MSACL Data Science 201 Textbook

Patrick Mathias
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 reproducible research?

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

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

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

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

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.

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. Review the contents of your current folder using `list.files()`
4. Now try to set your working directory using `setwd("test_dir")`. What happened?
5. Create a new test directory using `dir.create("test_dir")`
6. Review your current directory
7. Set your directory to the test directory you just created
8. Using the Files window (bottom right in RStudio, click on **Files** tab if on another tab), navigate to the test directory you just created and list the files. *Pro tip: The More menu here has shortcuts to set the currently displayed directory as your working directory and to navigate to the current working directory*
9. Navigate back to one level above the directory you created using `setwd("..")` and list the files
10. Delete the directory you created using the `unlink()` function. Learn more about how to use the function by reviewing the documentation: `?unlink`. Pay special attention to comments about deleting directories.

End Exercise

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

Exercise 3

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

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

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

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".

Adopt a style convention for coding

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

There is another solution to this problem that has tighter integration with RStudio: the packrat package. Packrat sets up a private package library and helps manage the version of each package you use in a project. It is very similar to checkpoint conceptually, but rather than locking packages to a date, you are capturing specific versions of each package. In some ways this is a better way to manage packages since the versions are more transparent, but there is more overhead in setting this up. There is a helpful walkthrough here: http://rstudio.github.io/packrat/walkthrough.html. In addition, there is integration with the RStudio IDE, which is detailed here: http://rstudio.github.io/packrat/rstudio.html.

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.

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.

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.

Hands-on with Git

We have placed the contents of this course into a Git repository (central hub - we will discuss in more detail later) 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.

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 4

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.

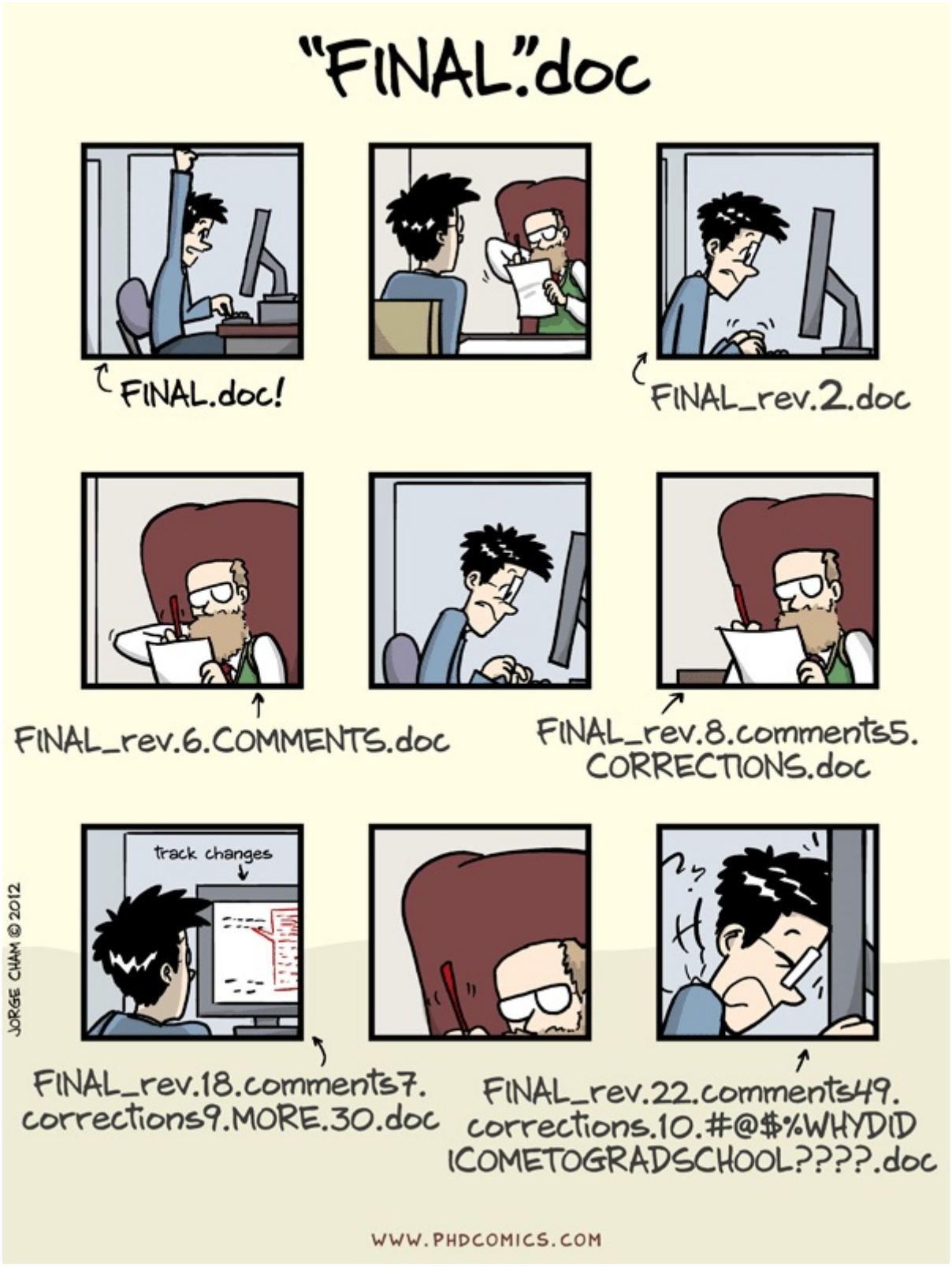


Figure 1: One of many justifications for using version control. Source: phdcomics.com

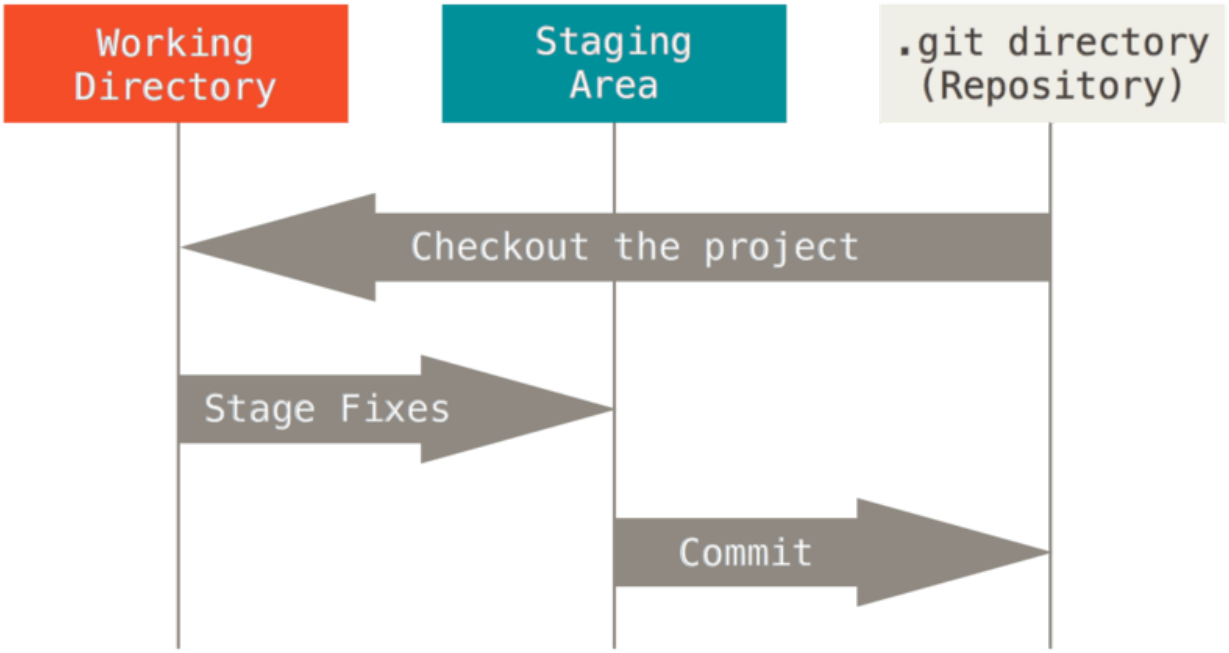


Figure 2: Git basic workflow. Source: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

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. 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.
5. Under the File menu within the RStudio application, select "New Project".
6. Select "Version Control" in the first Create Project prompt.
7. Select "Git" in the next Create Project from Version Control prompt.
8. 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.
9. 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 First 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_first_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. Now press the "Commit" button in the Git window. A new window will pop up showing the file(s) in the Git window, with a Commit message window on the right and the code below. Add "My first 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

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.

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 an email 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.

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 5

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

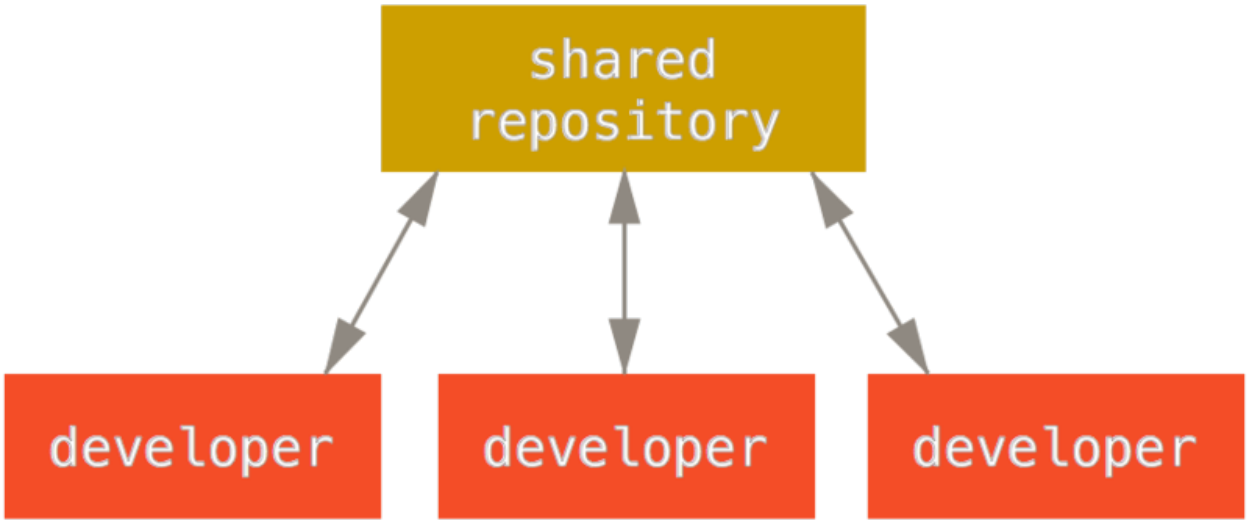


Figure 3: Centralized workflow with Git. Credit: https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

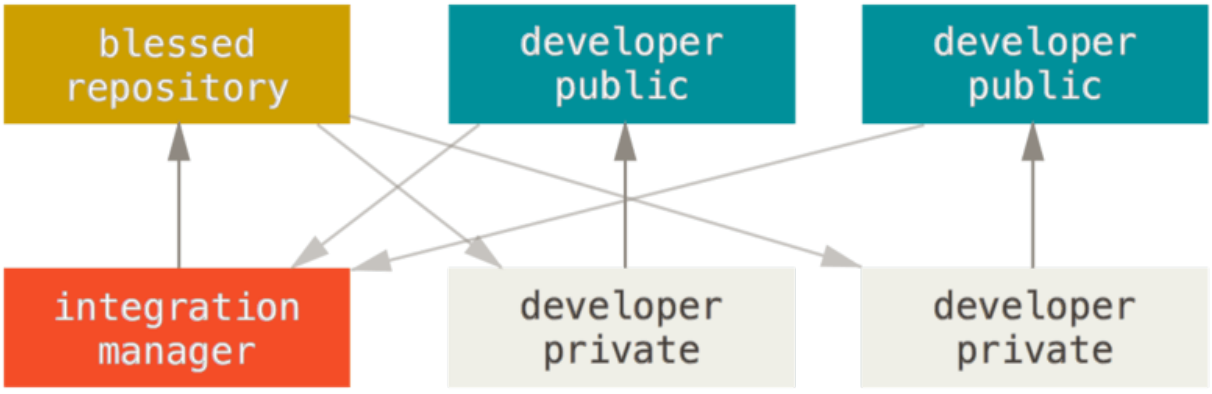


Figure 4: Integration manager workflow with Git. Credit: https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

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.

Optional steps: 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.

Optional steps: 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.

End Exercise

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

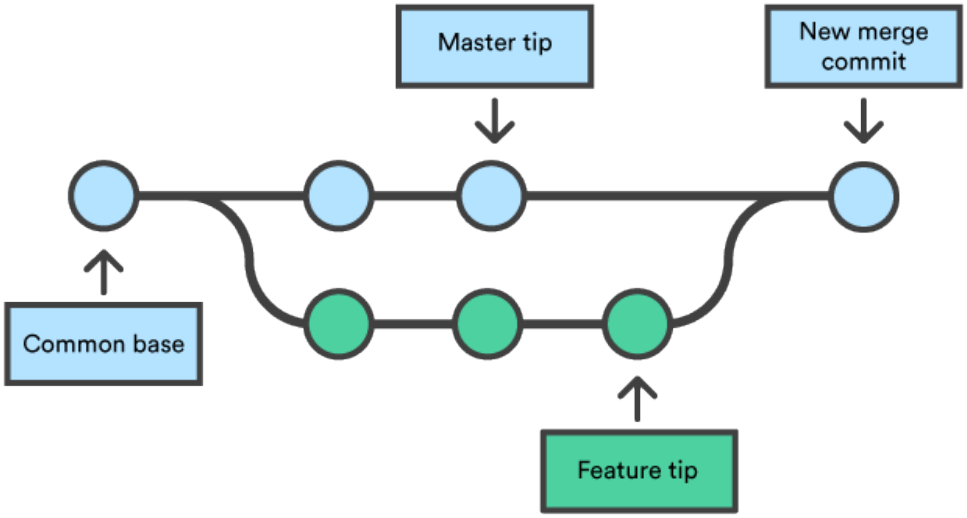


Figure 5: Branching in Git. Credit: https://www.atlassian.com/git/tutorials/using-branches/git-merge

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.

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

- 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
  - Using a version control system to track work and collaborate with others

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

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

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
```

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))
```

```
````
````

```
## [1] 20
```

And this is how the code chunk looks by default:

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

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

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

Flexibility in reporting: types of knitr output

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

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

Exercise 1

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

End Exercise

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

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

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

Exercise 2

1. Click the **Knit** button

2. You are being prompted to save the .Rmd file. Choose the "src" folder of your project and name the file sample_markdown_document
3. RStudio should produce output in .html format and display
4. Click the Open in Browser window and the same output should open in your default internet browser
5. If you find the folder you saved the .Rmd file there should also be a .html file you can open as well
6. Now, instead of hitting the **Knit** button, select the down arrow adjacent to it and click Knit to PDF
7. Repeat the previous step but knit to a Word document

End Exercise

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 3

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 2 file: "02 - 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 2 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

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

Base functions for reading and writing files

Reading files

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

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

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

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

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

Exercise 1

Let's run through the base reading function with a csv.

1. Use the base `read.csv()` function to read the "2017-01-06_s.csv" file in the data folder into a data frame.

```
base_load <- read.csv("data/2017-01-06_s.csv")
```

2. What is the internal structure of the object? (Recall the `str()` command to quickly view the structure.)

```
str(base_load)
```

```
## 'data.frame':    187200 obs. of  10 variables:
##  $ batchName           : Factor w/ 600 levels "b100302","b101197",..: 540 540 540 540 540 540 540 54
##  $ sampleName          : Factor w/ 24128 levels "s000001","s000002",..: 6605 6605 6605 6605 6605 660
##  $ compoundName        : Factor w/ 6 levels "codeine","hydrocodone",..: 4 3 6 1 2 5 4 3 6 1 ...
##  $ ionRatio            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ response            : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ concentration       : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ sampleType          : Factor w/ 4 levels "blank","qc","standard",..: 1 1 1 1 1 1 3 3 3 3 ...
##  $ expectedConcentration: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ usedForCurve        : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ samplePassed        : logi  FALSE TRUE TRUE TRUE TRUE TRUE ...
```

3. Summarize the data. (Recall the `summary()` function to view column types and characteristics about the data.)

```
summary(base_load)
```

```
##    batchName        sampleName           compoundName      ionRatio
##  b100302:   312   s035001:    24   codeine      :31200   Min.   :0.0000
##  b101197:   312   s035002:    24   hydrocodone  :31200   1st Qu.:0.0000
##  b101972:   312   s035003:    24   hydromorphone:31200   Median :0.8537
##  b102100:   312   s035004:    24   morphine     :31200   Mean   :0.6689
##  b102508:   312   s035005:    24   oxycodone    :31200   3rd Qu.:1.2479
##  b103050:   312   s035006:    24   oxymorphone  :31200   Max.   :1.7199
##  (Other):185328   (Other):187056
##     response      concentration       sampleType     expectedConcentration
##  Min.   :0.0000   Min.   :  0.00   blank   :  7200   Min.   :  0.00
##  1st Qu.:0.0000   1st Qu.:  0.00   qc      : 10800   1st Qu.:  0.00
##  Median :0.3128   Median : 44.24   standard: 25200   Median :  0.00
##  Mean   :0.9647   Mean   :135.83   unknown :144000   Mean   : 35.77
##  3rd Qu.:1.8650   3rd Qu.:264.20                     3rd Qu.:  0.00
##  Max.   :5.9517   Max.   :857.16                     Max.   :500.00
##
##  usedForCurve    samplePassed
##  Mode :logical   Mode :logical
##  FALSE:162803    FALSE:4678
##  TRUE :24397     TRUE :182522
##
##
##
##
```

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

```
base_load_nofactors <- read.csv("data/2017-01-06_s.csv",
                                stringsAsFactors = FALSE)
str(base_load_nofactors)
```

```
## 'data.frame':    187200 obs. of  10 variables:
##  $ batchName            : chr  "b802253" "b802253" "b802253" "b802253" ...
##  $ sampleName           : chr  "s253001" "s253001" "s253001" "s253001" ...
##  $ compoundName         : chr  "morphine" "hydromorphone" "oxymorphone" "codeine" ...
##  $ ionRatio             : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ response             : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ concentration        : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ sampleType           : chr  "blank" "blank" "blank" "blank" ...
##  $ expectedConcentration: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ usedForCurve         : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ samplePassed         : logi  FALSE TRUE TRUE TRUE TRUE TRUE ...
```

```
summary(base_load_nofactors)
```

```
##   batchName          sampleName        compoundName          ionRatio
##  Length:187200      Length:187200      Length:187200      Min.   :0.0000
##  Class :character   Class :character   Class :character   1st Qu.:0.0000
##  Mode  :character   Mode  :character   Mode  :character   Median :0.8537
##                                                           Mean   :0.6689
##                                                           3rd Qu.:1.2479
##                                                           Max.   :1.7199
##     response      concentration     sampleType
##  Min.   :0.0000   Min.   :  0.00   Length:187200
##  1st Qu.:0.0000   1st Qu.:  0.00   Class :character
##  Median :0.3128   Median : 44.24   Mode  :character
##  Mean   :0.9647   Mean   :135.83
##  3rd Qu.:1.8650   3rd Qu.:264.20
##  Max.   :5.9517   Max.   :857.16
##  expectedConcentration usedForCurve    samplePassed
##  Min.   :  0.00        Mode :logical   Mode :logical
##  1st Qu.:  0.00        FALSE:162803    FALSE:4678
##  Median :  0.00        TRUE :24397     TRUE :182522
##  Mean   : 35.77
##  3rd Qu.:  0.00
##  Max.   :500.00
```

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

End Exercise

Writing files

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

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
```

```
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

There are a few other important parameters: - `sep` indicates the field separator ("\t" for tab) - `row.names` is set to TRUE by default - in general this makes for an ugly output file becuase the first column shows the row number (I almost always set this to FALSE when I use the base function) - `na` indicates the string to use for missing data and is set to R's standard of "NA" by default - `append` can be set to TRUE if you would like to append your data frame/matrix to an existing file

Speeding things up with the *readr* package

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

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

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

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

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

Exercise 2

Now let's run through using the readr function for a csv: 1. Use the `read_csv()` function to read the "2017-01-06_s.csv" file into a data frame.

```
readr_load <- read_csv("data/2017-01-06_b.csv")
```

```
## 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 = "")
## )
```

2. What is the internal structure of the object?

```
str(readr_load)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    3600 obs. of  11 variables:
##  $ batchName              : chr  "b802253" "b802253" "b802253" "b802253" ...
##  $ instrumentName         : chr  "doc" "doc" "doc" "doc" ...
##  $ compoundName           : chr  "morphine" "hydromorphone" "oxymorphone" "codeine" ...
##  $ calibrationSlope       : num  0.00775 0.00768 0.00798 0.00819 0.00656 ...
##  $ calibrationIntercept   : num  -5.76e-06 -4.38e-05 5.15e-06 -5.02e-06 4.56e-05 1.26e-05 -7.96e-05 4
##  $ calibrationR2          : num  0.997 0.981 0.984 0.981 0.991 ...
##  $ batchPassed            : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
##  $ reviewerName           : chr  "Xavier" "Xavier" "Xavier" "Xavier" ...
##  $ batchCollectedTimestamp: POSIXct, format: "2017-01-06 21:40:00" "2017-01-06 21:40:00" ...
##  $ reviewStartTimestamp   : POSIXct, format: "2017-01-07 13:43:00" "2017-01-07 13:43:00" ...
##  $ reviewCompleteTimestamp: POSIXct, format: "2017-01-07 14:09:00" "2017-01-07 14:09:00" ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 11
##   .. ..$ batchName              : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ instrumentName         : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ compoundName           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ calibrationSlope       : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ calibrationIntercept   : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ calibrationR2          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ batchPassed            : list()
##   .. .. ..- attr(*, "class")= chr  "collector_logical" "collector"
##   .. ..$ reviewerName           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ batchCollectedTimestamp:List of 1
##   .. .. ..$ format: chr ""
##   .. .. ..- attr(*, "class")= chr  "collector_datetime" "collector"
##   .. ..$ reviewStartTimestamp   :List of 1
##   .. .. ..$ format: chr ""
##   .. .. ..- attr(*, "class")= chr  "collector_datetime" "collector"
##   .. ..$ reviewCompleteTimestamp:List of 1
##   .. .. ..$ format: chr ""
##   .. .. ..- attr(*, "class")= chr  "collector_datetime" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##   ..- attr(*, "class")= chr "col_spec"
```

3. Summarize the data.

```
summary(readr_load)
```

```
##   batchName         instrumentName     compoundName
```

```
##  Length:3600        Length:3600        Length:3600
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
##  calibrationSlope   calibrationIntercept calibrationR2    batchPassed
##  Min.   :0.003172   Min.   :-9.510e-05   Min.   :0.9800   Mode:logical
##  1st Qu.:0.006794   1st Qu.:-2.160e-05   1st Qu.:0.9860   TRUE:3600
##  Median :0.007060   Median : 6.965e-08   Median :0.9902
##  Mean   :0.007107   Mean   : 7.202e-08   Mean   :0.9899
##  3rd Qu.:0.007351   3rd Qu.: 2.180e-05   3rd Qu.:0.9938
##  Max.   :0.009626   Max.   : 1.082e-04   Max.   :1.0000
##  reviewerName       batchCollectedTimestamp
##  Length:3600        Min.   :2017-01-06 20:08:00
##  Class :character   1st Qu.:2017-01-13 22:55:15
##  Mode  :character   Median :2017-01-21 11:00:30
##                     Mean   :2017-01-21 10:58:03
##                     3rd Qu.:2017-01-28 23:24:30
##                     Max.   :2017-02-05 01:54:00
##  reviewStartTimestamp          reviewCompleteTimestamp
##  Min.   :2017-01-07 09:08:00   Min.   :2017-01-07 09:35:00
##  1st Qu.:2017-01-14 12:05:45   1st Qu.:2017-01-14 12:24:15
##  Median :2017-01-21 23:18:30   Median :2017-01-21 23:41:30
##  Mean   :2017-01-21 23:24:24   Mean   :2017-01-21 23:54:28
##  3rd Qu.:2017-01-29 11:17:15   3rd Qu.:2017-01-29 11:55:00
##  Max.   :2017-02-05 13:49:00   Max.   :2017-02-05 14:15:00
```

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.

```
readr_load_factors <- read_csv("data/2017-01-06_b.csv",
                               col_types = cols(
                                 compoundName = col_factor(NULL),
                                 sampleType = col_factor(NULL)
                                 )
                               )
```

```
## Warning: The following named parsers don't match the column names:
## sampleType
```

```
summary(readr_load_factors)
```

```
##   batchName          instrumentName            compoundName
##  Length:3600        Length:3600        morphine     :600
##  Class :character   Class :character   hydromorphone:600
##  Mode  :character   Mode  :character   oxymorphone  :600
##                                        codeine      :600
##                                        hydrocodone  :600
##                                        oxycodone    :600
##  calibrationSlope   calibrationIntercept calibrationR2    batchPassed
##  Min.   :0.003172   Min.   :-9.510e-05   Min.   :0.9800   Mode:logical
##  1st Qu.:0.006794   1st Qu.:-2.160e-05   1st Qu.:0.9860   TRUE:3600
##  Median :0.007060   Median : 6.965e-08   Median :0.9902
```

```
##  Mean   :0.007107   Mean   : 7.202e-08   Mean   :0.9899
##  3rd Qu.:0.007351   3rd Qu.: 2.180e-05   3rd Qu.:0.9938
##  Max.   :0.009626   Max.   : 1.082e-04   Max.   :1.0000
##  reviewerName       batchCollectedTimestamp
##  Length:3600        Min.   :2017-01-06 20:08:00
##  Class :character   1st Qu.:2017-01-13 22:55:15
##  Mode  :character   Median :2017-01-21 11:00:30
##                     Mean   :2017-01-21 10:58:03
##                     3rd Qu.:2017-01-28 23:24:30
##                     Max.   :2017-02-05 01:54:00
##  reviewStartTimestamp          reviewCompleteTimestamp
##  Min.   :2017-01-07 09:08:00   Min.   :2017-01-07 09:35:00
##  1st Qu.:2017-01-14 12:05:45   1st Qu.:2017-01-14 12:24:15
##  Median :2017-01-21 23:18:30   Median :2017-01-21 23:41:30
##  Mean   :2017-01-21 23:24:24   Mean   :2017-01-21 23:54:28
##  3rd Qu.:2017-01-29 11:17:15   3rd Qu.:2017-01-29 11:55:00
##  Max.   :2017-02-05 13:49:00   Max.   :2017-02-05 14:15:00
```

End Exercise

For reference we can compare the time required to run the base `read.csv()` function with the readr `read_csv()` function using `system.time()`.

Time to read with base:

```
system.time(base_load <- read.csv("data/2017-01-06_p.csv"))
```

Time to read with readr:

```
system.time(readr_load <- read_csv("data/2017-01-06_p.csv"))
```

Dealing with Excel files (gracefully)

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

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

Exercise 3

You might be able to guess what comes next: we'll read in an Excel file. 1. Use the `read_excel()` function to read the "orders_data_set.xlsx" file into a data frame 1. View a summary of the imported data 1. Now read in only the first 5 columns using the `range` parameter 1. Review the first 6 lines of the imported data

```
readxl_load <- read_excel("data/orders_data_set.xlsx")
summary(readxl_load)
```

```
##     Order ID        Patient ID     Description         Proc Code
##  Min.   : 10002   Min.   :500001   Length:45002       Length:45002
```

```
##  1st Qu.: 32669   1st Qu.:503350   Class :character   Class :character
##  Median : 55246   Median :506862   Mode  :character   Mode  :character
##  Mean   : 55133   Mean   :506897
##  3rd Qu.: 77627   3rd Qu.:510421
##  Max.   :100000   Max.   :513993
##
##  ORDER_CLASS_C_DESCR  LAB_STATUS_C   LAB_STATUS_C_DESCR ORDER_STATUS_C
##  Length:45002        Min.   :1.000   Length:45002       Min.   :2.000
##  Class :character    1st Qu.:3.000   Class :character   1st Qu.:5.000
##  Mode  :character    Median :3.000   Mode  :character   Median :5.000
##                      Mean   :3.061                      Mean   :4.783
##                      3rd Qu.:3.000                      3rd Qu.:5.000
##                      Max.   :5.000                      Max.   :5.000
##                      NA's   :7152                       NA's   :18
##  ORDER_STATUS_C_DESCR REASON_FOR_CANC_C REASON_FOR_CANC_C_DESCR
##  Length:45002         Min.   :   1.0    Length:45002
##  Class :character     1st Qu.:  11.0    Class :character
##  Mode  :character     Median :  11.0    Mode  :character
##                       Mean   : 437.2
##                       3rd Qu.:1178.0
##                       Max.   :1178.0
##                       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   Length:45002
##  1st Qu.:2017-09-15 23:32:30   Class :character
##  Median :2017-10-12 14:22:00   Mode  :character
##  Mean   :2017-10-11 06:52:47
##  3rd Qu.:2017-11-02 09:39:00
##  Max.   :2017-12-29 22:24:00
##  NA's   :7791
```

```
readxl_load_subset <- read_excel("data/orders_data_set.xlsx", range = cell_cols("A:E"))
head(readxl_load_subset)
```

```
## # A tibble: 6 x 5
##   `Order ID` `Patient ID` Description        `Proc Code` ORDER_CLASS_C_DE~
##        <dbl>        <dbl> <chr>              <chr>       <chr>
## 1      19766       511388 PROTHROMBIN TIME   PRO         Normal
## 2      88444       511388 BASIC METABOLIC P~ BMP         Normal
## 3      40477       508061 THYROID STIMULATI~ TSH         Normal
## 4      97641       508061 T4, FREE           T4FR        Normal
## 5      99868       505646 COMPREHENSIVE MET~ COMP        Normal
## 6      31178       505646 GLUCOSE SERUM, FA~ GLUF        Normal
```

End Exercise

Importing dirty data

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

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

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

```
# install.packages("janitor", dependencies = TRUE) # uncomment to install if needed
# the development version of janitor handles cleaning names better than the current CRAN version
library(janitor)
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>
```

Now we'll do a quick tabulation to count the different order classes in this orders data set:

```
readxl_load_cleaned %>% tabyl(order_class_c_descr)
```

```
##  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
```

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 effortless 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 similiarly 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 compabibility 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  1.68    0.0316  0.785  0.739
##  2  1.33    0.691  -0.357 -0.260
##  3 -0.664   0.172  -0.216  0.957
##  4  0.449  -0.601  -0.407 -0.319
##  5 -0.453   0.957   1.30  -0.200
##  6 -0.0148  0.0554 -0.744 -0.286
##  7  0.797  -2.40   -0.702 -0.961
##  8  0.936   1.42    1.11  -0.136
##  9  2.05   -0.643  -0.891  1.19
## 10 -1.42    0.128   0.274 -0.0548
```

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
##  0.46896920 -0.01907896  0.01563660  0.06739017
```

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 arent' arguments.

