxymorphone            18.3               20
## 10 b802253    s253003     codeine                21.8               20
## 11 b802253    s253003     hydrocodone            22.2               20
## 12 b802253    s253003     oxycodone              20.8               20
## 13 b802253    s253004     morphine               55.1               50
## 14 b802253    s253004     hydromorphone          66.5               50
## 15 b802253    s253004     oxymorphone            64.1               50
## 16 b802253    s253004     codeine                37.3               50
## 17 b802253    s253004     hydrocodone            55.0               50
## 18 b802253    s253004     oxycodone              43.1               50
## 19 b802253    s253005     morphine               99.2              100
## 20 b802253    s253005     hydromorphone          99.1              100
## # ... with 1 more variable: conc_ratio <dbl>
```

Another common operation is generating new columns to wrangle 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_csv("data/2017-01-06_b.csv") %>%
  clean_names()
```

```
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   instrumentName = col_character(),
##   compoundName = col_character(),
##   calibrationSlope = col_double(),
##   calibrationIntercept = col_double(),
##   calibrationR2 = col_double(),
##   batchPassed = col_logical(),
##   reviewerName = col_character(),
##   batchCollectedTimestamp = col_datetime(format = ""),
##   reviewStartTimestamp = col_datetime(format = ""),
##   reviewCompleteTimestamp = col_datetime(format = "")
## )
```

```
batch_jan_timestamps <- batch_jan %>%
  mutate(
    collect_datetime = ymd_hms(batch_collected_timestamp),
    collect_month = month(batch_collected_timestamp),
    collect_day_of_week = wday(batch_collected_timestamp),
    collect_week = week(batch_collected_timestamp),
    collect_week_alt = floor_date(collect_datetime, unit = "week")
    # floor_date to use datetime format but group to first day of week
  )
```

```
## mutate: new variable 'collect_datetime' (double) with 587 unique values and 0% NA

##          new variable 'collect_month' (double) with 2 unique values and 0% NA

##          new variable 'collect_day_of_week' (double) with 7 unique values and 0% NA

##          new variable 'collect_week' (double) with 6 unique values and 0% NA

##          new variable 'collect_week_alt' (double) with 6 unique values and 0% NA
```

```
summary(batch_jan_timestamps)
```

```
##   batch_name         instrument_name     compound_name      calibration_slope
##  Length:3600         Length:3600         Length:3600        Min.   :0.003172
##  Class :character    Class :character    Class :character   1st Qu.:0.006794
##  Mode  :character    Mode  :character    Mode  :character   Median :0.007060
##                                                             Mean   :0.007107
##                                                             3rd Qu.:0.007351
##                                                             Max.   :0.009626
##  calibration_intercept calibration_r2   batch_passed    reviewer_name
##  Min.   :-9.510e-05    Min.   :0.9800   Mode:logical    Length:3600
##  1st Qu.:-2.160e-05    1st Qu.:0.9860   TRUE:3600       Class :character
##  Median : 6.965e-08    Median :0.9902                   Mode  :character
##  Mean   : 7.202e-08    Mean   :0.9899
##  3rd Qu.: 2.180e-05    3rd Qu.:0.9938
##  Max.   : 1.082e-04    Max.   :1.0000
```

```
##  batch_collected_timestamp     review_start_timestamp
##  Min.   :2017-01-06 20:08:00   Min.   :2017-01-07 09:08:00
##  1st Qu.:2017-01-13 22:55:15   1st Qu.:2017-01-14 12:05:45
##  Median :2017-01-21 11:00:30   Median :2017-01-21 23:18:30
##  Mean   :2017-01-21 10:58:03   Mean   :2017-01-21 23:24:24
##  3rd Qu.:2017-01-28 23:24:30   3rd Qu.:2017-01-29 11:17:15
##  Max.   :2017-02-05 01:54:00   Max.   :2017-02-05 13:49:00
##  review_complete_timestamp     collect_datetime              collect_month
##  Min.   :2017-01-07 09:35:00   Min.   :2017-01-06 20:08:00   Min.   :1.000
##  1st Qu.:2017-01-14 12:24:15   1st Qu.:2017-01-13 22:55:15   1st Qu.:1.000
##  Median :2017-01-21 23:41:30   Median :2017-01-21 11:00:30   Median :1.000
##  Mean   :2017-01-21 23:54:28   Mean   :2017-01-21 10:58:03   Mean   :1.143
##  3rd Qu.:2017-01-29 11:55:00   3rd Qu.:2017-01-28 23:24:30   3rd Qu.:1.000
##  Max.   :2017-02-05 14:15:00   Max.   :2017-02-05 01:54:00   Max.   :2.000
##  collect_day_of_week  collect_week  collect_week_alt
##  Min.   :1.00        Min.   :1.00   Min.   :2017-01-01 00:00:00
##  1st Qu.:2.00        1st Qu.:2.00   1st Qu.:2017-01-08 00:00:00
##  Median :4.00        Median :3.00   Median :2017-01-15 00:00:00
##  Mean   :4.09        Mean   :3.39   Mean   :2017-01-17 17:31:12
##  3rd Qu.:6.00        3rd Qu.:4.00   3rd Qu.:2017-01-22 00:00:00
##  Max.   :7.00        Max.   :6.00   Max.   :2017-02-05 00:00:00
```

Note, there is an alternate week selector function, `isoweek()`. This one uses the ISO 8601 system and begins each week on a Monday. The `week()` selector returns the number of complete seven day periods that have occurred between the date and January 1st, plus one.

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.

Another common manipulation of interest to the laboratory is calculating the difference between 2 timestamps to analyze turnaround times (TATs). Lubridate has a duration data type and associated operator `%--%` for calculating differences between timestamps. Both variables we perform the calculation on should already be recognized by R as a datetime data type (e.g. POSIXct) and the calculation will return a special interval data type. The syntax is `start_time %--% end_time`. Here we calculate the turnaround time from collection to completing review of a batch:

```
batch_jan_tat <- batch_jan %>%
  mutate(tat_duration = batch_collected_timestamp %--% review_complete_timestamp) %>%
  select(tat_duration)
```

```
## mutate: new variable 'tat_duration' (double) with 600 unique values and 0% NA
```

```
## select: dropped 11 variables (batch_name, instrument_name, compound_name, calibration_slope, calibra
```

```
head(batch_jan_tat)
```

```
## # A tibble: 6 x 1
##   tat_duration
##   <Interval>
## 1 2017-01-06 21:40:00 UTC--2017-01-07 14:09:00 UTC
## 2 2017-01-06 21:40:00 UTC--2017-01-07 14:09:00 UTC
## 3 2017-01-06 21:40:00 UTC--2017-01-07 14:09:00 UTC
```

```
## 4 2017-01-06 21:40:00 UTC--2017-01-07 14:09:00 UTC
## 5 2017-01-06 21:40:00 UTC--2017-01-07 14:09:00 UTC
## 6 2017-01-06 21:40:00 UTC--2017-01-07 14:09:00 UTC
```

The duration data type will display both timestamps and store the difference between the times in seconds, but this isn't very helpful if we want to visualize the data in common units of measure such as minutes, hours, days, weeks, or months. Lubridate provides a helpful set of duration objects that can help convert calculated durations into a usable numeric value. Functions such as `dminutes()`, `dhours()`, and `ddays()` take an argument for the number of that unit to perform a calculation for and return a duration object that contains the number of seconds corresponding to the unit of interest. For example, `dhours(1)` will return a duration object with a value of 3600 (seconds). When you divide a duration by another duration the result will be a numeric data element. Here we can expand our TAT duration calculation for each row of the data frame into TATs in minutes, hours, and days:

```
batch_jan_tat <- batch_jan %>%
  mutate(tat_duration = batch_collected_timestamp %--% review_complete_timestamp,
         tat_minutes = tat_duration / dminutes(1),
         tat_hours = tat_duration / dhours(1),
         tat_days = tat_duration / ddays(1))
```

```
## mutate: new variable 'tat_duration' (double) with 600 unique values and 0% NA

##         new variable 'tat_minutes' (double) with 368 unique values and 0% NA

##         new variable 'tat_hours' (double) with 368 unique values and 0% NA

##         new variable 'tat_days' (double) with 368 unique values and 0% NA
```

Exercise 2

How long an average does it take to review each batch? Using the January batch data, 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. 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.

End Exercise

Grouped summaries with `group_by()` and `summarize()`

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_ir_stats <- samples_jan %>%
  filter(sample_type == "unknown", concentration > 0) %>%
  group_by(compound_name) %>%
  summarize(median_ir = median(ion_ratio),
            mean_ir = mean(ion_ratio),
            std_dev_ir = sd(ion_ratio))
```

```
## filter: removed 115,298 rows (62%), 71,902 rows remaining

## group_by: one grouping variable (compound_name)

## summarize: now 6 rows and 4 columns, ungrouped
```

```
samples_jan_ir_stats
```

```
## # A tibble: 6 x 4
##   compound_name median_ir mean_ir std_dev_ir
##   <fct>             <dbl>   <dbl>      <dbl>
## 1 morphine           1.24    1.20      0.168
## 2 hydromorphone      1.24    1.20      0.165
## 3 oxymorphone        1.24    1.20      0.165
## 4 codeine            1.24    1.20      0.166
## 5 hydrocodone        1.24    1.20      0.166
## 6 oxycodone          1.24    1.20      0.166
```

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

```
samples_jan_batch_ir_stats <- samples_jan %>%
  filter(sample_type == "unknown", concentration > 0) %>%
  group_by(batch_name, compound_name) %>%
  summarize(median_ir = median(ion_ratio),
            mean_ir = mean(ion_ratio),
            std_dev_ir = sd(ion_ratio))
```

```
## filter: removed 115,298 rows (62%), 71,902 rows remaining

## group_by: 2 grouping variables (batch_name, compound_name)

## summarize: now 3,600 rows and 5 columns, one group variable remaining (batch_name)
```

```
head(samples_jan_batch_ir_stats)
```

```
## # A tibble: 6 x 5
## # Groups:   batch_name [1]
##   batch_name compound_name median_ir mean_ir std_dev_ir
##   <chr>      <fct>             <dbl>   <dbl>      <dbl>
## 1 b100302    morphine           1.23    1.26     0.0698
## 2 b100302    hydromorphone      1.23    1.25     0.0634
## 3 b100302    oxymorphone        1.23    1.21     0.0743
## 4 b100302    codeine            1.23    1.25     0.0830
## 5 b100302    hydrocodone        1.29    1.27     0.0898
## 6 b100302    oxycodone          1.26    1.27     0.0760
```

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_cor_stats <- batch_jan_timestamps %>%
  group_by(instrument_name, compound_name, collect_week) %>%
  summarize(median_cor = median(calibration_r2),
            mean_cor = mean(calibration_r2),
            min_cor = min(calibration_r2),
            max_cor = max(calibration_r2))
```

```
## group_by: 3 grouping variables (instrument_name, compound_name, collect_week)
```

```
## summarize: now 234 rows and 7 columns, 2 group variables remaining (instrument_name, compound_name)
```

```
head(batch_jan_cor_stats)
```

```
## # A tibble: 6 x 7
## # Groups:   instrument_name, compound_name [2]
##   instrument_name compound_name collect_week median_cor mean_cor min_cor max_cor
##   <chr>           <chr>                <dbl>      <dbl>    <dbl>   <dbl>   <dbl>
## 1 bashful         codeine                  1      0.989    0.990   0.981   0.996
## 2 bashful         codeine                  2      0.991    0.990   0.981   0.999
## 3 bashful         codeine                  3      0.990    0.989   0.980   1.00
## 4 bashful         codeine                  4      0.992    0.991   0.983   0.998
## 5 bashful         codeine                  5      0.991    0.992   0.981   0.999
## 6 bashful         hydrocodone              1      0.989    0.990   0.985   0.997
```

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?

End Exercise

Whenever you use the `group_by()` function, the data frame preserves the groups, so it is important to recognize that subsequent operations you perform on the data will work on those groups. Performing a `summarize()` "peels off" one variable from the group. Let's revisit the the sample_states_jan data from the last exercise. We grouped by batch_name and compound_name and performed one `summarize()` to generate statistics for the concentrations by batch and compound. Reviewing the sample_stats_jan data frame shows that the output is still grouped by batch_name (see Groups output above the output table):

```
## filter: removed 43,200 rows (23%), 144,000 rows remaining
```

```
## group_by: 2 grouping variables (batch_name, compound_name)
```

```
## summarize: now 3,600 rows and 6 columns, one group variable remaining (batch_name)
```

```
head(sample_stats_jan)
```

```
## # A tibble: 6 x 6
## # Groups:   batch_name [1]
##   batch_name compound_name min_conc median_conc mean_conc max_conc
##   <chr>      <fct>            <dbl>       <dbl>     <dbl>    <dbl>
## 1 b100302    morphine             0        173.      182.     497.
```

```
## 2 b100302    hydromorphone        0         0         126.     477.
## 3 b100302    oxymorphone          0        28.2       106.     431.
## 4 b100302    codeine              0       107.        147.     444.
## 5 b100302    hydrocodone          0         0         126.     467.
## 6 b100302    oxycodone            0         0         122.     523.
```

If we run another summarize without updating the groups, statisitics will automatically be calculated by batch:

```
sample_stats_batch <- sample_stats_jan %>%
  summarize(batch_median_conc = median(median_conc))
```

```
## summarize: now 600 rows and 2 columns, ungrouped
```

```
sample_stats_batch
```

```
## # A tibble: 600 x 2
##    batch_name batch_median_conc
##    <chr>                  <dbl>
##  1 b100302                 14.1
##  2 b101197                 50.9
##  3 b101972                  0
##  4 b102100                 12.3
##  5 b102508                 51.9
##  6 b103050                 19.3
##  7 b103382                  0
##  8 b104730                 44.5
##  9 b106474                 73.9
## 10 b106839                  0
## # ... with 590 more rows
```

We may be more interested in medians of within-batch median by compound rather than batch. In that case the grouping of sample_stats_jan does not align with the summarization we'd like to do. We can call `group_by()` to re-establish a different grouping for the data set and use `summarize()` as expected:

```
sample_stats_compound <- sample_stats_jan %>%
  group_by(compound_name) %>%
  summarize(compound_median_conc = median(median_conc))
```

```
## group_by: one grouping variable (compound_name)
```

```
## summarize: now 6 rows and 2 columns, ungrouped
```

```
sample_stats_compound
```

```
## # A tibble: 6 x 2
##   compound_name compound_median_conc
##   <fct>                        <dbl>
## 1 morphine                      16.1
## 2 hydromorphone                 14.8
## 3 oxymorphone                   25.3
## 4 codeine                       17.2
## 5 hydrocodone                   13.6
## 6 oxycodone                     21.6
```

You may want to do a further data manipulation and transformation after an initial `group_by()` and `summarize()`. The safest practice to unsure you don't unintentionally perform grouped operations without realizing it is to use `ungroup()` to remove the groups:

```
sample_stats_jan_ungrouped <- sample_stats_jan %>%
  ungroup()
```

```
## ungroup: no grouping variables
```

```
summary(sample_stats_jan_ungrouped)
```

```
##   batch_name              compound_name    min_conc   median_conc
##  Length:3600        morphine     :600   Min.   :0   Min.   :  0.00
##  Class :character   hydromorphone:600   1st Qu.:0   1st Qu.:  0.00
##  Mode  :character   oxymorphone  :600   Median :0   Median : 17.61
##                     codeine      :600   Mean   :0   Mean   : 39.08
##                     hydrocodone  :600   3rd Qu.:0   3rd Qu.: 68.57
##                     oxycodone    :600   Max.   :0   Max.   :262.65
##    mean_conc         max_conc
##  Min.   : 39.54   Min.   :338.9
##  1st Qu.:112.65   1st Qu.:466.0
##  Median :129.40   Median :483.7
##  Mean   :129.97   Mean   :480.9
##  3rd Qu.:147.05   3rd Qu.:495.2
##  Max.   :223.91   Max.   :684.7
```

Scaling column-wise operations with `across()`

One common activity when transforming data is performing the same operation across multiple columns in your data frame. As an example let's revisit our January samples data set and calculate mean values for ion_ratio, response, and concentration for each distinct group of sample_type and compound_name:

```
samples_jan_means <- samples_jan %>%
  group_by(sample_type, compound_name) %>%
  summarize(ion_ratio = mean(ion_ratio),
            response = mean(response),
            concentration = mean(concentration))
```

```
## group_by: 2 grouping variables (sample_type, compound_name)

## summarize: now 24 rows and 5 columns, one group variable remaining (sample_type)
```

This may require multiple copies and pastes and would be quite frustrating if you had to perform the same operation on 20 columns rather than just 3. The `across()` function provides a mechanism to repeat the same operation across multiple columns:

```
samples_jan_means <- samples_jan %>%
  group_by(sample_type, compound_name) %>%
  summarize(across(ion_ratio:concentration, mean))
```

```
## group_by: 2 grouping variables (sample_type, compound_name)
```

```
## summarize: now 24 rows and 5 columns, one group variable remaining (sample_type)
```

The first argument to `across()` is the column(s) to operate on, and this can be provided as a list or using syntax you would use for the `select()` function. In the example above, 4 consecutive columns are selected with the `:`. The second argument to the `across()` function is a function or list of functions to apply to each column. This argument will accept formulas similar to those you may use with the purrr package, such as `~ .x /2` to divide by 2: the `~` tells R to evaluate the expression following it as a function and `.x` indicates a list or vector to operate on.

An additional helper function `where()` can select columns based on specific criteria rather than calling out columns specifically. In this case we specify calculating the mean on any column meeting the criteria of being a numeric data type:

```
samples_jan_means <- samples_jan %>%
  group_by(sample_type, compound_name) %>%
  summarize(across(where(is.numeric), mean))
```

```
## group_by: 2 grouping variables (sample_type, compound_name)
```

```
## summarize: now 24 rows and 6 columns, one group variable remaining (sample_type)
```

```
samples_jan_means
```

```
## # A tibble: 24 x 6
## # Groups:   sample_type [4]
##    sample_type compound_name ion_ratio response concentration expected_concentr~
##    <fct>       <fct>             <dbl>    <dbl>         <dbl>              <dbl>
##  1 blank       morphine           0        0              0                  0
##  2 blank       hydromorphone      0        0              0                  0
##  3 blank       oxymorphone        0        0              0                  0
##  4 blank       codeine            0        0              0                  0
##  5 blank       hydrocodone        0        0              0                  0
##  6 blank       oxycodone          0        0              0                  0
##  7 standard    morphine           1.03     1.19         167.               167.
##  8 standard    hydromorphone      1.03     1.19         168.               167.
##  9 standard    oxymorphone        1.03     1.20         168.               167.
## 10 standard    codeine            1.03     1.19         168.               167.
## # ... with 14 more rows
```

In this data set, there are no missing data, but it's helpful to know that the `na.rm = TRUE` argument that is often added to base summary statistics functions like `mean()` can be added to the `across()` function as well:

```
samples_jan_means <- samples_jan %>%
  group_by(sample_type, compound_name) %>%
  summarize(across(where(is.numeric), mean, na.rm = TRUE))
```

```
## group_by: 2 grouping variables (sample_type, compound_name)
```

```
## summarize: now 24 rows and 6 columns, one group variable remaining (sample_type)
```

```
samples_jan_means
```

```
## # A tibble: 24 x 6
## # Groups:   sample_type [4]
##    sample_type compound_name ion_ratio response concentration expected_concentr~
##    <fct>       <fct>             <dbl>    <dbl>         <dbl>              <dbl>
##  1 blank       morphine           0        0              0                  0
##  2 blank       hydromorphone      0        0              0                  0
##  3 blank       oxymorphone        0        0              0                  0
##  4 blank       codeine            0        0              0                  0
##  5 blank       hydrocodone        0        0              0                  0
##  6 blank       oxycodone          0        0              0                  0
##  7 standard    morphine           1.03     1.19         167.               167.
##  8 standard    hydromorphone      1.03     1.19         168.               167.
##  9 standard    oxymorphone        1.03     1.20         168.               167.
## 10 standard    codeine            1.03     1.19         168.               167.
## # ... with 14 more rows
```

Finally, `across()` allows you to perform multiple functions on multiple columns by providing a list of functions. Let's calculate the `min()` and `max()` across the numeric columns in the January samples data set:

```
min_max <- list(
  min = ~min(.x, na.rm = TRUE),
  max = ~max(.x, na.rm = TRUE)
)
samples_jan_minmax <- samples_jan %>%
  group_by(sample_type, compound_name) %>%
  summarize(across(where(is.numeric), min_max))
```

```
## group_by: 2 grouping variables (sample_type, compound_name)
```

```
## summarize: now 24 rows and 10 columns, one group variable remaining (sample_type)
```

```
samples_jan_minmax
```

```
## # A tibble: 24 x 10
## # Groups:   sample_type [4]
##    sample_type compound_name ion_ratio_min ion_ratio_max response_min
##    <fct>       <fct>                 <dbl>         <dbl>        <dbl>
##  1 blank       morphine                  0          0               0
##  2 blank       hydromorphone             0          0               0
##  3 blank       oxymorphone               0          0               0
##  4 blank       codeine                   0          0               0
##  5 blank       hydrocodone               0          0               0
##  6 blank       oxycodone                 0          0               0
##  7 standard    morphine                  0          1.54            0
##  8 standard    hydromorphone             0          1.72            0
##  9 standard    oxymorphone               0          1.68            0
## 10 standard    codeine                   0          1.58            0
## # ... with 14 more rows, and 5 more variables: response_max <dbl>,
## #   concentration_min <dbl>, concentration_max <dbl>,
## #   expected_concentration_min <dbl>, expected_concentration_max <dbl>
```

Another powerful use case for `across()` is using it in combination with `mutate()` to transform multiple columns at once. The syntax for using the function is the same as before: the first argument includes the columns to operate on and the second includes the function that will operate on those columns. In addition to the function to transform existing columns with, the `mutate()` function also expects the name(s) of any new columns to be included. As an example we may want to view rounded versions of numeric columns in our samples data set but retain the original columns. In this case we provide a single function within a list and add "rounded" to append to the existing column names for which the condition applies:

```
samples_jan_rounded <- samples_jan %>%
  mutate(across(where(is.numeric), list(rounded = ~ round(.x, 2))))
```

```
## mutate: new variable 'ion_ratio_rounded' (double) with 111 unique values and 0% NA

##         new variable 'response_rounded' (double) with 530 unique values and 0% NA

##         new variable 'concentration_rounded' (double) with 43,863 unique values and 0% NA

##         new variable 'expected_concentration_rounded' (double) with 10 unique values and 0% NA
```

```
samples_jan_rounded
```

```
## # A tibble: 187,200 x 14
##    batch_name sample_name compound_name ion_ratio response concentration
##    <chr>      <chr>       <fct>             <dbl>    <dbl>         <dbl>
##  1 b802253    s253001     morphine              0        0             0
##  2 b802253    s253001     hydromorphone         0        0             0
##  3 b802253    s253001     oxymorphone           0        0             0
##  4 b802253    s253001     codeine               0        0             0
##  5 b802253    s253001     hydrocodone           0        0             0
##  6 b802253    s253001     oxycodone             0        0             0
##  7 b802253    s253002     morphine              0        0             0
##  8 b802253    s253002     hydromorphone         0        0             0
##  9 b802253    s253002     oxymorphone           0        0             0
## 10 b802253    s253002     codeine               0        0             0
## # ... with 187,190 more rows, and 8 more variables: sample_type <fct>,
## #   expected_concentration <dbl>, used_for_curve <lgl>, sample_passed <lgl>,
## #   ion_ratio_rounded <dbl>, response_rounded <dbl>,
## #   concentration_rounded <dbl>, expected_concentration_rounded <dbl>
```

The result is that every numeric column has a mutated counterpart with rounded values.

Exercise 4

A common operation in some contexts is to group data together by date. For example, you may be interested in test volumes over time and want to count by various dates represented in your data set. To do this efficiently you may want to create new columns with dates corresponding to every timestamp in your data set. Recall that the `floor_date()` function acts as a type of rounding function and returns a timestamp that's rounded down to a specified time unit (e.g. unit = "day" as the second argument rounds to the date). For the batch_jan data, add an additional column for every timestamp variable using the `is.POSIXct()` (to identify timestamp columns) and `floor_date()` functions.

End Exercise

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("data/messy/2017-01-06-sample-messy.csv")
```

```
## Parsed with column specification:
## cols(
##   batch_name = col_character(),
##   sample_name = col_character(),
##   sample_type = col_character(),
##   morphine = col_double(),
##   hydromorphone = col_double(),
##   oxymorphone = col_double(),
##   codeine = col_double(),
##   hydrocodone = col_double(),
##   oxycodone = col_double()
## )
```

```
head(samples_jan_messy)
```

```
## # A tibble: 6 x 9
##   batch_name sample_name sample_type morphine hydromorphone oxymorphone codeine
##   <chr>      <chr>       <chr>          <dbl>         <dbl>       <dbl>   <dbl>
## 1 b100302    s302001     blank            0             0           0       0
## 2 b100302    s302002     standard         0             0           0       0
## 3 b100302    s302003     standard        18.4          21.6        21.6    16.9
## 4 b100302    s302004     standard        49.4          38.8        46.4    49.4
## 5 b100302    s302005     standard        86.5          97.1       106.    119.
## 6 b100302    s302006     standard       188.          189.        201.    197.
## # ... with 2 more variables: hydrocodone <dbl>, oxycodone <dbl>
```

In this case, we have sample_type and sample_name stored in the rows, compound_name spread across the column names, and concentrations stored in cells.

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`pivot_longer()` function makes datasets longer by increasing the number of rows and decreasing the number of columns. (Previous versions (i.e., prior to 1.0.0) of `tidyr` used the `gather()` function to gather data into tidy, longer formats. The `gather()` function isn't going away, but it is recommended to use `pivot_longer()` instead.)

