Data Wrangling with R

Alexandra L Emmons, Ph.D.
BTEP/GAU/CCR/NCI/NIH - email ncibtep@mail.nih.gov
Bioinformatics Training and Education Program

Table of Contents

Course Overview

Lesson 1: Course Introduction

Lesson 2: Getting Started with R

Lesson 3: Importing and reshaping data

Lesson 4: Data Visualization with ggplot2

Lesson 5: dplyr and the pipe

Lesson 6: Continuing with dplyr

Lesson 7: Introduction to Bioconductor -omics classes (containers)

Lesson 8: Data Wrangling Review and Practice

Practice Questions

Additional Resources

Course Overview

Welcome to the Data Wrangling with R course series

The purpose of this course is to introduce you to essential R packages and functions that will make your life easier when it comes time to explore, clean, transform, and summarize your data. This course will include a series of lessons for scientists with little to no experience in R.

Course objectives

- Learn how to navigate RStudio.
- Learn how to load different types of data formats.
- Get acquainted with the tidyverse packages, especially `dplyr`.
- Become familiar with functions useful for cleaning, transforming, and summarizing data.

While this course will not make you an expert R programmer or full-fledged data analyst, it will help you learn how to analyze real-life, messy data and prepare it for visualization and further analyses.

Course Expectations

This course will include a series of eight, one hour lessons. Each lesson will be held virtually using the Webex platform on Mondays / Wednesdays at 1 pm. Lessons will immediately be followed by a one-hour help session. Help sessions will be structured around a set of practice problems for you to test your new skills. Though, we welcome all questions!

> **DNAnexus**
>
> You will not be able to practice or work through this documentation using DNAnexus outside of class hours. If you are using this documentation asynchronously, please install R and RStudio *(https://bioinformatics.ccr.cancer.gov/docs/rtools/index.html)*.

Lesson 1: Introduction to R, RStudio, and the Tidyverse

This will be a no coding introduction to R, RStudio, and the Tidyverse. In this lesson, we will review some of the advantages of using R for data analysis and will get you acquainted with the RStudio environment.

Lesson 2: Getting started with R

Lesson 2 will focus on some of the basics of R programming including naming and assigning R objects, recognizing and using R functions, understanding data types and classes, becoming familiar with the R programming syntax.

Lesson 3: Importing and reshaping data

In lesson 3, we will learn how to import simple and complex data and how to avoid common mistakes. We will also learn how to reshape data, for example, from wide to long format, with `tidyr`.

Lesson 4: Data Visualization with ggplot2

Lesson 4 will be a brief reprieve from data wrangling. In this lesson, we will learn the basics of plotting with `ggplot2`.

Lesson 5: Introducing dplyr and the pipe (Part 1)

In Lesson 5, we will learn how to improve code interpretability with the pipe `%>%` from the `magrittr` package. We will also learn how to merge and filter data frames.

Lesson 6: Introducing dplyr and the pipe (Part 2)

In Lesson 6, we will continue to wrangle data using `dplyr`. This lesson will focus on functions such as `group_by()`, `arrange()`, `summarize()`, and `mutate()`.

Lesson 7: Introduction to Bioconductor -omics classes (containers)

In this lesson, we will learn about specialized data containers / classes that are shared across Bioconductor packages. These classes allow us to store and easily manage multiple -omics types. We will discuss some of the properties of these classes and gain insight into how to access and subset the data stored within.

Lesson 8: Data Wrangling Review and Practice

In Lesson 8, we will review many of the important concepts we learned throughout the course. We will also practice using our skills together on a realistic data set.

Required Course Materials

To participate in this class you will need your government-issued computer and a reliable internet connection. You do not need to download or install any software to participate in the class.

Lesson 1: Course Introduction

R Crash Course: A few things to know before diving into wrangling

Learning the Basics

Objectives
1. Learn about R objects
3. Learn how to recognize and use R functions
4. Learn about data types and accessors

Console vs. Script

We are going to begin by working in our console. In general, the console is used to run R code. If we want to run code quickly or test code, the console is the place to do this. If we want to keep our code and have a record of what we have been running to rerun or reference in the future, we should use the code editor to build an R script.

R can be used like a calculator

Let's use the console to run some basic mathematical operations.

```
398 + 783
```

```
## [1] 1181
```

```
475 / 5
```

```
## [1] 95
```

```
2 * 8906
```

```
## [1] 17812
```

```
(1 + (5 ** 0.5))/2
```

```
## [1] 1.618034
```

As you can see, ** were used for exponentiation. There are a number of other special operators used to perform math in R. Refer to this chart from datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html)* or an overview of mathematical operators used in R.

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
a%/%b	integer division (division where the remainder is discarded)
a%%b	modulus (returns the remainder after division)

Creating an R Script

If all R could do was function as a calculator, it wouldn't be very useful. R can be used for powerful analyses and visualizations.

As we learn more about R and begin implementing our first commands, we will keep a record of our commands using an R script. Remember, good annotation is key to reproducible data analysis. An R script can also be generated to run on its own without user interaction.

To create an R script, click File > New File > R Script or click on the new document icon (paper with a +) and select R script. You can save your now Untitled script by selecting the floppy disk icon. Give your script a meaningful name so that you can identify what it contains when returning to it later. R scripts end in .R. Save your R script to your working directory, which will be the default location on RStudio Server.

> **Important**
>
> Scripts are ordered. Running commands out of order will cause confusion later when you try to reproduce a given analysis step.

R Objects

Now that we have an R script, let's begin to work with R objects. Everything assigned a value in R is technically an object. Mostly we think of R objects as something in which a method (or function) can act on; however, R functions, too, are R object. R objects are what gets assigned to memory in R and are of a specific type or class. Objects include things like vectors, lists, matrices, arrays, factors, and data frames. Don't get too bogged down by terminology. Many of these terms will become clear as we begin to use them in our code. In order to be assigned to memory, an r object must be created.

Therefore, objects are data structures with specific attributes and methods that can be applied to them.

Creating and deleting objects

To create an R object, you need a name, a value, and an assignment operator (e.g., `<-` or `=`) *(https://blog.revolutionanalytics.com/2008/12/use-equals-or-arrow-for-assignment.html)*. R is case sensitive, so an object with the name "FOO" is not the same as "foo".

Let's create a simple object and run our code.

To run our code, we have a number of options. First, you can use the `Run` button above. This will run highlighted or selected code. You may also use the source button to run your entire script. My preferred method is to use keyboard shortcuts. Move your cursor to the code of interest and use `command + enter` for macs or `control + enter` for PCs. If a command is taking a long time to run and you need to cancel it, use `control + c` from the command line or `escape` in RStudio. Once you run the command, you will see the command print to the console in blue followed by the output. ***You do not need to highlight code to run it. If you do highlight code, make sure you are highlighting everything you plan to run. Highlighting can be a great way to only test small sections of nested code.***

```
a<-1 #You can and should annotate your code with comments for better
a #Simply call the name of the object to print the value to the scre
```

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

In this example, "a" is the name of the object, 1 is the value, and `<-` is the assignment operator. We inspect objects simply by typing or running their name.

Naming conventions and reproducibility

There are rules regarding the naming of objects.

1. Avoid spaces or special characters EXCEPT '_' and '.'
2. No numbers or symbols at the beginning of an object name.

   For example:

```
1a<-"apples" # this will throw and error
1a
```

```
## Error: <text>:1:2: unexpected symbol
## 1: 1a
##      ^
```

   In contrast:

```
a<-"apples" #this works fine
a
```

```
## [1] "apples"
```

   What do you think would have happened if we didn't put 'apples' in quotes? Try it.

```
a<-apples
```

```
## Error in eval(expr, envir, enclos): object 'apples' not found
```

3. Avoid common names with special meanings or assigned to existing functions (These will auto complete).

See the tidyverse style guide *(https://style.tidyverse.org/syntax.html)* for more information on naming conventions.

> **How do I know what objects have been created?**
>
> To view a list of the objects you have created, use `ls()` or look at your global environment pane.

Reassigning and deleting objects

To reassign an object, simply overwrite the object.

```
#object with gene named 'tp53'
gene_name<-"tp53"
gene_name
```

```
## [1] "tp53"
```

```
#if instead we want to reassign gene_name to a different gene, we wou
gene_name<-"GH1"
gene_name
```

```
## [1] "GH1"
```

Warning

R will not warn you when objects are being overwritten, so use caution.

To delete an object from memory:

```
# delete the object 'gene_name'
rm(gene_name)
```

```
#the object no longer exists, so calling it will result in an error
gene_name
```

```
## Error in eval(expr, envir, enclos): object 'gene_name' not found
```

Things to note

- R doesn't care about spaces in your code. However, it can vastly improve readability if you include them. For example, "thisissohardtoread" but "this is fine".
- You can use tab completion to quickly type object or function names.

```
For example:

```{py3 hl_lines="1-100"}
clifford<-"a big red dog"
```
Now, type "clif" into the console and hit tab.
```

- Quotes *(https://stat.ethz.ch/R-manual/R-devel/library/base/html/Quotes.html)* are used anytime you are entering character string values. Either single or double quotes can be used. Otherwise, R will think you are calling an object.

Using functions

A function in R (or any computing language) is a short program that takes some input and returns some output.

An R function has three key properties:

- Functions have a name (e.g. dir, getwd); note that functions are case sensitive!
- Following the name, functions have a pair of ()
- Inside the parentheses, a function may take 0 or more arguments --- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/00-introduction.html)*

To create a function, you can use the following syntax:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

Navigating directories

Now that we know what a function is, let's use them to navigate our directories.

Our first function will be `getwd()`. This simply prints your working directory (our default directory for saving files) and is the R equivalent of `pwd` (if you know unix coding).

```
#print our working directory
getwd()
```

```
[1] "/home/rstudio/"
```

How can we find out what arguments a function takes?

For details on function arguments and examples of how to use the function, we should check the package / function documentation. We can get help by preceding a function with `?` or `??` if the package library has not been loaded. We can also use the function `args()`.

Let's see this in action with `setwd()`. `setwd()` is used to change our working directory. If we want to know what argument it takes, we can try the help documentation.

```
?setwd()
```

getwd {base} R Documentation

Get or Set Working Directory

Description

`getwd` returns an absolute filepath representing the current working directory of the `R` process; `setwd(dir)` is used to set the working directory to `dir`.

Usage

```
getwd()
setwd(dir)
```

Arguments

`dir`	A character string: tilde expansion will be done.

Details

See files for how file paths with marked encodings are interpreted.

Value

`getwd` returns a character string or `NULL` if the working directory is not available. On Windows the path returned will use `/` as the path separator and be encoded in UTF-8. The path will not have a trailing `/` unless it is the root directory (of a drive or share on Windows).

`setwd` returns the current directory before the change, invisibly and with the same conventions as `getwd`. It will give an error if it does not succeed (including if it is not implemented).

Note

Note that the return value is said to be **an** absolute filepath: there can be more than one representation of the path to a directory and on some OSes the value returned can differ after changing directories and changing back to the same directory (for example if symbolic links have been traversed).

See Also

`list.files` for the *contents* of a directory.

`normalizePath` for a 'canonical' path name.

Examples

```
(WD <- getwd())
if (!is.null(WD)) setwd(WD)
```

As we can see from the help documentation, `setwd()` requires the argument `dir`, which requires a character string pointing to the correct directory. The path should be in quotes, and *you can use tab completion to fill in the path as needed.*

> **Note**
>
> R uses unix formatting for directories, so regardless of whether you have a Windows computer or a mac, the way you enter the directory information will be the same.

Function arguments are positional

Function arguments are positional, meaning the order matters unless the argument is explicitly stated. Let's see this in practice with the function `round()`.

`round()`

> rounds the values in its first argument to the specified number of decimal places (default 0) --- R help.

This implies that the first argument should be the number you want to round.

Let's see an example:

```
round(17.664, 2) #round 17.664 to 17.66
```

```
## [1] 17.66
```

```
round(2, 17.664) #round 2 (second argument ignored)
```

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

```
round(digits=2, 17.664) #explicitly state one of the arguments
```

```
## [1] 17.66
```

Some common functions

There are several functions that you will see repeatedly as you use R more and more.

One of those is `c()`, which is used to combine its arguments to form a vector.

> Vectors are probably the most used commonly used object type in R. A vector is a collection of values that are all of the same type (numbers, characters, etc.). The columns that make up a data frame, for example, are vectors of the same length. --- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html)*.

Let's create some vectors.

```

```

```
transcript_names<-c("TSPAN6","TNMD","SCYL3","GCLC") #We can create a

transcript_names<-c(TSPAN6,"TNMD",SCYL3,"GCLC") #Why doesn't this wo
```

```
## Error in eval(expr, envir, enclos): object 'TSPAN6' not found
```

```
transcript_counts <- c(679, 0, 467, 260,  60,   0) #combine numbers

sample_names<-c("1","B","3","D") #This is poor practice; stay consis

sample_names<-c("Sample1","Sample2","Sample3")

more_samps<-c("Sample4","Sample5")

sample_names<-c(sample_names,more_samps) #combine two vectors
```

Here is a short list of functions that are commonly used and good to keep in mind:

rbind(), cbind() - Combine vectors by row/column

grep() - regular expressions1

identical() - test if 2 objects are exactly equal

length() - no. of elements in vector

ls() - list objects in current environment

rep(x,n) - repeat the number x, n times

rev(x) - elements of x in reverse order

seq(x,y,n) - sequence (x to y, spaced by n)

sort(x) - sort the vector x

order(x) - list the sorted element numbers of x

tolower(),toupper() - Convert string to lower/upper case letters

unique(x) - remove duplicate entries from vector

round(x), signif(x), trunc(x) - rounding functions

month.abb/month.name - abbreviated and full names for months

pi, letters, (e.g. letters[7] = ”g”) LETTERS

lm - fit linear model

mean(x), weighted.mean(x), median(x), min(x), max(x), quantile(x)

sd() - standard deviation

summary(x) - a summary of x (mean, min, max)
--- Charles Dimaggio, columbia.edu *(http://www.columbia.edu/~cjd11/charles_dimaggio/DIRE/resources/R/rFunctionsList.pdf)*

You may also find this function reference card valuable: reference card *(https://cran.r-project.org/doc/contrib/Short-refcard.pdf)*

Explicitly calling a function

At times a function may be masked by another function. This can happen if two functions are named the same (e.g., `dplyr::filter()` vs `plyr::filter()`). We can get around this by explicitly calling a function from the correct package using the following syntax: `package::function()`.

A quick look at data frames

We will mostly be working with data frames throughout this course. Data frames hold tabular data, and as such are collections of vectors of the same length, but can be of different types.

Example data frame:

```
#create a data frame using data.frame()
df<-data.frame(id=paste("Sample",1:10,sep="_"), cell=rep(factor(c("ce
```

What do we mean by data types?

The data type of an R object affects how that object can be used or will behave. Examples of base R data types include numeric, integer, complex, character, and logical. R objects can also have certain assigned attributes (related to class), and these attributes will be important for how they interact with certain methods / functions. Ultimately, understanding the mode / type and class of an object will be important for how an object can be used in R. When the mode of an object is changed, we call this "coercion". You may see a coercion warning pop up when working with objects in the future.

Here are some of the most notable modes:

Mode (abbreviation)	Type of data
Numeric (num)	Numbers such floating point/decimals (1.0, 0.5, 3.14), there are also more specific numeric types (dbl - Double, int - Integer). These differences are not relevant for most beginners and pertain to how these values are stored in memory
Character (chr)	A sequence of letters/numbers in single ‘’ or double “ “ quotes
Logical	Boolean values - TRUE or FALSE

The mode or type of an object can be examined using `mode()` or `typeof()`. Its class can be examined using `class()`. Unlike modes and types, classes are unlimited and can be user defined. Classes can often be more informative. For example, `class()` may return data.frame, array, matrix, or factor.

```
od_600_value <- 0.47
typeof(od_600_value)
## [1] "double"

chr_position <- '1001701bp'
typeof(chr_position)
## [1] "character"

spock <- TRUE
typeof(spock)
## [1] "logical"

typeof(df)
## [1] "list"
class(df)
## [1] "data.frame"
```

Because data frames are made up of columns that can store different types of data, we can examine the overall structure of a data frame using `str()`.

```
str(df)
```

```
## 'data.frame':    20 obs. of  3 variables:
##  $ id    : chr  "Sample_1" "Sample_2" "Sample_3" "Sample_4" ...
##  $ cell  : Factor w/ 2 levels "cell_line_A",..: 1 1 1 1 1 1 1 1 1
##  $ counts: int  502 514 30 648 618 325 417 204 753 600 ...
```

There are functions that can gage types and classes directly, for example, `is.numeric()`, `is.character()`, `is.logical()`, `is.data.frame()`, `is.matrix()`, `is.factor()`.

What are factors?

> Factors can be thought of as vectors which are specialized for categorical data. Given R's specialization for statistics, this make sense since categorial and continuous variables are usually treated differently. Sometimes you may want to have data treated as a factor, but in other cases, this may be undesirable. *(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html)*

We will discuss factors more later when plotting. Functions most relevant to factors include `factor()` and `levels()`.

Data frame accessors

We can access a column of our data frame using `[]`, `[[]]`, or using the `$`. We can use `colnames()` and `rownames()` to access the column names and row names of a data frame.

For example:

```
df[["cell"]]
```

```
##  [1] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_A c
##  [7] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_B c
## [13] cell_line_B cell_line_B cell_line_B cell_line_B cell_line_B c
## [19] cell_line_B cell_line_B
## Levels: cell_line_A cell_line_B
```

```
df["cell"]
```

```
##           cell
## 1  cell_line_A
## 2  cell_line_A
## 3  cell_line_A
## 4  cell_line_A
## 5  cell_line_A
## 6  cell_line_A
## 7  cell_line_A
## 8  cell_line_A
## 9  cell_line_A
## 10 cell_line_A
## 11 cell_line_B
## 12 cell_line_B
## 13 cell_line_B
```

```
## 14 cell_line_B
## 15 cell_line_B
## 16 cell_line_B
## 17 cell_line_B
## 18 cell_line_B
## 19 cell_line_B
## 20 cell_line_B
```

```
df$cell
```

```
##  [1] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_A c
##  [7] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_B c
## [13] cell_line_B cell_line_B cell_line_B cell_line_B cell_line_B c
## [19] cell_line_B cell_line_B
## Levels: cell_line_A cell_line_B
```

```
colnames(df)
```

```
## [1] "id"     "cell"   "counts"
```

```
rownames(df)
```

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

Subsetting

Check out this chapter *(https://adv-r.hadley.nz/subsetting.html)* of *Advanced R* for more on subsetting operators.

Uploading and exporting files from RStudio Server

RStudio Server works via a web browser, and so you see this additional `Upload` option in the Files pane. If you select this option, you can upload files from your local computer into the server environment. If you select `More`, you will also see an `Export` option. You can use this to export the files created in the RStudio environment.

Saving your R environment (.Rdata)

When exiting RStudio, you will be prompted to save your R workspace or .RData. The .RData file saves the objects generated in your R environment. You can also save the .RData at any time using the floppy disk icon just below the Environment tab. You may also save your R workspace from the console using `save.image()`. You may load .Rdata by using `load()`.

Additional tips, tricks, and things to know

- You can use the up arrow on your keyboard when using the console to pull up previously used commands.
- Certain symbols in R always come in pairs, for example, parentheses and quotation marks. If you do not provide a pair, R will return a continuation character +, meaning it is waiting for more input to complete the command. You provide the missing information or press `ESCAPE`.
- You can print an object after creating using parantheses shortcuts.