```
all_samples <- list.files("data", pattern = "_s.csv") %>%
  file.path("data", .) %>%
  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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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_integer(),
##   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
##  Min.   :0.0000   Min.   :  0.00   Length:2244840
##  1st Qu.:0.0000   1st Qu.:  0.00   Class :character
##  Median :0.2982   Median : 42.55   Mode  :character
##  Mean   :0.9658   Mean   :134.46
##  3rd Qu.:1.8593   3rd Qu.:261.81
##  Max.   :9.2258   Max.   :860.59
##  expected_concentration used_for_curve  sample_passed
##  Min.   :  0.00         Mode :logical   Mode :logical
##  1st Qu.:  0.00         FALSE:1956363   FALSE:57190
##  Median :  0.00         TRUE :288477    TRUE :2187650
##  Mean   : 35.77
##  3rd Qu.:  0.00
##  Max.   :500.00
```

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

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

subsets of the spreadsheet
- The janitor package can help with cleaning up irregularly structured input files
- 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 diversion to discuss the tidyverse

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

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

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

```
table2
```

```
## # 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.

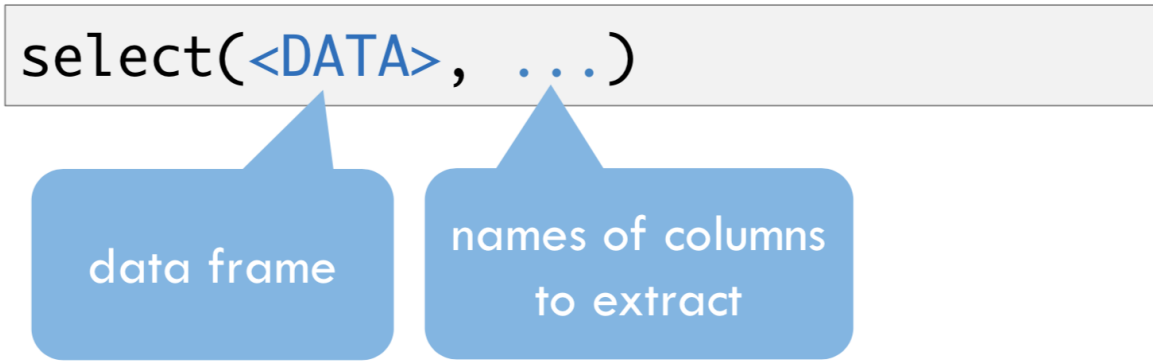
Manipulating data with dplyr

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

Carving your data set

Reducing a data set to a subset of columns and/or rows are common operations, particularly on the path to answering a specific set of questions about a data set.

If you need to go from a large number of columns (variables) to a smaller set, `select()` allows you to select spe-



cific 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)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    187200 obs. of  10 variables:
##  $ batch_name            : chr  "b802253" "b802253" "b802253" "b802253" ...
##  $ sample_name           : chr  "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  0 0 0 0 0 0 0 0 0 0 ...
##  $ response              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ concentration         : num  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: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ used_for_curve        : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ sample_passed         : logi  FALSE TRUE TRUE TRUE TRUE TRUE ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 10
##   .. ..$ batchName            : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ sampleName           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ compoundName         :List of 3
##   .. .. ..$ levels    : NULL
##   .. .. ..$ ordered   : logi FALSE
##   .. .. ..$ include_na: logi FALSE
##   .. .. ..- attr(*, "class")= chr  "collector_factor" "collector"
##   .. ..$ ionRatio             : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ response             : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ concentration        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
```

```
##   .. ..$ sampleType           :List of 3
##   .. .. ..$ levels    : NULL
##   .. .. ..$ ordered   : logi FALSE
##   .. .. ..$ include_na: logi FALSE
##   .. .. ..- attr(*, "class")= chr  "collector_factor" "collector"
##   .. ..$ expectedConcentration: list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ usedForCurve         : list()
##   .. .. ..- attr(*, "class")= chr  "collector_logical" "collector"
##   .. ..$ samplePassed         : list()
##   .. .. ..- attr(*, "class")= chr  "collector_logical" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##   ..- attr(*, "class")= chr "col_spec"
```

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

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

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

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

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:

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

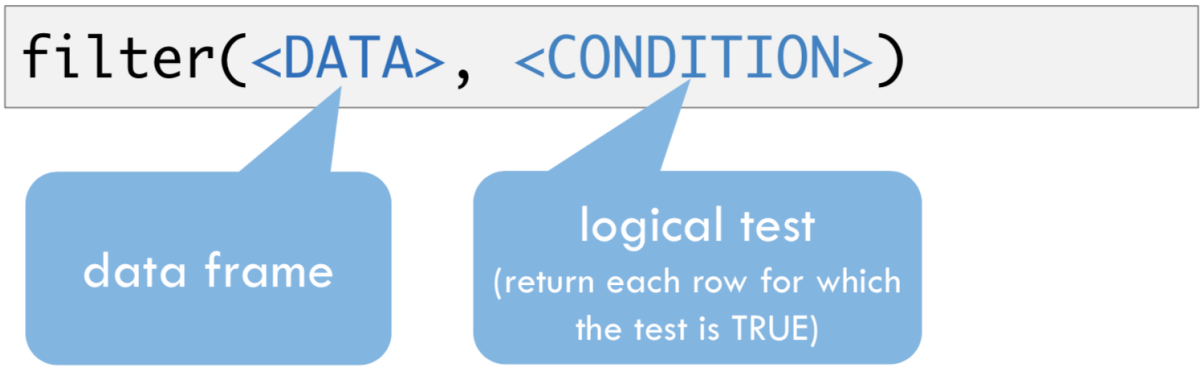


Figure 6: Syntax for `filter()`

```
## # 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 <int>, 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()
```

```
## # 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 <int>, 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()
```

```
## # 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 <int>, 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.

```
samples_jan_oxy_ir <- samples_jan %>%
  filter(sample_type == "unknown", concentration > 0, compound_name == "oxycodone") %>%
  select(batch_name, sample_name, compound_name, ion_ratio)
summary(samples_jan_oxy_ir)
```

```
##   batch_name        sample_name              compound_name
##  Length:12021       Length:12021       morphine     :    0
##  Class :character   Class :character   hydromorphone:    0
##  Mode  :character   Mode  :character   oxymorphone  :    0
##                                        codeine      :    0
##                                        hydrocodone  :    0
##                                        oxycodone    :12021
##    ion_ratio
##  Min.   :0.6279
##  1st Qu.:1.1718
##  Median :1.2351
##  Mean   :1.1999
##  3rd Qu.:1.2972
##  Max.   :1.6680
```

End Exercise

Expanding your data set

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

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

```
## # A tibble: 20 x 6
```

```
##    batch_name sample_name compound_name concentration expected_concen~
##    <chr>      <chr>       <fct>                 <dbl>            <int>
##  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)
```

```
## # A tibble: 20 x 6
##    batch_name sample_name compound_name concentration expected_concen~
##    <chr>      <chr>       <fct>                 <dbl>            <int>
##  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 manipulation is wrangling dates. The lubridate package offers a helpful toolset to quickly parse dates and times. The bread and butter parsing functions are named intuitively based on the order of year, month, date, and time elements. For example, `mdy("1/20/2018")` will convert the string into a date that R can use. There are other useful functions like `month()` and `wday()` that pull out a single element of the date to use for grouping operations, for example. Let's work with a different January data set that has batch data and parse the collection dates in a variety of ways:

```
batch_jan <- read_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
  )
summary(batch_jan_timestamps)
```

```
##   batch_name        instrument_name    compound_name
##  Length:3600        Length:3600        Length:3600
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
##  calibration_slope  calibration_intercept calibration_r2   batch_passed
```

```
##  Min.   :0.003172   Min.   :-9.510e-05    Min.   :0.9800   Mode:logical
##  1st Qu.:0.006794   1st Qu.:-2.160e-05    1st Qu.:0.9860   TRUE:3600
##  Median :0.007060   Median : 6.965e-08    Median :0.9902
##  Mean   :0.007107   Mean   : 7.202e-08    Mean   :0.9899
##  3rd Qu.:0.007351   3rd Qu.: 2.180e-05    3rd Qu.:0.9938
##  Max.   :0.009626   Max.   : 1.082e-04    Max.   :1.0000
##  reviewer_name      batch_collected_timestamp
##  Length:3600        Min.   :2017-01-06 20:08:00
##  Class :character   1st Qu.:2017-01-13 22:55:15
##  Mode  :character   Median :2017-01-21 11:00:30
##                     Mean   :2017-01-21 10:58:03
##                     3rd Qu.:2017-01-28 23:24:30
##                     Max.   :2017-02-05 01:54:00
##  review_start_timestamp        review_complete_timestamp
##  Min.   :2017-01-07 09:08:00   Min.   :2017-01-07 09:35:00
##  1st Qu.:2017-01-14 12:05:45   1st Qu.:2017-01-14 12:24:15
##  Median :2017-01-21 23:18:30   Median :2017-01-21 23:41:30
##  Mean   :2017-01-21 23:24:24   Mean   :2017-01-21 23:54:28
##  3rd Qu.:2017-01-29 11:17:15   3rd Qu.:2017-01-29 11:55:00
##  Max.   :2017-02-05 13:49:00   Max.   :2017-02-05 14:15:00
##  collect_datetime              collect_month   collect_day_of_week
##  Min.   :2017-01-06 20:08:00   Min.   :1.000   Min.   :1.00
##  1st Qu.:2017-01-13 22:55:15   1st Qu.:1.000   1st Qu.:2.00
##  Median :2017-01-21 11:00:30   Median :1.000   Median :4.00
##  Mean   :2017-01-21 10:58:03   Mean   :1.143   Mean   :4.09
##  3rd Qu.:2017-01-28 23:24:30   3rd Qu.:1.000   3rd Qu.:6.00
##  Max.   :2017-02-05 01:54:00   Max.   :2.000   Max.   :7.00
##   collect_week  collect_week_alt
##  Min.   :1.00   Min.   :2017-01-01 00:00:00
##  1st Qu.:2.00   1st Qu.:2017-01-08 00:00:00
##  Median :3.00   Median :2017-01-15 00:00:00
##  Mean   :3.39   Mean   :2017-01-17 17:31:12
##  3rd Qu.:4.00   3rd Qu.:2017-01-22 00:00:00
##  Max.   :6.00   Max.   :2017-02-05 00:00:00
```

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

Exercise 2

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

```
batch_jan_reviews <- batch_jan %>%
  mutate(review_start_timestamp = mdy_hm(review_start_timestamp),
         review_complete_timestamp = mdy_hm(review_complete_timestamp),
         review_duration = as.numeric(difftime(review_complete_timestamp, review_start_timestamp,
                                               units = "mins")))
```

```
## Warning: All formats failed to parse. No formats found.
```

```
## Warning: All formats failed to parse. No formats found.
```

```
# note: the output of difftime is a time interval, convert to numeric to avoid confusion downstream
reviews_jan_grouped <- batch_jan_reviews %>%
  distinct(batch_name, review_duration)
min(reviews_jan_grouped$review_duration)
```

```
## [1] NA
```

```
median(reviews_jan_grouped$review_duration)
```

```
## [1] NA
```

```
mean(reviews_jan_grouped$review_duration)
```

```
## [1] NA
```

```
max(reviews_jan_grouped$review_duration)
```

```
## [1] NA
```

End Exercise

Collapse (summarize) your data set

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

```
samples_jan %>%
  filter(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))
```

```
## # 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 %>%
  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)) %>%
  head()
```

```
## # 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_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))
```

```
## # A tibble: 234 x 7
## # Groups:   instrument_name, compound_name [?]
##    instrument_name compound_name collect_week median_cor mean_cor min_cor
##    <chr>           <chr>                <dbl>      <dbl>    <dbl>   <dbl>
##  1 bashful         codeine                  1      0.989    0.990   0.981
##  2 bashful         codeine                  2      0.991    0.990   0.981
##  3 bashful         codeine                  3      0.990    0.989   0.980
##  4 bashful         codeine                  4      0.992    0.991   0.983
##  5 bashful         codeine                  5      0.991    0.992   0.981
##  6 bashful         hydrocodone              1      0.989    0.990   0.985
##  7 bashful         hydrocodone              2      0.994    0.993   0.981
##  8 bashful         hydrocodone              3      0.990    0.989   0.982
##  9 bashful         hydrocodone              4      0.990    0.990   0.981
## 10 bashful         hydrocodone              5      0.987    0.988   0.980
## # ... with 224 more rows, and 1 more variable: max_cor <dbl>
```

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

Exercise 3

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

```
sample_stats_jan <- samples_jan %>%
  filter(sample_type == "unknown") %>%
  group_by(batch_name, compound_name) %>%
  summarize(min_conc = min(concentration),
            median_conc = median(concentration),
            mean_conc = mean(concentration),
            max_conc = max(concentration)
```

```
            )
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.
```

```
sample_means_jan <- sample_stats_jan %>%
  group_by(compound_name) %>%
  summarize(overall_mean = mean(mean_conc))
sample_means_jan
```

```
## # A tibble: 6 x 2
##   compound_name overall_mean
##   <fct>                <dbl>
## 1 morphine              129.
## 2 hydromorphone         131.
## 3 oxymorphone           132.
## 4 codeine               130.
## 5 hydrocodone           127.
## 6 oxycodone             130.
```

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
```

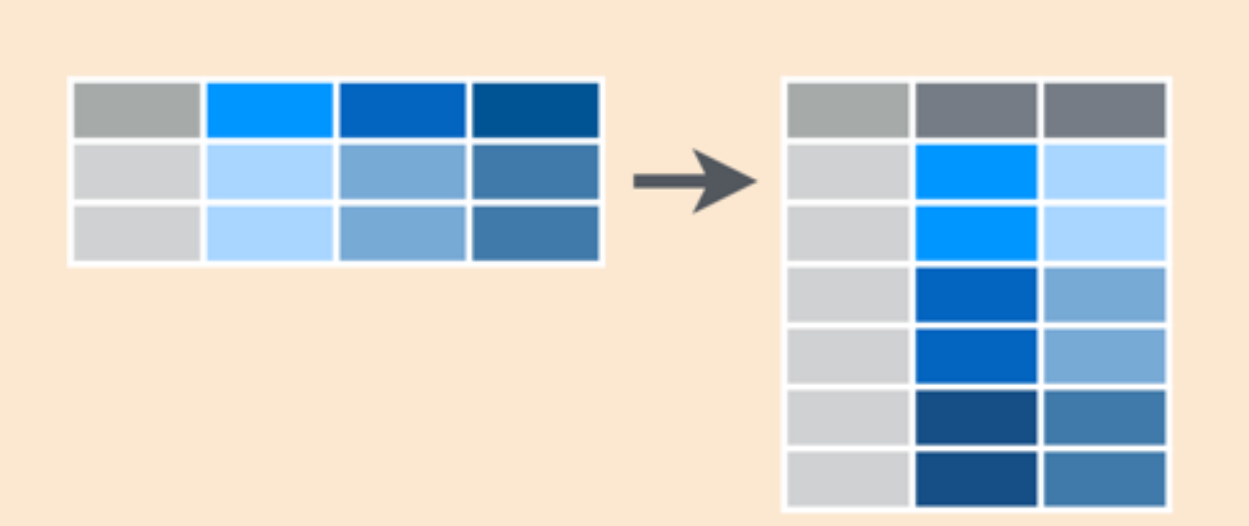


Figure 7: Gather operation

```
##   <chr>      <chr>       <chr>          <dbl>         <dbl>       <dbl>
## 1 b100302    s302001     blank            0             0           0
## 2 b100302    s302002     standard         0             0           0
## 3 b100302    s302003     standard        18.4          21.6        21.6
## 4 b100302    s302004     standard        49.4          38.8        46.4
## 5 b100302    s302005     standard        86.5          97.1       106.
## 6 b100302    s302006     standard       188.          189.        201.
## # ... with 3 more variables: codeine <dbl>, hydrocodone <dbl>,
## #   oxycodone <dbl>
```

This certainly isn't impossible to work with, but there are some challenges with not having separate observations on each row. Arguably the biggest challenges revolve around built-in tidyverse functionality, with grouping and plotting as the most prominent issues you might encounter. Luckily the tidyr package can help reshape your data.

The `gather()` function can take a list of columns as arguments, the key argument to name the variable you are gathering, and the value argument to name the new column with the values you extract from the old column.

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

```
## # 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    s302002     standard    morphine                0
## 3 b100302    s302003     standard    morphine               18.4
## 4 b100302    s302004     standard    morphine               49.4
## 5 b100302    s302005     standard    morphine               86.5
## 6 b100302    s302006     standard    morphine              188.
```

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

multiple columns into one column, whose variable also needs to be named (the value).

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

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

```
## # A tibble: 6 x 9
##   batch_name sample_name sample_type codeine hydrocodone hydromorphone
##   <chr>      <chr>       <chr>         <dbl>       <dbl>         <dbl>
## 1 b100302    s302001     blank           0           0             0
## 2 b100302    s302002     standard        0           0             0
## 3 b100302    s302003     standard       16.9        19.9          21.6
## 4 b100302    s302004     standard       49.4        56.0          38.8
## 5 b100302    s302005     standard      119.        114.           97.1
## 6 b100302    s302006     standard      197.        191.          189.
## # ... with 3 more variables: morphine <dbl>, oxycodone <dbl>,
## #   oxymorphone <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 so can be an independent exercise if you are intereted.

Exercise 4

The "2017-01-06-batch-messy.csv" file in the messy subdirectory of 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 obsevation (compound) - calibration slope, intercept, and R^2. However these appear on different lines. All 3 paramters need to appear on the same row.

After solving these problems, each row should contain a single compound with all three parameters appearing on that single row. Use `gather()` and `spread()` to tidy this data.

```
batch_jan_messy <- read_csv("data/messy/2017-01-06-batch-messy.csv")
```

```
## Parsed with column specification:
## cols(
##   batch_name = col_character(),
##   instrument_name = col_character(),
##   batch_passed = col_logical(),
##   reviewer_name = col_character(),
##   batch_collected_timestamp = col_datetime(format = ""),
##   review_start_timestamp = col_datetime(format = ""),
##   review_complete_timestamp = col_datetime(format = ""),
##   parameter = col_character(),
##   codeine = col_double(),
```

```
##   hydrocodone = col_double(),
##   hydromorphone = col_double(),
##   morphine = col_double(),
##   oxycodone = col_double(),
##   oxymorphone = col_double()
## )
```

```
head(batch_jan_messy, 10)
```

```
## # A tibble: 10 x 14
##    batch_name instrument_name batch_passed reviewer_name
##    <chr>      <chr>           <lgl>        <chr>
##  1 b802253    doc             TRUE         Xavier
##  2 b252474    sneezy          TRUE         Xavier
##  3 b856639    grumpy          TRUE         Brad
##  4 b678409    bashful         TRUE         Zachary
##  5 b829912    sneezy          TRUE         Xavier
##  6 b567436    doc             TRUE         Yolanda
##  7 b567885    sneezy          TRUE         Zachary
##  8 b220333    happy           TRUE         Zachary
##  9 b629177    happy           TRUE         Zachary
## 10 b731080    bashful         TRUE         Zachary
## # ... with 10 more variables: batch_collected_timestamp <dttm>,
## #   review_start_timestamp <dttm>, review_complete_timestamp <dttm>,
## #   parameter <chr>, codeine <dbl>, hydrocodone <dbl>,
## #   hydromorphone <dbl>, morphine <dbl>, oxycodone <dbl>,
## #   oxymorphone <dbl>
```

```
batch_jan_gathered <- batch_jan_messy %>%
  gather("codeine", "hydrocodone", "hydromorphone", "morphine", "oxycodone",
         "oxymorphone", key = "compound_name", value = "value")
batch_jan_tidy <- batch_jan_gathered %>%
  spread(key = "parameter", value = "value")
head(batch_jan_tidy, 10)
```

```
## # A tibble: 10 x 11
##    batch_name instrument_name batch_passed reviewer_name
##    <chr>      <chr>           <lgl>        <chr>
##  1 b802253    doc             TRUE         Xavier
##  2 b252474    sneezy          TRUE         Xavier
##  3 b856639    grumpy          TRUE         Brad
##  4 b678409    bashful         TRUE         Zachary
##  5 b829912    sneezy          TRUE         Xavier
##  6 b567436    doc             TRUE         Yolanda
##  7 b567885    sneezy          TRUE         Zachary
##  8 b220333    happy           TRUE         Zachary
##  9 b629177    happy           TRUE         Zachary
## 10 b731080    bashful         TRUE         Zachary
## # ... with 7 more variables: batch_collected_timestamp <dttm>,
## #   review_start_timestamp <dttm>, review_complete_timestamp <dttm>,
## #   compound_name <chr>, calibration_intercept <dbl>,
## #   calibration_r_2 <dbl>, calibration_slope <dbl>
```

Summary

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

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, 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()
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_integer(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
```

```
as_tibble(january_samples[187195:187200,])
```

```
## # A tibble: 6 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
## # ... with 4 more variables: sample_type <chr>,
## #   expected_concentration <int>, used_for_curve <lgl>,
## #   sample_passed <lgl>
```

```
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_integer(),
##   usedForCurve = col_logical(),
##   samplePassed = col_logical()
## )
```

```
as_tibble(february_samples[1:5,])
```

```
## # A tibble: 5 x 10
##   batch_name sample_name compound_name ion_ratio response concentration
##   <chr>      <chr>       <chr>             <dbl>    <dbl>         <dbl>
## 1 b593231    s231001     morphine              0        0             0
## 2 b593231    s231001     hydromorphone         0        0             0
## 3 b593231    s231001     oxymorphone           0        0             0
## 4 b593231    s231001     codeine               0        0             0
## 5 b593231    s231001     hydrocodone           0        0             0
## # ... with 4 more variables: sample_type <chr>,
```

```
## #   expected_concentration <int>, used_for_curve <lgl>,
## #   sample_passed <lgl>
```

```
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.

```
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 <int>, used_for_curve <lgl>,
## #   sample_passed <lgl>
```

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

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

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 remeber to the data are clean before you use.

Exercise 1

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

```
bind_rows(january_peaks,   february_samples)
```

```
## # A tibble: 936,000 x 14
##    batch_name sample_name compound_name chromatogram_na~ peak_area
##    <chr>      <chr>       <chr>         <chr>                <dbl>
##  1 b802253    s253001     morphine      quant                    0
##  2 b802253    s253001     morphine      qual                     0
##  3 b802253    s253001     hydromorphone quant                    0
##  4 b802253    s253001     hydromorphone qual                     0
##  5 b802253    s253001     oxymorphone   quant                    0
##  6 b802253    s253001     oxymorphone   qual                     0
##  7 b802253    s253001     codeine       quant                    0
##  8 b802253    s253001     codeine       qual                     0
##  9 b802253    s253001     hydrocodone   quant                    0
## 10 b802253    s253001     hydrocodone   qual                     0
## # ... with 935,990 more rows, and 9 more variables: peak_quality <dbl>,
```

```
## #   manually_modified <lgl>, ion_ratio <dbl>, response <dbl>,
## #   concentration <dbl>, sample_type <chr>, expected_concentration <int>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

```
bind_rows(january_batches, february_samples)
```

```
## # A tibble: 190,800 x 19
##    batch_name instrument_name compound_name calibration_slo~
##    <chr>      <chr>           <chr>                    <dbl>
##  1 b802253    doc             morphine               0.00775
##  2 b802253    doc             hydromorphone          0.00768
##  3 b802253    doc             oxymorphone            0.00798
##  4 b802253    doc             codeine                0.00819
##  5 b802253    doc             hydrocodone            0.00656
##  6 b802253    doc             oxycodone              0.00902
##  7 b252474    sneezy          morphine               0.00757
##  8 b252474    sneezy          hydromorphone          0.00663
##  9 b252474    sneezy          oxymorphone            0.00713
## 10 b252474    sneezy          codeine                0.00696
## # ... with 190,790 more rows, and 15 more variables:
## #   calibration_intercept <dbl>, calibration_r2 <dbl>, batch_passed <lgl>,
## #   reviewer_name <chr>, batch_collected_timestamp <dttm>,
## #   review_start_timestamp <dttm>, review_complete_timestamp <dttm>,
## #   sample_name <chr>, ion_ratio <dbl>, response <dbl>,
## #   concentration <dbl>, sample_type <chr>, expected_concentration <int>,
## #   used_for_curve <lgl>, sample_passed <lgl>
```

```
bind_rows(january_samples, february_samples) # the only January data frame with matching columns
```

```
## # A tibble: 374,400 x 10
##    batch_name sample_name compound_name ion_ratio response concentration
##    <chr>      <chr>       <chr>             <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 374,390 more rows, and 4 more variables: sample_type <chr>,
## #   expected_concentration <int>, used_for_curve <lgl>,
## #   sample_passed <lgl>
```

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.

```
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. As it turns out for this data set, no single column operates as a key. We can build a key by combinging 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 call a *primary key*, which is the unique identifier for each observation in a specific tibble. A *foreign key* is the same thing, only it uniqely 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.

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

Set operations `union`, `intersect`, and `setdiff`

Relational databases operate using operations from set theory. Joining two datasets by matching a common identifier is the most common application, however *dplyr* supports many more set operations which are useful in wrangling complex data. Unions, intersections, and the differences between dataset are often needed to get to the data you want.

These three commands will return a vector which is the unduplicated combination of the two input vectors. `union(A,B)` includes all the values found in both A and B. `intersect(A,B)` returns only those values found

in both A and B. `setdiff(A,B)` is order dependent, and returns the values of the first vector which are not also in the second vector.

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

```
##  [1]  1  2  3  4  5  6  7  8  9 10 12 14 16 18 20
```

```
intersect(A, B)
```

```
## [1]  2  4  6  8 10
```

```
setdiff(A, B)
```

```
## [1] 1 3 5 7 9
```

```
setdiff(B, A)
```

```
## [1] 12 14 16 18 20
```

These commands are good for checking matches between two vectors, and we can use them to rebuild the `select_peaks$keyB` foreign key without the risk of incorrect naming. First, let's reset `select_peaks`.

```
select_peaks <- january_peaks %>%
  filter(batch_name %in% c("b802253","b252474")) %>%
  unite(keyP, sample_name, compound_name, chromatogram_name, sep=":", remove=FALSE)
```

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

```
all_names <- unique(select_peaks$compound_name)
select_peaks$analyte <- NA
for (name in all_names[1:6]) {
  compoundPairIdx <- grep(name, all_names)
  theCompound <- intersect(all_names[compoundPairIdx], name)
  theInternalStandard <- setdiff(all_names[compoundPairIdx], name)
  select_peaks$analyte[select_peaks$compound_name == theInternalStandard] <- theCompound
  select_peaks$analyte[select_peaks$compound_name == theCompound] <- theCompound
}
```

Mutating join to add columns

Mutating joins operate in much the same way as the set operations, but on data frames instead of vectors, and with one critical difference: repeated values are retained. We took advantage of this earlier when using the left_join command, so that the `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("batch_name", "compound_name")
```

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.

