# Introductory R Course: How to Truly Excel at Data Analysis and Visualization

*Joe Rudolf*
*Daniel Herman*
*Niklas Krumm*
*Patrick Mathias*

## Contents

**1 Introduction to R** **2**
1.1 What is R? . . . 2
1.2 Installing R (and some other stuff to make R work) . . . 2
1.3 Configuring the RStudio Environment . . . 4
1.4 Basics of coding . . . 5
1.5 Creating reproducible data analyis . . . 7
1.6 Learning to Learn to Code . . . 10

**2 Method Validation in R: Reference Ranges** **11**
2.1 Overview . . . 11
2.2 Load data . . . 11
2.3 Viewing data tables . . . 13
2.4 Examine data distribution . . . 15
2.5 Is it normally distributed? . . . 16
2.6 Establishing a reference range . . . 19
2.7 How to verify a proposed reference range? . . . 24
2.8 Advanced: Using a Q-Q plot to examine a distribution . . . 26

**3 Method Validation in R: Method Comparison** **27**
3.1 Overview . . . 27
3.2 Load data . . . 27
3.3 Describe data and explore its distribution . . . 27
3.4 Overlapping histograms . . . 28
3.5 Method comparison (t-tests, and more) . . . 31
3.6 Regression . . . 32
3.7 Deming regression . . . 35
3.8 Passing-Bablock . . . 37
3.9 Compare methods by concordance relative to decision thresholds . . . 39

**4 Method Validation: Precision, Linearity, and Calibration Verification** **40**
4.1 Overview . . . 40
4.2 Precision . . . 41
4.3 Calibration Verification . . . 48
4.4 Linearity (optional) . . . 53

**5 Exploratory Data Analysis: Orienting Yourself with Your Data Set** **55**
5.1 Sequencing functions . . . 55
5.2 Loading data and reviewing data types . . . 56
5.3 Reshaping rectangular data . . . 59
5.4 Tabulating our data . . . 64
5.5 Simple summary visualizations . . . 68

**6 Exploratory Data Analysis: Plotting Time Data and Joining Data Sets** **75**
6.1 Plotting Time Points for Data Exploration . . . 76
6.2 Exploring Time Intervals and Plotting the Data . . . 78
6.3 Counting Tests Per Patient . . . 81
6.4 Date Differential by Patient and Test . . . 83
6.5 Bringing different data sets together: . . . 85

**7 Next Steps for Learning R** **86**
7.1 Online Books and Quick Reference Resources . . . 86
7.2 Courses and Other Active Learning Resources . . . 86

**8 Acknowledgements** **87**

# 1 Introduction to R

In this lesson we will cover some R basics including installing/configuring R, a quick introduction to coding, and the fundamentals of making our data analysis reporducible. This content can be a little dry but provides the necessary building blocks for the more interesting content to come. Hang in there and we will get to importing, manipulating, and visualizing real laboratory data ASAP!

## 1.1 What is R?

R is a statistical programming language. Using R you can load, analyze, and visualize data. R has the same functionality you are used to in other spreadsheet applications (e.g. Microsoft Excel) but offers greater functionality and flexibility.

R also provides an environment in which we can conduct reproducible data analysis. R affords us the capabilities to document our data analysis and save it in a way that we can revisit our analysis at a later date, update our approach when inspiration strikes, and share our analysis with friends.

## 1.2 Installing R (and some other stuff to make R work)

### 1.2.1 R

R is the programming language that we will be using to conduct our data analysis today.

To install R, go to the Comprehensive R Archive Network (CRAN). Select the version of R for your operating system (Windows, Mac OS, Linux)

### 1.2.2 RStudio

R studio is the development environment in which we will do our R programming.

Download and install RStudio. Select the installer for your Operating System.

When you launch RStudio you will be presented with with a series of panes. Let's start by getting oriented to the RStudio application.

In general, the left side of the screen is used to write and execute code, and the right side of the screen provides information about the environment you are working with. The output of your code typically appears on the left side, but can show up on the right, if you are creating a plot from the Console.

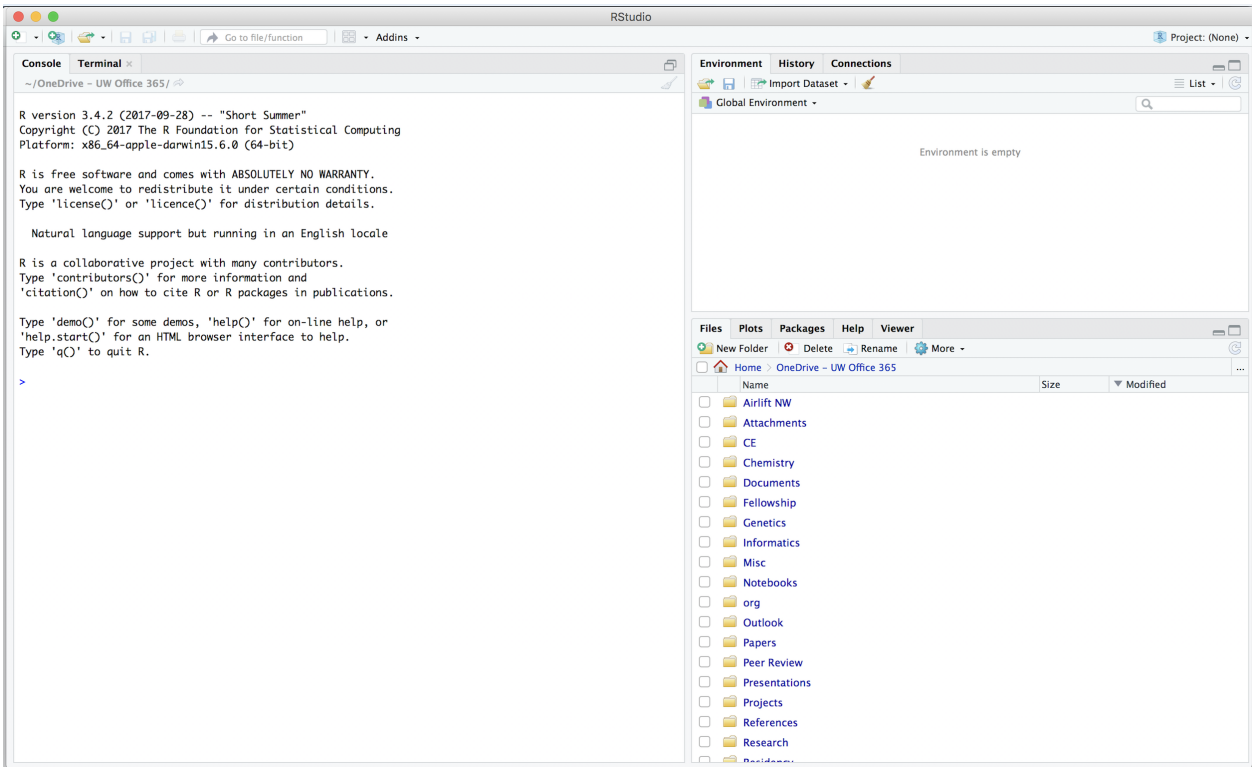



Figure 1: Screen shot of RStudio

When you first open R, on the left you will see the **Console** tab automatically open and fill the entire left side of the screen. The console is one place to enter and execute our R code.

By default the right side is split into an upper and lower pane, and each pane contains multiple tabs. If you select the **Files** tab on the lower right pane, you will see the file hierarchy for whatever the default working directory R is using. The **Plots** tab shows the output of the last plot that was generated using code in the console, and the **Packages** tab shows the list of packages that are installed on your computer (and therefore available to load for your computing session).

The **Environment** tab in the pane in the upper right corner of the screen shows any objects that have are available in the environment. As an example, if you imported data from a spreadsheet into an object called "data", a "data" object would appear in the environment tab and you could click on it to learn more information.

Another very useful tab on the top right is the **History** tab. Here we can see the historical record of the code we have executed in the console.

### 1.2.3 The Tidyverse

To enhance the functionality of R, we need to install additional packages. A package is a collection of functions that extend the capabilities of the base R programming language.

In this course we will be primarily using the tidyverse package. The tidyverse includes functions for reading data into the R environment, cleaning and manipulating data, and plotting our results.

Installing the tidyverse package is as easy as entering the following line of code in the RStudio console (the box on the left hand side of RStudio) and pressing the return (enter) key.

```
install.packages("tidyverse", dependencies = TRUE)
```

## 1.3 Configuring the RStudio Environment

### 1.3.1 Setting the Working Directory

Let's create a folder on our desktop called "R_intro" in which to save our work today. Now let's establish our R_intro folder as our working directory.

A working directory is where R will read our data from and save our work to. Let's set our working directory to to the R_intro folder in RStudio (Session -> Set Working Directory -> Choose Directory).

We can check to see that we have set the correct working directory by running the following code in the console.

```
getwd()
```

For the course we will want to set our working directory as the repo folder that you downloaded from git hub. Change the working directory to the folder that you downloaded from the github repo. Check the new working directory is correct using the `getwd` function.

**End Pre-class Homework: Once you have reached this point you are ready for class. Please stop here for now.**

### 1.3.2 Loading the tidyverse

We have previously installed the tidyverse package but to use the package in our analysis we need to load it during the session we intend to use the functions in.

Let's load the tidyverse with the following code.

```
suppressMessages(library(tidyverse))
```

**Exercise 1:**

There are a large number of packages that you can load to help accomplish a task or solve a problem. There is a repository called CRAN that hosts many useful packages: CRAN. Clicking on a package name will show you some basic information about the package, and many packages have supporting documentation and vignettes that walk you through how to use the package's functions.

Let's go ahead and practice loading another package we'll encounter later in the course: janitor.

1. Install the janitor package. There are a couple options to be familiar with:
    a. Use the `install.packages("")` function. Be sure to include the name of the package in quotes.
    b. Navigate to the Tools menu and select "Install Packages...". You will see a prompt that allows you to specify the names of one or more packages to install.
2. Load the janitor package to use during this session, similarly to how you loaded tidyverse before this exercise. Step 1 gets the package on your computer for future use; step 2 allows you to use the functions in the packages you load during that specific R session. If you exit R without saving you will have to reload the package to use its functions the next time you open up R.
3. Read about the janitor package by searching through all documentation using the command `??janitor`. You can find help materials for function or package that is already loaded into your current session of R using a single question mark. If you want to find information for a package or function that is not currently loaded in the environment, you can search all of the help documentation using two question marks.

**End exercise**

## 1.4 Basics of coding

In its most basic form R is a calculator

```
2 + 3 + 2
```

```
[1] 7
```

R supports a number of functions (e.g. the absolute value function) that can extend our calculator. We pass arguments to the function (e.g. data or other parameters) and the function will give us back a result.

For example:

```
abs(-77)
```

```
[1] 77
```

There are functions to support nearly all of the data analysis you may want to do. And remember you can always add more functions by installing packages that other users have developed and published. You can even write your own functions, though this won't be necessary for any of the analysis we perform in this introductory course.

We can also create objects and use them in our calculations.

To do this we use the following syntax (grammar):

object_name <- contents of our object

The <- performs an action called "assignment" and can be referred to as "object_name gets contents of our object". If you're not a fan of typing those two characters, RStudio has a keyboard shortcut: Alt - (hold down the Alt key and press the minus sign).

As an easy example, let's assign a new object "x" a value of 1.

```
x <- 1
```

Take a look at the Environment tab on the upper right. You can now see your object in that window.

**Exercise 2:**

1. Now let's assign that same object x another value (eg. 2).

```
x <- 2
```

Review your Environment tab. What happened to your object?

2. Next let's assign a new object a with the contents of our object x.

```
a <- x
```

3. Re-assign x to its original value 1.

```
x <- 1
```

Review the Environment tab. What happened to a? Was that the behavior you expected?

```
a
```

```
[1] 2
```

```
x
```

```
[1] 1
```

**End Exercise**

Now let's try something a little more sophisticated. Instead of putting a single number into our object, let's put a series of numbers into it using the `c()` function, which combines values into a vector.

```
x <- c(1, 2, 3, 4, 5)

x
```

```
[1] 1 2 3 4 5
```

The `c()` function is helpful for creating a vector on the fly, but for a longer vector one may not want to manually type in a bunch of values. There are other functions that can help with creating a vector such as `seq()`, which creates a series of numbers. Let's create an object by assigning a function and perform multiple functions on our new object.

```
# Assign the object
sequence_one_to_twenty <- seq(1, 20)

# Check the contents of the object
sequence_one_to_twenty
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
# Calculate the mean of the sequence
mean(sequence_one_to_twenty)
```

```
[1] 10.5
```

```
# Calculate the max of the sequence
max(sequence_one_to_twenty)
```

```
[1] 20
```

Remember to give your objects descriptive names so you can easily remember the data they contain.

To this point we've only created objects that have a single variable (or column). What if we want to create an object that has multiple rows and columns, like an Excel spreadsheet? Traditionally this type of data is called a data frame in R, but we will use a specific flavor of data frame called the tibble. Let's create a simple tibble by creating an object with strings y and combining that with x:

```
y <- c("a", "b", "c", "d", "e")

xy <- tibble(x, y)

xy
```

```
# A tibble: 5 x 2
      x y
  <dbl> <chr>
1    1. a
2    2. b
3    3. c
4    4. d
5    5. e
```

Review the Environment tab and press the button next to the xy object to see more details. We have created a tibble with two variables (columns) and five observations (rows), with one variable being number and the other a character.

Clinical laboratory data is commonly arranged in a spreadsheet format, where each row represents a single

observation or data collection point and each column represents a different measurement or attribute for the data set. The tidyverse set of packages in R is fine-tuned to work with data in this format and can help avoid some of the painful reformatting people often do in Excel to try and get Excel to work better with the data. In this course, most of the objects we will work with will have a tibble data structure, largely because that is the bulk of what we see in the real world. In general, keep in mind that the general structure for these data sets is that each row is some observation, measurement, or single time point and each column is a distinct attribute or measurement for the observation.

**Tip**: It is good practice to separate words in your object and variable names with underscores instead of spaces, because, like many other programming languages, whitespace is important in R. If you absolutely want to use whitespace in a variable name, you need to use the ` character (top left on your keyboard) on either side of the name.

For more inofrmation on the tibble data structure, refer to the Tibbles section in the R for Data Science book. For a more detailed discussion of objects and other gory details about how the R language works, refer to the R language definition.

## 1.5 Creating reproducible data analyis

Entering code in the console is a great way to develop and iterate your analysis. However, it is easy to lose track of the pieces of code we have executed in the console. To truly make our code reproducible, we need to place it in a container that we can edit, save, and execute sequentially. R Markdown files (extension .RMD) provide an authoring environment for us to do just that!

### 1.5.1 Creating an R Markdown File

Let's create a simple R Markdown file together.

Go to the menu bar at the top of the screen (File -> New File -> R Markdown).

Let's get oriented to the R Markdown environment.

1. Header information (author, title, date, output format) is a the top of the document surrounded by "—".
2. R code is surrounded by "“‘" and will execute when the document is compiled using the knit button. Instead of typing out those characters every time, you can insert a code chunk using keyboard short cuts: Ctrl+Alt+i for Windows and Command+Option+i for Mac.
3. Commentary can be added to the document and formatted with simple formatting options (# for a title heading)

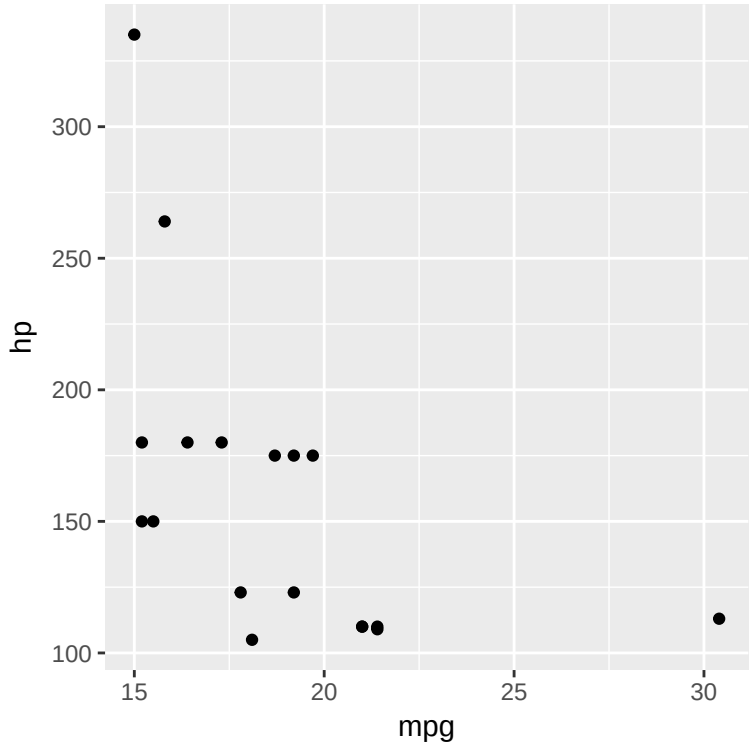
Copy and paste the following code (including the code chunk delimiters) into your R Markdown file.

```
# First let's load the tidyverse

library(tidyverse)

# Load a data set of cars into a dataframe
cars_data <- as_tibble(mtcars)

# Display table
cars_data
```

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

```
 1  21.0    6.  160.  110.  3.90  2.62  16.5    0.    1.    4.    4.
 2  21.0    6.  160.  110.  3.90  2.88  17.0    0.    1.    4.    4.
 3  22.8    4.  108.   93.  3.85  2.32  18.6    1.    1.    4.    1.
 4  21.4    6.  258.  110.  3.08  3.22  19.4    1.    0.    3.    1.
 5  18.7    8.  360.  175.  3.15  3.44  17.0    0.    0.    3.    2.
 6  18.1    6.  225.  105.  2.76  3.46  20.2    1.    0.    3.    1.
 7  14.3    8.  360.  245.  3.21  3.57  15.8    0.    0.    3.    4.
 8  24.4    4.  147.   62.  3.69  3.19  20.0    1.    0.    4.    2.
 9  22.8    4.  141.   95.  3.92  3.15  22.9    1.    0.    4.    2.
10  19.2    6.  168.  123.  3.92  3.44  18.3    1.    0.    4.    4.
# ... with 22 more rows
```

```
# Fiter the list to cars with fuel economy >= 15 mpg and horsepower >=100
mpg_hp <-filter(mtcars, mpg >= 15, hp >=100)

# Create a scatter plot of mpg vs horsepower
ggplot(data = mpg_hp) +
  geom_point(mapping = aes(x = mpg, y = hp))
```




Now let's save our R Markdown file to our working directory (File -> Save As) as "Our First R Markdown File.RMD"

### 1.5.2 Running our code

There are multiple options for running our code.

- When you are actively developing code within an R Markdown file, the quickest way to run individual lines of code is to put a cursor on a line and use the keyboard shortcut to run a single line: Ctrl+Enter on Windows or Command+Enter on Mac. If there is any output, it will appear below the code chunk.
- Instead of running a single line, you can run a whole "chunk" of code - everything between when the first ``` starts and when the second ``` ends. The shortcut to run a chunk of code that your cursor is

within is Shift+Ctrl+Enter for Windows and Shift+Command+Enter for Mac.
- Alternately, if you want to run all of the code within your document (and generate the document), the Knit button at the top of the R Markdown window provides different output options. In this course we will focus on generating an html file. This can be opened by most modern browsers. You can also generate pdf and docx files but those have some external dependencies that are not as consistently across all computers.

Since all of the code is complete, let's Knit to an html file.

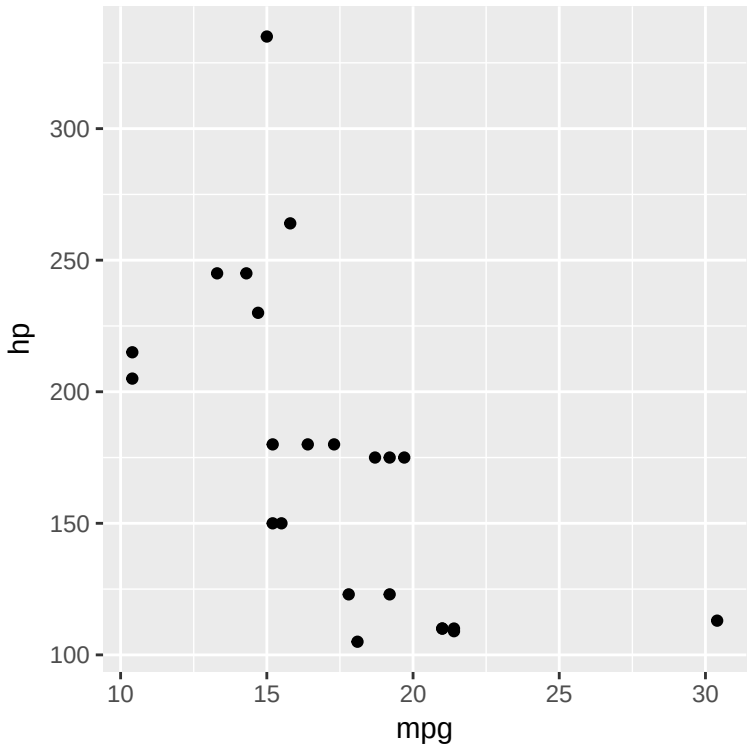
**Exercise 3:**

What if we wanted to look at mpg >= 10 and re-run the analysis? Review the code and figure out where the restriction on the mpg variable was made, and then modify the code to select fuel economy greater than or equal to 10, instead of 15. Knit the file again and observe the change in output.

```
# First let's load the tidyverse

library(tidyverse)

# Load a data set of cars into a dataframe
cars_data <- as_tibble(mtcars)

# Display table
cars_data
```

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21.0    6.  160.  110.  3.90  2.62  16.5    0.    1.    4.    4.
 2  21.0    6.  160.  110.  3.90  2.88  17.0    0.    1.    4.    4.
 3  22.8    4.  108.   93.  3.85  2.32  18.6    1.    1.    4.    1.
 4  21.4    6.  258.  110.  3.08  3.22  19.4    1.    0.    3.    1.
 5  18.7    8.  360.  175.  3.15  3.44  17.0    0.    0.    3.    2.
 6  18.1    6.  225.  105.  2.76  3.46  20.2    1.    0.    3.    1.
 7  14.3    8.  360.  245.  3.21  3.57  15.8    0.    0.    3.    4.
 8  24.4    4.  147.   62.  3.69  3.19  20.0    1.    0.    4.    2.
 9  22.8    4.  141.   95.  3.92  3.15  22.9    1.    0.    4.    2.
10  19.2    6.  168.  123.  3.92  3.44  18.3    1.    0.    4.    4.
# ... with 22 more rows
```

```
# Fiter the list to cars with fuel economy >= 10 mpg and horsepower >=100
mpg_hp <-filter(mtcars, mpg >= 10, hp >=100)

# Create a scatter plot of mpg vs horsepower
ggplot(data = mpg_hp) +
  geom_point(mapping = aes(x = mpg, y = hp))
```




**End Exercise**

R Markdown is a wonderful way to document our analysis and ensure its reproducibility. We will plan to use R Markdown throughout the course and recommend that you incorporate R Markdown documentation as your preferred authoring tool. Going forward, the easiest way to interact with the lessons is to open up the R Markdown file for the lesson (within the course folder that you download) within R Studio, follow along by executing code chunks, and then filling in any blanks for exercises.

### 1.5.3 Sharing our code

Documents can be knitted as html, PDF, or word files. These documents can be shared with your collaborators so others can see your results. Additionally you can share the R Markdown file itself (along with any data files used for your analysis) so that others can run (and tweak) your analysis.

## 1.6 Learning to Learn to Code

Learning to write code (including the functions available to you and the appropriate syntax to make your code work) can be frustrating! Learning a new language takes time so be patient with yourself. We all go through this process and your intution improves with your experience.

Inevitably, you will at times get stuck (especially when you are first starting). However, there are some great resources to get you un-stuck and part of learning to code is learning how to leverage the wealth of knowledge of others.

### 1.6.1 Help Command

If you know (or have an idea of a function) you would like to use, you can use the help command to find out more. Just type the "?" before the function of interest.