  For example,

```
(coolfeature<-"printing an object automatically")
```

```
## [1] "printing an object automatically"
```

Acknowledgments

Material from this lesson was either taken directly or adapted from the Intro to R and RStudio for Genomics lesson provided by datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/01-r-basics.html)*.

Additional Resources

Hands-on Programming with R *(https://rstudio-education.github.io/hopr/)*

R reference card *(https://cran.r-project.org/doc/contrib/Short-refcard.pdf)*

R specific search engine, rseek *(https://rseek.org/)*

Data import and reshape

Objectives
1. Learn to import multiple data types
2. Data reshape with `tidyr`: `pivot_longer()`, `pivot_wider()`, `separate()`, and `unite()`

Installing and loading packages

So far we have only worked with objects that we created in RStudio. We have not installed or loaded any packages. R packages extend the use of R programming beyond base R.

Where do we get R packages?

As a reminder, R packages are loadable extensions that contain code, data, documentation, and tests in a standardized shareable format that can easily be installed by R users. The primary repository for R packages is the Comprehensive R Archive Network (CRAN) *(https://cran.r-project.org/index.html)*. CRAN is a global network of servers that store identical versions of R code, packages, documentation, etc (cran.r-project.org). To install a CRAN package, use `install.packages()`.

Github is another common source used to store R packages; though, these packages do not necessarily meet CRAN standards so approach with caution. To install a Github package, use `library(devtools)` followed by `install_github()`. `devtools` is a CRAN package. If you have not installed it, you may use `install.packages("devtools")` prior to the previous steps.

Many genomics and other packages useful to biologists / molecular biologists can be found on Bioconductor. To install a Bioconductor package, you will first need to install `BiocManager`, a CRAN package (`install.packages("BiocManager")`). You can then use `BiocManager` to install the Bioconductor core packages or any specific package (e.g., `BiocManager::install("DESeq2")`).

Importing / exporting data

Before we can do anything with our data, we need to first import it into R. There are several ways to do this.

First, the RStudio IDE has a drop down menu for data import. Simply go to `File > Import Dataset` and select one of the options and follow the prompts.

We should pay close attention to the import functions and their arguments. Using the import arguments correctly can save us from a headache later down the road. You will notice two types of import functions under `Import Dataset` "from text": base R import functions and `readr` import functions. We will use both in this course.

Note

`Tidyverse` packages are generally against assigning `rownames` and instead prefer that all column data are treated the same, but there are times when this is beneficial and will be required for genomics data (e.g., See `SummarizedExperiment` *(https://bioconductor.org/packages/devel/bioc/vignettes/SummarizedExperiment/inst/doc/SummarizedExperiment.html)* from Bioconductor).

Loading Tidyverse

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidy
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.4     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyverse_
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to f
```

What is a tibble?

When loading tabular data with `readr`, the default object created will be a `tibble`. Tibbles are like data frames with some small but apparent modifications. For example, they can have numbers for column names, and the column types are immediately apparent when viewing. Additionally, when you call a tibble by running the object name, the entire data frame does not print to the screen, rather the first ten rows along with the columns that fit the screen are shown.

Reasons to use `readr` functions

> They are typically much faster (~10x) than their base equivalents. Long running jobs have a progress bar, so you can see what's happening. If you're looking for raw speed, try data.table::fread(). It doesn't fit quite so well into the tidyverse, but it can be quite a bit faster.

> They produce tibbles, they don't convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions.
>
> They are more reproducible. Base R functions inherit some behaviour from your operating system and environment variables, so import code that works on your computer might not work on someone else's. ---R4DS *(https://r4ds.had.co.nz/data-import.html#compared-to-base-r)*

Excel files (.xls, .xlsx)

Excel files are the primary means by which many people save spreadsheet data. .xls or .xlsx files store workbooks composed of one or more spreadsheets.

Importing excel files requires the R package `readxl`. While this is a tidyverse package, it is not core and must be loaded separately.

```
library(readxl)
```

The functions to import excel files are `read_excel()`, `read_xls()`, and `read_xlsx()`. The latter two are more specific based on file format, whereas the first will guess which format (.xls or .xlsx) we are working with.

Let's look at its basic usage using an example data set from the `readxl` package. To access the example data we use `readxl_example()`.

```
#makes example data accessible by storing the path
ex_xl<-readxl_example("datasets.xlsx")
ex_xl
```

```
## [1] "/Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources
```

Now, let's read in the data. The only required argument is a path to the file to be imported.

```
irisdata<-read_excel(ex_xl)
irisdata
```

```
## # A tibble: 150 × 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
```

```
##  2          4.9         3           1.4         0.2 setosa
##  3          4.7         3.2         1.3         0.2 setosa
##  4          4.6         3.1         1.5         0.2 setosa
##  5          5           3.6         1.4         0.2 setosa
##  6          5.4         3.9         1.7         0.4 setosa
##  7          4.6         3.4         1.4         0.3 setosa
##  8          5           3.4         1.5         0.2 setosa
##  9          4.4         2.9         1.4         0.2 setosa
## 10          4.9         3.1         1.5         0.1 setosa
## # ℹ 140 more rows
```

Notice that the resulting imported data is a tibble. This is a feature specific to tidyverse. Now, let's check out some of the additional arguments. We can view the help information using `?read_excel()`.

The arguments likely to be most pertinent to you are:

- `sheet` - the name or numeric position of the excel sheet to read.
- `col_names` - default TRUE uses the first read in row for the column names. You can also provide a vector of names to name the columns.
- `skip` - will allow us to skip rows that we do not wish to read in.
- `.name_repair` - automatically set to "unique", which makes sure that the column names are not empty and are all unique. `read_excel()` and * `readr` functions will not correct column names to make them syntactic. If you want corrected names, use `.name_repair = "universal"`.

Let's check out another example:

```
sum_air<-read_excel("./data/RNASeq_totalcounts_vs_totaltrans.xlsx")
```

```
## New names:
## • `` -> `...2`
## • `` -> `...3`
## • `` -> `...4`
```

```
sum_air
```

```
## # A tibble: 11 × 4
##    `Uses Airway Data`                ...2          ...3
##    <chr>                             <chr>         <chr>
##  1 Some RNA-Seq summary information <NA>          <NA>
##  2 <NA>                              <NA>          <NA>
```

```
##  3 Sample Name                    Treatment     Number of Transc
##  4 GSM1275863                     Dexamethasone 10768
##  5 GSM1275867                     Dexamethasone 10051
##  6 GSM1275871                     Dexamethasone 11658
##  7 GSM1275875                     Dexamethasone 10900
##  8 GSM1275862                     None          11177
##  9 GSM1275866                     None          11526
## 10 GSM1275870                     None          11425
## 11 GSM1275874                     None          11000
```

Upon importing this data, we can immediately see that something is wrong with the column names.

```
colnames(sum_air)
```

```
## [1] "Uses Airway Data" "...2"             "...3"             "...4
```

There are some extra rows of information at the beginning of the data frame that should be excluded. We can take advantage of additional arguments to load only the data we are interested in. We are also going to tell `read_excel()` that we want the names repaired to eliminate spaces.

```
sum_air<-read_excel("./data/RNASeq_totalcounts_vs_totaltrans.xlsx",
                    skip=3,.name_repair = "universal")
```

```
## New names:
## • `Sample Name` -> `Sample.Name`
## • `Number of Transcripts` -> `Number.of.Transcripts`
## • `Total Counts` -> `Total.Counts`
```

```
sum_air
```

```
## # A tibble: 8 × 4
##   Sample.Name Treatment     Number.of.Transcripts Total.Counts
##   <chr>       <chr>                         <dbl>        <dbl>
## 1 GSM1275863  Dexamethasone                 10768     18783120
## 2 GSM1275867  Dexamethasone                 10051     15144524
## 3 GSM1275871  Dexamethasone                 11658     30776089
## 4 GSM1275875  Dexamethasone                 10900     21135511
## 5 GSM1275862  None                          11177     20608402
```

```
## 6 GSM1275866  None                              11526     25311320
## 7 GSM1275870  None                              11425     24411867
## 8 GSM1275874  None                              11000     19094104
```

Name repair

Learn more about the `.name_repair` arguments here *(https://mpn.metworx.com/packages/tibble/2.1.3/reference/name-repair.html)*.

Tab-delimited files (.tsv, .txt)

In tab delimited files, data columns are separated by tabs.

To import tab-delimited files there are several options. There are base R functions such as `read.delim()` and `read.table()` as well as the `readr` functions `read_delim()`, `read_tsv()`, and `read_table()`.

Let's take a look at `?read.delim()` and `?read_delim()`, which are most appropriate if you are working with tab delimited data stored in a .txt file.

For `read.delim()`, you will notice that the default separator (`sep`) is white space, which can be one or more spaces, tabs, newlines. However, you could use this function to load a comma separated file as well; you simply need to use `sep = ","`. The same is true of `read_delim()`, except the argument is `delim` rather than `sep`.

Let's load sample information from the RNA-Seq project `airway` *(https://bioconductor.org/packages/release/data/experiment/html/airway.html)*. We will refer back to some of these data frequently throughout our lessons. The airway data is from Himes et al. (2014) *(https://pubmed.ncbi.nlm.nih.gov/24926665/)*. These data, which are available in R as a RangedSummarizedExperiment object, are from a bulk RNAseq experiment. In the experiment, the authors "characterized transcriptomic changes in four primary human ASM cell lines that were treated with dexamethasone," a common therapy for asthma. The airway package includes RNAseq count data from 8 airway smooth muscle cell samples. Each cell line includes a treated and untreated negative control.

Using `read.delim()`:

```
smeta<-read.delim("./data/airway_sampleinfo.txt")
head(smeta)
```

```
##   SampleName    cell   dex albut        Run avgLength Experiment
## 1 GSM1275862  N61311 untrt untrt SRR1039508       126  SRX384345 S
## 2 GSM1275863  N61311   trt untrt SRR1039509       126  SRX384346 S
## 3 GSM1275866 N052611 untrt untrt SRR1039512       126  SRX384349 S
```

```
## 4 GSM1275867 N052611   trt untrt SRR1039513         87  SRX384350 S
## 5 GSM1275870 N080611 untrt untrt SRR1039516        120  SRX384353 S
## 6 GSM1275871 N080611   trt untrt SRR1039517        126  SRX384354 S
##      BioSample
## 1 SAMN02422669
## 2 SAMN02422675
## 3 SAMN02422678
## 4 SAMN02422670
## 5 SAMN02422682
## 6 SAMN02422673
```

Some other arguments of interest for `read.delim()`:

`row.names` - used to specify row names.

`col.names` - use to specify column names if `header = FALSE`.

`skip` - Similar to `read_excel()`, used to skip a number of lines preceding the data we are interested in importing.

`check.names` - makes names syntactically valid and unique.

Using `read_delim()`:

```
smeta2<-read_delim("./data/airway_sampleinfo.txt")
```

```
## Rows: 8 Columns: 9
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (8): SampleName, cell, dex, albut, Run, Experiment, Sample, B
## dbl (1): avgLength
##
## ℹ Use `spec()` to retrieve the full column specification for this
## ℹ Specify the column types or set `show_col_types = FALSE` to quie
```

```
smeta2
```

```
## # A tibble: 8 × 9
##   SampleName cell    dex   albut Run       avgLength Experiment Sa
##   <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <c
## 1 GSM1275862 N61311  untrt untrt SRR10395…       126 SRX384345  SI
## 2 GSM1275863 N61311  trt   untrt SRR10395…       126 SRX384346  SI
## 3 GSM1275866 N052611 untrt untrt SRR10395…       126 SRX384349  SI
## 4 GSM1275867 N052611 trt   untrt SRR10395…        87 SRX384350  SI
## 5 GSM1275870 N080611 untrt untrt SRR10395…       120 SRX384353  SI
## 6 GSM1275871 N080611 trt   untrt SRR10395…       126 SRX384354  SI
```

```
## 7 GSM1275874 N061011 untrt untrt SRR10395…       101 SRX384357  SI
## 8 GSM1275875 N061011 trt   untrt SRR10395…        98 SRX384358  SI
```

Comma separated files (.csv)

In comma separated files the columns are separated by commas and the rows are separated by new lines.

To read comma separated files, we can use the specific functions `?read.csv()` and `?read_csv()`.

Let's see this in action:

```
cexamp<-read.csv("./data/surveys_datacarpentry.csv")
head(cexamp)
```

```
##   record_id month day year plot_id species_id sex hindfoot_length
## 1         1     7  16 1977       2         NL   M              32
## 2         2     7  16 1977       3         NL   M              33
## 3         3     7  16 1977       2         DM   F              37
## 4         4     7  16 1977       7         DM   M              36
## 5         5     7  16 1977       3         DM   M              35
## 6         6     7  16 1977       1         PF   M              14
```

The arguments are the same as `read.delim()`.

Let's check out `read_csv()`:

```
cexamp2<-read_csv("./data/surveys_datacarpentry.csv")
```

```
## Rows: 35549 Columns: 9
## ── Column specification ────────────────────────────────────────
## Delimiter: ","
## chr (2): species_id, sex
## dbl (7): record_id, month, day, year, plot_id, hindfoot_length, we
##
## ℹ Use `spec()` to retrieve the full column specification for this
## ℹ Specify the column types or set `show_col_types = FALSE` to quie
```

```
cexamp2
```

```
## # A tibble: 35,549 × 9
##    record_id month   day  year plot_id species_id sex   hindfoot_
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>
##  1         1     7    16  1977       2 NL         M
##  2         2     7    16  1977       3 NL         M
##  3         3     7    16  1977       2 DM         F
##  4         4     7    16  1977       7 DM         M
##  5         5     7    16  1977       3 DM         M
##  6         6     7    16  1977       1 PF         M
##  7         7     7    16  1977       2 PE         F
##  8         8     7    16  1977       1 DM         M
##  9         9     7    16  1977       1 DM         F
## 10        10     7    16  1977       6 PF         F
## # ℹ 35,539 more rows
```

Other file types

There are a number of other file types you may be interested in. For genomic specific formats, you will likely need to install specific packages; check out Bioconductor *(https://bioconductor.org/)* for packages relevant to bioinformatics.

For information on importing other files types (e.g., json, xml, google sheets), check out this chapter *(https://jhudatascience.org/tidyversecourse/get-data.html)* from Tidyverse Skills for Data Science *(https://jhudatascience.org/tidyversecourse/)* by Carrie Wright, Shannon E. Ellis, Stephanie C. Hicks and Roger D. Peng.

An Example

Let's load in a count matrix from `airway` to work with and reshape.

```
aircount<-read.delim("./data/head50_airway_nonnorm_count.txt")
head(aircount)
```

```
##                         X Accession.SRR1039508 Accession.SRR1039
## 1   ENSG00000000003.TSPAN6                 679                 4
## 2     ENSG00000000005.TNMD                   0
## 3     ENSG00000000419.DPM1                 467                 5
## 4    ENSG00000000457.SCYL3                 260                 2
## 5 ENSG00000000460.C1orf112                  60
## 6      ENSG00000000938.FGR                   0
##   Accession.SRR1039512 Accession.SRR1039513 Accession.SRR1039516
## 1                  873                  408                 1138
## 2                    0                    0                    0
```

```
## 3                   621                   365                   587
## 4                   263                   164                   245
## 5                    40                    35                    78
## 6                     2                     0                     1
##   Accession.SRR1039517 Accession.SRR1039520 Accession.SRR1039521
## 1                 1047                   770                   572
## 2                    0                     0                     0
## 3                  799                   417                   508
## 4                  331                   233                   229
## 5                   63                    76                    60
## 6                    0                     0                     0
```

Because this is a count matrix, we want to save column 'X', which was automatically named, as row names rather than a column.

Let's reload and overwrite the previous object:

```
aircount<-read.delim("./data/head50_airway_nonnorm_count.txt",
                     row.names = 1)
head(aircount)
```

```
##                         Accession.SRR1039508 Accession.SRR103950
## ENSG00000000003.TSPAN6                  679                  448
## ENSG00000000005.TNMD                      0                    0
## ENSG00000000419.DPM1                    467                  515
## ENSG00000000457.SCYL3                   260                  211
## ENSG00000000460.C1orf112                 60                   55
## ENSG00000000938.FGR                       0                    0
##                         Accession.SRR1039512 Accession.SRR103951
## ENSG00000000003.TSPAN6                  873                  408
## ENSG00000000005.TNMD                      0                    0
## ENSG00000000419.DPM1                    621                  365
## ENSG00000000457.SCYL3                   263                  164
## ENSG00000000460.C1orf112                 40                   35
## ENSG00000000938.FGR                       2                    0
##                         Accession.SRR1039516 Accession.SRR103951
## ENSG00000000003.TSPAN6                 1138                 1047
## ENSG00000000005.TNMD                      0                    0
## ENSG00000000419.DPM1                    587                  799
## ENSG00000000457.SCYL3                   245                  331
## ENSG00000000460.C1orf112                 78                   63
## ENSG00000000938.FGR                       1                    0
##                         Accession.SRR1039520 Accession.SRR103952
## ENSG00000000003.TSPAN6                  770                  572
## ENSG00000000005.TNMD                      0                    0
```

```
## ENSG00000000419.DPM1                417                508
## ENSG00000000457.SCYL3               233                229
## ENSG00000000460.C1orf112             76                 60
## ENSG00000000938.FGR                   0                  0
```

Working with row names

There are functions *(https://tibble.tidyverse.org/reference/rownames.html)* specific to the `tibble` package for working with row names. `column_to_rownames()` could also have been used to assign column `X` to rows.

Data reshape

What do we mean by reshaping data?

Data reshaping is one aspect of tidying our data. The shape of our data is determined by how values are organized across rows and columns. When reshaping data we are most often wrangling the data from wide to long format or vice versa. To tidy the data we will need to (1) know the difference between observations and variables, and (2) potentially resolve cases in which a single variable is spread across multiple columns or a single observation is spread across multiple rows (R4DS *(https://r4ds.had.co.nz/tidy-data.html)*).

It is difficult to provide a single definition for what is wide data vs long data, as both can take different forms, and both can be considered tidy depending on the circumstance.

Important

Remember, while we are interested in getting data into a "tidy" format, your data should ultimately be wrangled into a format that is going to work with downstream analyses.

In general, in **wide data** there is often a single metric spread across multiple columns. This type of data often, but not always, takes on a matrix like appearance.

While in **long data**, each variable tends to have its own column.

See this example from R4DS:

Figure 12.2: Pivoting `table4` into a longer, tidy form.

However, these definitions depend on what you are ultimately considering a variable and what you are considering an observation.

For example, which of the following data representations is the tidy option?