At first it may be confusing that some joins result in making duplicates of rows from one tibble. However, it is extremely handy when you are using one dataset to label another.

```
goodDuplication <- inner_join(
  x = select_samples[, c(1:4, 7)],
  y = select_batches[, c(1:6)],
  by = c("batch_name", "compound_name")
)
```

Filtering join to check the overlap

We created the `byBatch$keyB` explicitly by looking directly at the data. The compound naming scheme in `byPeak` is more complicated in that the internal standard isn't identified in `byBatch` or `bySample`, so we fixed this using a new column *analyte*.

Sometimes it is useful to look for these complications using the semi_join and anti_join commands. The *semi_join(A,B)* returns all rows of A where there is a match from B, but keeps only the columns of A, and does not duplicate a row if there are multiple matches. The *anti_join(A,B)* is the inverse, returning all rows from A where there is no match from B. We still want to create the 'analyte' column for clarity, so one approach would be:

```
select_batches <- january_batches %>% # reset, no keyB
  filter(batch_name %in% c("b802253","b252474"))
select_peaks <- january_peaks %>% # reset, no keyB or keyP
  filter(batch_name %in% c("b802253","b252474")) %>%
  mutate(analyte = compound_name)

unique(select_peaks$analyte) # notice the similar naming scheme
```

```
##  [1] "morphine"         "hydromorphone"    "oxymorphone"
##  [4] "codeine"          "hydrocodone"      "oxycodone"
##  [7] "morphine-13C"     "hydromorphone-d3" "oxymorphone-d3"
## [10] "codeine-d6"       "hydrocodone-d3"   "oxycodone-d3"
```

```
select_peaks <- select_peaks %>%
  mutate(analyte = sub("-.*$", "", analyte)) # use of substitution and regex
select_peaks$analyte <- sub("-.*$", "", select_peaks$analyte) # notice how we used that similarity?

noMatch <- anti_join(select_peaks, select_batches)
```

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

```
noMatch <- anti_join(select_peaks, select_batches,
                     by=c("batch_name","analyte"="compound_name"))

justMatch <- semi_join(select_peaks, select_batches,
                       by=c("batch_name","analyte"="compound_name"))
```

Exercise 2:

Join the batch and peak data. Start from the reset tibbles (`select_batches` and `select_peaks`) built in the prior code chunk, so the `keyB` and `keyP` variables are not present.

```
select_peaks$analyte <- sub("-.*$", "", select_peaks$analyte) # ensure the variable has been modified
exercise_two <- left_join(select_batches, select_peaks,
                          by=c("batch_name","compound_name" = "analyte"))
```

End Exercise

Summary

- `rbind` and `cbind` add rows (or columns) to an existing data frame
- `union`, `intersect`, and `setdiff` return a combination of two vectors
- Relational data merges two data frames on the common columns, called keys
  - A primary key is a unique identifier for every row in a data frame (the presence of `keyB` in `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
- `semi_join` and `anti_join` are filtering joins which check for overlap

Stronger visualizations with ggplot2

Plotting Data With Default Graphics

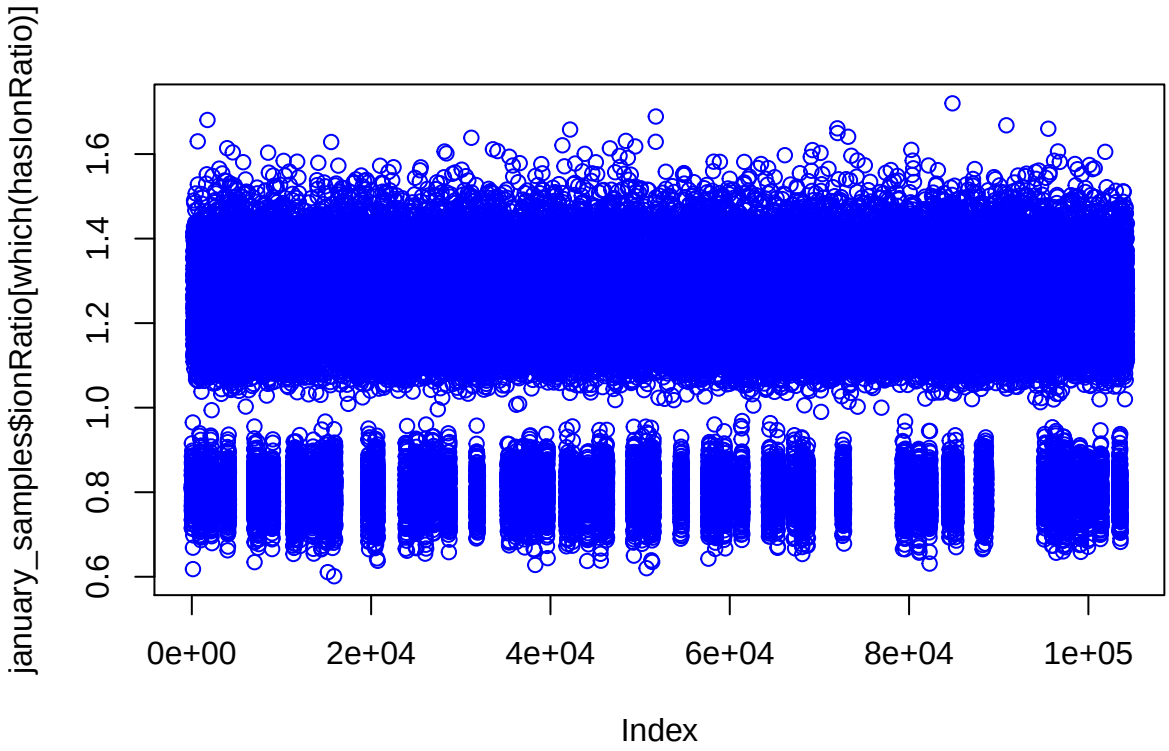
Default R comes with several basic plotting commands – `plot` to draw an X,Y graph, `points` to add X,Y points to the current graph, `barplot` to draw vertical or horizontal bars, `boxplot` to draw box-and-whisker plots, `hist` to build and draw a histogram, and many other plot types or plot-specific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

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

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

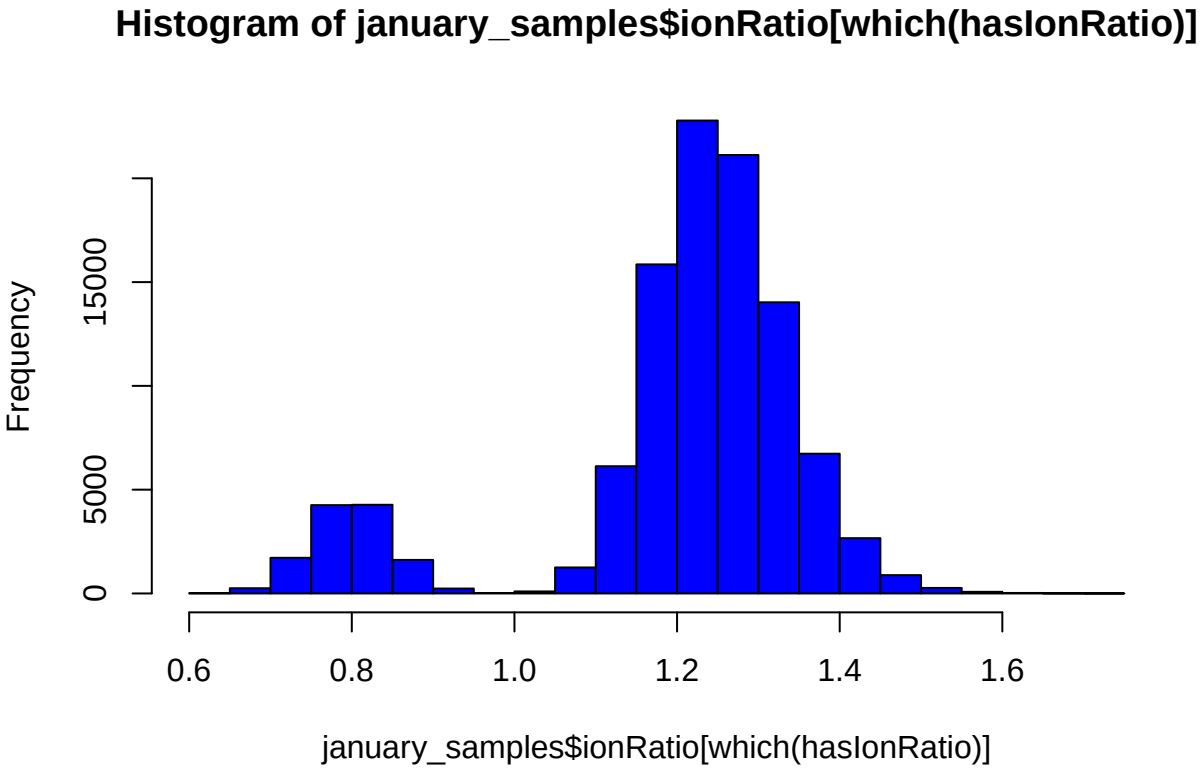
```
## Parsed with column specification:
## cols(
##   batchName = col_character(),
##   sampleName = col_character(),
##   compoundName = col_character(),
##   ionRatio = col_double(),
##   response = col_double(),
##   concentration = col_double(),
##   sampleType = col_character(),
##   expectedConcentration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

```
january_samples$idx <- c(1:nrow(january_samples))
hasIonRatio <- january_samples$ionRatio > 0
plot(january_samples$ionRatio[which(hasIonRatio)],col='blue')
```



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

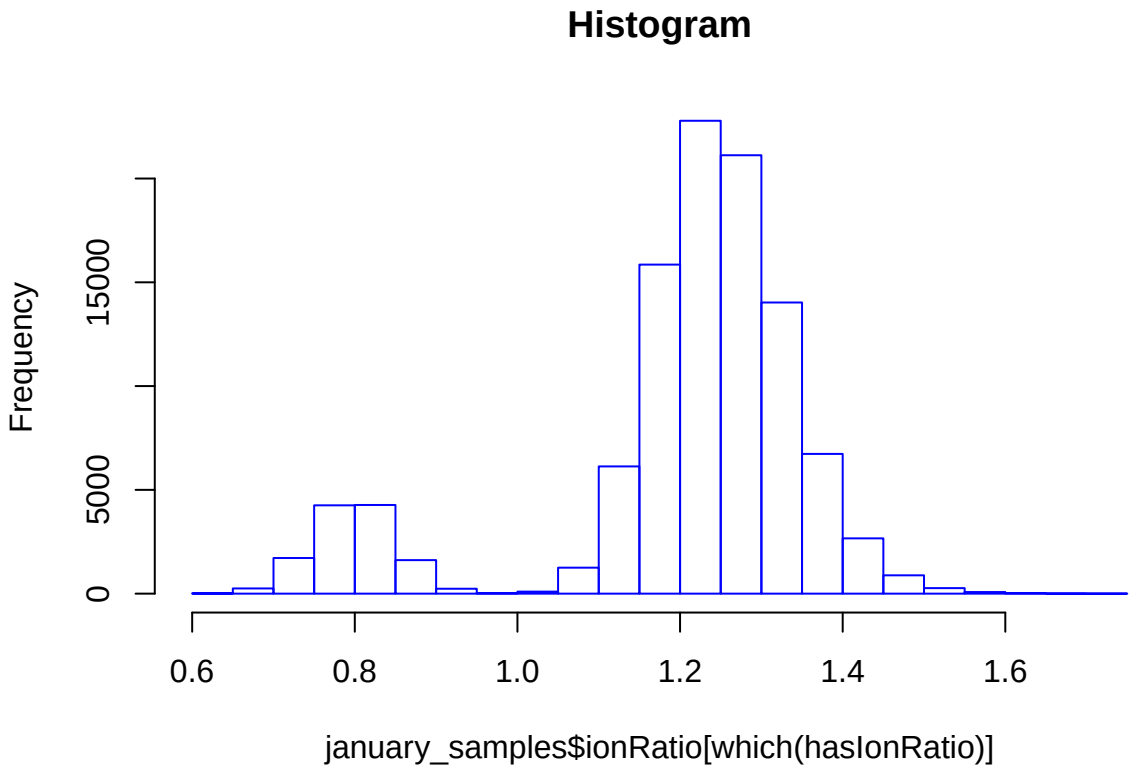
```
hist(january_samples$ionRatio[which(hasIonRatio)],col='blue')
```



In order to plot the histogram with blue outline, to match the blue open circles of the first plot, you need to

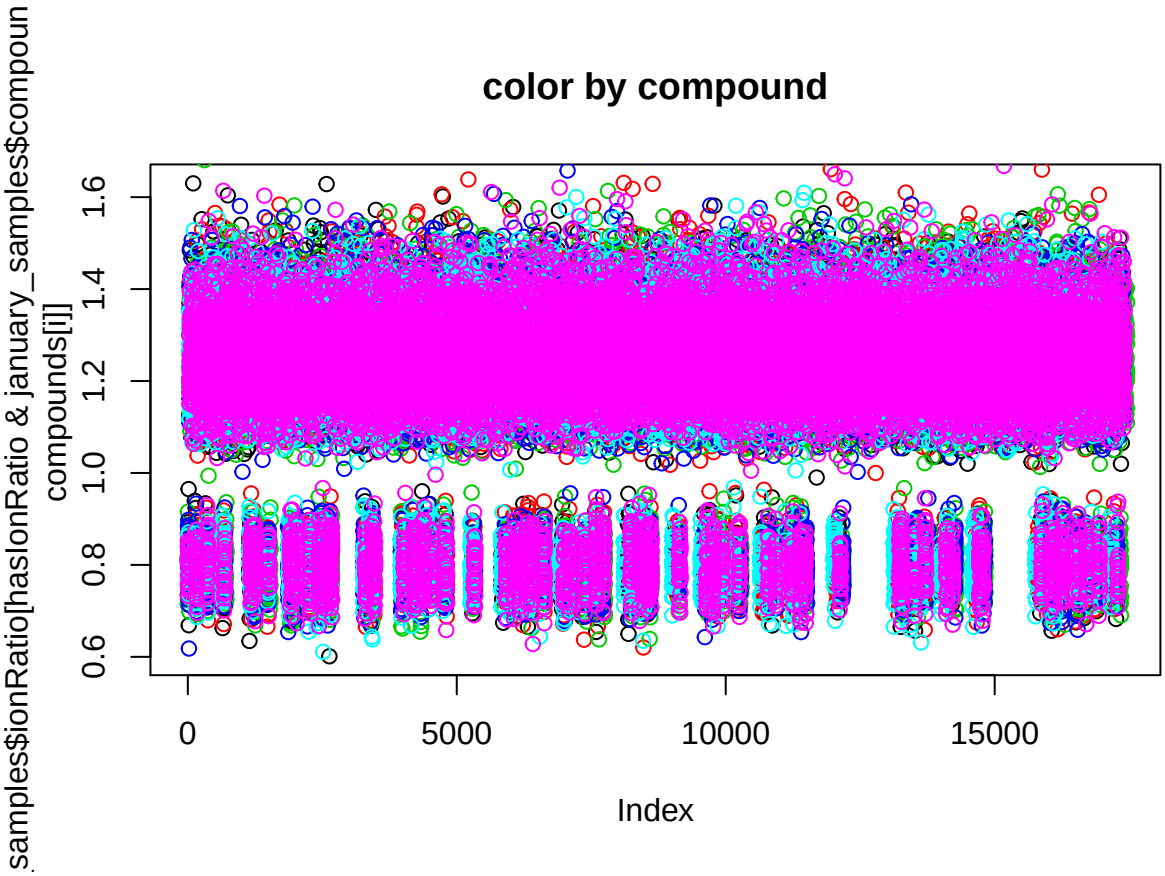
specify a different variable.

```
hist(january_samples$ionRatio[which(hasIonRatio)],border='blue',main='Histogram')
```



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

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



Plotting Data With *ggplot2*

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

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

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

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

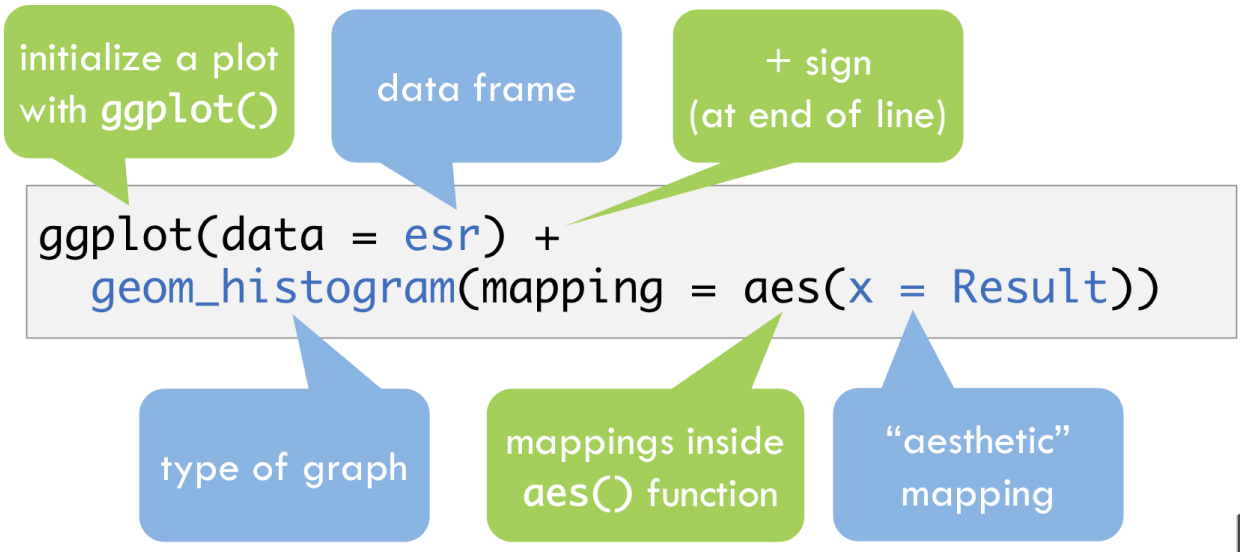
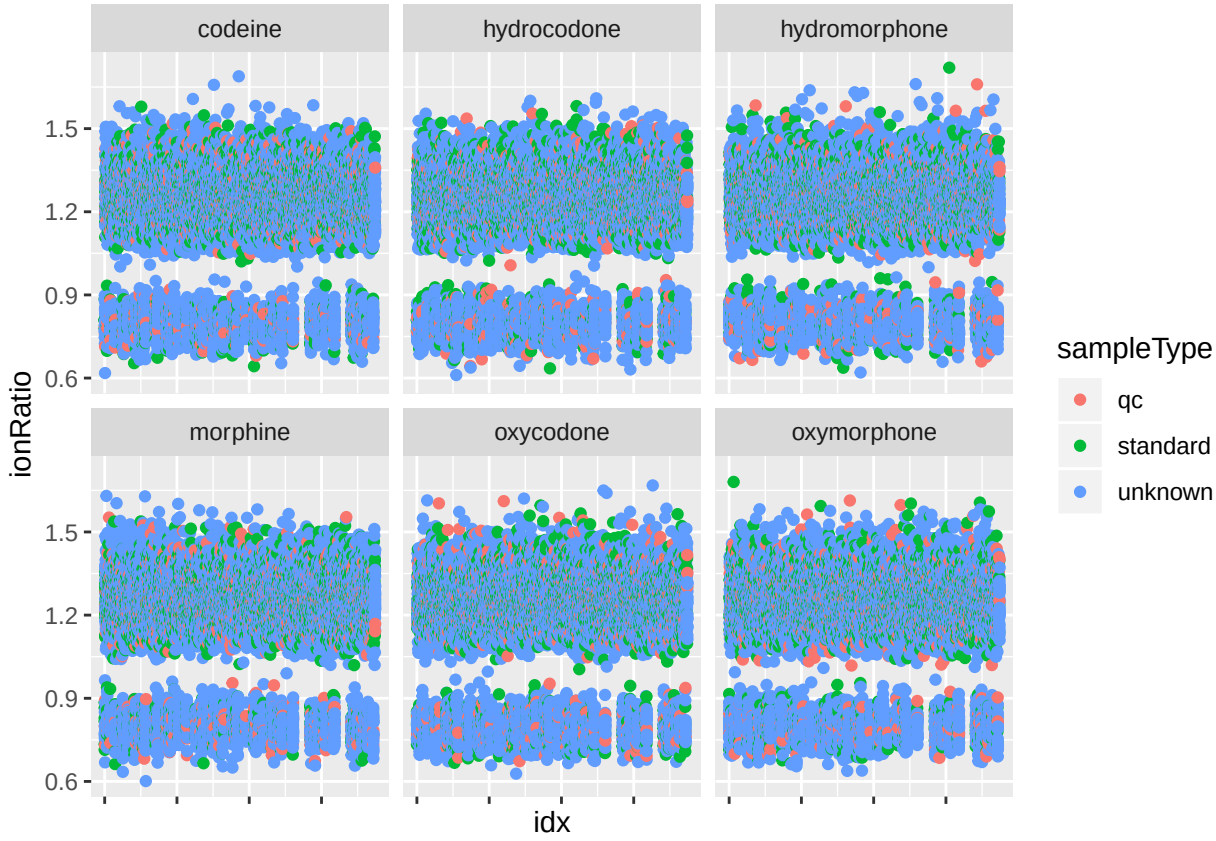
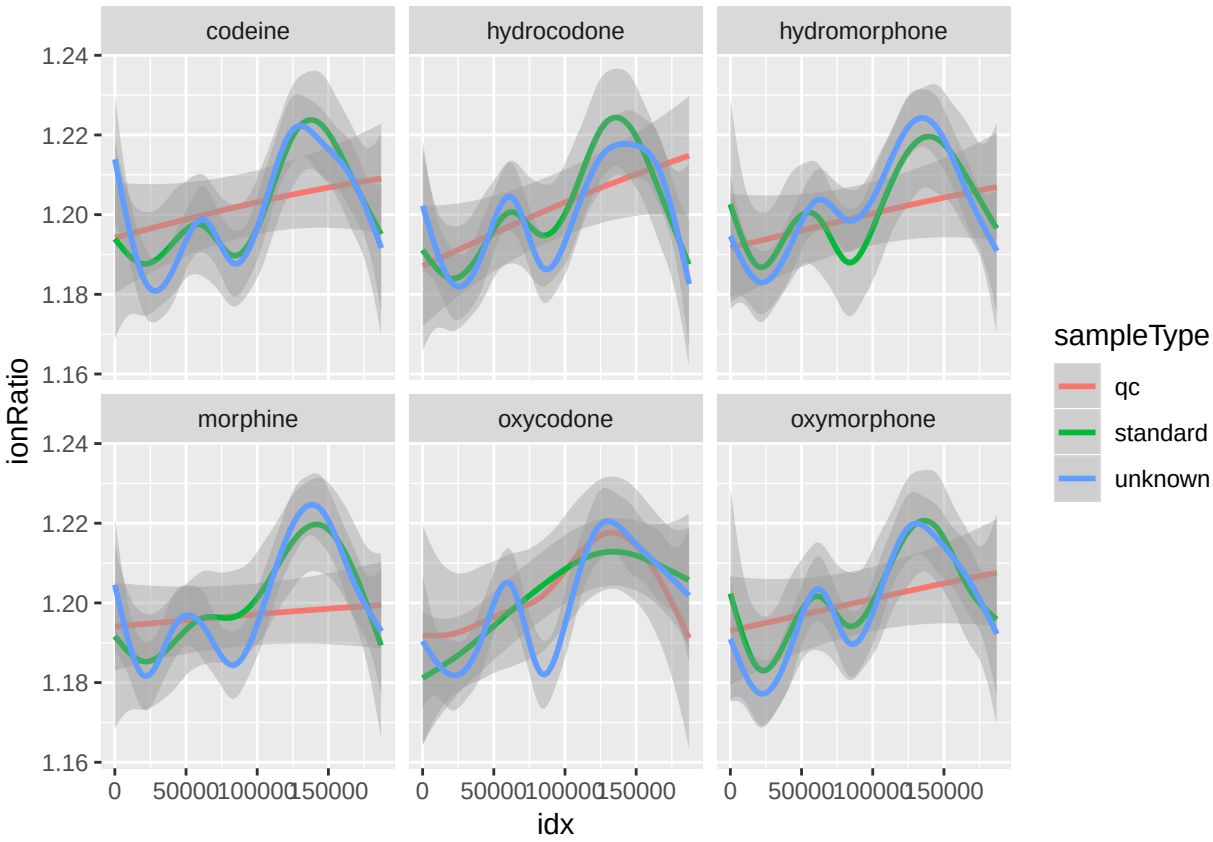


Figure 8: Syntax for ggplot