```
?abs()
```

### 1.6.2 The Internet

Chances are, if you are facing a data anlysis dilemma, many others have faced the exact same (or at least very similar) challenge. Head to your favorite search engine.

Just ask your question (making sure to include "R" in there somewhere) and you will likely find what you are looking for in the first few results

"how to mutate column in r"

Chances are one of your first hits will be Stack Overflow. Stack Overflow is a great resource for developers to ask and answer questions. If you have a question, it has likely been answered at least once on Stack Overflow.

### 1.6.3 Cheat Sheets

The folks who make RStudio offer a number of really great cheat sheets on their website that you can download/print and keep next to your desk as quick references.

### 1.6.4 A Good Book

Even in the 21st century, it never hurts to have a good book by your side.

The content in this course is oriented around the tidyverse. If you would like to continue your studies leveraging the tidyverse, a great companion is R for Data Science. This book is available freely in a digital format here and print versions are available as well.

### 1.6.5 Taking a Break

Sometimes you won't be able to brute force your way through a problem and your other resources (help functions, the internet, and your reference books) will seemingly let you down. This can be very frustrating! Don't despair, walk away for a bit. Oftentimes it just takes a fresh set of eyes to reframe your question or rethink you approach, and what seemed impossible, suddenly becomes obvious.

# 2 Method Validation in R: Reference Ranges

## 2.1 Overview

In this section we will walk through examples of how to use R to analyze standard method validation experiments, beginning with reference range evaluation.

We will use real data acquired as part of a method transition for immunoassays (unconjugated estriol, alpha fetoprotein, and inhibin-A) used for prenatal risk assessment for Trisomy 21, Trisomy 18, and open neural tube defects.

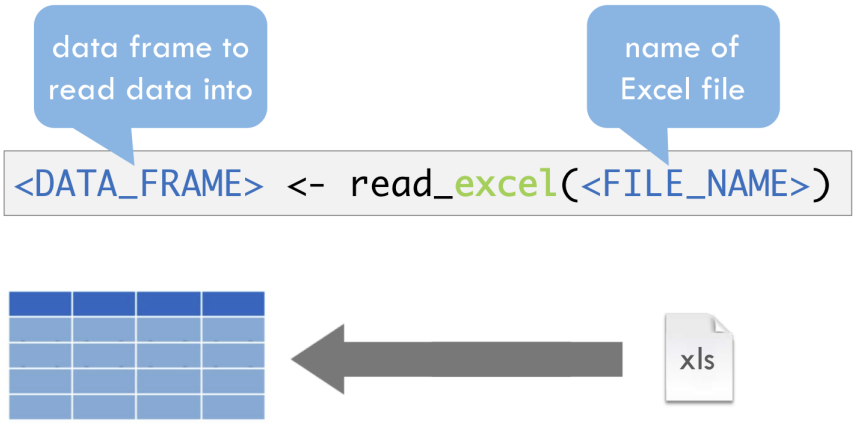
Relatively "clean" data has already been organized in separate tabs in an excel document.

## 2.2 Load data

First, let's look at the data in our spreadsheet program

### 2.2.1 How to load data?

Here's a graphical representation of how a function such as `read_excel` can assign values to a new object:




### 2.2.2 How does read_excel work?

Click the green arrow to execute this code chunk:

```
?read_excel
```

We see that there are three different functions grouped together, a `read_excel` function that is a wrapper for two more specific functions `read_xls` and `read_xlsx`. Since our Excel document is in .xlsx format, we can either use the general `read_excel` or `read_xlsx` function. We need to specify the path to our excel file and the worksheet we would like to load in. Here is an example:

### 2.2.3 Example: Load Method Evaluation data

Let's first load one of these data sets. To do this we will use the `read_excel` function. We need to specify the path to our excel file and the worksheet we would like to load in. Below is an example, which does the following:

1. R uses the `read_excel` function to open a excel spreadsheet located on the `path` "data/Method_Validation.data.xlsx".
2. We specify a `sheet`, which corresponds to the worksheet name (the 'tabs' in the lower left when you open an excel sheet).
3. The result is *assigned* via the `<-` operation (also known as the assignment operator) to the object to the left of the assignement operator.

```
uE3 <- read_excel(path="data/Method_Validation.data.xlsx",
                  sheet="MS uE3")
uE3  # Take a peak
```

```
# A tibble: 146 x 3
   specimen method_a method_b
   <chr>       <dbl>    <dbl>
 1 C1          1.52     1.56
 2 C2          0.809    1.37
 3 C3          1.11     1.43
 4 C4          0.440    0.471
 5 C5          0.350    0.508
```

```
 6 C6         0.0600    0.109
 7 C7         1.47      1.76
 8 C8         1.95      2.37
 9 C9         1.43      1.60
10 C10        0.650     0.739
# ... with 136 more rows
```

**Exercise 1:** Load in the alpha fetoprotein (AFP) method comparison data into a tibble named `afp`

```
afp <- read_excel(path="data/Method_Validation.data.xlsx",
                  sheet="MS AFP")
afp
```

```
# A tibble: 161 x 3
   specimen method_a method_b
   <chr>       <dbl>    <dbl>
 1 B1           42.8     39.3
 2 B2           42.3     44.4
 3 B3           42.3     44.4
 4 B4           20.6     19.7
 5 B5           20.6     19.7
 6 B6           67.1     59.6
 7 B7           67.1     59.6
 8 B8           41.0     35.6
 9 B9           32.0     28.8
10 B10          23.0     21.7
# ... with 151 more rows
```

**End exercise**

What is a path?

- *Paths describe the location of a file in a filesystem (e.g., your computer, a network drive, etc.). In this case, we are telling the function* `read_excel` *the location of our file, and the name of the Excel Sheet to open.*
- *These are called "arguments" to the function. Functions "take" arguments (or alternatively functions are "passed" arguments).*

### 2.2.4 More on data importing

What about reading other sorts of data? Check out the R **cheat sheet** specifically for importing data.

Other cheat sheets exist as well, check them out here

## 2.3 Viewing data tables

### 2.3.1 QC data import

Once you have loaded some data into an R object (in this case, `afp`), you should always briefly visually and manually inspect it. Things to check for include:

- Did the variable names import properly? [Need to be able to refer to each variable]
- Did variables get split appropriately [Do we have the right number of columns?]
- Did any lines (observations) get skipped? Did the entire file get loaded?

### 2.3.2 Discuss the tbl data structure

Let's next take a look at the data we have now saved as `afp`. We can do this in a few different ways.

```
afp
```

```
# A tibble: 161 x 3
   specimen method_a method_b
   <chr>       <dbl>    <dbl>
 1 B1           42.8     39.3
 2 B2           42.3     44.4
 3 B3           42.3     44.4
 4 B4           20.6     19.7
 5 B5           20.6     19.7
 6 B6           67.1     59.6
 7 B7           67.1     59.6
 8 B8           41.0     35.6
 9 B9           32.0     28.8
10 B10          23.0     21.7
# ... with 151 more rows
```

Note that it looks very similar to its original form in Excel. There are 3 columns and 161 rows. Each column has a specfic data format (`chr`, `dbl`, etc.). The `read_excel` function assigned these automatically. For our purposes, it is important that our result columns are numeric (`dbl` means "double wide" and is a type of numeric data format).

There are a few ways to examine an object in R:

1. Printing the contents to the console by simply executing a line with the object's name (as above)
2. In R-Studio, objects can also be *inspected* using the "Environment" tab (usually in the top right pane).

**Exercise 2:** Click on `afp`. **End exercise**

### 2.3.3 Look at head of table

A quick way to look at only the top `n` rows is using the `head` function:

```
head(afp, n=3)     # top 3 rows
```

```
# A tibble: 3 x 3
  specimen method_a method_b
  <chr>       <dbl>    <dbl>
1 B1           42.8     39.3
2 B2           42.3     44.4
3 B3           42.3     44.4
```

- Any text following a "#" is considered a "comment" in R, and **is not part of the code**.
- You can use comments to help explain to others (or future-you) what a particular section of code is doing, or to quickly "comment out" a portion of the code that you do not want to run.

### 2.3.4 Isolate an individual variable

We can use the `$` shorthand to isolate *individual* variable. Note that this is no longer a table, but just an array of values.

```
afp$method_a
```

```
  [1]  42.80  42.30  42.30  20.60  20.60  67.10  67.10  41.00  32.00  23.00
 [11]  20.00  45.70  26.00  70.00  36.00  10.10  10.10  27.50  27.50   4.76
 [21]  62.00  56.00  36.30  36.30  49.80  75.90  39.10  47.10  27.70  22.90
 [31]  46.20  75.80  58.10  17.00  62.40  45.60  45.30  36.50  64.60  36.30
 [41]  73.90  36.70  51.80  46.00  19.20  50.70  31.50  57.70  63.50  44.00
 [51] 113.00  36.70  42.10  39.40  82.10  97.90  43.60  25.10  61.80 113.00
 [61]  24.60  21.40  51.20  43.60  16.00  40.70  49.50  25.90  33.40  31.50
 [71]  31.70  44.50  32.50  23.50  45.50  20.30  36.50  21.10  65.90  51.20
 [81]  37.90  35.50  41.60  41.70  24.30  29.30  30.50  52.60  37.90  56.00
 [91]  81.60  17.60  50.10  20.80  37.00  33.20  25.40  72.70  59.90  36.20
[101]  32.20  23.10  36.30  37.60  26.00  31.20  26.80  28.40  22.80  40.40
[111]  27.10  23.00  26.30  22.70  83.80  67.40  27.70  23.50  44.40  57.20
[121]  42.10  26.00  36.90  29.00  42.10  30.60  29.60  34.70  35.90 316.00
[131]  57.30 126.00  24.40  45.90  37.90  71.90  39.70  22.60  35.00  25.90
[141]  19.90  19.70  33.00 127.00  64.00  40.00  19.00  47.00  35.00 117.00
[151]  79.00  27.00  46.00  38.10  38.10  31.40  31.40  53.60  53.60  14.00
[161]  77.00
```

## 2.4 Examine data distribution

Let's focus on the maternal serum AFP values from Method A first.

```
mean(x = afp$method_a)  # Mean
```

```
[1] 44.14634
```

```
median(x = afp$method_a) # Median
```

```
[1] 37.6
```

```
sd(x = afp$method_a) # Standard deviation
```

```
[1] 30.69505
```

*In calling functions (i.e.* ***mean****), the values can be passed in order the functions arguments, or they can be explicitly assigned (****arg*** *=* ***value****). In general, it is good practice to specify the argument receiving each value, because this is safer and easier to follow. However, because this makes statement much longer, it is common to not explicitly list arguments for common functions with few, easily interpretable arguments. As an example, since* ***mean*** *and* ***sd*** *only take single values, we can simply write and read:*

```
mean(afp$method_a)  # Mean, shorter
```

```
[1] 44.14634
```

```
sd(afp$method_a) # Standard deviation
```

```
[1] 30.69505
```

**Exercise 3:** Using the functions `mean` and `sd`, write code that will calculate the Coefficient of Variation (CV) for the assay. Hint: The CV is defined as:

$$CV(\%) = \sigma/\mu * 100$$

```
cv <- sd(afp$method_a) / mean(afp$method_a) * 100  # Coefficient of variation
print(cv)
```

```
[1] 69.53024
```

**End exercise**

## 2.5 Is it normally distributed?

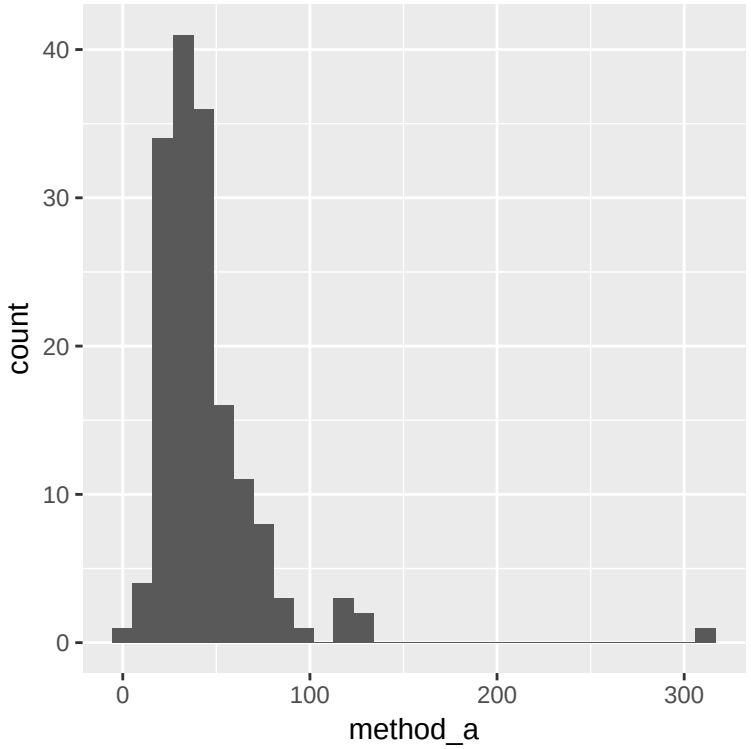
There are many approaches to determining "normality" of a distribution. Calculation of mean, median, mode and SDs can help. Visual inspection of the data will also quickly reveal if the data is normal, skewed, multimodal, etc.

> A famous example of how summary statistics can mask true differences is demonstrated by Anscombe's quartet. FJ Anscombe, an English statistician, famously said in 1973 that "Computers should make both calculations and graphs. Both sorts of output should be studied; each will contribute to understanding." An entertaining (involving T-rex) and informative visualization of this problem can be found here: https://www.autodeskresearch.com/publications/samestats

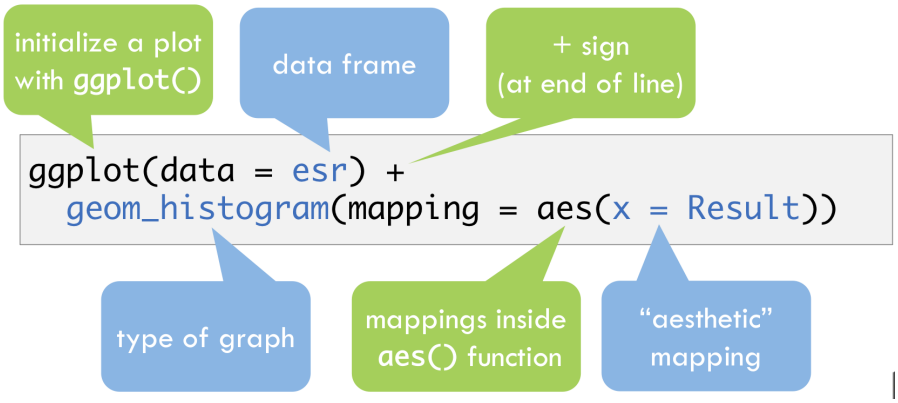
We will use the `ggplot` library to visualize data.

> The idea of ggplot is to use graphical "building blocks" and combine them to create just about any kind of graphical display you want. Here is an in-depth tutorial of ggplot.

```
ggplot(data=afp) +
  geom_histogram(aes(x=method_a))
```

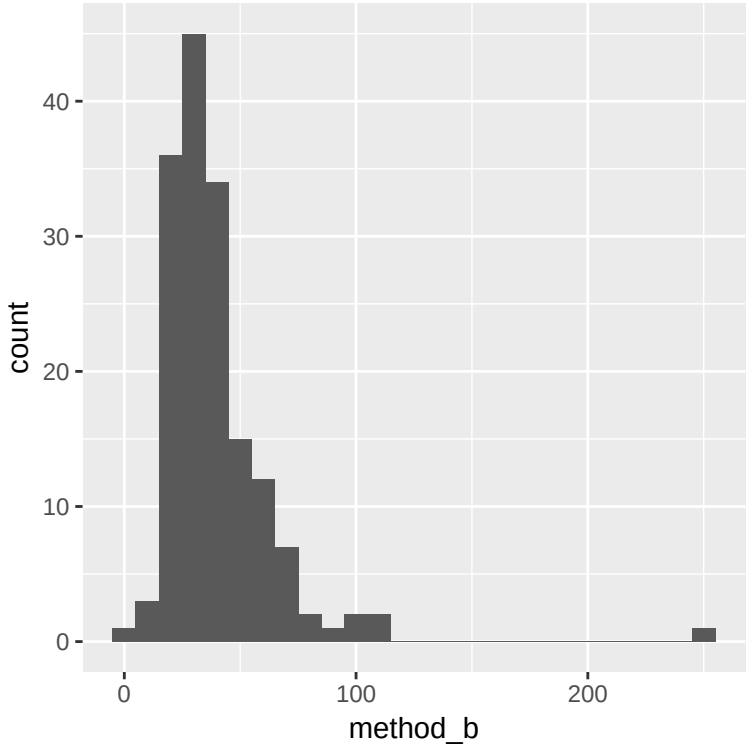



What's going on here? Take a look at this visual breakdown:




**Exercise 4:** 1. Make a second histogram plotting `method_b` values. Select reasonable `binwidth` or `bins`.

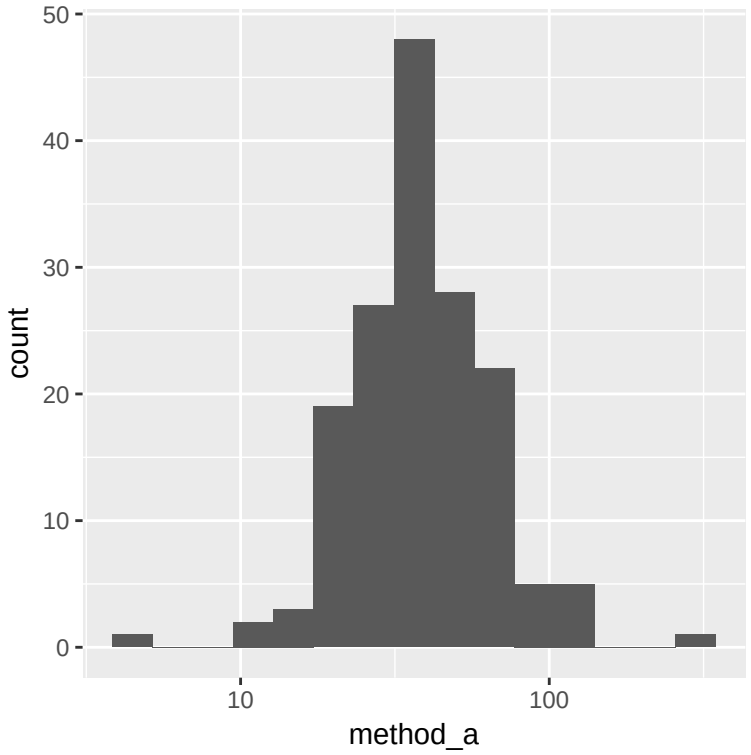
```
ggplot(data=afp) +
  geom_histogram(binwidth = 10, aes(x=method_b))
```




Does it look normal?

2. Transform the x-axis to log-scale using `scale_x_log10` and visually inspect for normality

```
ggplot(data=afp) +
  geom_histogram(bins=15, aes(x=method_a)) +
  scale_x_log10()    # Change x-axis scaling
```




**End exercise**

How can we quantify this?

The **skew** and **kurtosis** are so-called **moments** of a distribution and can be used as measure of normality (or non-normality).

```
library(moments)            # load the moments R library, which provides the functions
skewness(afp$method_a)
```

```
[1] 4.832325
```

```
kurtosis(afp$method_a)
```

```
[1] 40.14582
```

**Exercise 5:** Let's assess the `skewness` of the log-transformed method_a data. Use the `log10` function to transform results.

```
method_a_logged <- log10(afp$method_a)
skewness(method_a_logged)
```

```
[1] 0.09492914
```

**End exercise**

Another option is to use the **Shapiro-Wilk test of normality**:

```
shapiro.test(afp$method_a)
```

```
    Shapiro-Wilk normality test

data:  afp$method_a
W = 0.64436, p-value < 2.2e-16
```

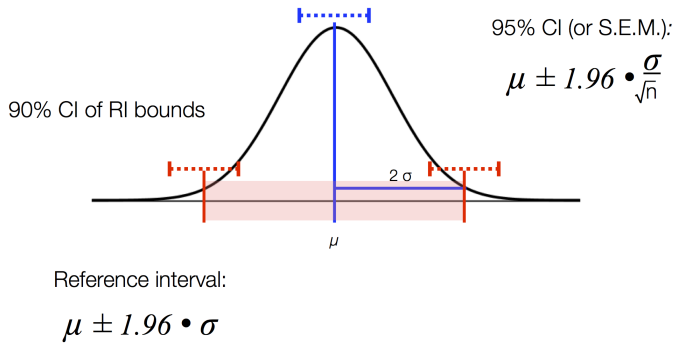
## 2.6 Establishing a reference range

### 2.6.1 Overview

Reference ranges can be established in several different ways. Two of the most common approaches are termed **parametric** and **non-parametric**.

| | Parametric | Non-parametric |
|---|---|---|
| Good for | Normal distributions | Skewed, log-normal, or other non-normal distributions |
| Assumptions | Assumes normality | No assumptions about underlying distribution |
| Center of distribution | Mean | Median |
| Advantages | More power | Less affected by outliers, Simple to calculate |
| Disadvantages | Affected by outliers + skew | Less power, requires more samples |
| CLSI Recommended Approach | No | Yes |

### 2.6.2 Parametric Reference Ranges




$$range = \mu ś 1.96 * \sigma$$

Where $\mu$ is the mean (average) of the distribution and $\sigma$ is the standard deviation.

We can calculate these for our results in `method_a` using R:

```
mu <- mean(afp$method_a)
sigma <- sd(afp$method_a)

lower_bound <- mu - 1.96 * sigma
upper_bound <- mu + 1.96 * sigma

sprintf("The reference range assuming normal distribution is (%f, %f)", lower_bound, upper_bound)
```

```
[1] "The reference range assuming normal distribution is (-16.015970, 104.308640)"
```

**Exercise 6:** Calculate the parametric reference range assuming that the results are log-normal (rather than normally distributed). Hint: First transform the data as above with the `log()` function, and then

"untransform" back into real numbers with `exp()`

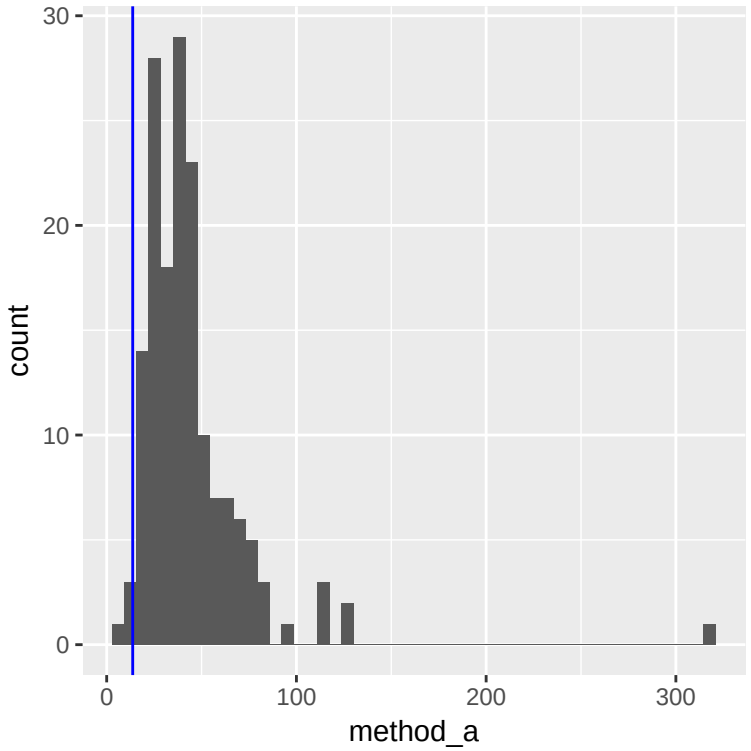
```
mu <- mean(log(afp$method_a))
sigma <- sd(log(afp$method_a))

lower_bound_transformed <- mu - 1.96 * sigma
upper_bound_transformed <- mu + 1.96 * sigma

lower_bound_untransformed <- exp(lower_bound_transformed)
upper_bound_untransformed <- exp(upper_bound_transformed)

sprintf("The reference range assuming log-normal distribution is (%3.1f, %3.1f)", lower_bound_untransfo
```

```
[1] "The reference range assuming log-normal distribution is (13.7, 106.3)"
```

**End exercise**

To visually inspect, let's graphically show the bounds of the reference intervals on the histogram we made above. A simple way to do this is to use the `geom_vline` function of ggplot. - Start with with the code for the histogram above, add additional lines of code to display the bounds of the parametric reference interval. (*Hint: Remember that ggplot allows you to "add" new elements to a plot*)

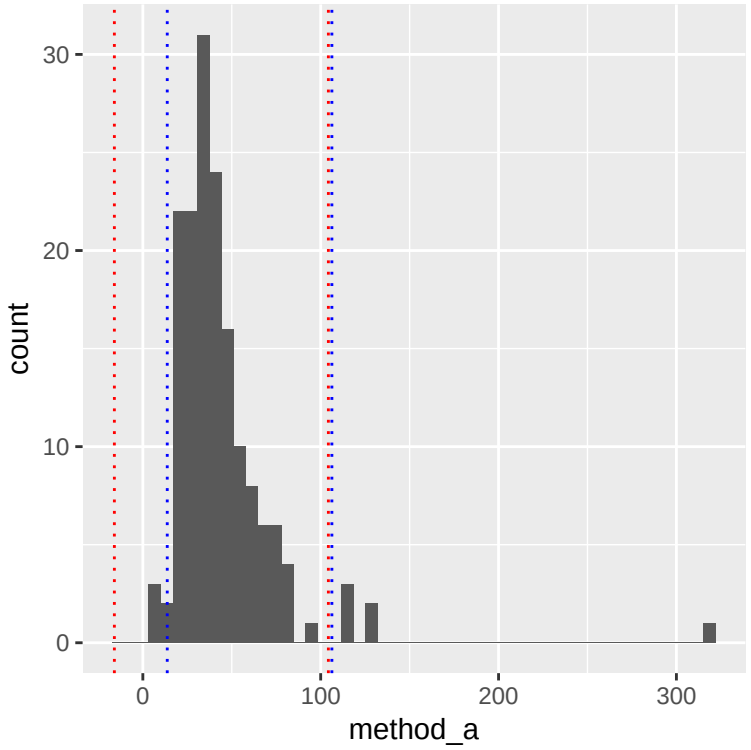
```
# Code from exercise answer above
mu <- mean(log(afp$method_a))
sigma <- sd(log(afp$method_a))
lower_bound_transformed <- mu - 1.96 * sigma
lower_bound_untransformed <- exp(lower_bound_transformed)

# new code to plot
ggplot(data=afp) +
  geom_histogram(bins=50, aes(x=method_a)) +
  geom_vline(xintercept=lower_bound_untransformed, color="blue")
```



**Exercise 7:** Add the additional 3 bounds we have just calculated to this plot. Change to a dashed line by specifying `linetype` in the `geom_vline` function.

1. Look up the function call for `geom_vline`. What kind of arguments does it take?
2. Start with with the code for the histogram above, add additional lines of code to display the bounds of the parametric reference interval. (*Hint: ggplot allows you to "add" new elements to a plot*)

```
g <- ggplot(data=afp) +
  geom_histogram(bins=50, aes(x=method_a)) +
  geom_vline(xintercept=lower_bound_untransformed, linetype=3, color="blue") +
  geom_vline(xintercept=upper_bound_untransformed, linetype=3, color="blue") +
  geom_vline(xintercept=lower_bound, linetype=3, color="red") +
  geom_vline(xintercept=upper_bound, linetype=3, color="red")
g
```




**End exercise**

### 2.6.3 Non-parametric Reference Ranges

Non-parametric reference ranges are simply the middle 95% of the distribution. Using R, this is done with the straightforward command `quantile`:

```
non_parametric_bounds <- quantile(x=afp$method_a, probs = c(0.025, 0.975))
print(non_parametric_bounds)
```

```
 2.5% 97.5%
   16   113
```

Can we verify that the `non_parametric_bounds` includes 95% of the data? Sure thing! We will do this with some new operations and a feature in R called masking. First we need to be able to refer to each bound separately. Don't want to get lost in the weeds here, but there are several different ways of extracting elements of the vector `non_parametric_bounds`

```
non_parametric_bounds
```

```
 2.5% 97.5%
   16   113
```

```
non_parametric_bounds[1]
```

```
2.5%
  16
```

```
non_parametric_bounds[[1]]
```

```
[1] 16
```

```
non_parametric_bounds[["2.5%"]]
```

```
[1] 16
```

Let's first ask which elements of `method_a` are greater than our lower-bound?