```
samples_jan_tidy_longer <-samples_jan_messy %>%
  pivot_longer(cols = c(-batch_name, -sample_name, -sample_type), names_to = "compound_name", values_to
```

```
## pivot_longer: reorganized (morphine, hydromorphone, oxymorphone, codeine, hydrocodone, ...) into (co
```

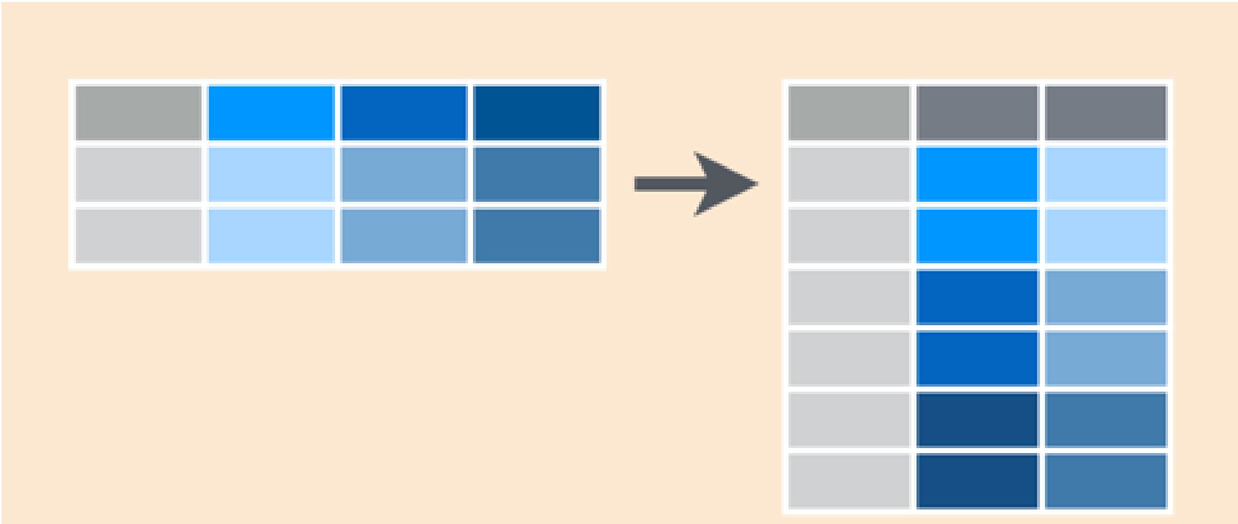


Figure 9: Pivot longer operation

```
head(samples_jan_tidy_longer)
```

```
## # A tibble: 6 x 5
##   batch_name sample_name sample_type compound_name concentration
##   <chr>      <chr>       <chr>       <chr>                 <dbl>
## 1 b100302    s302001     blank       morphine                  0
## 2 b100302    s302001     blank       hydromorphone             0
## 3 b100302    s302001     blank       oxymorphone               0
## 4 b100302    s302001     blank       codeine                   0
## 5 b100302    s302001     blank       hydrocodone               0
## 6 b100302    s302001     blank       oxycodone                 0
```

The syntax takes some getting used to, so it's important to remember that you are taking column names and placing those into rows, so you have to name that variable (via `names_to` argument) – where do you want the names to go, and you are also putting values across multiple columns into one column, whose variable also needs to be named (via the `values_to` argument) – where do you want the values to go. You have to provide the dataframe you want it to work on and which columns should be gathered. Here, we specified all but the first three columns.

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 inverse operation of `pivot_longer()` using the `pivot_wider()` function. The `pivot_wider()` function increases the number of columns and decreases the number of rows, making data messy (non-tidy). (As above, the older approach of using the `spread()` function isn't going away, but it is recommended to use `pivot_wider()` instead.)

Similar to the syntax shown above: in the `names_from` argument, you specify the variable that needs to be used to generate multiple new columns – where do you get the names from; you also specify the `values_from` argument to indicate which variable will be used to populate the values of those new columns – where do you get the values from.

Let's apply these inverse functions on the data sets we just tidied:

```r
# using pivot_wider
samples_jan_remessy_wider <- samples_jan_tidy_longer %>%
  pivot_wider(names_from = "compound_name", values_from = "concentration")
```

```
## pivot_wider: reorganized (compound_name, concentration) into (morphine, hydromorphone, oxymorphone,
```

```r
head(samples_jan_remessy_wider)
```

```
## # A tibble: 6 x 9
##   batch_name sample_name sample_type morphine hydromorphone oxymorphone codeine
##   <chr>      <chr>       <chr>          <dbl>         <dbl>       <dbl>   <dbl>
## 1 b100302    s302001     blank            0             0           0       0
## 2 b100302    s302002     standard         0             0           0       0
## 3 b100302    s302003     standard        18.4          21.6        21.6    16.9
## 4 b100302    s302004     standard        49.4          38.8        46.4    49.4
## 5 b100302    s302005     standard        86.5          97.1       106.    119.
## 6 b100302    s302006     standard       188.          189.        201.    197.
## # ... with 2 more variables: hydrocodone <dbl>, oxycodone <dbl>
```

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. We will demonstrate use of `unite()` in the next lesson.

Exercise 5

The "2017-01-06-batch-messy.csv" file in the messy subdirectory of the data dir 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 observation (compound) - calibration slope, intercept, and R^2. However these appear on different lines. All 3 parameters 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 `pivot_longer()` and `pivot_wider()` to reformat this data.

End Exercise

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
- `across()` scales columnwise operations
- `pivot_longer()` and `pivot_wider()` allow you to pivot your data frame to and from a tidy data structure

Blending data from multiple files and sources

Joining Relational Data

The database example for this class has three different tibbles: 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 in specific batches by instrument, 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.

```
january_batches <- read_csv("data/2017-01-06_b.csv") %>%
  clean_names() #use help to check out what this does
january_samples <- read_csv("data/2017-01-06_s.csv") %>%
  clean_names()
january_peaks <- read_csv("data/2017-01-06_p.csv") %>%
  clean_names()
select_batches <- january_batches %>%
  filter(batch_name %in% c("b802253", "b252474"))
select_samples <- january_samples %>%
  filter(batch_name %in% c("b802253", "b252474"))
select_peaks <- january_peaks %>%
  filter(batch_name %in% c("b802253", "b252474"))
```

Blending Data

Simple addition of rows and columns

Sometimes, you need to combine data stored in more than one file. For example, managing the QC deviations across twelve separate months of reports. To do this in R, you can read each file and then merge them together either by row, or by column. The idea behind *tidy data* is that each column is a variable, each row is an observation, and each element is a value. If you know that your data sources have the same shape (same variables and same observations), you can safely combine them with an `bind_rows` to append the second source of data at the end of the first.

```
january_samples <- read_csv("data/2017-01-06_s.csv") %>%
  clean_names()
```

```
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
```

```
##   samplePassed = col_logical()
## )
```

```
february_samples <- read_csv("data/2017-02-06_s.csv") %>%
  clean_names()
```

```
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration = col_double(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
```

```
two_months <- bind_rows(january_samples, february_samples)
```

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 two_months is the sum of the first two months of sample-level data. Another way to see this is to add a column name for the ".id" argment in the `bind_rows` call.

```
two_months_id <- bind_rows(january_samples, february_samples, .id = "month") #.id is the dataframe iden
```

```
two_months[187195:187204,]
```

```
## # A tibble: 10 x 10
##    batch_name sample_name compound_name ion_ratio response concentration
##    <chr>      <chr>       <chr>             <dbl>    <dbl>         <dbl>
##  1 b208048    s048052     morphine           1.23    1.21          165.
##  2 b208048    s048052     hydromorphone      0       0               0
##  3 b208048    s048052     oxymorphone        1.28    0.447          65.8
##  4 b208048    s048052     codeine            1.20    1.42          230.
##  5 b208048    s048052     hydrocodone        0       0               0
##  6 b208048    s048052     oxycodone          0       0               0
##  7 b593231    s231001     morphine           0       0               0
##  8 b593231    s231001     hydromorphone      0       0               0
##  9 b593231    s231001     oxymorphone        0       0               0
## 10 b593231    s231001     codeine            0       0               0
## # ... with 4 more variables: sample_type <chr>, expected_concentration <dbl>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

```
c(nrow(january_samples), nrow(february_samples), nrow(two_months))
```

```
## [1] 187200 187200 374400
```

```r
two_months_id[187195:187204,]
```

```
## # A tibble: 10 x 11
##    month batch_name sample_name compound_name ion_ratio response concentration
##    <chr> <chr>      <chr>       <chr>             <dbl>    <dbl>         <dbl>
##  1 1     b208048    s048052     morphine           1.23    1.21          165.
##  2 1     b208048    s048052     hydromorphone      0       0               0
##  3 1     b208048    s048052     oxymorphone        1.28    0.447          65.8
##  4 1     b208048    s048052     codeine            1.20    1.42          230.
##  5 1     b208048    s048052     hydrocodone        0       0               0
##  6 1     b208048    s048052     oxycodone          0       0               0
##  7 2     b593231    s231001     morphine           0       0               0
##  8 2     b593231    s231001     hydromorphone      0       0               0
##  9 2     b593231    s231001     oxymorphone        0       0               0
## 10 2     b593231    s231001     codeine            0       0               0
## # ... with 4 more variables: sample_type <chr>, expected_concentration <dbl>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

As long as the two tibbles have the same number of columns and the same column names, the `bind_rows` 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. That makes `bind_rows` useful but remember to make sure the data are clean before you use this function.

Exercise 1

Try to use `bind_rows()` to combine all of the sample data from February and each of the three tibbles containing January data. Do any of them work? What does the data look like?

First try binding a peaks file with the February samples file:

Observe the number of columns and visualize the new data frame directly.

Next try the batches and samples file:

Now bind both samples files:

End Exercise

There is an related command called `bind_cols` which will append columns to a tibble, but it also requires very clean data. This command will not check to make sure the order of values are correct between the two things being bound.

```r
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")

desired_bind   <- bind_cols(incomplete_data, additional_columns)
head(desired_bind)
```

```
## # A tibble: 4 x 5
##   sampleName compoundName  concentration expectedConcentration sampleType
##   <chr>      <chr>                 <dbl>                 <dbl> <chr>
```

```
## 1 123456     morphine                   34                       20 standard
## 2 123456     hydromorphone              35                       30 standard
## 3 123456     codeine                    44                       40 standard
## 4 123456     hydrocodone                45                       40 standard
```

Binding using relationships between data objects

Using *dplyr* there is another way of binding data which does not require the items being combined to be identical in shape. It does require adopting a relational database approach to the design of your data structures. This is, at the core, the primary idea behind *tidy* data.

Primary and foreign keys

A key is the variable in a tibble – or combination of variables in a tibble – that uniquely defines every row. In our data, `batch_name` is present in each tibble but is insufficient to define a specific row. If we want to join data from different tables and ensure it matches to the correct row, we need a key. As it turns out for this data set, no single column operates as a key. We can build a key by combining two (or three) columns. Here is how to combine values which are not unique to an individual observation in order to create a key which is unique to each observation. We create the key for the `select_peaks` data using a dplyr alternative function to `paste()` (base R) called `unite()`. This function takes the data as the first argument (piped in this examples), and then will put together specified columns using a separator you specify. If you don't want to remove the variables used to construct the key, you add the "remove = FALSE" argument.

```
select_batches <- select_batches %>%
  unite(keyB, c(batch_name, compound_name), sep=":", remove = FALSE)
```

This creates what is called a *primary key*, which is the unique identifier for each observation in a specific tibble. A *foreign key* is the same thing, only it uniquely identifies an observation in another tibble. The `left_join` command joins two tibbles based on matching the *primary key* in the first tibble with the *foreign key* in the second tibble.

```
#create keys in peaks and samples tables
select_peaks <- select_peaks %>%
  unite(keyB, c(batch_name, compound_name), sep=":", remove = FALSE)

select_samples <- select_samples %>%
  unite(keyB, c(batch_name, compound_name), sep=":", remove = FALSE)

#join by key
combined <- left_join(select_samples, select_batches, by= "keyB")
```

```
## left_join: added 13 columns (batch_name.x, compound_name.x, batch_name.y, instrument_name, compound_n

##             > rows only in x     0

##             > rows only in y  (   0)

##             > matched rows     624

##             >                 =====

##             > rows total       624
```

Mutating join to add columns

Mutating joins operate in much the same way as the set operations (`union()`, `intersect()`, `setdiff()`), 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 `select_batches$keyB` got repeated for both the Quant and the Qual peak entries in `select_peaks`. Having built the `select_batches` primary key, and correctly included it as a foreign key in `select_peaks`, correctly joining them into a single data frame is straightforward.

```
select_peaksWide <- left_join(select_peaks,select_batches)
```

```
## Joining, by = c("keyB", "batch_name", "compound_name")

## left_join: added 9 columns (instrument_name, calibration_slope, calibration_intercept, calibration_r2

##              > rows only in x   1,248

##              > rows only in y  (    0)

##              > matched rows     1,248

##              >                 =======

##              > rows total       2,496
```

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.

Exercise 2:

Join the data from the `select_samples` dataset with the data from `select_peaksWide`. Try using a `left_join()` and a `right_join()` to see the difference. We want to join the peaks and batches data to the samples in a way that eliminates the rows with internal standard information – so retain data that is in `select_samples`.

End Exercise

Back to our problem

We started out with the intention to combine data from three tables so we could report qualifier and quantifier peak areas of QC samples from select batches, noting which instrument was used. We now know how to create a dataset we need for this analysis.

Exercise 3

(1) Join `january_peaks`, `january_batches`, and `january_samples`. *Hint: first create a key in each table. Join to retain rows in samples table.*

(2) Filter this joined dataset for QCs and group by instrument, compound name, expected concentration, and chromatogram.

Summarize the grouped data to find the mean, sd, and cv of peak areas and the number of qcs for a given condition.

(3) Create a graphic from the joined dataset showing boxplots of peak areas for the qualifier and quantifier peaks of each compound at each expected QC concentration, colored by instrument.

Bonus! Create a boxplot showing QC concentration by compound and instrument for each expected concentration.

Summary

- `rbind` and `cbind` add rows (or columns) to an existing data frame
- 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 `select_batches`)
  - A foreign key is a unique identifier for another data frame (the presence of `keyB` in `select_peaks`)
- `inner_join`, `full_join`, `left_join`, and `right_join` are mutating joins which add columns of one table to another

Using databases

Motivations for working with relational databases

Managing your data within text or Excel files is often the default approach since instruments (whether mass spectrometers or any other lab instrument) generate the data in this format. Files may be spread out over multiple directories and if multiple files are required for analysis they are copied into one location to work with. You may either manually copy and paste the data together into one large file or import multiple files into your environment (possibly into one data frame) within your analysis code. This pattern presents a practical challenge under a few scenarios: - You are collecting longitudinal data and want to work with a large number of files over some time period (dozens or more). - The entire data set you are working with is large and exceeds the memory of the system you are analyzing data on. - The data set you are working with natively exists in a relational database and cutting out the process of extracting the data and importing into R can make your analysis more efficient or effective. One compelling use case is developing a dashboard that automatically refreshes when the database has new data.

One approach to managing data in these scenarios is to store it in a relational database and connect to the data with a database connection using R. Many of the tidyverse packages such as dplyr have built-in compatibility with relational databases that is supported with a package called dbplyr. This allows R to translate the code that you write into the native language of the database. You can then take advantage of the functionality of a database without having to be an expert in the database language (although it definitely helps to know the basics of the language).

Connecting to databases with R

Connecting to a database is analogous to reading data into a file: specific functions are required to interact with the outside data source. The DBI package allows R to communicate with various relational database management systems (RDBMS). This package provides a general mechanism to connect but in addition each specific RDBMS also requires a separate package to support the appropriate syntax and translate commands into the specific RDBMS commands. For example, the RSQLite package allows connection to SQLite.

The first step in connecting to a database is to use the `dbConnect()` function from the DBI package. This function accepts a number of arguments to configure the database connection, but the most important is the definition of the driver. For example, connecting to a SQLite database can be done by calling `RSQLite::SQLite()` as the first argument (RSQLite is the name of the package for the driver and `SQLite()` is the function called to set up the connection). The other arguments to provide the function include the location of the database (e.g. a file or a host server name), database name (often the host has multiple databases), username, password, and port (for databases configured to use a specific port).

Setup

1. Add your "project_data.sqlite" file (downloaded prior to class) containing a database of the course mass spec project results into your "data" folder.
2. Install a couple packages specific to different "flavors" of SQL:

```
install.packages(c("RSQLite", "RPostgres"), dependencies = TRUE)
```

End Setup

As an example, let's connect to a publicly available PostgreSQL database and provide all the details required to establish the connection. Below we connect to a database hosted by RNAcentral, which requires authentication (user and password) to access the database in addition to specifying the server and database name that you are connecting to. `dbConnect()` uses all of these arguments to establish a connection.

```
exampledb <- dbConnect(RPostgres::Postgres(),
                       host = 'hh-pgsql-public.ebi.ac.uk',   # server address
                       port = 5432,   # PostgreSQL TCP port is 5432 by default
                       dbname = 'pfmegrnargs',   # specific database to access (may be multiple dbs)
                       user = 'reader',
                       password = 'NWDMCE5xdipIjRrp')
```

If that executes successfully, you should see an object in your environment called exampledb that is a PqConnection type. R now has the connection info in your environment and you can use that connection to access specific tables. The database we've connected to stores RNA sequences in a table called "rna". We can use the dplyr function `tbl()` to create a table from the PostgreSQL data source. The function takes a data source as the first argment, and in the case of a database, will take a table name to generate a table. We can then use a familiar function to perform an operation on the table.

```
rna <- tbl(exampledb, "rna")
head(rna, 10)
```

```
## # Source:   lazy query [?? x 9]
## # Database: postgres [reader@hh-pgsql-public.ebi.ac.uk:5432/pfmegrnargs]
##        id upi   timestamp           userstamp crc64   len seq_short seq_long
##     <int6> <chr> <dttm>              <chr>     <chr> <int> <chr>     <chr>
##   1 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    0E96~   613 GCAGTCGA~ <NA>
##   2 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    1B05~    89 GGAAAGGT~ <NA>
```

```
##  3 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    B48E~   330 CGCGCAGG~ <NA>
##  4 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    27B7~   236 TACCGAAG~ <NA>
##  5 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    BE4F~    85 GGGTGAGT~ <NA>
##  6 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    1AEF~   440 CCTACGGG~ <NA>
##  7 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    65EF~    73 CGATCCAG~ <NA>
##  8 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    6687~   269 GAGCACGT~ <NA>
##  9 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    3C2C~    85 GCGAGCGT~ <NA>
## 10 1.00e7 URS0~ 2018-03-12 14:27:08 rnacen    F252~    85 GCGGATGT~ <NA>
## # ... with 1 more variable: md5 <chr>
```

Note that the rna object in the environment is not actually a tibble/data frame - it's a list. R actually does not immediately pull the data in when you use the `tbl()` function - it generates and stores a query to perform on the database. When you need to retrieve data, for example using the `head()` function or when you'd like to generate a plot, it will perform the query at that time. If you want to pull all the data in prior to when you actually need to perform local operations on it, you can use the `collect()` function.

Finally, when you no longer need to work with the database, you'll probably want to close the connection using the `dbDisconnect()` function.

```
dbDisconnect(exampledb)
```

Let's practice connecting with a much simpler database. The project data for this course has been converted into a SQLite database, which has the advantage of storing the whole database in a single file. With this database, many of the connection and authentication details are not required - you just need to point the connection function to the file location.

Exercise 1

The project dataset for the course is included in a file called "project_data.sqlite", so we will connect to this this SQLite database.

1. Use the `dbConnect()` function to connect to this database, and refer to the following site for some additional info on connecting: https://cran.r-project.org/web/packages/RSQLite/vignettes/RSQLite.html. The first argument is the driver, and for SQLite, the second argument indicates the location of the database (or a temporary one to create one on the fly). Load this connection into an object called projectdb.
2. The data in this database mirrors the structure of the files we've seen already. There are tables for batches, samples, and peaks. Connect to the "sample" table and view the first 10 records.
3. Using the dplyr tools we have learned thus far, perform a `summary()` on only the standards from the sample table (sample_type == 'standard') and note the minimum, median, and maximum concentrations. Hint: you may need to bring the data into your local environment for `summary()` to perform as expected.

End Exercise

The basics of Structured Query Language (SQL)

The principles of relational databases were developed by Edward Codd at IBM in the early 1970s and were based on relational algebra. Using these principles, another group developed a programming language that evolved into Structured Query Language (originally called SEQUEL and pronounced either as "S-Q-L" or "sequel") to represent these principles. Core database principles have actually been adopted heavily by tidyverse package developers. You actually already know these concepts because we have introduced them in previous lessons but did not call them out as database concepts. These concepts include: - Data are represented as a group of tables, which is analagous to working with a group of data frames. - The principles of tidy data are adapted from common relational database practices: - Observations are represented by rows (often called tuples in relational database speak) - Variables are stored in columns (commonly referred to as fields) - Tables are linked together with variables that are shared - this principle is used to join data sets

SQL was intended to be more accessible than many of the programming languages of that time, and the basic syntax for queries is relatively simple. The most basic query has two "clauses": - a SELECT clause chooses the columns to return in a query - this is identical to the `select()` functionality from the dplyr package - a FROM clause chooses the table for which the columns are returned - this is analogous to specifying the data frame you apply a function to

RStudio has nice multi-language support that allows us to run SQL within a Markdown file, provided we supply the connection to run the query on. As an example, let's consider our project data database we connected to in the last exercise. If we wanted to retrieve sample name, compound name, and ion ratio from a "sample" table, we would write the following SQL query:

```sql
SELECT
        sample_name, compound_name, ion_ratio
FROM
        sample
LIMIT
        10;
```

Table 1: Displaying records 1 - 10

sample_name	compound_name	ion_ratio
s253001	morphine	0
s253001	hydromorphone	0
s253001	oxymorphone	0
s253001	codeine	0
s253001	hydrocodone	0
s253001	oxycodone	0
s253002	morphine	0
s253002	hydromorphone	0
s253002	oxymorphone	0
s253002	codeine	0

If we want to only obtain data for one specific coumpound, e.g. morphine, we add a WHERE clause with a logical condition, functioning identically to the `filter()` command.

```sql
SELECT
        sample_name, compound_name, ion_ratio
FROM
        sample
WHERE
        compound_name = 'morphine'   -- Note the single quotes
```

Table 2: Displaying records 1 - 10

sample_name	compound_name	ion_ratio
s253001	morphine	0.0000000
s253002	morphine	0.0000000
s253003	morphine	0.7348524
s253004	morphine	0.8170208
s253005	morphine	0.8847819
s253006	morphine	0.7138970
s253007	morphine	0.8650822
s253008	morphine	0.8288112
s253009	morphine	0.0000000
s253010	morphine	0.0000000

Note a few minor details in the above query that are different than R syntax: - equality is represented with one equal sign ("assingment" of an object is not done in a similar way in SQL so there is no risk from this one symbol being used for multiple things) - the string 'morphine' is enclosed in single quotes and SQL is strict about only using single quotes (unlike R) - comments are added with two dashes (– unlike # for comments in R)

One final basic concept in SQL is one you have already learned: the different types of joins in R are pulled exactly from SQL. Recognizing the SQL syntax is the final hurdle. Joins are performed within the FROM clause of the query. If we want to join the sample table with the batch table by the batch name and the compound name, we perform the following query:

```
SELECT
        sample.batch_name, sample.sample_name, sample.compound_name, sample.concentration,
        batch.instrument_name, batch.reviewer_name, batch.calibration_slope
FROM
        sample
        INNER JOIN batch ON sample.batch_name = batch.batch_name
          AND sample.compound_name = batch.compound_name
LIMIT  10;
```

Table 3: Displaying records 1 - 10

batch_name	sample_name	compound_name	concentration	instrument_name	reviewer_name	calibration_slope
b802253	s253001	morphine	0	doc	Xavier	0.0077502
b802253	s253001	hydromorphone	0	doc	Xavier	0.0076783
b802253	s253001	oxymorphone	0	doc	Xavier	0.0079751
b802253	s253001	codeine	0	doc	Xavier	0.0081921
b802253	s253001	hydrocodone	0	doc	Xavier	0.0065643
b802253	s253001	oxycodone	0	doc	Xavier	0.0090238
b802253	s253002	morphine	0	doc	Xavier	0.0077502
b802253	s253002	hydromorphone	0	doc	Xavier	0.0076783
b802253	s253002	oxymorphone	0	doc	Xavier	0.0079751
b802253	s253002	codeine	0	doc	Xavier	0.0081921

There are few more details to consider in the query above: - When joining multiple tables, columns may be derived from one or more of the source tables so SQL wants explicit specification of the the source of the column. The syntax for specifying the table for a column is "table.column". - Asterisks can be used to select all columns from a specific table. Rather than calling out the tables as above, you can also just use a single asterisk to query all columns from all tables joined in the FROM clause - The keys for the join must be specified using ON. Most major flavors of SQL do not attempt to automatically identify keys like the join functions in R.

SQL is not the focus of this course, but let's do a quick exercise to practice writing a query.

Exercise 2

We will connect to the same projectdb database.

1. Retrieve sample and batch data (like the example above) for oxycodone (compound_name) and unknown samples (sample_type). Collect only the first 20 results.

```
SELECT

FROM

WHERE
```

2. Disconnect from the project database (hint: this is R code, not SQL).

End Exercise

The above examples and exercise are a very basic introduction to SQL. We will not cover more detail in this course because many more complicated queries are arguably better represented in R. If you primarily draw from dplyr for your data manipulation functions, R will translate your code into SQL automatically so there is limited need to learn SQL immediately yet still be able to take advantage of database functionality. However, having a solid understanding of SQL is helpful because much of the tidyverse functions and conventions are derived from core logical operations that are bread and butter SQL activites.

Keep in mind that there are actually a variety of implementations of SQL (based on different vendors, openly developed tools, etc.) that each have differences in syntax. Some examples include: - Microsoft SQL Server - PostgreSQL - MySQL - SQLite While many SQL commands and clauses are identical between SQL flavors, even some basic commands can vary dramatically. One example: the analogy of `head()` (i.e. return only the top n rows) is TOP() within the SELECT clause in Microsoft SQL Server and a separate LIMIT clause after other clauses in PostgreSQL.

Security Considerations

If you are working with sensitive data such as protected health information, security is a major consideration in interacting with databases. This is most relevant when interacting with sensitive data on a remote server, for which you have to supply credentials to R to establish a connection. **A general best practice is to avoid storing credentials (username, password) in plain text in your code.** This practice presents a particular risk if you are committing code to a repository that can be accessed remotely, but can also be an issue if your files are accessible to other users on the same system.

How do we set up our connection to avoid having to type out database usernames and passwords? There are a handful of ways to handle this, and will cover two explicitly.

The keyring package

Windows, Mac OS X, and Linux all have internal mechanisms to store credentials which we can take advantage of to store database credentials. The keyring package allows you to use your operating system password to access your database credentials, rather than having to remember multiple different usernames and passwords (this is a trickier problem if you work with multiple databases). You supply your database password once using the `key_set()` function, and then `key_list()` and `key_get()` functions retrieve your username and password, respectively. As long as your are signed into your operating system, you will be able to retrieve the credentials with those commands.

A sample connection call:

```
con <- dbConnect(odbc::odbc(),
  Driver   = "SQLServer",
  Server   = "my-database",
  Port     = 1433,
  Database = "default",
  UID      = keyring::key_list("my-database")[1,2], # format to retrieve username
  PWD      = keyring::key_get("my-database")) # retrieves password
```

This keying-based configuration is effective in situations where you are confident you will be signed in.

The config package

An alternative to storing credentials in the OS is to set up a configuration file that contains the database connection details that is not shared in a repository or with other users of the system. The config package allows you to create a config.yml file that contains a simple key:value pair structure with the necessary connection details.

An example file:

```
default:
  datawarehouse:
    driver: 'Postgres'
    server: 'mydb-test.company.com'
    uid: 'local-account'
    pwd: 'my-password'
    port: 5432
    database: 'regional-sales'
```

This data can be accessed using the `get()` function from the config package plus supplying generic connection details that reference the file.