```
tibble(iris)
```

```
## # A tibble: 150 × 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ℹ 140 more rows
```

```
iris_long<-tibble(iris) %>% rownames_to_column("Iris_id")
pivot_longer(iris_long,2:5,names_to="Measurement_location",values_to=
```

```
## # A tibble: 600 × 4
##    Iris_id Species Measurement_location Measurement
##    <chr>   <fct>   <chr>                       <dbl>
```

```
##  1 1       setosa  Sepal.Length                5.1
##  2 1       setosa  Sepal.Width                 3.5
##  3 1       setosa  Petal.Length                1.4
##  4 1       setosa  Petal.Width                 0.2
##  5 2       setosa  Sepal.Length                4.9
##  6 2       setosa  Sepal.Width                 3
##  7 2       setosa  Petal.Length                1.4
##  8 2       setosa  Petal.Width                 0.2
##  9 3       setosa  Sepal.Length                4.7
## 10 3       setosa  Sepal.Width                 3.2
## # ℹ 590 more rows
```

Regardless, you may want one format or the other depending on your analysis goals. Many of the tidyverse tools (e.g., `ggplot2`) seem to work better with long format data.

The tools we use to go from wide to long and long to wide are from the package `tidyr`. Because we already loaded the package `tidyverse`, we do not need to load `tidyr`, as it is a core package.

`pivot_wider()` and `pivot_longer()`

`pivot_wider()` and `pivot_longer()` have replaced the functions `gather()` and `spread()`. `pivot_wider()` converts long format data to wide, while `pivot_longer()` converts wide format data to long.

If you haven't guessed already, our count matrix is currently in wide format. If we wanted to merge these data with sample metadata and plot various aspects of the data using ggplot2, we would likely want these data in long format.

Let's check out the help documentation `?pivot_longer()`. This function requires the data and the columns we want to combine. There are also a number of optional arguments involving the name column and the value column.

```
l_air<-pivot_longer(aircount,1:length(aircount),names_to ="Sample",
                    values_to= "Count")
head(l_air)
```

```
## # A tibble: 6 × 2
##   Sample               Count
##   <chr>                <int>
## 1 Accession.SRR1039508   679
## 2 Accession.SRR1039509   448
## 3 Accession.SRR1039512   873
## 4 Accession.SRR1039513   408
```

```
## 5 Accession.SRR1039516  1138
## 6 Accession.SRR1039517  1047
```

Notice that the row names were dropped. While we would want to keep row names if we were working with this matrix as is, because we want a long data frame, we will need to first put the row names into a column. For this, we will use `rownames_to_column()` from the tidyverse package tibble.

```
#save row names as a column
aircount<-rownames_to_column(aircount,"Feature")
head(aircount["Feature"])
```

```
##                   Feature
## 1   ENSG00000000003.TSPAN6
## 2     ENSG00000000005.TNMD
## 3     ENSG00000000419.DPM1
## 4    ENSG00000000457.SCYL3
## 5 ENSG00000000460.C1orf112
## 6      ENSG00000000938.FGR
```

```
#pivot longer...again
l_air<-pivot_longer(aircount,starts_with("Accession"),
                    names_to =c("Sample"),values_to= "Count")
head(l_air)
```

```
## # A tibble: 6 × 3
##   Feature                Sample               Count
##   <chr>                  <chr>                <int>
## 1 ENSG00000000003.TSPAN6 Accession.SRR1039508   679
## 2 ENSG00000000003.TSPAN6 Accession.SRR1039509   448
## 3 ENSG00000000003.TSPAN6 Accession.SRR1039512   873
## 4 ENSG00000000003.TSPAN6 Accession.SRR1039513   408
## 5 ENSG00000000003.TSPAN6 Accession.SRR1039516  1138
## 6 ENSG00000000003.TSPAN6 Accession.SRR1039517  1047
```

How can we get this back to a wide format? We can use `?pivot_wider()`. This requires two additional arguments beyond the data argument: `names_from` and `values_from`. The first, `names_from` should be the name of the column containing the new column names for your wide data. `values_from` is the column that contains the values to fill the rows of your wide data columns.

Let's pivot the data from long to wide.

```
w_air<-tidyr::pivot_wider(l_air,names_from = Sample,
                          values_from = Count)
head(w_air)
```

```
## # A tibble: 6 × 9
##   Feature         Accession.SRR1039508 Accession.SRR1039509 Acces
##   <chr>                          <int>                <int>
## 1 ENSG0000000000…                  679                  448
## 2 ENSG0000000000…                    0                    0
## 3 ENSG0000000041…                  467                  515
## 4 ENSG0000000045…                  260                  211
## 5 ENSG0000000046…                   60                   55
## 6 ENSG0000000093…                    0                    0
## # ℹ 5 more variables: Accession.SRR1039513 <int>, Accession.SRR103
## #   Accession.SRR1039517 <int>, Accession.SRR1039520 <int>,
## #   Accession.SRR1039521 <int>
```

Note

There are many optional arguments for both of these functions. These are there to help you reshape seemingly complicated data schemes. Don't get discouraged. The examples in the help documentation are extremely helpful.

Unite and separate

There are two additional functions from `Tidyr` that are very useful for organizing data: `unite()` and `separate()`. These are used to split or combine columns.

For example, you may have noticed that our feature column from our example data is really two types of information combined (an Ensembl id and a gene abbreviation). If we want to separate this column into two, we could easily do this with the help of `separate()`.

Let's see this in action. We want to separate the column `Feature` at the first `.`.

```
l_air2<-separate(l_air, Feature, into=c("Ensembl_ID","gene_abb"),
                 sep=".",remove=TRUE)
```

```
## Warning: Expected 2 pieces. Additional pieces discarded in 400 ro
## 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

```
head(l_air2)
```

```
## # A tibble: 6 × 4
##   Ensembl_ID gene_abb Sample               Count
##   <chr>      <chr>    <chr>                <int>
## 1 ""         ""       Accession.SRR1039508   679
## 2 ""         ""       Accession.SRR1039509   448
## 3 ""         ""       Accession.SRR1039512   873
## 4 ""         ""       Accession.SRR1039513   408
## 5 ""         ""       Accession.SRR1039516  1138
## 6 ""         ""       Accession.SRR1039517  1047
```

Did you notice that warning?

Always take note of warnings. In this case, our column did not separate as expected. It appears that the `sep` argument needs some adjustment. The description of the function `separate()` suggests that it can use a regular expression to separate columns. This is our first clue. The `sep` argument can't interpret what we are telling it to do. The `.` has a special meaning in regular expressions; it matches any character.

> If `.` matches any character, how do you match a literal `.`? You need to use an "escape" to tell the regular expression you want to match it exactly, not use its special behaviour. Like strings, regexps use the backslash, `\`, to escape special behaviour. So to match an `.`, you need the regexp `\.`. Unfortunately this creates a problem. We use strings to represent regular expressions, and `\` is also used as an escape symbol in strings. So to create the regular expression `\.` we need the string `"\\."`. --- stringr vignette *(https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html)*

Lost?

If this explanation of the `\` makes little sense to you, check out this reddit post *(https://www.reddit.com/r/learnprogramming/comments/13bb5pa/why_double_slashes_in_regex_expressions/?rdt=59580)* for more help.

```
l_air2<-separate(l_air, Feature, into=c("Ensembl_ID","gene_abb"),
                 sep="\\.",remove=TRUE)

head(l_air2)
```

```
## # A tibble: 6 × 4
##   Ensembl_ID      gene_abb Sample               Count
##   <chr>           <chr>    <chr>                <int>
## 1 ENSG00000000003 TSPAN6   Accession.SRR1039508   679
```

```
## 2 ENSG00000000003 TSPAN6   Accession.SRR1039509   448
## 3 ENSG00000000003 TSPAN6   Accession.SRR1039512   873
## 4 ENSG00000000003 TSPAN6   Accession.SRR1039513   408
## 5 ENSG00000000003 TSPAN6   Accession.SRR1039516  1138
## 6 ENSG00000000003 TSPAN6   Accession.SRR1039517  1047
```

`unite()` is simply the opposing function to `separate()`. Let's use `unite()` to combine our columns (`Ensemble_ID` and `gene_abb`) back together. This time we will use a `_` between our ensembleID and gene abbreviations.

```
l_air3<-unite(l_air2, "Feature", c(Ensembl_ID,gene_abb),sep="_")
head(l_air3)
```

```
## # A tibble: 6 × 3
##   Feature                Sample               Count
##   <chr>                  <chr>                <int>
## 1 ENSG00000000003_TSPAN6 Accession.SRR1039508   679
## 2 ENSG00000000003_TSPAN6 Accession.SRR1039509   448
## 3 ENSG00000000003_TSPAN6 Accession.SRR1039512   873
## 4 ENSG00000000003_TSPAN6 Accession.SRR1039513   408
## 5 ENSG00000000003_TSPAN6 Accession.SRR1039516  1138
## 6 ENSG00000000003_TSPAN6 Accession.SRR1039517  1047
```

A word about regular expressions

As you continue to work in R, at some point you will need to incorporate regular expressions into your code. We used a regular expression escape above `\\.` to denote that we wanted to match a literal `.` rather than its regex (short for regular expression) alternative, which matches any character except for a new line.

Regular expressions can be exceedingly complicated and like anything require time and practice. We will not take a deep dive into regular expressions in this course. A great place to start with regular expressions is Chapter 14: Strings *(https://r4ds.had.co.nz/strings.html#strings)* from R4DS. You may also find this stringr vignette *(https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html)* helpful.

Acknowledgements

Material from this lesson was inspired by R4DS *(https://r4ds.had.co.nz/data-import.html)* and Tidyverse Skills for Data Science *(https://jhudatascience.org/tidyversecourse/)*. The survey data *(https://figshare.com/articles/dataset/Portal_Project_Teaching_Database/1314459/10)* loaded in the section on comma separated files was taken from a datacarpentry.org lesson *(https://datacarpentry.org/R-ecology-lesson/index.html)*.

Additonal Resources

readr / readxl cheatsheet
Tidyr cheatsheet
Stringr / regex cheatsheet

Introduction to ggplot2

Objectives

1. Learn the ggplot2 syntax.
2. Build a ggplot2 general template.

By the end of the course, students should be able to create simple, pretty, and effective figures.

Data Visualization in the tidyverse

In lesson 3, we learned how to read and save excel spreadsheet data to a R object using the tidyverse package `readxl`. Today we will use some example data from an excel spreadsheet to learn the basics of `ggplot2`, a tidyverse core package.

```
#data import from excel
exdata<-readxl::read_xlsx("./data/RNASeq_totalcounts_vs_totaltrans.x
                          1,.name_repair = "universal", skip=3)
```

```
## New names:
## • `Sample Name` -> `Sample.Name`
## • `Number of Transcripts` -> `Number.of.Transcripts`
## • `Total Counts` -> `Total.Counts`
```

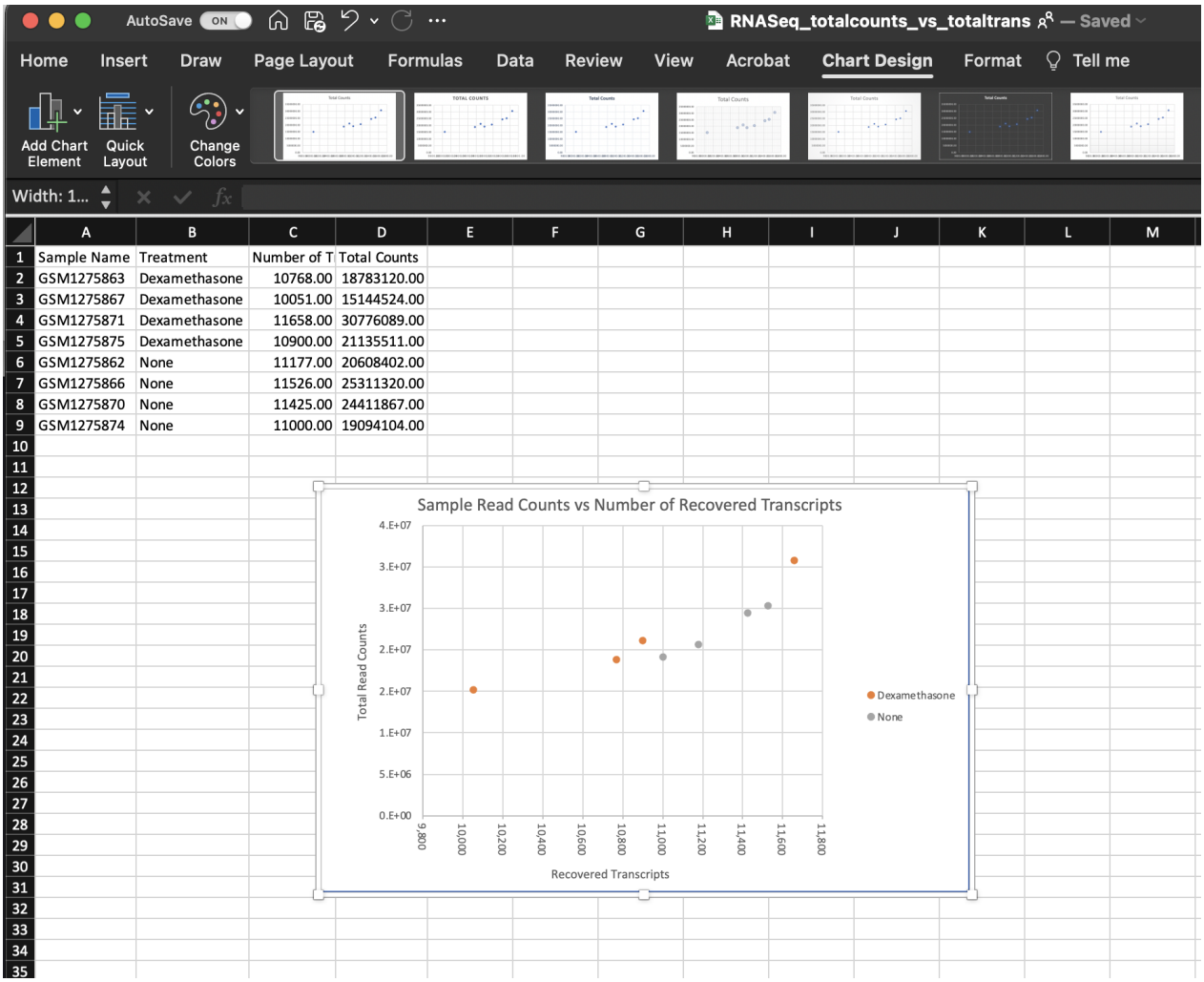
```
exdata
```

```
## # A tibble: 8 × 4
##   Sample.Name Treatment     Number.of.Transcripts Total.Counts
##   <chr>       <chr>                         <dbl>        <dbl>
## 1 GSM1275863  Dexamethasone                 10768     18783120
## 2 GSM1275867  Dexamethasone                 10051     15144524
## 3 GSM1275871  Dexamethasone                 11658     30776089
## 4 GSM1275875  Dexamethasone                 10900     21135511
## 5 GSM1275862  None                          11177     20608402
## 6 GSM1275866  None                          11526     25311320
## 7 GSM1275870  None                          11425     24411867
## 8 GSM1275874  None                          11000     19094104
```

These data include total transcript read counts summed by sample and the total number of transcripts recovered by sample that had at least 100 reads. These data derive from a bulk RNAseq experiment described by Himes et al. (2014) *(https://pubmed.ncbi.nlm.nih.gov/24926665/)* and introduced in lesson 3. As a reminder, the authors "characterized transcriptomic changes in four primary human ASM cell lines that were treated with dexamethasone," a common therapy for asthma. Each cell line included a treated and untreated negative control resulting in a total sample size of 8.

Plotting with Excel

We could plot this data in Excel and get something like this:



	A	B	C	D
1	Sample Name	Treatment	Number of T	Total Counts
2	GSM1275863	Dexamethasone	10768.00	18783120.00
3	GSM1275867	Dexamethasone	10051.00	15144524.00
4	GSM1275871	Dexamethasone	11658.00	30776089.00
5	GSM1275875	Dexamethasone	10900.00	21135511.00
6	GSM1275862	None	11177.00	20608402.00
7	GSM1275866	None	11526.00	25311320.00
8	GSM1275870	None	11425.00	24411867.00
9	GSM1275874	None	11000.00	19094104.00

While this isn't bad, it took an unnecessary amount of time to create, and there weren't a lot of options for customization.

RECOMMENDATION

You should save metadata or other tabular data as either comma separated files (.csv) or tab-delimited files (.txt, .tsv). Using these file extensions will make it easier to use the data with bioinformatic programs. There are multiple functions available to read in delimited data in R. We will see a few of these over the next few weeks.

Why `ggplot2`?

Outside of base R plotting, one of the most popular packages used to generate graphics in R is `ggplot2`, which is associated with a family of packages collectively known as the tidyverse. `GGplot2` allows the user to create informative plots quickly by using a 'grammar of graphics' implementation, which is described as "a coherent system for describing and building graphs" R4DS *(https://r4ds.had.co.nz/data-*

visualisation.html#:~:text=ggplot2%20implements%20the%20grammar%20of,applying%20it%20in%20many

We will see this in action shortly. The power of this package is that plots are built in layers and few changes to the code result in very different outcomes. This makes it easy to reuse parts of the code for very different figures.

Being a part of the tidyverse collection, `ggplot2` works best with data organized so that individual observations are in rows and variables are in columns.

Getting started with ggplot2

To begin plotting, we need to load our `ggplot2` library. Package libraries must be loaded every time you open and use R. If you haven't yet installed the ggplot2 package on your local machine, you will need to do that using `install.packages("ggplot2")`.

```
#load the ggplot2 library; you could also load library(tidyverse)
library(ggplot2)
```

Getting help

The R community is extensive and getting help is now easier than ever with a simple web search. If you can't figure out how to plot something, give a quick web search a try. Great resources include internet tutorials, R bookdowns, and stackoverflow. You should also use the help features within RStudio to get help on specific functions or to find vignettes. Try entering `ggplot2` in the help search bar in the lower right panel under the `Help` tab.

The ggplot2 template

The following represents the basic ggplot2 template.

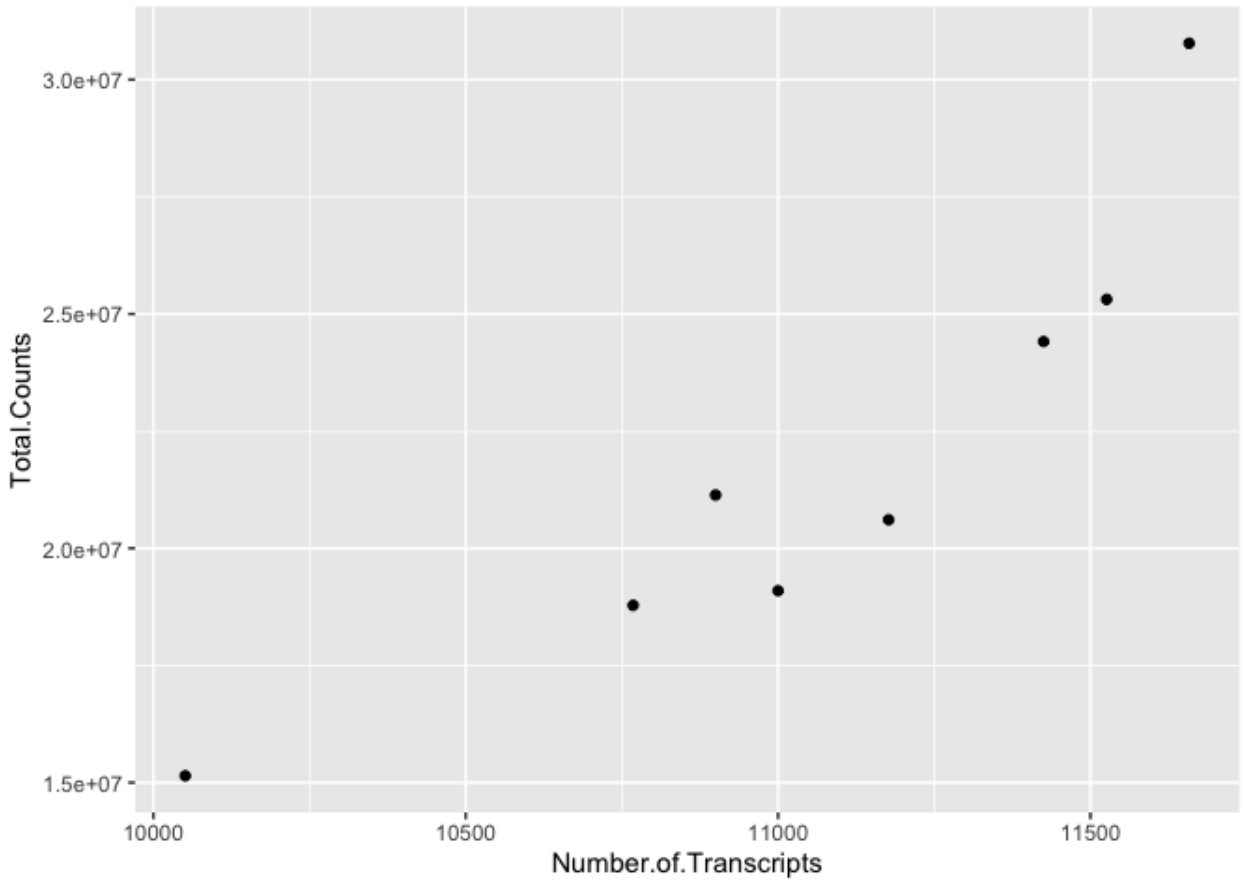
```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

The only required components to begin plotting are the data we want to plot, geom function(s), and mapping aesthetics. Notice the + symbol following the `ggplot()` function. This symbol will precede each additional layer of code for the plot, and it is important that it is **placed at the end of the line**. More on geom functions and mapping aesthetics to come.

Let's see this template in practice.

What is the relationship between total transcript sums per sample and the number of recovered transcripts per sample?