```
mask <- afp$method_a > non_parametric_bounds[[1]]
mask
```

```
  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [12]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
 [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [56]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
 [67]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [78]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [89]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[100]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[111]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[122]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[133]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[144]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[155]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

The result is a boolean (TRUE/FALSE) array. Then lets combine our upper and lower bounds

```
mask <- (afp$method_a > non_parametric_bounds[[1]]) &
                (afp$method_a < non_parametric_bounds[[2]])
mask
```

```
  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [12]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
 [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
 [56]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
 [67]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [78]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [89]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[100]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[111]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[122]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
[133]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[144] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
[155]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

This intermediate data structure `mask` is convenient, because we can easily ask how many elements it contains and how many are `TRUE`:

```
length(mask)
```

```
[1] 161
```

```
sum(mask == TRUE)
```

```
[1] 150
```

```
proportion_in_range = sum(mask == TRUE) / length(mask)
print(proportion_in_range)
```

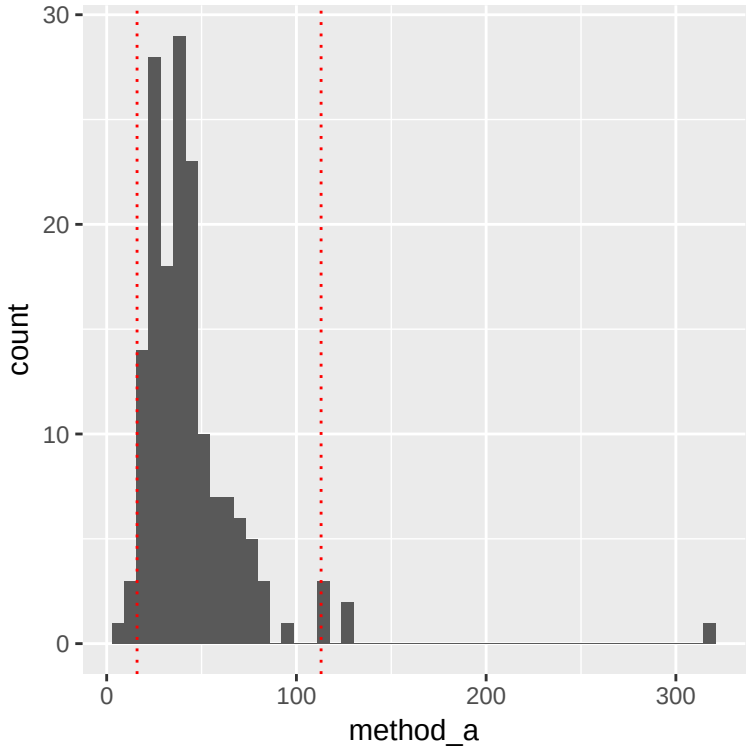
```
[1] 0.931677
```

```
#Alternative
sum(mask) / length(mask)
```

```
[1] 0.931677
```

**Exercise 8:** Plot these proposed reference intervals on histogram

```
ggplot(data=afp) +
  geom_histogram(bins=50, aes(x=method_a)) +
  geom_vline(xintercept = non_parametric_bounds["2.5%"], color="red", linetype=3) +
  geom_vline(xintercept = non_parametric_bounds["97.5%"], color="red", linetype=3)
```

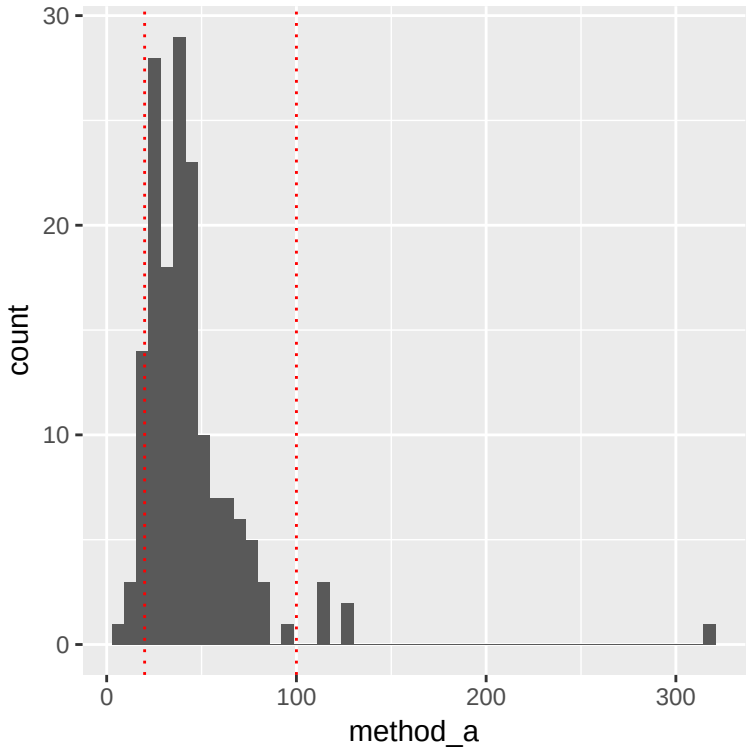



**End exercise**

## 2.7 How to verify a proposed reference range?

The lab had previously used a reference range of 20-100 for this particular test.

```
g <-  ggplot(data=afp)
g <- g + geom_histogram(bins=50, aes(x=method_a))
g <- g + geom_vline(xintercept = 20, color="red", linetype=3)
g <- g + geom_vline(xintercept = 100, color="red", linetype=3)
g
```




Is this range appropriate for our observed results?

```
n_samples <- length(afp$method_a)

old_range <- c(20, 100) #proposed range
n_inrange <- sum(afp$method_a >= old_range[1] &
                   afp$method_a <= old_range[2])

chisq.test(c(n_inrange, n_samples - n_inrange), p=c(0.95, 0.05))
```

```
    Chi-squared test for given probabilities

data:  c(n_inrange, n_samples - n_inrange)
X-squared = 10.474, df = 1, p-value = 0.00121
```

```
binom.test(x=n_inrange, n=n_samples, p = 0.95, alternative = "two.sided")
```

```
    Exact binomial test

data:  n_inrange and n_samples
```

```
number of successes = 144, number of trials = 161, p-value =
0.003319
alternative hypothesis: true probability of success is not equal to 0.95
95 percent confidence interval:
 0.8363277 0.9372757
sample estimates:
probability of success
             0.8944099
```

**Exercise 9:** Evaluate our empirically derived parametric reference ranges

```
n_samples <- nrow(afp)
ref_range <- c(lower_bound, upper_bound)
sprintf("The reference range assuming normal distribution is (%3.1f, %3.1f)",
        ref_range[1], ref_range[2])
```

```
[1] "The reference range assuming normal distribution is (-16.0, 104.3)"
```

```
n_inrange <- sum(afp$method_a >= ref_range[1] &
                   afp$method_a <= ref_range[2])
binom.test(x=n_inrange, n=n_samples, p = 0.95, alternative = "two.sided")
```

```
    Exact binomial test

data:  n_inrange and n_samples
number of successes = 155, number of trials = 161, p-value =
0.5874
alternative hypothesis: true probability of success is not equal to 0.95
95 percent confidence interval:
 0.9206528 0.9862029
sample estimates:
probability of success
             0.9627329
```

```
ref_range <- c(lower_bound_untransformed, upper_bound_untransformed)
sprintf("The reference range assuming log-normal distribution is (%3.1f, %3.1f)",
        ref_range[1], ref_range[2])
```

```
[1] "The reference range assuming log-normal distribution is (13.7, 106.3)"
```

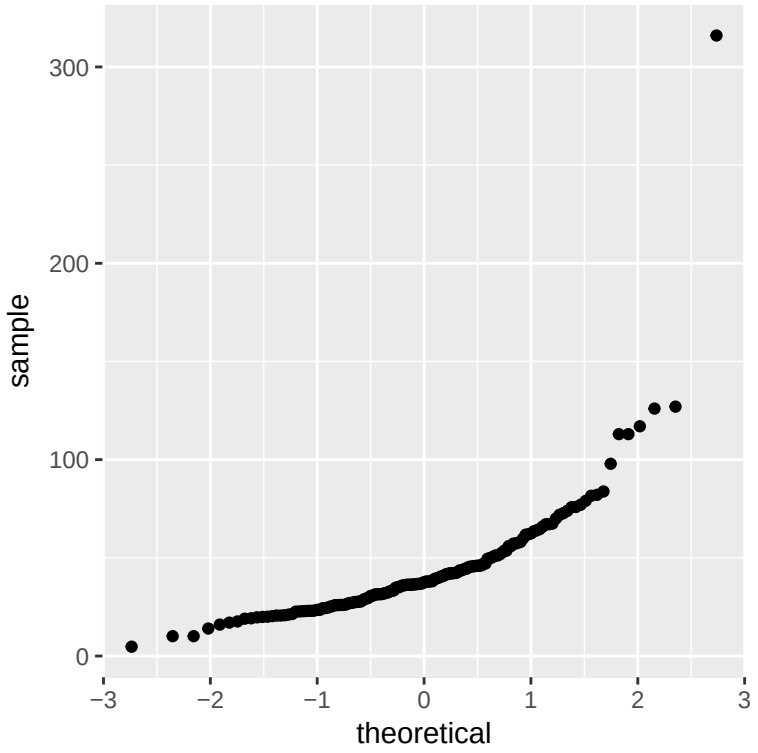
```
n_inrange <- sum(afp$method_a >= ref_range[1] &
                   afp$method_a <= ref_range[2])
binom.test(x=n_inrange, n=n_samples, p = 0.95, alternative = "two.sided")
```

```
    Exact binomial test

data:  n_inrange and n_samples
number of successes = 152, number of trials = 161, p-value =
0.7155
alternative hypothesis: true probability of success is not equal to 0.95
95 percent confidence interval:
 0.8965376 0.9741231
sample estimates:
probability of success
             0.9440994
```

**End exercise**

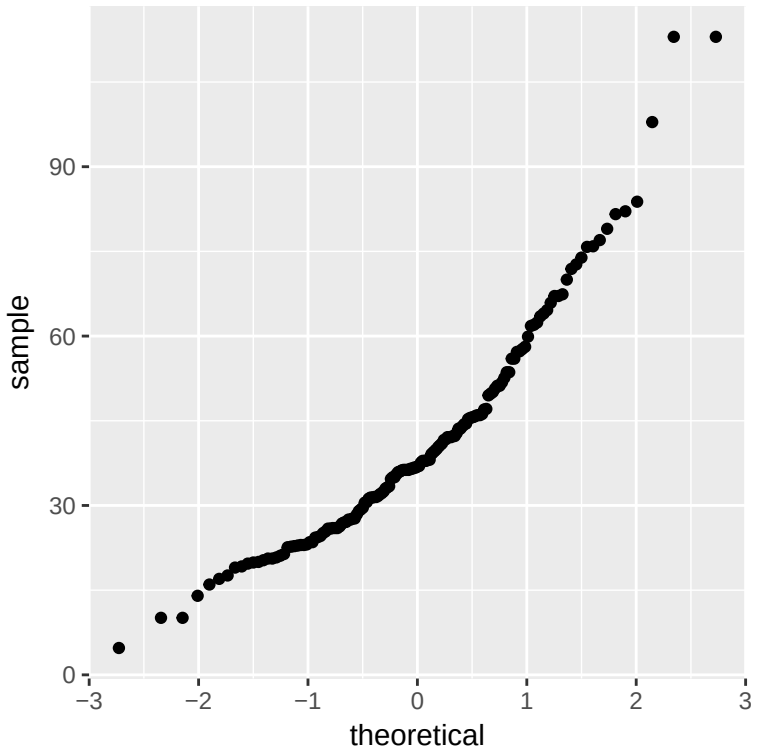
## 2.8 Advanced: Using a Q-Q plot to examine a distribution

```
ggplot(data=afp) +
  geom_qq(aes(sample=method_a))
```




Are there outliers? How can you remove them?

```
upper_outlier_bound <- mean(afp$method_a) + 3 * IQR(afp$method_a)

afp_removed_outliers <- filter(afp,
                               method_a < upper_outlier_bound)
ggplot(afp_removed_outliers) +
  geom_qq(aes(sample=method_a))
```




# 3 Method Validation in R: Method Comparison

## 3.1 Overview

In this section we will continue exploring how to use R in method validation by comparing results from two different methods.

## 3.2 Load data

Let's load in hCG data, just as we did in the previous session.

```
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,...
```

## 3.3 Describe data and explore its distribution

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

```
hcg %>%
  summarize(method_a_mean = mean(method_a),
```

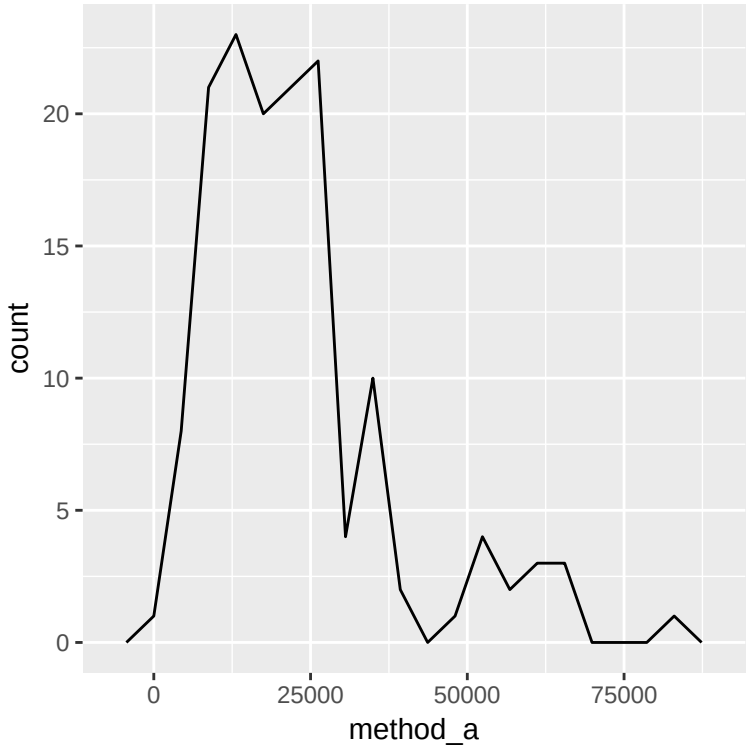
```
            method_a_sd = sd(method_a),
            method_b_mean = mean(method_b),
            method_b_sd = sd(method_b))
```

```
# 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.
```

## 3.4 Overlapping histograms

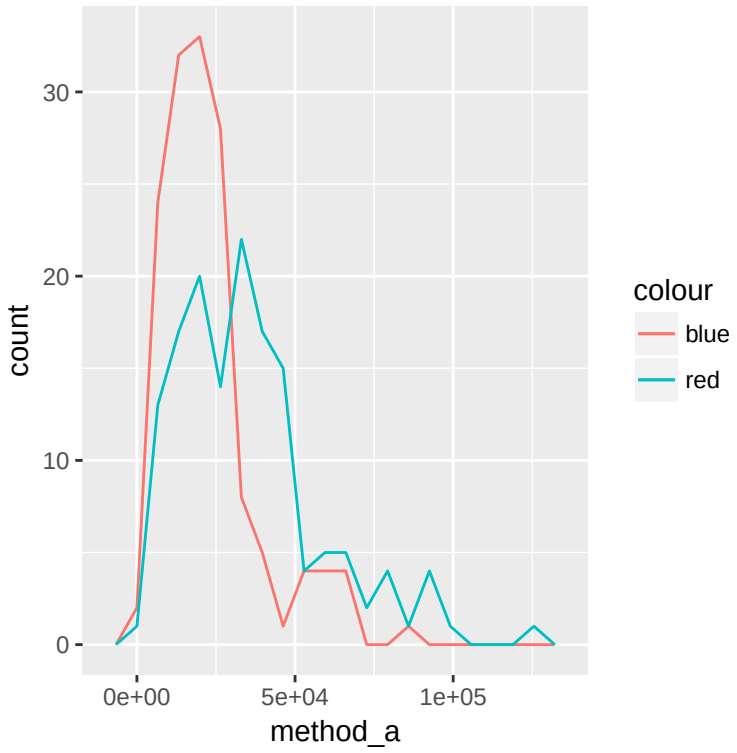
What if we want to plot the distribution of both `method_a` and `method_b` in the same plot? We've used `geom_histogram` previously. Let's try the related ggplot function `geom_freqpoly`, starting with a single method.

```
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`.

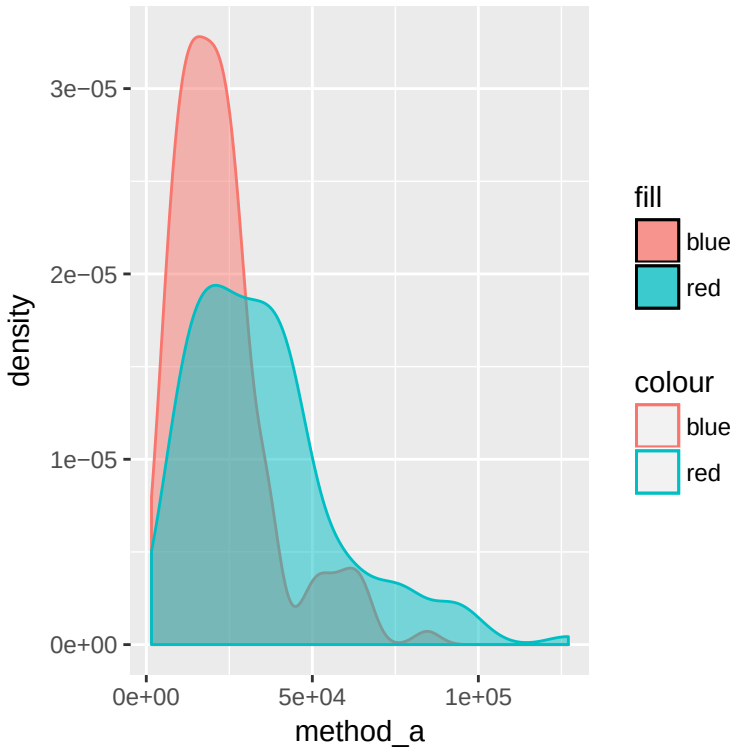
```
ggplot(data = hcg) +
  geom_freqpoly(bins=20, aes(x=method_a, color="blue")) +
  geom_freqpoly(bins=20, aes(x=method_b, color="red"))
```




**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**

### 3.4.1 (Optional) Tibble reorganization: `gather`

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

Another way is to create a "long dataframe" using the `gather` function. 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

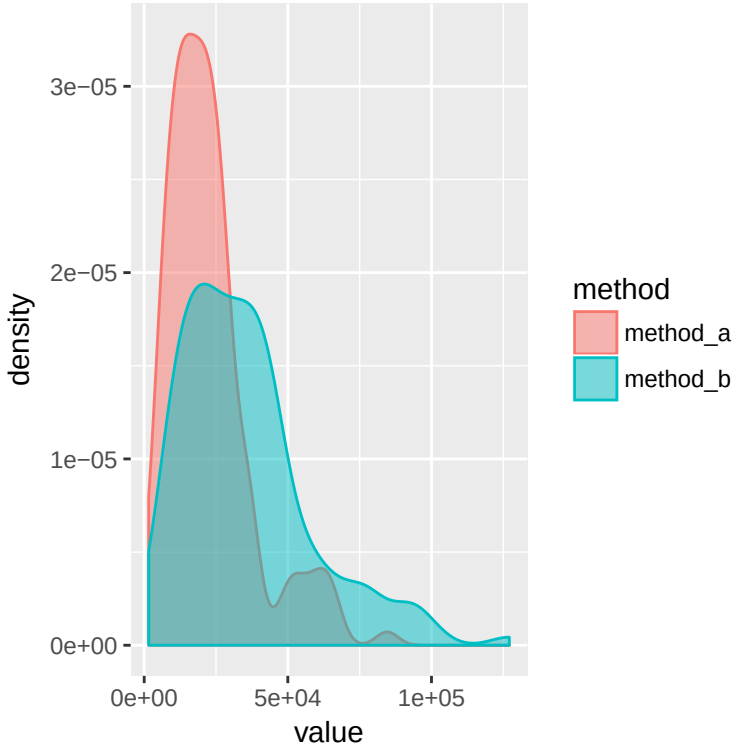
```
long_hcg <- hcg %>%
  gather(key="method", value="value",
         -specimen)
head(long_hcg)
```

```
# 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)
```




## 3.5 Method comparison (t-tests, and more)

### 3.5.1 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

```r
# 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**

### 3.5.2 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

```r
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
```

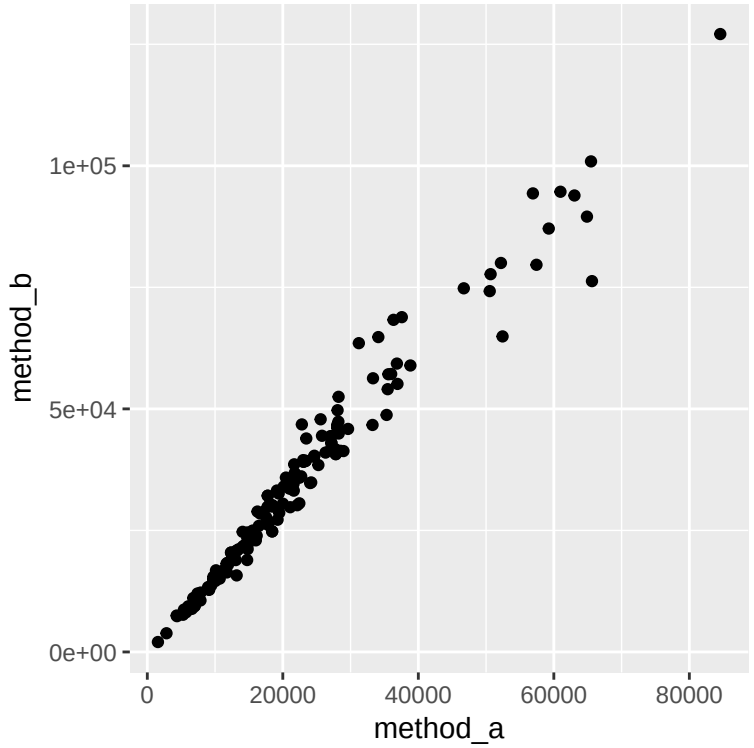
**End Exercise**

## 3.6 Regression

### 3.6.1 Simple linear regression

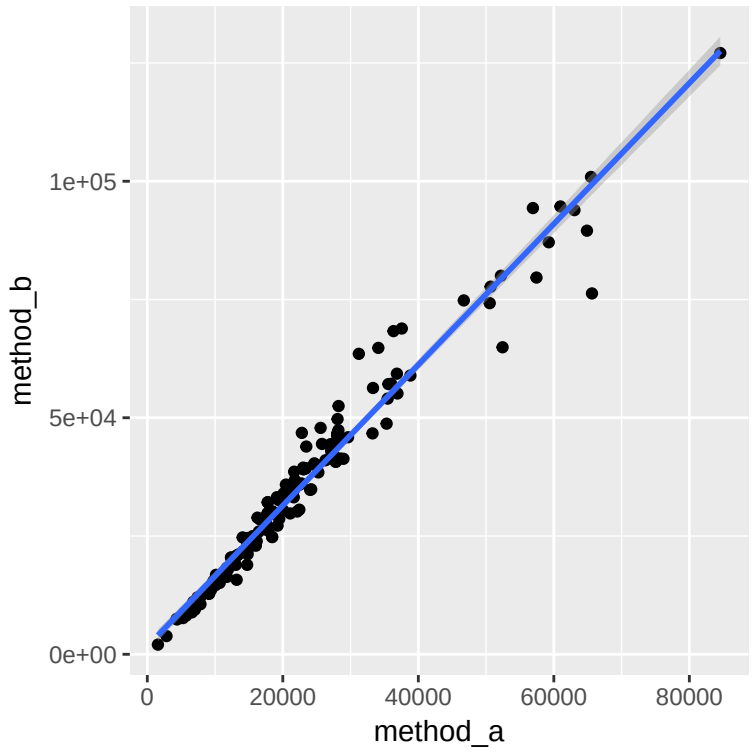
Let's begin by simply plotting `method_a` and `method_b` as a scatter plot. Notice how we are using the `aes()` to define "mappings" from our data to the x and y coordinates:

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



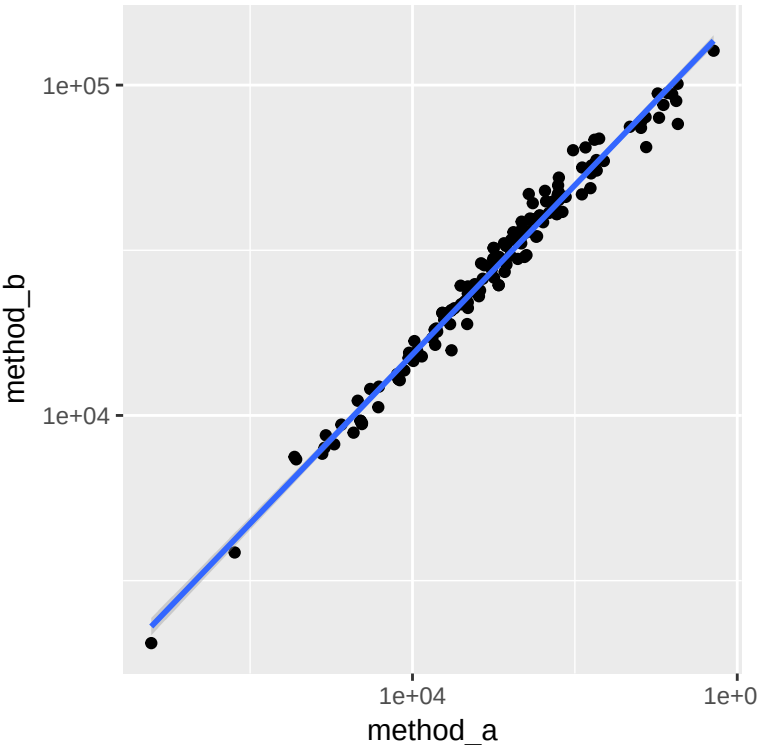

Adding a least-squares regression line is easy with a little bit of magic from `ggplot`. The `lm` (Linear Model) function does all the work here!