```
dw <- config::get("datawarehouse")

con <- DBI::dbConnect(odbc::odbc(),
   Driver = dw$driver,
   Server = dw$server,
   UID    = dw$uid,
   PWD    = dw$pwd,
   Port   = dw$port,
   Database = dw$database
)
```

Using the config package can be a good option for automating connections to the dashboards. One security consideration with a configuration file is that other users on your server/system may be able to see your config.yml file unless you explicitly make it available only to yourself. It is a good idea to restrict the file to only allow yourself access to it. On Linux and Mac OS X, that can be done with `chmod 600 "filename"`.

Exercise 3

For the last exercise, we will reconnect to the publicly available database we viewed initially, but we will use the config package to connect.

1. Install the config package with `install.packages("config")`.
2. Create a config.yml file in the same directory as this R Markdown document and include the following info:

- host
- dbname
- port
- username
- password Note that exact names of the configuration fields are dependent on the driver (the example above is for a different type of connection than PostgreSQL).

3. Connect to the database and retrieve the first 20 entries of the rna table, similarly to what we retrieved in the original example.
4. Disconnect from the database.

End Exercise

Additional Resources

The content in this lesson captures an abbreviated version of RStudio's guide to connecting to databases. Please refer that resource to learn more about databases and R.

SQL is a very powerful tool in some settings because it is the primary route to retrieve data. So knowing some basics can unlock new data sources. There are a large number of resources online for learning SQL that can be pulled up by simplying searching for something along the lines of "learn SQL". One helpful resource that cuts across both theory and syntax is Stanford's openly-available, self-paced database course.

Summary

- Databases can provide better support than working with files when data sets are large or longitudinal data is collected over time.
- `dbConnect()` enables connections to databases but specific drivers are required for specific types of databases.
- Functions from dplyr can be translated to SQL to allow access to data without writing SQL queries.
- Security considerations are important for database connections, especially if sensitive information is stored - The keyring and config packages can support best practices for maintaining credentials.

Exploring lab order data

Overview of lesson activities

In this lesson we will gain more experience with some of the tools we have discussed throughout this course and ask you to dive into a new data set to answer a variety of questions. For many of the questions we will ask, there is no right or wrong way to answer the question. However, this is an opportunity to use new functions you have learned so far in this course. Our answers to the questions will primarily use tidyverse functions, but regardless of how you answer questions, you are looking for output of code to be the same.

Introduction to data set

The data set for this lesson is derived from orders for clinical laboratory tests in an electronic health record system in a set of outpatient clinics. The orders were deidentified and time-shifted (and approved for use as a teaching resource). There are two files: - "orders_data_set.xlsx" represents the data as one row per order and includes the bulk of the details - "order_details.csv" maintains the one row per order structure and include ancillary information about how a test was ordered

There are some column pairs with very similar names: one variable is a code ("_C") and the other is a description ("_C_DESCR"). This is largely done for covenience in querying the data or subsetting it without typing long strings. Because some may not be familiar with this type of data, we include a small data dictionary below to explain some of the data.

Variable	Description
Order ID	Key for order
Patient ID	Key for patient
Description	Text description of lab test
Proc Code	Procedure code for lab test
ORDER_CLASS_C_DESCR	Setting test is intended to be performed in (eg. Normal = regular blood draw)
LAB_STATUS_C	Status of laboratory result
ORDER_STATUS_C	Status of order
REASON_FOR_CANC_C	Cancellation reason (if applicable)
Order Time	Timestamp for time of original test order
Result Time	Timestamp for more recent result in the record
Review Time	Timestamp for provider acknowledgment of review of result
Department	Clinic associated with test order
ordering_route	Structure/menu in health record from which order was placed
pref_list_type	Category of preference list (if applicable)

Data import and preparation

We have a data set that is spread out over a couple files, with varying formats for variable names, and we want to consider what data types would be most appropriate for each of our variables. The overall goal of our analysis is to understand the metadata associated with this set of orders and identify any trends that would be useful in making changes to the electronic ordering and lab or clinic workflows. At this point it might be a little abstract because we are exploring the data but we know a few things we can address up front: - there are two files whose data could probably live in one data frame - the column names in the file have variable formatting - there are timestamps for which we may want to provide trends over time

Exercise 1

Let's work on addressing the above issues.

1. Import the data from each file
2. Clean variable names
3. Assess the relationship between the data in both files. Evaluate whether there is a one-to-one mapping, a many-to-one mapping, etc. (Hint: doing some exploration with various join functions can help answers quickly - helpful reference)

4. Consider which variables you may want to represent as factors (eg. for quick visual summaries) and convert
5. Assess the time span for orders and consider if there are specific time periods over which you may want to aggregate orders to view trends (eg. daily, weekly, monthly, yearly). Add additional variables to parse out these date components (and save yourself some work in the future). Refer to lesson 4 and lubridate documentation.
6. Summarize the data

End Exercise

Exploration of data

Let's take a high level look at the data, with some areas to explore:

- Overall orders over time - are there any dramatic changes in volume over the time period?
- Which tests are most commonly ordered?
- What is the overall cancellation rate and has it changed over time?

General hint for upcoming exercises: review documentation on janitor package and/or table function.

Exercise 2

1. Plot the order volume over the duration of the data set, at the level of day and week. (Keep in mind how ggplot parses time some geoms - it may be based on seconds.)

2. Plot or tabulate the breakdown of test orders in the data set (using description or procedure code). Use the `slice_head()` function to restrict the output to the top 25 most commonly ordered tests. (Review the help documentation for `slice()` and related functions.)

3. Plot and/or tabulate cancellations over time for the data set.

4. Explore whether there are specific tests that are cancelled more frequently than others. Restrict the output to the top 25 tests.

End Exercise

Answering clinic-specific questions

Based on some preliminary analysis and past knowledge, we want to dig into clinic-specific practices for ordering tests.

Exercise 3

The following is a list of questions regarding clinic-specific characteristics of orders that we would like to answer:

- Which clinics order the highest volume of tests?
- Which clinics have the highest numbers and rates of test cancellation?
- Are there any clinics collecting blood at the clinic as opposed to at blood draw?
- Which clinics are using SmarSets (order sets) most extensively?
- Which clinics continue to use Provider Preference Lists, which are discouraged?

End Exercise

Evaluating turnaround times for result review

Unfortunately this data set is missing crucial timestamps needed to assess lab turnaround times. Assessing the time between order and result might be interesting, but there are various workflow variations that make this difficult to interpret. What is more straightforward to interpret, however, is the duration between when a test is resulted and when that result is reviewed by the repsonsible provider. We do not have provider identifiers in this data set, but we can still assess the result-to-review turnaround time by clinic.

Exercise 4

Develop a visualization that shows the distribution of different result-to-review intervals, separated by clinic.

End Exercise

```
## Parsed with column specification:
## cols(
##   order_id = col_double(),
##   ordering_route = col_character(),
##   pref_list_type = col_character()
## )

## left_join: added 2 columns (ordering_route, pref_list_type)

##            > rows only in x        0

##            > rows only in y  (     0)

##            > matched rows     45,002

##            >                ========

##            > rows total       45,002

## mutate: converted 'description' from character to factor (0 new NA)

##         converted 'proc_code' from character to factor (0 new NA)

##         converted 'order_class_c_descr' from character to factor (0 new NA)

##         converted 'lab_status_c_descr' from character to factor (0 new NA)

##         converted 'order_status_c_descr' from character to factor (0 new NA)

##         converted 'reason_for_canc_c_descr' from character to factor (0 new NA)

##         converted 'department' from character to factor (0 new NA)

##         converted 'ordering_route' from character to factor (0 new NA)

##         converted 'pref_list_type' from character to factor (0 new NA)
```

```
## mutate: new variable 'order_month' (double) with 4 unique values and 0% NA

##         new variable 'order_week' (double) with 13 unique values and 0% NA

##     order_id       patient_id                              description
##  Min.   : 10002   Min.   :500001   COMPREHENSIVE METABOLIC PANEL: 3639
##  1st Qu.: 32669   1st Qu.:503350   HEMOGLOBIN A1C, HPLC         : 2470
##  Median : 55246   Median :506862   CBC, DIFF                    : 2393
##  Mean   : 55133   Mean   :506897   BASIC METABOLIC PANEL        : 2174
##  3rd Qu.: 77627   3rd Qu.:510421   GC&CHLAM NUCLEIC ACID DETECTN: 2164
##  Max.   :100000   Max.   :513993   CBC (HEMOGRAM)               : 1979
##                                    (Other)                      :30183
##    proc_code         order_class_c_descr  lab_status_c
##  COMP   : 3639   Clinic Collect: 6427    Min.   :1.000
##  A1C    : 2470   External      :  401    1st Qu.:3.000
##  CBD    : 2393   Historical    :    5    Median :3.000
##  BMP    : 2174   Normal        :36326    Mean   :3.061
##  GCCTAD : 2164   On Site       : 1843    3rd Qu.:3.000
##  CBC    : 1979                           Max.   :5.000
##  (Other):30183                           NA's   :7152
##             lab_status_c_descr order_status_c  order_status_c_descr
##  Edited Result - FINAL: 1238    Min.   :2.000   Canceled : 9270
##  Final result         :36508    1st Qu.:5.000   Completed:35553
##  In process           :   81    Median :5.000   Sent     :  161
##  Preliminary result   :   23    Mean   :4.783   NA's     :   18
##  NA's                 : 7152    3rd Qu.:5.000
##                                 Max.   :5.000
##                                 NA's   :18
##  reason_for_canc_c
##  Min.   :   1.0
##  1st Qu.:  11.0
##  Median :  11.0
##  Mean   : 437.2
##  3rd Qu.:1178.0
##  Max.   :1178.0
##  NA's   :37794
##                                                                  reason_for_canc_c_descr
##  Auto-canceled. Patient no show and/or specimen not received within 60 days.: 4337
##  Canceled by Lab, see Result History.                                       : 2255
##  Cancel, order changed                                                      :  118
##  Auto-canceled, specimen not received within 14 days                        :   90
##  Error                                                                      :   67
##  (Other)                                                                    :  341
##  NA's                                                                       :37794
##    order_time                   result_time
##  Min.   :2017-08-13 11:59:00   Min.   :2017-06-15 00:00:00
##  1st Qu.:2017-09-05 11:16:00   1st Qu.:2017-09-07 12:51:00
##  Median :2017-09-27 08:48:00   Median :2017-09-29 14:06:30
##  Mean   :2017-09-27 09:39:30   Mean   :2017-09-30 17:11:17
##  3rd Qu.:2017-10-19 13:45:00   3rd Qu.:2017-10-23 18:39:30
##  Max.   :2017-11-11 19:49:00   Max.   :2017-12-29 07:37:00
##                                NA's   :7152
##   review_time                                        department
##  Min.   :2017-08-15 09:16:00   INFECTIOUS DISEASE CLINIC    :11861
```

```
##  1st Qu.:2017-09-15 23:32:30   INTERNAL MEDICINE CLINIC     : 6330
##  Median :2017-10-12 14:22:00   FAMILY MEDICINE CLINIC       : 3501
##  Mean   :2017-10-11 06:52:47   NEIGHBORHOOD CLINIC          : 3128
##  3rd Qu.:2017-11-02 09:39:00   INTERNATIONAL MEDICINE CLINIC: 2499
##  Max.   :2017-12-29 22:24:00   RHEUMATOLOGY CLINIC          : 2422
##  NA's   :7791                  (Other)                      :15261
##              ordering_route                   pref_list_type   order_month
##  Clinician Orders   : 4174   Clinic Preference List  :30483   Min.   : 8.000
##  External Order     :    5   None                    : 6422   1st Qu.: 9.000
##  OP Orders Navigator:36541   Provider Preference List: 8097   Median : 9.000
##  Results Console    :  179                                    Mean   : 9.351
##  SmartSet           : 4101                                    3rd Qu.:10.000
##  NA's               :    2                                    Max.   :11.000
##
##    order_week
##  Min.   :33.00
##  1st Qu.:36.00
##  Median :39.00
##  Mean   :38.98
##  3rd Qu.:42.00
##  Max.   :45.00
##
```

Using nested data frames to scale analyses

The orders data set is a good one to introduce the concepts of nested data frames and list-columns. These features in R help with automating analyses and keeping outputs together.

A nested data frame is one that includes a list-column of data frames. In this format, each row is a meta-observation, meaning the data are held in columns that define the observation and one or more list-columns of data frames that hold the individual data components. This may be easier to understand by comparing our orders data set in a nested and unnested format.

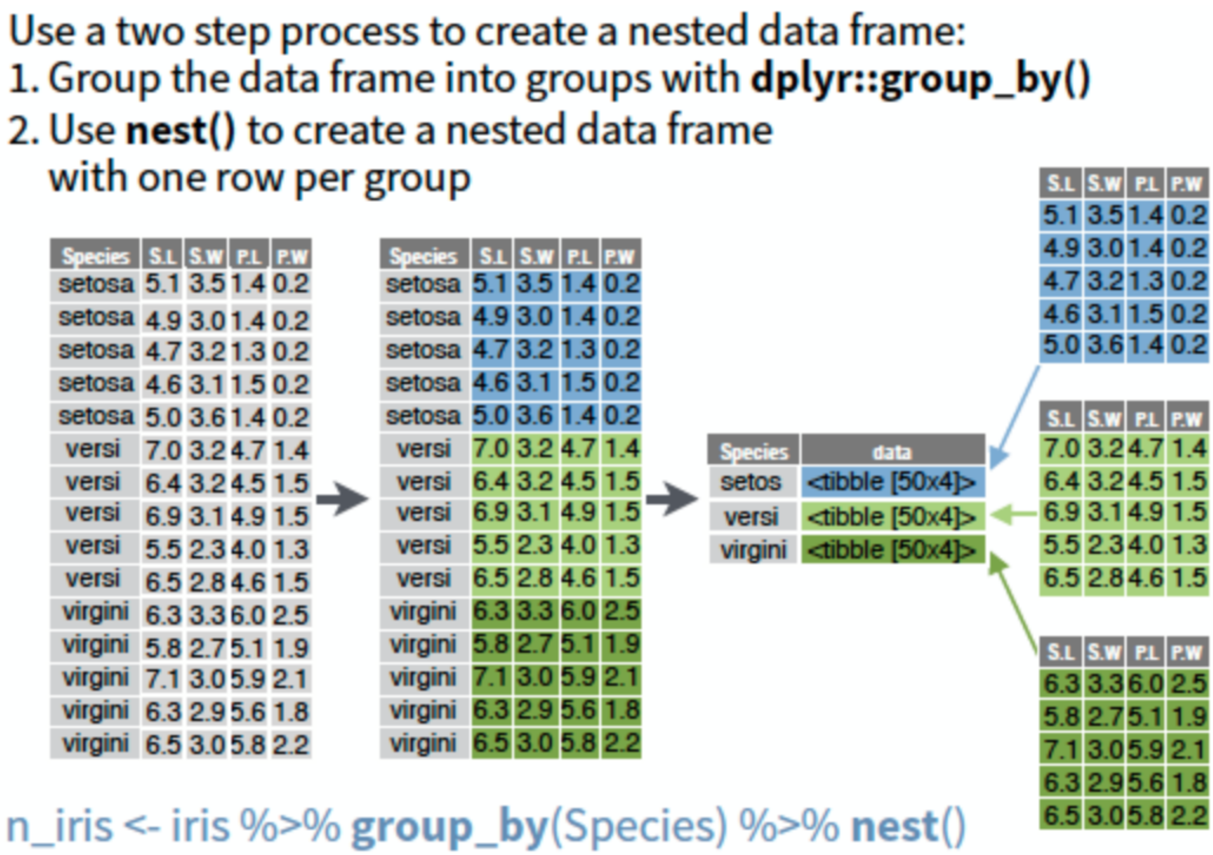
The orders data set has 45002 rows of 20 variables. So far, we've been exploring the data set as a whole. However, we've seen that there are differences among some facets of the data, such as department. It may be useful then to group the data by department and perform the same analyses for each department. This can be done by copying and pasting code and substituting department name for each of the 20 departments. This may be manageable for small numbers of groups, BUT, this is the 201 R class, so we're going to learn how to use R to automate this type of work, so you can scale such analyses easily. We'll see how data in nested or list-column format combined with the map functions we learned about earlier is well designed for this.

The syntax is relatively straightforward and similar to what we've seen previously when we grouped by variables and summarized. We first group our data by a selected variable or variables and pass this grouped data frame to `nest()`. Let's see what this nested data frame looks like.

```
orders_nest <- orders %>%
                group_by(department) %>%
                nest()
```

```
## group_by: one grouping variable (department)
```

You should see two columns. One for the department and the second is a list-column that contains all the data for this department. Since there are 20 unique departments, there are 20 rows in the nested data frame.



We can access individual components from our nested data frame using [[]] or $ selector notation.

```
orders_nest$data # all 20 data frames in the data column
```

```
## [[1]]
## # A tibble: 6,330 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    19766     511388 PROTHROMBI~ PRO       Normal                     NA
##  2    88444     511388 BASIC META~ BMP       Normal                     NA
##  3    50728     501184 COMPREHENS~ COMP      Normal                     NA
##  4    91635     501184 CBC (HEMOG~ CBC       Normal                     NA
##  5    23789     507392 CHR PAIN D~ UCPD1B    Normal                      3
##  6    17359     513008 CULTURE:VI~ VCIR      Normal                     NA
##  7    22570     501142 CHR PAIN D~ UCPD1B    Normal                      3
##  8    51714     513163 URIC ACID,~ URIC      Normal                      3
##  9    31718     513163 BASIC META~ BMP       Normal                      3
## 10    73740     513163 CBC, DIFF   CBD       Normal                      3
## # ... with 6,320 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[2]]
## # A tibble: 1,486 x 18
```

```
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    40477     508061 THYROID ST~ TSH       Normal                      3
##  2    97641     508061 T4, FREE    T4FR      Normal                      3
##  3    75867     504805 GLUCOSE, W~ 82962     On Site                     3
##  4    75528     510431 THYROID ST~ TSH       Normal                      3
##  5    98672     510431 T4, FREE    T4FR      Normal                      3
##  6    84224     511065 GLUCOSE, W~ 82962     On Site                     3
##  7    80303     507712 GLUCOSE, W~ 82962     On Site                     3
##  8    77270     508305 ANTI THYRO~ ATPO2     Normal                     NA
##  9    59682     503047 GLUCOSE, W~ 82962     On Site                     3
## 10    95169     503047 HEMOGLOBIN~ A1CRPD    Normal                      3
## # ... with 1,476 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[3]]
## # A tibble: 601 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    99868     505646 COMPREHENS~ COMP      Normal                      3
##  2    31178     505646 GLUCOSE SE~ GLUF      Normal                      3
##  3    87245     505646 HEMOGLOBIN~ A1C       Normal                      3
##  4    50160     505646 LIPID PANEL LIPID     Normal                      3
##  5    99743     508791 LIPID PANEL LIPID     Normal                      3
##  6    93938     508791 HEMOGLOBIN~ A1C       Normal                      3
##  7    87463     508791 COMPREHENS~ COMP      Normal                      3
##  8    78615     510909 APOLIPOPRO~ APOB      Normal                      3
##  9    64333     510909 THYROID ST~ TSH       Normal                      3
## 10    61621     510909 HEMOGLOBIN~ A1C       Normal                      3
## # ... with 591 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[4]]
## # A tibble: 1,026 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    26516     511303 URIC ACID,~ URIC      Normal                      3
##  2    47649     511303 STANDARD D~ UDRSS     Normal                      5
##  3    10968     511303 THYROID ST~ TSH       Normal                      3
##  4    55526     511303 BASIC META~ BMP       Normal                      3
##  5    58870     511303 HEPATIC FU~ HFPA      Normal                      3
##  6    45209     511303 IRON BINDI~ IBCD      Normal                      3
##  7    34373     511303 IRON, SERUM FE        Normal                      5
##  8    44060     511303 B_TYPE NAT~ BNAP      Normal                      3
##  9    23368     511303 URINALYSIS~ UAC       Normal                      3
## 10    95109     511303 FERRITIN    FER       Normal                      3
## # ... with 1,016 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
```

```
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[5]]
## # A tibble: 2,093 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    79887     510902 CBC (HEMOG~ CBC       Normal                     NA
##  2    17602     507405 RENAL FUNC~ RENFP     Normal                      3
##  3    41429     507405 RENAL FUNC~ RENFP     Normal                      3
##  4    53695     510463 U/A NONAUT~ 81002     On Site                     3
##  5    29415     503167 PARATHYROI~ IPTH      Normal                      3
##  6    49907     503167 URINE PROT~ UPCRAT    Normal                      3
##  7    13350     503167 RENAL FUNC~ RENFP     Normal                     NA
##  8    79131     504256 MAGNESIUM   MG        Normal                     NA
##  9    74554     509817 BASIC META~ BMP       Normal                     NA
## 10    81892     509758 VITAMIN D ~ VITDG2    Normal                      3
## # ... with 2,083 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[6]]
## # A tibble: 3,501 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    42783     506703 HEPATITIS ~ HCAB      Normal                      3
##  2    11884     506703 HEPATITIS ~ HAVIGM    Normal                      3
##  3    98932     506703 HEPATITIS ~ HBSS      Normal                      3
##  4    83642     506703 HEPATITIS ~ HAVIGG    Normal                      3
##  5    12236     502031 U/A NONAUT~ 81002     On Site                     3
##  6    59945     511013 CBC, DIFF   CBD       Clinic Collect              3
##  7    33517     501790 U/A NONAUT~ 81002     On Site                    NA
##  8    90317     513135 GLUCOSE, W~ 82962     On Site                    NA
##  9    28414     513135 THYROID ST~ TSH       Clinic Collect              3
## 10    66783     513135 CHOLESTERO~ CHOL      Normal                      3
## # ... with 3,491 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[7]]
## # A tibble: 3,128 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    22780     503847 HEMOGLOBIN~ A1C       Clinic Collect              3
##  2    24316     503847 BASIC META~ BMP       Clinic Collect              3
##  3    24503     503847 LIPID PANEL LIPID     Clinic Collect              3
##  4    97595     506119 CBC, DIFF   CBD       Clinic Collect              3
##  5    15073     506119 BASIC META~ BMP       Clinic Collect              3
##  6    78705     503144 CBC, DIFF   CBD       Clinic Collect              3
```

```
## 7    91697     503144 RENAL FUNC~ RENFP     Clinic Collect              3
## 8    99656     508435 GLUCOSE, W~ 82962     On Site                     3
## 9    24085     511453 U/A NONAUT~ 81002     On Site                     3
## 10   90665     511453 GLUCOSE, W~ 82962     On Site                     3
## # ... with 3,118 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[8]]
## # A tibble: 503 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    99523     510460 PROTHROMBI~ PRO       Normal                      3
##  2    26411     501568 GROUP A ST~ BSARD     Normal                      3
##  3    14980     501568 R/O BETA S~ BSC       Normal                      3
##  4    33230     501568 LAB ADD ON~ LADDON    Normal                      3
##  5    34386     511328 HEMOGLOBIN~ A1C       Normal                      3
##  6    63108     511328 LIPID PANEL LIPID     Normal                      3
##  7    94945     511328 URINE SCRE~ UMALSP    Normal                      3
##  8    10561     511340 FERRITIN    FER       Normal                      3
##  9    86003     511340 IRON BINDI~ IBCD      Normal                      3
## 10    88863     511340 CBC (HEMOG~ CBC       Normal                      3
## # ... with 493 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[9]]
## # A tibble: 503 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    83813     504593 CHR PAIN D~ UCPD1B    Clinic Collect              3
##  2    66540     502678 CHR PAIN D~ UCPD1B    Clinic Collect              3
##  3    81501     501080 CHR PAIN D~ UCPD1B    Clinic Collect              3
##  4    57781     508385 CHR PAIN D~ UCPD1B    Clinic Collect              3
##  5    27831     509043 CHR PAIN D~ UCPD1B    Clinic Collect              3
##  6    12660     513930 CHR PAIN D~ UCPD1B    Clinic Collect              3
##  7    73489     504215 CHR PAIN D~ UCPD1B    Clinic Collect              3
##  8    30170     503959 OPIOID CON~ UOPIAC    Clinic Collect              3
##  9    55092     510557 CHR PAIN D~ UCPD1B    Clinic Collect              3
## 10    78907     512761 CHR PAIN D~ UCPD1B    Clinic Collect              3
## # ... with 493 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[10]]
## # A tibble: 2,211 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
```

```
## 1    61321     512524 HEPATIC FU~ HFPA      Normal                        1
## 2    63568     513876 COMPREHENS~ COMP      Normal                        3
## 3    29849     513876 HEPATITIS ~ HCVQNT    Normal                       NA
## 4    46095     513876 HEPATITIS ~ HBSS      Normal                       NA
## 5    39759     513876 HEPATITIS ~ HAVIGG    Normal                       NA
## 6    80259     513876 PROTHROMBI~ PRO       Normal                        3
## 7    71530     513876 CBC, DIFF   CBD       Normal                        3
## 8    37697     510569 HEPATITIS ~ HBSS      Normal                        3
## 9    14623     510569 COMPREHENS~ COMP      Normal                        3
## 10   44648     510569 HEPATITIS ~ HAVIGG    Normal                        3
## # ... with 2,201 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[11]]
## # A tibble: 2,179 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
## 1    55347     510095 URINE PREG~ 81025     On Site                       3
## 2    27773     511000 URINE PREG~ 81025     On Site                       3
## 3    43511     511000 PATHOLOGY,~ SURG      Clinic Collect                3
## 4    80696     501931 WET MOUNTS~ Q0111     On Site                       3
## 5    21481     501931 R/O YEAST ~ YSTF      Clinic Collect                3
## 6    22998     510095 PATHOLOGY,~ SURG      Clinic Collect                3
## 7    47686     502539 PREGNANCY ~ UPG       Normal                        3
## 8    11078     506835 17-HYDROXY~ OHPROG    Normal                        3
## 9    81410     506835 DHEA SULFA~ DHEAS     Normal                        3
## 10   51073     506835 TESTOSTERO~ FTTEST    Normal                        3
## # ... with 2,169 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[12]]
## # A tibble: 724 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
## 1    86858     510501 BASIC META~ BMP       Normal                        3
## 2    53879     510501 URINALYSIS~ UACRC     Normal                        3
## 3    54707     510501 CBC (HEMOG~ CBC       Normal                        3
## 4    17821     510501 PROTHROMBI~ PRO       Normal                        3
## 5    55146     510501 PARTIAL TH~ PTT       Normal                        3
## 6    68140     511149 CRP, HIGH ~ HSCRP     Normal                        3
## 7    23688     511149 CBC (HEMOG~ CBC       Normal                        3
## 8    82147     511149 SED RATE    ESR       Normal                        3
## 9    77846     504561 URINALYSIS~ UACRC     Normal                        3
## 10   95958     504561 BASIC META~ BMP       Normal                        3
## # ... with 714 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
```