```
#let's plot our data
ggplot(data=exdata) +
  geom_point(aes(x=Number.of.Transcripts, y = Total.Counts))
```



We can easily see that there is a relationship between the number of transcripts per sample and the total transcripts recovered per sample. `ggplot2` default parameters are great for exploratory data analysis. But, with only a few tweaks, we can make some beautiful, publishable figures.

Let's take a closer look at the above code

The first step in creating this plot was initializing the ggplot object using the function `ggplot()`. Remember, we can look further for help using `?ggplot()`. The function `ggplot()` takes data, mapping, and further arguments. However, none of this needs to actually be provided at the initialization phase, which creates the coordinate system from which we build our plot. But, typically, you should at least call the data at this point.

The data we called was from the data frame `exdata`, which we created above. Next, we provided a geom function (`geom_point()`), which created a scatter plot. This scatter plot required mapping information, which we provided for the x and y axes. More on this in a moment.

Let's break down the individual components of the code.

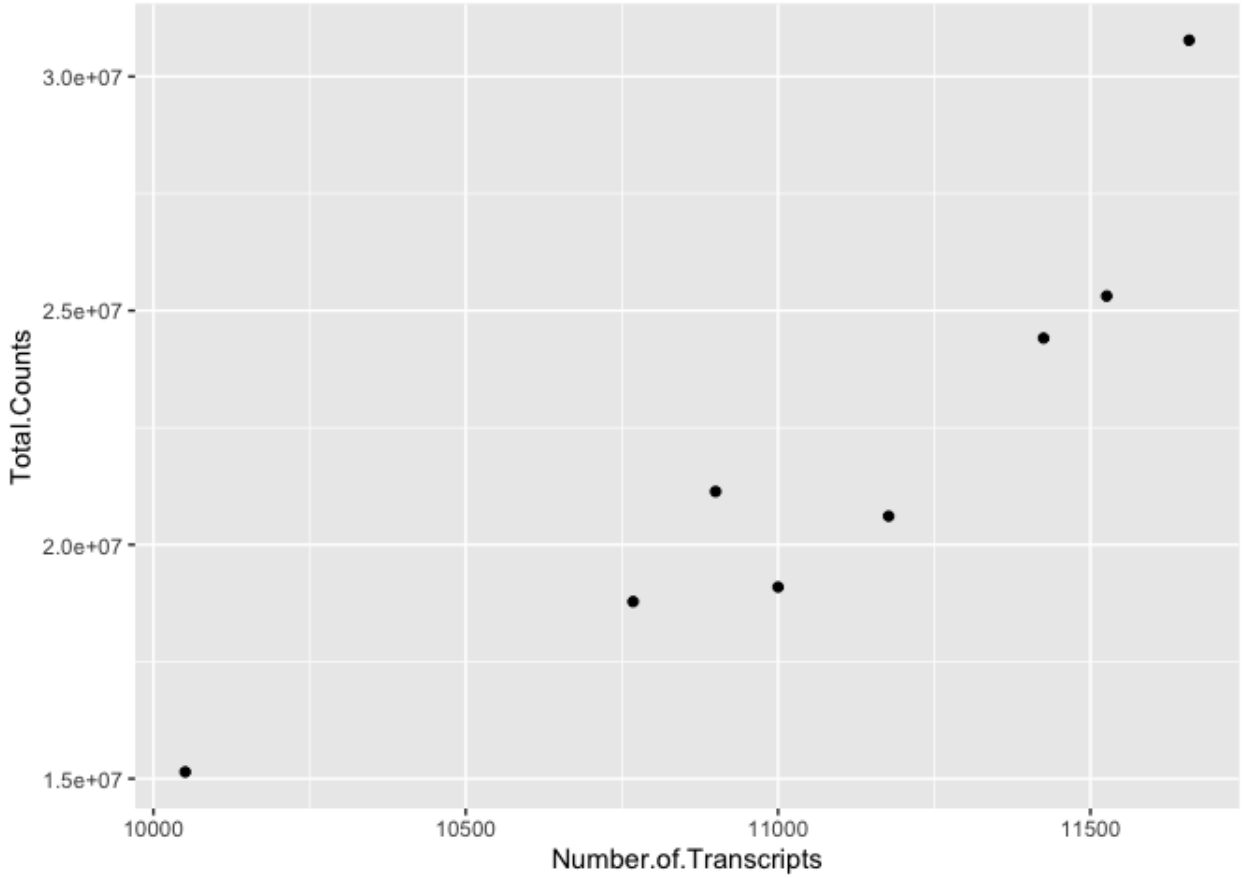
```
#What does running ggplot() do?
ggplot(data=exdata)
```



```
#What about just running a geom function?
geom_point(data=exdata,aes(x=Number.of.Transcripts, y = Total.Counts)
```

```
## mapping: x = ~Number.of.Transcripts, y = ~Total.Counts
## geom_point: na.rm = FALSE
## stat_identity: na.rm = FALSE
## position_identity
```

```
#what about this
ggplot() +
geom_point(data=exdata,aes(x=Number.of.Transcripts, y = Total.Counts)
```



Geom functions

> A geom is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. --- R4DS *(https://r4ds.had.co.nz/data-visualisation.html#geometric-objects)*

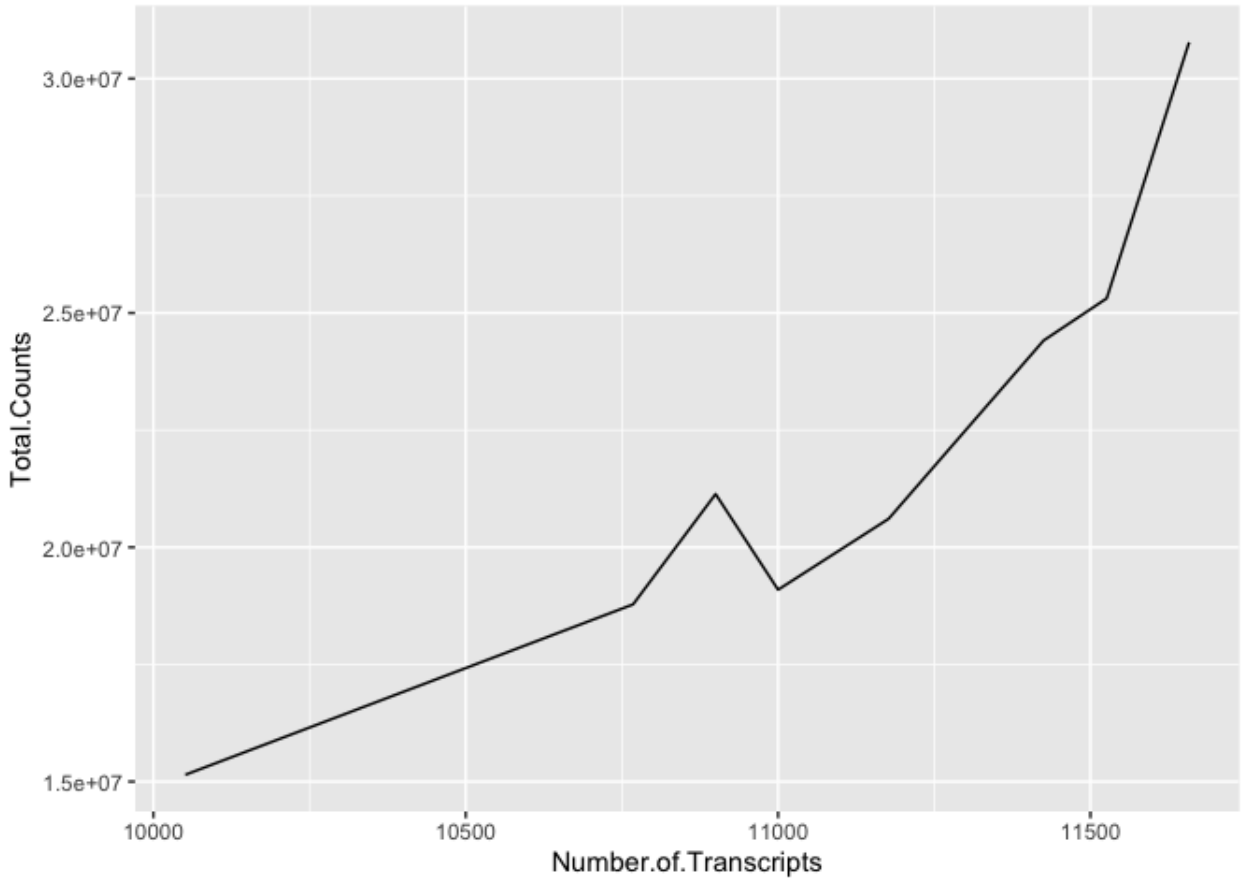
There are multiple geom functions that change the basic plot type or the plot representation. We can create scatter plots (`geom_point()`), line plots (`geom_line()`,`geom_path()`), bar plots (`geom_bar()`, `geom_col()`), line modeled to fitted data (`geom_smooth()`), heat maps (`geom_tile()`), geographic maps (`geom_polygon()`), etc.

> ggplot2 provides over 40 geoms, and extension packages provide even more (see https://exts.ggplot2.tidyverse.org/gallery/ *(https://exts.ggplot2.tidyverse.org/gallery/)* for a sampling). The best way to get a comprehensive overview is the ggplot2 cheatsheet, which you can find at http://rstudio.com/resources/cheatsheets. --- R4DS *(https://r4ds.had.co.nz/data-visualisation.html)*

You can also see a number of options pop up when you type `geom` into the console, or you can look up the `ggplot2` documentation in the help tab.

We can see how easy it is to change the way the data is plotted. Let's plot the same data using `geom_line()`.

```
ggplot(data=exdata) +
  geom_line(aes(x=Number.of.Transcripts, y = Total.Counts))
```



Mapping and aesthetics (`aes()`)

The geom functions require a mapping argument. The mapping argument includes the `aes()` function, which "describes how variables in the data are mapped to visual properties (aesthetics) of geoms" (ggplot2 R Documentation). If not included it will be inherited from the `ggplot()` function.

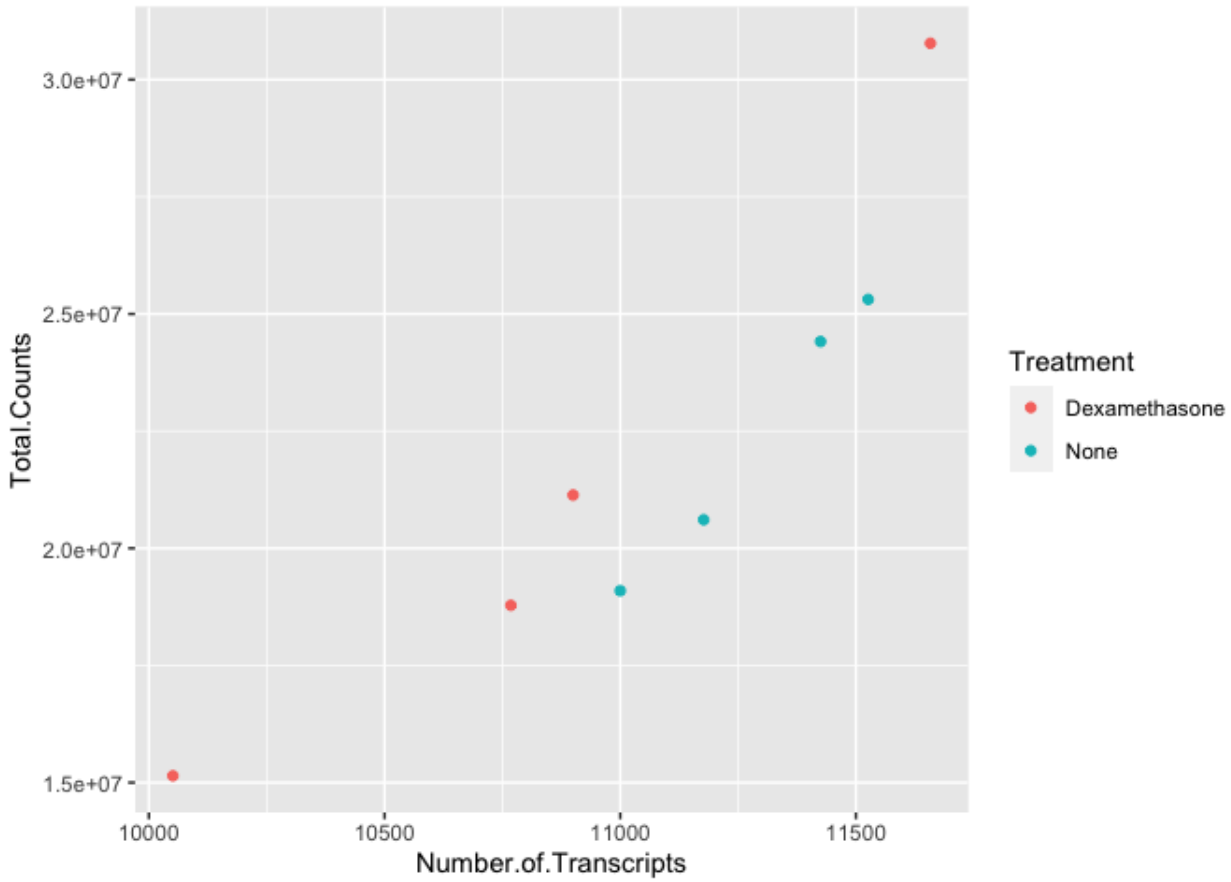
> An aesthetic is a visual property of the objects in your plot.---R4DS *(https://r4ds.had.co.nz/data-visualisation.html)*

Mapping aesthetics include some of the following:
1. the x and y data arguments
2. shapes
3. color
4. fill
5. size
6. linetype
7. alpha

This is not an all encompassing list.

Let's return to our plot above. Is there a relationship between treatment ("dex") and the number of transcripts or total counts?

```
#adding the color argument to our mapping aesthetic
ggplot(exdata) +
  geom_point(aes(x=Number.of.Transcripts, y = Total.Counts,
                 color=Treatment))
```



There is potentially a relationship. ASM cells treated with dexamethasone in general have lower total numbers of transcripts and lower total counts.

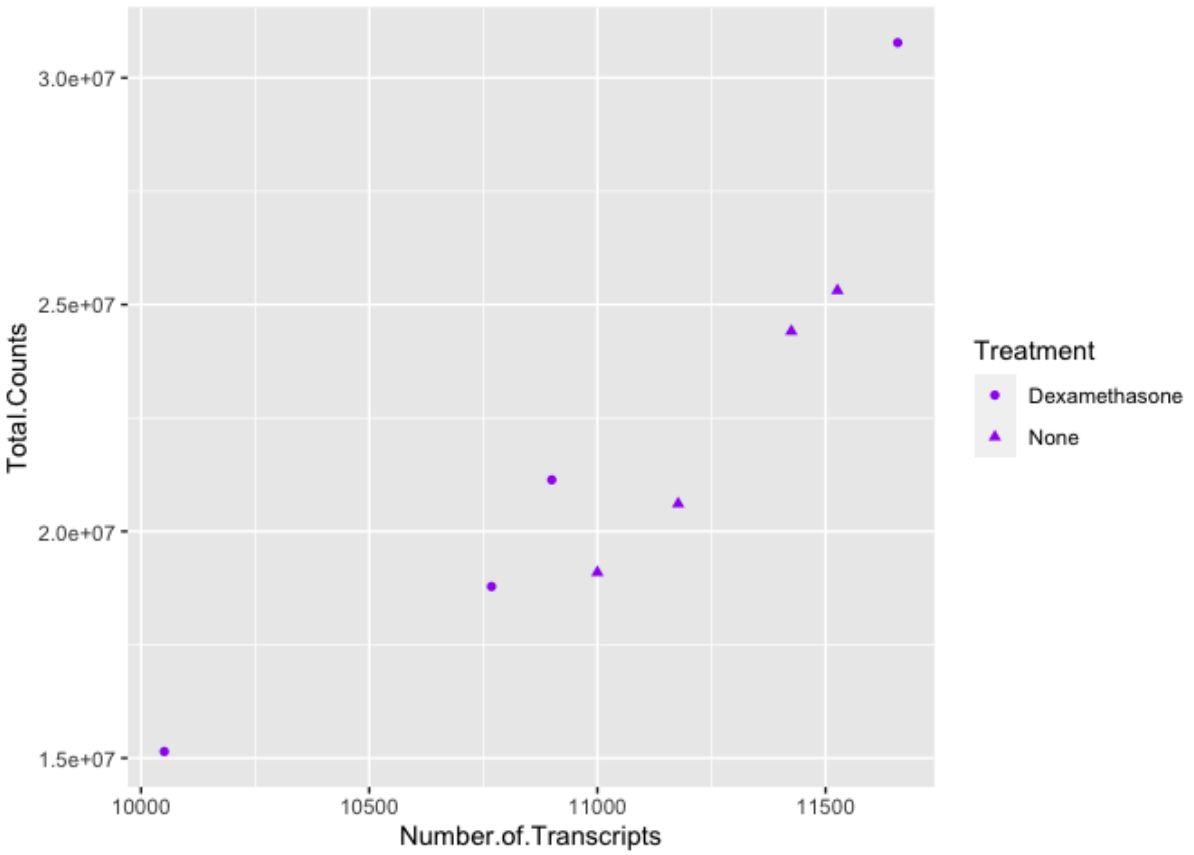
Notice how we changed the color of our points to represent a variable, in this case. To do this, we set color equal to 'Treatment' within the `aes()` function. This mapped our aesthetic, color, to a variable we were interested in exploring.

Mappings outside of `aes()`

Aesthetics that are not mapped to our variables are placed outside of the `aes()` function. These aesthetics are manually assigned and do not undergo the same scaling process as those within `aes()`.

For example