```
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 
```

## 3.7 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.

```
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 -563.564609 1969.98470
Slope       1.527165 NA    1.459922    1.61316
```

```
# 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
```

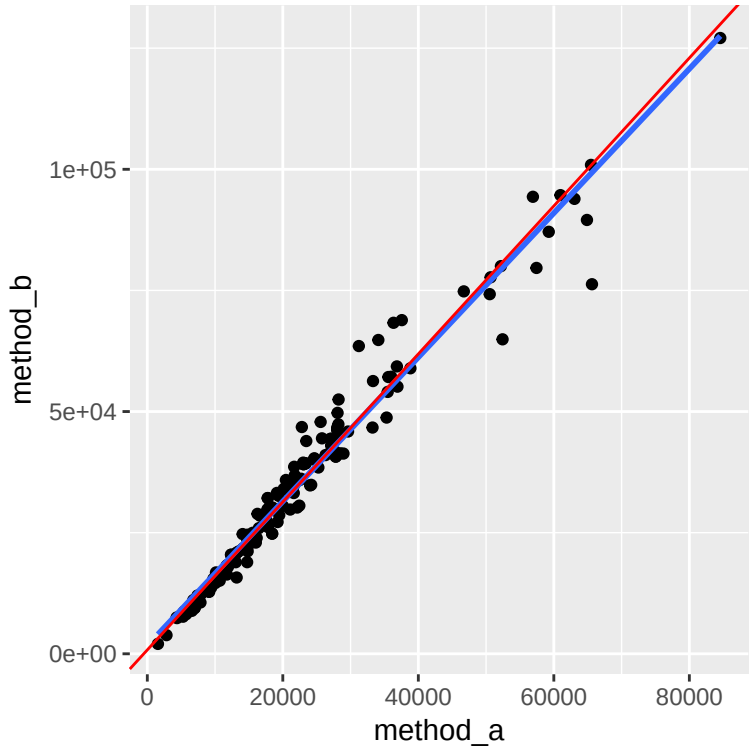
```
wdeming_results@para          # "para" short for "parameters"-- this is a library/method specific term h
```

```
                  EST SE          LCI        UCI
Intercept -595.104531 NA -1148.213941 -376.30964
Slope        1.592368 NA     1.562192    1.63894
```

**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.

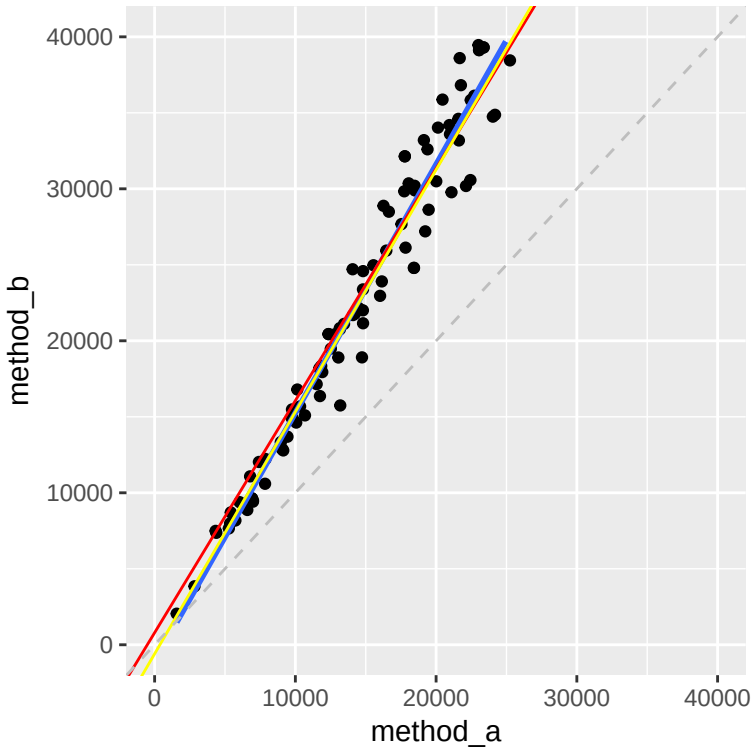
```
ggplot(hcg) +
  geom_point(aes(x=method_a, y=method_b))  +
  geom_smooth(method = "lm", aes(x=method_a, y=method_b), se=FALSE) +
  geom_abline(intercept = deming_results@para[1], slope = deming_results@para[2],
              color="red")
```




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.

```
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")
```



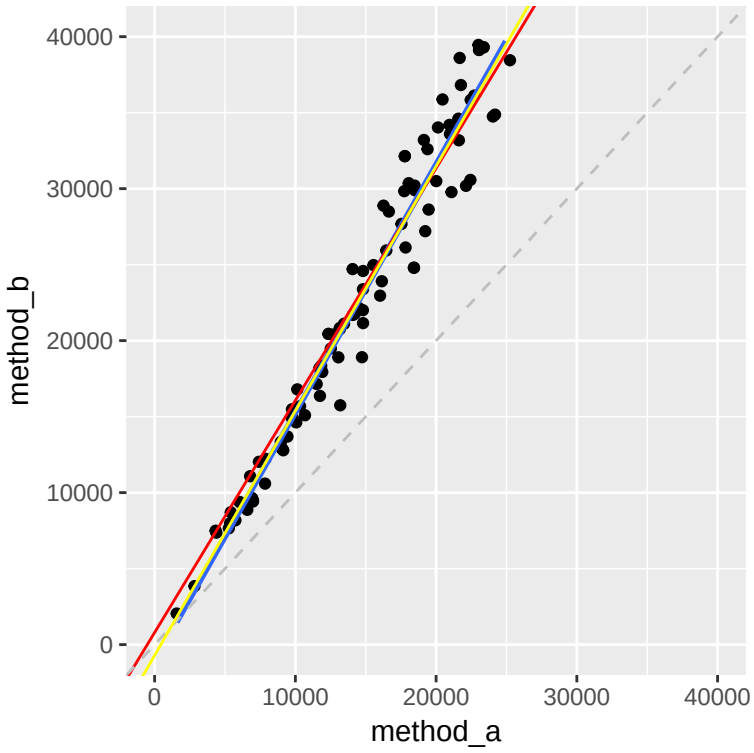

### 3.8 Passing-Bablock

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

```
                EST SE          LCI        UCI
Intercept -709.366764 NA -1544.614250 -75.022580
Slope        1.609158 NA     1.553464   1.667845
```

**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")
```

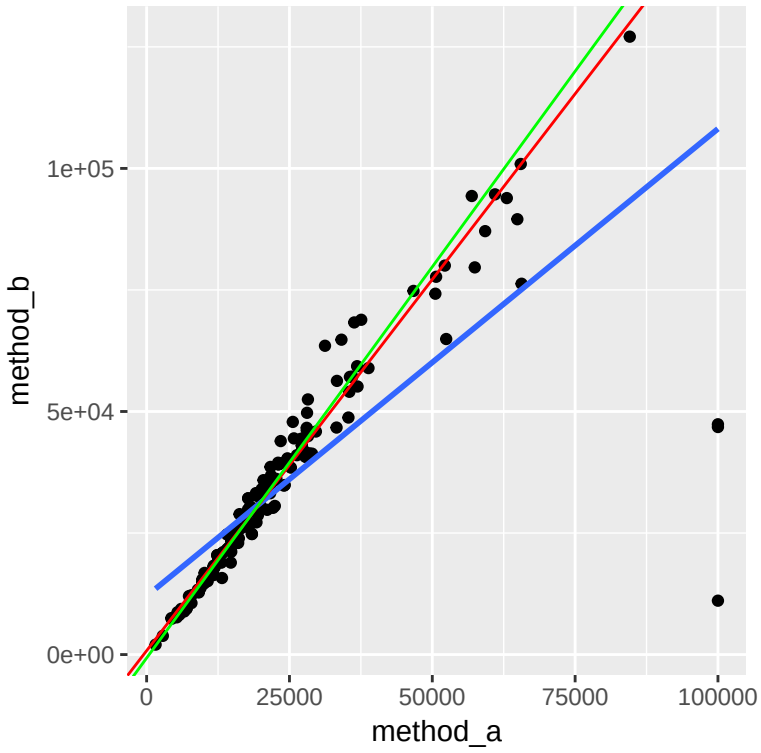



**End Exercise**

### 3.8.1 Extra-credit: Outlier robustness

How "robust" are each of these methods to outliers? Let's try it out.

```
# Step 1: make a copy of the data so we don't change the original
hcg_with_outliers <- hcg

# Step 2: modify the data to include some outliers for method_a (fake data!)
hcg_with_outliers$method_a[10:12] <- 100000

# Step 3: Re-run fits
deming_results_outliers <- mcreg(hcg$method_a, hcg$method_b,
                        method.reg = "Deming")
PB_results_outliers <- mcreg(hcg$method_a, hcg$method_b, method.reg = "PaBa")

# Step 3: same plotting code as above, using our new fake data
ggplot(hcg_with_outliers) +
  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_outliers@para[1],
              slope = deming_results_outliers@para[2], color="red") +
  geom_abline(intercept = PB_results_outliers@para[1],
              slope = PB_results_outliers@para[2], color="green")
```




That is quite a difference!

## 3.9 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 `cut()` function to break a numerical range into multiple a set of factor levels (categories): For instance, for method_a you could run `cut(hcg$method_a, breaks=c(0, 20, 40, Inf), labels=c("low","middle","high"))` to convert to a factor for 0-20, 20-40, 40-Inf.

*Hint #2*: Look at previous code for inspiration!

```
hcg %>%
  mutate(method_a_bin = cut(method_a,
                            breaks=c(-Inf, 25000, 50000, Inf),
                            labels=c("low","middle","high")),    # Create factor indicator for method_a
         method_b_bin = cut(method_b,
                            breaks=c(-Inf, 25000, 50000, Inf),
                            labels=c("low","middle","high"))) %>%
  count(method_a_bin, method_b_bin)  %>%
  spread(method_b_bin, n, fill=0, sep=".")    # Spreads method_b_pos from a single variable to a variabl
```

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

**End Exercise**

# 4 Method Validation: Precision, Linearity, and Calibration Verification

## 4.1 Overview

In this section, we will evaluate analytical method precision, calibration verification, and linearity.

## 4.2 Precision

### 4.2.1 Load data

**Exercise 1:** Load the `Precision` tab of `Method_Validation.data.xlsx` into the object `precision`.

```
precision <- read_xlsx(path = "data/Method_Validation.data.xlsx", sheet = "Precision")
str(precision)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':   60 obs. of  7 variables:
 $ Sample : num  1 2 3 4 5 6 7 8 9 10 ...
 $ Analyte: chr  "AFP" "AFP" "AFP" "AFP" ...
 $ BLANK  : logi  NA NA NA NA NA NA ...
 $ P1     : num  7.97 7.65 8.29 8.12 8.52 8.23 8.61 7.93 8.31 8.21 ...
 $ P2     : num  26.2 26.8 25.2 25.1 26 ...
 $ P3     : num  NA NA NA NA NA NA NA NA NA NA ...
 $ P4     : logi  NA NA NA NA NA NA ...
```

```
head(precision)
```

```
# A tibble: 6 x 7
  Sample Analyte BLANK    P1    P2    P3 P4
   <dbl> <chr>   <lgl> <dbl> <dbl> <dbl> <lgl>
1     1. AFP     NA     7.97  26.2    NA NA
2     2. AFP     NA     7.65  26.8    NA NA
3     3. AFP     NA     8.29  25.2    NA NA
4     4. AFP     NA     8.12  25.1    NA NA
5     5. AFP     NA     8.52  26.0    NA NA
6     6. AFP     NA     8.23  25.6    NA NA
```

**End exercise**

### 4.2.2 Describe data and explore its distribution

Let's figure out what we have got. Looks like there are 7 variables (Sample, Analyte, BLANK, P1, P2, P3, P4) with a maximum of 60 observations. However, there are many `NA`'s, which indicates missing values. In this set we have missing data just because it was not loaded into our example dataset, so we can focus on describing the data that is present. There are lots of ways to describe missingess.

Start by asking what is missing with function `is.na()`.

```
is.na(precision$P2)
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[56]  TRUE  TRUE  TRUE  TRUE  TRUE
```

How many missing values are there? We can count `TRUE`'s like in the previous section using `sum` and `length`:

```
tmp <- is.na(precision)     # Apply `is.na` to each element of `precision`
sum(tmp) / length(tmp) * 100  # Calculate percent missing
```

```
[1] 42.85714
```

```
sprintf("There are %d (%.0f%%) missing values.",
        sum(tmp),
        sum(tmp) / length(tmp) * 100)
```

```
[1] "There are 180 (43%) missing values."
```

We can also apply a tidy-er approach:

```
precision %>%
  map_df(is.na) %>%   # Apply `is.na` to each element
  summarize_all(sum)   # Apply `sum` to each variable
```

```
# A tibble: 1 x 7
  Sample Analyte BLANK    P1    P2    P3    P4
   <int>   <int> <int> <int> <int> <int> <int>
1      0       0    60     0    30    30    60
```

**Exercise 2**: How many different analytes (`Analyte`) are there? - Use the `unique` function to distill an object to a set of unique observations.

```
precision$Analyte
```

```
 [1] "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP"
[12] "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP"
[23] "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "AFP" "uE3" "uE3" "uE3"
[34] "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3"
[45] "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3" "uE3"
[56] "uE3" "uE3" "uE3" "uE3" "uE3"
```

```
unique(precision$Analyte)
```

```
[1] "AFP" "uE3"
```

```
sprintf("There are %d different analytes", length(unique(precision$Analyte)))
```

```
[1] "There are 2 different analytes"
```

**End exercise**

### 4.2.3 Assess precision

Let's focus on the first control `P1`'s inter-day measurements for `AFP`.

To match how we did this in 02a, we can extract this data from the tibble into a vector named `tmp`. Note that following `%>%` the `.` refers to the current version of `tmp`, after enacting the previous lines of code

```
tmp <- precision %>%
  filter(Analyte == "AFP") %>%  # Include only observations of AFP only
  select(P1)  %>%   # Select specific column
  .[["P1"]]      # Extract variable into a vector
tmp
```

```
 [1] 7.97 7.65 8.29 8.12 8.52 8.23 8.61 7.93 8.31 8.21 7.18 7.48 7.70 8.86
[15] 7.24 7.79 8.28 7.74 8.00 7.48 7.92 7.58 7.61 8.34 7.76 7.61 8.26 8.49
[29] 8.09 8.87
```

Then calculate the `mean`, `sd`, and `CV` as earlier. Note that to format each of the numbers we can use `sprintf` and specify the total number of digits and the number of digits following the decimal (e.g. `%2.1f` refers to a number with 2 totals digits and 1 digit after decimal)

```
mean(tmp)
```

```
[1] 8.004
```

```
sd(tmp)
```

```
[1] 0.4396441
```

```
sd(tmp) / mean(tmp) * 100
```

```
[1] 5.492804
```

```
sprintf("Mean = %2.1f, SD = %2.2f, CV = %3.1f%%",
        mean(tmp),
        sd(tmp),
        sd(tmp) / mean(tmp) * 100)
```

```
[1] "Mean = 8.0, SD = 0.44, CV = 5.5%"
```

To scale such calculations to multiple variables, it is much easier to do tidy way using `summarize` to calculate the `mean` across all observations of a specific variable `P1`:

```
precision %>%
  filter(Analyte == "AFP") %>%
  summarize(mean_P1 = mean(P1))
```

```
# A tibble: 1 x 1
  mean_P1
    <dbl>
1    8.00
```

**Exercise 3:** Add to the above result the standard deviation and CV for variable `P1`, by adding additional arguments to `summarize`

```
precision %>%
  filter(Analyte == "AFP") %>%
  summarize(mean_P1 = mean(P1),
            sd_P1 = sd(P1),
            CV_P1 = sd(P1) / mean(P1) * 100)
```

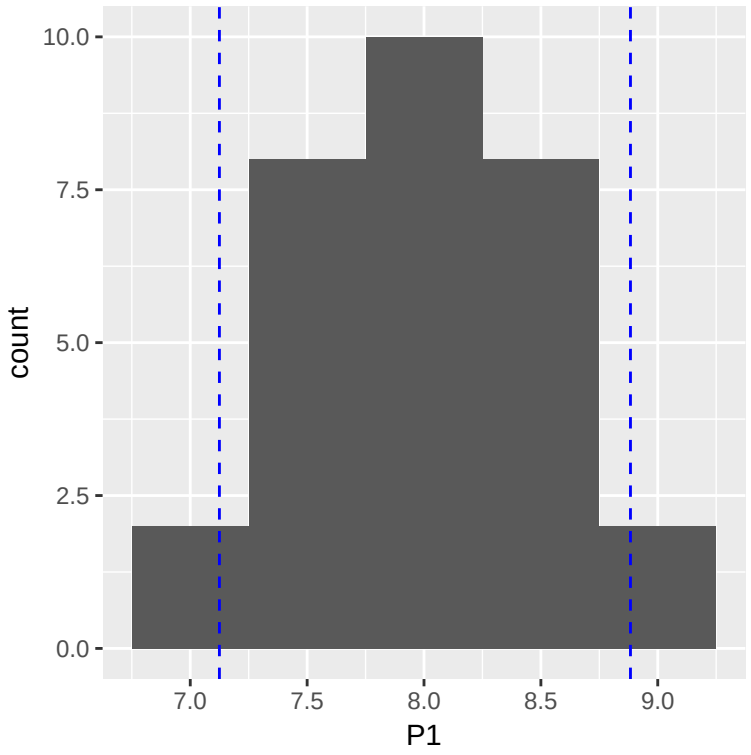
```
# A tibble: 1 x 3
  mean_P1 sd_P1 CV_P1
    <dbl> <dbl> <dbl>
1    8.00 0.440  5.49
```

**End exercise**

### 4.2.4 Visualize

**Exercise 4:** Plot histogram of these results with vertical lines marking parametric 95% central range. Consider setting the `binwidth`

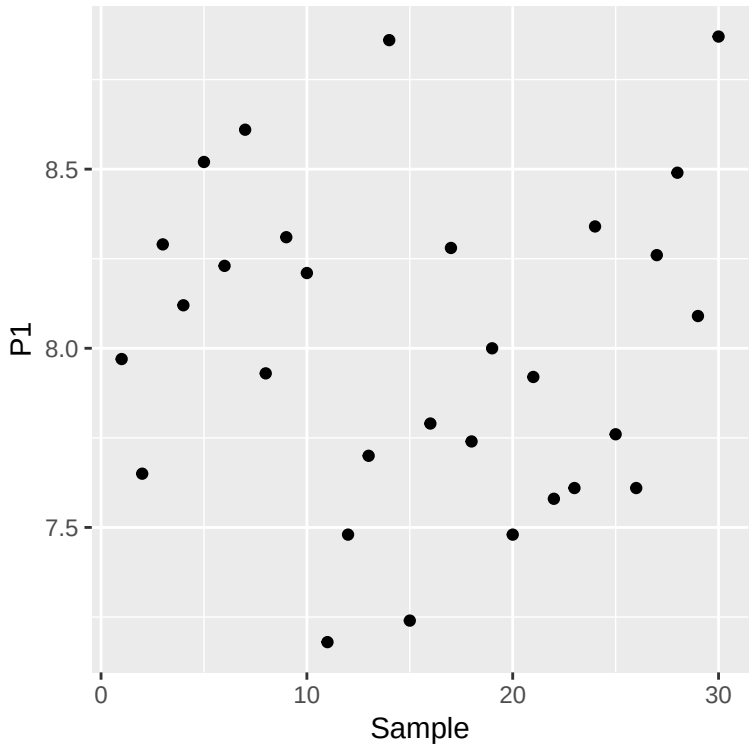
```
g <- precision %>%
  filter(Analyte == "AFP") %>%
  ggplot()
g + geom_histogram(aes(x=P1), binwidth=0.5) +
  geom_vline(aes(xintercept = mean(P1) + 2 * sd(P1)), linetype=2, color="blue") +
  geom_vline(aes(xintercept = mean(P1) - 2 * sd(P1)), linetype=2, color="blue")
```




**End exercise**

As expected, interday replicates of QC samples look relatively normally distributed. The other way we classically look at such QC results is longitudinally. Let's plot results over time using `geom_point`

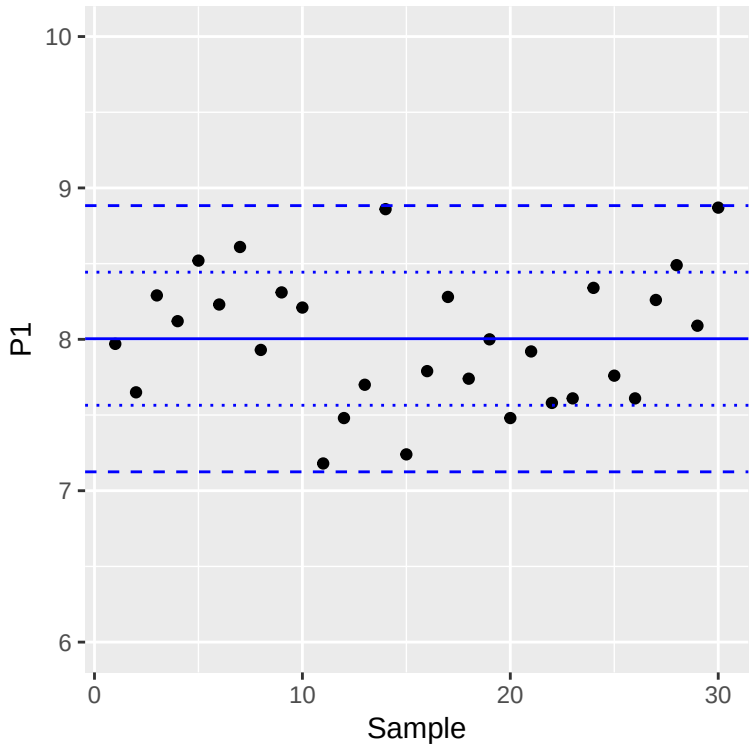
```
g <- precision %>%
  filter(Analyte == "AFP") %>%
  ggplot()
g + geom_point(aes(x=Sample, y=P1))
```




**Exercise 5:** Let's customize this plot to make it look a bit more useful

- Add horizontal lines using `geom_hline` for the mean, +/- 1 SD, and +/-1 2SDs
- Change the y-axis range using `ylim` to 6 - 10

```
g <- precision %>%
  filter(Analyte == "AFP") %>%
  ggplot()
g + geom_point(aes(x=Sample, y=P1)) +
  geom_hline(aes(yintercept = mean(P1)), linetype=1, color="blue") +
  geom_hline(aes(yintercept = mean(P1) + 1 * sd(P1)), linetype=3, color="blue") +
  geom_hline(aes(yintercept = mean(P1) + 2 * sd(P1)), linetype=2, color="blue") +
  geom_hline(aes(yintercept = mean(P1) - 1 * sd(P1)), linetype=3, color="blue") +
  geom_hline(aes(yintercept = mean(P1) - 2 * sd(P1)), linetype=2, color="blue") +
  ylim(6, 10)
```




Is this imprecision acceptable? . . . depends on analytical and clinical goals.

### 4.2.5 LOB (Optional)

Limit of the Blank is the minimum concentration that a sample without the analyte will rarely be as high as. We often approximate this as the 95th percentile of distribution of results from measuring a BLANK sample.

Unfortunately, our loaded datasset has no measures of a blank sample:

```
precision$BLANK
```

```
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[24] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[47] NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

So, let's simulate some. Let's simulate normally distributed data with mean 1.6 and standard deviation of 1.2 using `rnorm`.