```
## #   order_month <dbl>, order_week <dbl>
##
## [[13]]
## # A tibble: 11,861 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    48260     508653 COMPREHENS~ COMP      Normal                      3
##  2    53921     508653 CBC, DIFF   CBD       Normal                      3
##  3    58381     508653 SEROLOGIC ~ SYPHS     Normal                      5
##  4    17124     508653 GC&CHLAM N~ GCCTAD    Normal                      3
##  5    98089     508653 GC&CHLAM N~ GCCTAD    Normal                      3
##  6    87763     508653 GC&CHLAM N~ GCCTAD    Normal                      3
##  7    22841     511004 HIV ANTIGE~ HVAGAB    Normal                     NA
##  8    74600     511004 HEPATITIS ~ HBSA      Normal                     NA
##  9    61276     505483 BASIC META~ BMP       Normal                     NA
## 10    79148     505483 ANTI TOXOP~ TOXOG     Normal                     NA
## # ... with 11,851 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[14]]
## # A tibble: 1,015 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    41411     506437 PTH-RELATE~ RPTHRP    Normal                      3
##  2    65645     506437 VITAMIN D ~ VITDG2    Normal                      3
##  3    64871     506437 HEMOGLOBIN~ A1C       Normal                      3
##  4    57461     508851 BASIC META~ BMP       Normal                      3
##  5    32532     512508 LIPID PANEL LIPID     Normal                      3
##  6    44536     512508 THYROID ST~ TSH       Normal                      3
##  7    20962     512948 CHRONIC PA~ UCPDS     Normal                     NA
##  8    35222     512508 CHR PAIN D~ UCPD1B    Normal                      3
##  9    94655     500485 THYROID ST~ TSH       Normal                      3
## 10    63171     500485 HEMOGLOBIN~ A1C       Normal                      3
## # ... with 1,005 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[15]]
## # A tibble: 2,499 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    50012     501660 LIPID PANEL LIPID     Normal                      3
##  2    38223     501660 CBC, DIFF   CBD       Normal                      3
##  3    36299     501660 URIC ACID,~ URIC      Normal                      3
##  4    47819     501660 COMPREHENS~ COMP      Normal                      3
##  5    66313     501660 VITAMIN D ~ VITDG2    Normal                      3
##  6    37682     501203 VITAMIN D ~ VITDG2    Normal                      3
##  7    43926     501203 COMPREHENS~ COMP      Normal                      3
##  8    11326     501203 LIPID PANEL LIPID     Normal                      3
```

```
## 9    67843     501203 CBC, DIFF   CBD       Normal                        3
## 10   30756     502138 CBC, DIFF   CBD       Normal                        3
## # ... with 2,489 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[16]]
## # A tibble: 902 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    24866     503976 URINE PREG~ 81025     On Site                     3
##  2    44996     510987 VITAMIN D ~ VITDG2    Normal                      3
##  3    90492     510987 BASIC META~ BMP       Normal                      3
##  4    73371     510987 HEMOGLOBIN~ A1C       Normal                      3
##  5    81208     510987 CBC, DIFF   CBD       Normal                      3
##  6    11015     510814 VITAMIN D ~ VITDG2    Normal                      3
##  7    35768     510814 CBC, DIFF   CBD       Normal                      3
##  8    98020     510814 ZINC PROTO~ ZPPH      Normal                      3
##  9    75804     510814 LEAD        PB        Normal                      3
## 10    97951     510814 HIV ANTIGE~ HVAGAB    Normal                      3
## # ... with 892 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[17]]
## # A tibble: 781 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    50337     508808 CBC, DIFF   CBD       Normal                     NA
##  2    35868     508808 TYPE AND S~ TSCR      Normal                     NA
##  3    98477     505878 HEPATITIS ~ HAVIGG    Normal                      3
##  4    14470     505878 HEPATITIS ~ HBCA      Normal                      3
##  5    53160     505878 HEPATITIS ~ HBSA      Normal                      3
##  6    49114     509403 HEPATIC FU~ HFPA      Normal                     NA
##  7    10492     508361 HEPATITIS ~ HBSA      Normal                      3
##  8    19794     508361 HEPATITIS ~ HBCA      Normal                      3
##  9    12509     508361 HEPATITIS ~ HAVIGG    Normal                      3
## 10    46062     503197 CRP, HIGH ~ HSCRP     Normal                     NA
## # ... with 771 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[18]]
## # A tibble: 657 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    66536     506248 COMPREHENS~ COMP      Normal                      3
##  2    12174     506248 CBC, DIFF   CBD       Normal                      3
```

```
## 3     66374     509724 CBC, DIFF   CBD       Normal                        3
## 4     97052     509724 COMPREHENS~ COMP      Normal                        3
## 5     91016     508759 CBC (HEMOG~ CBC       Normal                        3
## 6     78403     508759 COMPREHENS~ COMP      Normal                        3
## 7     45997     509823 CBC, DIFF   CBD       Normal                        3
## 8     78835     509823 COMPREHENS~ COMP      Normal                        3
## 9     89187     505974 MITOCHONDR~ MITPAN    Normal                        3
## 10    96199     509444 LYME DISEA~ RLYMD     Normal                        3
## # ... with 647 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[19]]
## # A tibble: 2,422 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
## 1     39794     508604 CELL COUNT~ FCCNT     Normal                        3
## 2     73174     507693 URINALYSIS~ UAC       Normal                        3
## 3     71219     507693 URINE C/S   URNXC     Normal                       NA
## 4     82705     513123 SED RATE    ESR       Normal                       NA
## 5     77427     513123 COMPREHENS~ COMP      Normal                       NA
## 6     83585     513123 CBC, DIFF   CBD       Normal                       NA
## 7     33707     513123 CRP, HIGH ~ HSCRP     Normal                       NA
## 8     38398     510005 CRP, HIGH ~ HSCRP     Normal                        3
## 9     57991     510005 CBC, DIFF   CBD       Normal                        3
## 10    62417     510005 COMPREHENS~ COMP      Normal                        3
## # ... with 2,412 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
##
## [[20]]
## # A tibble: 580 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
## 1     73273     501740 TACROLIMUS~ TACROG    Normal                        3
## 2     56230     501740 COMPREHENS~ COMP      Normal                        3
## 3     88922     501740 CBC, DIFF   CBD       Normal                        3
## 4     50170     501740 MAGNESIUM   MG        Normal                        3
## 5     13215     513015 OCCULT BLO~ SOCULT    Normal                       NA
## 6     40011     513015 CBC, DIFF   CBD       Normal                        3
## 7     62529     513015 COMPREHENS~ COMP      Normal                        3
## 8     47408     513015 PROTEIN EL~ ELPP      Normal                        3
## 9     47865     513015 FIBRINOGEN  FIBCL     Normal                        3
## 10    13639     513015 CBC, DIFF   CBD       Normal                        3
## # ... with 570 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
```

```r
orders_nest[[2]][[1]] # data frame for first location
```

```
## # A tibble: 6,330 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    19766     511388 PROTHROMBI~ PRO       Normal                     NA
##  2    88444     511388 BASIC META~ BMP       Normal                     NA
##  3    50728     501184 COMPREHENS~ COMP      Normal                     NA
##  4    91635     501184 CBC (HEMOG~ CBC       Normal                     NA
##  5    23789     507392 CHR PAIN D~ UCPD1B    Normal                      3
##  6    17359     513008 CULTURE:VI~ VCIR      Normal                     NA
##  7    22570     501142 CHR PAIN D~ UCPD1B    Normal                      3
##  8    51714     513163 URIC ACID,~ URIC      Normal                      3
##  9    31718     513163 BASIC META~ BMP       Normal                      3
## 10    73740     513163 CBC, DIFF   CBD       Normal                      3
## # ... with 6,320 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
```

```r
orders_nest$data[[1]] # data frame for the first location
```

```
## # A tibble: 6,330 x 18
##    order_id patient_id description proc_code order_class_c_d~ lab_status_c
##       <dbl>      <dbl> <fct>       <fct>     <fct>                   <dbl>
##  1    19766     511388 PROTHROMBI~ PRO       Normal                     NA
##  2    88444     511388 BASIC META~ BMP       Normal                     NA
##  3    50728     501184 COMPREHENS~ COMP      Normal                     NA
##  4    91635     501184 CBC (HEMOG~ CBC       Normal                     NA
##  5    23789     507392 CHR PAIN D~ UCPD1B    Normal                      3
##  6    17359     513008 CULTURE:VI~ VCIR      Normal                     NA
##  7    22570     501142 CHR PAIN D~ UCPD1B    Normal                      3
##  8    51714     513163 URIC ACID,~ URIC      Normal                      3
##  9    31718     513163 BASIC META~ BMP       Normal                      3
## 10    73740     513163 CBC, DIFF   CBD       Normal                      3
## # ... with 6,320 more rows, and 12 more variables: lab_status_c_descr <fct>,
## #   order_status_c <dbl>, order_status_c_descr <fct>, reason_for_canc_c <dbl>,
## #   reason_for_canc_c_descr <fct>, order_time <dttm>, result_time <dttm>,
## #   review_time <dttm>, ordering_route <fct>, pref_list_type <fct>,
## #   order_month <dbl>, order_week <dbl>
```

To go back to a flat data frame, we use `unnest()` and specify which column we want to unnest, in this case it is the data column.

```r
orders_unnested <- orders_nest %>% unnest(data) # looks just like orders
```

Back to our nested data frame, we now have data separated in a way that makes it easy to apply the same analysis across the data for all departments individually. Similar to previous lessons using `summarize()` and the different `map()` functions, we need a function to apply to our data. In the previous cases, we used existing functions in R: `mean()`, `read_excel()`. Here, we will review how to create our own function and map this across our list-column.

We saw above that there are differences in use of the discouraged provider preference lists. If we want to know how these vary by department, in more detail, we can create a table similar to those above, but for each department.

This table shows the top 10 orders most frequently ordered from a provider preference list by department and calculates the percent usage of the provider preference list for each test.

```
orders %>%
  filter(department == "ENDOCRINOLOGY CLINIC") %>%
  tabyl(description, pref_list_type) %>%
  arrange(desc(`Provider Preference List`)) %>%
  slice_head(n = 10) %>%
  adorn_totals("row") %>%
  adorn_percentages("row") %>%
  adorn_pct_formatting()
```

```
## filter: removed 43,516 rows (97%), 1,486 rows remaining

## slice_head: removed 547 rows (98%), 10 rows remaining

##                      description Clinic Preference List  None
##      THYROID STIMULATING HORMONE                   8.2% 12.1%
##                         T4, FREE                  10.6%  9.3%
##               HEMOGLOBIN A1C, HPLC                26.6%  5.5%
##                               T3                   8.1%  4.1%
##                      LIPID PANEL                   7.9%  4.8%
##        TESTOSTERONE, FREE & TOTAL                  6.9%  0.0%
##   URINE SCREEN, MICROALBUMINURIA                  50.0% 12.9%
##    COMPREHENSIVE METABOLIC PANEL                  32.1%  0.0%
##                        ESTRADIOL                   9.1%  4.5%
##            VITAMIN D (25 HYDROXY)                 30.8%  3.8%
##                            Total                  16.7%  7.7%
##  Provider Preference List
##                     79.7%
##                     80.1%
##                     68.0%
##                     87.8%
##                     87.3%
##                     93.1%
##                     37.1%
##                     67.9%
##                     86.4%
##                     65.4%
##                     75.6%
```

Great - so, we could iterate over all 20 departments manually by changing the department in the `filter()` call – or we can automate this process. First we will build a function that creates this table for whatever dataset is supplied to it. We make small modifications to the code above to create our function:

1. We need to identify that we are writing a function - use `function(){}`

2. Within the () supply the name for a data argument, in this case df, as we will be passing a data frame to our function.

3. Within the {} we write the code we want carried out. This should be the code from above with a change to the input data. Since we will be iterating over our list column that is specific to a department, we don't need the filter step.

4. We execute the entire code chunk to save our new function to our environment.

```
pro_pref_sum <- function(df) {
  df %>%
  tabyl(description, pref_list_type) %>%
    arrange(desc(`Provider Preference List`)) %>%
    slice_head(n = 10) %>%
    adorn_totals("row") %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting()
}
```

Now we have our user defined function ready. We can test that it performs as expected by trying on data for of the departments.

```
pro_pref_sum(orders_nest$data[[2]]) # look at endocrinology
```

```
## slice_head: removed 547 rows (98%), 10 rows remaining
```

```
##                      description Clinic Preference List  None
##     THYROID STIMULATING HORMONE                    8.2% 12.1%
##                        T4, FREE                   10.6%  9.3%
##             HEMOGLOBIN A1C, HPLC                  26.6%  5.5%
##                               T3                   8.1%  4.1%
##                      LIPID PANEL                   7.9%  4.8%
##      TESTOSTERONE, FREE & TOTAL                    6.9%  0.0%
##  URINE SCREEN, MICROALBUMINURIA                   50.0% 12.9%
##    COMPREHENSIVE METABOLIC PANEL                  32.1%  0.0%
##                        ESTRADIOL                   9.1%  4.5%
##           VITAMIN D (25 HYDROXY)                  30.8%  3.8%
##                            Total                  16.7%  7.7%
##  Provider Preference List
##                     79.7%
##                     80.1%
##                     68.0%
##                     87.8%
##                     87.3%
##                     93.1%
##                     37.1%
##                     67.9%
##                     86.4%
##                     65.4%
##                     75.6%
```

We will use `map()` to apply our function to each of our data frames in the data list column, generating a table for each department.

Remember the syntax is: map(.x, .f). We'll create a new column to hold the data for our summary tables.

```
orders_nest <- orders_nest %>%
              mutate(pro_pref = map(data, pro_pref_sum))
```

```
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
## slice_head: removed 547 rows (98%), 10 rows remaining
```

```
## mutate (grouped): new variable 'pro_pref' (list) with 20 unique values and 0% NA
```

Our data are now stored with the analyses. Hopefully you can imagine the power of this skill and the ability to easily apply functions (built-in or user-defined) across features of a data set.

Exercise 5

Let's do an exercise to create a plot for each department to show the distribution of the result review times.

Write a function that creates a density plot for the review TAT and apply it to the data for each department. Add the plots as a new column in your nested data set.

End Exercise

Predictions using linear regression

Overview of data

Though we commonly think about linear regression in the context of calibrations or method comparisons, it is also a widely applied tool for predictive modeling. In this lesson we will use data from a targeted metabolomics experiment in children with chronic kidney disease to build a linear model that predicts their glomerular filtration rate (GFR). This data is provided in two files. One has values for the outcome (GFR) for each subject ID and the other includes values for several predictors (e.g., creatinine, BUN, various endogenous metabolites) measured for each subject ID.

We will need to use our previously learned skills to read in the data and join the two sets by subject.

```
#load in CKD_data.csv and CKD_GFR.csv
data <- read_csv("data/CKD_data.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   scr = col_double(),
##   bun = col_double(),
##   cyc_db = col_double(),
##   Albumin = col_double(),
##   upcratio = col_double(),
##   ADMA = col_double(),
##   SDMA = col_double(),
##   Creatinine = col_double(),
##   Kynurenine = col_double(),
##   Trp = col_double(),
##   Phe = col_double(),
##   Tyr = col_double()
## )
```

```
glimpse(data)
```

```
## Rows: 200
## Columns: 13
## $ id         <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1...
## $ scr        <dbl> 1.93, 2.72, 0.88, 0.78, 1.28, 1.41, 0.58, 1.34, 3.42, 2....
## $ bun        <dbl> 21, 37, 12, 27, 27, 19, 17, 13, 45, 39, 80, 52, 41, 23, ...
## $ cyc_db     <dbl> 0.82, 1.90, 0.65, 1.31, 1.50, 1.60, 1.04, 1.04, 3.04, 2....
## $ albumin    <dbl> 4.1, 4.2, 4.2, 4.4, 3.6, 4.2, 4.5, 3.8, 4.1, 4.0, 4.2, 5...
## $ upcratio   <dbl> 1.39, 0.38, 0.33, 0.26, 0.57, 0.04, 0.12, 5.17, 0.26, 0....
## $ adma       <dbl> 0.41, 0.69, 0.57, 0.39, 0.87, 0.48, 0.52, 1.14, 0.46, 0....
## $ sdma       <dbl> 0.70, 1.45, 0.49, 0.33, 2.31, 0.74, 0.46, 1.42, 0.96, 1....
## $ creatinine <dbl> 138.61, 274.34, 78.81, 63.05, 226.21, 150.50, 70.84, 176...
## $ kynurenine <dbl> 3.98, 7.35, 1.76, 2.41, 5.91, 4.29, 3.19, 6.18, 8.08, 6....
## $ trp        <dbl> 78.18, 45.02, 58.62, 34.64, 62.36, 52.21, 49.28, 101.75,...
## $ phe        <dbl> 79.45, 110.28, 74.47, 39.81, 75.40, 58.66, 53.82, 93.30,...
## $ tyr        <dbl> 77.00, 79.61, 65.22, 44.55, 94.24, 60.66, 68.07, 110.73,...
```

```
gfr <- read_csv("data/CKD_GFR.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   igfrc = col_double()
## )
```

```
glimpse(gfr)
```

```
## Rows: 200
## Columns: 2
## $ id    <dbl> 498, 128, 13, 2, 183, 197, 78, 174, 91, 123, 168, 41, 130, 10...
## $ igfrc <dbl> 24, 18, 20, 21, 21, 21, 21, 21, 21, 22, 23, 23, 23, 23, 23, 2...
```

```r
#join by ID, convert ID to factor
ckd <- left_join(gfr, data, by = "id") %>%
        mutate(id = factor(id))
```

```
## left_join: added 12 columns (scr, bun, cyc_db, albumin, upcratio, ...)

##            > rows only in x     0

##            > rows only in y  (  0)

##            > matched rows     200

##            >                 =====

##            > rows total       200

## mutate: converted 'id' from double to factor (0 new NA)
```

```r
glimpse(ckd)
```

```
## Rows: 200
## Columns: 14
## $ id         <fct> 498, 128, 13, 2, 183, 197, 78, 174, 91, 123, 168, 41, 13...
## $ igfrc      <dbl> 24, 18, 20, 21, 21, 21, 21, 21, 21, 22, 23, 23, 23, 23, ...
## $ scr        <dbl> 2.80, 3.14, 4.09, 2.72, 4.49, 3.86, 2.65, 3.06, 3.23, 2....
## $ bun        <dbl> 44, 45, 41, 37, 53, 44, 37, 52, 34, 47, 51, 47, 38, 61, ...
## $ cyc_db     <dbl> 2.63, 2.50, 2.97, 1.90, 4.94, 2.93, 2.25, 3.15, 3.04, 1....
## $ albumin    <dbl> 3.4, 4.2, 3.1, 4.2, 3.4, 4.1, 3.9, 4.5, 3.2, 3.8, 4.8, 3...
## $ upcratio   <dbl> 6.79, 2.01, 1.50, 0.38, 1.11, 0.29, 7.52, 0.04, 14.23, 0...
## $ adma       <dbl> 0.99, 0.66, 0.77, 0.69, 0.93, 0.85, 0.66, 0.82, 0.75, 0....
## $ sdma       <dbl> 2.33, 1.68, 1.96, 1.45, 2.22, 2.45, 1.19, 1.77, 1.43, 0....
## $ creatinine <dbl> 308.30, 293.63, 521.61, 274.34, 519.03, 512.06, 290.77, ...
## $ kynurenine <dbl> 5.57, 8.72, 6.55, 7.35, 4.51, 7.47, 11.15, 9.95, 4.77, 4...
## $ trp        <dbl> 36.89, 42.22, 45.49, 45.02, 47.73, 49.58, 50.33, 79.21, ...
## $ phe        <dbl> 73.72, 74.61, 116.18, 110.28, 98.99, 82.66, 85.27, 124.2...
## $ tyr        <dbl> 45.39, 51.82, 66.85, 79.61, 91.04, 68.90, 41.97, 94.26, ...
```

```r
#how many subjects do we have? how many variables?
```

Quick EDA

Let's look at the summary statistics:

```r
summary(ckd)
```

```
##        id          igfrc            scr              bun            cyc_db     
##  1      :  1   Min.   :18.00   Min.   :0.350   Min.   : 5.0   Min.   :0.620  
##  2      :  1   1st Qu.:29.00   1st Qu.:0.930   1st Qu.:17.0   1st Qu.:1.020  
##  3      :  1   Median :52.50   Median :1.330   Median :29.0   Median :1.300  
```

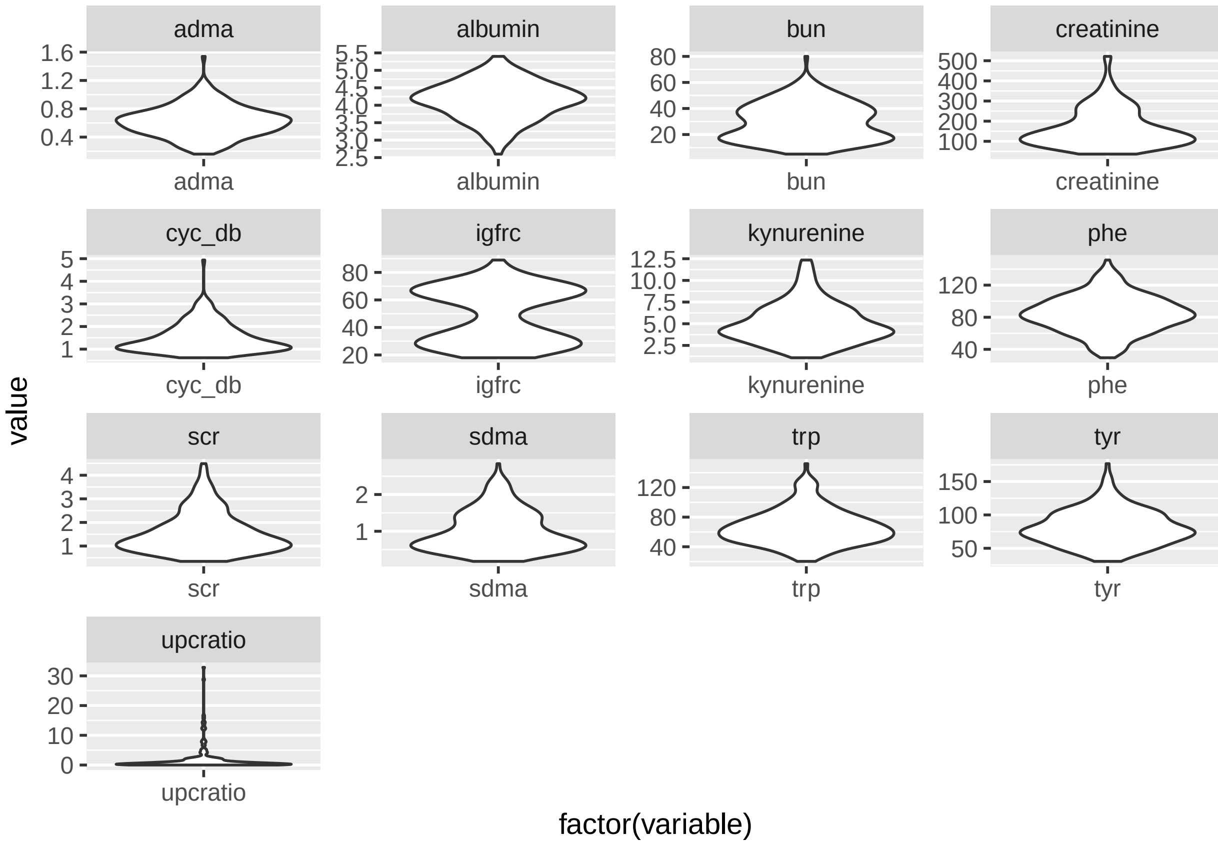
```
## 4      :  1   Mean   :48.84   Mean   :1.560   Mean   :29.7   Mean   :1.493
## 5      :  1   3rd Qu.:67.00   3rd Qu.:1.975   3rd Qu.:40.0   3rd Qu.:1.855
## 6      :  1   Max.   :89.00   Max.   :4.490   Max.   :80.0   Max.   :4.940
## (Other):194
##    albumin        upcratio           adma            sdma
## Min.   :2.600   Min.   : 0.0000   Min.   :0.160   Min.   :0.180
## 1st Qu.:3.800   1st Qu.: 0.1775   1st Qu.:0.480   1st Qu.:0.560
## Median :4.200   Median : 0.4800   Median :0.620   Median :0.845
## Mean   :4.138   Mean   : 1.8991   Mean   :0.624   Mean   :1.001
## 3rd Qu.:4.500   3rd Qu.: 1.7800   3rd Qu.:0.750   3rd Qu.:1.380
## Max.   :5.400   Max.   :32.8700   Max.   :1.540   Max.   :2.840
##
##   creatinine       kynurenine          trp              phe
## Min.   : 36.50   Min.   : 1.080   Min.   : 20.25   Min.   : 29.56
## 1st Qu.: 98.82   1st Qu.: 3.510   1st Qu.: 49.91   1st Qu.: 69.24
## Median :137.25   Median : 4.525   Median : 63.41   Median : 83.56
## Mean   :166.06   Mean   : 5.074   Mean   : 66.71   Mean   : 84.17
## 3rd Qu.:226.30   3rd Qu.: 6.543   3rd Qu.: 79.50   3rd Qu.: 98.84
## Max.   :521.61   Max.   :12.350   Max.   :152.22   Max.   :151.26
##
##      tyr
## Min.   : 30.39
## 1st Qu.: 62.45
## Median : 77.16
## Mean   : 80.77
## 3rd Qu.: 99.19
## Max.   :176.77
##
```

Let's take a quick look at the distributions of our variables using violin plots. We can do this with a few lines of code, if we gather our data into a long format and then use the `facet_wrap` function to create small tiled plots for each variable.