```
#map the shape aesthetic to the variable "dex"
#use the color purple across all points (NOT mapped to a variable)
ggplot(exdata) +
  geom_point(aes(x=Number.of.Transcripts, y = Total.Counts,
                 shape=Treatment), color="purple")
```



We can also see from this that 'Treatment' could be mapped to other aesthetics. In the above example, we see it mapped to shape rather than color. **By default, ggplot2 will only map six shapes at a time, and if your number of categories goes beyond 6, the remaining groups will go unmapped.** This is by design because it is hard to discriminate between more than six shapes at any given moment. This is a clue from ggplot2 that you should choose a different aesthetic to map to your variable. However, if you choose to ignore this functionality, you can manually assign more than six shapes *(https://r-graphics.org/recipe-scatter-shapes)*.

We could have just as easily mapped it to alpha, which adds a gradient to the point visibility by category, or we could map it to size. There are multiple options, so feel free to explore a little with your plots.

Defaults

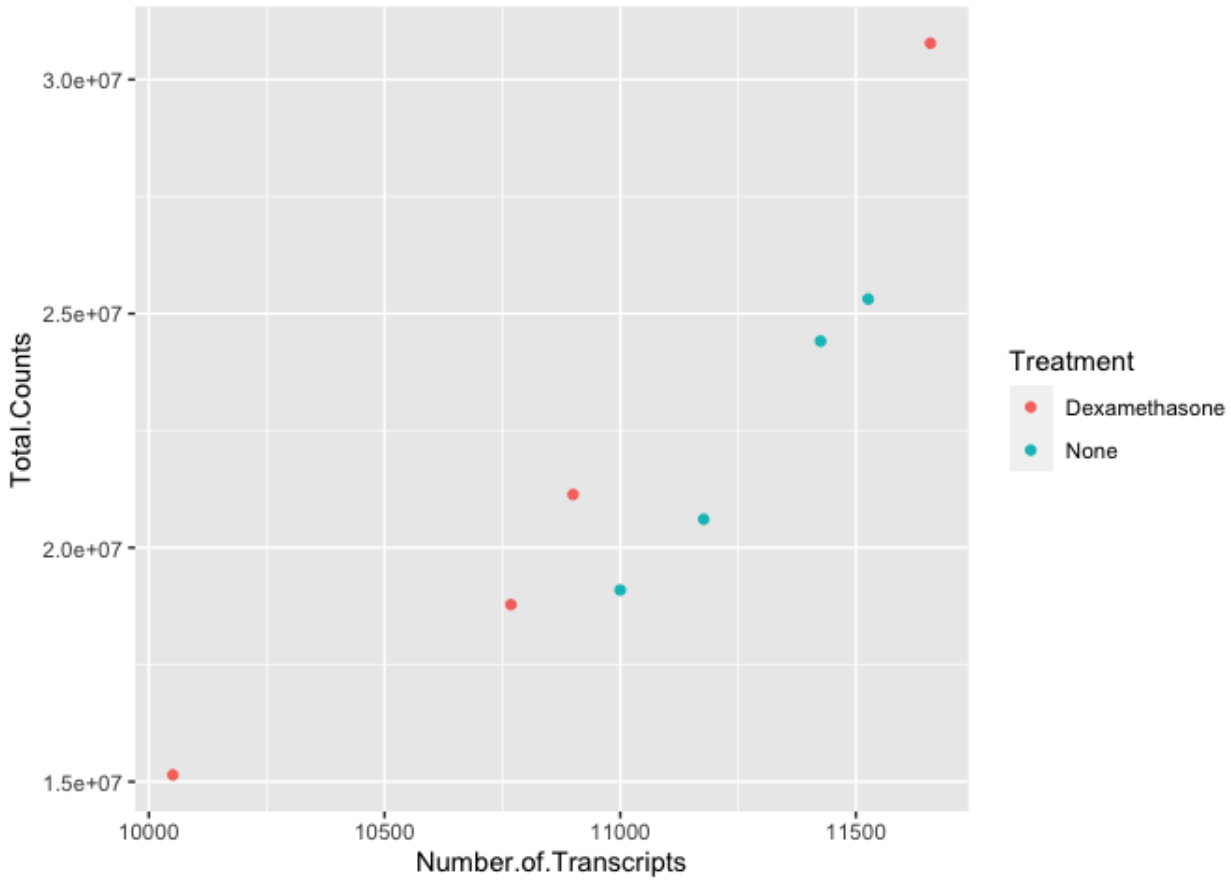
The assignment of color, shape, or alpha to our variable was automatic, with a unique aesthetic level representing each category (i.e., 'Dexamethasone', 'none') within our variable. You will also notice that ggplot2 automatically created a legend to explain the levels of the aesthetic mapped. We can change aesthetic parameters - what colors are used, for example - by adding additional layers to the plot.

R objects can also store figures

As we have discussed, R objects are used to store things created in R to memory. This includes plots.

```
scatter_plot<-ggplot(exdata) +
  geom_point(aes(x=Number.of.Transcripts, y = Total.Counts,
                 color=Treatment))

scatter_plot
```

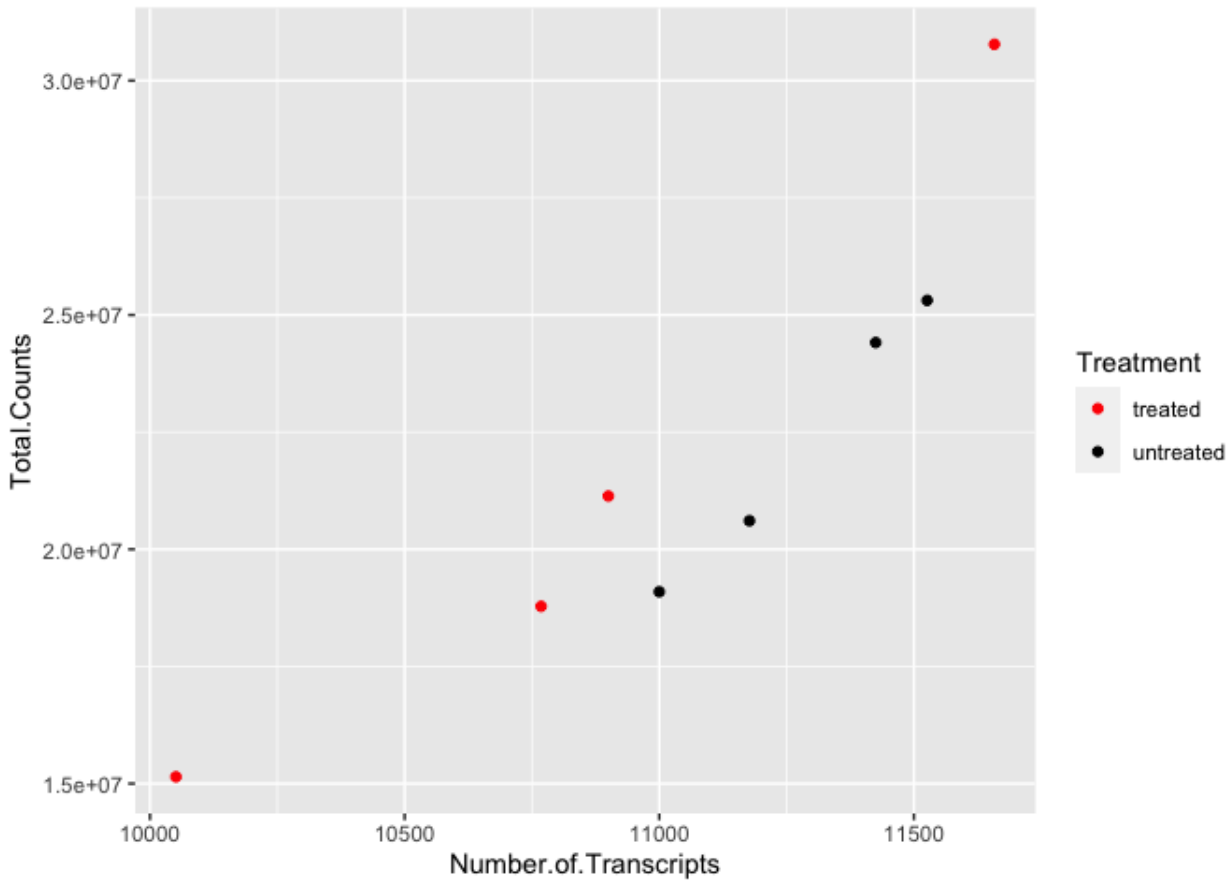


We can add additional layers directly to our object. We will see how this works by defining some colors for our 'dex' variable.

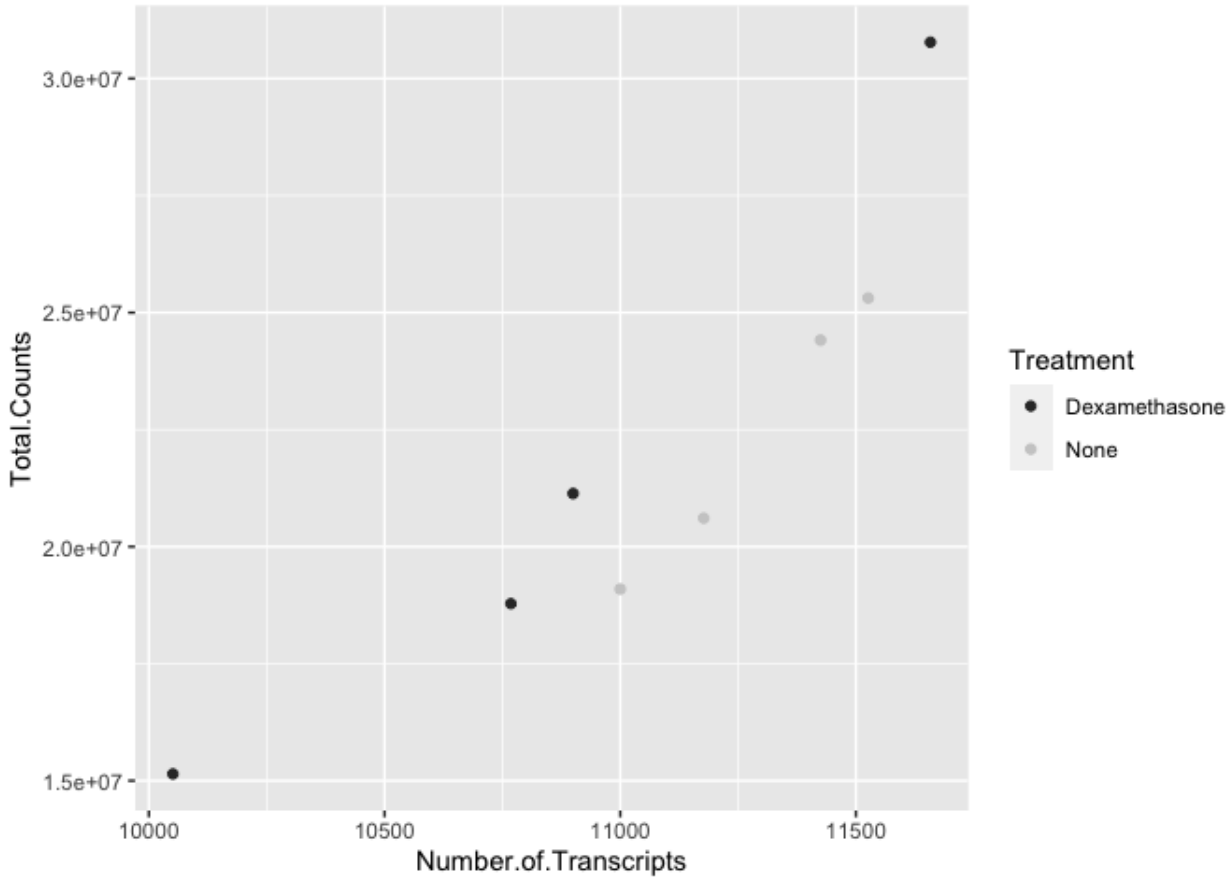
Colors

`ggplot2` will automatically assign colors to the categories in our data. Colors are assigned to the fill and color aesthetics in `aes()`. We can change the default colors by providing an additional layer to our figure. To change the color, we use the scale_color functions: `scale_color_manual()`, `scale_color_brewer()` *(https://r-graph-gallery.com/38-rcolorbrewers-palettes.html)*, `scale_color_grey()`, etc. We can also change the name of the color labels in the legend using the `labels` argument of these functions

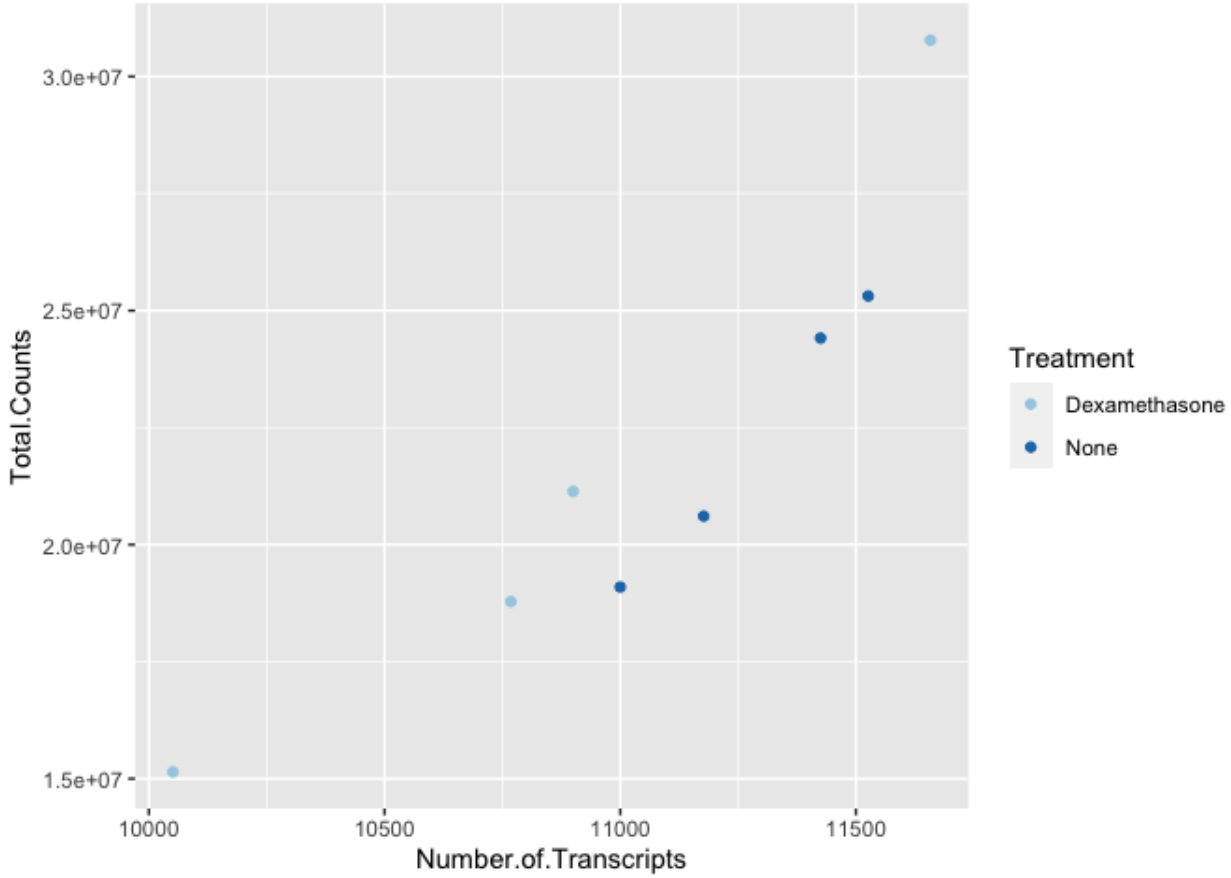
```
scatter_plot +
  scale_color_manual(values=c("red","black"),
                     labels=c('treated','untreated'))
```



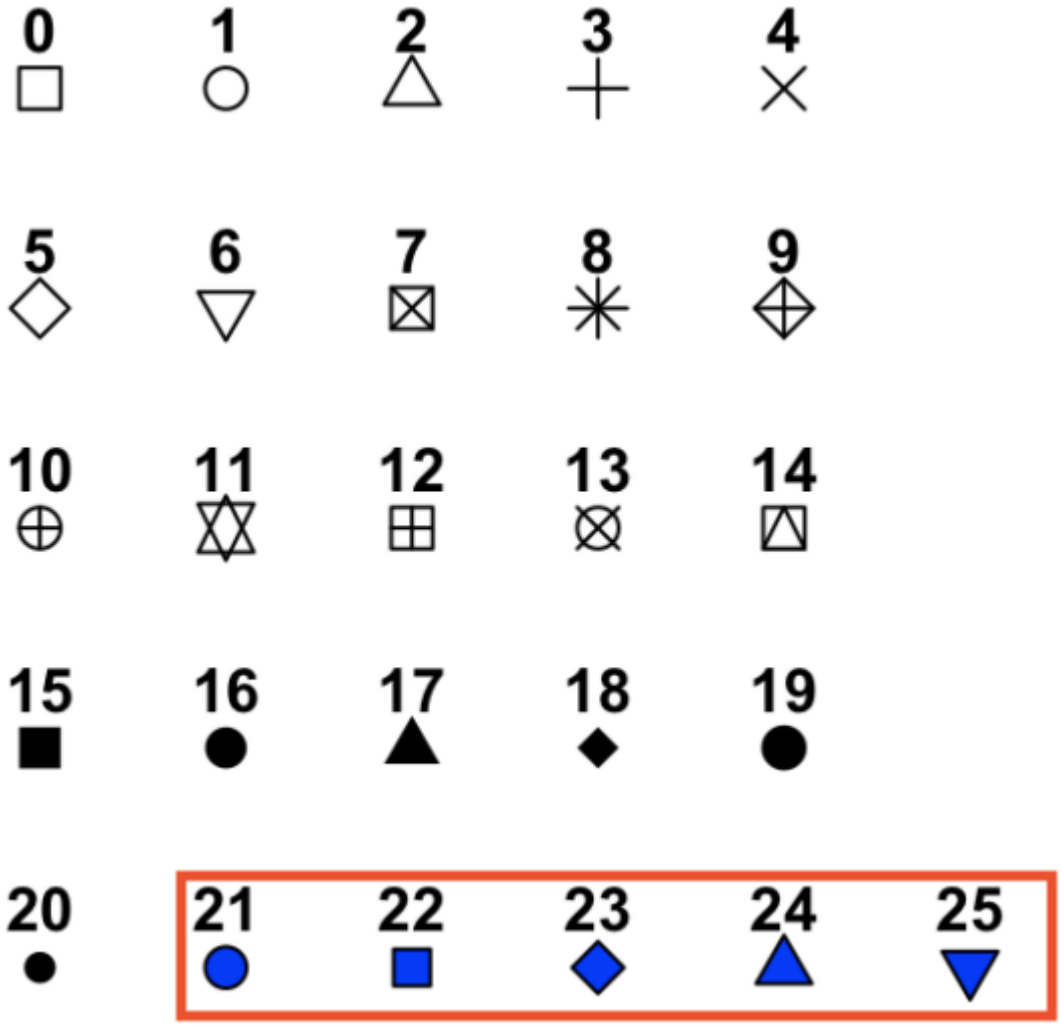