```
n_samples <- 1e4
set.seed(13)   # Make `random` data simulation reproducible

x <- rnorm(n=n_samples, mean = 1.6, sd = 1.2)    # Generate simulated data
head(x)
```

```
[1] 2.265192 1.263674 3.730196 1.824784 2.971031 2.098631
```

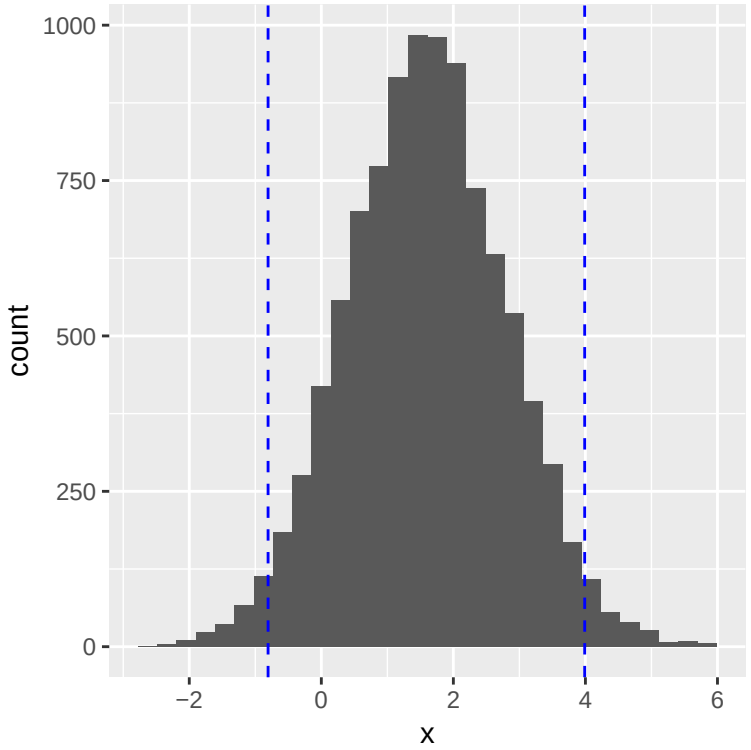
```
sim_precision <- tibble(sample = 1:n_samples,  # Put data into a table
                        x = x)
head(sim_precision)
```

```
# A tibble: 6 x 2
  sample     x
   <int> <dbl>
```

```
1       1  2.27
2       2  1.26
3       3  3.73
4       4  1.82
5       5  2.97
6       6  2.10
```

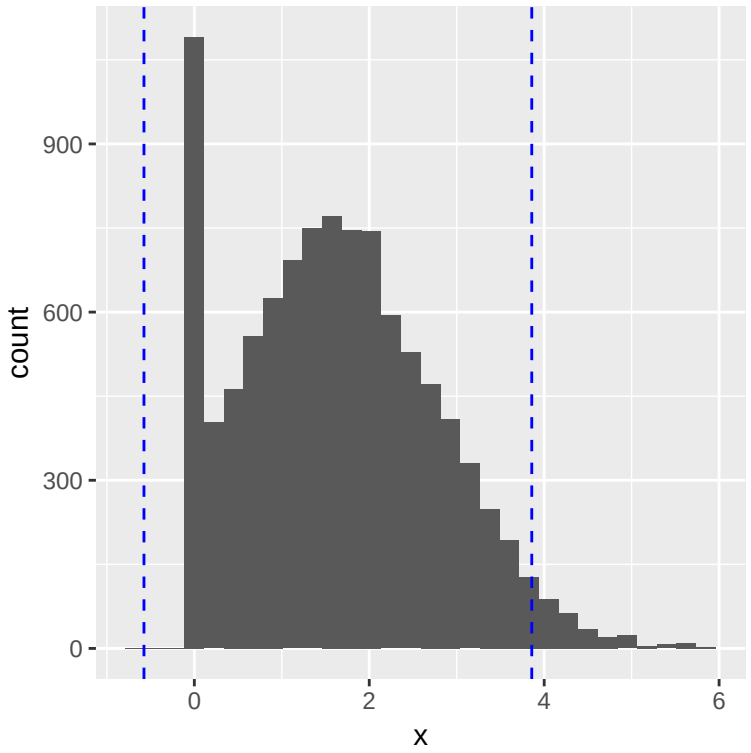
Spot check the histogram of this simulated data:

```
ggplot(data=sim_precision) +
  geom_histogram(aes(x=x)) +
  geom_vline(aes(xintercept = mean(x) + 2 * sd(x)), linetype=2, color="blue") +
  geom_vline(aes(xintercept = mean(x) - 2 * sd(x)), linetype=2, color="blue")
```




Note that there are simulated results less than 0. As an aside, here is one option for handling these:

```
tmp <- sim_precision
mask <- tmp$x < 0               # boolean mask
observation_vec <- which(mask)   # vector of affected (TRUE) columns
tmp$x[observation_vec] <- 0      # Convert these to 0

# replot
ggplot(data=tmp) +
  geom_histogram(aes(x=x)) +
  geom_vline(aes(xintercept = mean(x) + 2 * sd(x)), linetype=2, color="blue") +
  geom_vline(aes(xintercept = mean(x) - 2 * sd(x)), linetype=2, color="blue")
```




**Exercise 6:** Calculate LOB as 1.645 SDs above mean of the blank

```
mean(sim_precision$x)
```

```
[1] 1.591381
```

```
sd(sim_precision$x)
```

```
[1] 1.198844
```

```
mean(sim_precision$x) + 1.645*sd(sim_precision$x)
```

```
[1] 3.56348
```

Alternatives for similar calculations

```
quantile(x, probs = 0.95)  # non-parametric 95th percentile
```

```
     95%
3.564523
```

```
mean(x) + qnorm(0.95) * sd(x)   # Extract the SD factor for the 95th percentile in normal distribution
```

```
[1] 3.563304
```

**End exercise**

## 4.3 Calibration Verification

### 4.3.1 Load data for AFP

```
afp <- read_xlsx(path = "data/Method_Validation.data.xlsx",
                 sheet = "Linearity") %>%
```

```
  filter(Test == "AFP")
str(afp)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':   7 obs. of  6 variables:
 $ Sample        : chr  "S0" "S1" "S2" "S3" ...
 $ Test          : chr  "AFP" "AFP" "AFP" "AFP" ...
 $ Result_1      : num  0.09 2.71 5.91 30.29 122.97 ...
 $ Result_2      : num  0.09 2.64 6.08 29.54 117.89 ...
 $ Result_3      : num  0.03 3.09 5.98 31.71 116.8 ...
 $ Assigned_Value: num  0 2.5 5.3 25 98 503 2700
```

```
head(afp)
```

```
# A tibble: 6 x 6
  Sample Test  Result_1 Result_2 Result_3 Assigned_Value
  <chr>  <chr>    <dbl>    <dbl>    <dbl>          <dbl>
1 S0     AFP     0.0900   0.0900   0.0300           0.
2 S1     AFP     2.71     2.64     3.09             2.50
3 S2     AFP     5.91     6.08     5.98             5.30
4 S3     AFP    30.3     29.5     31.7             25.0
5 S4     AFP   123.     118.     117.              98.0
6 S5     AFP   569.     576.     573.             503.
```

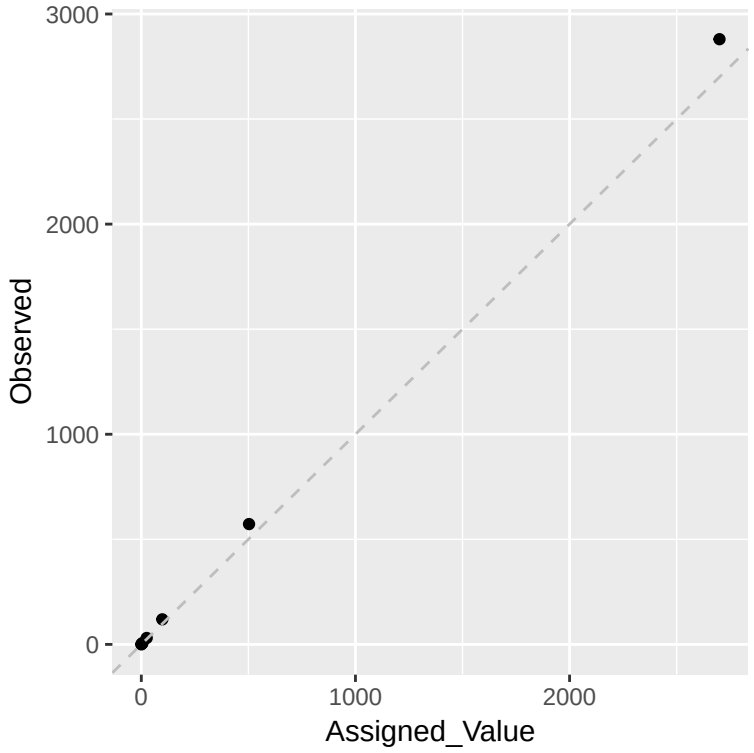
Calculate average result for each set of sample test replicates using `mutate`

```
afp <- afp %>%
  mutate(Observed = (Result_1 + Result_2 + Result_3) / 3)
```

### 4.3.2 Visualize calibration verification

Plot calverification results for AFP

```
ggplot(data=afp) +
  geom_point(aes(x=Assigned_Value, y=Observed)) +
  geom_abline(slope=1, intercept=0, linetype=2, color="gray")
```




How far off are observed values from assigned?

```
afp <- afp %>%
  mutate(value_diff = Observed - Assigned_Value) %>%
  mutate(value_percent_diff = value_diff / Assigned_Value * 100,
         recovery = Observed / Assigned_Value * 100)

afp$value_percent_diff
```

```
[1]       Inf 12.533333 13.018868 22.053333 21.653061 13.848244  6.669136
```

```
afp$recovery
```

```
[1]      Inf 112.5333 113.0189 122.0533 121.6531 113.8482 106.6691
```

Do these meet our goals?

- Let's apply a simple goal of % difference $< 30\%$.
- Do all dilutions meet this threshold?
- Note use of `abs` for absolute value

```
tmp <- afp %>%
  mutate(pass_calvar = abs(value_percent_diff) < 30)

# Visualize the relevant variables
tmp %>%
  select(Sample, value_percent_diff, recovery, pass_calvar)
```

```
# A tibble: 7 x 4
  Sample value_percent_diff recovery pass_calvar
  <chr>               <dbl>    <dbl> <lgl>
1 S0                  Inf        Inf  FALSE
2 S1                   12.5      113. TRUE
```

```
3 S2                  13.0      113. TRUE
4 S3                  22.1      122. TRUE
5 S4                  21.7      122. TRUE
6 S5                  13.8      114. TRUE
7 90% S6               6.67     107. TRUE
```

```
# Display the failed observations
tmp %>%
  filter(!pass_calvar)
```

```
# A tibble: 1 x 11
  Sample Test  Result_1 Result_2 Result_3 Assigned_Value Observed
  <chr>  <chr>    <dbl>    <dbl>    <dbl>          <dbl>    <dbl>
1 S0     AFP     0.0900   0.0900   0.0300             0.   0.0700
# ... with 4 more variables: value_diff <dbl>, value_percent_diff <dbl>,
#   recovery <dbl>, pass_calvar <lgl>
```

**Exercise 7:** Seems that our acceptability criteria was too simplistic, because it only considered relative differences.

- What about criteria for %difference $< 30\%$ or absolute difference $< 1$?

  Note: multiple boolean vectors can be combined together using the bitwise OR operator | (e.g. '(is.na(A)) | (A < 5)).

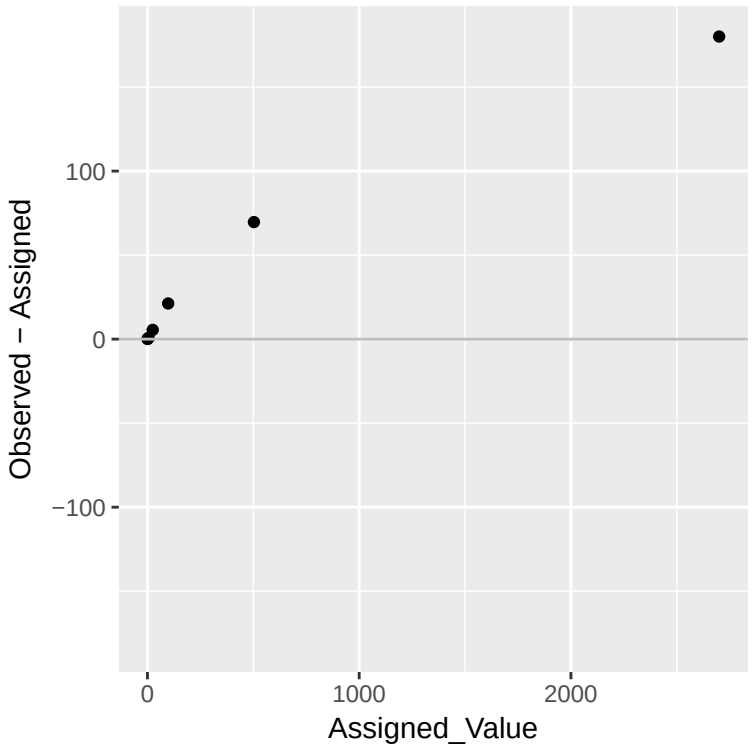
```
afp %>%
  filter(Test == "AFP") %>%
  mutate(pass_calvar = (abs(value_percent_diff) < 30) | (abs(value_diff) < 1)) %>%
  filter(!pass_calvar)
```

```
# A tibble: 0 x 11
# ... with 11 variables: Sample <chr>, Test <chr>, Result_1 <dbl>,
#   Result_2 <dbl>, Result_3 <dbl>, Assigned_Value <dbl>, Observed <dbl>,
#   value_diff <dbl>, value_percent_diff <dbl>, recovery <dbl>,
#   pass_calvar <lgl>
```

**End exercise**

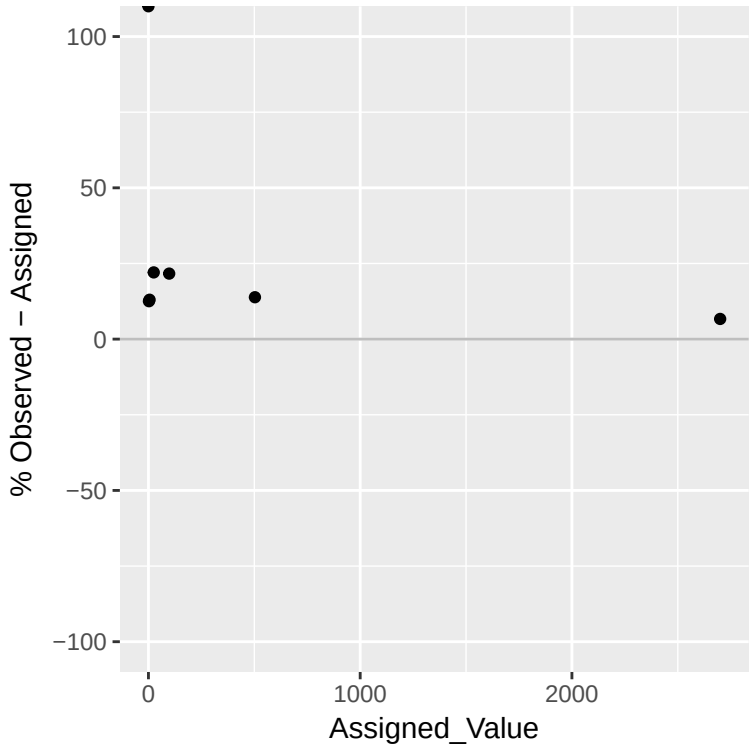
Let's plot these absolute differences

```
tmp <- afp

# Figure out plot symmetric y-axis limits
max_diff <- abs( max(tmp$value_diff, na.rm=T) )

ggplot(data=tmp) +
  geom_point(aes(x=Assigned_Value, y=value_diff)) +
  geom_hline(yintercept = 0, linetype=1, color="gray") +
  ylim(-max_diff, max_diff) +
  ylab("Observed - Assigned")       # Change y-axis label
```




**Exercise 8:** Adapt the above code to plot the percent differences

```
g <- ggplot(data=tmp) +
  geom_point(aes(x=Assigned_Value, y=value_percent_diff)) +
  geom_hline(yintercept = 0, linetype=1, color="gray") +
  ylim(-100, 100) +
  ylab("% Observed - Assigned")
g
```




**End exercise**

## 4.4 Linearity (optional)

The dataset gives us the assigned values for each sample, so we can back calculate the expected ratios between each sample based on dilution

```
afp <- afp %>%
  mutate(dilution = Assigned_Value / max(Assigned_Value))
afp$dilution
```

```
[1] 0.0000000000 0.0009259259 0.0019629630 0.0092592593 0.0362962963
[6] 0.1862962963 1.0000000000
```

**Exercise 9:** Calculate each samples results based on expected dilution factor and observed result in S6.

- Assume there is no contribution from S0 in the mixing experiment

```
Observed_S6 <- max(afp$Observed)
afp <- afp %>%
  mutate(expected_result = Observed_S6 * dilution)

afp %>%
  select(Sample, Observed, expected_result)
```

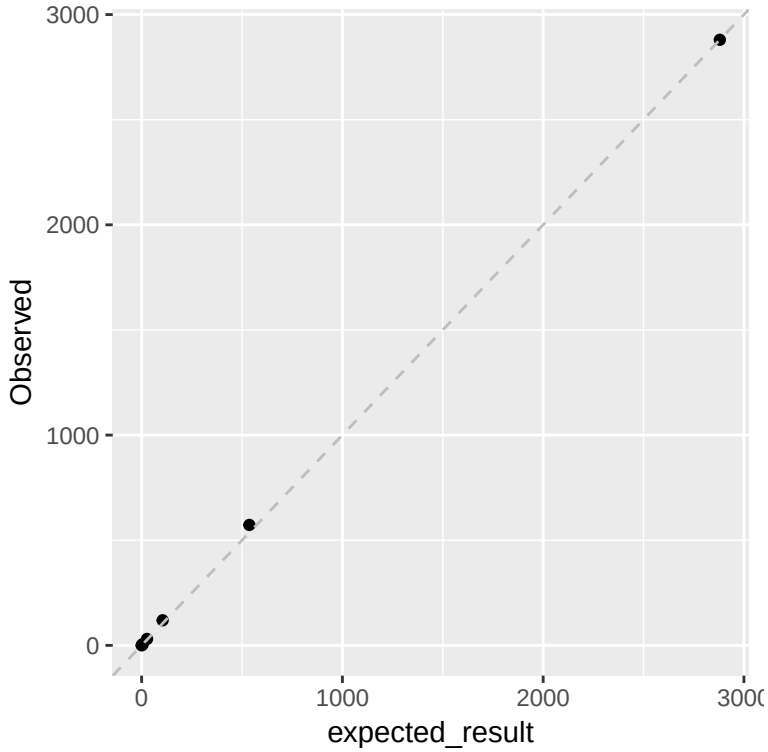
```
# A tibble: 7 x 3
  Sample  Observed expected_result
  <chr>      <dbl>           <dbl>
1 S0        0.0700            0.
2 S1        2.81              2.67
3 S2        5.99              5.65
4 S3       30.5              26.7
```

```
5 S4      119.              105.
6 S5      573.              537.
7 90% S6 2880.             2880.
```

**End exercise**

Plot linearity results

```
ggplot(data=afp) +
  geom_point(aes(x=expected_result, y=Observed)) +
  geom_abline(slope=1, intercept=0, linetype=2, color="gray")
```




Evaluate linearity results

```
tmp <- afp %>%
  mutate(value_diff = Observed - expected_result) %>%
  mutate(percent_value_diff = value_diff / expected_result * 100,
         recovery = Observed / expected_result * 100) %>%
  mutate(pass_linearity = (abs(value_diff) < 1) | (abs(percent_value_diff) < 30))

# Visualize the relevant variables
tmp %>%
  select(Sample, value_diff, percent_value_diff, recovery, pass_linearity)
```

```
# A tibble: 7 x 5
  Sample value_diff percent_value_diff recovery pass_linearity
  <chr>       <dbl>              <dbl>    <dbl> <lgl>
1 S0         0.0700             Inf        Inf  TRUE
2 S1         0.147                5.50     105. TRUE
3 S2         0.337                5.95     106. TRUE
4 S3         3.85                14.4      114. TRUE
5 S4        14.7                 14.0      114. TRUE
```

```
6 S5        36.1                  6.73     107. TRUE
7 90% S6     0.                   0.       100. TRUE
```

```
# Visualize the relevant observations
tmp %>%
  filter(!pass_linearity)
```

```
# A tibble: 0 x 14
# ... with 14 variables: Sample <chr>, Test <chr>, Result_1 <dbl>,
#   Result_2 <dbl>, Result_3 <dbl>, Assigned_Value <dbl>, Observed <dbl>,
#   value_diff <dbl>, value_percent_diff <dbl>, recovery <dbl>,
#   dilution <dbl>, expected_result <dbl>, percent_value_diff <dbl>,
#   pass_linearity <lgl>
```

# 5 Exploratory Data Analysis: Orienting Yourself with Your Data Set

In this section of the course we will walk through a common workflow: exploring a new data set. Before we dive into the data set, we will touch briefly on one way of writing code that can help make it easier to follow.

## 5.1 Sequencing functions

When you are working with a data set, you often need to manipulate it multiple times in a defined sequence of events. Let's start with a non-sensical example that can help illustrate the issue (adapted from the tidyverse style guide).

Let's say we want to apply the functions hop, scoop, and bop to the foo_foo data frame, in that order. One way to approach that is to start with the data, apply the function, and write the output back into the original data frame.

```
# 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)
```

R allows you to nest functions within one another, but this can get horribly confusing because following a specific sequence of operations requires you to start from the inside of the expression and expand outwards.

```
# 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)
```

You want to try and avoid doing things this way because the sequence of operations is so non-intuitive.

Explicitly showing the functions sequentially by line is helpful for readability but it does require some unnecessary typing to keep repeating the name of the data set. R allows you to "pipe" a data frame from one function to another using this funny looking operator: %>%. This can cut down on unnecessary code but also preserves the nice formatting that makes it obvious what functions are applied in what order.

```
# 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.

## 5.2 Loading data and reviewing data types

First let's refresh your memory on loading in a data set. We have an Excel file that contains our main data set in the data folder called "orders_data_set.xlsx". After loading the file into a variable (in this case a data frame) called "orders", look at the structure of the data using the `str()` function.

```
orders <- read_excel("data/orders_data_set.xlsx")
str(orders)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':   45002 obs. of  15 variables:
 $ Order ID               : num  19766 88444 40477 97641 99868 ...
 $ Patient ID             : num  511388 511388 508061 508061 505646 ...
 $ Description            : chr  "PROTHROMBIN TIME" "BASIC METABOLIC PANEL" "THYROID STIMULATING HORMONE
 $ Proc Code              : chr  "PRO" "BMP" "TSH" "T4FR" ...
 $ ORDER_CLASS_C_DESCR    : chr  "Normal" "Normal" "Normal" "Normal" ...
 $ LAB_STATUS_C           : num  NA NA 3 3 3 3 3 3 3 3 ...
 $ LAB_STATUS_C_DESCR     : chr  NA NA "Final result" "Final result" ...
 $ ORDER_STATUS_C         : num  4 4 5 5 5 5 5 5 5 5 ...
 $ ORDER_STATUS_C_DESCR   : chr  "Canceled" "Canceled" "Completed" "Completed" ...
 $ REASON_FOR_CANC_C      : num  11 11 NA NA NA NA NA NA NA NA ...
 $ REASON_FOR_CANC_C_DESCR: chr  "Auto-canceled. Patient no show and/or specimen not received within 60
 $ Order Time             : POSIXct, format: "2017-08-13 11:59:00" "2017-08-13 11:59:00" ...
 $ Result Time            : POSIXct, format: NA NA ...
 $ Review Time            : POSIXct, format: NA NA ...
 $ Department             : chr  "INTERNAL MEDICINE CLINIC" "INTERNAL MEDICINE CLINIC" "ENDOCRINOLOGY CL
```

**Exercise 1:**

1. Which fields of the data frame are characters?
2. Which are numbers?
3. Of those fields above, which would be best represented as factors?
4. Let's start by running a summary (`summary`) of one of those character fields that we think should be a factor. What information can we determine from that summary?

```
 Length     Class      Mode
  45002 character character
```

5. Let's convert one of those fields to a factor using the `as.factor()` command and then run a summary. What additional information do you see by converting to a factor?

```
Clinic Collect       External     Historical         Normal        On Site
          6427            401              5          36326           1843
```

**End Exercise**

Tip: White space generally has meaning in programming. When a variable name has a space in it, you can use the ` character (look to the top left on your keyboard) around the variable name to make sure R understands what variable you are referring to. As an example, `summary(orders$Proc Code)` will not work but `summary(orders$`Proc Code`)` will.