```
long_data <- ckd %>% pivot_longer(cols = 2:14,
                                 names_to = "variable",
                                 values_to = "value")
```

```
## pivot_longer: reorganized (igfrc, scr, bun, cyc_db, albumin, ...) into (variable, value) [was 200x14
```

```
ggplot(long_data, aes(factor(variable), value)) +
  geom_violin() + facet_wrap(~variable, scale="free")
```

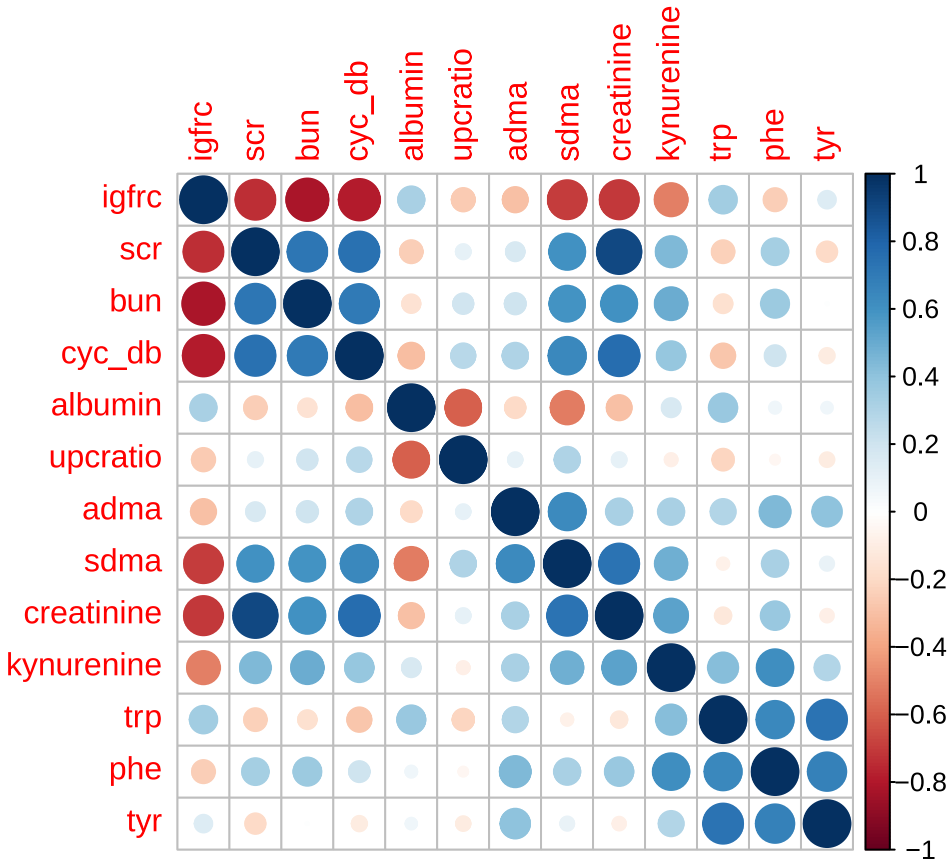


We want to predict iGFRc from the other variables. Let's get an idea of how the predictors correlate with iGFRc and each other. The `cor` function calculates the pairwise correlations for us and the `corrplot` function helps us to visualize these correlations.

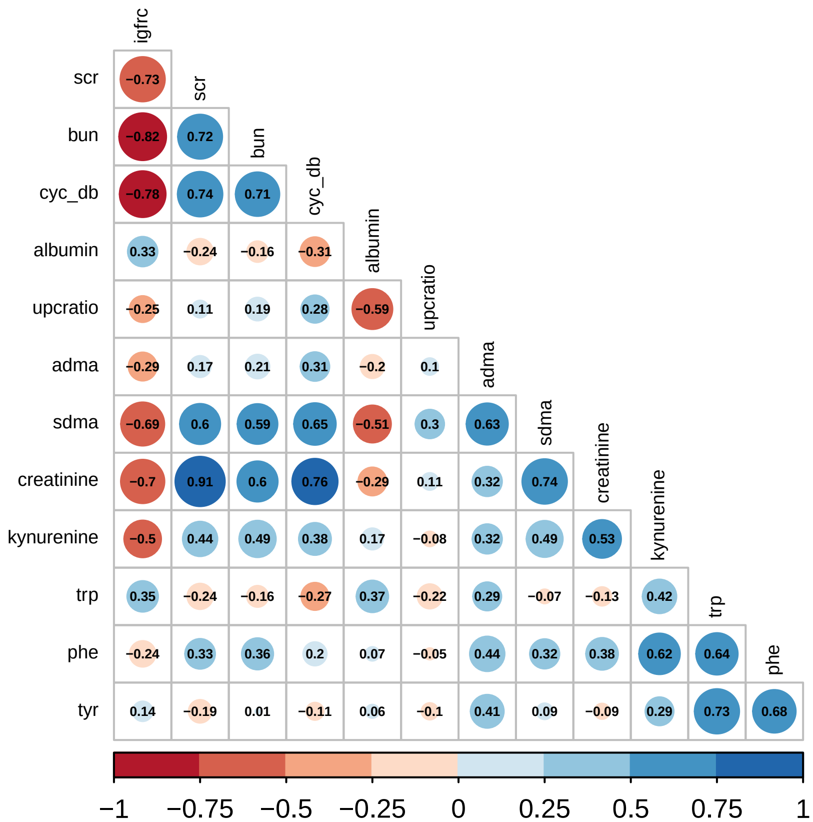
```
cors <- ckd %>%
        select(-id) %>%
        cor(use = 'pairwise.complete.obs')
```

```
## select: dropped one variable (id)
```

```
corrplot(cors) #default, but can we improve the information display? customize using available argument
```




```
corrplot(cors, type="lower", method="circle", addCoef.col="black",
         number.cex=0.45, tl.cex = 0.55, tl.col = "black",
         col=brewer.pal(n=8, name="RdBu"), diag=FALSE)
```




```
#corrplot.mixed(cors) #instead of above, can also try a different function from package
```

The correlations range from low to high and in both directions. It looks like we have several candidate predictors for iGFRc - some of which should be familiar and obvious to you. We also see that several predictors are highly correlated with each other. This is something we will come back to later and need to consider as we select predictors for our model.

Simple linear regression

Let's perform a simple linear regression to predict iGFRc. This is a model with a single predictor. In `R` we can use the `lm` function to fit linear models. We specify our formula and data in the function call. The `R` formula format is response ~ predictor(s). A formula has an implied intercept term, thus y ~ x is fit as y ~ x + int. The intercept term can be removed, if desired. When fitting a linear model y ~ x - 1 specifies a line through the origin. A model with no intercept can be also specified as y ~ x + 0 or y ~ 0 + x. You can assign a formula to a variable and use the variable name instead of the formula notation in model fitting. This may be useful if you want to compare different types of models for the same formula. In a later section we will learn how to specify formulas with more than one variable.

We will first fit the model and then examine the model output. `lm` returns an object of class "lm". This has special attributes we can explore to learn about our model and its fit of our data.

```
#fit the linear model
# lm(formula = ___, data = ___)
slr_fit <- lm(igfrc ~ scr, ckd)
```

```r
#print the model
slr_fit
```

```
##
## Call:
## lm(formula = igfrc ~ scr, data = ckd)
##
## Coefficients:
## (Intercept)          scr
##       75.34       -16.99
```

```r
#what is the equation of our model? Does this make sense to you?
```

Examining our model

Next, we want to know more about our model - is it a good fit? What are the predicted and residual values? There are several ways to examine the output from a model fit. We'll use two common ways here. Recall the `summary` function we've used before to summarize the statistics of our data set. We can use this same function on our model fit object, but we'll get a very different output.

```r
#view a summary of the model
summary(slr_fit)
```

```
##
## Call:
## lm(formula = igfrc ~ scr, data = ckd)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -26.3906 -10.9110   0.5643  10.7681  27.5325
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   75.342      1.996   37.74   <2e-16 ***
## scr          -16.987      1.124  -15.11   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.48 on 198 degrees of freedom
## Multiple R-squared:  0.5356, Adjusted R-squared:  0.5333
## F-statistic: 228.4 on 1 and 198 DF,  p-value: < 2.2e-16
```

```r
#extract information
coef(slr_fit)
```

```
## (Intercept)         scr
##    75.34187   -16.98668
```

A model fit summary is fine for scrolling through and enables you to extract some components individually, but is not well designed for extracting the information in a straightforward or tidy way. We often do want to use this information later or collate it in some way, maybe even as a data frame. The `broom` package was designed for this very problem. We will learn more about three of its functions.

The `tidy` function takes the coefficient information and organizes it into a dataframe where each row holds data for one term of the model.

```
tidy(slr_fit)
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)     75.3      2.00      37.7 2.22e-92
## 2 scr            -17.0      1.12     -15.1 8.04e-35
```

```
# aha!
```

You may also want to see the actual values with the fitted values and their residuals, or bring them into a format you can analyze further. This is done using the `augment` function - because it augments the original data with the information from the model.

```
head(augment(slr_fit))
```

```
## # A tibble: 6 x 8
##   igfrc   scr .fitted .resid .std.resid   .hat .sigma  .cooksd
##   <dbl> <dbl>   <dbl>  <dbl>      <dbl>  <dbl>  <dbl>    <dbl>
## 1    24  2.8   27.8    -3.78     -0.283 0.0157   13.5 0.000636
## 2    18  3.14  22.0    -4.00     -0.300 0.0223   13.5 0.00103
## 3    20  4.09   5.87   14.1       1.08  0.0495   13.5 0.0301
## 4    21  2.72  29.1    -8.14     -0.608 0.0143   13.5 0.00269
## 5    21  4.49  -0.928  21.9       1.68  0.0646   13.4 0.0977
## 6    21  3.86   9.77   11.2       0.851 0.0417   13.5 0.0158
```

```
#if you want to add the fit-related columns to the entire data frame, specify the data frame
head(augment(slr_fit, ckd))
```

```
## # A tibble: 6 x 20
##   id    igfrc   scr   bun cyc_db albumin upcratio  adma  sdma creatinine
##   <fct> <dbl> <dbl> <dbl>  <dbl>   <dbl>    <dbl> <dbl> <dbl>      <dbl>
## 1 498      24  2.8     44   2.63     3.4    6.79   0.99  2.33       308.
## 2 128      18  3.14    45   2.5      4.2    2.01   0.66  1.68       294.
## 3 13       20  4.09    41   2.97     3.1    1.5    0.77  1.96       522.
## 4 2        21  2.72    37   1.9      4.2    0.38   0.69  1.45       274.
## 5 183      21  4.49    53   4.94     3.4    1.11   0.93  2.22       519.
## 6 197      21  3.86    44   2.93     4.1    0.290  0.85  2.45       512.
## # ... with 10 more variables: kynurenine <dbl>, trp <dbl>, phe <dbl>,
## #   tyr <dbl>, .fitted <dbl>, .resid <dbl>, .std.resid <dbl>, .hat <dbl>,
## #   .sigma <dbl>, .cooksd <dbl>
```

Finally, you can use the `glance` function to get a single row of the performance and error metrics. This format becomes very useful when comparing different fits on the same data.

```
glance(slr_fit)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>
## 1     0.536         0.533  13.5      228. 8.04e-35     1  -803. 1612. 1622.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Exercise 1:

Select a variable (other than SCr) and perform a single variable regression for iGFRc using the ckd dataset. Determine the model equation and R2 value. How did your model fit compare to our SCr example?

End exercise

Making predictions from our model

Now that we've created a model, we want to use it to make predictions. This is done using the `predict` function. We will add our predicted values to the ckd data set as a new column, iGFR_pred.

```
ckd <- ckd %>%
        mutate(igfr_pred = round(predict(slr_fit, ckd),0))
```

```
## mutate: new variable 'igfr_pred' (double) with 54 unique values and 0% NA
```

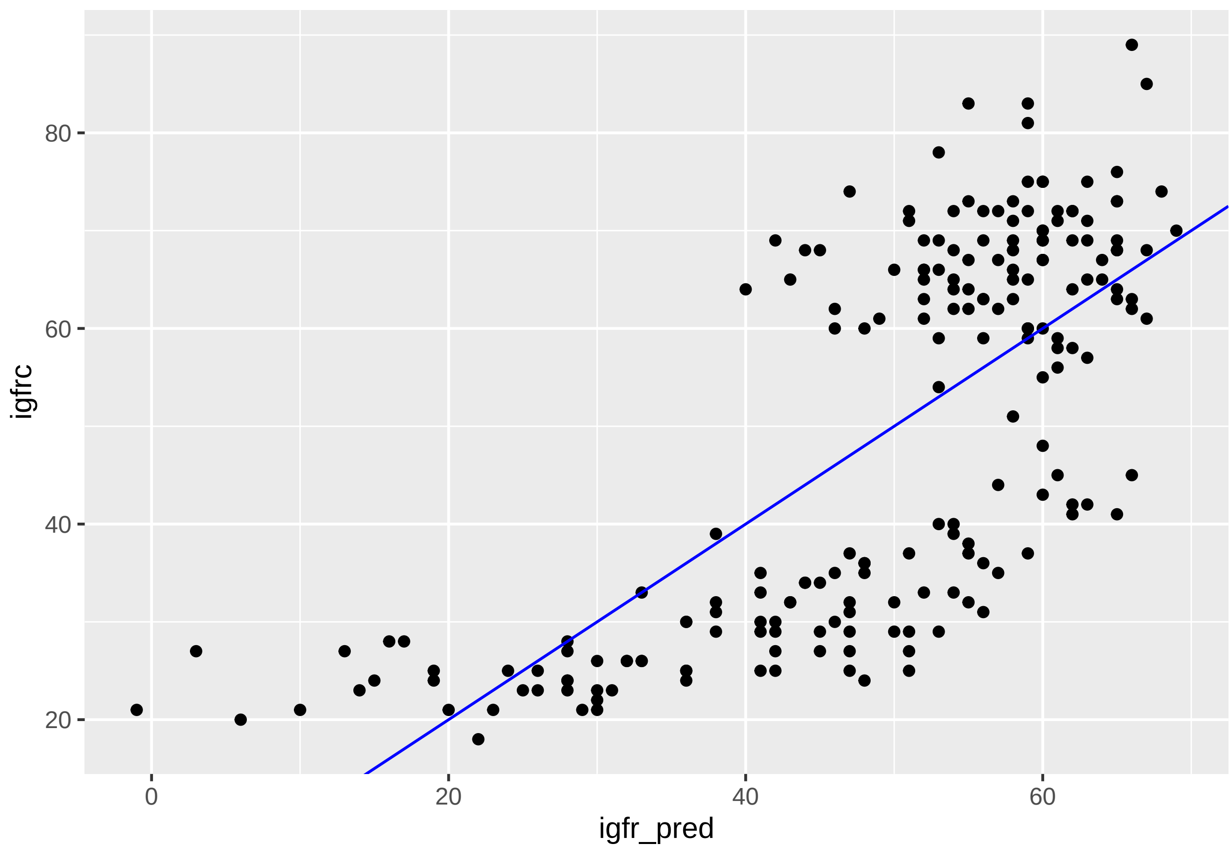
The values we get from the predict call are identical to the fitted.values from the model fit call since we used the same data for both functions.

```
comp <- cbind(round(slr_fit$fitted.values,0), ckd$igfr_pred)
head(comp)
```

```
##   [,1] [,2]
## 1   28   28
## 2   22   22
## 3    6    6
## 4   29   29
## 5   -1   -1
## 6   10   10
```

We can plot the actual vs predicted values to gain a sense of how well the model is predicting iGFRc.

```
# Make a plot to compare predictions to actual (prediction on x axis).
ggplot(ckd, aes(x = igfr_pred, y = igfrc)) +
  geom_point() +
  geom_abline(color = "blue")
```



I hope we can improve on this model! So far, the R2 is around 0.5 and the plot of actual versus predicted values does not look linear. An obvious next step is to add complexity to the model and use other available variables to try to better predict iGFRc.

Multivariate linear regression

With multivariate linear regression, we will use more than one dependent variable to predict our independent variable. We can do this using the same `lm` function we used above, but we change the formula to include the additonal variables. This can be as extreme as y ~ . to regress the response by ALL available predictors. Though we may evaluate this type of model, we have to be particularly careful of multicolinearity from highly correlated dependent variables, as this will introduce problems into the prediction.

Split the data

Now is a good time to introduce the concept of train and test data sets. This is a fundamental practice in predictive modeling. We will randomly split our data into two groups: a training set and a test set. We will fit our model to one set (train) and use the other (test) for making predictions. The test set is sometimes called the internal validation set. We can then assess a model's expected performance by comparing the error in the train and test sets. A commonly used approach splits the data 75:25 as train:test.

```
#reload data to remove SLR predictions
data <- read_csv("data/CKD_data.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
```

```
##   id = col_double(),
##   scr = col_double(),
##   bun = col_double(),
##   cyc_db = col_double(),
##   Albumin = col_double(),
##   upcratio = col_double(),
##   ADMA = col_double(),
##   SDMA = col_double(),
##   Creatinine = col_double(),
##   Kynurenine = col_double(),
##   Trp = col_double(),
##   Phe = col_double(),
##   Tyr = col_double()
## )
```

```
gfr <- read_csv("data/CKD_GFR.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   igfrc = col_double()
## )
```

```
#join by ID, convert ID to factor
ckd <- left_join(gfr, data, by = "id") %>%
        mutate(id = factor(id))
```

```
## left_join: added 12 columns (scr, bun, cyc_db, albumin, upcratio, ...)

##              > rows only in x     0

##              > rows only in y  (   0)

##              > matched rows     200

##              >                 =====

##              > rows total       200

## mutate: converted 'id' from double to factor (0 new NA)
```

```
# sample(x, size, replace = FALSE, prob = NULL)
set.seed(622) #so we all get same random numbers
train <- sample(nrow(ckd), nrow(ckd) * 0.75)
test <- -train

ckd_train <- ckd[train, ] %>%
              select(-id)
```

```
## select: dropped one variable (id)
```

```
ckd_test <- ckd[test, ] %>%
              select(-id)
```

```
## select: dropped one variable (id)
```

```
# alternatively, two dplyr versions that only work on tibbles:
# sample_n(tbl, size, replace = FALSE, weight = NULL, .env = NULL)
# sample_frac(tbl, size = 1, replace = FALSE, weight = NULL,
#   .env = NULL)
```

We will use values in the train and test variables we created as indices to assign rows to one group or the other.

Let's build a model for iGFRc based on all variables (except id). We will fit it to the training data.

```
#fit the model
mod_full <- lm(igfrc ~ ., data = ckd_train)

#check out the model info
glance(mod_full)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>
## 1     0.856         0.843  7.70      67.8 1.72e-51    12  -512. 1053. 1095.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

```
tidy(mod_full) %>%
    arrange(p.value)
```

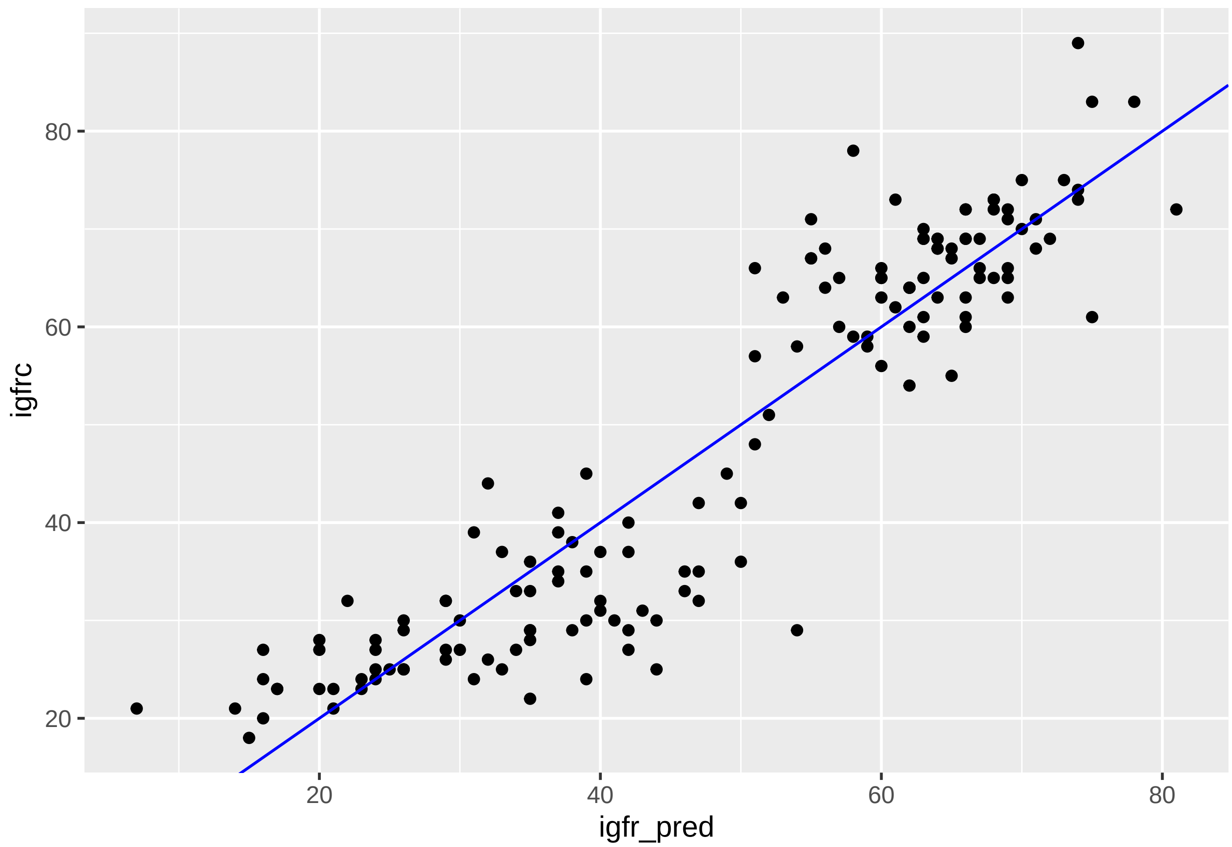
```
## # A tibble: 13 x 5
##    term        estimate std.error statistic       p.value
##    <chr>          <dbl>     <dbl>     <dbl>         <dbl>
##  1 trp           0.381     0.0584    6.53   0.00000000121
##  2 (Intercept)  62.8       9.68      6.49   0.00000000146
##  3 kynurenine   -3.09      0.509    -6.08   0.0000000115
##  4 bun          -0.482     0.103    -4.67   0.00000703
##  5 cyc_db       -5.63      1.90     -2.96   0.00366
##  6 adma        -11.6       4.76     -2.44   0.0160
##  7 phe          -0.0982    0.0625   -1.57   0.118
##  8 albumin       2.96      2.19      1.35   0.178
##  9 tyr           0.0234    0.0481    0.487  0.627
## 10 creatinine    0.0107    0.0289    0.371  0.711
## 11 sdma         -1.18      3.29     -0.357  0.721
## 12 upcratio      0.0345    0.194     0.178  0.859
## 13 scr          -0.227     2.84     -0.0799 0.936
```

```
#add the predicted values to the train set
ckd_train <- ckd_train %>%
              mutate(igfr_pred = round(augment(mod_full)$.fitted,0))
```

```
## mutate: new variable 'igfr_pred' (double) with 58 unique values and 0% NA
```

Plot the actual vs predicted for the training set fit for mod.full.

```
ggplot(ckd_train, aes(x = igfr_pred, y = igfrc)) +
  geom_point() +
  geom_abline(color = "blue")
```



Looks pretty reasonable and much improved over the simple linear regression model.

Let's predict the iGFRc values for our test set to see how the model does when predicting new data.

Exercise 2:

Predict the iGFRc values for the test set using the mod.full and plot the actual vs predicted values.

End exercise

Evaluating model performance

We can examine how well the model is predicting iGFRc in a few ways: (1) plot the actual vs predicted values, (2) plot the residuals vs predicted values, and (3) plot a qq plot or histogram of the residuals. The residuals are the difference between the actual and predicted values. We will also calculate the root mean squared error (RMSE), as this metric is often used to express the error for a model so its performance can be compared to that from other models.
Linear regression relies on several assumptions (though it can be pretty robust even if some assumptions are violated to some degree).

These assumptions include: - The relationship between the response and predictors is linear and additive - The errors are independent (i.e., not serially correlated) - The errors have constant variance (i.e., have homoscedasticity) - The errors are normally distributed

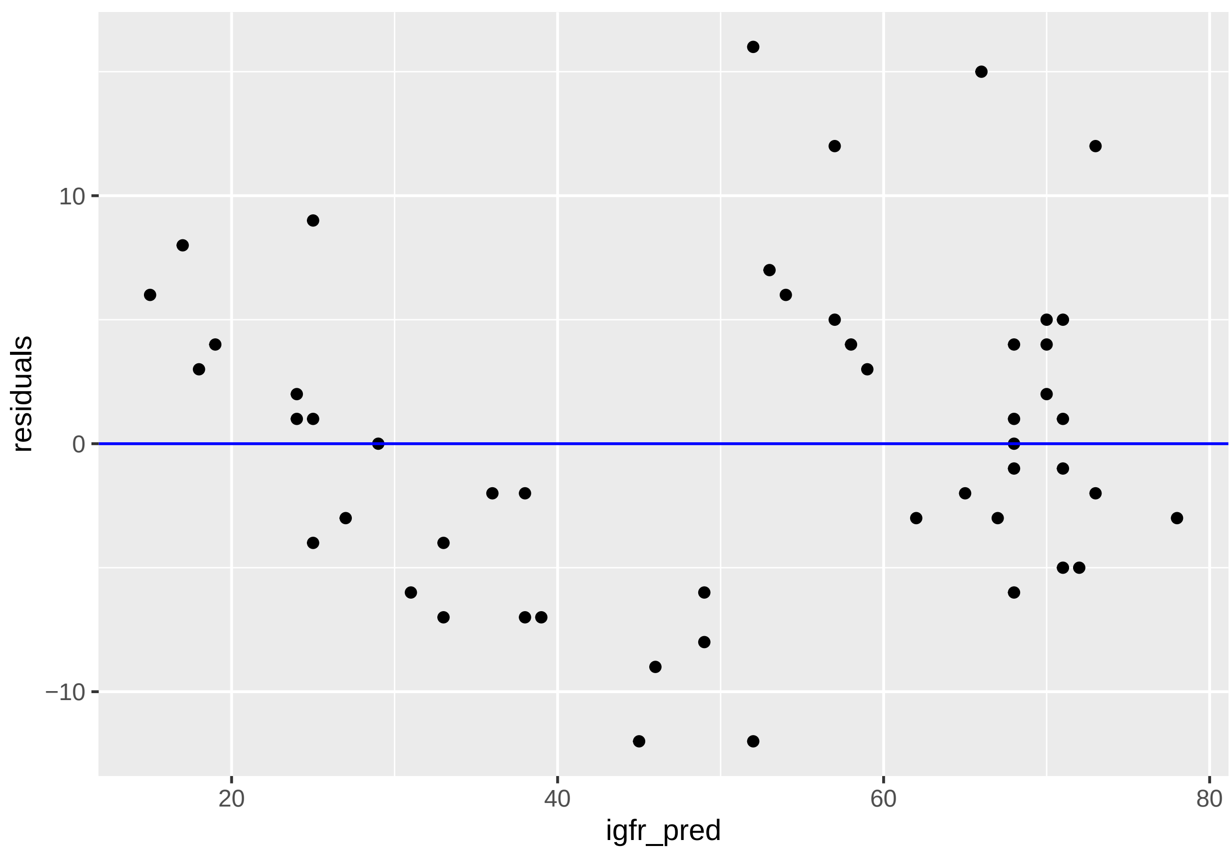
We can examine the residuals to make sure our assumptions are valid for a particular model. We are looking for low residuals that have similar variance over the range of predicted values and are normally distributed. We also look for a linear trend in the actual vs observed values.