```
scatter_plot +
  scale_color_grey()
```



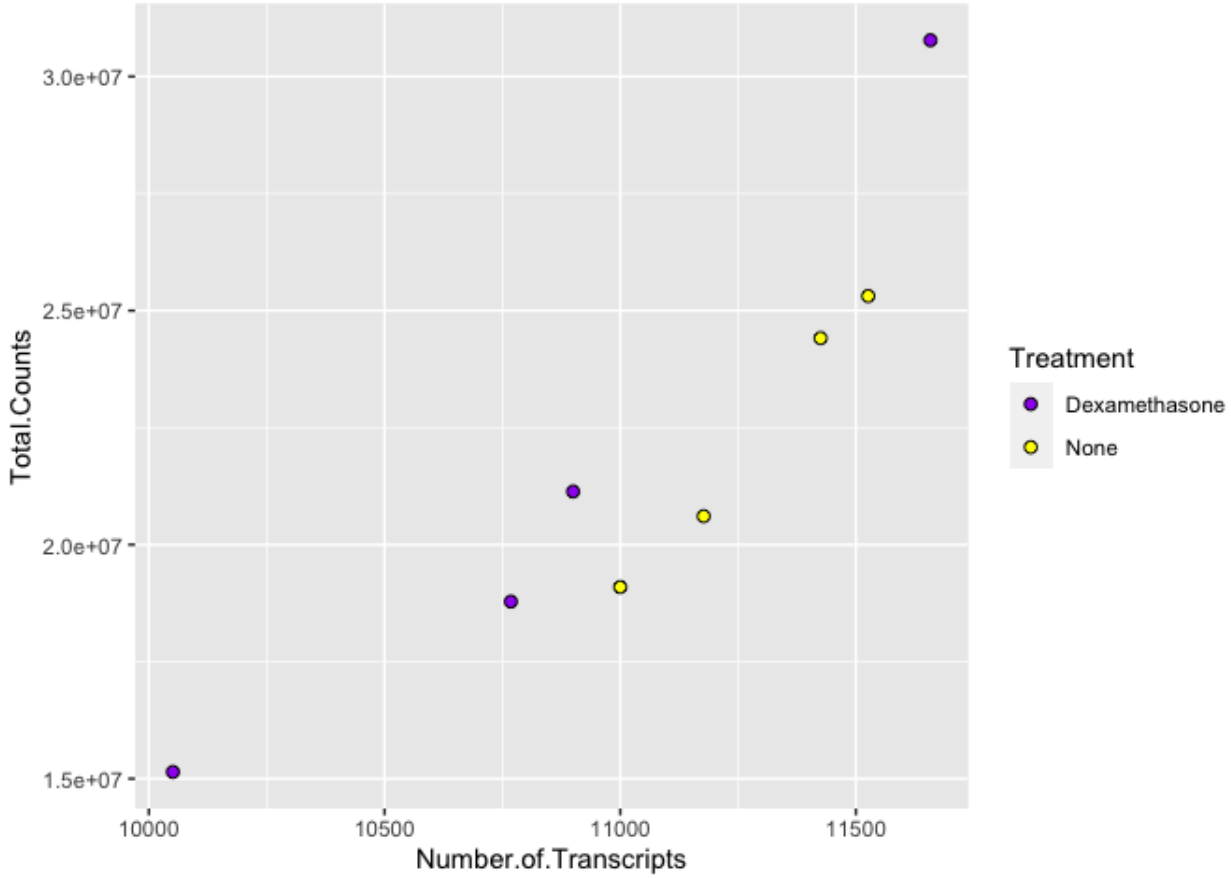

```
scatter_plot +
  scale_color_brewer(palette = "Paired")
```



Similarly, if we want to change the fill, we would use the scale_fill options. To apply scale_fill to shape, we will have to alter the shapes, as only some shapes take a fill argument. Refer to the shapes in the red box in the figure below.




```
ggplot(data=exdata) +
  geom_point(aes(x=Number.of.Transcripts, y = Total.Counts,fill=Treat
             shape=21,size=2) + #increase size and change points
  scale_fill_manual(values=c("purple", "yellow"))
```



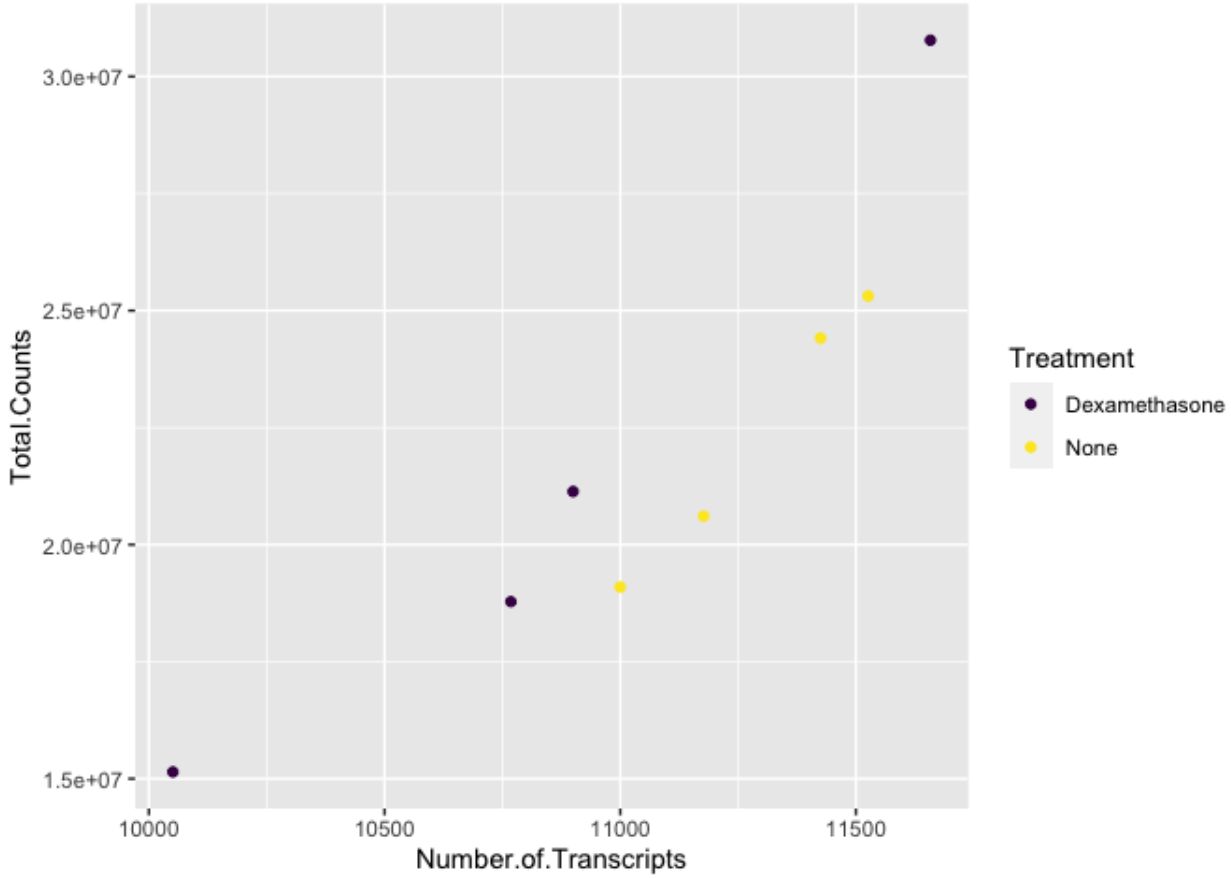
There are a number of ways to specify the color argument including by name, number, and hex code. Here *(https://www.r-graph-gallery.com/ggplot2-color.html)* is a great resource from the R Graph Gallery *(https://www.r-graph-gallery.com/index.html)* for assigning colors in R.

There are also a number of complementary packages in R that expand our color options. One of my favorites is `viridis`, which provides colorblind friendly palettes. `randomcoloR` is a great package if you need a large number of unique colors.

```
library(viridis) #Remember to load installed packages before use
```

```
## Loading required package: viridisLite
```

```
scatter_plot + scale_color_viridis(discrete=TRUE, option="viridis")
```



`Paletteer` contains a comprehensive set of color palettes, if you want to load the palettes from multiple packages all at once. See the Github page *(https://github.com/EmilHvitfeldt/paletteer)* for details.

Returning to our grammar of graphics

Remember, to create a plot all you you need are the `data`, `geom_function(s)`, and `mapping` arguments.

However, there are additional components that can be added to our core components to enable us to generate even more diverse plot types.

Our grammar of graphics:

- one or more datasets,
- one or more geometric objects that serve as the visual representations of the data, – for instance, points, lines, rectangles, contours,
- descriptions of how the variables in the data are mapped to visual properties (aesthetics) of the geometric objects, and an associated scale (e. g., linear, logarithmic, rank),

- a facet specification, i.e. the use of multiple similar subplots to look at subsets of the same data,
- one or more coordinate systems *(https://ggplot2.tidyverse.org/reference/#coordinate-systems)*,
- optional parameters that affect the layout and rendering, such text size, font and alignment, legend positions,
- statistical summarization rules

---(Holmes and Huber, 2021 *(https://web.stanford.edu/class/bios221/book/03-chap.html#the-grammar-of-graphics)*)

Our template can therefore be expanded to include these additional components:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>
  ) +
  <FACET_FUNCTION> +
  <COORDINATE SYSTEM> +
  <THEME>
```

Facets

A way to add variables to a plot beyond mapping them to an aesthetic is to use facets or subplots. There are two primary functions to add facets, `facet_wrap()` and `facet_grid()`. If faceting by a single variable, use `facet_wrap()`. If multiple variables, use `facet_grid()`. The first argument of either function is a formula, with variables separated by a ~ (See below). Variables must be discrete (not continuous).

Let's return to the airway count data to see how facets are useful. Here, we are going to compare scaled and unscaled count data using a density plot.

> A density plot shows the distribution of a numeric variable. --- R Graph Gallery *(https://r-graph-gallery.com/density-plot.html)*

In our example data, `density_data`, the gene counts were scaled to account for technical and composition differences using the trimmed mean of M values (TMM) from EdgeR (Robinson and Oshlack 2010), but non-normalized values remained for comparison. Thus, we can compare scaled vs unscaled counts by sample using faceting.

```
#density plot
```

```
#let's grab the data and take a look
density_data<-read.csv("./data/density_data.csv",
                       stringsAsFactors=TRUE)

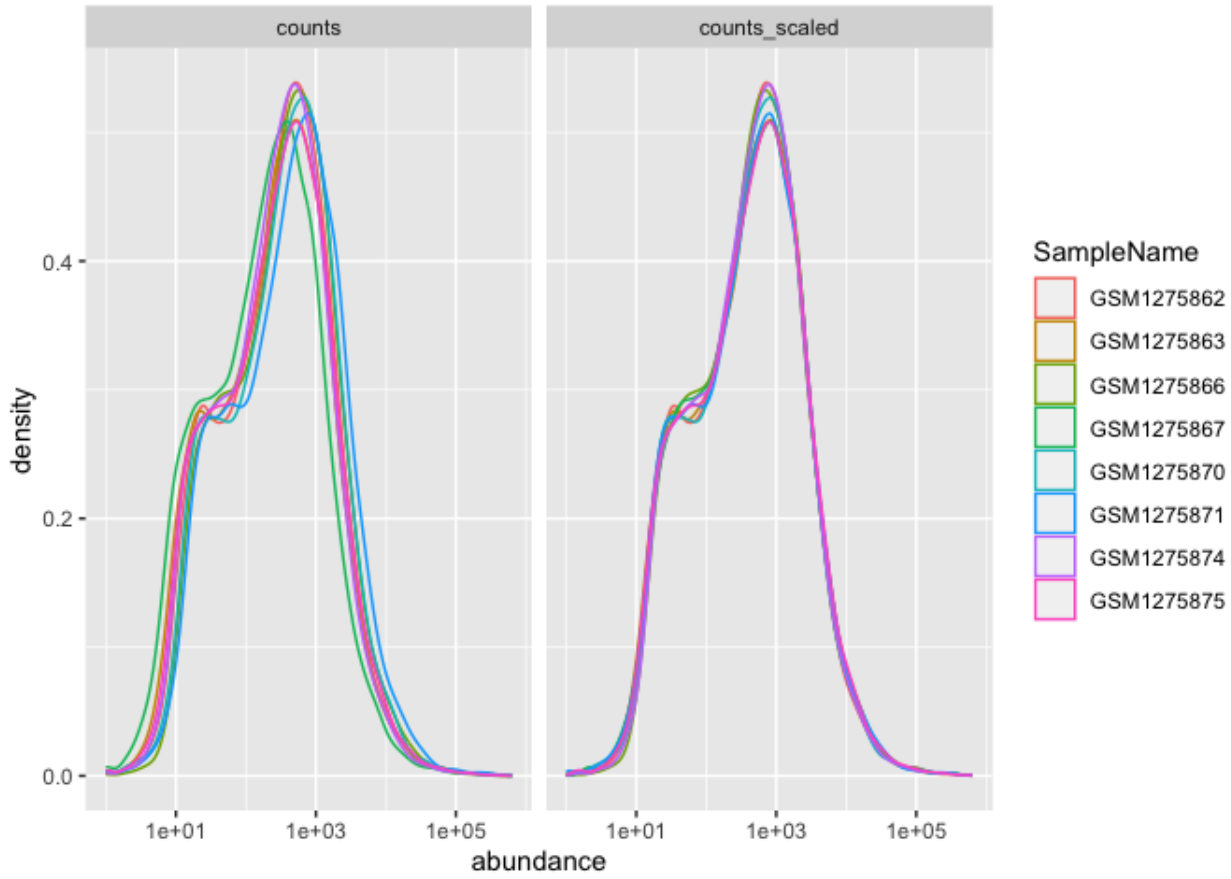
head(density_data)
```

```
##           feature sample SampleName   cell   dex albut        Run
## 1 ENSG00000000003    508 GSM1275862 N61311 untrt untrt SRR1039508
## 2 ENSG00000000003    508 GSM1275862 N61311 untrt untrt SRR1039508
## 3 ENSG00000000419    508 GSM1275862 N61311 untrt untrt SRR1039508
## 4 ENSG00000000419    508 GSM1275862 N61311 untrt untrt SRR1039508
## 5 ENSG00000000457    508 GSM1275862 N61311 untrt untrt SRR1039508
## 6 ENSG00000000457    508 GSM1275862 N61311 untrt untrt SRR1039508
##   Experiment    Sample    BioSample transcript ref_genome .abundar
## 1  SRX384345 SRS508568 SAMN02422669     TSPAN6       hg38      TRU
## 2  SRX384345 SRS508568 SAMN02422669     TSPAN6       hg38      TRU
## 3  SRX384345 SRS508568 SAMN02422669       DPM1       hg38      TRU
## 4  SRX384345 SRS508568 SAMN02422669       DPM1       hg38      TRU
## 5  SRX384345 SRS508568 SAMN02422669      SCYL3       hg38      TRU
## 6  SRX384345 SRS508568 SAMN02422669      SCYL3       hg38      TRU
##   multiplier        source abundance
## 1   1.415149        counts  679.0000
## 2   1.415149 counts_scaled  960.8864
## 3   1.415149        counts  467.0000
## 4   1.415149 counts_scaled  660.8748
## 5   1.415149        counts  260.0000
## 6   1.415149 counts_scaled  367.9388
```

```
#plot
ggplot(data= density_data)+
  aes(x=abundance,
            color=SampleName)+ #initialize ggplot
  geom_density() + #call density plot geom
  facet_wrap(~source) + #use facet_wrap; see ~source
  scale_x_log10()#scales the x axis using a base-10 log transformatic
```

```
## Warning: Transformation introduced infinite values in continuous x
```

```
## Warning: Removed 140 rows containing non-finite values (`stat_dens
```



The distributions of sample counts did not differ greatly between samples before scaling, but regardless, we can see that the distributions are more similar after scaling.

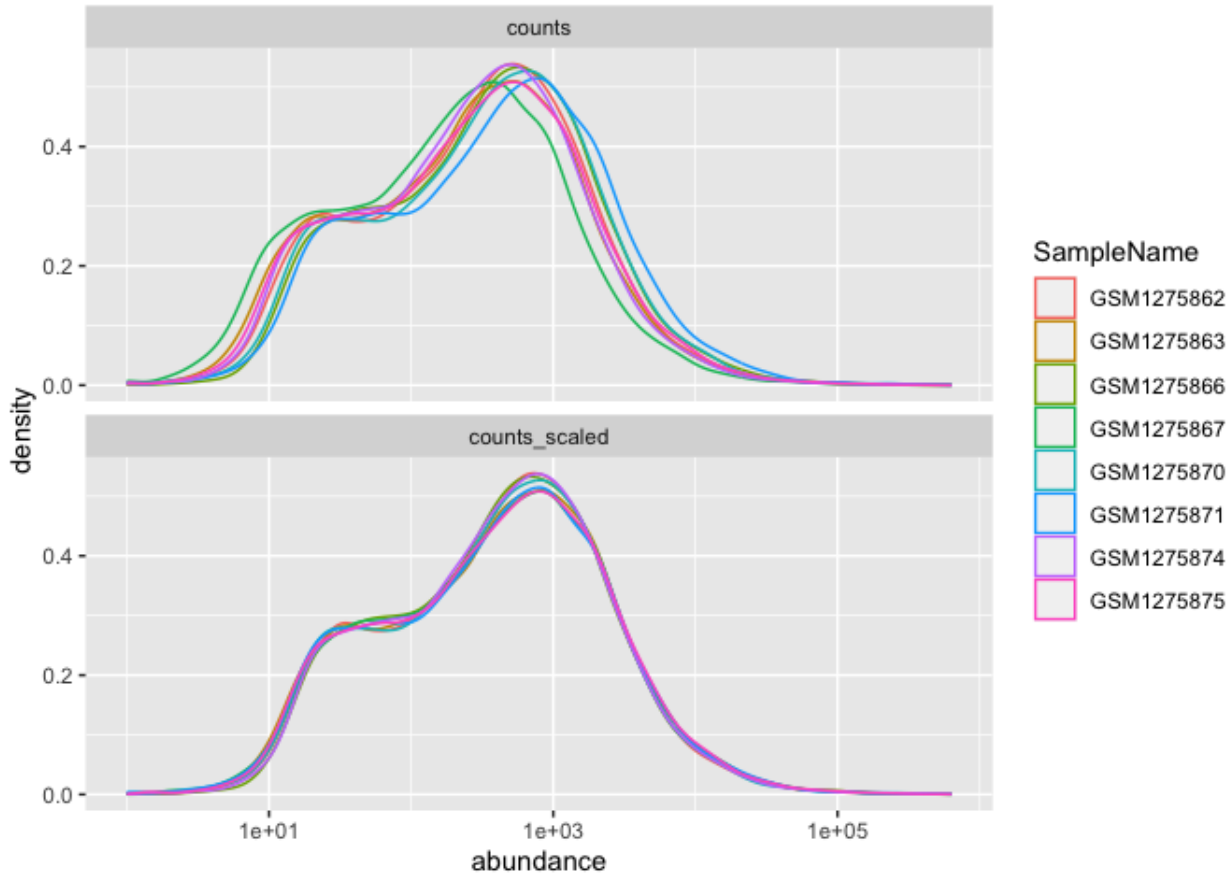
Here, faceting allowed us to visualize multiple features of our data. We were able to see count distributions by sample as well as normalized vs non-normalized counts.