White spaces in names can be annoying to deal with. To get around this, we can rename variable names to remove white spaces, and, in addition, we can convert everyting to a single case (lowercase by default). The janitor package has some handy data science tools, including the ability to clean up variable names in one line using the `clean_names` function:

```
orders <- read_excel("data/orders_data_set.xlsx") %>%
  clean_names()
str(orders)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':   45002 obs. of  15 variables:
 $ order_id               : num  19766 88444 40477 97641 99868 ...
 $ patient_id             : num  511388 511388 508061 508061 505646 ...
 $ description            : chr  "PROTHROMBIN TIME" "BASIC METABOLIC PANEL" "THYROID STIMULATING HORMONE
 $ proc_code              : chr  "PRO" "BMP" "TSH" "T4FR" ...
 $ order_class_c_descr    : chr  "Normal" "Normal" "Normal" "Normal" ...
 $ lab_status_c           : num  NA NA 3 3 3 3 3 3 3 3 ...
 $ lab_status_c_descr     : chr  NA NA "Final result" "Final result" ...
 $ order_status_c         : num  4 4 5 5 5 5 5 5 5 5 ...
 $ order_status_c_descr   : chr  "Canceled" "Canceled" "Completed" "Completed" ...
 $ reason_for_canc_c      : num  11 11 NA NA NA NA NA NA NA NA ...
 $ reason_for_canc_c_descr: chr  "Auto-canceled. Patient no show and/or specimen not received within 60
 $ order_time             : POSIXct, format: "2017-08-13 11:59:00" "2017-08-13 11:59:00" ...
 $ result_time            : POSIXct, format: NA NA ...
 $ review_time            : POSIXct, format: NA NA ...
 $ department             : chr  "INTERNAL MEDICINE CLINIC" "INTERNAL MEDICINE CLINIC" "ENDOCRINOLOGY CL
```

As you may have seen already, there is more than one way to do the same thing when you're programming. We used `str()` to look at the structure of a data frame or other object. Another way to do this is to use the `glimpse()` function, which produces very similar output but is organized a little more neatly (with 1 line per variable).

```
glimpse(orders)
```

```
Observations: 45,002
Variables: 15
$ order_id               <dbl> 19766, 88444, 40477, 97641, 99868, 311...
$ patient_id             <dbl> 511388, 511388, 508061, 508061, 505646...
$ description            <chr> "PROTHROMBIN TIME", "BASIC METABOLIC P...
$ proc_code              <chr> "PRO", "BMP", "TSH", "T4FR", "COMP", "...
$ order_class_c_descr    <chr> "Normal", "Normal", "Normal", "Normal"...
$ lab_status_c           <dbl> NA, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...
$ lab_status_c_descr     <chr> NA, NA, "Final result", "Final result"...
$ order_status_c         <dbl> 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,...
$ order_status_c_descr   <chr> "Canceled", "Canceled", "Completed", "...
$ reason_for_canc_c      <dbl> 11, 11, NA, NA, NA, NA, NA, NA, NA, NA...
$ reason_for_canc_c_descr <chr> "Auto-canceled. Patient no show and/or...
$ order_time             <dttm> 2017-08-13 11:59:00, 2017-08-13 11:59...
$ result_time            <dttm> NA, NA, 2017-09-20 11:59:00, 2017-09-...
$ review_time            <dttm> NA, NA, 2017-09-21 20:34:00, 2017-09-...
$ department             <chr> "INTERNAL MEDICINE CLINIC", "INTERNAL ...
```

The `str` and `glimpse` functions are helpful to get a quick snapshot of the data, but sometimes you want a little more detail about the data in your data frame.

```
summary(orders)
```

```
    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
```

The `summary` function is most useful when you want to quickly glance at distributions of numerical data and times. You can quickly see the minimum and maximum times and some data on the distribution. It is less helpful when you have characters. One way to deal with this issue to convert a character variable into a factor using the `as.factor()`function.

**Exericse 2:**

Pull a summary of the description variable (only), after converting to a factor using the `as.factor` function. Which test is most frequently ordered in this data set?

```
COMPREHENSIVE METABOLIC PANEL          HEMOGLOBIN A1C, HPLC
                         3639                          2470
                   CBC, DIFF         BASIC METABOLIC PANEL
                         2393                          2174
GC&CHLAM NUCLEIC ACID DETECTN                CBC (HEMOGRAM)
                         2164                          1979
```

**End Exercise**

## 5.3 Reshaping rectangular data

Much of the laboratory data we work with has a tibble (data frame) structure, with rows as observations and columns as variables. Sometimes we start with a giant spreadsheet with tens of columns where we only care about a few columns. Sometimes we start with a spreadsheet with thousands or millions of rows and only care about a small subset of those. There are a variety of functions in the dplyr package that help us reshape our rectangular data quickly.

### 5.3.1 Make your data skinny with `select()`

When starting with a wide tibble (lots of columns), we can make it skinny by selecting specific columns using the `select()` function. In addition to supplying the data frame as the first argument (not needed if using the pipe), `select()` expects the names of the variables you would like to extract.

```
orders_skinny <- orders %>%
  select(order_id, patient_id, description, order_time)
glimpse(orders_skinny)
```

```
Observations: 45,002
Variables: 4
$ order_id    <dbl> 19766, 88444, 40477, 97641, 99868, 31178, 87245, 5...
$ patient_id  <dbl> 511388, 511388, 508061, 508061, 505646, 505646, 50...
$ description <chr> "PROTHROMBIN TIME", "BASIC METABOLIC PANEL", "THYR...
$ order_time  <dttm> 2017-08-13 11:59:00, 2017-08-13 11:59:00, 2017-08...
```

### 5.3.2 Shorten your tall data with `filter()`

Alternately, we might have a long tibble (lots of rows) but only want to select specific rows meeting some criteria. One of the most useful functions in the dplyr package is `filter()`, which allows you to select specific rows from a data frame. The arguments to the function include the data frame (which can be skipped if you use a pipe) and then one or more conditions to select the rows you want.

Let's extract the rows associated with complete blood count orders. The procedure code for those rows is "CBC", so the condition we apply to the filter function is `proc_code == CBC`. Note that we use two equal signs instead of one to indicate an equality condition - you will get an error if you use a single equal sign.

```
cbc_orders <- orders %>%
  filter(proc_code == "CBC")
# note the two equal signs for evaluating equality
# (because one equal is for assignment)
glimpse(cbc_orders)
```

```
Observations: 1,979
Variables: 15
$ order_id               <dbl> 95066, 79887, 91635, 54707, 47425, 478...
$ patient_id             <dbl> 501844, 510902, 501184, 510501, 512673...
$ description            <chr> "CBC (HEMOGRAM)", "CBC (HEMOGRAM)", "C...
$ proc_code              <chr> "CBC", "CBC", "CBC", "CBC", "CBC", "CB...
$ order_class_c_descr    <chr> "Normal", "Normal", "Normal", "Normal"...
$ lab_status_c           <dbl> 3, NA, NA, 3, NA, 3, 3, 3, 3, NA, 3, 3...
$ lab_status_c_descr     <chr> "Final result", NA, NA, "Final result"...
$ order_status_c         <dbl> 5, 4, 4, 5, 4, 5, 5, 5, 5, 4, 5, 5, 4,...
```

```
$ order_status_c_descr    <chr> "Completed", "Canceled", "Canceled", "...
$ reason_for_canc_c       <dbl> NA, 11, 11, NA, 11, NA, NA, NA, NA, 11...
$ reason_for_canc_c_descr <chr> NA, "Auto-canceled. Patient no show an...
$ order_time              <dttm> 2017-08-14 09:35:00, 2017-08-14 09:46...
$ result_time             <dttm> 2017-08-15 14:19:00, NA, NA, 2017-08-...
$ review_time             <dttm> 2017-08-16 14:10:00, NA, NA, 2017-08-...
$ department              <chr> "CARDIOLOGY CLINIC", "NEPHROLOGY CLINI...
```

With that step, we have now create a tibble that only has the rows with a CBC order.

### 5.3.3 Sort your data with `arrange()`

In some cases we do not want to remove rows from our tibble but would like to re-order the rows to manually review the data or perform some other operations. The `arrange()` function can order the rows by the variables you specify. Be default, it will sort from smallest to largest, but you can also sort in descending order using the `desc()` function as an argument.

We may want to sort first by patient ID, then by order time.

```
orders_arranged <- orders %>%
  arrange(patient_id, order_time)
glimpse(orders_arranged)
```

```
Observations: 45,002
Variables: 15
$ order_id                <dbl> 25646, 39483, 91634, 33565, 98224, 375...
$ patient_id              <dbl> 500001, 500002, 500002, 500002, 500002...
$ description             <chr> "CHRONIC PAIN DRUG SCREEN, URN", "THYR...
$ proc_code               <chr> "UCPDS", "TSH", "COMP", "CBC", "LIPID"...
$ order_class_c_descr     <chr> "Normal", "Clinic Collect", "Clinic Co...
$ lab_status_c            <dbl> 5, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
$ lab_status_c_descr      <chr> "Edited Result - FINAL", "Final result...
$ order_status_c          <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,...
$ order_status_c_descr    <chr> "Completed", "Completed", "Completed",...
$ reason_for_canc_c       <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA...
$ reason_for_canc_c_descr <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA...
$ order_time              <dttm> 2017-09-21 10:51:00, 2017-09-21 11:07...
$ result_time             <dttm> 2017-09-22 14:13:00, 2017-09-21 14:22...
$ review_time             <dttm> 2017-10-09 20:02:00, 2017-10-02 14:42...
$ department              <chr> "INFECTIOUS DISEASE CLINIC", "NEIGHBOR...
```

**Exercise 3:**

Let's take these functions out for a spin. Generate a tibble that includes only the BMP orders with the following variables: order ID and time stamp variables (order_time, result_time, review_time). Sort the data by order_time, starting with the latest time stamp in the data set and ending with the earliest.

```
Observations: 2,174
Variables: 4
$ order_id    <dbl> 10691, 76449, 74995, 22407, 20305, 23919, 39761, 5...
$ order_time  <dttm> 2017-11-10 16:39:00, 2017-11-10 16:38:00, 2017-11...
$ result_time <dttm> 2017-11-10 17:29:00, 2017-11-10 18:05:00, 2017-11...
$ review_time <dttm> 2017-11-17 11:48:00, 2017-11-11 09:46:00, 2017-11...
```

**End Exercise**

### 5.3.4 Add extra columns with `mutate()`

Earlier we saw that factors can be helpful when summarizing data. What if we wanted to permanently convert one of our variables to a factor within the data frame? We could do this many different ways, but let's start by creating a new variable (column) using the `mutate()` function. `mutate()` allows you to perform a function on one or more variables to create (or overwrite) a variable within the same observation (row).

Here we take our orders data frame, pipe it into the `mutate()` function and create a new variable called department_fctr that applies the `as.factor` function on our original department variable. We can run the `summary` function to see the results.

```
orders <- orders %>%
  mutate(department_fctr = as.factor(department))
summary(orders$department_fctr) %>% head()
```

```
BEHAVIORAL HEALTH CLINIC        CARDIOLOGY CLINIC     ENDOCRINOLOGY CLINIC
                     503                     1026                     1486
  FAMILY MEDICINE CLINIC  GASTROENTEROLOGY CLINIC         GERIATRIC CLINIC
                    3501                      781                     1015
```

**Exercise 4:**

Let's go ahead and convert the description variable to a factor using the `mutate` and `as.factor` functions. This time, rather than creating a new variable, overwrite the original variable by giving it its original name.

```
COMPREHENSIVE METABOLIC PANEL          HEMOGLOBIN A1C, HPLC
                        3639                          2470
                  CBC, DIFF         BASIC METABOLIC PANEL
                        2393                          2174
GC&CHLAM NUCLEIC ACID DETECTN                CBC (HEMOGRAM)
                        2164                          1979
```

**End Exercise**

Factors can be very handy when you want to look at quick summaries of the data. In some cases you may want all of your variables that are characters to actually be factors. We're jumping into a little more advanced concepts, but let's briefly cover one way to convert multiple variables into factors at once.

If we decide to make description, proc_code, order_class_c_descr, lab_status_c_descr, order_status_c_descr, and reason_for_canc_c_descr all into factors, we could use `mutate` and call out every variable (on separate lines for readability):

```
orders_factors <- orders %>%
  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))
summary(orders_factors)
```

```
    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   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
                     department_fctr
INFECTIOUS DISEASE CLINIC    :11861
INTERNAL MEDICINE CLINIC     : 6330
FAMILY MEDICINE CLINIC       : 3501
NEIGHBORHOOD CLINIC          : 3128
INTERNATIONAL MEDICINE CLINIC: 2499
```

```
 RHEUMATOLOGY CLINIC          : 2422
 (Other)                      :15261
```

An extension of the `mutate` function is `mutate_at`, which serves the same purpose but allows you to choose multiple columns at once. If you're applying the same function to multiple columns, this is a handy way to do that with less code.

```
orders <- orders %>%
  mutate_at(c("description", "proc_code", "order_class_c_descr",
              "lab_status_c_descr", "order_status_c_descr",
              "reason_for_canc_c_descr"), as.factor)
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   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
                      department_fctr
 INFECTIOUS DISEASE CLINIC    :11861
 INTERNAL MEDICINE CLINIC     : 6330
 FAMILY MEDICINE CLINIC       : 3501
 NEIGHBORHOOD CLINIC          : 3128
 INTERNATIONAL MEDICINE CLINIC: 2499
 RHEUMATOLOGY CLINIC          : 2422
 (Other)                      :15261
```

Yet another way to do this is to use `mutate_if(is.character, as.factor)` but beware!

Older functions to import files in R automatically convert character variables into factors. This can be helpful for variables like order status where there are only a handful of different possible values for the variable (called "levels" of a factor). But factors behave differently than characters, and your code can produce unexpected output instead of failing.

## 5.4 Tabulating our data

A very common task when analyzing data is tabluating counts based on one or more variables. In Excel this is commonly handled with pivot tables. R has multiple functions that can help you quickly create tables. To demonstrate some tabulations, let's first focus on a single test and return to our CBC orders tibble we created in the previous section. We want to tabulate the counts of CBC orders based on a single variable, department, ie. we wnat ot determine the number of CBCs ordered by each department. To start out, we use the `table` function in base R (no additional package is required to call the function).

```
table(cbc_orders$department)
```

```
     BEHAVIORAL HEALTH CLINIC             CARDIOLOGY CLINIC
                            4                            70
         ENDOCRINOLOGY CLINIC          FAMILY MEDICINE CLINIC
                           22                           148
      GASTROENTEROLOGY CLINIC               GERIATRIC CLINIC
                           63                            88
   HEMATOLOGY-ONCOLOGY CLINIC             HEPATOLOGY CLINIC
                           24                           217
    INFECTIOUS DISEASE CLINIC       INTERNAL MEDICINE CLINIC
                          265                           276
INTERNATIONAL MEDICINE CLINIC            NEIGHBORHOOD CLINIC
                           72                           187
            NEPHROLOGY CLINIC            NEUROSURGERY CLINIC
```

```
                     267                          141
            OB GYN CLINIC           OPHTHAMOLOGY CLINIC
                      86                           13
        PEDIATRIC CLINICS           RHEUMATOLOGY CLINIC
                       7                            2
TRANSITIONAL CARE CLINIC
                      27
```

```
# recall the dollar sign syntax to indicate a variable from an object
```

Now let's generate a more complex table to visualize the number of CBC orders by department AND split out by order status.

```
table(cbc_orders$department, cbc_orders$order_status_c_descr)
```

```
                              Canceled Completed Sent
  BEHAVIORAL HEALTH CLINIC           2         2    0
  CARDIOLOGY CLINIC                 27        43    0
  ENDOCRINOLOGY CLINIC               5        17    0
  FAMILY MEDICINE CLINIC            12       136    0
  GASTROENTEROLOGY CLINIC           26        37    0
  GERIATRIC CLINIC                  15        73    0
  HEMATOLOGY-ONCOLOGY CLINIC        10        13    0
  HEPATOLOGY CLINIC                 56       161    0
  INFECTIOUS DISEASE CLINIC         51       214    0
  INTERNAL MEDICINE CLINIC          46       230    0
  INTERNATIONAL MEDICINE CLINIC      8        64    0
  NEIGHBORHOOD CLINIC               24       163    0
  NEPHROLOGY CLINIC                109       157    1
  NEUROSURGERY CLINIC               20       120    1
  OB GYN CLINIC                     11        74    1
  OPHTHAMOLOGY CLINIC                3        10    0
  PEDIATRIC CLINICS                  2         5    0
  RHEUMATOLOGY CLINIC                0         2    0
  TRANSITIONAL CARE CLINIC           6        21    0
```

**Exercise 5:**

Let's practice making tables. This time, create a table showing the breakdown of CBC with differential (procedure code CBD) by department and order class. Which clinics draw a majority of their own samples? Which clinics use the most external orders (ie. use results from outside labs)?

```
                              Clinic Collect External Historical Normal
  BEHAVIORAL HEALTH CLINIC                 0        0          0      1
  CARDIOLOGY CLINIC                        0        0          0     19
  ENDOCRINOLOGY CLINIC                     0        0          0      1
  FAMILY MEDICINE CLINIC                 107        0          0      2
  GASTROENTEROLOGY CLINIC                  0        0          0     32
  GERIATRIC CLINIC                         0        0          0     44
  HEMATOLOGY-ONCOLOGY CLINIC               0        0          0    121
  HEPATOLOGY CLINIC                        0        1          0     53
  INFECTIOUS DISEASE CLINIC                7       99          0    851
  INTERNAL MEDICINE CLINIC                 4        0          0    174
  INTERNATIONAL MEDICINE CLINIC            0        0          0    195
  NEIGHBORHOOD CLINIC                     75        0          0      7
```

```
  NEPHROLOGY CLINIC                          0        0          0     11
  NEUROSURGERY CLINIC                        0        0          0      6
  OB GYN CLINIC                              0        0          0     10
  OPHTHAMOLOGY CLINIC                        0        0          0    159
  PEDIATRIC CLINICS                          0        0          0     57
  RHEUMATOLOGY CLINIC                        0        8          0    330
  TRANSITIONAL CARE CLINIC                   0        0          0     19

                               On Site
  BEHAVIORAL HEALTH CLINIC           0
  CARDIOLOGY CLINIC                  0
  ENDOCRINOLOGY CLINIC               0
  FAMILY MEDICINE CLINIC             0
  GASTROENTEROLOGY CLINIC            0
  GERIATRIC CLINIC                   0
  HEMATOLOGY-ONCOLOGY CLINIC         0
  HEPATOLOGY CLINIC                  0
  INFECTIOUS DISEASE CLINIC          0
  INTERNAL MEDICINE CLINIC           0
  INTERNATIONAL MEDICINE CLINIC      0
  NEIGHBORHOOD CLINIC                0
  NEPHROLOGY CLINIC                  0
  NEUROSURGERY CLINIC                0
  OB GYN CLINIC                      0
  OPHTHAMOLOGY CLINIC                0
  PEDIATRIC CLINICS                  0
  RHEUMATOLOGY CLINIC                0
  TRANSITIONAL CARE CLINIC           0
```

**End Exercise**

### 5.4.1 Taking a quick look at your data with `skim()` (Optional)

The skimr package is worth knowing about. This does some work for you in breaking down distributions of different variables and showing the amount of missing data.

```
#install.packages("skimr")
library(skimr)
skim(orders)
```

### 5.4.2 Creating more complicated tables (Optional)

We used the janitor package earlier to help clean up variable (column) names. This package also includes helpful functions for tabulating. The basic `tabyl` funcationality works similarly to the `table` function in base R, but the syntax of the arguments is different. Rather than explicitly calling the combination of object and variable name (indicated by the "$"), the `tabyl` function adopts the syntax we've seen before where the first argument is the object (tibble in this case) that can be piped in, and the variables can serve as inputs to the function without calling the name of the object.

```
cbc_orders %>%
  tabyl(department, order_status_c_descr)
```

```
                department Canceled Completed Sent NA_
  BEHAVIORAL HEALTH CLINIC        2         2    0   0
```

```
                    CARDIOLOGY CLINIC       27        43    0   0
                  ENDOCRINOLOGY CLINIC       5        17    0   0
                FAMILY MEDICINE CLINIC       12       136    0   0
              GASTROENTEROLOGY CLINIC       26        37    0   0
                     GERIATRIC CLINIC       15        73    0   0
           HEMATOLOGY-ONCOLOGY CLINIC       10        13    0   1
                   HEPATOLOGY CLINIC       56       161    0   0
            INFECTIOUS DISEASE CLINIC       51       214    0   0
             INTERNAL MEDICINE CLINIC       46       230    0   0
         INTERNATIONAL MEDICINE CLINIC       8        64    0   0
                  NEIGHBORHOOD CLINIC       24       163    0   0
                    NEPHROLOGY CLINIC      109       157    1   0
                  NEUROSURGERY CLINIC       20       120    1   0
                        OB GYN CLINIC       11        74    1   0
                  OPHTHAMOLOGY CLINIC        3        10    0   0
                    PEDIATRIC CLINICS        2         5    0   0
                  RHEUMATOLOGY CLINIC        0         2    0   0
               TRANSITIONAL CARE CLINIC        6        21    0   0
```

The basic functionality of `tabyl` does not improve much over the base `table` function, but there are a series of helper functions beginning with `adorn_` that can modify the table output to be something more helpful. In the example below, we use `adorn_percentages` to convert from raw counts to percentages calculated across the rows. We also include the raw counts using the `adorn_ns()` function to see the full data set.

```
cbc_orders %>%
  tabyl(department, order_status_c_descr) %>%
  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)
```

```
                    department    Canceled     Completed     Sent      NA_
        BEHAVIORAL HEALTH CLINIC 50.0%   (2)  50.0%    (2) 0.0% (0) 0.0% (0)
                CARDIOLOGY CLINIC 38.6%  (27)  61.4%   (43) 0.0% (0) 0.0% (0)
              ENDOCRINOLOGY CLINIC 22.7%   (5)  77.3%   (17) 0.0% (0) 0.0% (0)
            FAMILY MEDICINE CLINIC  8.1%  (12)  91.9%  (136) 0.0% (0) 0.0% (0)
          GASTROENTEROLOGY CLINIC 41.3%  (26)  58.7%   (37) 0.0% (0) 0.0% (0)
                 GERIATRIC CLINIC 17.0%  (15)  83.0%   (73) 0.0% (0) 0.0% (0)
       HEMATOLOGY-ONCOLOGY CLINIC 41.7%  (10)  54.2%   (13) 0.0% (0) 4.2% (1)
                HEPATOLOGY CLINIC 25.8%  (56)  74.2%  (161) 0.0% (0) 0.0% (0)
         INFECTIOUS DISEASE CLINIC 19.2%  (51)  80.8%  (214) 0.0% (0) 0.0% (0)
          INTERNAL MEDICINE CLINIC 16.7%  (46)  83.3%  (230) 0.0% (0) 0.0% (0)
     INTERNATIONAL MEDICINE CLINIC 11.1%   (8)  88.9%   (64) 0.0% (0) 0.0% (0)
              NEIGHBORHOOD CLINIC 12.8%  (24)  87.2%  (163) 0.0% (0) 0.0% (0)
                NEPHROLOGY CLINIC 40.8% (109)  58.8%  (157) 0.4% (1) 0.0% (0)
              NEUROSURGERY CLINIC 14.2%  (20)  85.1%  (120) 0.7% (1) 0.0% (0)
                    OB GYN CLINIC 12.8%  (11)  86.0%   (74) 1.2% (1) 0.0% (0)
              OPHTHAMOLOGY CLINIC 23.1%   (3)  76.9%   (10) 0.0% (0) 0.0% (0)
                PEDIATRIC CLINICS 28.6%   (2)  71.4%    (5) 0.0% (0) 0.0% (0)
              RHEUMATOLOGY CLINIC  0.0%   (0) 100.0%    (2) 0.0% (0) 0.0% (0)
           TRANSITIONAL CARE CLINIC 22.2%   (6)  77.8%   (21) 0.0% (0) 0.0% (0)
                             Total 21.9% (433)  77.9% (1542) 0.2% (3) 0.1% (1)
```

The janitor package is very useful for cleaning up dirty, manually curated spreadsheets. You can read more about it here.

## 5.5 Simple summary visualizations

One of R's greatest strengths is its ability to create complex visualizations without writing a large amount of complex code. When we reviewed our method validation data, we were oriented to the ggplot2 package. With our orders data, we can dive into how plots using this package work.

The general syntax for plotting with ggplot follows this structure:

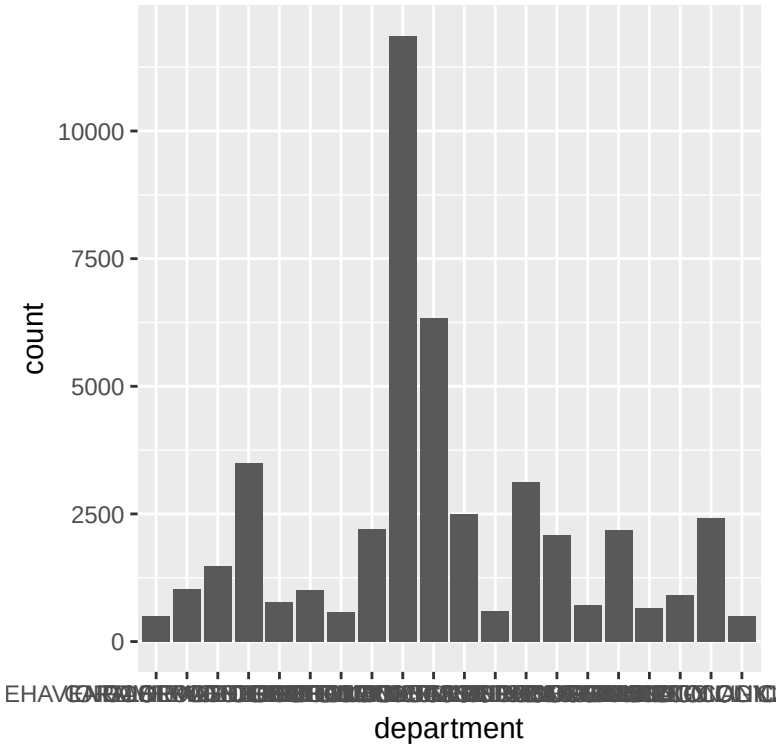
```
ggplot(data = <tibble>) +
  <geom_function>(mapping = aes(<mapping(s)>))
```

The data is expected to be in a tibble (data frame) where each row represents a unique observation and each column represents a distinct variable ("tidy" data - for more on tidy data info read here). Code for plotting with this package includes the following components:

1. The `ggplot()` function identifies the data set you want to plot
2. A variety of different geom functions produce the visualizations that you layer on top of your data set
3. The input for these geom functions is a mapping of which variables to plot

Let's look at a simple visualization of our orders data set, so our orders data set is the input for the `ggplot()` function. We are interested in the breakdown of order volumes by department. One straight-forward visualization is a bar chart, so we use the `geom_bar()` function and provide the department as the input into the `aes()`, which is an aesthetic. In this case, we provide department as the x-axis. Note that the geom function follows the `ggplot()` function after a "+" (this is a distinct situation from using pipes).