```
## mutate: new variable 'igfr_pred' (double) with 32 unique values and 0% NA
```

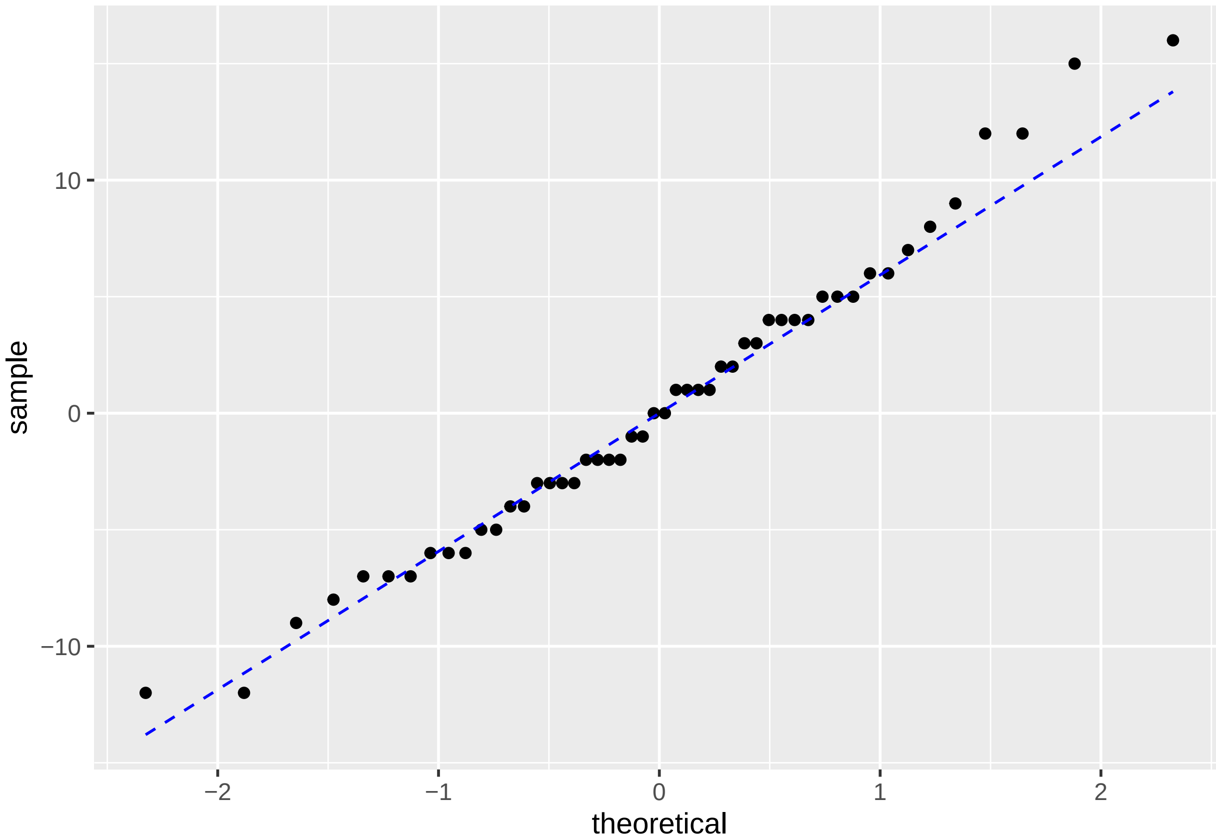
```
# Make a residual plot (prediction on x axis).
ckd_test <- ckd_test %>%
        mutate(residuals = igfrc - igfr_pred)
```

```
## mutate: new variable 'residuals' (double) with 23 unique values and 0% NA
```

```
ggplot(ckd_test, aes(x = igfr_pred, y = residuals)) +
  geom_point() +
  geom_hline(yintercept = 0, color = "blue")
```




```
# Make a QQ plot of the residuals
ggplot(ckd_test, aes(sample = residuals)) +
  stat_qq() + stat_qq_line(linetype = 2, color = "blue")
```




```
# alternatively can visually assess normality via histogram of residuals
# ggplot(ckd_test, aes(residuals)) +
#   geom_histogram()
```

We can use the `rmse` function from the `Metrics` package to calculate the RMSE for the training and test set predictions. If our model is not overfit, we expect the values for the two sets to be similar. As you might expect, a lower RMSE indicates a better fitting model. Another commonly calculated error metric, Mean Absolute Percent Error (MAPE), can be calculated using the `mape` function from the `Metrics` package.

```
# rmse(actual, predicted)
rmse(ckd_train$igfrc, mod_full$fitted.values) #7.36
```

```
## [1] 7.362656
```

```
rmse(ckd_test$igfrc, ckd_test$igfr_pred) #6.37
```

```
## [1] 6.368673
```

```
# mape(actual, predicted)
mape(ckd_train$igfrc, mod_full$fitted.values) #0.15 or 15%
```

```
## [1] 0.1475183
```

```r
mape(ckd_test$igfrc, ckd_test$igfr_pred) #0.12 or 12%
```

```
## [1] 0.120474
```

Examining collinearity

As mentioned before, we need to be careful when several predictors have strong correlation. The variance inflation factor (VIF) can be calculated for each model to determine how much the variance of a regression coefficient is inflated due to multicollinearity in the model.

The smallest possible value of VIF is one (absence of multicollinearity). As a rule of thumb, a VIF value that exceeds 5 or 10 indicates a problematic amount of collinearity.

```r
vif(mod_full)
```

```
##        scr        bun     cyc_db    albumin   upcratio       adma       sdma
##  14.609031   5.389522   4.028942   3.399489   2.052432   2.743919   8.384208
## creatinine kynurenine        trp        phe        tyr
##  19.355677   3.691210   4.989289   5.312483   4.192178
```

As we expected! There are several VIF above 5 or 10 in our model. Though it seems to fit well, the coefficients may be unstable due to the multicollinearity, making the model's performance on new data unpredictable.

When multicollinearity is present, a first consideration is to remove highly correlated variables, since the presence of multicollinearity implies that the information that this variable provides about the response is redundant in the presence of the other variables. Removal of one or more variables may have an unexpected effect on other variables.

Feature engineering

Feature engineering is the process of creating and selecting the best predictors for a model. This is an area where the 'art' of modeling is practiced and can have a great impact on results. The scope of this topic is beyond our time in this course, but we will do a brief exercise in variable selection to attempt to resolve our collinearity problem.

Let's go back and review the information from our full model fit to get an idea of what variables we may want to keep and not.

```r
#sort the variables by the p value of their coefficients
tidy(mod_full) %>%
    arrange(p.value)
```

```
## # A tibble: 13 x 5
##    term        estimate std.error statistic       p.value
##    <chr>          <dbl>     <dbl>     <dbl>         <dbl>
##  1 trp            0.381     0.0584     6.53  0.00000000121
##  2 (Intercept)  62.8        9.68       6.49  0.00000000146
##  3 kynurenine   -3.09       0.509     -6.08  0.0000000115
##  4 bun          -0.482      0.103     -4.67  0.00000703
##  5 cyc_db       -5.63       1.90      -2.96  0.00366
##  6 adma        -11.6        4.76      -2.44  0.0160
##  7 phe          -0.0982     0.0625    -1.57  0.118
```

```
##  8 albumin       2.96      2.19      1.35   0.178
##  9 tyr           0.0234    0.0481    0.487  0.627
## 10 creatinine    0.0107    0.0289    0.371  0.711
## 11 sdma         -1.18      3.29     -0.357  0.721
## 12 upcratio      0.0345    0.194     0.178  0.859
## 13 scr          -0.227     2.84     -0.0799 0.936
```

```
#if we select the 'most' significant variables with pvalues ~ 0.05:
# Trp, Kynurenine, BUN, CYC_DB, Phe, ADMA
```

To specify a formula for multiple variables, we use the y ~ a + b + c format, where a, b, and c are the independent variables we want to include in the model.

Exercise 3:

(1) Run the code chunk below to reset the data variables.

```
#reload data to remove full model predictions
data <- read_csv("data/CKD_data.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   scr = col_double(),
##   bun = col_double(),
##   cyc_db = col_double(),
##   Albumin = col_double(),
##   upcratio = col_double(),
##   ADMA = col_double(),
##   SDMA = col_double(),
##   Creatinine = col_double(),
##   Kynurenine = col_double(),
##   Trp = col_double(),
##   Phe = col_double(),
##   Tyr = col_double()
## )
```

```
gfr <- read_csv("data/CKD_GFR.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   igfrc = col_double()
## )
```

```
#join by ID, convert ID to factor
ckd <- left_join(gfr, data, by = "id") %>%
        mutate(id = factor(id))
```

```
## left_join: added 12 columns (scr, bun, cyc_db, albumin, upcratio, ...)
```

```
##            > rows only in x     0
```

```
##              > rows only in y  (  0)

##              > matched rows     200

##              >                 =====

##              > rows total       200

## mutate: converted 'id' from double to factor (0 new NA)
```

```
# sample(x, size, replace = FALSE, prob = NULL)
set.seed(622) #so we all get same random numbers
train <- sample(nrow(ckd), nrow(ckd) * 0.75)
test <- -train

ckd_train <- ckd[train, ] %>%
              select(-id)
```

```
## select: dropped one variable (id)
```

```
ckd_test <- ckd[test, ] %>%
              select(-id)
```

```
## select: dropped one variable (id)
```

(2) Fit a new model, mod2, that uses Trp, Kynurenine, BUN, CYC_DB, Phe, ADMA to predict iGFRc in the training set. Add the predicted values to the training set as a new variable, iGFR_pred.

(3) Write out the equation for this model. Does it make sense, based on your prior knowledge?

(4) Find the R2, RMSE, and MAPE values for the model fit on the training set.

(5) Check for collinearity.

(6) Examine the residuals and actual vs predicted.

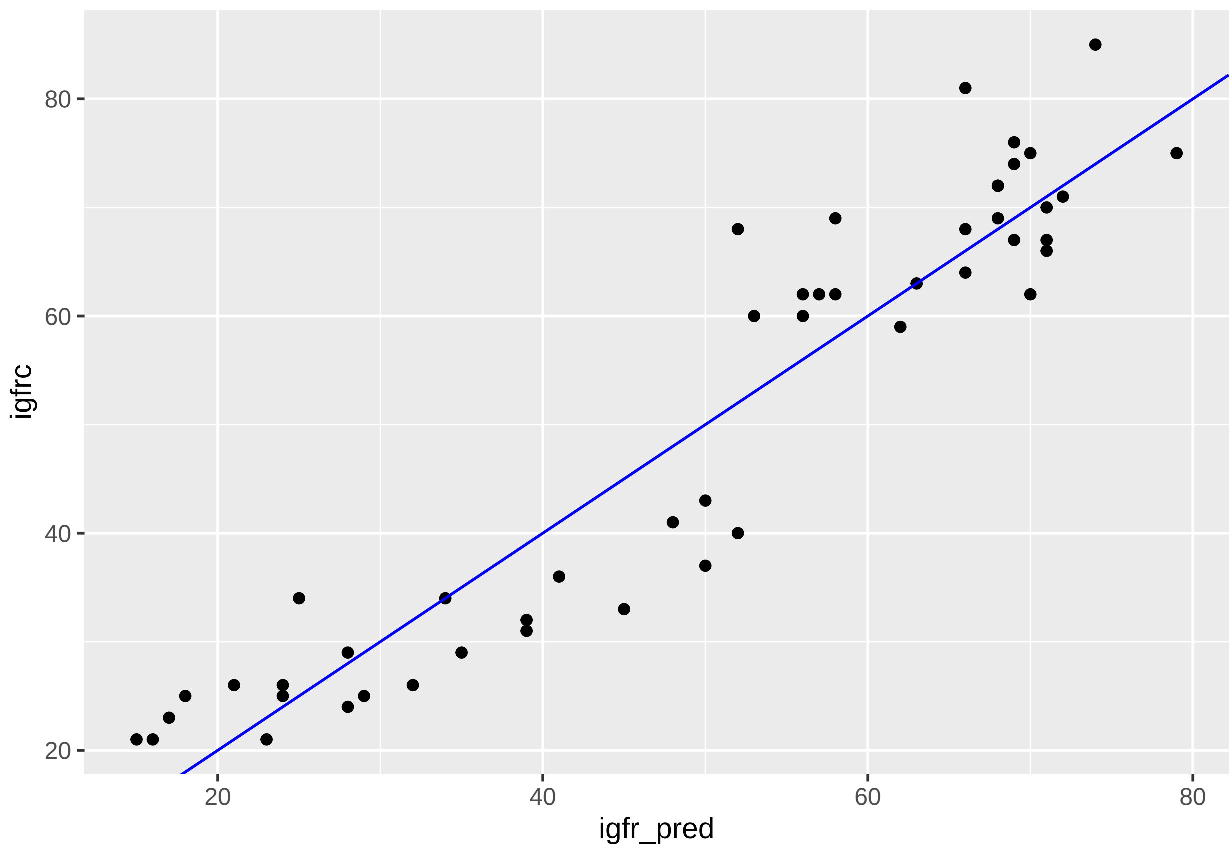
End Exercise

As the last step, we'll confirm the performance on our test set.

```
#predict values for test set and add as new column, iGFR_pred
ckd_test <- ckd_test %>%
              mutate(igfr_pred = round(predict(mod2, ckd_test),0))
```

```
## mutate: new variable 'igfr_pred' (double) with 33 unique values and 0% NA
```

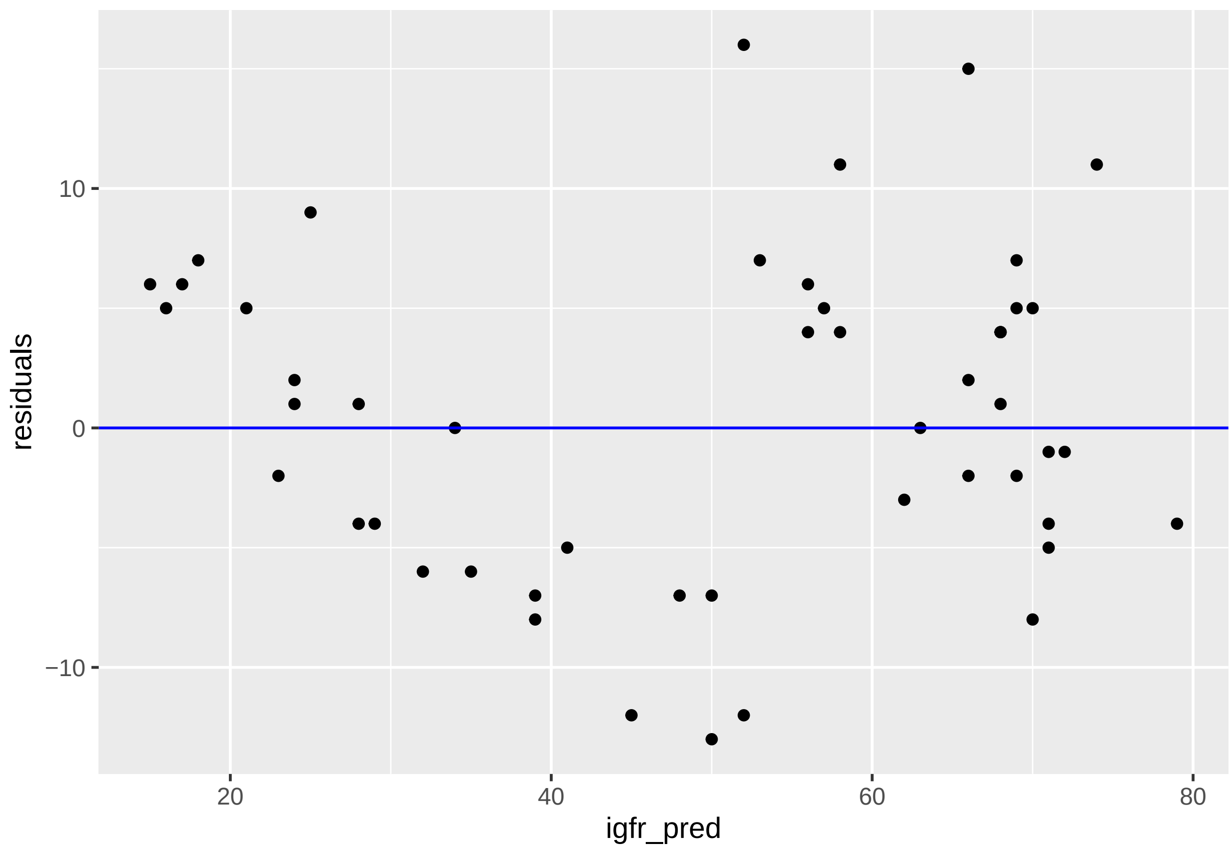
```
#plot actual vs predicted for test set
ggplot(ckd_test, aes(x = igfr_pred, y = igfrc)) +
  geom_point() +
  geom_abline(color = "blue")
```



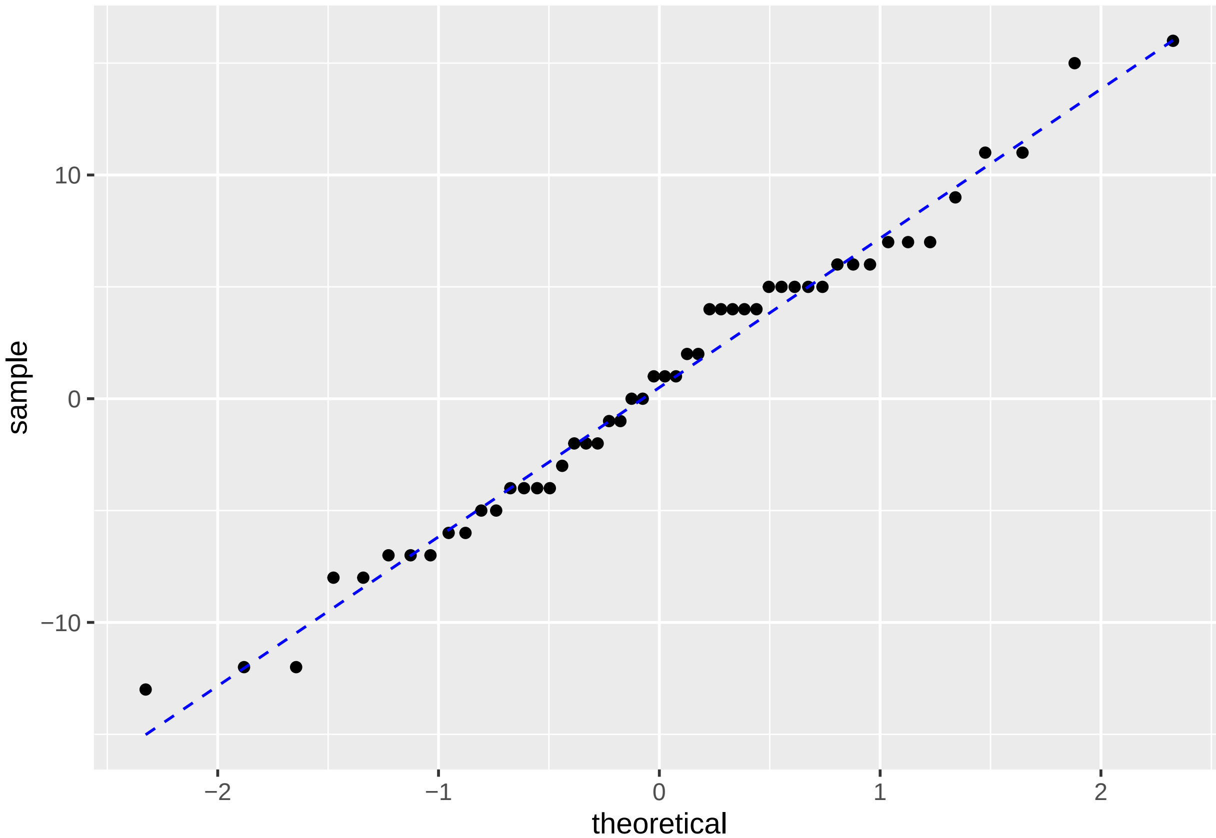

```
# Make a residual plot (prediction on x axis).
ckd_test <- ckd_test %>%
        mutate(residuals = igfrc - igfr_pred)
```

```
## mutate: new variable 'residuals' (double) with 21 unique values and 0% NA
```

```
ggplot(ckd_test, aes(x = igfr_pred, y = residuals)) +
  geom_point() +
  geom_hline(yintercept = 0, color = "blue")
```




```r
# QQ plot of the residuals
ggplot(ckd_test, aes(sample = residuals)) +
  stat_qq() + stat_qq_line(linetype = 2, color = "blue")
```




```
#calculate test set error
rmse(ckd_test$igfrc, ckd_test$igfr_pred) #6.66
```

```
## [1] 6.657327
```

```
mape(ckd_test$igfrc, ckd_test$igfr_pred) #0.13 or 13%
```

```
## [1] 0.1305303
```

We solved our collinearity problem and didn't really lose anything on performance. This also made our model less complex. The RMSE and MAPE values for the train and test sets are similar and the residual plots look pretty good. From our prior knowledge, the variables and signs of the coefficients in the model seem reasonable. These results suggest we could expect similar performance from our model when it is applied to new data that is of a similar range as our train and test sets.

Acknowledgement

The data used in this lesson was simulated from a data set generated in collaboration with Dr. Ellen Brooks. Prior to simulation, the metabolomics data was processed and cleaned by Dr. David Lin. The lesson design was influenced by the DataCamp course: Supervised Learning in R: Regression.

Summary

- Linear regression is a widely applied tool in predictive modeling and machine learning.
- There are 4 primary assumptions in multivariate linear regression that must be evaluated for a given model.
- Best practice is to randomly split data into train and test sets, used to fit and evaluate the model.
- Collinearity can be a problem with multivariate linear models.

Classifications using linear regression

Overview of data

In the previous lession we applied linear regression to make quantitative predictions. In this lesson, we will learn how a different type of linear regression, logistic regression, can be used to make class or category predictions. In its most basic form, this type of prediction is binary, meaning it has only two options: yes (1) or no (0); disease or no disease, etc. Using the same core data set from the previous lesson, we will attempt to classify children with chronic kidney disease by CKD stage as stage 2 vs stage 3b. The iGFRc column has been removed for this lesson, as this is how CKD stage is determined.

Our data is provided in two files. One has values for the outcome (Stage) for each subject ID and the other includes values for several predictors (e.g., creatinine, BUN, various endogenous metabolites) measured for each subject ID.

We will need to use our previously learned skills to read in the data and join the two sets by subject.

```
#load in CKD_data.csv and CKD_stage.csv
data <- read_csv("data/CKD_data.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   scr = col_double(),
##   bun = col_double(),
##   cyc_db = col_double(),
##   Albumin = col_double(),
##   upcratio = col_double(),
##   ADMA = col_double(),
##   SDMA = col_double(),
##   Creatinine = col_double(),
##   Kynurenine = col_double(),
##   Trp = col_double(),
##   Phe = col_double(),
##   Tyr = col_double()
## )
```

```
glimpse(data)
```

```
## Rows: 200
## Columns: 13
## $ id         <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1...
## $ scr        <dbl> 1.93, 2.72, 0.88, 0.78, 1.28, 1.41, 0.58, 1.34, 3.42, 2....
## $ bun        <dbl> 21, 37, 12, 27, 27, 19, 17, 13, 45, 39, 80, 52, 41, 23, ...
```

```
## $ cyc_db     <dbl> 0.82, 1.90, 0.65, 1.31, 1.50, 1.60, 1.04, 1.04, 3.04, 2....
## $ albumin    <dbl> 4.1, 4.2, 4.2, 4.4, 3.6, 4.2, 4.5, 3.8, 4.1, 4.0, 4.2, 5...
## $ upcratio   <dbl> 1.39, 0.38, 0.33, 0.26, 0.57, 0.04, 0.12, 5.17, 0.26, 0....
## $ adma       <dbl> 0.41, 0.69, 0.57, 0.39, 0.87, 0.48, 0.52, 1.14, 0.46, 0....
## $ sdma       <dbl> 0.70, 1.45, 0.49, 0.33, 2.31, 0.74, 0.46, 1.42, 0.96, 1....
## $ creatinine <dbl> 138.61, 274.34, 78.81, 63.05, 226.21, 150.50, 70.84, 176...
## $ kynurenine <dbl> 3.98, 7.35, 1.76, 2.41, 5.91, 4.29, 3.19, 6.18, 8.08, 6....
## $ trp        <dbl> 78.18, 45.02, 58.62, 34.64, 62.36, 52.21, 49.28, 101.75,...
## $ phe        <dbl> 79.45, 110.28, 74.47, 39.81, 75.40, 58.66, 53.82, 93.30,...
## $ tyr        <dbl> 77.00, 79.61, 65.22, 44.55, 94.24, 60.66, 68.07, 110.73,...
```

```
stage <- read_csv("data/CKD_stage.csv") %>% clean_names()
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   Stage = col_character()
## )
```

```
glimpse(stage)
```

```
## Rows: 200
## Columns: 2
## $ id    <dbl> 498, 128, 13, 2, 183, 197, 78, 174, 91, 123, 168, 41, 130, 10...
## $ stage <chr> "CKD3b", "CKD3b", "CKD3b", "CKD3b", "CKD3b", "CKD3b", "CKD3b"...
```

```
#join by ID, convert ID and Stage variables to factors
ckd <- left_join(stage, data, by = "id") %>%
        mutate(id = factor(id),
               stage = factor(stage))
```

```
## left_join: added 12 columns (scr, bun, cyc_db, albumin, upcratio, ...)