Note the help options with `?facet_wrap()`. How would we make our plot facets vertical rather than horizontal?

```
ggplot(data= density_data)+ #initialize ggplot
  geom_density(aes(x=abundance,
                   color=SampleName)) + #call density plot geom
  facet_wrap(~source, ncol=1) + #use the ncol argument
  scale_x_log10()
```

```
## Warning: Transformation introduced infinite values in continuous x
```

```
## Warning: Removed 140 rows containing non-finite values (`stat_dens
```

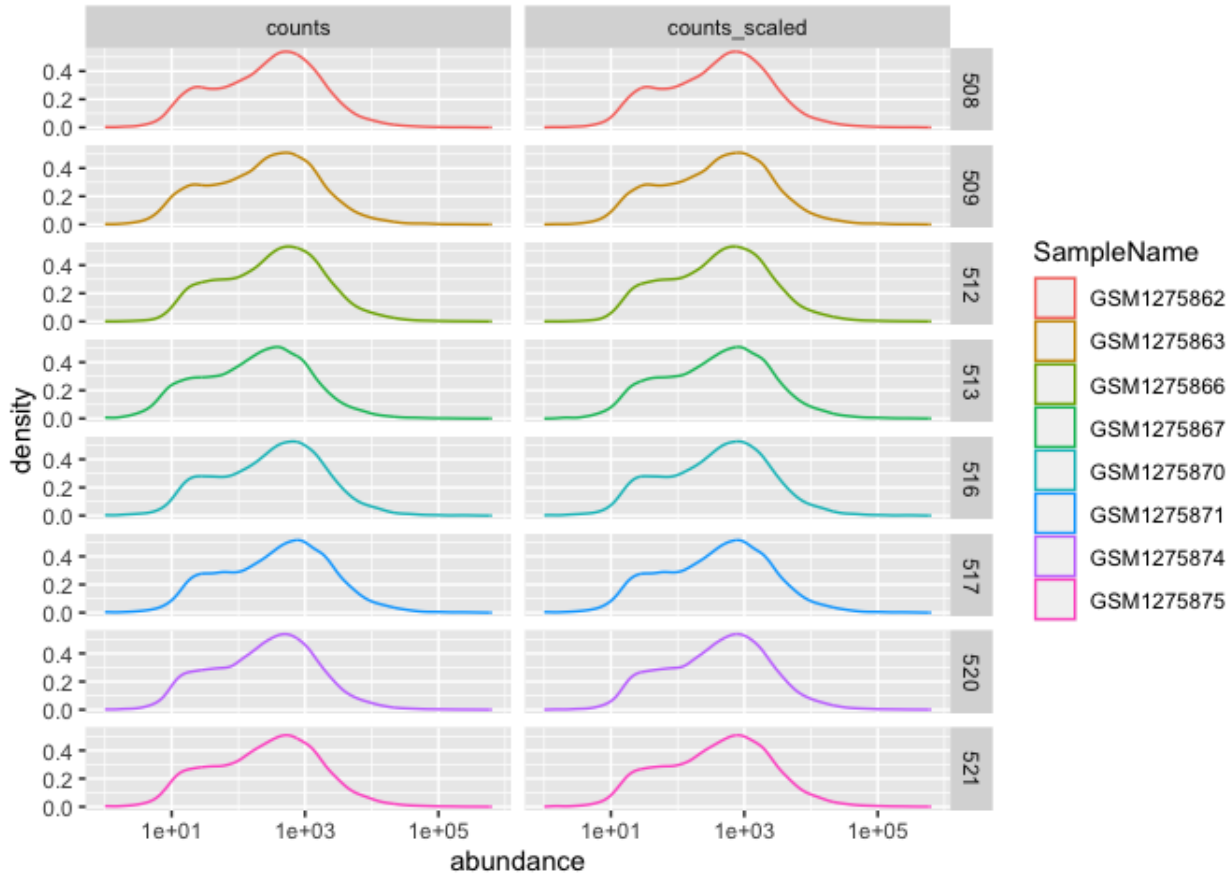


We could plot each sample individually using `facet_grid()`

```
ggplot(data= density_data)+ #initialize ggplot
  geom_density(aes(x=abundance,
                   color=SampleName)) + #call density plot geom
  facet_grid(as.factor(sample)~source) + # formula is sample ~ source
  scale_x_log10()
```

```
## Warning: Transformation introduced infinite values in continuous x
```

```
## Warning: Removed 140 rows containing non-finite values (`stat_dens
```



Labels, legends, scales, and themes

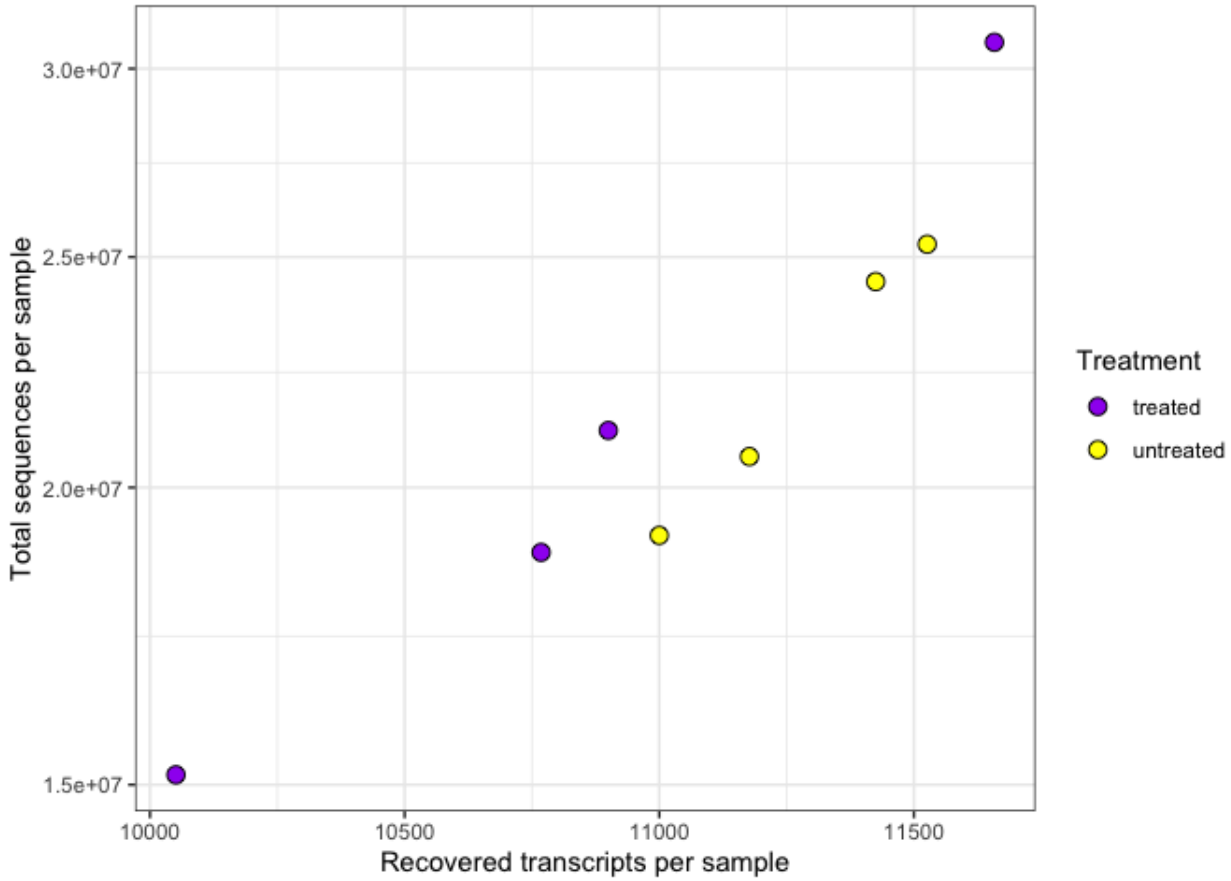
How do we ultimately get our figures to a publishable state? The bread and butter of pretty plots really falls to the additional non-data layers of our ggplot2 code. These layers will include code to label the axes *(https://ggplot2.tidyverse.org/reference/labs.html)*, scale the axes *(https://ggplot2.tidyverse.org/reference/#scales)*, and customize the legends *(https://ggplot2.tidyverse.org/articles/faq-customising.html#legends)* and theme *(https://ggplot2.tidyverse.org/reference/theme.html)*.

Let's make our original figure publishable

```
ggplot(exdata) +
  geom_point(aes(x=Number.of.Transcripts, y = Total.Counts,
                 fill=Treatment),
             shape=21,size=3) +
    #can change labels of fill levels along with colors
  scale_fill_manual(values=c("purple", "yellow"),
                    labels=c('treated','untreated'))+

  labs(x="Recovered transcripts per sample",
       y="Total sequences per sample", fill="Treatment")+
  scale_y_continuous(trans="log10") + #log transform the y axis
```

```
  theme_bw() #add a complete theme black / white
```



Saving plots (`ggsave()`)

Finally, we have a quality plot ready to publish. The next step is to save our plot to a file. The easiest way to do this with ggplot2 is `ggsave()`. This function will save the last plot that you displayed by default. Look at the function parameters using `?ggsave()`.

```
ggsave("Plot1.png",width=5.5,height=3.5,units="in",dpi=300)
```

Resource list

1. ggplot2 cheatsheet
2. The R Graph Gallery *(https://www.r-graph-gallery.com/)*
3. The R Graphics Cookbook *(https://r-graphics.org/recipe-quick-bar)*

Acknowledgements

Material from this lesson was adapted from Chapter 3 of R for Data Science *(https://r4ds.had.co.nz/data-visualisation.html)* and from a 2021 workshop entitled *Introduction to Tidy Transciptomics (https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html)* by Maria Doyle and Stefano Mangiola.

Introduction to `dplyr` and the %>%

Objectives

Today we will begin to wrangle data using the tidyverse package, `dplyr`. To this end, you will learn:

1. how to filter data frames using `dplyr`
2. how to employ the pipe (`%>%`) operator to link functions

What is `dplyr`?

> The package dplyr tries to provide easy tools for the most common data manipulation tasks. It was built to work directly with data frames. The thinking behind it was largely inspired by the package plyr which has been in use for some time but suffered from being slow in some cases. --- datacarpentry.com *(https://datacarpentry.org/genomics-r-intro/05-dplyr/index.html)*

Read more about `dplyr` at https://dplyr.tidyverse.org/articles/programming.html *(https://dplyr.tidyverse.org/articles/programming.html)*.

Loading `dplyr`

We do not need to load the `dplyr` package separately, as it is a core `tidyverse` package. If you need to install and load only `dplyr`, use `install.packages("dplyr")` and `library(dplyr)`.

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidy
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.4     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyverse_
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to f
```

Importing data

For this lesson, we will use sample metadata and differential expression results from the `airway` RNA-Seq project.

Let's begin by importing the data.

```
#sample information
smeta<-read_delim("./data/airway_sampleinfo.txt")
```

```
## Rows: 8 Columns: 9
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (8): SampleName, cell, dex, albut, Run, Experiment, Sample, B
## dbl (1): avgLength
##
## ℹ Use `spec()` to retrieve the full column specification for this
## ℹ Specify the column types or set `show_col_types = FALSE` to quie
```

```
smeta
```

```
## # A tibble: 8 × 9
##   SampleName cell    dex   albut Run       avgLength Experiment Sa
##   <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <c
## 1 GSM1275862 N61311  untrt untrt SRR10395…       126 SRX384345  SR
## 2 GSM1275863 N61311  trt   untrt SRR10395…       126 SRX384346  SR
## 3 GSM1275866 N052611 untrt untrt SRR10395…       126 SRX384349  SR
## 4 GSM1275867 N052611 trt   untrt SRR10395…        87 SRX384350  SR
## 5 GSM1275870 N080611 untrt untrt SRR10395…       120 SRX384353  SR
## 6 GSM1275871 N080611 trt   untrt SRR10395…       126 SRX384354  SR
## 7 GSM1275874 N061011 untrt untrt SRR10395…       101 SRX384357  SR
## 8 GSM1275875 N061011 trt   untrt SRR10395…        98 SRX384358  SR
```

```
#let's use our differential expression results
dexp<-read_delim("./data/diffexp_results_edger_airways.txt")
```

```
## Rows: 15926 Columns: 10
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (4): feature, albut, transcript, ref_genome
```

```
## dbl (5): logFC, logCPM, F, PValue, FDR
## lgl (1): .abundant
##
## ℹ Use `spec()` to retrieve the full column specification for this
## ℹ Specify the column types or set `show_col_types = FALSE` to quie
```

```
dexp
```

```
## # A tibble: 15,926 × 10
##    feature  albut transcript ref_genome .abundant   logFC logCPM
##    <chr>    <chr> <chr>      <chr>      <lgl>       <dbl>  <dbl>
##  1 ENSG000… untrt TSPAN6     hg38       TRUE      -0.390    5.06 3
##  2 ENSG000… untrt DPM1       hg38       TRUE       0.198    4.61
##  3 ENSG000… untrt SCYL3      hg38       TRUE       0.0292   3.48
##  4 ENSG000… untrt C1orf112   hg38       TRUE      -0.124    1.47
##  5 ENSG000… untrt CFH        hg38       TRUE       0.417    8.09 2
##  6 ENSG000… untrt FUCA2      hg38       TRUE      -0.250    5.91 1
##  7 ENSG000… untrt GCLC       hg38       TRUE      -0.0581   4.84
##  8 ENSG000… untrt NFYA       hg38       TRUE      -0.509    4.13 4
##  9 ENSG000… untrt STPG1      hg38       TRUE      -0.136    3.12
## 10 ENSG000… untrt NIPAL3     hg38       TRUE      -0.0500   7.04
## # ℹ 15,916 more rows
## # ℹ 1 more variable: FDR <dbl>
```

We can get an idea of the structure of these data by using `str()` or `glimpse()`. `glimpse()`, from tidyverse, is similar to `str()` but provides somewhat cleaner output.

```
glimpse(smeta)
```

```
## Rows: 8
## Columns: 9
## $ SampleName <chr> "GSM1275862", "GSM1275863", "GSM1275866", "GSM1
## $ cell       <chr> "N61311", "N61311", "N052611", "N052611", "N080
## $ dex        <chr> "untrt", "trt", "untrt", "trt", "untrt", "trt"
## $ albut      <chr> "untrt", "untrt", "untrt", "untrt", "untrt", "u
## $ Run        <chr> "SRR1039508", "SRR1039509", "SRR1039512", "SRR1
## $ avgLength  <dbl> 126, 126, 126, 87, 120, 126, 101, 98
## $ Experiment <chr> "SRX384345", "SRX384346", "SRX384349", "SRX3843
## $ Sample     <chr> "SRS508568", "SRS508567", "SRS508571", "SRS5085
## $ BioSample  <chr> "SAMN02422669", "SAMN02422675", "SAMN02422678"
```

```
glimpse(dexp)
```

```
## Rows: 15,926
## Columns: 10
## $ feature    <chr> "ENSG00000000003", "ENSG00000000419", "ENSG0000
## $ albut      <chr> "untrt", "untrt", "untrt", "untrt", "untrt", "u
## $ transcript <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH", "
## $ ref_genome <chr> "hg38", "hg38", "hg38", "hg38", "hg38", "hg38"
## $ .abundant  <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE
## $ logFC      <dbl> -0.390100222, 0.197802179, 0.029160865, -0.124
## $ logCPM     <dbl> 5.059704, 4.611483, 3.482462, 1.473375, 8.0891
## $ F          <dbl> 3.284948e+01, 6.903534e+00, 9.685073e-02, 3.77
## $ PValue     <dbl> 0.0003117656, 0.0280616149, 0.7629129276, 0.554
## $ FDR        <dbl> 0.002831504, 0.077013489, 0.844247837, 0.68232
```

Now that we have some data to work with, let's start subsetting.

Subsetting data in base R

Base R uses bracket notation for subsetting. For example, if we want to subset the data frame `iris` to include only the first 5 rows and the first 3 columns, we could use

```
iris[1:5,1:3]
```

```
##   Sepal.Length Sepal.Width Petal.Length
## 1          5.1         3.5          1.4
## 2          4.9         3.0          1.4
## 3          4.7         3.2          1.3
## 4          4.6         3.1          1.5
## 5          5.0         3.6          1.4
```

While this type of subsetting is useful, it is not always the most readable or easy to employ, especially for beginners. This is where `dplyr` comes in. The `dplyr` package in the `tidyverse` world simplifies data wrangling with easy to employ and easy to understand functions specific for data manipulation in data frames.

Subsetting with `dplyr`

How can we select only columns of interest and rows of interest? We can use `select()` and `filter()` from `dplyr`.

Subsetting by column (`select()`)

To subset by column, we use the function `select()`. We can include and exclude columns, reorder columns, and rename columns using `select()`.

Select a few columns from our differential expression results (`dexp`).

We can select the columns we are interested in by first calling the data frame object (`dexp`) followed by the columns we want to select (`transcript`,`logFC`,`FDR`). All arguments are separated by a comma. The order of the arguments will determine the order of the columns in the new data frame.

```
#select the gene / transcript, logFC, and FDR corrected p-value
#first argument is the df followed by columns to select
ex1<-select(dexp, transcript, logFC, FDR)
ex1
```

```
## # A tibble: 15,926 × 3
##    transcript   logFC     FDR
##    <chr>         <dbl>   <dbl>
##  1 TSPAN6     -0.390  0.00283
##  2 DPM1        0.198  0.0770
##  3 SCYL3       0.0292 0.844
##  4 C1orf112   -0.124  0.682
##  5 CFH         0.417  0.00376
##  6 FUCA2      -0.250  0.0186
##  7 GCLC       -0.0581 0.794
##  8 NFYA       -0.509  0.00126
##  9 STPG1      -0.136  0.478
## 10 NIPAL3     -0.0500 0.695
## # ℹ 15,916 more rows
```

We can rename while selecting.

```
#rename using the syntax new_name = old_name
ex1<-select(dexp, gene=transcript, logFoldChange = logFC, FDRpvalue=F
ex1
```

```
## # A tibble: 15,926 × 3
##    gene     logFoldChange FDRpvalue
##    <chr>            <dbl>     <dbl>
##  1 TSPAN6         -0.390    0.00283
```

```
##  2 DPM1            0.198    0.0770
##  3 SCYL3           0.0292   0.844
##  4 C1orf112       -0.124    0.682
##  5 CFH             0.417    0.00376
##  6 FUCA2          -0.250    0.0186
##  7 GCLC           -0.0581   0.794
##  8 NFYA           -0.509    0.00126
##  9 STPG1          -0.136    0.478
## 10 NIPAL3         -0.0500   0.695
## # ℹ 15,916 more rows
```

Note

If you want to retain all columns, you could also use `rename()` *(https://dplyr.tidyverse.org/reference/rename.html)* from `dplyr` to rename columns.

Excluding columns

We can select all columns, leaving out ones that do not interest us using a `-` sign. This is helpful if the columns to keep far outweigh those to exclude. We can similarly use the `!` to negate a selection.