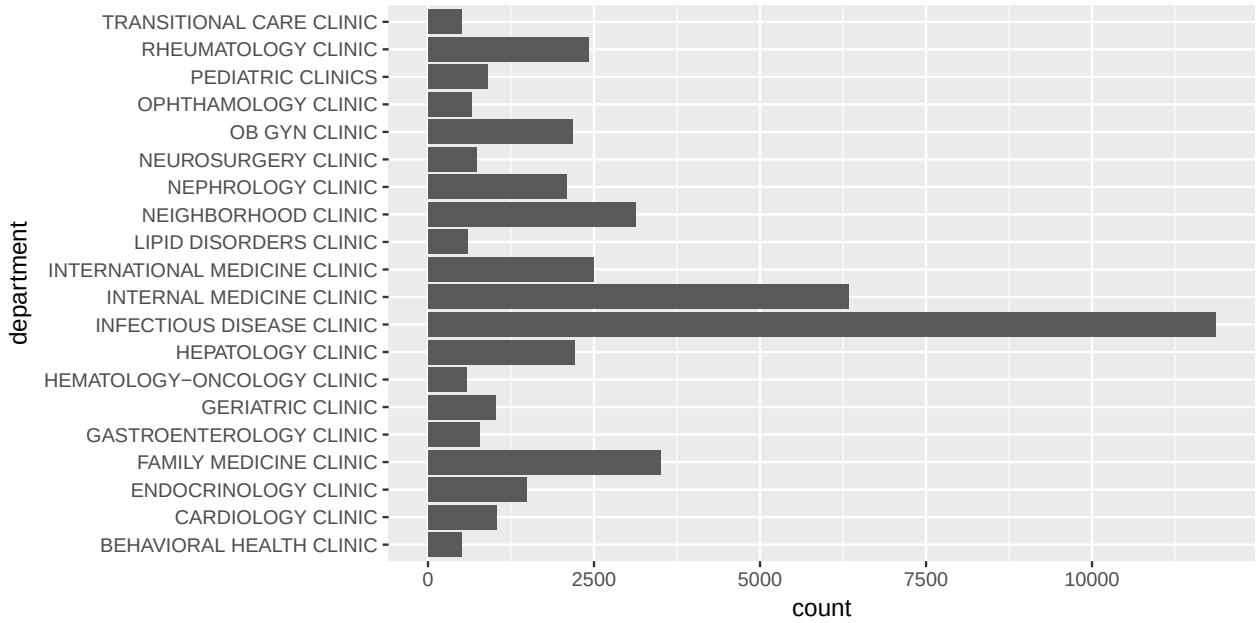
```
ggplot(data = orders) +
  geom_bar(mapping = aes(x = department))
```




The definitely shows the data, but the x-axis is not readable because there are so many categories. The easiest solution is to flip the axes using the `coord_flip()` function.

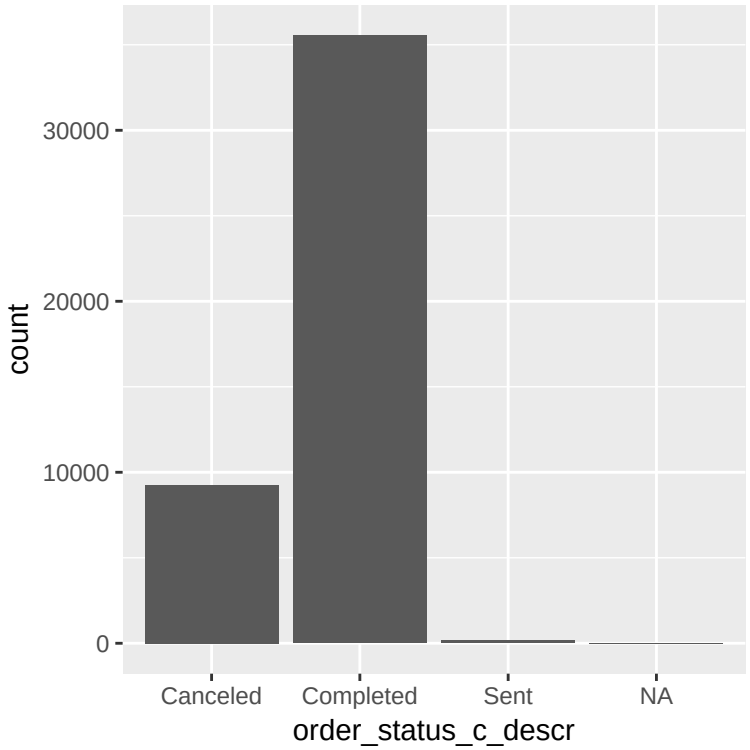
```
ggplot(data = orders) +
  geom_bar(mapping = aes(x = department)) +
```

```
coord_flip()
```

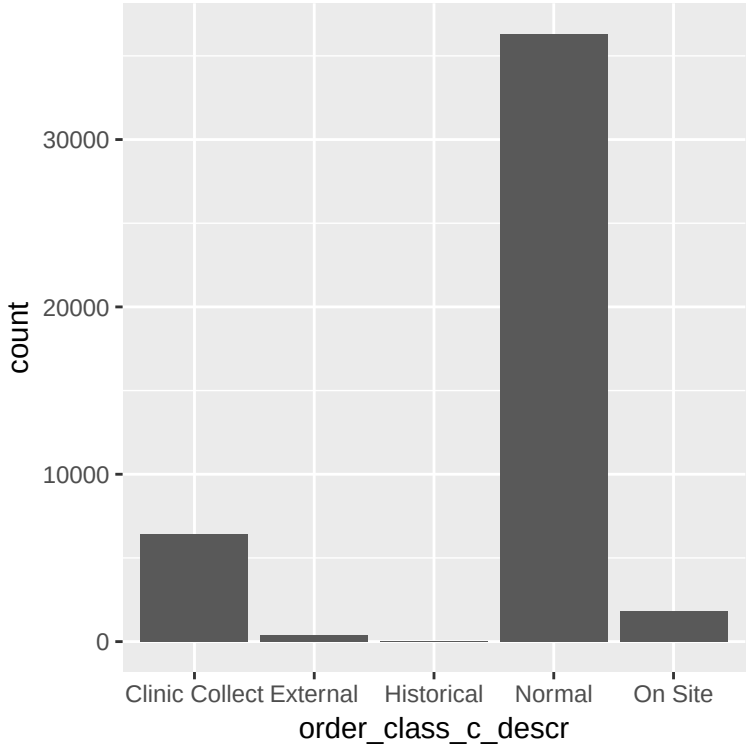



**Exercise 6:**

1. Plot a bar graph of the orders data set showing the breakdown of order status.



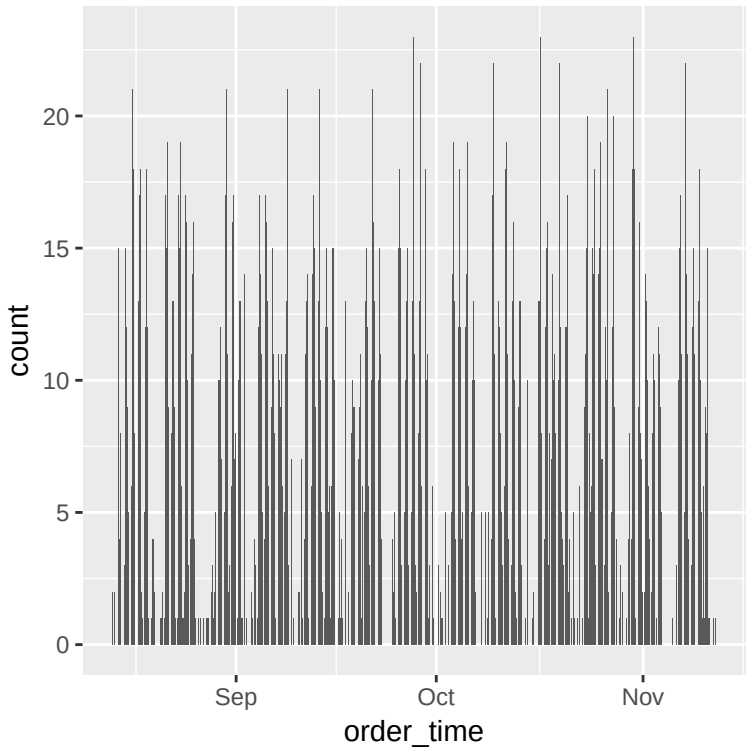

2. Plot a bar graph showing the breakdown of order class.




**End Exercise**

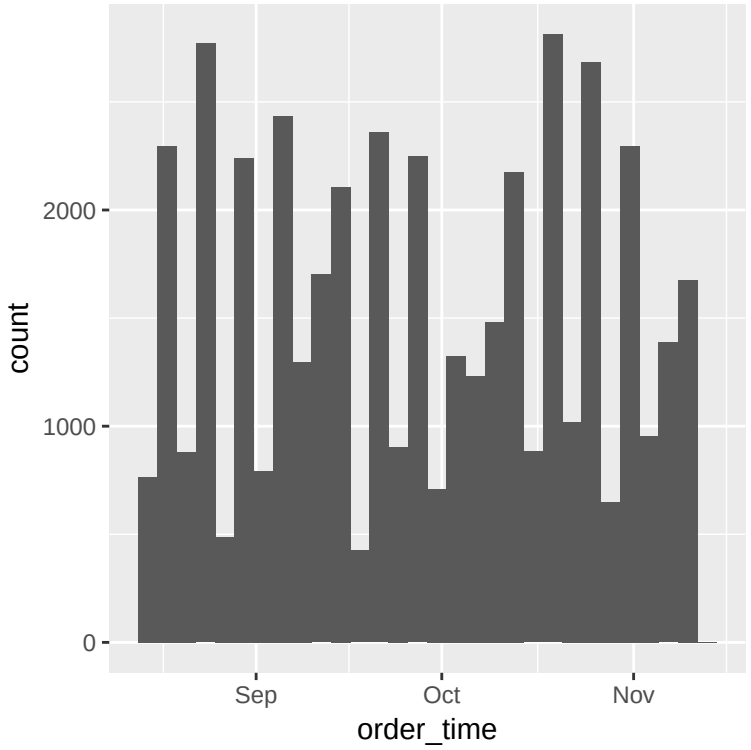
Plotting order volumes as a function of time can be helpful across a variety of settings: monitoring a utilization intervention, analyzing seasonality for specific tests, creating projections for workload, etc. We can revisit the bar graph but map the time of order to the x-axis.

```
ggplot(data = orders) +
  geom_bar(mapping = aes(x = order_time))
```




This bar chart looks very uneven, with bars that are spread far apart. Let's try a geom function that is related to `geom_bar()` but intended to visualize distributions of continuous variables like time: `geom_histogram()`.

```
ggplot(data = orders) +
  geom_histogram(mapping = aes(x = order_time))
```

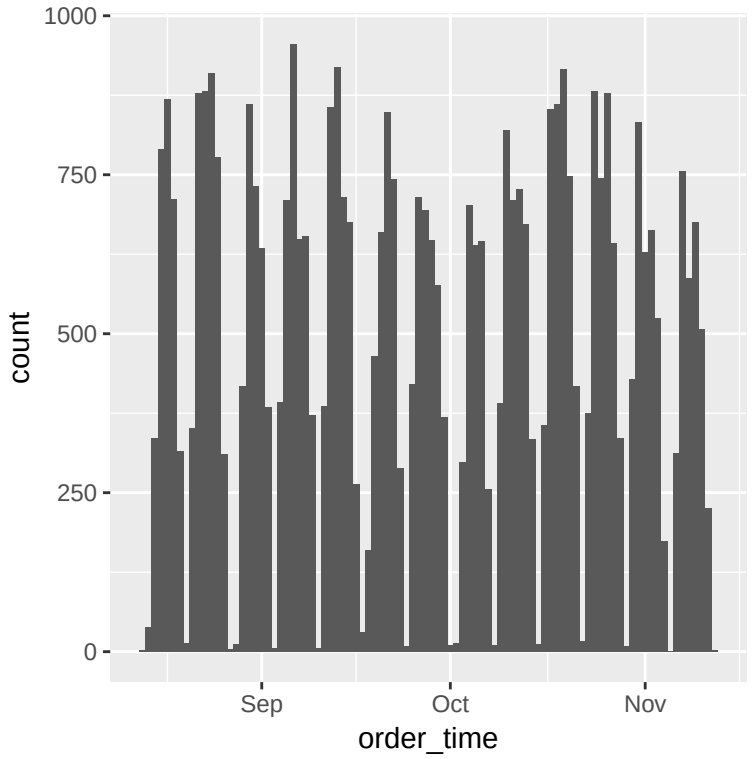



Note that R output a warning: "`stat_bin()` using `bins = 30`. Pick better value with `binwidth`." By default

`geom_histogram()` splits the continous variable into 30 bins, which makes the bars look very uneven over time. That split is arbitrary and probably does not reflect the true distribution of data.

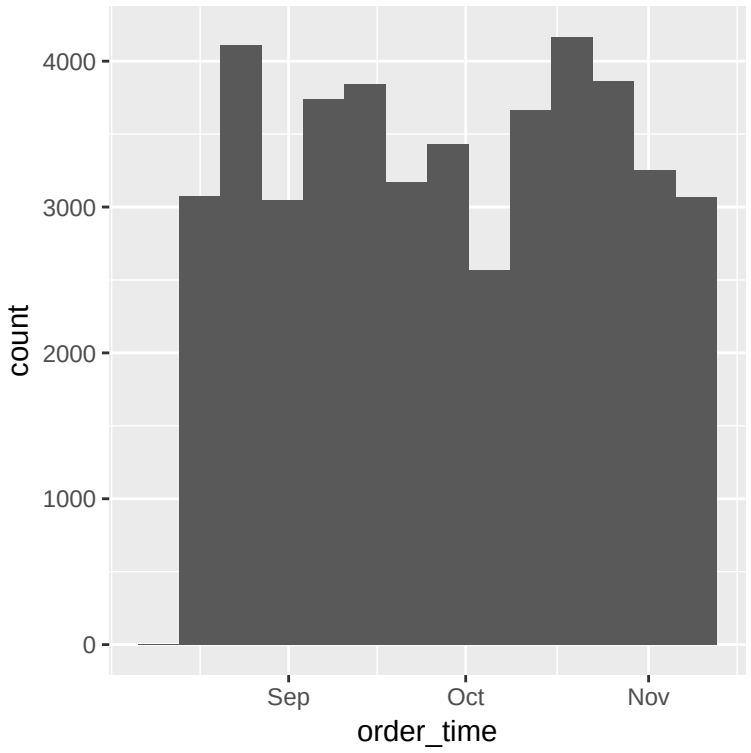
How do we fix this? The `geom_histogram()` function uses seconds as its default unit of time (but will use days if the variable is a date rather than a time), so we can define a binwidth that is multiples of seconds to reflect days or weeks. For example, one day = 60 *60 * 24 seconds.

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



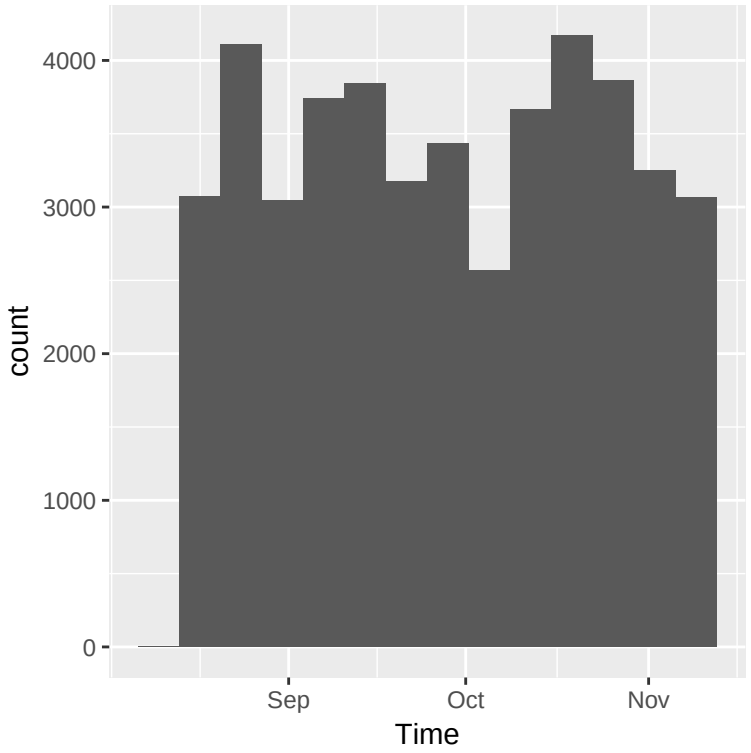

That pattern looks more accurate, with decreased volumes occuring on weekends. Alternately, we can look at volumes by week instead of by day.

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



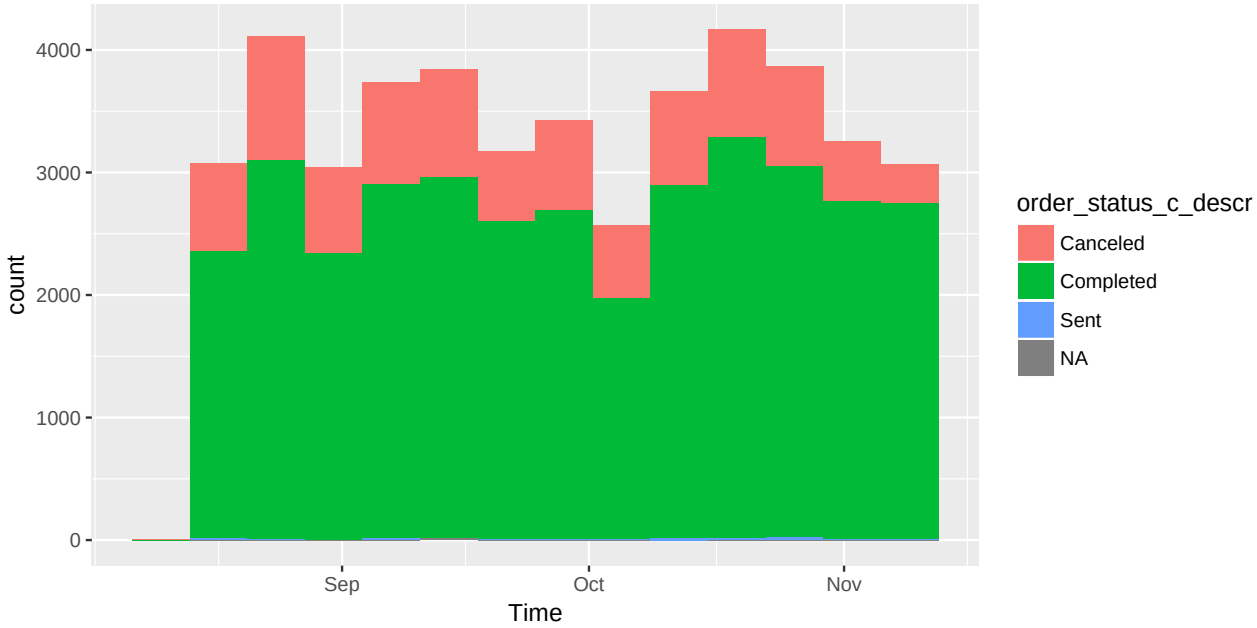

What if we want to change the x-axis label of our plot? We simply add another layer called `xlab()` and add the string we want to insert.

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



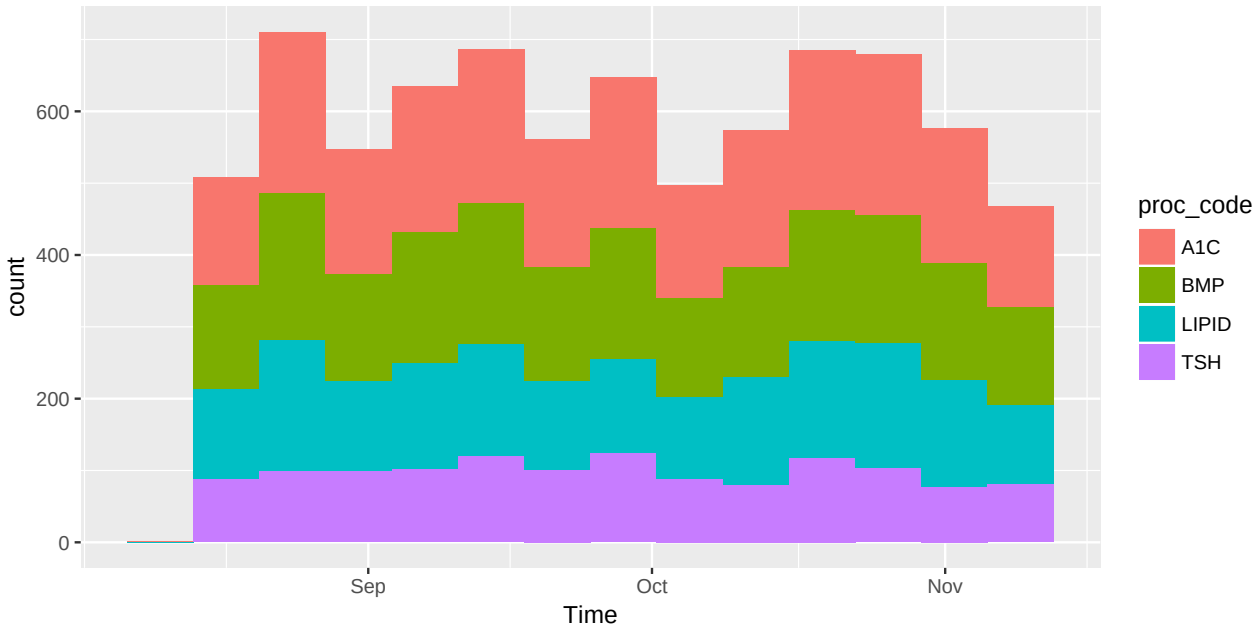
Now let's add another mapping to our plot. We may want to analyze the number of orders that are cancelled and visualize that alongside the total number of orders. We simply add another variable to the aesthetic function, and instead of plotting against an axis, we are going to fill in the bars with colors based on order status.

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




**Exercise 7:**

Plot the weekly volume of a subset of common orders: basic metabolic panel, hemoglobin A1C, lipid panel, and TSH (proc_codes BMP, A1C, LIPID, TSH) and show the breakdown of tests by fill in the bar chart.




**End Exercise**

# 6 Exploratory Data Analysis: Plotting Time Data and Joining Data Sets

In this section, we will build on the previous section of exploring a data set. We will introduce the concept of working with time data including calculating time differences. This is very useful when working with clinical laboratory data (think turnaround times). At the end of the lesson we will also discuss the basics of joining separate data sets together using 'join' functions.

Let's get started by loading the orders_data_set (which we used in the last lesson). We will use the `read_excel` function and couple that to the `clean_names` function to tidy up those column names.

```
orders <- read_excel("data/orders_data_set.xlsx") %>%
clean_names()
```

Now let's use `glimpse` to take a peak at our data.

```
glimpse(orders)
```

```
Observations: 45,002
Variables: 15
$ order_id                <dbl> 19766, 88444, 40477, 97641, 99868, 311...
$ patient_id              <dbl> 511388, 511388, 508061, 508061, 505646...
$ description             <chr> "PROTHROMBIN TIME", "BASIC METABOLIC P...
$ proc_code               <chr> "PRO", "BMP", "TSH", "T4FR", "COMP", "...
$ order_class_c_descr     <chr> "Normal", "Normal", "Normal", "Normal"...
$ lab_status_c            <dbl> NA, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...
$ lab_status_c_descr      <chr> NA, NA, "Final result", "Final result"...
$ order_status_c          <dbl> 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,...
$ order_status_c_descr    <chr> "Canceled", "Canceled", "Completed", "...
$ reason_for_canc_c       <dbl> 11, 11, NA, NA, NA, NA, NA, NA, NA, NA...
$ reason_for_canc_c_descr <chr> "Auto-canceled. Patient no show and/or...
$ order_time              <dttm> 2017-08-13 11:59:00, 2017-08-13 11:59...
```

```
$ result_time             <dttm> NA, NA, 2017-09-20 11:59:00, 2017-09-...
$ review_time             <dttm> NA, NA, 2017-09-21 20:34:00, 2017-09-...
$ department              <chr> "INTERNAL MEDICINE CLINIC", "INTERNAL ...
```

So far we have worked with data types that include numbers and characters. Take a look at order_time, result_time, and review_time in our glimpse. They have a different data type . This stands for date time. When we loaded the orders data R recognized this data as having the format of a date and time. Having our data classified as date times gives us additional functionality to work with those data elements in interesting ways. We can isolate components of a date and time, or perform date math (add and subtract dates from one another).

We can select just the date component of a date time:

```
date_of_order <- date(orders$order_time)

print('Date of order:')
```

```
[1] "Date of order:"
```

```
head(date_of_order)
```

```
[1] "2017-08-13" "2017-08-13" "2017-08-13" "2017-08-13" "2017-08-14"
[6] "2017-08-14"
```

Or the week component:

```
week_of_order <- week(orders$order_time)

print('Week of order:')
```

```
[1] "Week of order:"
```

```
head(week_of_order)
```

```
[1] 33 33 33 33 33 33
```

Here, the number "33" represents the 33rd week of the year.

**Exercise 1:**

Can you isolate the day of the week from the order? Hint: use the `wday` function.

```
day_of_week_of_order <- wday(orders$order_time)

print('Day of Week of Order:')
```

```
[1] "Day of Week of Order:"
```

```
head(day_of_week_of_order)
```

```
[1] 1 1 1 1 2 2
```

Here, the number "1" represents a Sunday.