##              > rows only in x     0

##              > rows only in y  (  0)

##              > matched rows     200

##              >                 =====

##              > rows total       200

## mutate: converted 'id' from double to factor (0 new NA)

##         converted 'stage' from character to factor (0 new NA)
```

```r
glimpse(ckd)
```

```
## Rows: 200
## Columns: 14
## $ id         <fct> 498, 128, 13, 2, 183, 197, 78, 174, 91, 123, 168, 41, 13...
## $ stage      <fct> CKD3b, CKD3b, CKD3b, CKD3b, CKD3b, CKD3b, CKD3b, CKD3b, ...
## $ scr        <dbl> 2.80, 3.14, 4.09, 2.72, 4.49, 3.86, 2.65, 3.06, 3.23, 2....
## $ bun        <dbl> 44, 45, 41, 37, 53, 44, 37, 52, 34, 47, 51, 47, 38, 61, ...
## $ cyc_db     <dbl> 2.63, 2.50, 2.97, 1.90, 4.94, 2.93, 2.25, 3.15, 3.04, 1....
## $ albumin    <dbl> 3.4, 4.2, 3.1, 4.2, 3.4, 4.1, 3.9, 4.5, 3.2, 3.8, 4.8, 3...
## $ upcratio   <dbl> 6.79, 2.01, 1.50, 0.38, 1.11, 0.29, 7.52, 0.04, 14.23, 0...
## $ adma       <dbl> 0.99, 0.66, 0.77, 0.69, 0.93, 0.85, 0.66, 0.82, 0.75, 0....
## $ sdma       <dbl> 2.33, 1.68, 1.96, 1.45, 2.22, 2.45, 1.19, 1.77, 1.43, 0....
## $ creatinine <dbl> 308.30, 293.63, 521.61, 274.34, 519.03, 512.06, 290.77, ...
## $ kynurenine <dbl> 5.57, 8.72, 6.55, 7.35, 4.51, 7.47, 11.15, 9.95, 4.77, 4...
## $ trp        <dbl> 36.89, 42.22, 45.49, 45.02, 47.73, 49.58, 50.33, 79.21, ...
## $ phe        <dbl> 73.72, 74.61, 116.18, 110.28, 98.99, 82.66, 85.27, 124.2...
## $ tyr        <dbl> 45.39, 51.82, 66.85, 79.61, 91.04, 68.90, 41.97, 94.26, ...
```

```r
#how many subjects do we have? how many variables? how many subjects in each class?
```

Quick EDA

Let's look at the summary statistics. We also need to be aware of any class bias. This is a situation where one class is over-represented in the data. If so, this can create problems with the modeling. The ideal state is for classes to be balanced. There are several ways to handle class imbalance problems, but they are outside the scope of this course. For this activity, we have provided data that is balanced.

```r
prop.table(table(ckd$stage))
```

```
##
##  CKD2 CKD3b
##   0.5   0.5
```

```r
summary(ckd)
```

```
##        id        stage          scr             bun            cyc_db
##  1      :  1   CKD2 :100   Min.   :0.350   Min.   : 5.0   Min.   :0.620
##  2      :  1   CKD3b:100   1st Qu.:0.930   1st Qu.:17.0   1st Qu.:1.020
##  3      :  1               Median :1.330   Median :29.0   Median :1.300
##  4      :  1               Mean   :1.560   Mean   :29.7   Mean   :1.493
##  5      :  1               3rd Qu.:1.975   3rd Qu.:40.0   3rd Qu.:1.855
##  6      :  1               Max.   :4.490   Max.   :80.0   Max.   :4.940
##  (Other):194
##     albumin         upcratio            adma             sdma
##  Min.   :2.600   Min.   : 0.0000   Min.   :0.160   Min.   :0.180
##  1st Qu.:3.800   1st Qu.: 0.1775   1st Qu.:0.480   1st Qu.:0.560
##  Median :4.200   Median : 0.4800   Median :0.620   Median :0.845
##  Mean   :4.138   Mean   : 1.8991   Mean   :0.624   Mean   :1.001
##  3rd Qu.:4.500   3rd Qu.: 1.7800   3rd Qu.:0.750   3rd Qu.:1.380
```

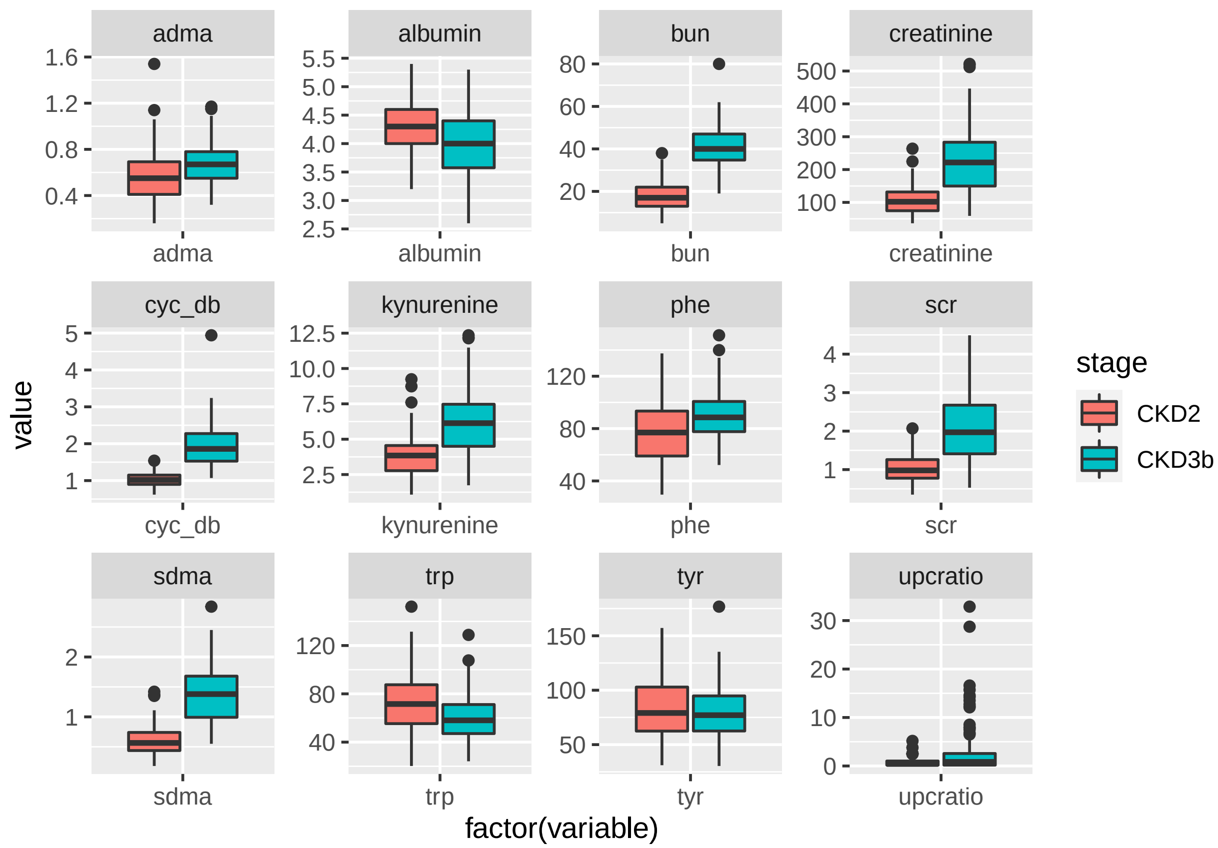
```
##  Max.   :5.400   Max.   :32.8700   Max.   :1.540   Max.   :2.840  
##                                                                      
##    creatinine       kynurenine          trp               phe        
##  Min.   : 36.50   Min.   : 1.080   Min.   : 20.25   Min.   : 29.56  
##  1st Qu.: 98.82   1st Qu.: 3.510   1st Qu.: 49.91   1st Qu.: 69.24  
##  Median :137.25   Median : 4.525   Median : 63.41   Median : 83.56  
##  Mean   :166.06   Mean   : 5.074   Mean   : 66.71   Mean   : 84.17  
##  3rd Qu.:226.30   3rd Qu.: 6.543   3rd Qu.: 79.50   3rd Qu.: 98.84  
##  Max.   :521.61   Max.   :12.350   Max.   :152.22   Max.   :151.26  
##                                                                      
##       tyr        
##  Min.   : 30.39  
##  1st Qu.: 62.45  
##  Median : 77.16  
##  Mean   : 80.77  
##  3rd Qu.: 99.19  
##  Max.   :176.77  
## 
```

We can create boxplots for each variable, filled by Stage, to see if there are differences in distributions across the classes. This may provide clues about which variables may be good predictors for CKD Stage.

```
long_data <- pivot_longer(data = ckd, names_to = "variable",
                         values_to = "value", cols = 3:14)
```

```
## pivot_longer: reorganized (scr, bun, cyc_db, albumin, upcratio, ...) into (variable, value) [was 200
```

```
ggplot(long_data, aes(factor(variable), value, fill = stage)) +
  geom_boxplot() + facet_wrap(~variable, scale="free")
```

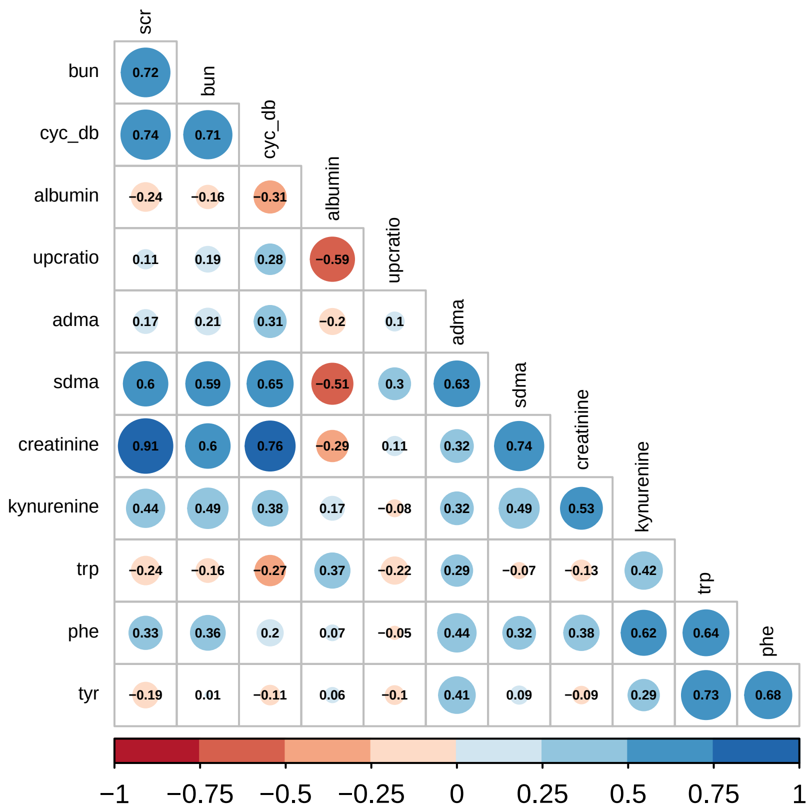


From the boxplots, it looks like we have several candidate predictors for Stage - some of which should be familiar and obvious to you. Collinearity is a problem for logistic regression that must be addressed for multivariate models. As before, we should have an idea of how the predictors correlate with each other.

```
cors <- ckd %>%
        select(-id, -stage) %>%
        cor(use = 'pairwise.complete.obs')
```

```
## select: dropped 2 variables (id, stage)
```

```
corrplot(cors, type="lower", method="circle", addCoef.col="black",
         number.cex=0.45, tl.cex = 0.55, tl.col = "black",
         col=brewer.pal(n=8, name="RdBu"), diag=FALSE)
```



The correlations range from low to high and in both directions. This is something we need to consider as we select predictors for our model.

Logistic regresssion

We can think about the probability or likelihood of a binary outcome as being between 0 and 1. Since the values of the outcome are then limited to 0 through 1, we don't apply standard linear regression. If we tried to do this, our fit may be problematic and even result in an impossible value (i.e., values < 0 or > 1). We need a model that restricts values to 0 through 1. The logistic regression is one such model.

Instead of selecting coefficients that minimized the squared error terms from the best fit line, like we used in linear regression, the coefficients in logistic regression are selected to maximize the likelihood of predicting a high probability for observations actually belonging to class 1 and predicting a low probability for observations actually belonging to class 0.

Assumptions of logistic regression: - The outcome is a binary or dichotomous variable like yes vs no, positive vs negative, 1 vs 0. - There is a linear relationship between the logit of the outcome and each predictor variables. The logit function is logit(p) = log(p/(1-p)), where p is the probabilities of the outcome. - There are no influential values (extreme values or outliers) in the continuous predictors. - There are no high intercorrelations (i.e. multicollinearity) among the predictors.

Similar to the previous lesson, we will split the data, fit a model and then examine the model output on train and test data. In this case, we will use the `glm` function, which is commonly used for fitting Generalized Linear Models, of which logistic regression is one form. We specify that we want to use logistic regression using the argument `family = "binomial"`. This returns an object of class "glm", which inherits from the class "lm". Therefore, it also includes attributes we can explore to learn about our model and its fit of our data.

A major difference is that logistic regression does not return a value for the observation's class, it returns an estimated probability of an observation's class membership. The probability ranges from 0 to 1 and value assignment to a class is based on a threshold. The default threshold is 0.5, but should be adjusted for the purpose of the prediction. Simple and multivariate versions of logistic regression are possible. Since we explored the difference with the linear regression, we will start this lesson with the multivariate model we ended with in the previous lesson.

Split the data

The data was provided to you after processing and cleaning, so we are able to skip these critical steps for this lesson. We start our modeling process by splitting our data into 75:25 train:test sets.

```
set.seed(439) #so we all get same random numbers
train <- sample(nrow(ckd), nrow(ckd) * 0.75)
test <- -train

ckd_train <- ckd[train, ] %>%
              select(-id)
```

```
## select: dropped one variable (id)
```

```
ckd_test <- ckd[test, ] %>%
              select(-id)
```

```
## select: dropped one variable (id)
```

Fit the model

We will fit a new model, modGLM, that uses SCr, BUN, and Kynurenine to predict Stage in the training set. As before, we will add the predicted probability values to the training set as a new variable, Stage_prob. The function `contrasts` shows what `R` is considering as the reference state for the prediction.

```
contrasts(ckd$stage) #what is R considering the reference? CKD3b: 0 = N, 1 = Y
```

```
##       CKD3b
## CKD2      0
## CKD3b     1
```

```
mod_glm <- glm(stage ~ scr + bun + kynurenine, data = ckd_train, family = "binomial")

# what is in .fitted? Log odds.
head(augment(mod_glm))
```

```
## # A tibble: 6 x 10
##   stage   scr   bun kynurenine .fitted  .resid .std.resid    .hat .sigma .cooksd
##   <fct> <dbl> <dbl>      <dbl>   <dbl>   <dbl>      <dbl>   <dbl>  <dbl>   <dbl>
## 1 CKD2   1.36    23       3.74   -2.15 -0.469     -0.476  0.0318   0.573 9.84e-4
## 2 CKD3b  2.17    41       9.68    6.12  0.0662     0.0663 0.00450  0.574 2.49e-6
## 3 CKD3b  2.69    47       4.07    6.49  0.0550     0.0551 0.00313  0.574 1.19e-6
## 4 CKD2   0.64    12       5.06   -5.67 -0.0828    -0.0830 0.00449  0.574 3.89e-6
## 5 CKD2   1.34    13       6.18   -4.74 -0.132     -0.133  0.0111   0.574 2.48e-5
## 6 CKD3b  1.8     33       4.03    1.46  0.646      0.663  0.0500   0.572 3.21e-3
```

```r
# If we want probabilities for comparison, then we need to predict the train using type = "response"
#add the predicted values to the train set and set the type argument to response
ckd_train <- ckd_train %>%
              mutate(stage_prob = predict(mod_glm, ckd_train,
                                          type = "response"))
```

```
## mutate: new variable 'stage_prob' (double) with 150 unique values and 0% NA
```

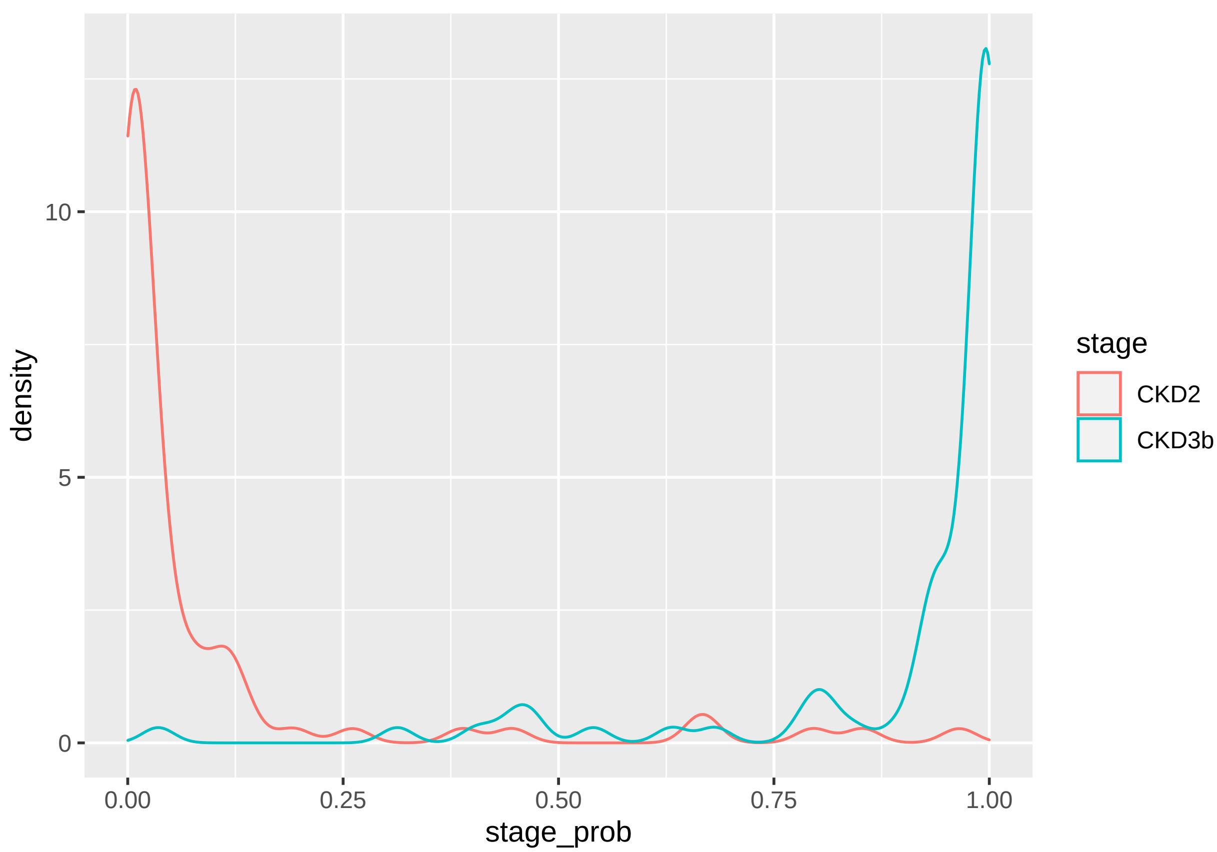
```r
#using threshold 0.5, convert probabilities to predicted stage
ckd_train$stage_pred<- ifelse(ckd_train$stage_prob > 0.5, "CKD3b", "CKD2")
```

How did the model do at predicting stage in our training data? We can calculate the accuracy of the model and plot the density of the predicted probabilities by class.

```r
#calculate accuracy, if == statement is TRUE, value = 1, otherwise = 0
mean(ckd_train$stage_pred == ckd_train$stage)
```

```
## [1] 0.9266667
```

```r
ggplot(ckd_train, aes(stage_prob, color = stage)) +
  geom_density()
```



Examining our model

Recalling the helpful functions we used from the `broom` package, we can examine our model. We see that the parameters for the logistic regression model are different than those we saw in the previous lesson on linear regression. R2 is not relevant for logistic regression. Instead, to compare models, we rely on parameters called AIC and BIC. These are the Akaike Information Criterion and the Bayesian Information Criterion. Each tries to balance model fit and parsimony and each penalizes differently for number of parameters. Models with the lowest AIC and lowest BIC are preferred.

Exercise 1:

Examine modGLM using the `glance()` and `tidy()` functions of the `broom` package. What is the AIC and BIC for this model? What are the coefficients for each term of the model?

End exercise

Examining collinearity

As mentioned before, we need to be careful when several predictors have strong correlation. Remember that we can calculate the variance inflation factor (VIF) for each model to determine how much the variance of a regression coefficient is inflated due to multicollinearity in the model. We want VIF values close to 1 (meaning no multicollinearity) and less than 5.

```
vif(mod_glm)
```

```
##        scr        bun kynurenine
##   1.119578   1.124932   1.005801
```

There does not seem to be a collinearity problem in our model.

Making predictions from our model

When we use the predict function on this model, it will predict the log(odds) of the Y variable. This is not what we ultimately want since we want to determine the predicted Stage. To convert it into prediction probability scores that are bound between 0 and 1, we specify type = "response".

```
#predict on test
table(ckd_test$stage) #CKD3b ~ 50%
```

```
##
##  CKD2 CKD3b
##    25    25
```

```
ckd_test$stage_prob <- predict(mod_glm, ckd_test, type = "response")
```

With the predicted probabilities, we can now apply a threshold and assign each row to either the CKD3b or CKD2 class, based on probability. We will start with a threshold of 0.5. We know the actual assignment from the Stage column (of this training data) so we can calculate the accuracy of our model to predict class.

```
ckd_test$stage_pred<- ifelse(ckd_test$stage_prob > 0.5, "CKD3b", "CKD2")
mean(ckd_test$stage_pred == ckd_test$stage) #0.9
```

```
## [1] 0.9
```

Exercise 2:

Select a different threshold and determine the accuracy of the model for that threshold setting.

End exercise

Build ROC curve as alternative to accuracy

Sometimes calculating the accuracy is not good enough to determine model performance (especially when there is class imbalance and accuracy can be misleading) and using a threshold of 0.5 may not be optimal. We can use the `pROC` package functions to build an ROC curve and find the area under the curve (AUC) and view the effects of changing the cutoff value on model performance.

```
# Convert Stage to numeric probability variable (0 or 1)
ckd_test <- ckd_test %>%
            mutate(stage_num = ifelse(stage == "CKD3b", 1, 0))
```

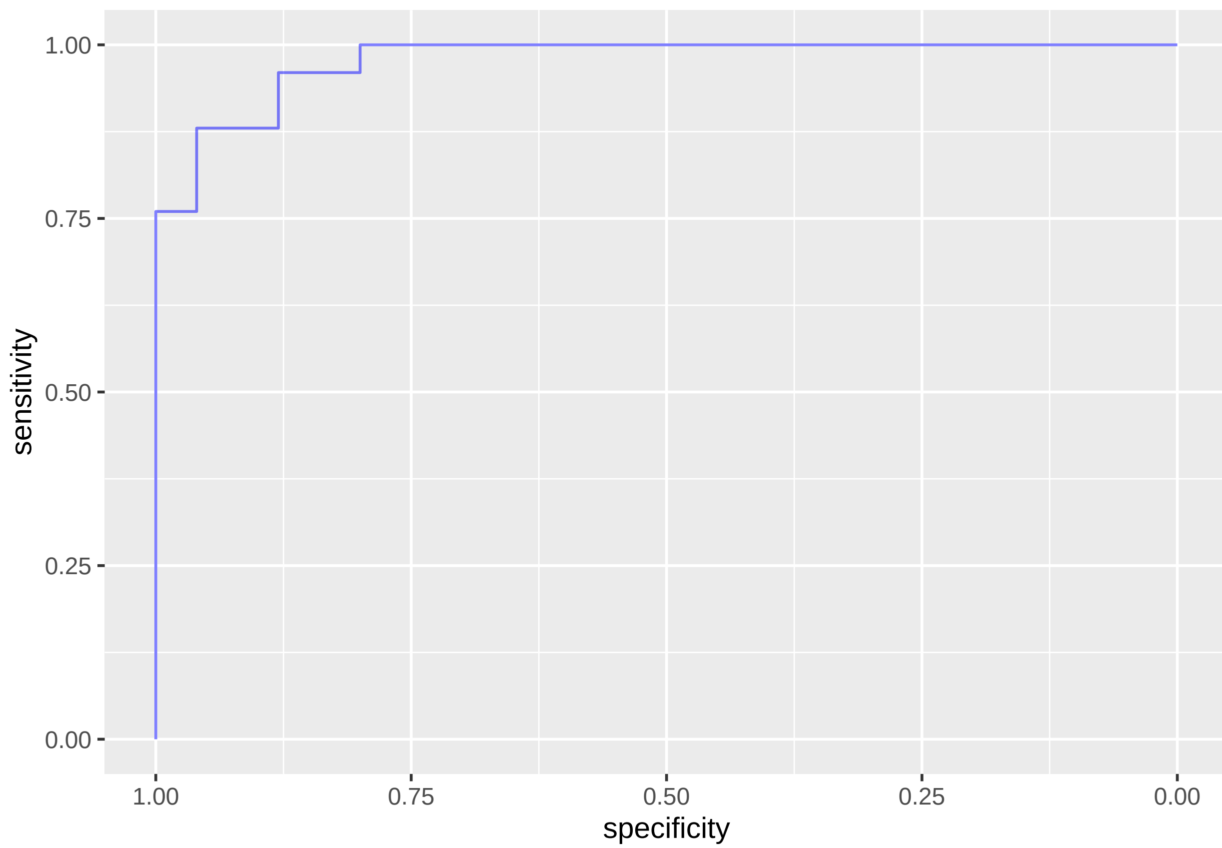
```
## mutate: new variable 'stage_num' (double) with 2 unique values and 0% NA
```

```
# Create a ROC curve object from columns of actual and predicted probabilities
roc_ckd <- roc(ckd_test$stage_num, ckd_test$stage_prob)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

```
# Plot the ROC curve object
ggroc(roc_ckd, alpha = 0.5, colour = "blue")
```




```r
# Calculate the area under the curve (AUC)
auc(roc_ckd) #0.9776
```

```
## Area under the curve: 0.9776
```

```r
# find the optimal thresholds for different criteria
coords(roc_ckd, "best")
```

```
##   threshold specificity sensitivity
## 1 0.3767651        0.88        0.96
## 2 0.7121563        0.96        0.88
```

As expected, we were able to build a strong classifier model. Most real-world situations have less separation than we found in this lesson. In those cases, one must consider the purpose of the classifier and weight the importance of false positives versus false negatives. The ROC curve is helpful to find the optimal cutoff in those cases. Additional calculations of a confusion matrix to determine the sensitivity and specificity of the model would also be warranted.