```
g + geom_smooth() + facet_wrap(~compoundName)
```

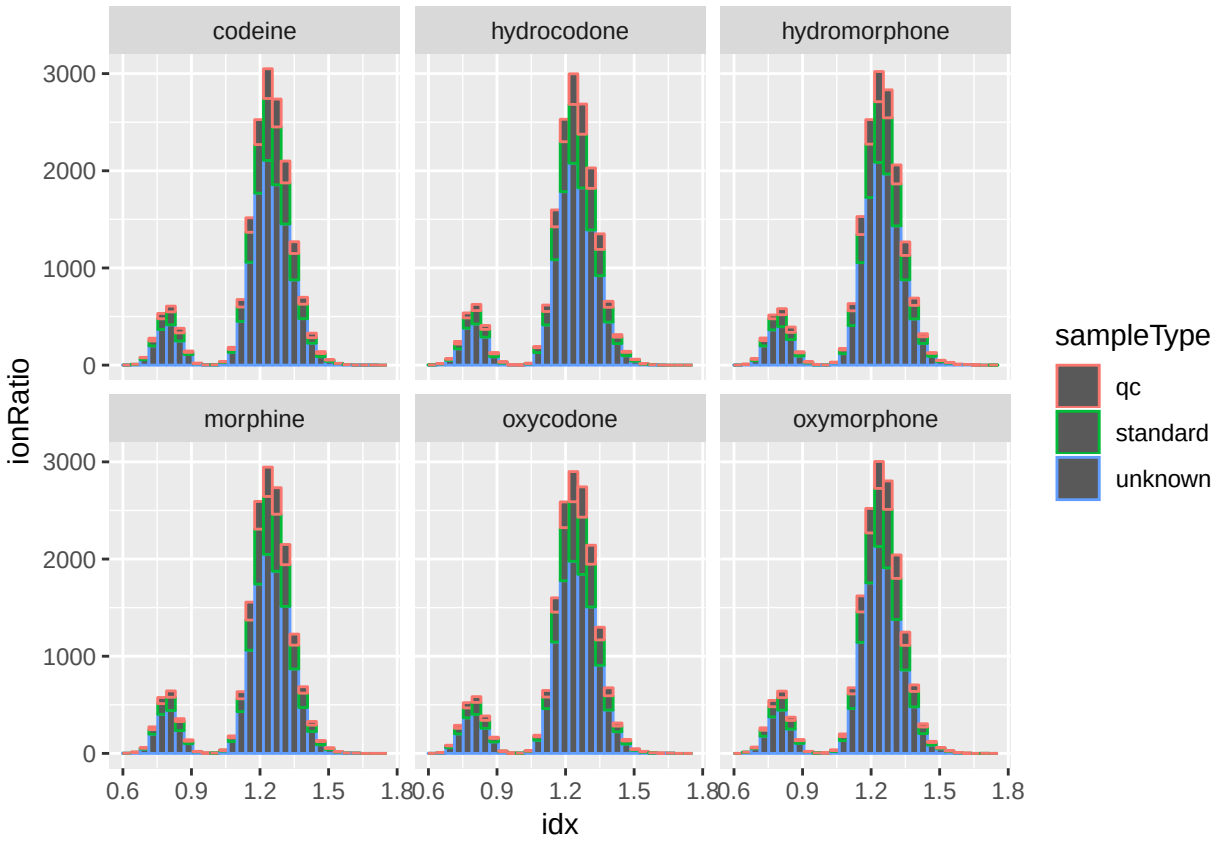
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



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

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

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

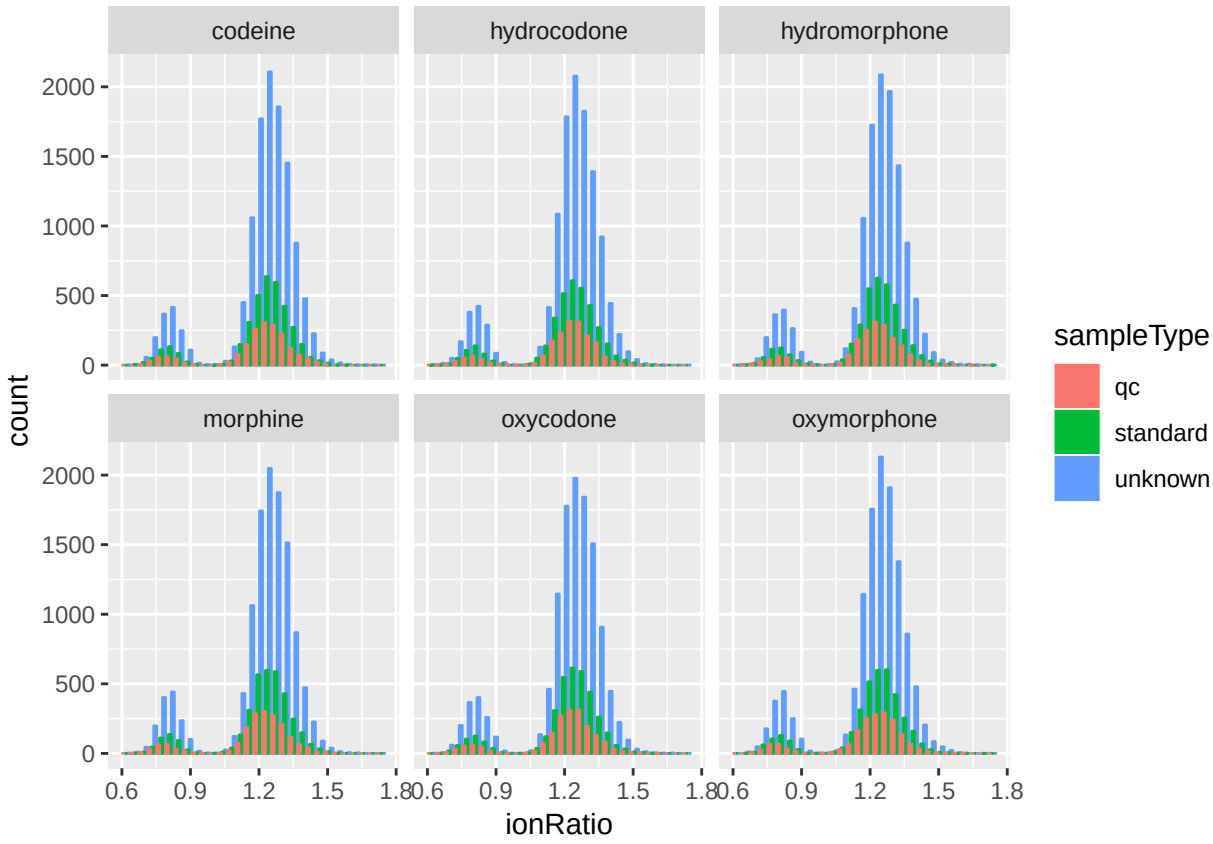


We could easily spend the whole class session on this package, but the above plots showcase the basic syntax. The cheatsheet downloadable from the link at the end of this lesson provides additional examples of what can be done.

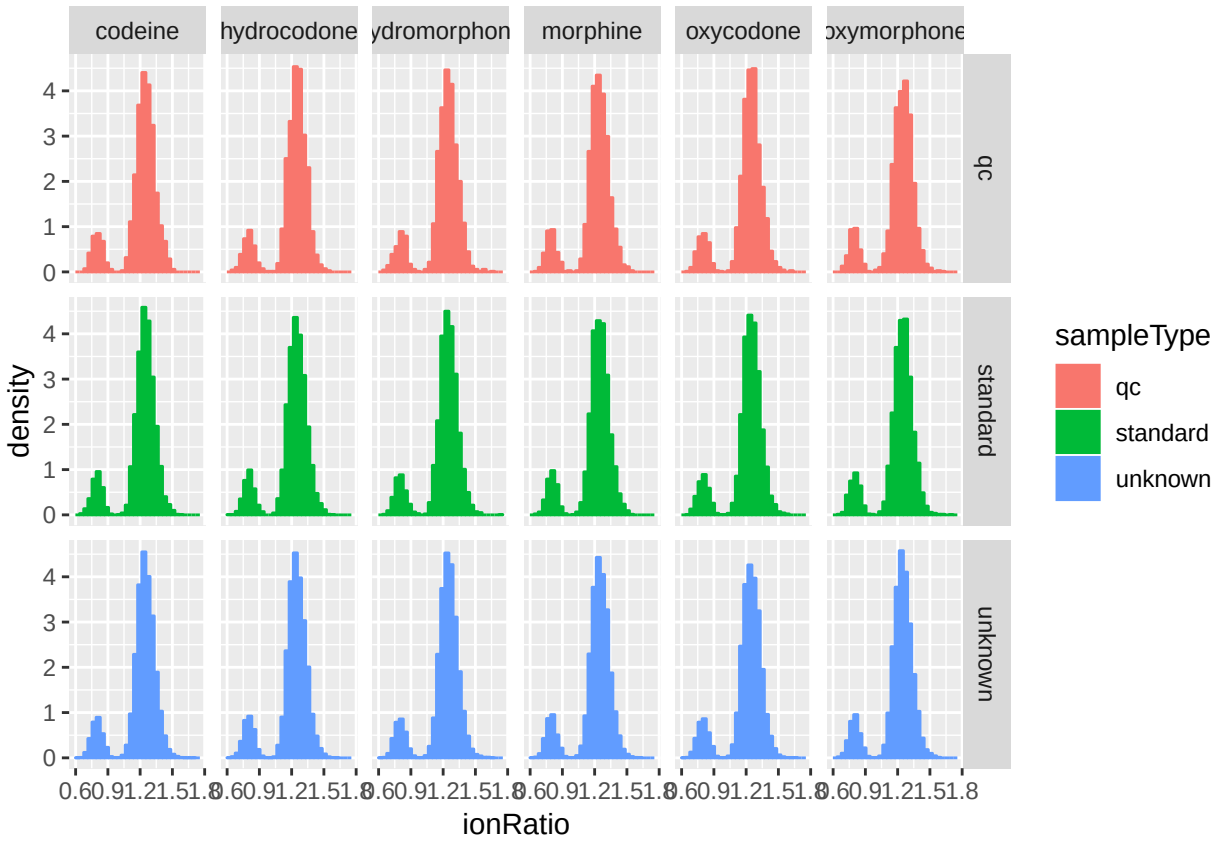
Exercise 1

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

```
g <- january_samples %>%
  filter(ionRatio > 0) %>%
  ggplot(aes(x = ionRatio, colour = sampleType, fill = sampleType))
g + geom_histogram(position='dodge', bins=30) + facet_wrap(~compoundName)
```




```
g + geom_histogram(aes(y=..density..), bins=30) + facet_grid(sampleType~compoundName)
```



End Exercise

Summary

- ggplot2 cheatsheat
- download a PDF comparison of both packages

Method comparison & fitting models to the data

Overview of data

A common activity in the clinical laboratory and in translation of an assay into the clinic is performing a method comparison of a new method with a previous method. In this lesson we will explore data from two methods for a human chorionic gonadotropin (hCG), which in this case is part of a panel to assess pregnant women for a fetus with trisomy 21 (Down's syndrome) or a neural tube defect. This data has values for a method A and a method B in a simple spreadsheet, with sheets corresponding to each of the assays in the panel (alpha fetoprotein and unconjugated estriol being the other two tests).

First we start by loading in hCG data from the "MS HCG" sheet.

```
hcg <- read_excel(path="data/method_validation_data.xlsx",
                  sheet="MS HCG")
glimpse(hcg)
```

```
## Observations: 146
## Variables: 3
```

```
## $ specimen <chr> "A1", "A2", "A3", "A4", "A5", "A6", "A7", "A8", "A9",...
## $ method_a <dbl> 27818, 6918, 9055, 23375, 27169, 59253, 18425, 5410, ...
## $ method_b <dbl> 40695, 9633, 12880, 39303, 42995, 87097, 24797, 8710,...
```

Let's use pipes to summarize and calculate a few statistics:

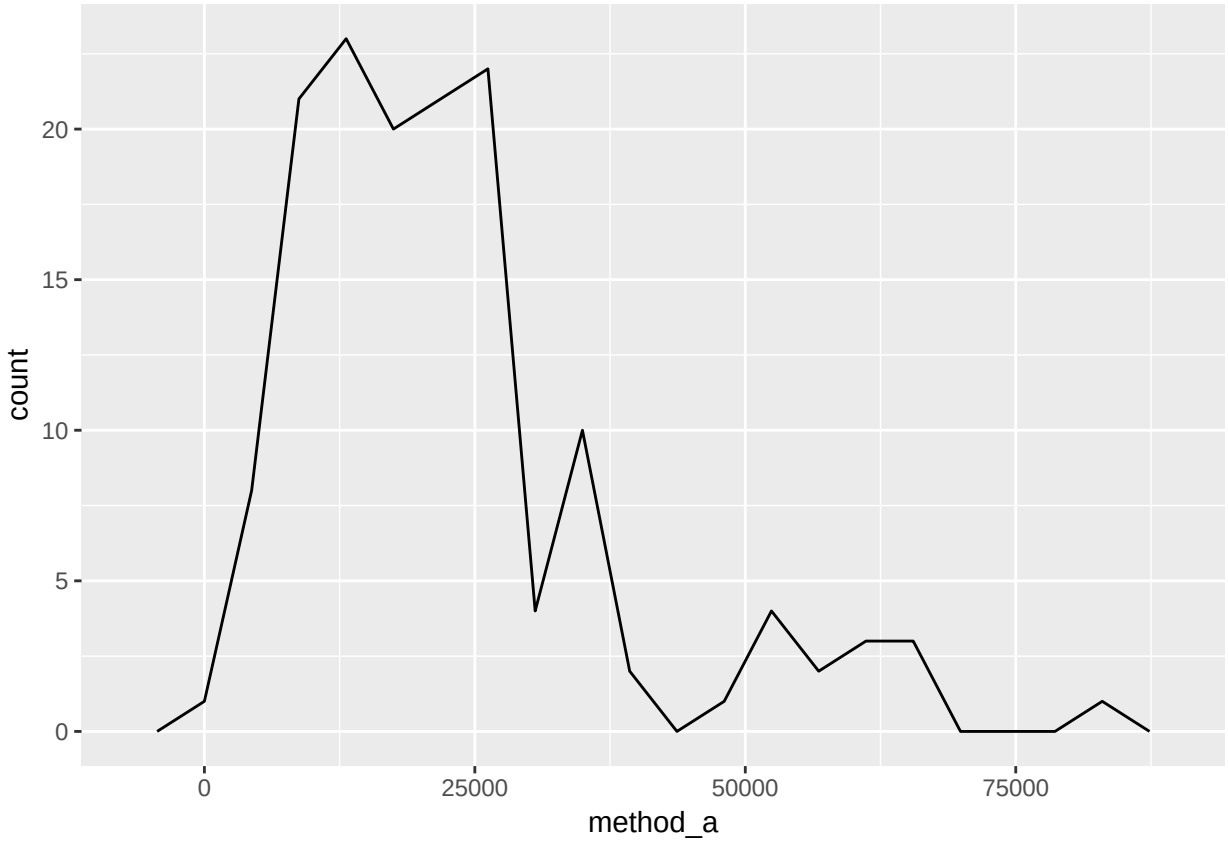
```
hcg %>%
  summarize(method_a_mean = mean(method_a),
            method_a_sd = sd(method_a),
            method_b_mean = mean(method_b),
            method_b_sd = sd(method_b))
```

```
## # A tibble: 1 x 4
##   method_a_mean method_a_sd method_b_mean method_b_sd
##           <dbl>       <dbl>         <dbl>       <dbl>
## 1        22677.      14966.        35428.      22689.
```

Overlapping histograms

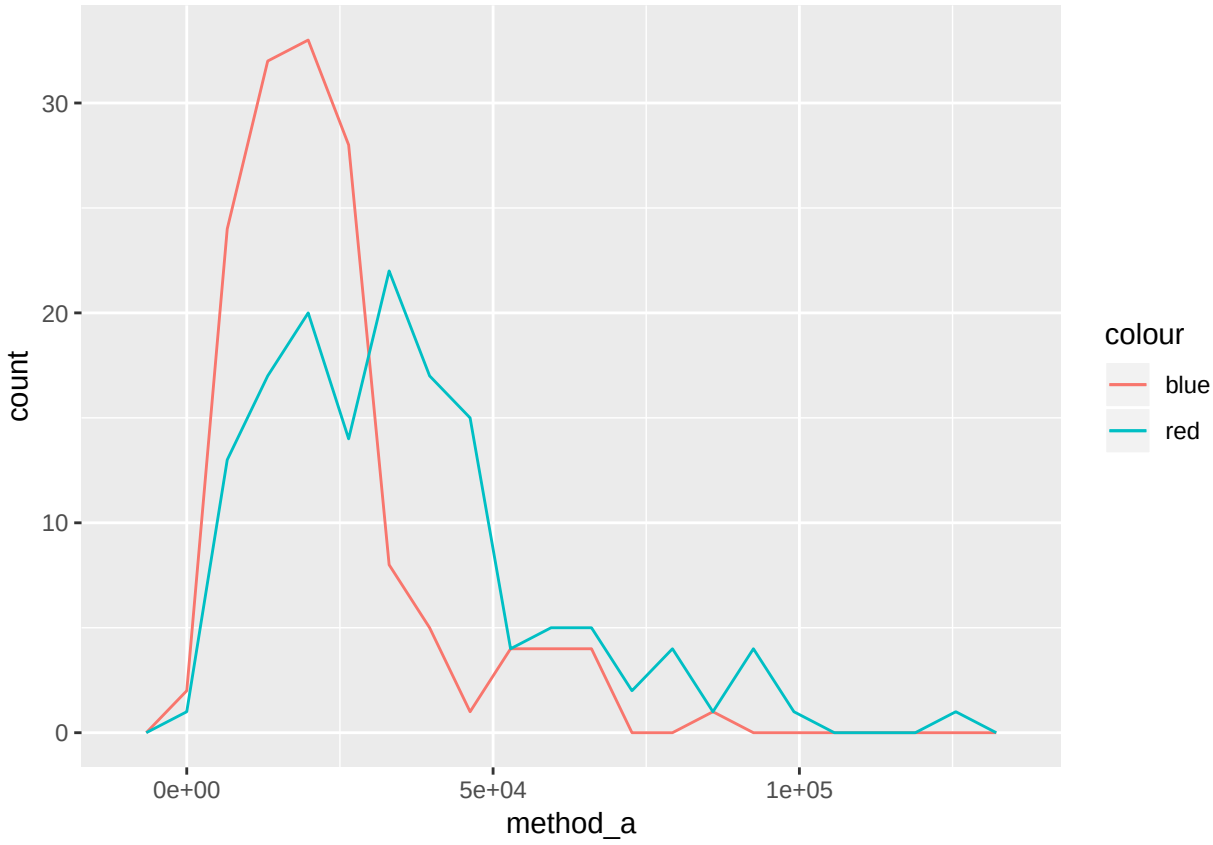
What if we want to plot the distribution of both `method_a` and `method_b` in the same plot? We've used `geom_histogram` previously. Let's try the related ggplot function `geom_freqpoly`, starting with a single method.

```
ggplot(data = hcg) +
  geom_freqpoly(bins=20,
                aes(x=method_a))
```



Now, let's add a second method and mark the two using `geom_freqpoly`.

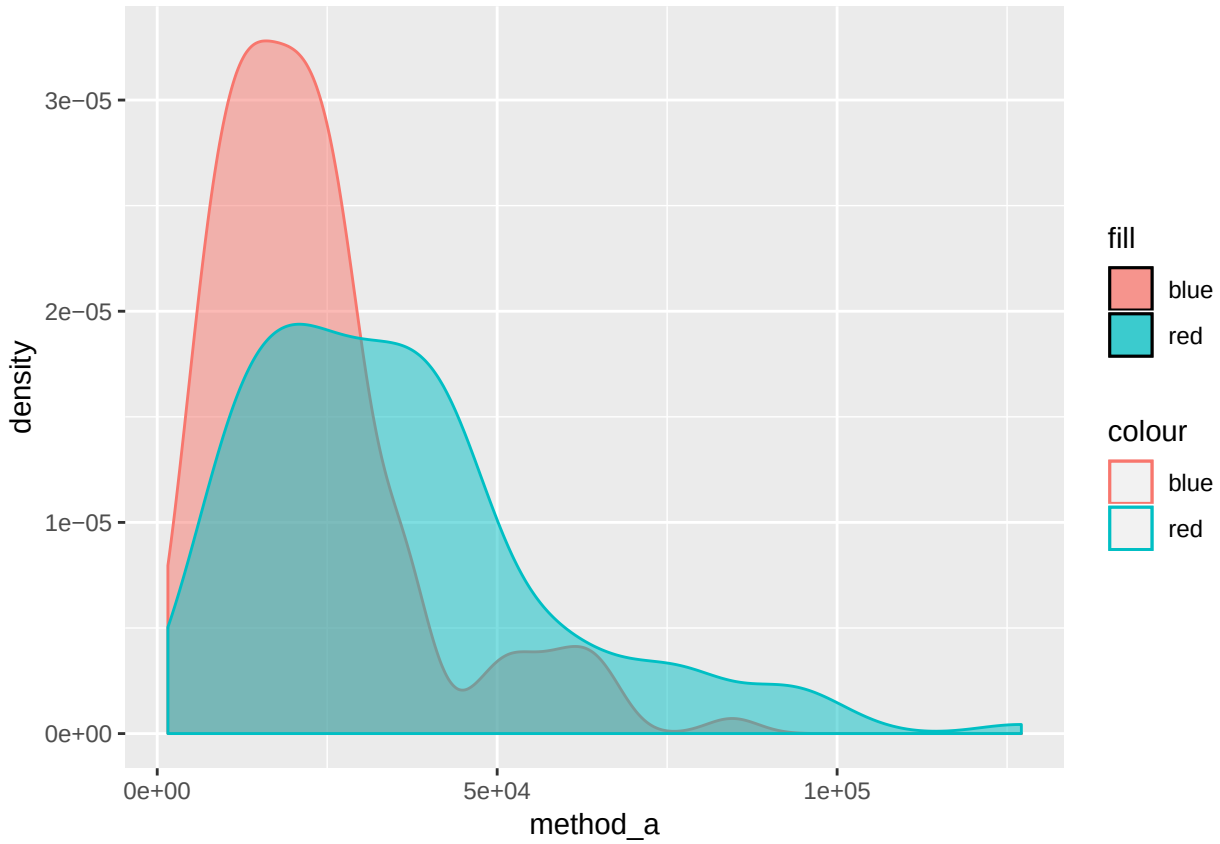
```
ggplot(data = hcg) +
  geom_freqpoly(bins=20, aes(x=method_a, color="blue")) +
  geom_freqpoly(bins=20, aes(x=method_b, color="red"))
```



Exercise 1:

Make a similar display of method a and method b distributions using the `geom_density` function. Set the `fill` and `color` functions to distinguish between the two methods. Test using the `alpha` parameter to increase the shape translucency

```
ggplot(data = hcg) +
  geom_density(aes(x=method_a, fill="blue", color="blue"), alpha = 0.5) +
  geom_density(aes(x=method_b, fill="red", color="red"), alpha = 0.5)
```



End exercise

Reorganizing a tibble to facilitate plotting

These plots require making separate calls to the `geom_*` functions for each method. This works, but makes legends needing manual cleanup and does not scale well.

Instead, another way is to wrangle the data for a plot is to create a "long dataframe" using the `gather` function. Recall that this "unpacks" multiple columns into just two columns, where the **first column** is the `key` and the **second column** is the `value`: assets/data_gather.png

```
long_hcg <- hcg %>%
  gather(key="method", value="value",
         -specimen)
head(long_hcg)
```

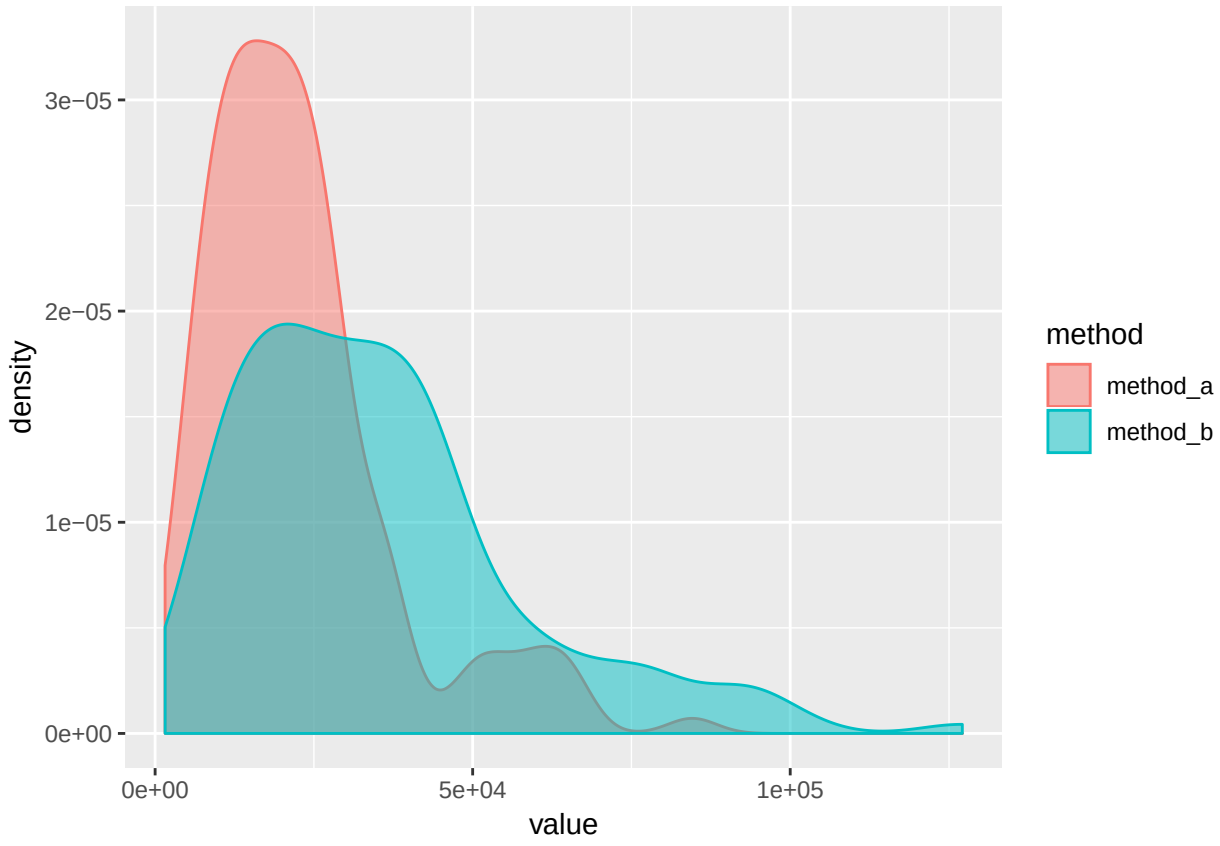
```
## # A tibble: 6 x 3
##   specimen method   value
##   <chr>    <chr>    <dbl>
## 1 A1       method_a 27818
## 2 A2       method_a  6918
## 3 A3       method_a  9055
## 4 A4       method_a 23375
## 5 A5       method_a 27169
## 6 A6       method_a 59253
```

Take a moment to compare the `hcg` and `long_hcg` objects. How are they different? Note that when we have a "long" datafarme, every row is a named observation (also known as a key-value pair). For reference, the

reverse transformation (from "long" to "wide") is done with the `spread` function.

Once in a long-form format, we can tell ggplot to use both the `value` variable for our x-axis and the `method` variable to set the coloring (fill and color) and the resulting legend.

```
ggplot(long_hcg) +
  geom_density(aes(x=value, fill=method, color=method),
               alpha=0.5)
```



Method comparison (t-tests, and more)

Using a statistical test

R is a statistical programming language, so simple statistical testing is straightforward:

```
# Note we are using the paired=TRUE variant of the t.test, since we have paired measurements.
t.test(hcg$method_a, hcg$method_b,
       paired=TRUE)
```

```
##
##  Paired t-test
##
## data:  hcg$method_a and hcg$method_b
## t = -18.143, df = 145, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -14139.80 -11361.65
## sample estimates:
```

```
## mean of the differences
##               -12750.73
```

For more information on the `t.test` function, (follow this link)[https://www.statmethods.net/stats/ttest.html].

Exercise 2: Evaluate parametric comparability of method means after log-transformation

```
# Note we are using the paired=TRUE variant of the t.test, since we have paired measurements.
t.test(log(hcg$method_a), log(hcg$method_b),
       paired=TRUE)
```

```
##
##  Paired t-test
##
## data:  log(hcg$method_a) and log(hcg$method_b)
## t = -54.713, df = 145, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.4561474 -0.4243406
## sample estimates:
## mean of the differences
##               -0.440244
```

End exercise

Using the RIGHT statistical test

Is `t.test` the right function? Consider the histograms above and our previous work with log normalizing the values.

Populations	Parametric	Non-parametric
Two populations	t-test	Mann-Whitney U
Many populations	ANOVA	Kruskal Wallis / one-way anova
Populations across several treatments/times	repeated measures ANOVA	Friedman test

Exercise 3: Using the table above, select the *right* test for comparing `method_a` and `method_b`. Look up the function call using google, R documentation or any other source. Write out the function and calculate a p-value below

```
wilcox.test(hcg$method_a, hcg$method_b, paired=TRUE)
```

```
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  hcg$method_a and hcg$method_b
## V = 0, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
```

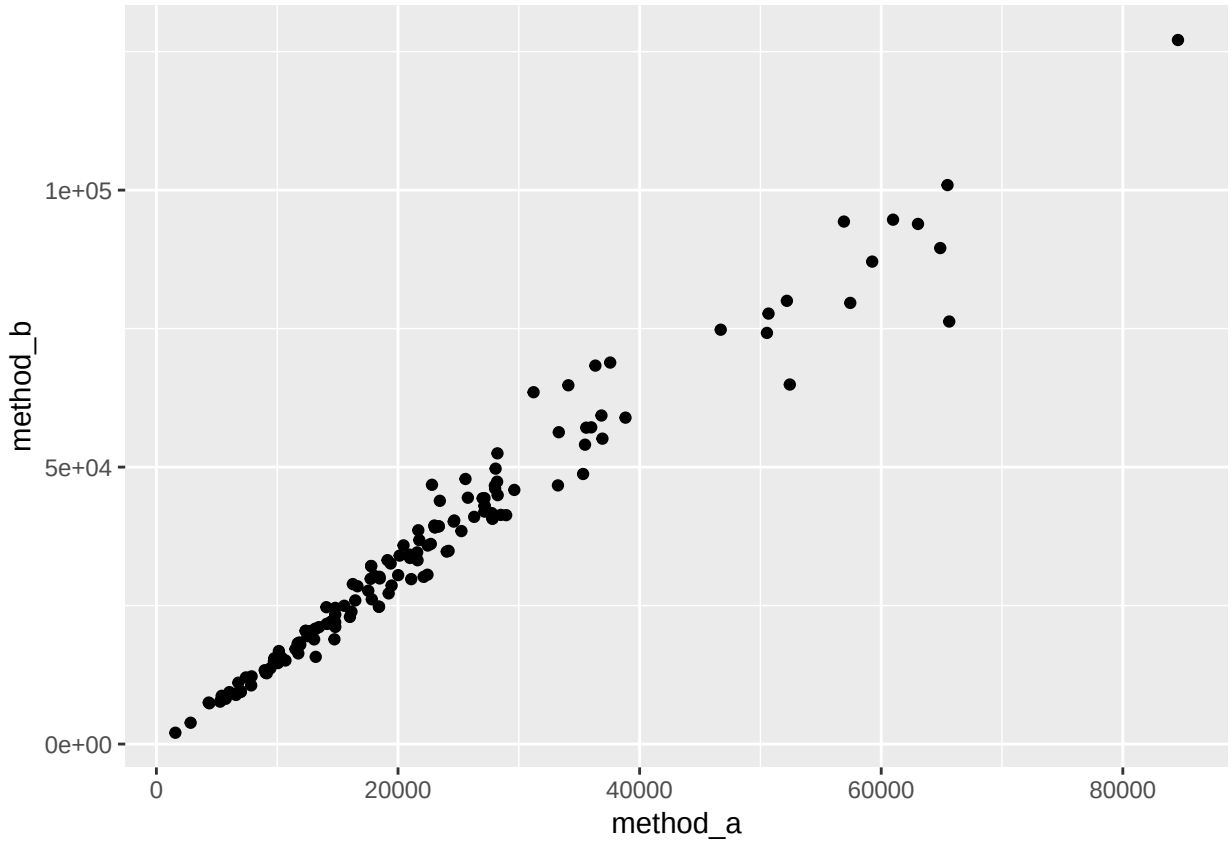
End Exercise

Regression

Simple linear regression

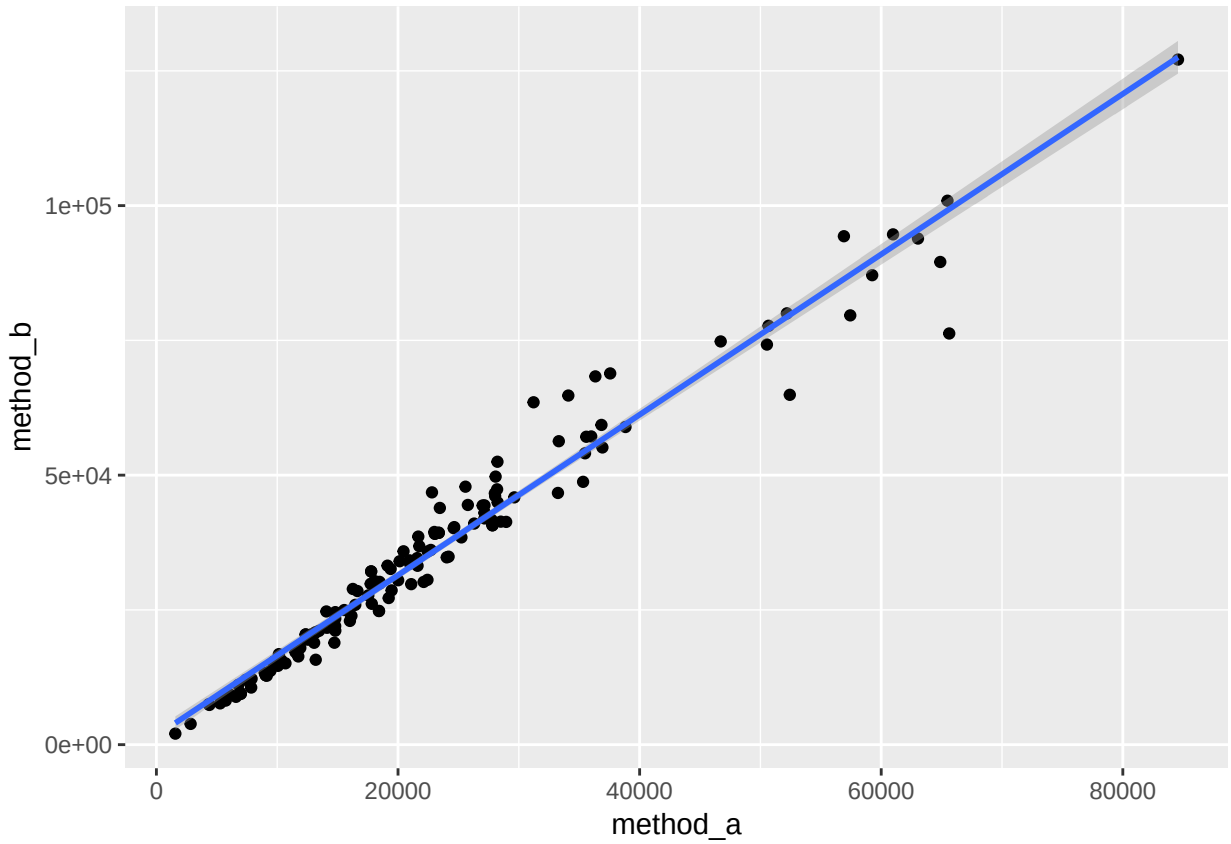
Let's begin by simply plotting `method_a` and `method_b` as a scatter plot. Notice how we are using the `aes()` to define "mappings" from our data to the x and y coordinates:

```
ggplot(hcg) +
  geom_point(aes(x = method_a, y = method_b))
```



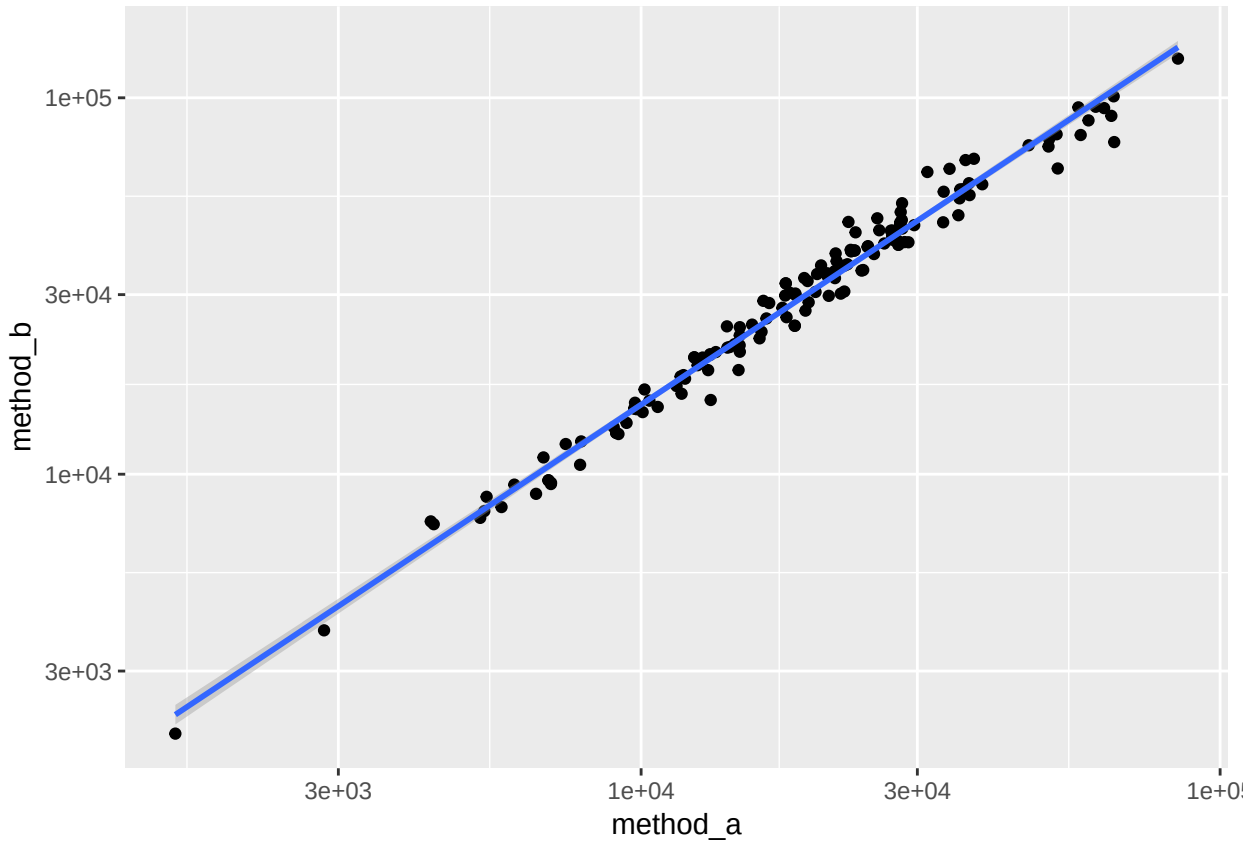
Adding a least-squares regression line is easy with a little bit of magic from `ggplot`. The `lm` (Linear Model) function does all the work here!

```
ggplot(hcg) +
  geom_point(aes(x = method_a, y = method_b)) +
  geom_smooth(method = "lm", aes(x = method_a, y = method_b))
```



It's tough to see the fit in the low result range, so we can transform our axis:

```
ggplot(hcg) +
  geom_point(aes(x = method_a, y = method_b)) +
  geom_smooth(method = "lm", aes(x = method_a, y = method_b)) +
  scale_x_log10() + scale_y_log10()
```



What if we want to just extract the coefficients of the linear model? We can utilize R's formula notation format and the `lm` function:

```
regression <- lm(method_b ~ method_a, hcg)
summary(regression)
```

```
##
## Call:
## lm(formula = method_b ~ method_a, data = hcg)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -23067.5  -2079.0   -617.9   2030.8  15375.0
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.682e+03  6.544e+02    2.57   0.0112 *
## method_a    1.488e+00  2.411e-02   61.72   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4345 on 144 degrees of freedom
## Multiple R-squared:  0.9636, Adjusted R-squared:  0.9633
## F-statistic:  3809 on 1 and 144 DF,  p-value: < 2.2e-16
```

```
coef(regression)
```

```
## (Intercept)    method_a
```

```
## 1682.003927    1.488099
```

Deming regression

In fact, a least-squares regression, while a good approximation for this type of data, assumes that the x-dimension has no measurement error so it minimizes errors only in the y-dimension. The *Deming regression* differs from the simple linear regression in that it accounts for errors in observations on both the x- and the y- axes, thus making it more suitable for estimating a best-fit line between two measured variables.

The Deming regression and other similar regressions are not built into the standard linear models function in R. Fortunately, there is a package written by staff at Roche called mcr to perform some of these bread and butter clinical laboratory regression methods. One issue with the package is that the output is not straightforward to work with. Many of the mcr functions produce a complex object that does not fit a standard tidy framework. The useful values can be extracted by digging more deeply into the output.

A Deming regression of methods A and B appears below. To pull the quantitative data we care out of the model, we use the `@` character and call the specific component of the object we care about (in this care "para").

```
#install.packages("mcr", dependencies = TRUE)
library(mcr)            # remember, you may need to install.packages("mcr") in the console first!
deming_results <- mcreg(hcg$method_a, hcg$method_b,
                        method.reg = "Deming")
deming_results@para         # "para" short for "parameters"
```

```
##                   EST SE         LCI         UCI
## Intercept 796.108298 NA -553.549433 1903.868311
## Slope       1.527165 NA    1.457119    1.605384
```

```
# this is a library/method specific term here
```

To see what slots (attributes) `deming_results` has beyond `para`, look at `glimpse(deming_results)`

Exercise 4:

Another issue in standard regression is if the random error scales with analyte concentration, then the observed absolute errors are higher for the higher concentrations. This means that error at the high-end is more heavily factored into the regression fit. One way of adjusting for this is weighting errors for high concentrations less.

Run the regression using the weighted deming method of the `mcreg` function. How do the slope and intercept differ?

```
wdeming_results <- mcreg(hcg$method_a, hcg$method_b,
                        method.reg = "WDeming")
```

```
## The global.sigma is calculated with Linnet's method
```

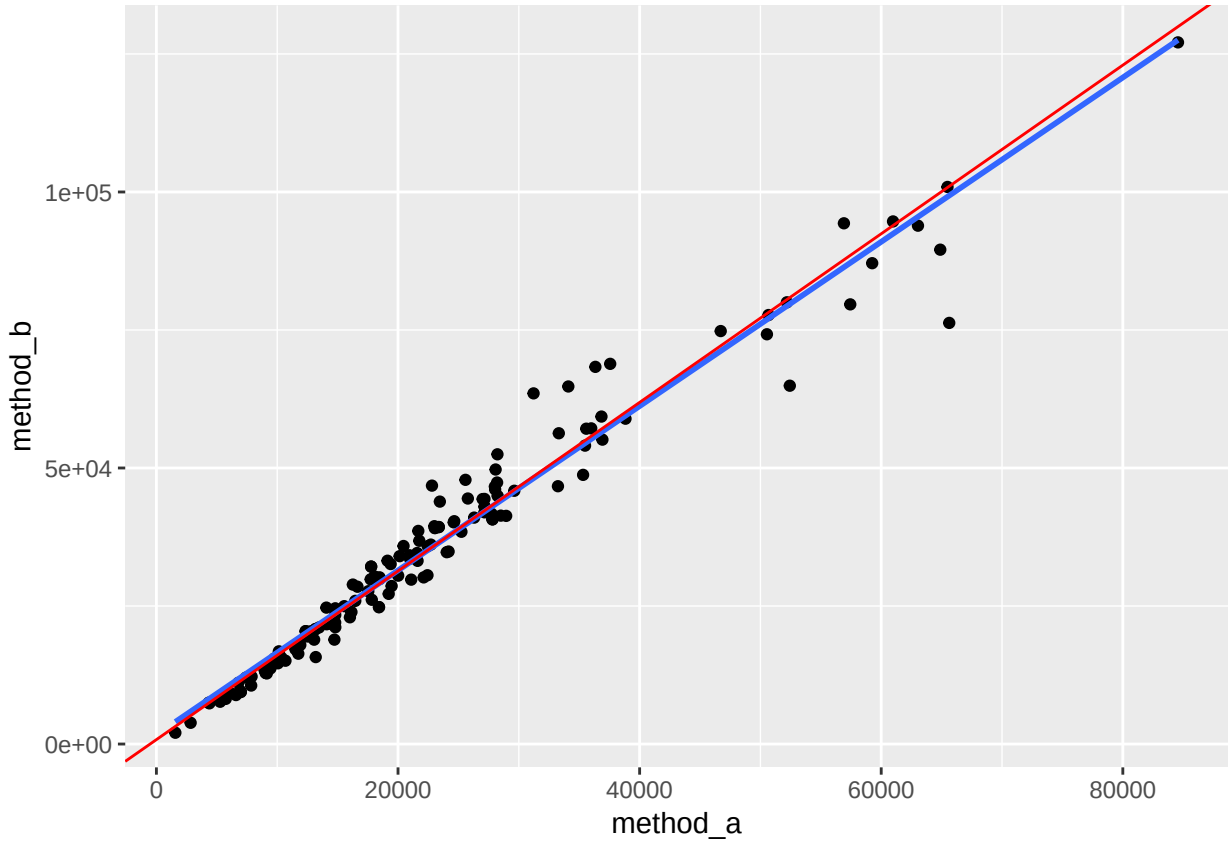
```
wdeming_results@para         # "para" short for "parameters"-- this is a library/method specific term h
```

```
##                   EST SE          LCI         UCI
## Intercept -595.104531 NA -1183.951083 -358.228834
## Slope        1.592368 NA     1.558162    1.638723
```

End exercise

Now let's add the lines to our plot. We can use the `geom_abline()` ggplot function to add a line with a slope and intercept. The intercept and slope are stored in `deming_results@para[1]` and `deming_results@para[2]`, respectively.

```
ggplot(hcg) +
  geom_point(aes(x = method_a, y = method_b)) +
  geom_smooth(method = "lm", aes(x = method_a, y = method_b), se = FALSE) +
  geom_abline(
    intercept = deming_results@para[1], slope = deming_results@para[2],
    color = "red"
  )
```



What about the weighted deming fit line? What about the subrange between 0 and 40000? Let's also add a 1:1 line to help interpret fit.

```
wdeming_results <- mcreg(hcg$method_a, hcg$method_b,
                         method.reg = "WDeming")
```

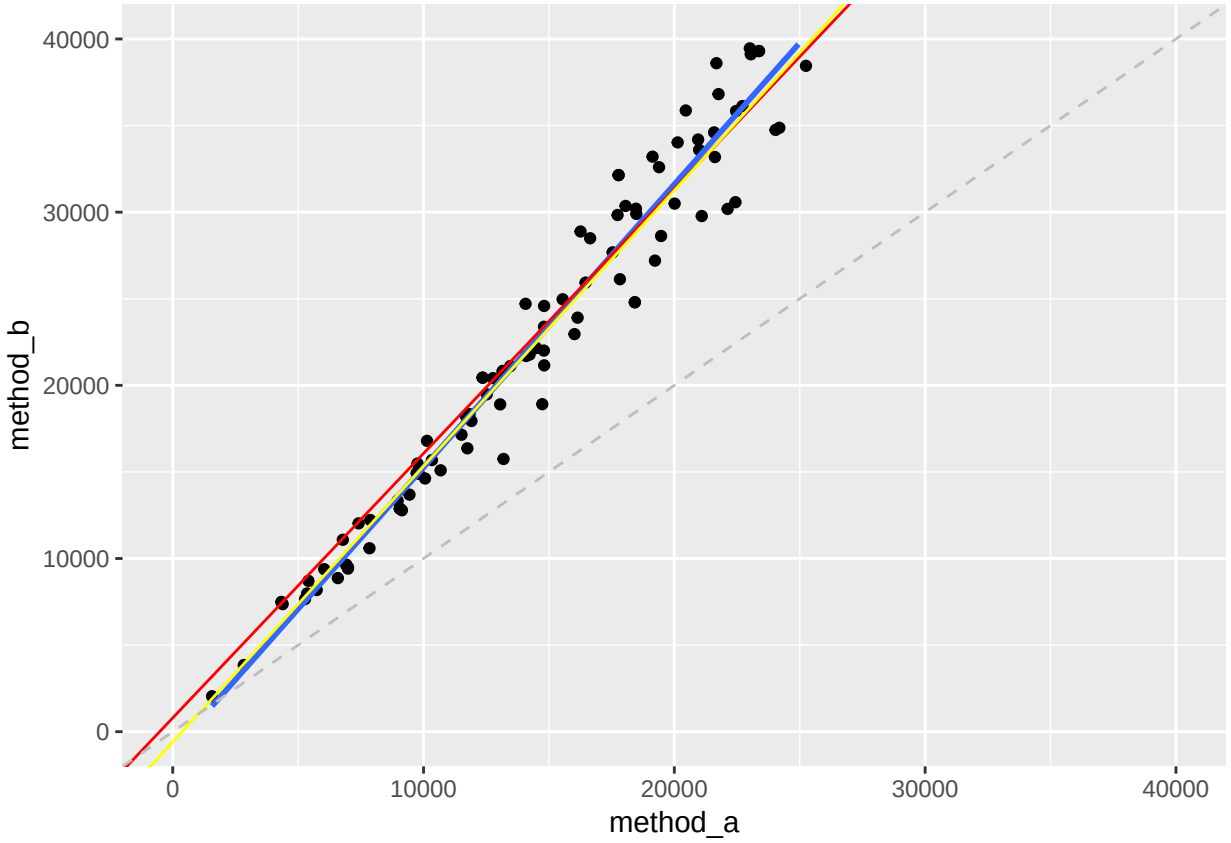
```
## The global.sigma is calculated with Linnet's method
```

```
ggplot(hcg) +
  geom_point(aes(x=method_a, y=method_b))  +
  geom_smooth(method = "lm", aes(x=method_a, y=method_b), se=FALSE) +   #blue
  geom_abline(intercept = deming_results@para[1], slope = deming_results@para[2],
              color="red") +
  geom_abline(intercept = wdeming_results@para[1], slope = wdeming_results@para[2],
              color="yellow") +
  xlim(0, 40000) + ylim(0, 40000) +
  geom_abline(intercept=0, slope=1, linetype=2, color="gray")
```

```
## Warning: Removed 51 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 51 rows containing missing values (geom_point).
```

```
## Warning: Removed 1 rows containing missing values (geom_smooth).
```



Passing-Bablock

Rather than performing a Deming, we may want to perform a Passing-Bablock regression instead. What is the distinct between Deming and Passing-Bablok? At a high level, the Deming regression assumes that errors are distributed normally, while Passing-Bablok is non-parametric and does not make this assumption.

We extract the useful model parameters similarly to the Deming regression:

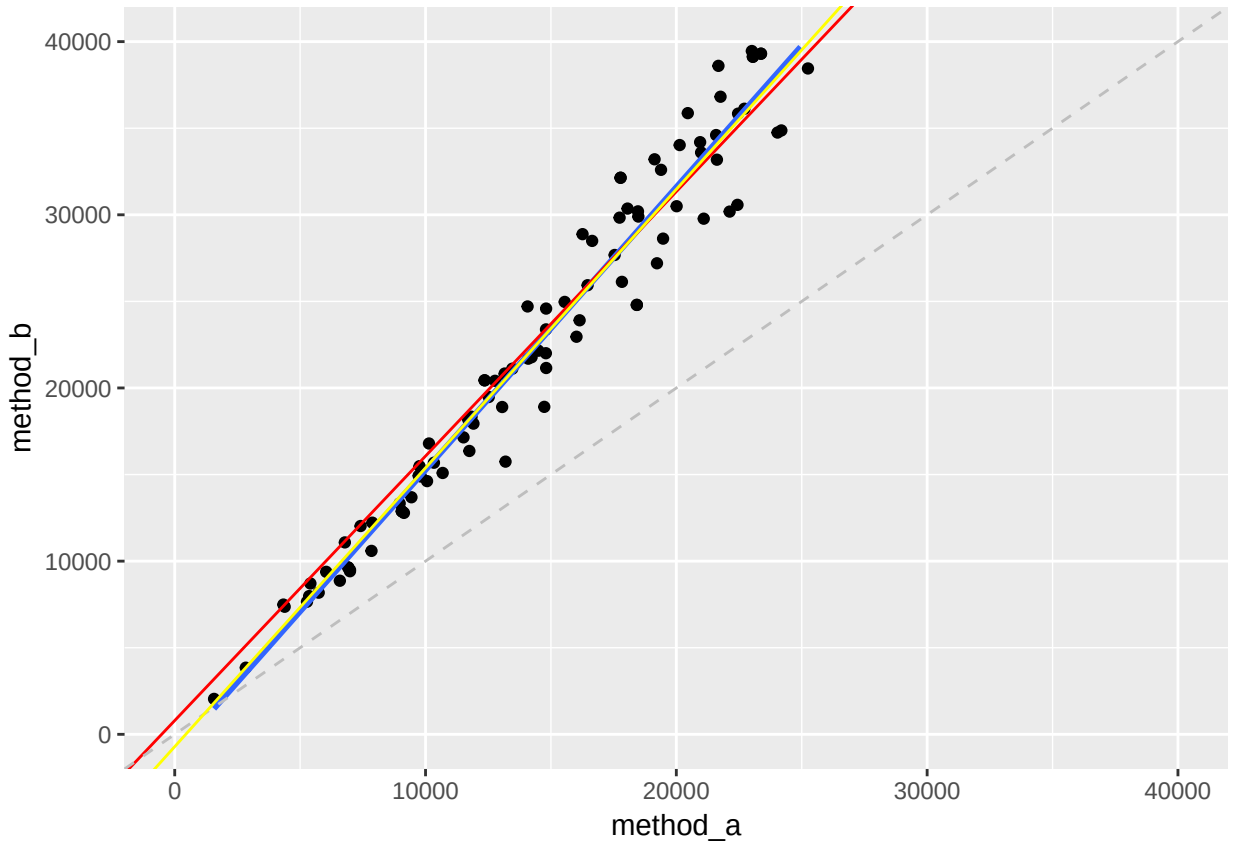
```
PB_results <- mcreg(hcg$method_a, hcg$method_b, method.reg = "PaBa")
PB_results@para
```

```
##                   EST SE          LCI        UCI
## Intercept -709.366764 NA -1500.907956 -78.337292
## Slope        1.609158 NA     1.554645   1.672028
```

Exercise 5: Add another `geom_abline` to the plot above for the Passing-Bablock regression coefficients determined above.

```
ggplot(hcg) +
  geom_point(aes(x=method_a, y=method_b))  +
  geom_smooth(method = "lm", aes(x=method_a, y=method_b), se=FALSE) +  #blue
  geom_abline(intercept = deming_results@para[1], slope = deming_results@para[2],
              color="red") +
  xlim(0, 40000) + ylim(0, 40000) +
  geom_abline(intercept=0, slope=1, linetype=2, color="gray") +
  geom_abline(intercept = PB_results@para[1], slope = PB_results@para[2], color="yellow")
```

```
## Warning: Removed 51 rows containing non-finite values (stat_smooth).
## Warning: Removed 51 rows containing missing values (geom_point).
## Warning: Removed 1 rows containing missing values (geom_smooth).
```



End Exercise

Compare methods by concordance relative to decision thresholds

Next, let's compare method A and B using decision thresholds. For the purpose of this tutorial, we will simply use 25,000 as our threshold.

```
threshold <- 25000

tmp <- hcg %>%
  mutate(method_a_pos = method_a > threshold,    # Create binary indicator for method_a
         method_b_pos = method_b > threshold)
  table(x=tmp$method_a_pos, y=tmp$method_b_pos)
```

```
##         y
## x        FALSE TRUE
##   FALSE     58   40
##   TRUE       0   48
```

Looking at this table, method_a and method_b are *discordant* across our threshold in 40 cases, and *concordant* in 58 + 48 cases.

A tidy way to do this without using `tmp` and `table` is:

```
threshold <- 25000

hcg %>%
  mutate(method_a_pos = method_a > threshold,    # Create binary indicator for method_a
         method_b_pos = method_b > threshold) %>%
  count(method_a_pos, method_b_pos)
```

```
## # A tibble: 3 x 3
##   method_a_pos method_b_pos     n
##   <lgl>        <lgl>        <int>
## 1 FALSE        FALSE           58
## 2 FALSE        TRUE            40
## 3 TRUE         TRUE            48
```

Now to convert this into a standard concordance table:

```
hcg %>%
  mutate(method_a_pos = method_a > threshold,    # Create binary indicator for method_a
         method_b_pos = method_b > threshold) %>%
  count(method_a_pos, method_b_pos)  %>%
  spread(method_b_pos, n, fill=0, sep=".")    # Spreads method_b_pos from a single variable
```

```
## # A tibble: 2 x 3
##   method_a_pos method_b_pos.FALSE method_b_pos.TRUE
##   <lgl>                     <dbl>             <dbl>
## 1 FALSE                        58                40
## 2 TRUE                          0                48
```

```
  # to a variable for each value
```

Exercise 6

Write code to compare accuracy across two different decision thresholds (25000 and 50000, for example)

Hint #1: In the `mutate` function, use the `case_when` function to break a numerical range into multiple a set of factor levels (categories).

Hint #2: Look at previous code for inspiration!