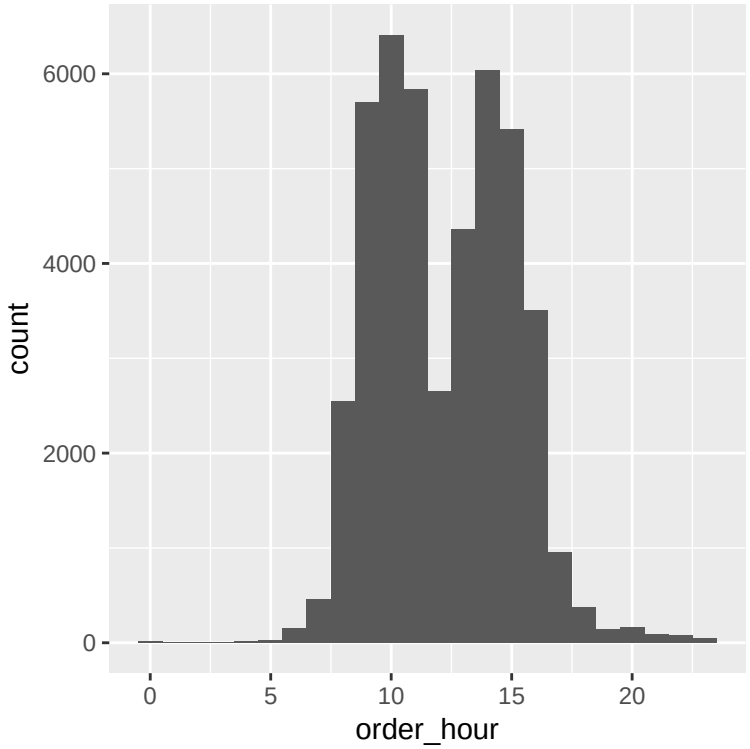
For more interesting date time functions, check out the R Studio Dates and Times Cheat Sheet.

**End exercise**

## 6.1 Plotting Time Points for Data Exploration

Let's use the `hour` function to isolate the hour component of our order and then use `ggplot` to visualize the data. This will give us a window into the time of day that ordering providers are placing their orders.

```
order_hour <- hour(orders$order_time)

ggplot(data = orders) +
  geom_histogram(mapping = aes(x = order_hour), bins = 24)
```

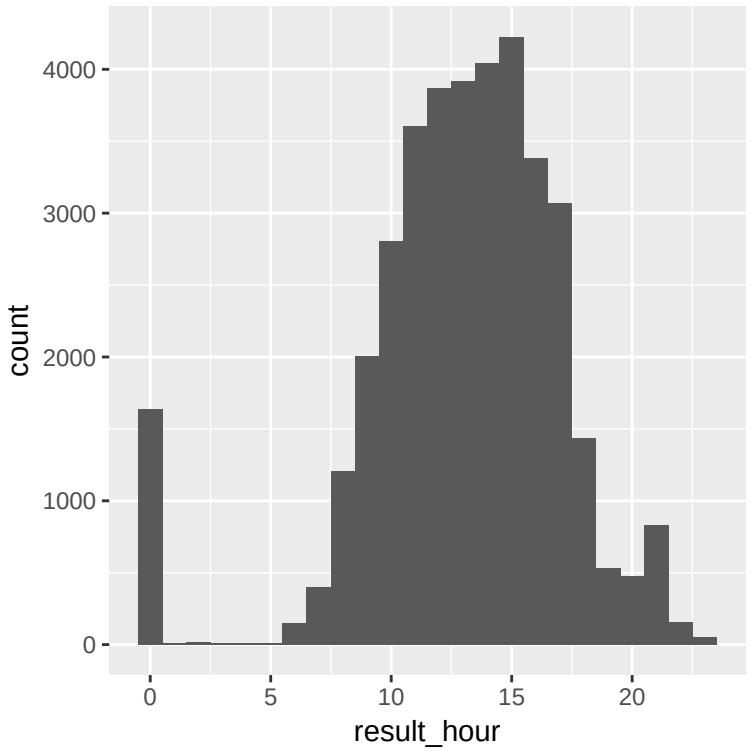



Our data visualization shows that most orders are placed in the mid-morning and mid-afternoon with a break for lunch. Also, as we might expect there aren't a lot of orders placed in the middle of the night.

**Exercise 2:**

Can you perform a similar analysis and visualization to look at the result_time data using the `hour` function and `ggplot`?

```
result_hour <- hour(orders$result_time)

ggplot(data = orders) +
  geom_histogram(mapping = aes(x = result_hour), bins = 24)
```




It looks like the lab gets busy issuing results around 5 in the morning and that the evening rush begins to tail off around 9-10 at night.

**End exercise**

## 6.2 Exploring Time Intervals and Plotting the Data

So far in this lesson, we have looked at single time points. It's also very useful to calculate and plot time differences. Let's illustrate this concept by calculating the difference between the order and result times (turnaround time)

Not all tests that are ordered are completed. We can look at a `summary` of the order status get a sense for this.

```
summary(as.factor(orders$order_status_c_descr))
```

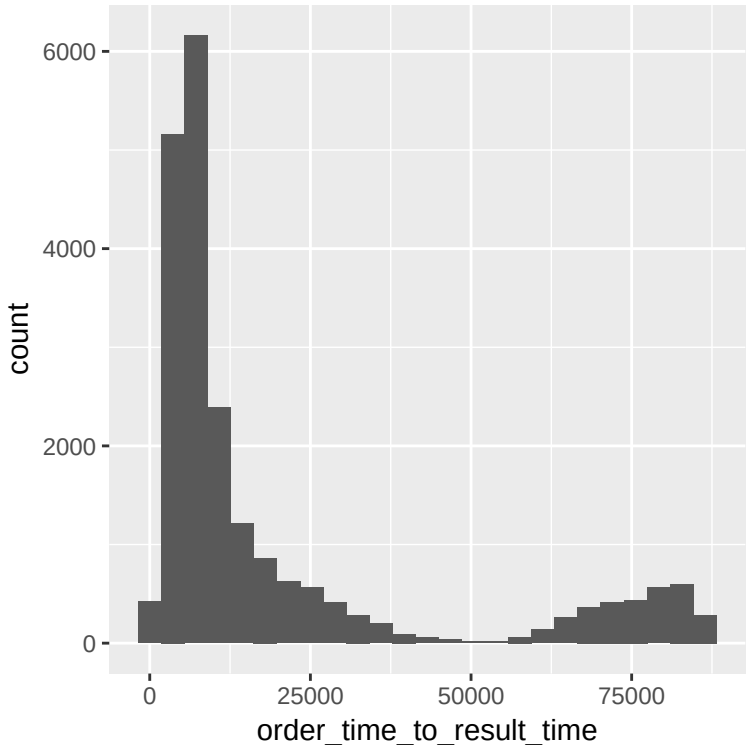
```
 Canceled Completed      Sent      NA's
     9270     35553       161        18
```

Results that aren't completed won't have a result time so we should `filter` our data first to only "Completed results". That will allow us to work only with test results that have both an order time and result time.

```
completed_orders <- orders %>%
  filter(order_status_c_descr == "Completed")
```

Now that we are working with completed orders only, we can add a column to our data frame and populate it with the difference between the order and result times. We will use the `mutate` function to create the column and calculate the time difference. We will also use the `filter` function to limit the window we are looking at to 24 hours (60 seconds x 60 minutes x 24 hours). We are limiting our time window because the data has a long tail (think sendouts) that we will exclude so the plot is easier to read. Then we can use `ggplot` to plot the data.

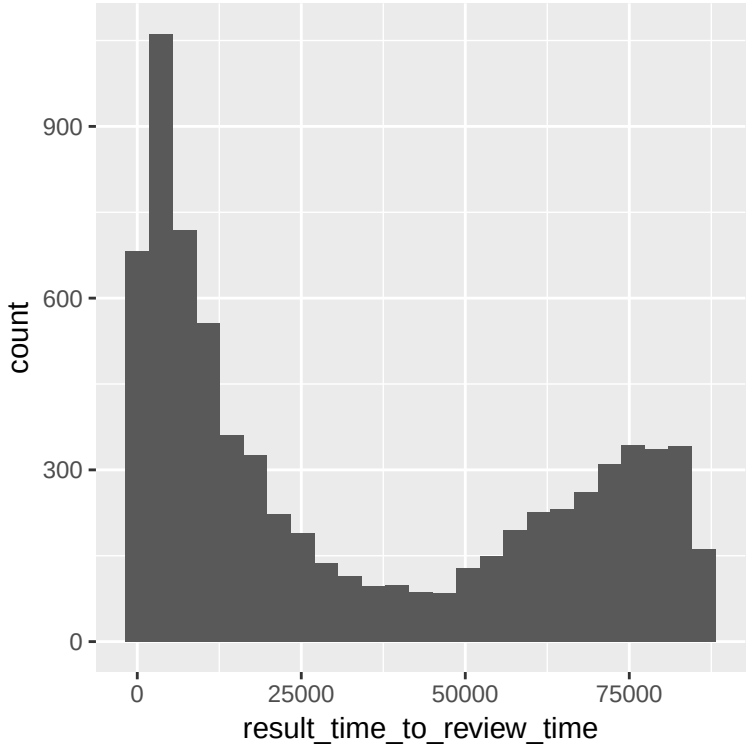
```
order_to_result_delta <- completed_orders %>%
  mutate(order_time_to_result_time = result_time - order_time) %>%
  filter(order_time_to_result_time > 0 & order_time_to_result_time < 60*60*24)


ggplot(data = order_to_result_delta) +
  geom_histogram(mapping = aes(x = order_time_to_result_time), binwidth = 60*60)
```




**Exercise 3:**

What if we wanted to look at result time to review time instead? Can you leverage the same approach as above to calculate and plot the difference between review time and the result time? This will give us a look at the time between when results are available and when those results are reviewed by the ordering provider.

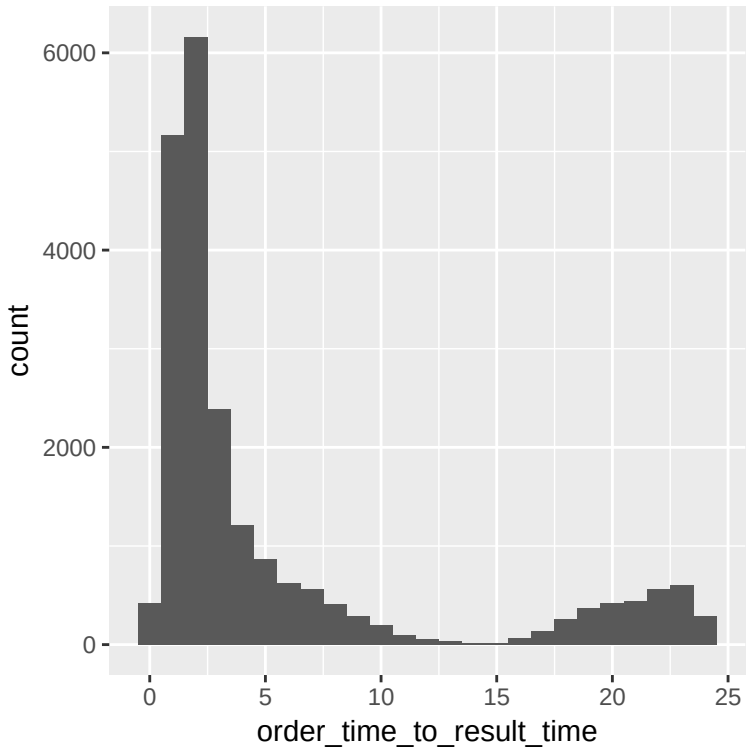
```
result_to_review_delta <- completed_orders %>%
  mutate(result_time_to_review_time = review_time - result_time) %>%
  filter(result_time_to_review_time > 0 & result_time_to_review_time < 60*60*24 )

ggplot(data = result_to_review_delta) +
  geom_histogram(mapping = aes(x = result_time_to_review_time), binwidth = 60*60)
```




**End exercise**

Note that the default date difference calculation is in seconds. We can make some changes to our code to look a this data in other time components (such as hours). The `difftime` function allows us to specify a unit, for example hours in the example below. We also have to make some changes to our `filter` logic and `ggplot` binwidth criteria to reflect that we are working with hours now and not seconds.

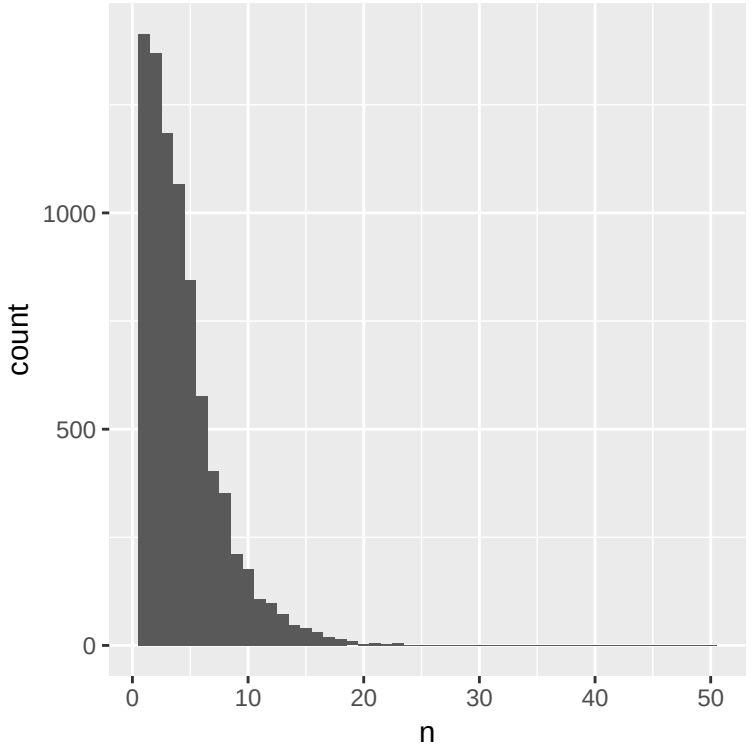
```
order_to_result_delta <- completed_orders %>%
  mutate(order_time_to_result_time = difftime(result_time, order_time, units = "hour")) %>%
  filter(order_time_to_result_time > 0 & order_time_to_result_time < 24)

ggplot(data = order_to_result_delta) +
  geom_histogram(mapping = aes(x = order_time_to_result_time), binwidth = 1)
```




## 6.3 Counting Tests Per Patient

We will return to date time calculations in a bit but first let's quickly revisit tabulating data. `Count` is a useful function for tabulation. Say we wanted to know the number of completed tests per patient in our data set. Since our data set has one test per patient per line, we can count the frequency of a patient's record number (MRN) as measure of the number of orders per patient.

```
orders_count <- completed_orders %>%
  count(patient_id)

ggplot(data = orders_count) +
  geom_histogram(mapping = aes(x = n), binwidth = 1)
```

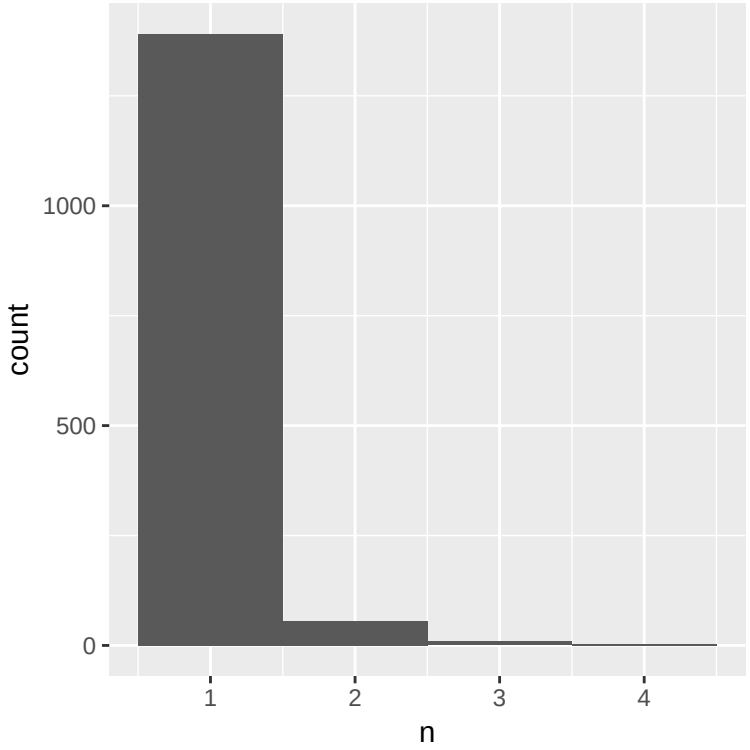



The vast majority of patient's received 10 or less tests during the time window that this data set covers. It looks like there are a few patients that received as many as 50 tests.

**Exercise 4:**

What if we wanted to know only number of CBCs per patient? Can you use the `filter` function to limit the data CBC orders before counting?

```
cbc_orders <- filter(completed_orders, proc_code == "CBC")

#Group and Count

cbc_count <- cbc_orders %>%
count(patient_id)

#Plot

ggplot(data = cbc_count) +
  geom_histogram(mapping = aes(x = n), binwidth = 1)
```




**End exercise**

## 6.4 Date Differential by Patient and Test

Now that we have refreshed our tabulation skills, let's revist working with dates. Sometimes it is useful to look at the interval between orders for a given test. Let's look at the time interval between CBCs for individual patients in our data set.

To measure the difference in time between CBCs for a given patient we need to first make a combination of all of a patient's CBCs with all of the patient's other CBCs. The `join` function allows us to join a data set to itself or another data set using a shared key, an attribute (or column) shared between the two data sets. The nuances of joins exceed the scope of an introductory coding course, but we provide two examples below as templates for your future analysis. If you would like to read more on joins, the "R for Data Science" book has a nice chapter on Relational Data.

Let's create the combination of patient CBCs, calculate the difference between CBC times, and display the output.

```
cbc_orders <- filter(completed_orders, proc_code == "CBC")

cbc1 = select(cbc_orders, patient_id, result_time)
cbc2 = select(cbc_orders, patient_id, result_time)

cbc_join <- full_join(cbc1, cbc2, by= "patient_id")

cbc_join <- cbc_join %>%
  mutate(cbc_time_diff = difftime(result_time.x, result_time.y, units = "days")) %>%
  filter(cbc_time_diff > 0)

cbc_join <- cbc_join %>%
  group_by(patient_id, result_time.x) %>%
```

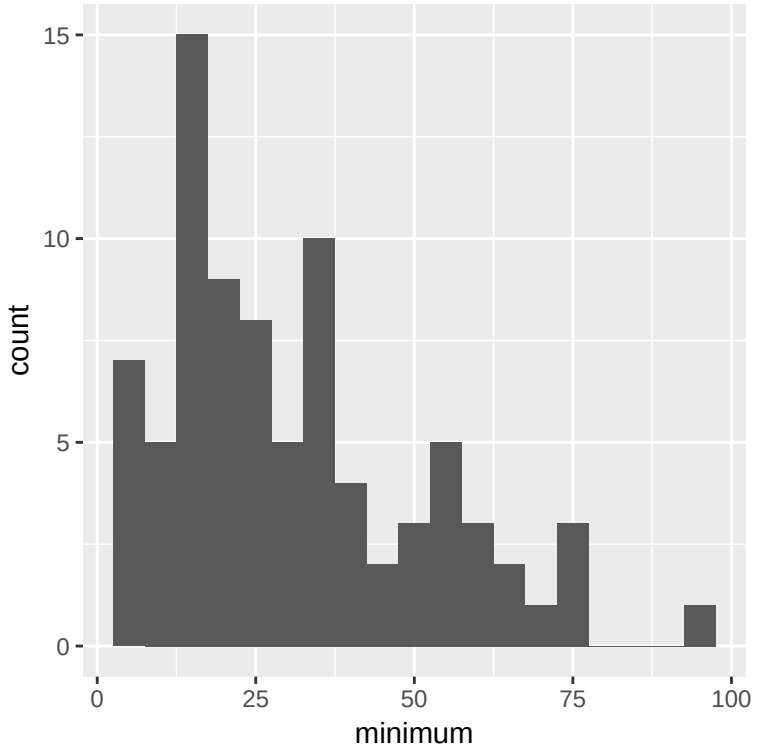
```
  summarize(minimum = min(cbc_time_diff))

cbc_join
```

```
# A tibble: 83 x 3
# Groups:   patient_id [?]
   patient_id result_time.x       minimum
        <dbl> <dttm>              <time>
 1    500249. 2017-10-11 13:19:00 34.1006944444444
 2    500385. 2017-10-11 10:45:00 29.8298611111111
 3    500651. 2017-12-07 09:41:00 96.9291666666667
 4    500763. 2017-09-08 10:05:00 8.74583333333333
 5    500801. 2017-10-12 13:30:00 57.9868055555556
 6    501216. 2017-10-31 09:54:00 53.9777777777778
 7    501250. 2017-09-15 13:19:00 27.89375
 8    501316. 2017-10-06 09:21:00 15.0145833333333
 9    501575. 2017-10-16 16:13:00 27.1631944444444
10    501575. 2017-10-31 12:13:00 14.8333333333333
# ... with 73 more rows
```

This gives us the patient's record number, the result time of the CBC, and the amount of time that has elapsed since the patient's previous CBC.

Now let's plot the intervals using our old friend 'ggplot'.

```
ggplot(data = cbc_join) +
  geom_histogram(mapping = aes(x = minimum), binwidth = 5)
```




We can see that intervals between CBCs range from 1 to nearly 100 days in this data set, but most repeat CBCs are performed before ~45 days have elapsed.

## 6.5 Bringing different data sets together:

The `join` function can also help us to bring two different data sets together for analysis using a shared key variable.

Let's load a second data set called order_details. This data set includes information about the ordering route and whether the order originiated from a preference list. This data set is a Comma Separated Variable (CSV) file so we need to use the `read_csv` function instead of our usual `read_excel` function.

```
orders_details <- read_csv("data/order_details.csv")

head(orders_details)
```

```
# A tibble: 6 x 3
  order_id ordering_route      pref_list_type
     <int> <chr>               <chr>
1    19766 OP Orders Navigator Clinic Preference List
2    88444 OP Orders Navigator Clinic Preference List
3    40477 OP Orders Navigator Provider Preference List
4    97641 OP Orders Navigator Provider Preference List
5    99868 OP Orders Navigator Provider Preference List
6    31178 OP Orders Navigator Provider Preference List
```

We can join the order details data to our completed orders data using the order_id as the shared key for linking the data sets.

```
orders_details <- read_csv("data/order_details.csv")

joined_data <- completed_orders %>%
  left_join(orders_details, by = "order_id") %>%
  count(pref_list_type)

joined_data
```

```
# A tibble: 3 x 2
  pref_list_type              n
  <chr>                   <int>
1 Clinic Preference List   25030
2 None                      4418
3 Provider Preference List  6105
```

We can see that most completed orders originate from clinic prefernce lists, followed by individual provider preference lists, and then from searching the test menu in the electronic medical record (the "None" category)

What if we just wanted to look at preference list types for CBC? We need to use the `filter` function and the proc_code of "CBC".

```
orders_details <- read_csv("data/order_details.csv")

CBC <- completed_orders %>%
  filter(proc_code == "CBC") %>%
  left_join(orders_details, by = "order_id") %>%
  count(pref_list_type)

CBC
```

```
# A tibble: 3 x 2
  pref_list_type              n
```

```
  <chr>                    <int>
1 Clinic Preference List    1249
2 None                        49
3 Provider Preference List   244
```

**Exercise 5:**

What if we just wanted to look at Factor V Leiden. Hint: proc_code = "F5DNA")

```
orders_details <- read_csv("data/order_details.csv")

F5DNA <- completed_orders %>%
  filter(proc_code == "F5DNA") %>%
  left_join(orders_details, by = "order_id") %>%
  count(pref_list_type)

F5DNA
```

```
# A tibble: 2 x 2
  pref_list_type             n
  <chr>                  <int>
1 Clinic Preference List     3
2 None                       1
```

**End exercise**

# 7 Next Steps for Learning R

This course is a collection of knowledge from the instructors that have largely been learned/inspired/borrowed from other resources. By applying our knowledge to common lab workflows, we have attempted to show you the value of using R (or any other programming language) to accomplish important and common data analysis tasks. The internet contains a vast number of resources, including free online texts. In addition there are online courses that may be helpful. Below are a handful of resources that can serve as references as you continue to develop your skills. As an obligatory disclaimer, this is not a comprehensive list and links may not work indefinitely.

## 7.1 Online Books and Quick Reference Resources

- RStudio Cheat Sheets: https://www.rstudio.com/resources/cheatsheets/

  Super-helpful one pagers that illustrate useful functions and code covering a variety of R topics

- R For Data Science: http://r4ds.had.co.nz

  This is a core resource for this course and many others and is frequently consulted, even by veterans

- R Markdown: The Definitive Guide: https://bookdown.org/yihui/rmarkdown/

  Just published resource digging into the nitty gritty of R Markdown and creating great documents and presentations using R

## 7.2 Courses and Other Active Learning Resources

- Mass Spectrometry: Applications to the Clinical Lab (MSACL) Short Courses: https://www.msacl.org

Data Science Short Courses at US and Europe conferences - Basic and intermediate level courses with about 15 hours of content. (Advanced special topics course planned for 2019 US.)

- Swirl: https://swirlstats.com

  Interative R tutorials within the R console - nice refresher for this course content and there are more advanced lessons too!

- Coursera R courses: https://www.coursera.org/courses?query=r

  The Johns Hopkins series have historically been popular for learning R but there are courses from a variety of institutions

- Software Carpentry: https://software-carpentry.org/lessons/

  Research-focused computing lessons that cover variety of topics in addition to R (including Python and Linux)

# 8 Acknowledgements

In addition to the resources above, the authors acknowledge Daniel Holmes, Steve Master, Stephan Kadauke, and Keith Baggerly, who have provided materials, ideas, and great tips and discussions about teaching R programming.