Acknowledgement

The data used in this lesson was simulated from a data set generated in collaboration with Dr. Ellen Brooks. Prior to simulation, the metabolomics data was processed and cleaned by Dr. David Lin. The lesson design was influenced by the DataCamp course: Supervised Learning in R: Regression.

Summary

- Logistic regression is a widely applied tool in predictive modeling and machine learning for classification problems.
- There are 4 primary assumptions in logistic regression that must be evaluated for a given model.
- Best practice is to randomly split data into train and test sets, used to fit and evaluate the model.
- Collinearity can be a problem with logistic regresion models.
- Logistic regression does not use R2, but relies on AIC as a metric of fit.
- The prediction accuracy of a classification model depends on the class balance and selected probability threshold. Consider AUC and other measures instead.
- As with any other application of ROC curves, optimal cut-off should be chosen according to the application of the classifier and the "costs" of false positives and false negatives

Bringing it all together

Working within your project

For the final project we are going to take advantage of the project structure we built at the beginning of the course and perform data exploration on the large mass spec data set we have encountered in various lessons in the course.

Project setup

1. Navigate to the Session menu (top bar) and select "New Session" to start a new RStudio window.
2. Use the project shortcut menu on the top right of the RStudio window (down arrow next to current project title) and select your msacl-201-project from the project list.
3. Locate your class data file (data.zip) and place an uncompressed version into your project directory. It should replace the existing (but empty) data folder you built in lesson 1.
4. Copy the R Markdown file for this lesson ("11 - Bringing It All Together.Rmd") into the "src" folder in your msacl-201-project directory. Normally when you open up a project you will start a new R Markdown or other R file from scratch, but we've included some starter code to get you started in the lesson Rmd file.
5. If you haven't already installed them, install the here and renv packages: `install.packages(c("renv", "here"), dependencies = TRUE)`.
6. Initialize your working directory using the `here::here()` command. This should set your reference directory to the main project directory. (The `set_here()` command can be used to force this to another directory if needed.)
7. Initialize your project specific repostories using the `renv::init()` command. This will capture the current versions of packages used by files in your project. (Package updates or additions can be captured by running `renv::snapshot()`.)

End project setup

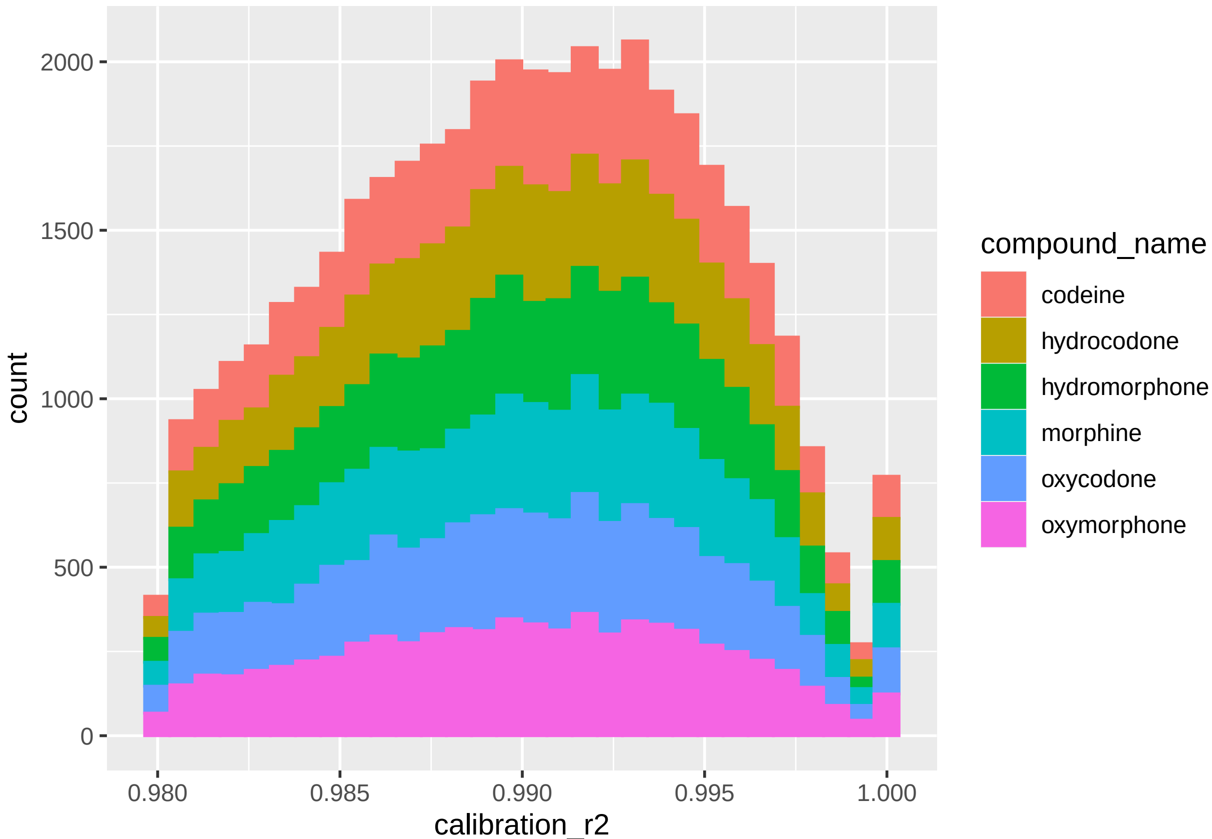
From Import to Graph

Let's pull all our data together and explore the big data set. What follows are the steps to replicate the discovery of one particular problem in the mock data: excessively good R^2 data.

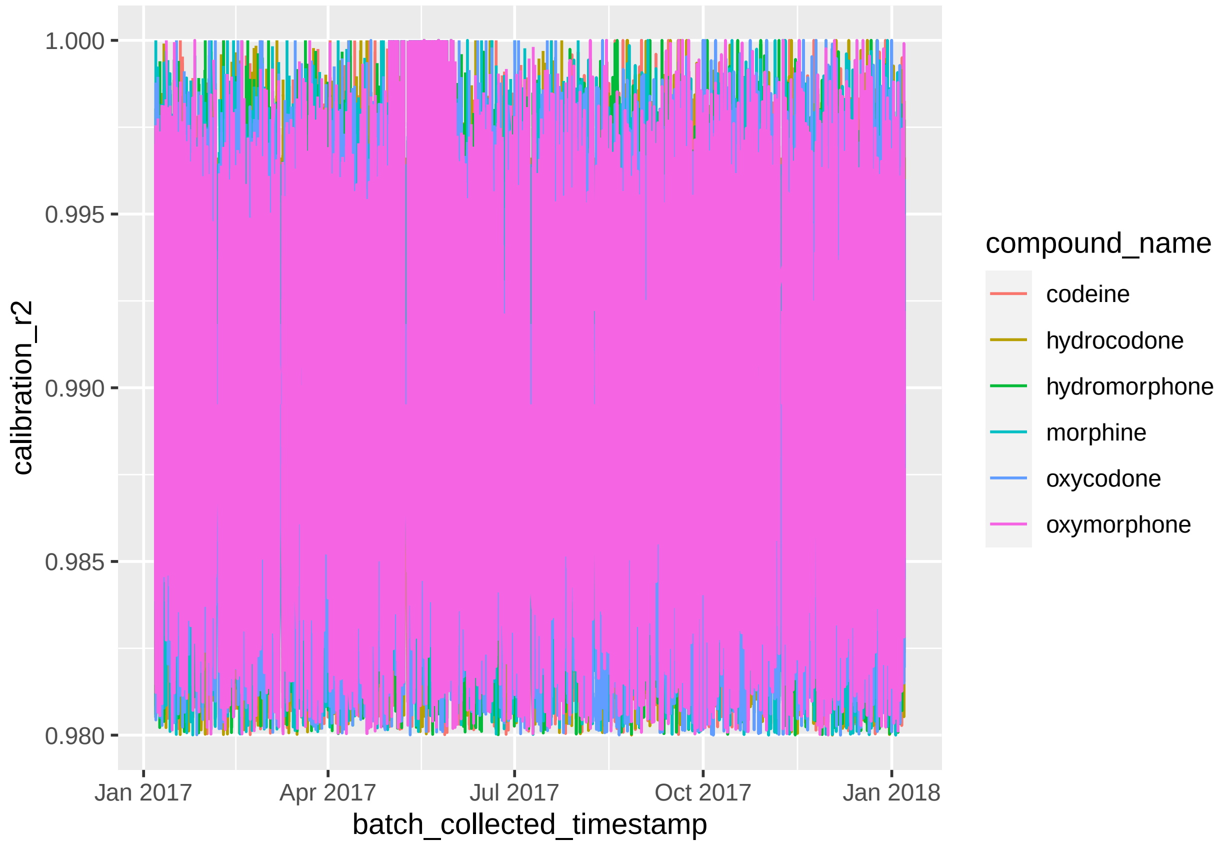
```
all_batches <- dir_ls(here("data"), glob = "*_b.csv") %>%
  #list.files("data/", pattern = "_b.csv$") %>%
  #file.path("data", .) %>%
```

```
  map_dfr(read_csv) %>%
  clean_names()
```

```
ggplot(all_batches, aes(x = calibration_r2, color = compound_name, fill = compound_name)) +
  geom_histogram(bins = 30)
```




```
ggplot(all_batches, aes(x = batch_collected_timestamp, y = calibration_r2, color = compound_name)) +
  geom_line()
```

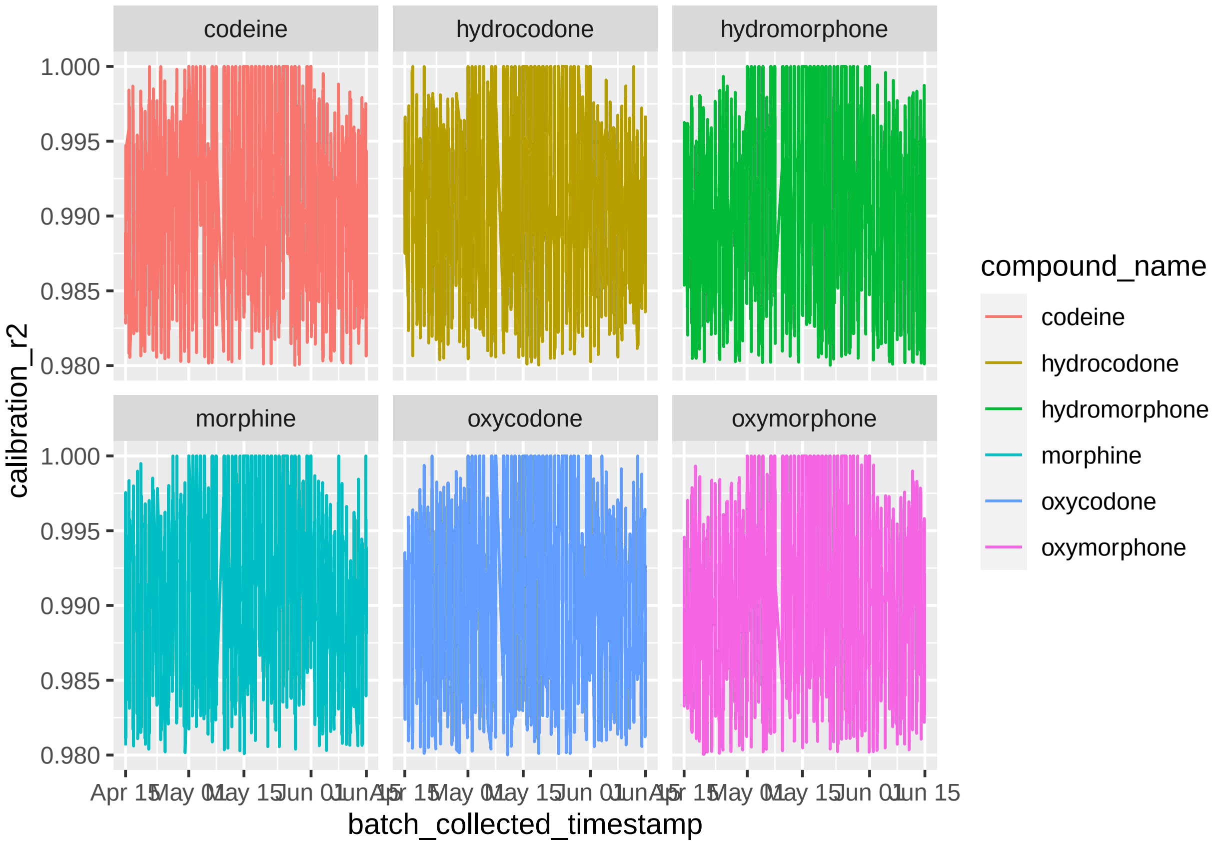


There's something interesting going on with the R^2 values in the month of May, where a large number of them report a value of 1.0 – a perfect fit. Let's focus on that month, and spread out the data so we can clarify whether it's all compounds or just oxymorphone (the magenta color on top).

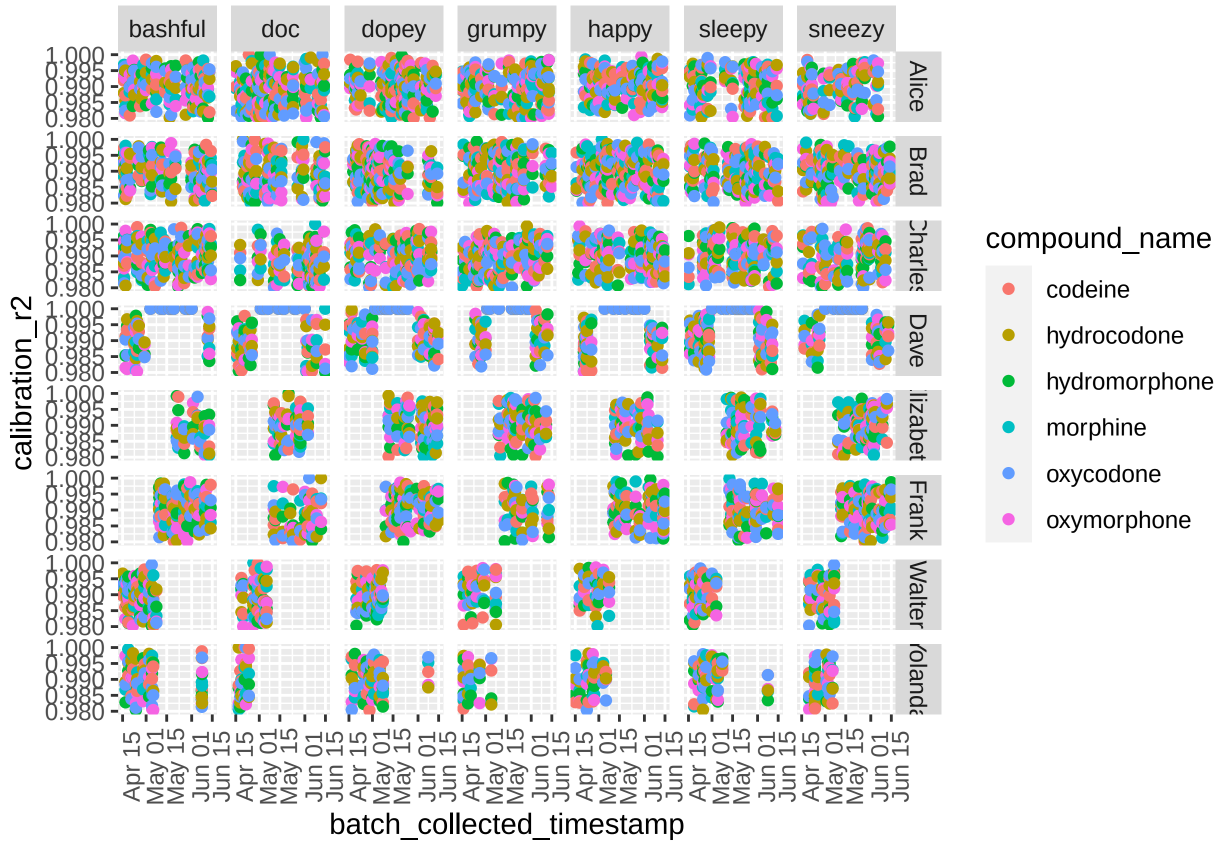
```
may_plot <- all_batches %>%
  filter(batch_collected_timestamp > ymd("2017-04-15"), batch_collected_timestamp < ymd("2017-06-15")) %
  ggplot(aes(x = batch_collected_timestamp, y = calibration_r2, color = compound_name))
```

```
## filter: removed 35,970 rows (83%), 7,200 rows remaining
```

```
may_plot +
  geom_line() +
  facet_wrap(~ compound_name)
```




```r
may_plot +
  geom_point() +
  facet_grid(reviewer_name ~ instrument_name) +
  theme(axis.text.x = element_text(angle = 90))
```



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

From Graph to Result

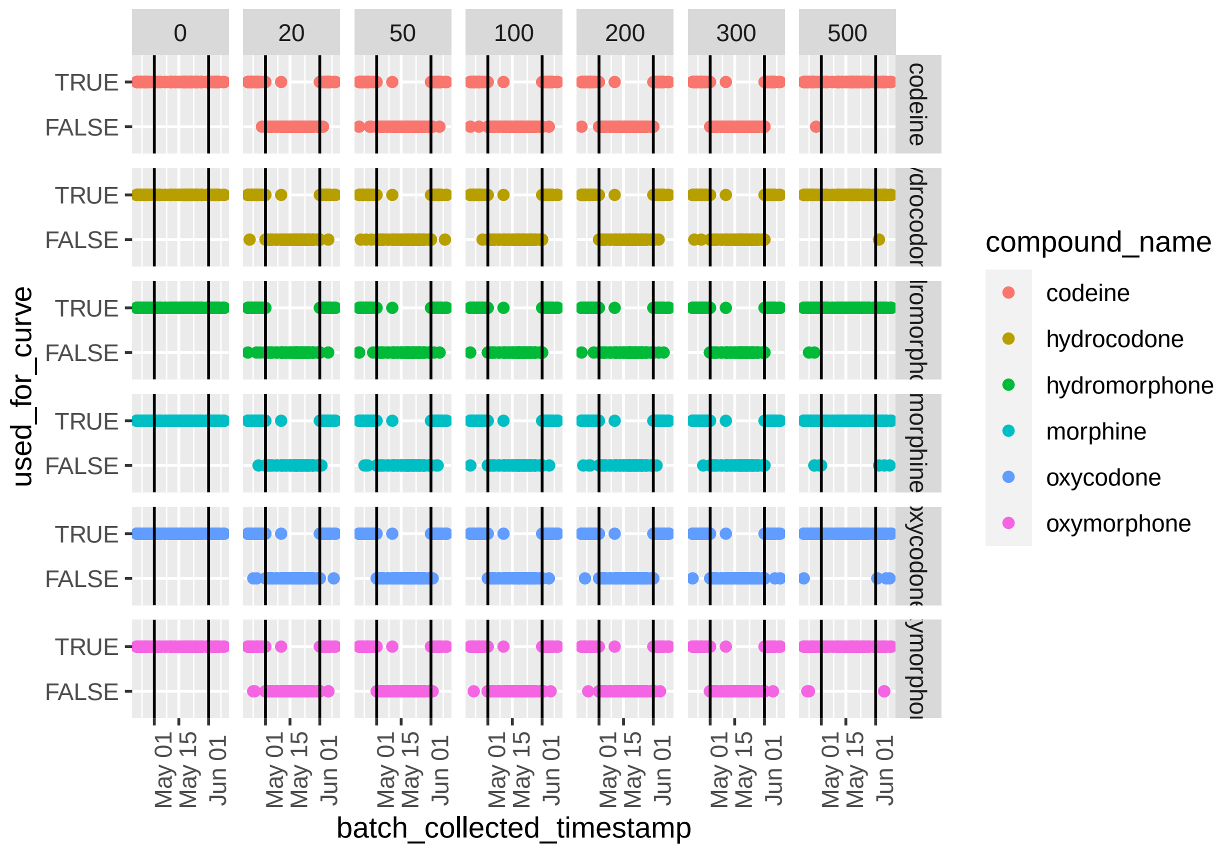
Based on the batch-level data, we can see that 'Dave' – and apparently only Dave – has perfect R^2 values on every batch of data he reviewed throughout the month of May. Digging deeper will require merging information from the batch level with information at the sample (and possibly peak) level.

```
all_samples <- dir_ls(here("data"), glob = "*_s.csv") %>%
  map_dfr(read_csv) %>%
  clean_names()
daves_data <- all_samples %>%
  left_join(select(all_batches, -calibration_slope, -calibration_intercept)) %>%
  filter(
    batch_collected_timestamp > ymd("2017-04-20"),
    batch_collected_timestamp < ymd("2017-06-10"),
    sample_type == "standard",
    reviewer_name == "Dave"
  )
```

The following plots of `daves_data` 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.

```
daves_data %>%
  ggplot(aes(x = batch_collected_timestamp, y = used_for_curve, color = compound_name)) +
```

```
  geom_point() +
  facet_grid(compound_name ~ expected_concentration) +
  geom_vline(xintercept = as.numeric(as_datetime(c("2017-05-01", "2017-06-01"))),
    linetype = 1,
    colour = "black") +
  theme(axis.text.x = element_text(angle = 90))
```



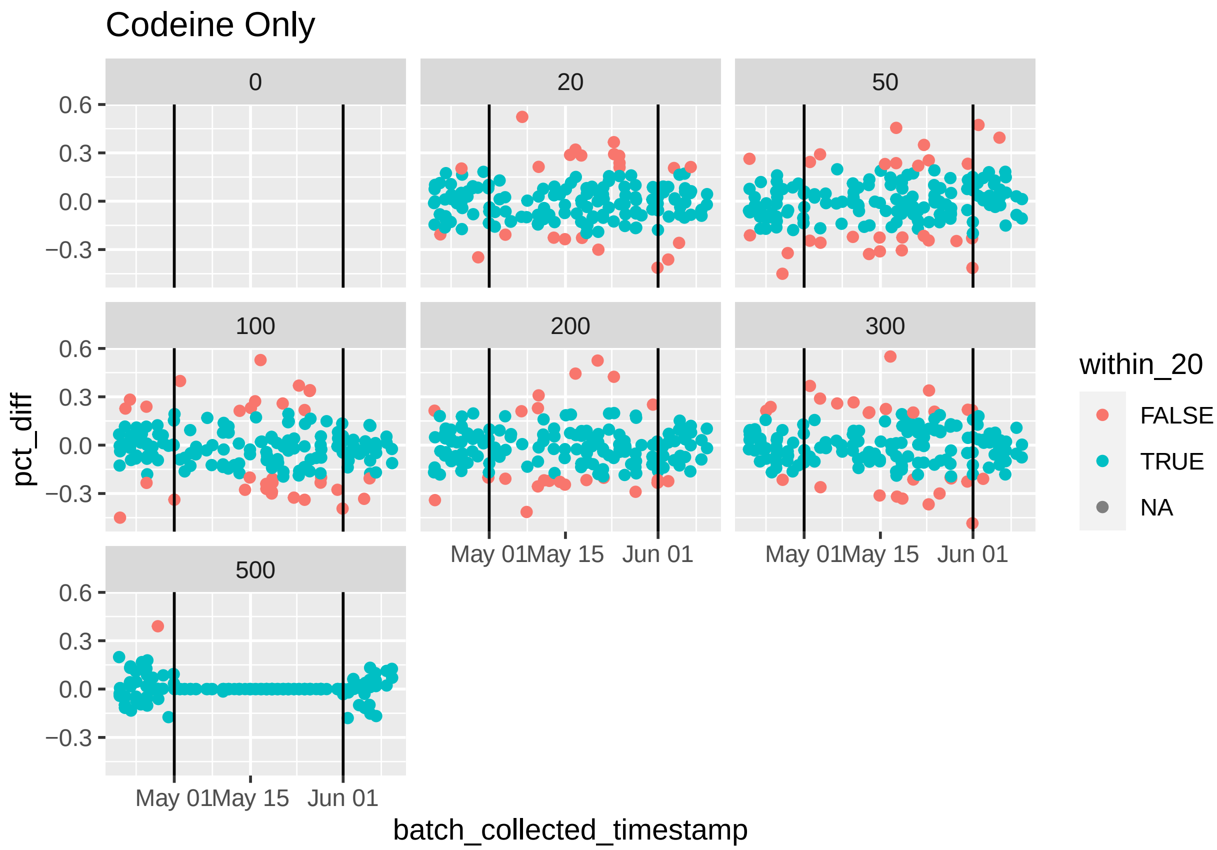

```
daves_data <- daves_data %>%
  mutate(pct_diff = (concentration - expected_concentration) / expected_concentration,
         within_20 = abs(pct_diff) <= 0.2)
```

```
## mutate: new variable 'pct_diff' (double) with 5,318 unique values and 14% NA

##         new variable 'within_20' (logical) with 3 unique values and 14% NA
```

```
daves_data %>%
  filter(compound_name == "codeine") %>%
  ggplot(aes(x = batch_collected_timestamp, y = pct_diff, color = within_20)) +
  geom_point() +
  facet_wrap(~ expected_concentration) +
  ggtitle("Codeine Only") +
  geom_vline(xintercept = as.numeric(as_datetime(c("2017-05-01", "2017-06-01"))),
    linetype = 1,
    colour = "black")
```

```
## filter: removed 5,740 rows (83%), 1,148 rows remaining
```



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 problems based on ion ratios

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. Within the 1 year of data provided, identify any potential issues with the data that might suggest problems with workflows, training, or other issues that could impact quality of the results.

This is a very open ended exercise, so consider the following areas to explore:

- Consider all of the qualitative data elements that could influence the observed ion ratios: visualize data as a function of combinations of these variables
- Time-based trending can make abrupt changes very obvious
- Data from all three file types (batches, samples, and peaks) are important in isolating isssues
- When there are a lot of data point to visualize, consider aggregation and/or visualizations that represent statistical summaries or fitting

End of Exercise