```
hcg %>%
  mutate(method_a_bin = case_when(
    method_a <= 25000 ~ "low",
    method_a > 25000 & method_a <= 50000 ~ "middle",
    method_a > 50000 ~ "high")) %>%
  mutate(method_b_bin = case_when(
    method_b <= 25000 ~ "low",
    method_b > 25000 & method_b <= 50000 ~ "middle",
    method_b > 50000 ~ "high"
  )) %>%
  count(method_a_bin, method_b_bin)  %>%
  spread(key = method_b_bin, value = n, fill = 0) # Spreads method_b_pos from a single variable to a va
```

```
## # A tibble: 3 x 4
##   method_a_bin  high   low middle
##   <chr>        <dbl> <dbl>  <dbl>
## 1 high            13     0      0
## 2 low              0    58     40
## 3 middle          13     0     22
```

End Exercise

Acknowledgement

This lesson was adapted from Daniel Herman's lesson on method comparison that was taught in the AACC Basic R Course.

Summary

- When comparing methods, a useful first step is a comparison of summary statistics and distributions
- Performing a statistical test to assess whether the distributions from two methods are likely to have originated from the same distribution is helpful, and the test choice should be informed by the types of distributions
- Built-in models are available for the most common types of fits in ggplot2
- However, additional complexity is needed to overlay more sophisticated models such as Deming regression and Passing Bablok
  - Data from more complex models can be extracted and used to manually generate lines of fit

Beyond the csv – parsing xml files

Flat and Structured Files

The functions from readr *read_csv*, *read_tsv*, and *read_excel* (discussed in Lesson 5) open so called "Flat" files that are often called 'flat' because they have a simple structure of delimited rows and columns. Some raw instrument data is difficult to describe fully using a simple row and column approach. Several formats have been created to represent more structured data, and some have evolved into standards used across instrument applications. XML is an example of a sophisticated format that has been used to implement many standards and defacto standards.

The extensible markup language (xml)

XML is first, simply a text file which follows very specific grammaric rules. On top of this grammar, there is a semantic layer which is application specific. There are packages for reading generic XML files such as **xml2**. For this lesson, we will focus on a more specific XML file format for representing mass spectrometry raw data. Using XML, several specifications (definition of how the semantic content for mass spectrometry data is represented using the XML grammar)

XML is a topic all to itself with whole books available on almost ever aspect of the system. In general, if you find you need to read an XML file, it's best to look for a package to read the specific type of XML file you need. XML was designed for programmers, and unless you have a very specific reason, often the most pragmatic approach to XML is to look for a way to avoid having to read it directly.

With standards like mzXML, mzData, mzML, and other mass spectrometry formats (including the older netCDF format - not an XML format), there are many groups who have contributed packages to CRAN to work with these files. It is important to find a package which works the way you need, since each package will have different capabilities and different requirements.

One of the more useful XML packages for mass spectrometry is currently **mzR** which is part of the **Bioconductor** suite of tools. Bioconductor is a giant collection of packages, and sometimes only part of the suite is needed. The Bioconductor group has provided a mechanism to install just a part of the overall collection, which we will use to install mzR:

```
## try http:// if https:// URLs are not supported
#source("https://bioconductor.org/biocLite.R")
#biocLite("mzR")

library(mzR)
```

```
## Loading required package: Rcpp
```

```
# More on the mzR package: https://bioconductor.org/packages/devel/bioc/vignettes/mzR/inst/doc/mzR.html

mzXML <- openMSfile("data/7MIX_STD_110802_1.mzXML")

header(mzXML,10)
```

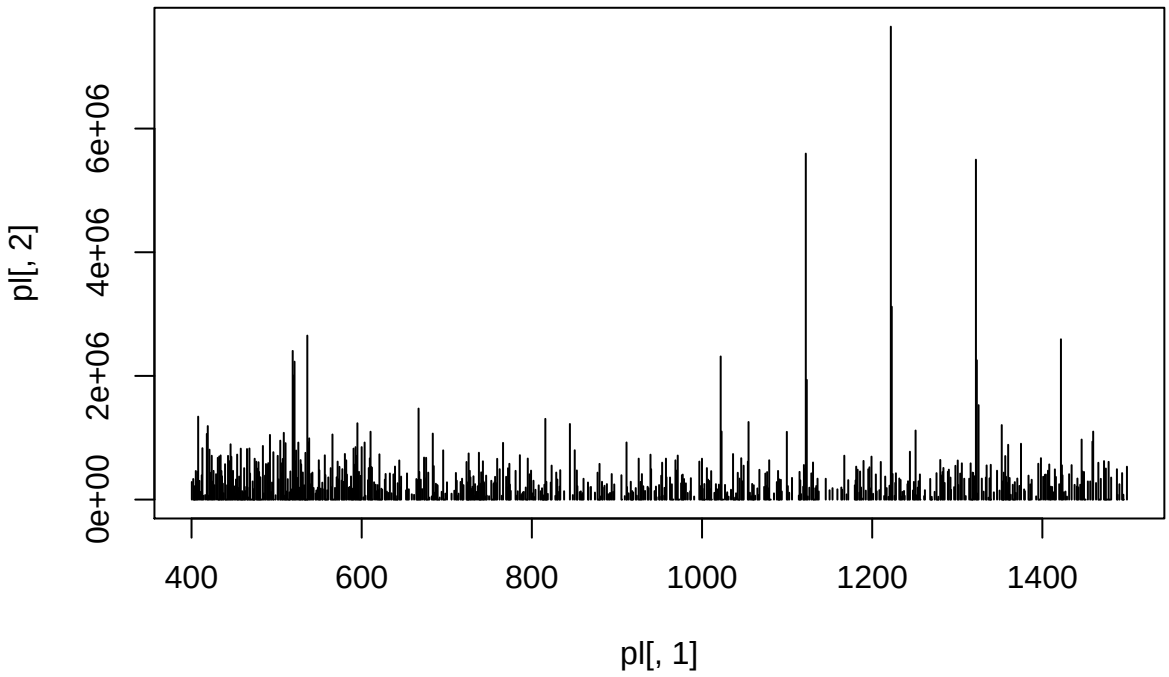
```
## $seqNum
## [1] 10
##
## $acquisitionNum
## [1] 10
##
## $msLevel
## [1] 1
##
## $polarity
## [1] 1
##
## $peaksCount
## [1] 683
##
## $totIonCurrent
## [1] 257119000
##
## $retentionTime
## [1] 13.69
##
## $basePeakMZ
## [1] 1221.92
##
## $basePeakIntensity
## [1] 7647700
##
## $collisionEnergy
## [1] 0
##
## $ionisationEnergy
## [1] 0
##
## $lowMZ
## [1] 400
##
## $highMZ
## [1] 1500
##
## $precursorScanNum
```

```
## [1] 0
##
## $precursorMZ
## [1] 0
##
## $precursorCharge
## [1] 0
##
## $precursorIntensity
## [1] 0
##
## $mergedScan
## [1] 0
##
## $mergedResultScanNum
## [1] 0
##
## $mergedResultStartScanNum
## [1] 0
##
## $mergedResultEndScanNum
## [1] 0
##
## $injectionTime
## [1] 0
##
## $filterString
## [1] NA
##
## $spectrumId
## [1] "controllerType=0 controllerNumber=1 scan=10"
##
## $centroided
## [1] TRUE
```

```
peaksCount(mzXML,10)
```

```
## [1] 683
```

```
pl <- peaks(mzXML,10)
plot(pl[,1], pl[,2], type="h", lwd=1)
```




```
header(mzXML,11)
```

```
## $seqNum
## [1] 11
##
## $acquisitionNum
## [1] 11
##
## $msLevel
## [1] 2
##
## $polarity
## [1] 1
##
## $peaksCount
## [1] 83
##
## $totIonCurrent
## [1] 6800420
##
## $retentionTime
## [1] 14.86
##
## $basePeakMZ
## [1] 989.947
##
## $basePeakIntensity
## [1] 2482250
##
## $collisionEnergy
## [1] 35
##
## $ionisationEnergy
```

```
## [1] 0
##
## $lowMZ
## [1] 295
##
## $highMZ
## [1] 2000
##
## $precursorScanNum
## [1] 10
##
## $precursorMZ
## [1] 1121.98
##
## $precursorCharge
## [1] 0
##
## $precursorIntensity
## [1] 284516
##
## $mergedScan
## [1] 0
##
## $mergedResultScanNum
## [1] 0
##
## $mergedResultStartScanNum
## [1] 0
##
## $mergedResultEndScanNum
## [1] 0
##
## $injectionTime
## [1] 0
##
## $filterString
## [1] NA
##
## $spectrumId
## [1] "controllerType=0 controllerNumber=1 scan=11"
##
## $centroided
## [1] TRUE
```

```
hdx<-header(mzXML)
```

Exercise 1: Find the MS Level 1 (Survey) Scans

Notice that the header contains the variable `msLevel` which is set to 1 for a survey scan (MS) or 2 for an MS/MS scan (MSˆn). In the exercise, you will create chromatogram from the total ion current of all the MS survey scans. Step 1 will be to create a vector of the survey scan numbers:

```
survey <- which(hdx$msLevel == 1)
```

End of Exercise

Exercise 2: Create a `tibble` to hold and label the desired spectra Next create a tibble to hold the sum of the ion current from each survey spectrum:

```
totalIonCurrent = tibble(index = survey, time = 0.0, inten = 0.0)
```

End of Exercise

Exercise 3: Load the `tibble` with the total ion current from each spectrum Finally fill the tibble with the information from the XML file using functions of mzR

```
for( i in 1:length(survey)) {
  totalIonCurrent$time[i] <- header(mzXML,survey[i])$retentionTime
  totalIonCurrent$inten[i] <- sum(peaks(mzXML,survey[i])[,2])
}
```

End of Exercise

Now the TIC can be plotted using either base or ggplot functions:

```
plot(totalIonCurrent$time, totalIonCurrent$inten, type="l")
```

Summary

- **xml2** can be used for for reading and writing generic xml files
- **mzR** can be used to read and write most mass spectrometry file formats
- XML formats are text files which are verbose, large and slow to process, but they are resistent to changes in instrumentation, computer platforms and experimental designs.

Exploratory data analysis

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

Variable	Description
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

```
orders_raw <- read_excel("data/orders_data_set.xlsx") %>%
  clean_names()
order_details <- read_csv("data/order_details.csv") %>%
  clean_names()
```

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

```r
# relationship is one-to-one so inner_join/left_join/right_join all work
orders_joined <- orders_raw %>%
  left_join(order_details, by = "order_id")
# may be some variation in which fields you choose to turn into factors
# numeric codes could be factors too
orders <- orders_joined %>%
  mutate(description = as.factor(description),
         proc_code = as.factor(proc_code),
         order_class_c_descr = as.factor(order_class_c_descr),
         lab_status_c_descr = as.factor(lab_status_c_descr),
         order_status_c_descr = as.factor(order_status_c_descr),
         reason_for_canc_c_descr = as.factor(reason_for_canc_c_descr),
         department = as.factor(department),
         ordering_route = as.factor(ordering_route),
         pref_list_type = as.factor(pref_list_type)) %>%
  mutate(order_month = month(order_time),
         order_week = week(order_time))
summary(orders)
```

```
##     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
##  Clinician Orders   : 4174   Clinic Preference List  :30483
##  External Order     :    5   None                    : 6422
##  OP Orders Navigator:36541   Provider Preference List: 8097
##  Results Console    :  179
##  SmartSet           : 4101
##  NA's               :    2
##
##   order_month       order_week
##  Min.   : 8.000   Min.   :33.00
##  1st Qu.: 9.000   1st Qu.:36.00
##  Median : 9.000   Median :39.00
##  Mean   : 9.351   Mean   :38.98
##  3rd Qu.:10.000   3rd Qu.:42.00
##  Max.   :11.000   Max.   :45.00
##
```

End Exercise

Exploration of data

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

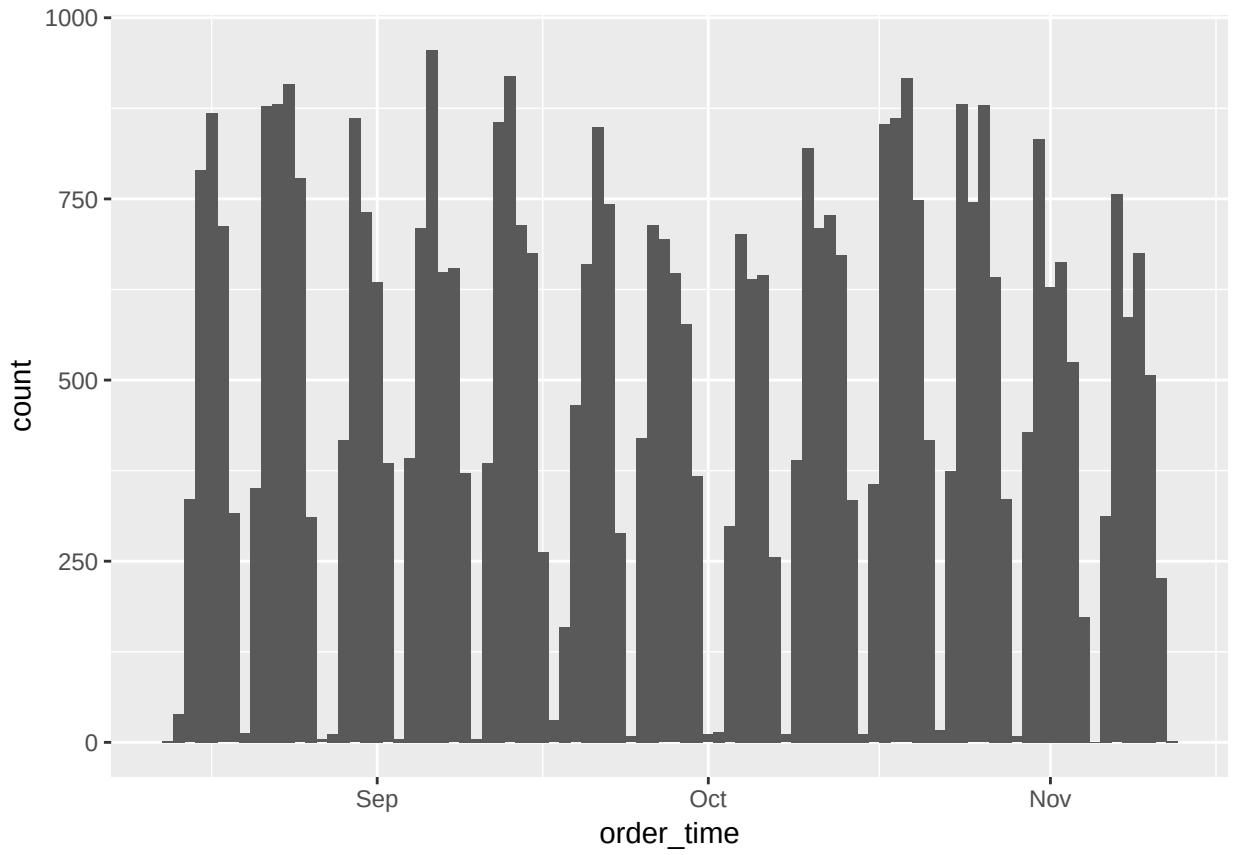
- Overall orders over time - are there any dramatic changes in volume over the time period?
- Which tests are most commonly ordered?
- What is the overall cancellation rate and has it changed over time?

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

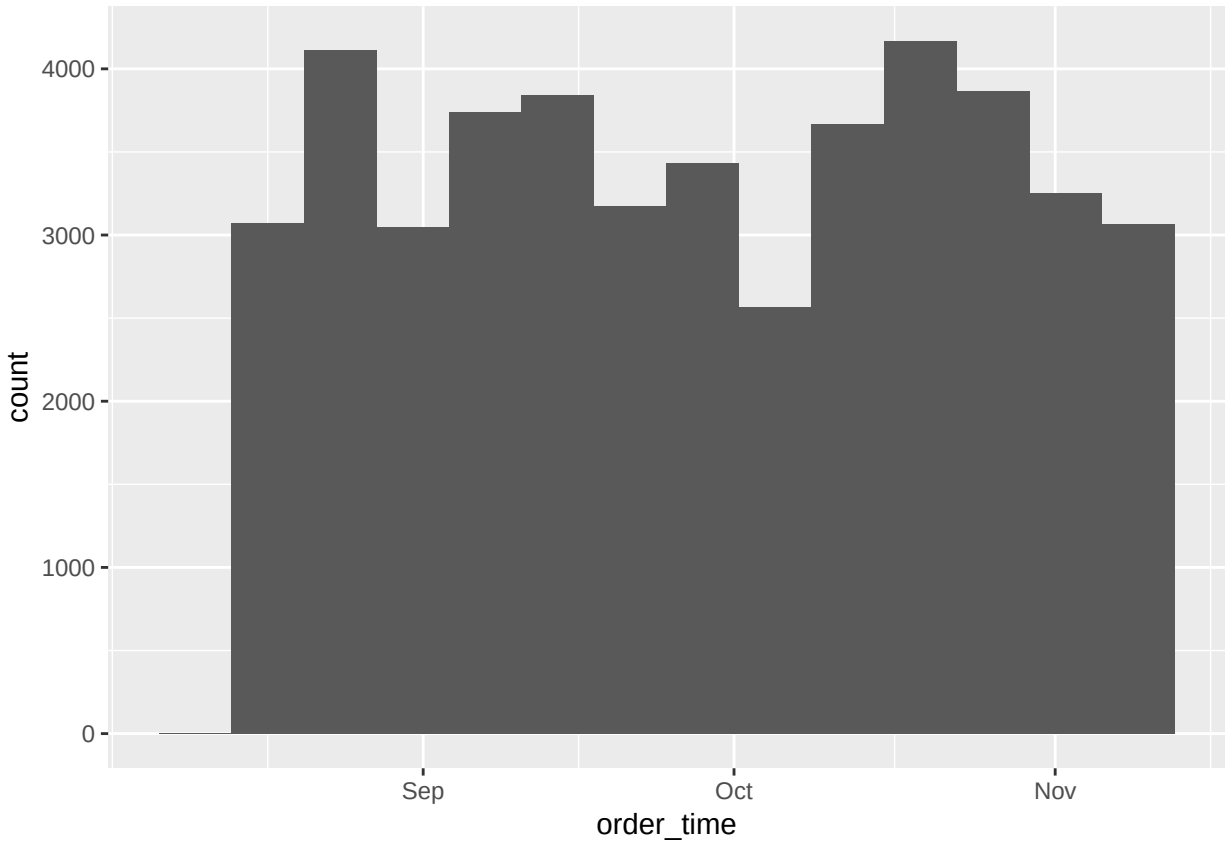
Exercise 2

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.)

```
ggplot(data = orders) +
  geom_histogram(mapping = aes(x = order_time), binwidth = 60*60*24)
```




```
ggplot(data = orders) +
  geom_histogram(mapping = aes(x = order_time), binwidth = 60*60*24*7)
```



2. Plot or tabulate the breakdown of test orders in the data set (using description or procedure code). Focus on the top 25 tests.

```
orders %>%
  tabyl(description) %>%
  arrange(desc(n)) %>%
  top_n(25)
```

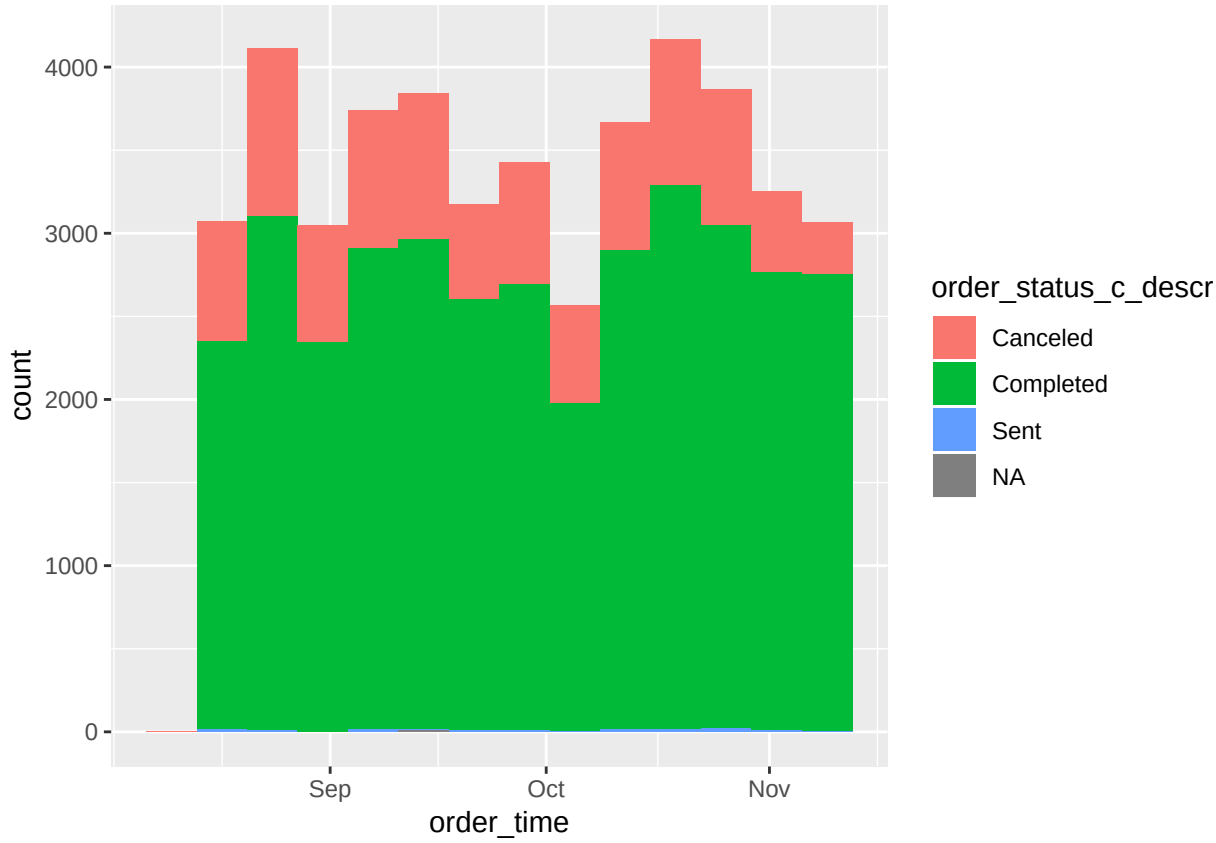
```
## Selecting by percent
```

```
##                        description    n     percent
## 1   COMPREHENSIVE METABOLIC PANEL 3639 0.080863073
## 2            HEMOGLOBIN A1C, HPLC 2470 0.054886449
## 3                       CBC, DIFF 2393 0.053175414
## 4           BASIC METABOLIC PANEL 2174 0.048308964
## 5   GC&CHLAM NUCLEIC ACID DETECTN 2164 0.048086752
## 6                  CBC (HEMOGRAM) 1979 0.043975823
## 7                     LIPID PANEL 1848 0.041064842
## 8           HIV1 RNA QUANTITATION 1314 0.029198702
## 9     THYROID STIMULATING HORMONE 1288 0.028620950
## 10  SEROLOGIC SYPHILIS PANEL, SRM 1081 0.024021155
## 11 CHR PAIN DRUG RSK1 W/CNSLT,URN  985 0.021887916
## 12  HIV ANTIGEN AND ANTIBODY SCRN  887 0.019710235
## 13 URINE SCREEN, MICROALBUMINURIA  846 0.018799164
## 14   GLUCOSE, WHOLE BLOOD, ONSITE  822 0.018265855
## 15         HEPATIC FUNCTION PANEL  741 0.016465935
## 16       URINALYSIS COMPLETE, URN  682 0.015154882
## 17 HEPATITIS C AB WITH REFLEX PCR  663 0.014732679
```

```
## 18                       SED RATE  608 0.013510511
## 19            CRP, HIGH SENSITIVITY  591 0.013132750
## 20                 PROTHROMBIN TIME  586 0.013021643
## 21           VITAMIN D (25 HYDROXY)  577 0.012821652
## 22           HEPATITIS C RNA, QUANT  501 0.011132839
## 23 URINALYSIS WITH REFLEX CULTURE  459 0.010199547
## 24 T CELL SUBSET _ CD4 & CD8 ONLY  440 0.009777343
## 25                 LAB ADD ON ORDER  435 0.009666237
```

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

```
ggplot(data = orders) +
  geom_histogram(mapping = aes(x = order_time, fill = order_status_c_descr), binwidth = 60*60*24*7)
```




```
orders %>%
  tabyl(order_status_c_descr, order_week) %>%
  adorn_totals("row") %>% # tabulate operations below across rows
  adorn_percentages("row") %>% # express counts as percentages
  adorn_pct_formatting() %>% # clean up percentages for nicer printing
  adorn_ns() # add back in counts (N's)
```

```
##  order_status_c_descr           33           34          35          36
##              Canceled  7.8%  (719) 10.8% (1004) 7.6%  (703) 8.9%  (824)
##             Completed  6.6% (2338)  8.7% (3093) 6.6% (2340) 8.2% (2898)
##                  Sent 11.8%   (19)  8.1%   (13) 1.2%    (2) 9.9%   (16)
##                  <NA>  0.0%    (0)  0.0%    (0) 0.0%    (0) 0.0%    (0)
##                 Total  6.8% (3076)  9.1% (4110) 6.8% (3045) 8.3% (3738)
##            37          38          39          40          41          42
##   9.2%  (857) 6.3%  (587) 8.0%  (739) 6.3%  (584)  8.3%  (769) 9.3%  (866)
```

```
##   8.3% (2949) 7.3% (2588) 7.6% (2685) 5.5% (1972)  8.1% (2884) 9.2% (3279)
##   1.2%    (2) 7.5%   (12) 6.8%   (11) 5.0%    (8) 10.6%   (17) 9.3%   (15)
##  88.9%   (16) 5.6%    (1) 0.0%    (0) 0.0%    (0)  0.0%    (0) 5.6%    (1)
##   8.5% (3824) 7.1% (3188) 7.6% (3435) 5.7% (2564)  8.2% (3670) 9.2% (4161)
##            43          44          45
##   8.9%  (823) 5.2%  (483) 3.4%  (312)
##   8.5% (3026) 7.7% (2755) 7.7% (2746)
##  16.1%   (26) 7.5%   (12) 5.0%    (8)
##   0.0%    (0) 0.0%    (0) 0.0%    (0)
##   8.6% (3875) 7.2% (3250) 6.8% (3066)
```

4. Explore whether there are specific tests that are cancelled more frequently than others. Focus on the top 25 tests.

```
orders %>%
  #filter(description %in% orders_top25$description) %>%
  tabyl(description, order_status_c_descr) %>%
  arrange(desc(Completed)) %>%
  top_n(25, Completed) %>%
  adorn_totals("row") %>% # tabulate operations below across rows
  adorn_percentages("row") %>% # express counts as percentages
  adorn_pct_formatting() %>% # clean up percentages for nicer printing
  adorn_ns() # add back in counts (N's)
```

```
##                     description     Canceled     Completed      Sent
##   COMPREHENSIVE METABOLIC PANEL 17.3%  (630) 82.4%  (3000) 0.2%  (7)
##            HEMOGLOBIN A1C, HPLC 17.7%  (436) 82.2%  (2031) 0.1%  (2)
##                       CBC, DIFF 15.6%  (374) 84.0%  (2009) 0.4%  (9)
##   GC&CHLAM NUCLEIC ACID DETECTN 13.8%  (299) 86.2%  (1865) 0.0%  (0)
##           BASIC METABOLIC PANEL 22.8%  (496) 77.0%  (1674) 0.2%  (4)
##                 CBC (HEMOGRAM) 21.9%  (433) 77.9%  (1542) 0.2%  (3)
##                    LIPID PANEL 22.4%  (414) 77.4%  (1431) 0.1%  (1)
##           HIV1 RNA QUANTITATION 16.1%  (211) 83.9%  (1103) 0.0%  (0)
##     THYROID STIMULATING HORMONE 17.2%  (221) 82.5%  (1063) 0.2%  (3)
##  CHR PAIN DRUG RSK1 W/CNSLT,URN  9.4%   (93) 90.6%   (892) 0.0%  (0)
##    GLUCOSE, WHOLE BLOOD, ONSITE  5.4%   (44) 93.4%   (768) 1.2% (10)
##   SEROLOGIC SYPHILIS PANEL, SRM 29.3%  (317) 70.6%   (763) 0.0%  (0)
##   HIV ANTIGEN AND ANTIBODY SCRN 24.7%  (219) 75.3%   (668) 0.0%  (0)
##  URINE SCREEN, MICROALBUMINURIA 22.2%  (188) 77.5%   (656) 0.1%  (1)
##  HEPATITIS C AB WITH REFLEX PCR 20.1%  (133) 79.9%   (530) 0.0%  (0)
##        URINALYSIS COMPLETE, URN 24.2%  (165) 75.8%   (517) 0.0%  (0)
##          HEPATIC FUNCTION PANEL 30.4%  (225) 69.4%   (514) 0.3%  (2)
##                        SED RATE 17.8%  (108) 81.2%   (494) 1.0%  (6)
##           CRP, HIGH SENSITIVITY 17.8%  (105) 81.2%   (480) 1.0%  (6)
##          VITAMIN D (25 HYDROXY) 17.0%   (98) 83.0%   (479) 0.0%  (0)
##                PROTHROMBIN TIME 24.1%  (141) 75.8%   (444) 0.2%  (1)
##                LAB ADD ON ORDER  0.9%    (4) 99.1%   (431) 0.0%  (0)
##          HEPATITIS C RNA, QUANT 26.7%  (134) 72.9%   (365) 0.4%  (2)
##        PAP SMEAR (OPTIONAL HPV)  6.6%   (26) 92.2%   (365) 1.3%  (5)
##  T CELL SUBSET _ CD4 & CD8 ONLY 17.3%   (76) 82.7%   (364) 0.0%  (0)
##                          Total 18.6% (5590) 81.2% (24448) 0.2% (62)
##        NA_
##  0.1%  (2)
##  0.0%  (1)
##  0.0%  (1)
```

```
##  0.0%  (0)
##  0.0%  (0)
##  0.1%  (1)
##  0.1%  (2)
##  0.0%  (0)
##  0.1%  (1)
##  0.0%  (0)
##  0.0%  (0)
##  0.1%  (1)
##  0.0%  (0)
##  0.1%  (1)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0% (10)
```

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?

Clinics ordering the highest volume of tests:

```
orders %>%
  tabyl(department) %>%
  arrange(desc(n))
```

```
##                       department     n    percent
## 1      INFECTIOUS DISEASE CLINIC 11861 0.26356606
## 2       INTERNAL MEDICINE CLINIC  6330 0.14066042
## 3         FAMILY MEDICINE CLINIC  3501 0.07779654
## 4            NEIGHBORHOOD CLINIC  3128 0.06950802
## 5  INTERNATIONAL MEDICINE CLINIC  2499 0.05553087
## 6            RHEUMATOLOGY CLINIC  2422 0.05381983
## 7              HEPATOLOGY CLINIC  2211 0.04913115
## 8                  OB GYN CLINIC  2179 0.04842007
## 9              NEPHROLOGY CLINIC  2093 0.04650904
## 10          ENDOCRINOLOGY CLINIC  1486 0.03302075
```

```
## 11             CARDIOLOGY CLINIC  1026 0.02279899
## 12              GERIATRIC CLINIC  1015 0.02255455
## 13             PEDIATRIC CLINICS   902 0.02004355
## 14       GASTROENTEROLOGY CLINIC   781 0.01735478
## 15           NEUROSURGERY CLINIC   724 0.01608817
## 16           OPHTHAMOLOGY CLINIC   657 0.01459935
## 17         LIPID DISORDERS CLINIC   601 0.01335496
## 18    HEMATOLOGY-ONCOLOGY CLINIC   580 0.01288832
## 19      BEHAVIORAL HEALTH CLINIC   503 0.01117728
## 20      TRANSITIONAL CARE CLINIC   503 0.01117728
```

Clinics with the highest numbers and rates of cancels:

```
orders %>%
  tabyl(department, order_status_c_descr) %>%
  arrange(desc(Completed)) %>%
  adorn_totals("row") %>%
  adorn_percentages("row") %>%
  adorn_pct_formatting() %>%
  adorn_ns()
```

```
##                     department     Canceled     Completed       Sent
##      INFECTIOUS DISEASE CLINIC 20.0% (2378) 79.8%  (9463) 0.2%  (20)
##       INTERNAL MEDICINE CLINIC 23.4% (1482) 76.5%  (4843) 0.1%   (4)
##         FAMILY MEDICINE CLINIC 11.8%  (412) 88.2%  (3087) 0.0%   (1)
##            NEIGHBORHOOD CLINIC 15.6%  (487) 84.0%  (2626) 0.0%   (1)
##  INTERNATIONAL MEDICINE CLINIC 12.1%  (302) 87.7%  (2191) 0.2%   (6)
##            RHEUMATOLOGY CLINIC 17.6%  (427) 80.9%  (1959) 1.5%  (36)
##                 OB GYN CLINIC 10.8%  (235) 88.8%  (1936) 0.4%   (8)
##              HEPATOLOGY CLINIC 24.8%  (548) 74.7%  (1652) 0.5%  (11)
##              NEPHROLOGY CLINIC 37.0%  (775) 62.3%  (1303) 0.7%  (15)
##           ENDOCRINOLOGY CLINIC 17.6%  (262) 80.0%  (1189) 2.4%  (35)
##               GERIATRIC CLINIC 19.4%  (197) 80.6%   (818) 0.0%   (0)
##              CARDIOLOGY CLINIC 32.7%  (335) 67.3%   (690) 0.1%   (1)
##              PEDIATRIC CLINICS 29.3%  (264) 70.5%   (636) 0.2%   (2)
##            NEUROSURGERY CLINIC 16.9%  (122) 82.5%   (597) 0.7%   (5)
##            OPHTHAMOLOGY CLINIC 15.1%   (99) 84.9%   (558) 0.0%   (0)
##         GASTROENTEROLOGY CLINIC 34.2%  (267) 64.7%   (505) 1.2%   (9)
##       BEHAVIORAL HEALTH CLINIC 12.1%   (61) 87.9%   (442) 0.0%   (0)
##     HEMATOLOGY-ONCOLOGY CLINIC 26.4%  (153) 72.8%   (422) 0.7%   (4)
##       TRANSITIONAL CARE CLINIC 17.7%   (89) 81.7%   (411) 0.6%   (3)
##         LIPID DISORDERS CLINIC 62.4%  (375) 37.4%   (225) 0.0%   (0)
##                         Total 20.6% (9270) 79.0% (35553) 0.4% (161)
##        NA_
##  0.0%  (0)
##  0.0%  (1)
##  0.0%  (1)
##  0.4% (14)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
```

```
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.0%  (0)
##  0.2%  (1)
##  0.0%  (0)
##  0.2%  (1)
##  0.0% (18)
```

Rates of clinic collections:

```
orders %>%
  tabyl(department, order_class_c_descr) %>%
  arrange(desc(`Clinic Collect`)) %>%
  adorn_totals("row") %>%
  adorn_percentages("row") %>%
  adorn_pct_formatting() %>%
  adorn_ns()
```

```
##                     department Clinic Collect   External Historical
##          FAMILY MEDICINE CLINIC   71.3% (2497) 0.0%   (0)   0.0% (0)
##             NEIGHBORHOOD CLINIC   71.5% (2235) 0.0%   (1)   0.0% (0)
##                   OB GYN CLINIC   33.5%  (731) 0.0%   (0)   0.0% (0)
##        BEHAVIORAL HEALTH CLINIC   89.5%  (450) 0.0%   (0)   0.0% (0)
##       INFECTIOUS DISEASE CLINIC    1.4%  (163) 2.5% (302)   0.0% (1)
##        INTERNAL MEDICINE CLINIC    2.5%  (161) 0.0%   (0)   0.0% (0)
##               PEDIATRIC CLINICS    6.0%   (54) 0.0%   (0)   0.0% (0)
##  INTERNATIONAL MEDICINE CLINIC    1.6%   (39) 0.0%   (0)   0.0% (0)
##             NEUROSURGERY CLINIC    5.2%   (38) 0.7%   (5)   0.0% (0)
##               CARDIOLOGY CLINIC    2.0%   (21) 0.0%   (0)   0.0% (0)
##               NEPHROLOGY CLINIC    0.6%   (12) 0.8%  (17)   0.0% (0)
##             OPHTHAMOLOGY CLINIC    1.5%   (10) 0.0%   (0)   0.0% (0)
##     HEMATOLOGY-ONCOLOGY CLINIC    1.2%    (7) 0.0%   (0)   0.0% (0)
##               HEPATOLOGY CLINIC    0.2%    (4) 0.5%  (11)   0.0% (0)
##        TRANSITIONAL CARE CLINIC    0.8%    (4) 0.0%   (0)   0.0% (0)
##                GERIATRIC CLINIC    0.1%    (1) 0.0%   (0)   0.4% (4)
##           ENDOCRINOLOGY CLINIC    0.0%    (0) 2.0%  (29)   0.0% (0)
##         GASTROENTEROLOGY CLINIC    0.0%    (0) 0.0%   (0)   0.0% (0)
##          LIPID DISORDERS CLINIC    0.0%    (0) 0.0%   (0)   0.0% (0)
##             RHEUMATOLOGY CLINIC    0.0%    (0) 1.5%  (36)   0.0% (0)
##                           Total   14.3% (6427) 0.9% (401)   0.0% (5)
##          Normal      On Site
##  17.7%   (621) 10.9%  (383)
##  22.2%   (694)  6.3%  (198)
##  49.8%  (1086) 16.6%  (362)
##  10.5%    (53)  0.0%    (0)
##  95.8% (11361)  0.3%   (34)
##  94.5%  (5982)  3.0%  (187)
##  81.3%   (733) 12.7%  (115)
##  92.1%  (2301)  6.4%  (159)
##  94.1%   (681)  0.0%    (0)
##  97.8%  (1003)  0.2%    (2)
##  96.2%  (2013)  2.4%   (51)
```

```
##  98.2%   (645)  0.3%    (2)
##  98.6%   (572)  0.2%    (1)
##  99.3%  (2195)  0.0%    (1)
##  93.4%   (470)  5.8%   (29)
##  97.4%   (989)  2.1%   (21)
##  78.3%  (1164) 19.7%  (293)
##  99.9%   (780)  0.1%    (1)
##  99.3%   (597)  0.7%    (4)
##  98.5%  (2386)  0.0%    (0)
##  80.7% (36326)  4.1% (1843)
```

Rate of SmartSet usage:

```
orders %>%
  tabyl(department, ordering_route) %>%
  arrange(desc(`SmartSet`)) %>%
  adorn_totals("row") %>%
  adorn_percentages("row") %>%
  adorn_pct_formatting() %>%
  adorn_ns() %>%
  select(department, SmartSet)
```

```
##                      department     SmartSet
##      INFECTIOUS DISEASE CLINIC 20.8% (2472)
##       BEHAVIORAL HEALTH CLINIC 91.8%  (462)
##                   OB GYN CLINIC 11.5%  (251)
##              NEPHROLOGY CLINIC 10.5%  (220)
##            NEIGHBORHOOD CLINIC  4.8%  (149)
##       INTERNAL MEDICINE CLINIC  2.3%  (145)
##         FAMILY MEDICINE CLINIC  3.3%  (115)
##               PEDIATRIC CLINICS  7.5%   (68)
##  INTERNATIONAL MEDICINE CLINIC  2.6%   (65)
##              HEPATOLOGY CLINIC  2.1%   (47)
##        GASTROENTEROLOGY CLINIC  3.6%   (28)
##           ENDOCRINOLOGY CLINIC  1.8%   (27)
##              GERIATRIC CLINIC  2.6%   (26)
##       TRANSITIONAL CARE CLINIC  3.6%   (18)
##            OPHTHAMOLOGY CLINIC  0.6%    (4)
##            RHEUMATOLOGY CLINIC  0.2%    (4)
##              CARDIOLOGY CLINIC  0.0%    (0)
##     HEMATOLOGY-ONCOLOGY CLINIC  0.0%    (0)
##         LIPID DISORDERS CLINIC  0.0%    (0)
##            NEUROSURGERY CLINIC  0.0%    (0)
##                          Total  9.1% (4101)
```

Rates of Provider Preference List Usage:

```
orders %>%
  tabyl(department, pref_list_type) %>%
  arrange(desc(`Provider Preference List`)) %>%
  adorn_totals("row") %>%
  adorn_percentages("row") %>%
  adorn_pct_formatting() %>%
  adorn_ns()
```

```
##                     department Clinic Preference List         None
```

```
##      INFECTIOUS DISEASE CLINIC          58.7%  (6967) 25.2% (2989)
##            RHEUMATOLOGY CLINIC          27.5%   (666)  1.9%   (46)
##              HEPATOLOGY CLINIC          48.3%  (1067) 11.1%  (246)
##           ENDOCRINOLOGY CLINIC          41.3%   (613)  6.3%   (94)
##       INTERNAL MEDICINE CLINIC          85.1%  (5386)  5.3%  (335)
##         LIPID DISORDERS CLINIC          13.0%    (78)  0.8%    (5)
##              NEPHROLOGY CLINIC          61.0%  (1276) 21.9%  (459)
##  INTERNATIONAL MEDICINE CLINIC          82.0%  (2048)  4.7%  (118)
##            NEIGHBORHOOD CLINIC          77.7%  (2429) 14.5%  (452)
##            NEUROSURGERY CLINIC          74.3%   (538)  1.8%   (13)
##               GERIATRIC CLINIC          73.4%   (745)  9.8%   (99)
##              CARDIOLOGY CLINIC          86.5%   (888)  3.2%   (33)
##              PEDIATRIC CLINICS          81.4%   (734)  9.1%   (82)
##        GASTROENTEROLOGY CLINIC          79.1%   (618) 11.4%   (89)
##       TRANSITIONAL CARE CLINIC          79.7%   (401)  7.0%   (35)
##            OPHTHAMOLOGY CLINIC          92.1%   (605)  3.0%   (20)
##     HEMATOLOGY-ONCOLOGY CLINIC          88.8%   (515)  8.4%   (49)
##                  OB GYN CLINIC          79.6%  (1734) 19.7%  (430)
##         FAMILY MEDICINE CLINIC          89.5%  (3135) 10.4%  (365)
##       BEHAVIORAL HEALTH CLINIC           8.0%    (40) 92.0%  (463)
##                           Total          67.7% (30483) 14.3% (6422)
##  Provider Preference List
##              16.1% (1905)
##              70.6% (1710)
##              40.6%  (898)
##              52.4%  (779)
##               9.6%  (609)
##              86.2%  (518)
##              17.1%  (358)
##              13.3%  (333)
##               7.9%  (247)
##              23.9%  (173)
##              16.8%  (171)
##              10.2%  (105)
##               9.5%   (86)
##               9.5%   (74)
##              13.3%   (67)
##               4.9%   (32)
##               2.8%   (16)
##               0.7%   (15)
##               0.0%    (1)
##               0.0%    (0)
##              18.0% (8097)
```

End Exercise

Evaluating turnaround times for result review

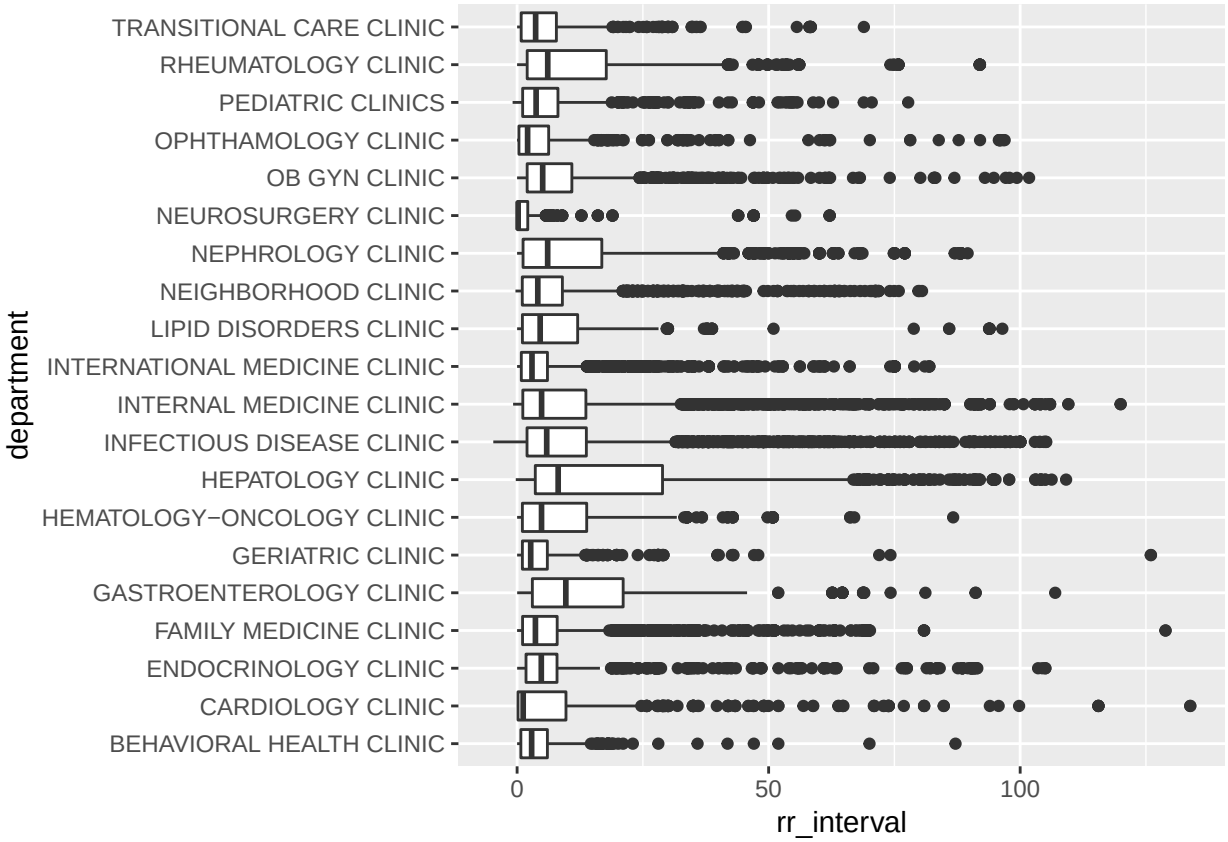
Unfortunately this data set is missing crucial timestamps needed to assess lab turnaround times. Assessing the time between order and result might be interesting, but there are various workflow variations that make this difficult to interpret. What is more straightforward to interpret, however, is the 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.

```
orders <- orders %>%
  mutate(rr_interval = as.numeric(difftime(review_time, result_time, units = "days")))
ggplot(orders, aes(x = department, y = rr_interval)) +
  geom_boxplot() +
  coord_flip()
```

```
## Warning: Removed 7791 rows containing non-finite values (stat_boxplot).
```



End Exercise

Bringing it all together from import to graph to result

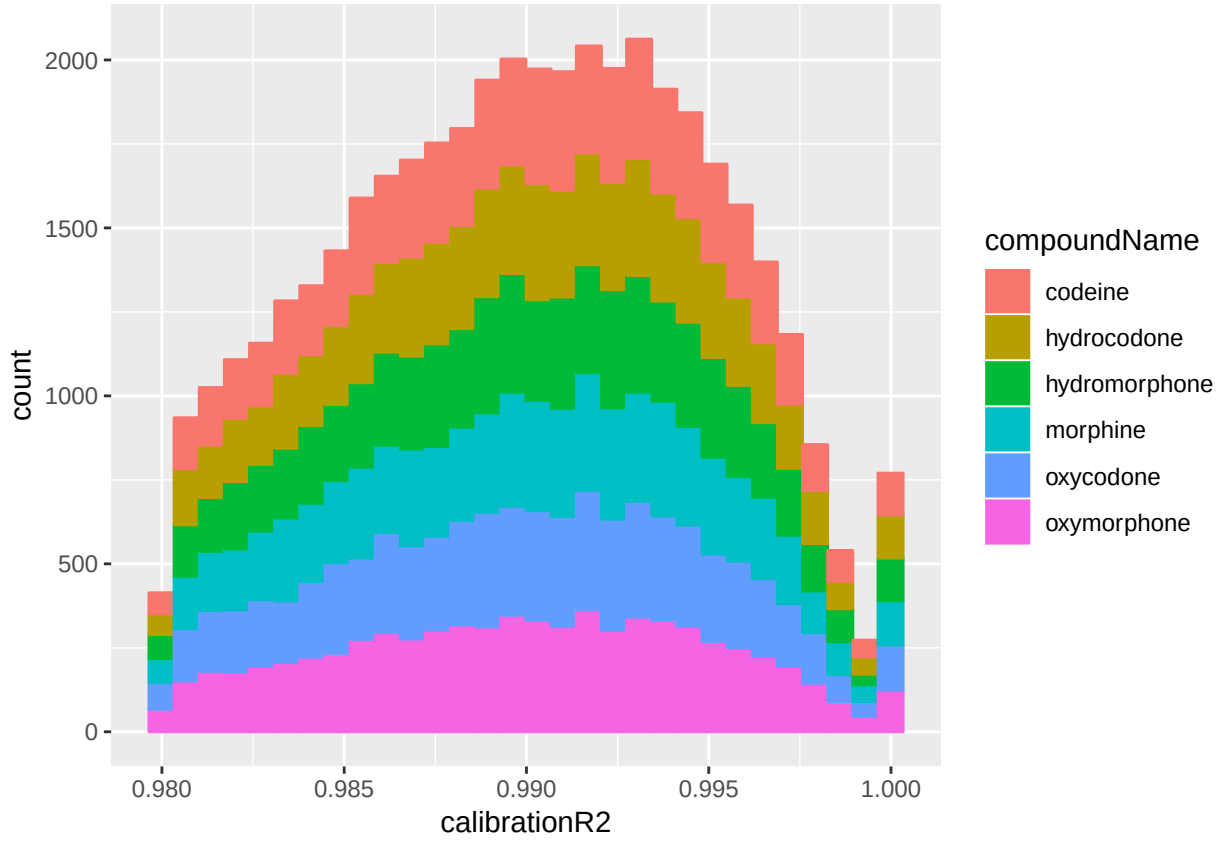
From Import to Graph

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

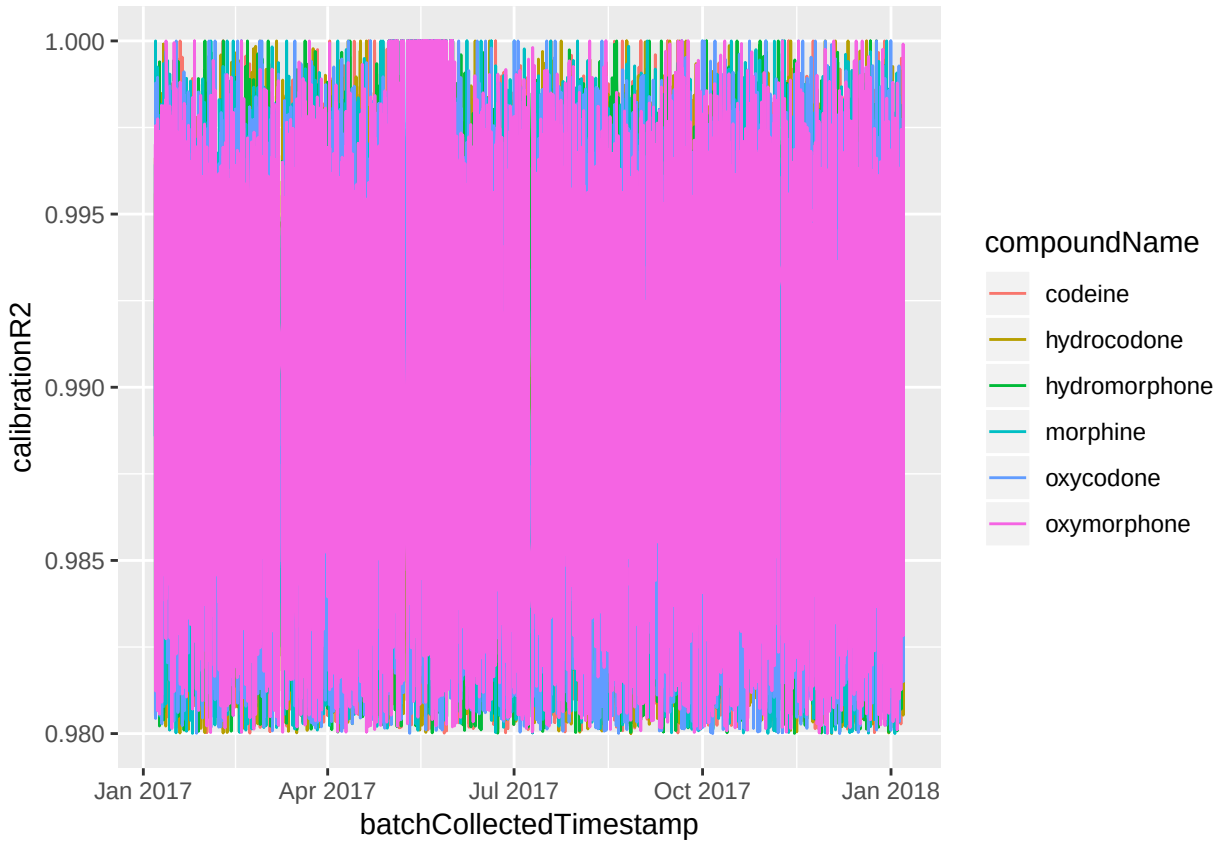
```
oneYearBatches <- list.files("data/", pattern = "_b.csv$") %>%
  file.path("data", .) %>%
  map_dfr(read_csv) %>%
```

```
  as_tibble() %>%
  type_convert()
```

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

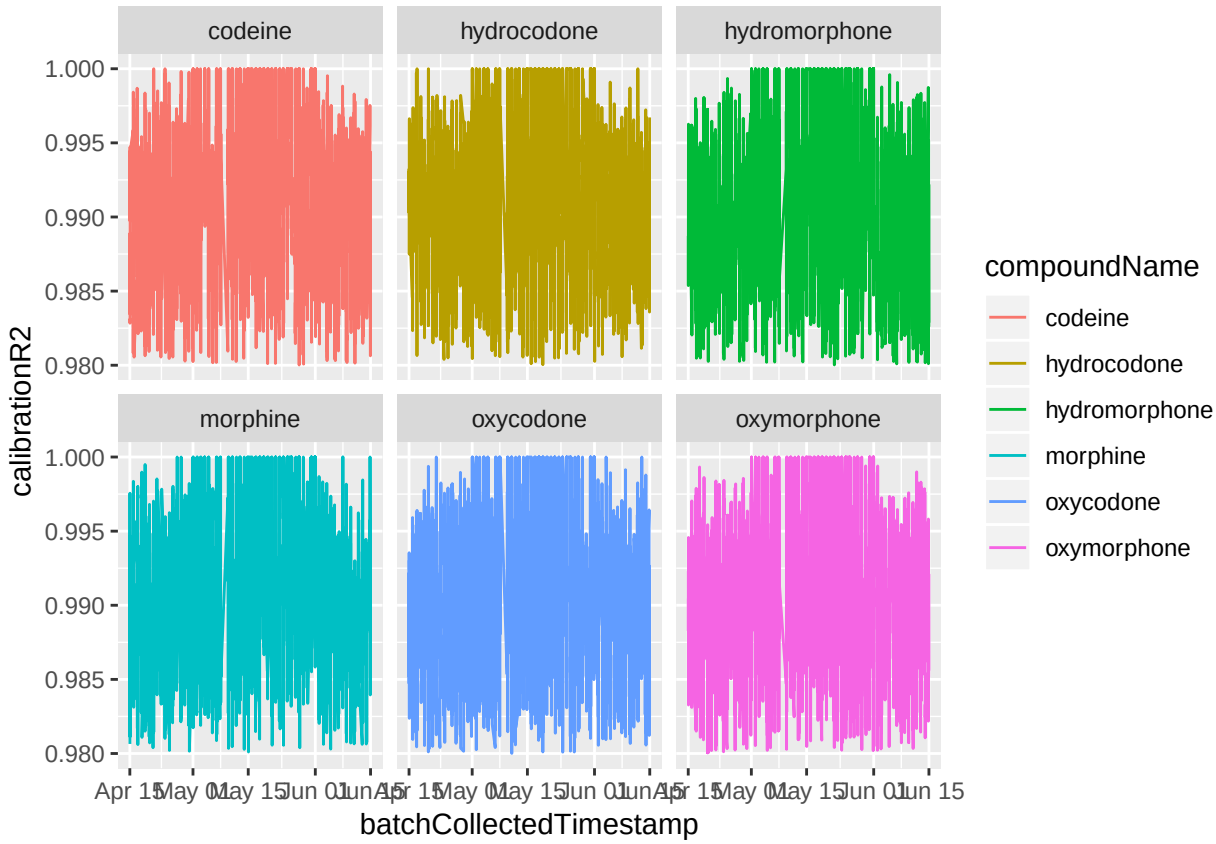



```
ggplot(oneYearBatches, aes(x = batchCollectedTimestamp, y = calibrationR2, color = compoundName)) +
  geom_line()
```

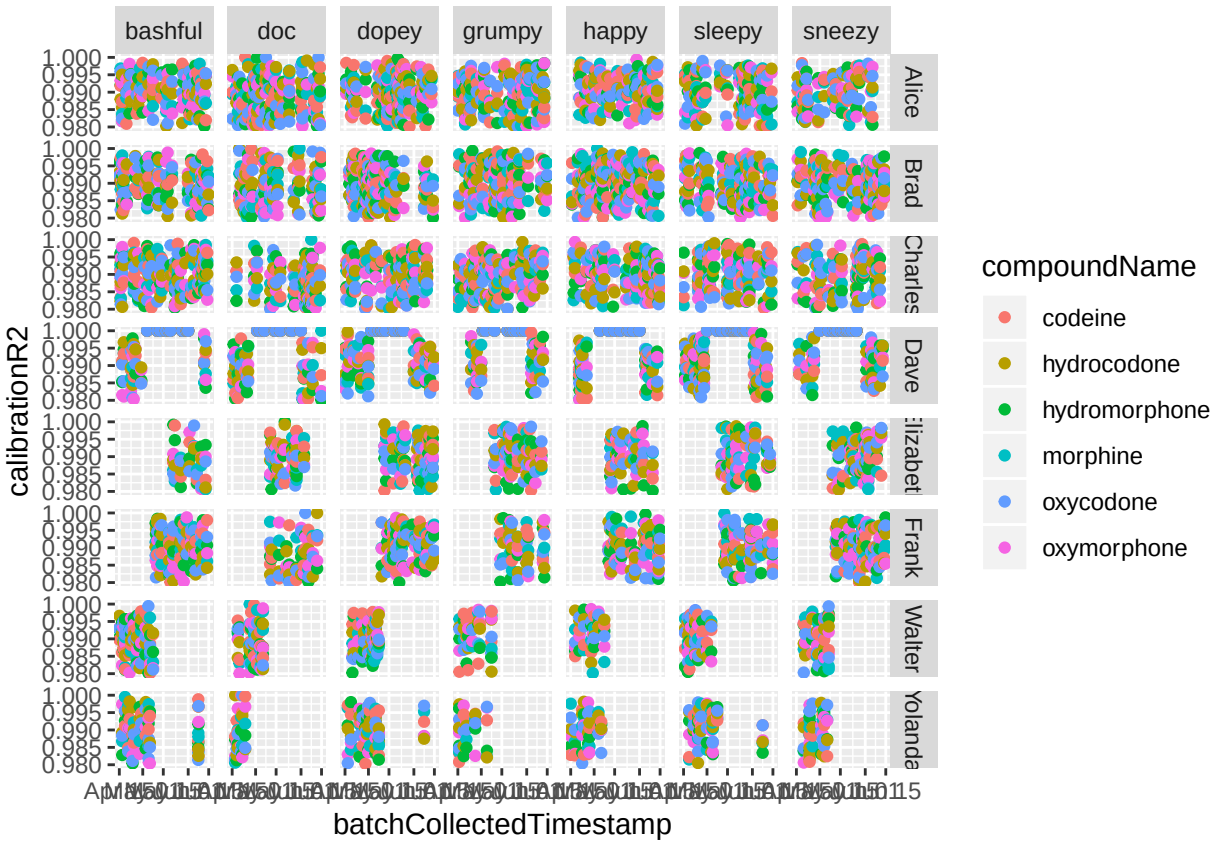


There's something interesting going on with the R^2 values in the month of May, where a large number of them report a value of 1.0 – a perfect fit. Let's focus on that month, and spread out the data so we can clarify whether it's all compounds or just oxymorphone (the magenta color on top).

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




```
mayPlot +
  geom_point() +
  facet_grid(reviewerName ~ instrumentName)
```



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

From Graph to Result

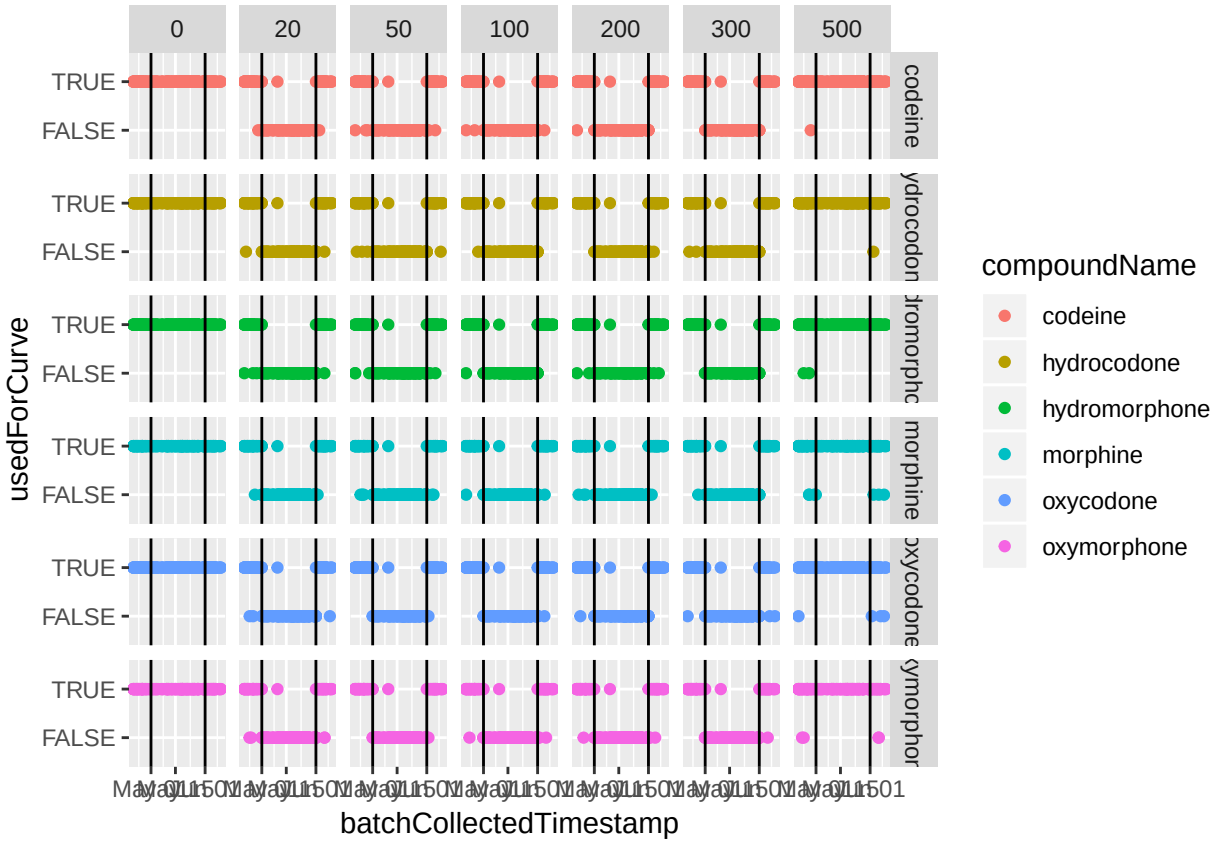
Based on the batch-level data, we can see that 'Dave' – and apparently only Dave – has perfect R^2 values on every batch of data he reviewed throughout the month of May. Digging deeper will require merging information from the batch level with information at the sample (and possibly peak) level.

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

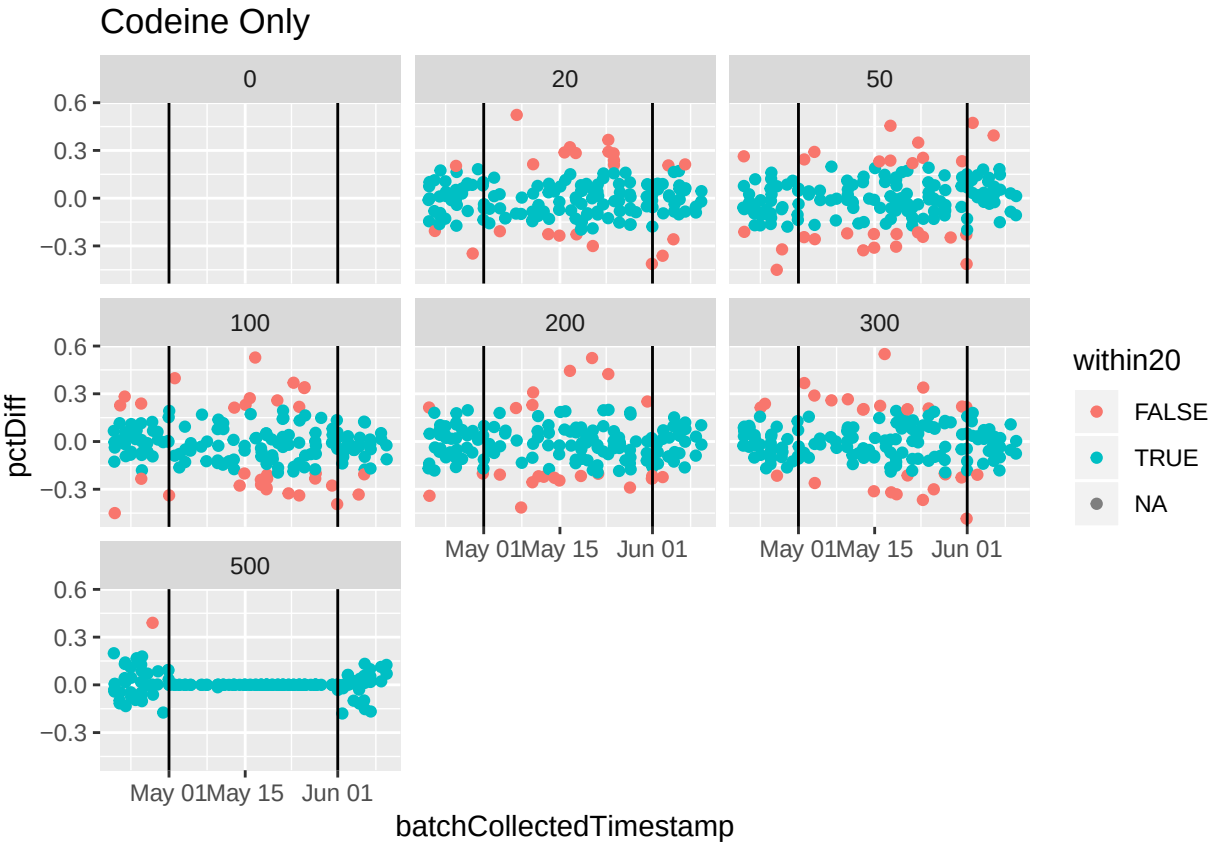
The following plots of `davesData` provide compelling evidence for what happened: Dave unselected the middle five calibrators in order to draw a straight line and maximize the R^2 term.

```
davesData %>%
  ggplot(aes(x = batchCollectedTimestamp, y = usedForCurve, color = compoundName)) +
  geom_point() +
  facet_grid(compoundName ~ expectedConcentration) +
```

```
  geom_vline(xintercept = as.numeric(as_datetime(c("2017-05-01", "2017-06-01"))),
    linetype = 1,
    colour = "black")
```




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



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

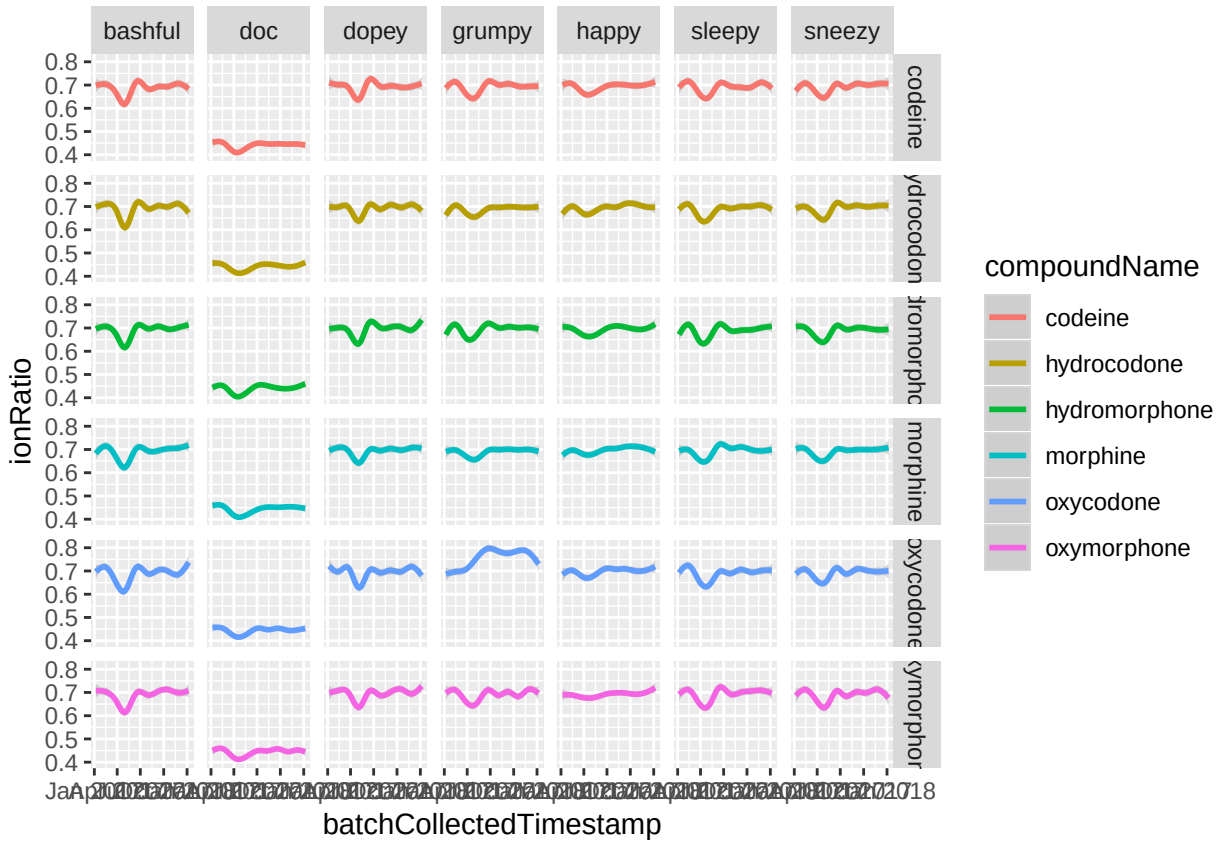
We intentionally included several other issues within the database, which will require aggregation and plotting to discover.

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

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

```
## Joining, by = c("batchName", "compoundName")
```

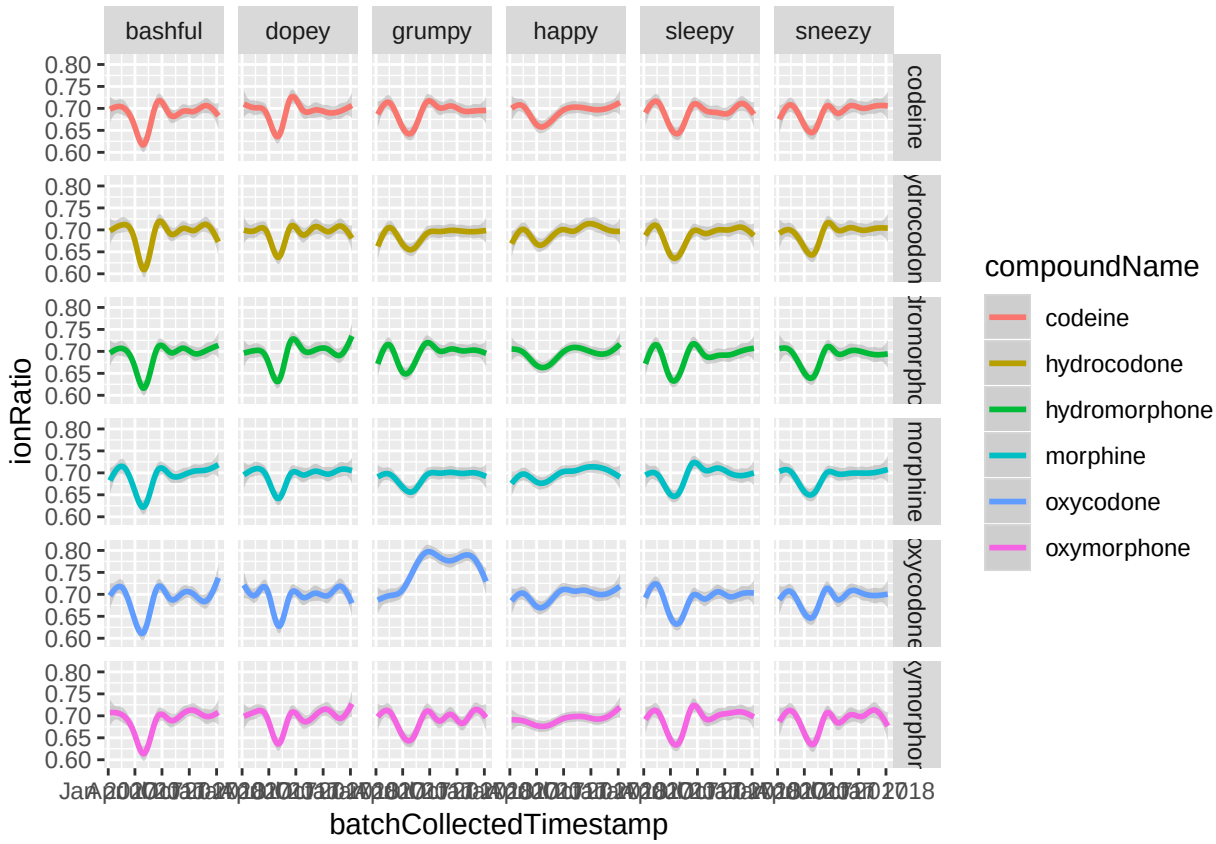
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```




```
# investigate later
# Quants and quals got flipped

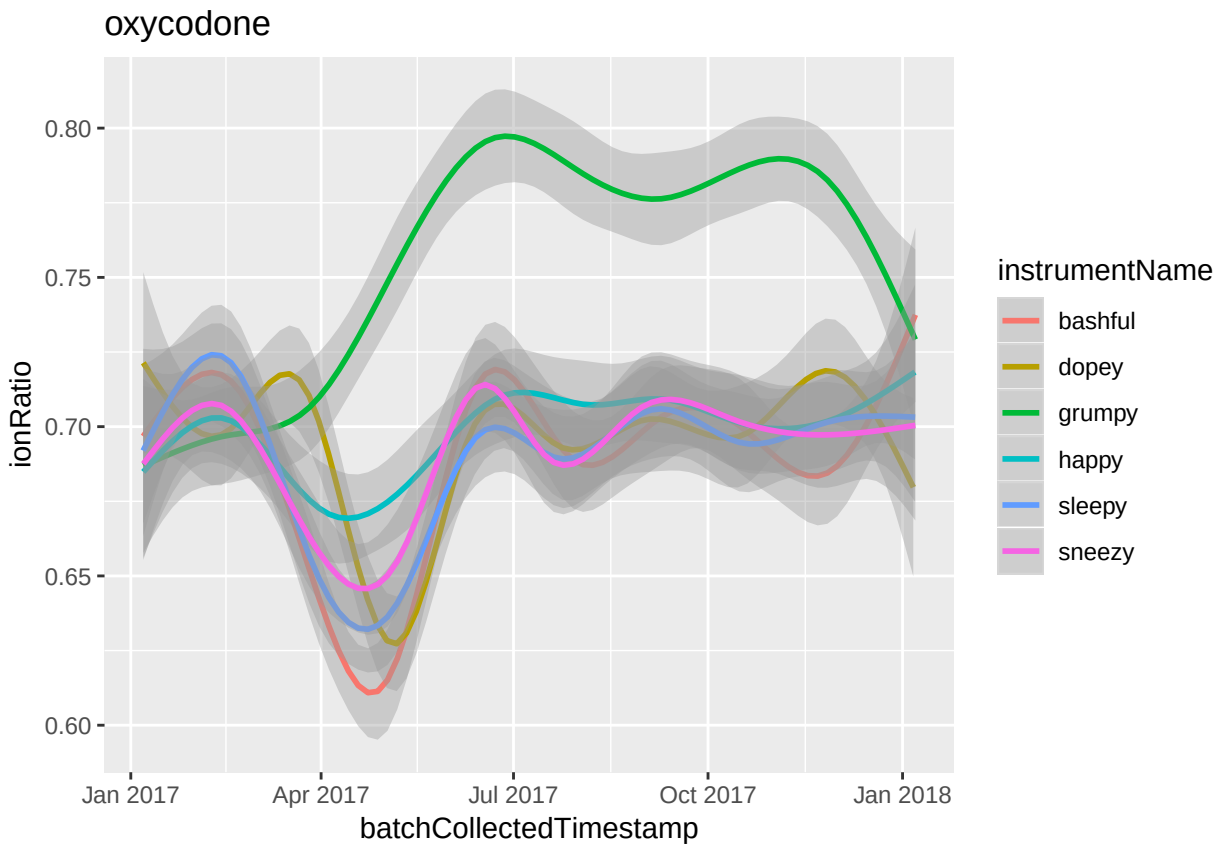
# 2 #
oneYearSamples %>%
  left_join(oneYearBatches) %>%
  filter(instrumentName != "doc") %>%
  ggplot(aes(x = batchCollectedTimestamp, y = ionRatio, color = compoundName)) +
  geom_smooth() +
  facet_grid(compoundName ~ instrumentName) # grumpy+oxycodone looks least like the others
```

```
## Joining, by = c("batchName", "compoundName")
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



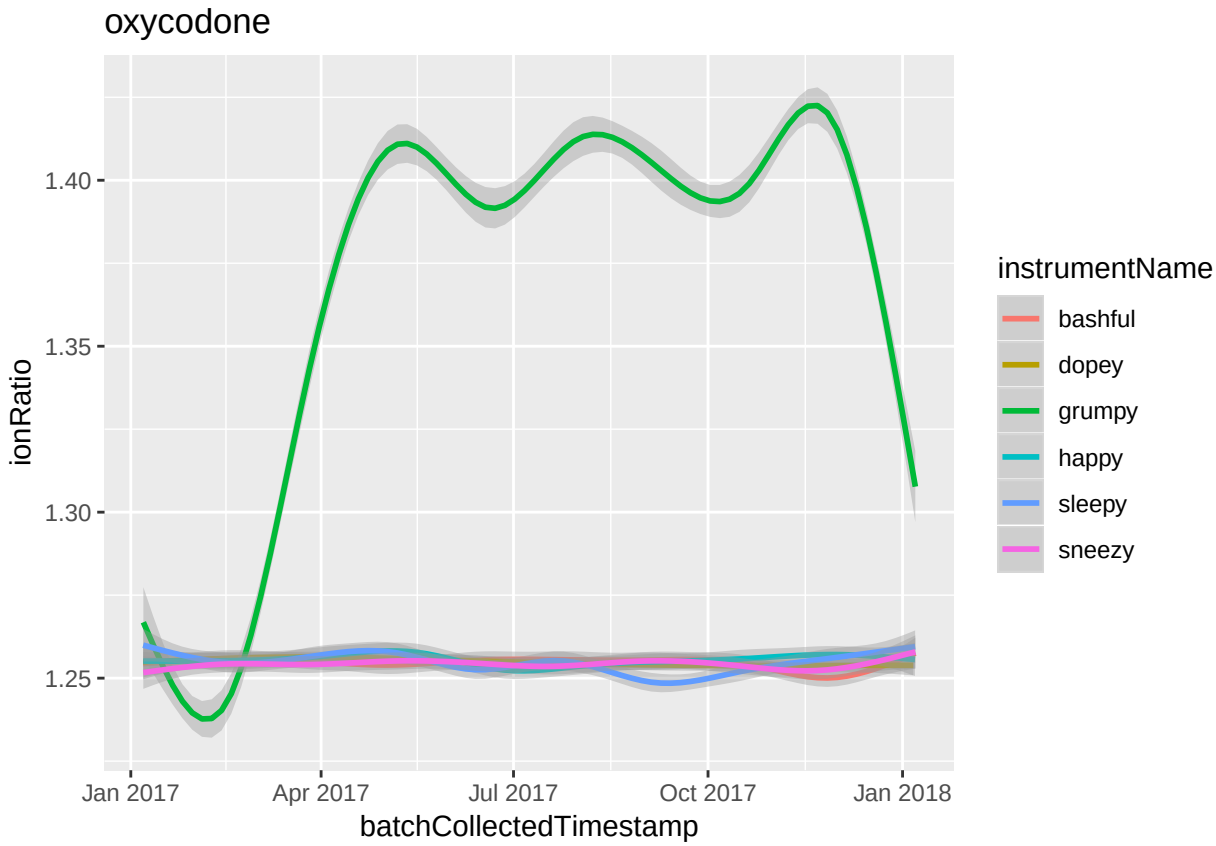

```
# 3 #
oneYearSamples %>%
  left_join(oneYearBatches) %>%
  filter(compoundName == "oxycodone" & instrumentName != "doc") %>%
  ggplot(aes(x = batchCollectedTimestamp, y = ionRatio, color = instrumentName)) +
  ggtitle("oxycodone") +
  geom_smooth() # grumpy+oxycodone clearly outlying
```

```
## Joining, by = c("batchName", "compoundName")
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```




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

```
## Joining, by = c("batchName", "compoundName")
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



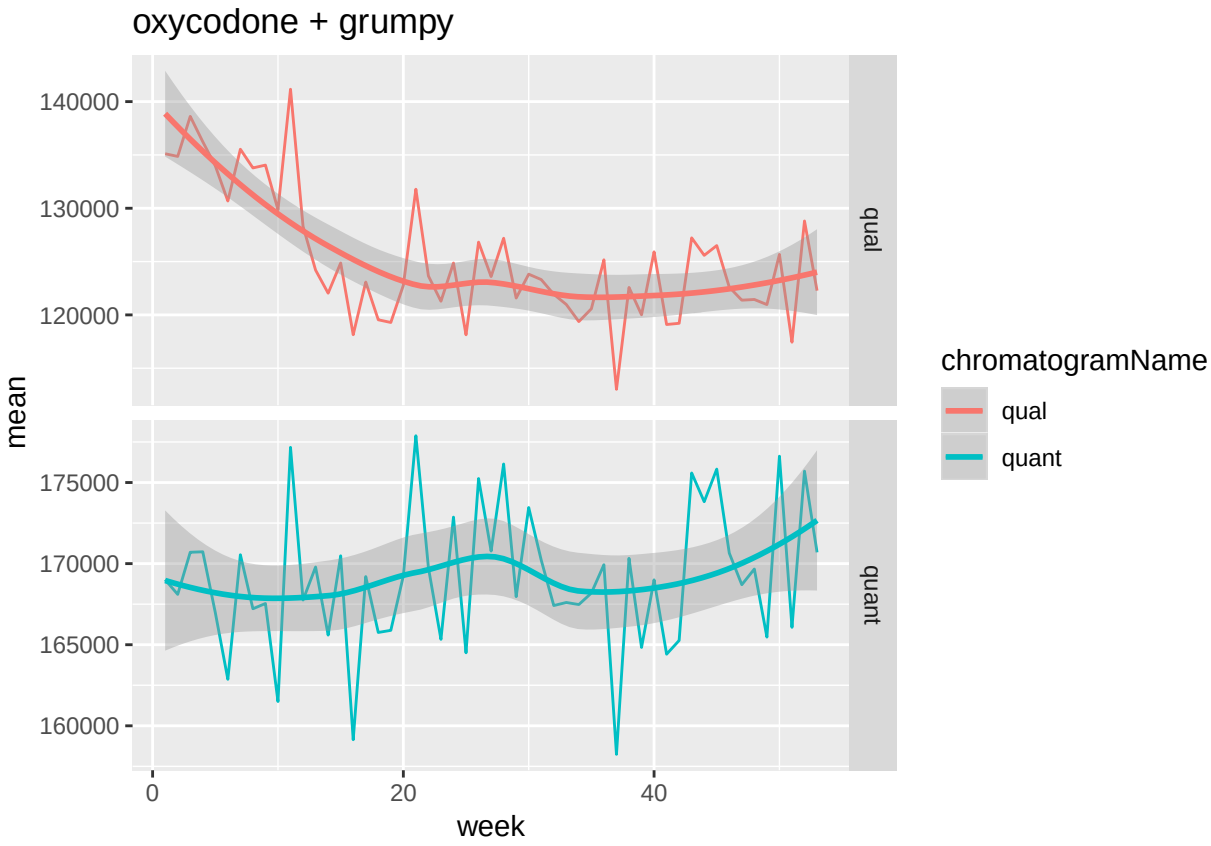

```
# 5 #
oneYearPeaks <- list.files("data/", pattern = "_p.csv$") %>%
  file.path("data", .) %>%
  map_dfr(read_csv, col_types = cols())

meanByWeek <- oneYearPeaks %>%
  left_join(oneYearBatches) %>%
  filter(compoundName == "oxycodone" & instrumentName == "grumpy" & peakArea > 0) %>%
  mutate(week = week(batchCollectedTimestamp)) %>%
  group_by(week, chromatogramName) %>%
  summarise(mean = mean(peakArea), sd = sd(peakArea), n = n())
```

```
## Joining, by = c("batchName", "compoundName")
```

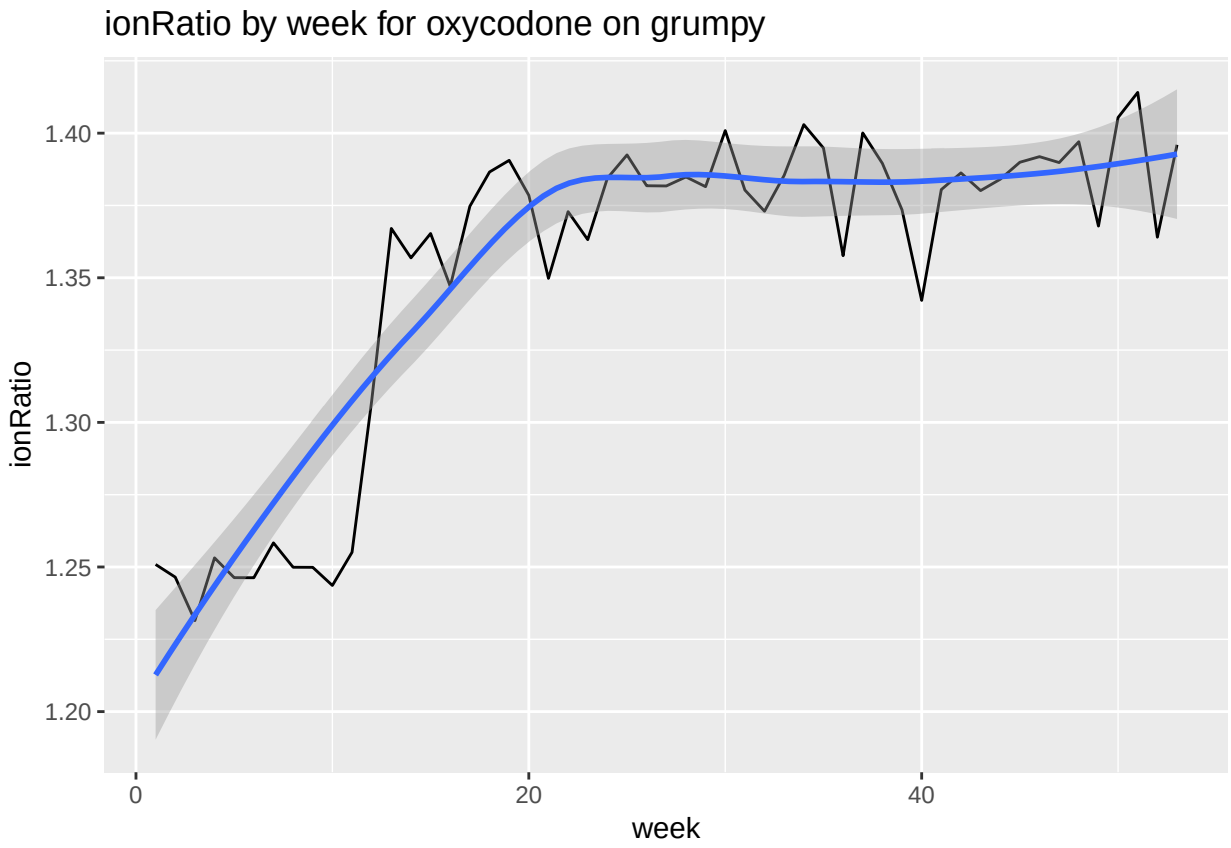
```
ggplot(meanByWeek, aes(x = week, y = mean, color = chromatogramName)) +
  geom_line() +
  geom_smooth() +
  ggtitle("oxycodone + grumpy") +
  facet_grid(chromatogramName ~ ., scales = "free_y") # quant is constant, qual drops
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




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

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



End of Exercise