```
ex2<-select(dexp, -feature)
ex2
```

```
## # A tibble: 15,926 × 9
##    albut transcript ref_genome .abundant   logFC logCPM       F
##    <chr> <chr>      <chr>      <lgl>       <dbl>  <dbl>   <dbl>
##  1 untrt TSPAN6     hg38       TRUE      -0.390    5.06 32.8    0
##  2 untrt DPM1       hg38       TRUE       0.198    4.61  6.90   0
##  3 untrt SCYL3      hg38       TRUE       0.0292   3.48  0.0969 0
##  4 untrt C1orf112   hg38       TRUE      -0.124    1.47  0.377  0
##  5 untrt CFH        hg38       TRUE       0.417    8.09 29.3    0
##  6 untrt FUCA2      hg38       TRUE      -0.250    5.91 14.9    0
##  7 untrt GCLC       hg38       TRUE      -0.0581   4.84  0.167  0
##  8 untrt NFYA       hg38       TRUE      -0.509    4.13 44.9    0
##  9 untrt STPG1      hg38       TRUE      -0.136    3.12  1.04   0
## 10 untrt NIPAL3     hg38       TRUE      -0.0500   7.04  0.350  0
## # ℹ 15,916 more rows
```

```
ex2<-select(dexp, !feature)
ex2
```

```
## # A tibble: 15,926 × 9
##    albut transcript ref_genome .abundant   logFC logCPM       F
##    <chr> <chr>      <chr>      <lgl>       <dbl>  <dbl>   <dbl>
##  1 untrt TSPAN6     hg38       TRUE      -0.390    5.06 32.8    0
##  2 untrt DPM1       hg38       TRUE       0.198    4.61  6.90   0
##  3 untrt SCYL3      hg38       TRUE       0.0292   3.48  0.0969 0
##  4 untrt C1orf112   hg38       TRUE      -0.124    1.47  0.377  0
##  5 untrt CFH        hg38       TRUE       0.417    8.09 29.3    0
##  6 untrt FUCA2      hg38       TRUE      -0.250    5.91 14.9    0
##  7 untrt GCLC       hg38       TRUE      -0.0581   4.84  0.167  0
##  8 untrt NFYA       hg38       TRUE      -0.509    4.13 44.9    0
##  9 untrt STPG1      hg38       TRUE      -0.136    3.12  1.04   0
## 10 untrt NIPAL3     hg38       TRUE      -0.0500   7.04  0.350  0
## # ℹ 15,916 more rows
```

We can **reorder** using `select()`.

For readability, let's move the transcript column to the front.

```
#you can reorder columns and call a range of columns using select().
ex3<-select(dexp, transcript:FDR,albut)
ex3
```

```
## # A tibble: 15,926 × 9
##    transcript ref_genome .abundant   logFC logCPM       F   PValue
##    <chr>      <chr>      <lgl>       <dbl>  <dbl>   <dbl>    <dbl>
##  1 TSPAN6     hg38       TRUE      -0.390    5.06 32.8    0.00031
##  2 DPM1       hg38       TRUE       0.198    4.61  6.90   0.0281
##  3 SCYL3      hg38       TRUE       0.0292   3.48  0.0969 0.763
##  4 C1orf112   hg38       TRUE      -0.124    1.47  0.377  0.555
##  5 CFH        hg38       TRUE       0.417    8.09 29.3    0.00046
##  6 FUCA2      hg38       TRUE      -0.250    5.91 14.9    0.00405
##  7 GCLC       hg38       TRUE      -0.0581   4.84  0.167  0.692
##  8 NFYA       hg38       TRUE      -0.509    4.13 44.9    0.00010
##  9 STPG1      hg38       TRUE      -0.136    3.12  1.04   0.335
## 10 NIPAL3     hg38       TRUE      -0.0500   7.04  0.350  0.569
## # ℹ 15,916 more rows
```

Note

This also would have excluded the feature column.

Selecting a range of columns

Notice that we can select a range of columns using the `:`. We could also deselect a range of columns or deselect a range of columns while adding a column back.

```
ex3<-select(dexp, -(albut:F),logFC)
ex3
```

```
## # A tibble: 15,926 × 4
##    feature           PValue     FDR   logFC
##    <chr>              <dbl>   <dbl>   <dbl>
##  1 ENSG00000000003 0.000312 0.00283 -0.390
##  2 ENSG00000000419 0.0281   0.0770   0.198
##  3 ENSG00000000457 0.763    0.844    0.0292
##  4 ENSG00000000460 0.555    0.682   -0.124
##  5 ENSG00000000971 0.000463 0.00376  0.417
##  6 ENSG00000001036 0.00405  0.0186  -0.250
##  7 ENSG00000001084 0.692    0.794   -0.0581
##  8 ENSG00000001167 0.000100 0.00126 -0.509
##  9 ENSG00000001460 0.335    0.478   -0.136
## 10 ENSG00000001461 0.569    0.695   -0.0500
## # ℹ 15,916 more rows
```

Helper functions

We can also include helper functions such as `starts_with()` and `ends_with()`

```
select(dexp, transcript, starts_with("log"), FDR)
```

```
## # A tibble: 15,926 × 4
##    transcript   logFC logCPM     FDR
##    <chr>        <dbl>  <dbl>   <dbl>
##  1 TSPAN6     -0.390    5.06 0.00283
##  2 DPM1        0.198    4.61 0.0770
##  3 SCYL3       0.0292   3.48 0.844
##  4 C1orf112   -0.124    1.47 0.682
##  5 CFH         0.417    8.09 0.00376
##  6 FUCA2      -0.250    5.91 0.0186
##  7 GCLC       -0.0581   4.84 0.794
##  8 NFYA       -0.509    4.13 0.00126
##  9 STPG1      -0.136    3.12 0.478
```

```
## 10 NIPAL3     -0.0500   7.04 0.695
## # ℹ 15,916 more rows
```

There are a number of other **selection helpers**. See the help documentation for `select` *(https://dplyr.tidyverse.org/reference/select.html)* for more information `?dplyr::select()`.

Select columns of a particular type

There are many other ways to select multiple columns. You may commonly be interested in selecting all numeric columns or all factors. The syntax below can be used for this purpose.

```
select(dexp, where(is.numeric)) #or
```

```
## # A tibble: 15,926 × 5
##      logFC logCPM       F   PValue     FDR
##      <dbl>  <dbl>   <dbl>    <dbl>   <dbl>
##  1 -0.390    5.06 32.8    0.000312 0.00283
##  2  0.198    4.61  6.90   0.0281   0.0770
##  3  0.0292   3.48  0.0969 0.763    0.844
##  4 -0.124    1.47  0.377  0.555    0.682
##  5  0.417    8.09 29.3    0.000463 0.00376
##  6 -0.250    5.91 14.9    0.00405  0.0186
##  7 -0.0581   4.84  0.167  0.692    0.794
##  8 -0.509    4.13 44.9    0.000100 0.00126
##  9 -0.136    3.12  1.04   0.335    0.478
## 10 -0.0500   7.04  0.350  0.569    0.695
## # ℹ 15,916 more rows
```

```
select_if(dexp, is.numeric) #select_if is a scoped verb function
```

```
## # A tibble: 15,926 × 5
##      logFC logCPM       F   PValue     FDR
##      <dbl>  <dbl>   <dbl>    <dbl>   <dbl>
##  1 -0.390    5.06 32.8    0.000312 0.00283
##  2  0.198    4.61  6.90   0.0281   0.0770
##  3  0.0292   3.48  0.0969 0.763    0.844
##  4 -0.124    1.47  0.377  0.555    0.682
##  5  0.417    8.09 29.3    0.000463 0.00376
##  6 -0.250    5.91 14.9    0.00405  0.0186
##  7 -0.0581   4.84  0.167  0.692    0.794
##  8 -0.509    4.13 44.9    0.000100 0.00126
##  9 -0.136    3.12  1.04   0.335    0.478
```

```
## 10 -0.0500   7.04  0.350  0.569    0.695
## # ℹ 15,916 more rows
```

Subsetting by row (`filter()`)

To subset by row, we use the function `filter()`.

> filter() only includes rows where the condition is TRUE; it excludes both FALSE and NA values. ---R4DS *(https://r4ds.had.co.nz/transform.html#filter-rows-with-filter)*

Now let's filter the rows from `smeta` based on a condition. Let's look at only the treated samples in `dex` (i.e., `trt`) using the function `filter()`. The first argument is the data frame (e.g., `smeta`) followed by the expression(s) to filter the data frame.

```
filter(smeta, dex == "trt") #we've seen == notation before
```

To complete these filter phrases you will often need to include comparison operators such as the == above. These operators help us evaluate relations. For example, `a == b` is asking if `a` and `b` are equivalent. It is a logical comparison that when evaluated will return TRUE or FALSE. The filter function will then return rows that evaluate to TRUE.

Try the following:

```
a <- 1
b <- 1
a == b
```

```
## [1] TRUE
```

Keep these comparison operators in mind for filtering.

Comparison operators

Comparison Operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	Not equal
==	equal

Comparison Operator	Description
a | b	a or b
a & b	a and b

We may want to combine filtering parameters using AND or OR phrasing and the operators & and |.

For example, if we only wanted to return rows where `dex == trt` and `cell==N61311`, we can use:

```
filter(smeta, dex == "trt" & cell == "N61311")
```

```
## # A tibble: 1 × 9
##   SampleName cell   dex   albut Run        avgLength Experiment Sa
##   <chr>      <chr>  <chr> <chr> <chr>          <dbl> <chr>      <c
## 1 GSM1275863 N61311 trt   untrt SRR1039509       126 SRX384346  SR
```

A , is treated the same as & in the case of `filter()`.

```
filter(smeta, dex == "trt", cell == "N61311")
```

```
## # A tibble: 1 × 9
##   SampleName cell   dex   albut Run        avgLength Experiment Sa
##   <chr>      <chr>  <chr> <chr> <chr>          <dbl> <chr>      <c
## 1 GSM1275863 N61311 trt   untrt SRR1039509       126 SRX384346  SR
```

We can also filter by one condition or another using the |.

```
filter(smeta,cell == "N080611" | cell == "N61311")
```

```
## # A tibble: 4 × 9
##   SampleName cell    dex   albut Run        avgLength Experiment Sa
##   <chr>      <chr>   <chr> <chr> <chr>          <dbl> <chr>      <c
## 1 GSM1275862 N61311  untrt untrt SRR10395…       126 SRX384345  SR
## 2 GSM1275863 N61311  trt   untrt SRR10395…       126 SRX384346  SR
## 3 GSM1275870 N080611 untrt untrt SRR10395…       120 SRX384353  SR
## 4 GSM1275871 N080611 trt   untrt SRR10395…       126 SRX384354  SR
```

The %in% operator

Used to match elements of a vector.

> %in% returns a logical vector indicating if there is a match or not for its left operand.
> --- match R Documentation.

The returned logical vector will be the length of the vector to the left. Its basic usage:

```
smeta$SampleName %in% c("GSM1275871","GSM1275863")
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
c("GSM1275871","GSM1275863") %in% smeta$SampleName
```

```
## [1] TRUE TRUE
```

We can combine the `%in%` operator with `filter()`.

```
#filter for two cell lines
filter(smeta,cell %in% c("N061011", "N052611"))
```

```
## # A tibble: 4 × 9
##   SampleName cell    dex   albut Run       avgLength Experiment Sa
##   <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <c
## 1 GSM1275866 N052611 untrt untrt SRR10395…       126 SRX384349  SR
## 2 GSM1275867 N052611 trt   untrt SRR10395…        87 SRX384350  SR
## 3 GSM1275874 N061011 untrt untrt SRR10395…       101 SRX384357  SR
## 4 GSM1275875 N061011 trt   untrt SRR10395…        98 SRX384358  SR
```

Including multiple phrases

```
#use `|` operator
#look at only results with named genes (not NAs)
#and those with a log fold change greater than 2
#and either a p-value or an FDR corrected p_value < or = to 0.01
#The comma acts as &
sig_annot_transcripts<-
  filter(dexp, !is.na(transcript),
```

```
        abs(logFC) > 2, (PValue | FDR <= 0.01))
```

Filtering across columns

Past versions of dplyr included powerful variants of `filter`, `select`, and other functions to help perform tasks across columns. You may see functions such as `filter_all`, `filter_if`, and `filter_at`. Functions like these can still be used but have been superseded by `across` *(https://dplyr.tidyverse.org/reference/across.html)*. However, `across` has been deprecated in the case of filter and replaced by `if_any()` and `if_all()`.

> Both functions operate similarly to across() but go the extra mile of aggregating the results to indicate if all the results are true when using if_all(), or if at least one is true when using if_any() ---tidyverse.org *(https://www.tidyverse.org/blog/2021/02/dplyr-1-0-4-if-any/)*

Let's briefly see this in action.

```
f<-filter(dexp, if_all(PValue:FDR, ~ . < 0.05))
```

Anonymous functions

The code above includes an anonymous function. Read more here *(https://jennybc.github.io/purrr-tutorial/ls03_map-function-syntax.html#anonymous_function,_formula)*. You may also find this stackoverflow post *(https://stackoverflow.com/questions/56532119/dplyr-piping-data-difference-between-and-x)* useful.

Subsetting rows by position

There are times when you may want to subset your data by position, for example, the first or last number of rows. There are a series of functions in the tidyverse that facilitate this type of subsetting. The primary function is `slice()`, which has several commonly used helper functions including `slice_head()`, `slice_tail()`, `slice_min()`, and `slice_max()`. See the `slice()` *(https://dplyr.tidyverse.org/reference/slice.html)* documentation for more information.

Introducing the pipe

Often we will apply multiple functions to wrangle a data frame into the state that we need it. For example, maybe you want to select and filter. What are our options? We could run one step after another, saving an object for each step, or we could nest a function within a function, but these can affect code readability and clutter our work space, making it difficult to follow what we or someone else did.

For example,

```
#Run one step at a time with intermediate objects.
#We've done this a few times above
#select gene, logFC, FDR
dexp_s<-select(dexp, transcript, logFC, FDR)

#Now filter for only the genes "TSPAN6" and DPM1
#Note: we could have used %in%
tspanDpm<- filter(dexp_s, transcript == "TSPAN6" | transcript=="DPM1"

#Nested code example
tspanDpm<- filter(select(dexp, c(transcript, logFC, FDR)),
                  transcript == "TSPAN6" | transcript=="DPM1" )
```

Let's explore how piping streamlines this. Piping (using `%>%`) allows you to employ multiple functions consecutively, while improving readability. The output of one function is passed directly to another without storing the intermediate steps as objects. You can pipe from the beginning (reading in the data) all the way to plotting without storing the data or intermediate objects, *if you want*. Pipes in R come from the `magrittr` package, which is a dependency of `dplyr`.

Pipe

Read more info about the magrittr pipe here *(https://magrittr.tidyverse.org/reference/pipe.html)*. There is also a native R pipe, `|>`, as of `R 4.1.0`. Read more about the difference between `%>%` and `|>` here *(https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/)*.

To pipe, we have to first call the data and then pipe it into a function. The output of each step is then piped into the next step.

Let's see how this works

```
tspanDpm <- dexp %>% #call the data and pipe to select()
  select(transcript, logFC, FDR) %>% #select columns of interest
  filter(transcript == "TSPAN6" | transcript=="DPM1" ) #filter
```

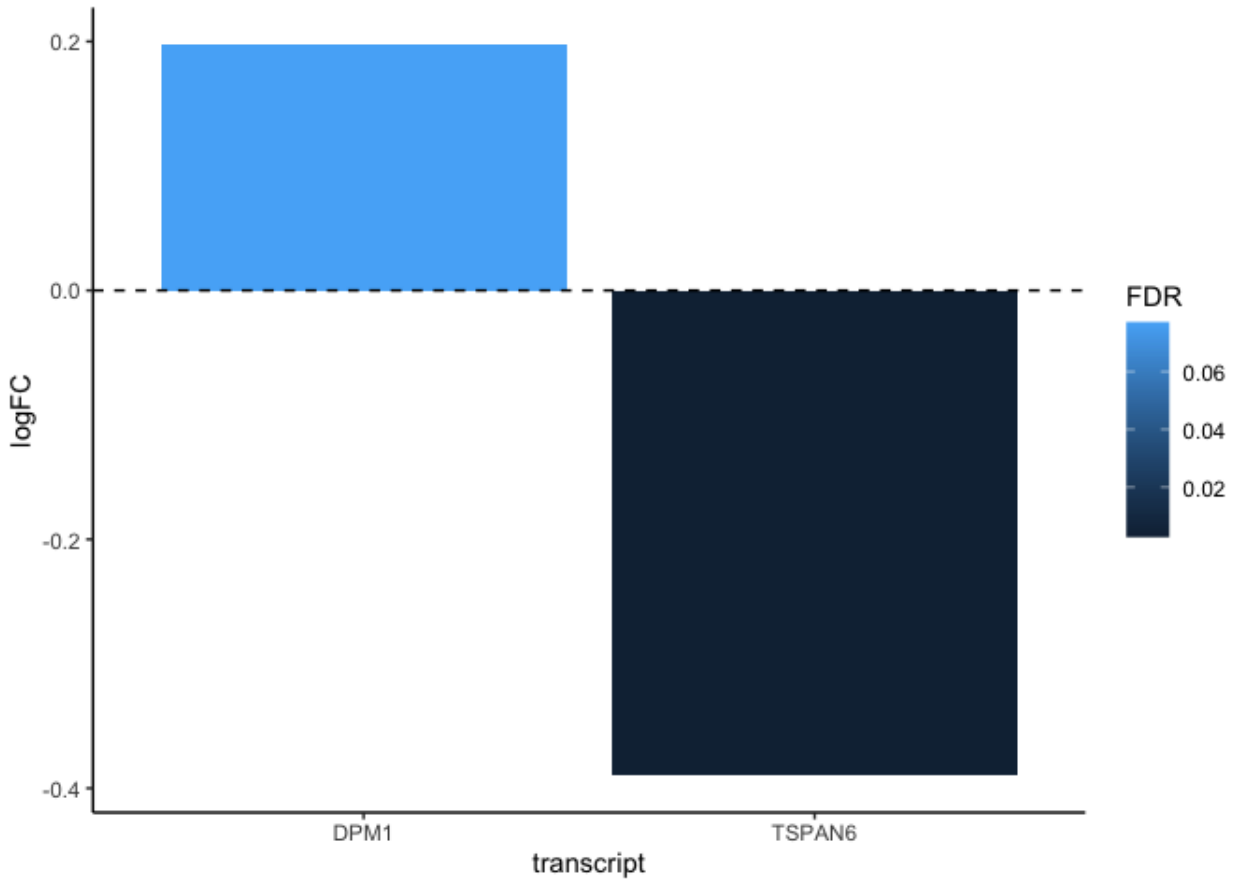
Notice that the data argument has been dropped from `select()` and `filter()`. This is because the pipe passes the input from the left to the right. The `%>%` must be at the end of each line.

Piping from the beginning:

```
read_delim("./data/diffexp_results_edger_airways.txt") %>% #read data
  select(transcript, logFC, FDR) %>% #select columns of interest
  filter(transcript == "TSPAN6" | transcript=="DPM1" ) %>% #filter
```

```
  ggplot(aes(x=transcript,y=logFC,fill=FDR)) + #plot
  geom_bar(stat = "identity") +
  theme_classic() +
  geom_hline(yintercept=0, linetype="dashed", color = "black")
```

```
## Rows: 15926 Columns: 10
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (4): feature, albut, transcript, ref_genome
## dbl (5): logFC, logCPM, F, PValue, FDR
## lgl (1): .abundant
##
## ℹ Use `spec()` to retrieve the full column specification for this
## ℹ Specify the column types or set `show_col_types = FALSE` to quie
```



The dplyr functions by themselves are somewhat simple, but by combining them into linear workflows with the pipe, we can accomplish more complex manipulations of data frames. ---datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/05-dplyr/index.html)*

Reordering rows

There are many steps that can be taken following subsetting (i.e., filtering by rows and columns); one of which is reordering rows. In the tidyverse, reordering rows is largely done by `arrange()`. Arrange will reorder a variable from smallest to largest, or in the case of characters, alphabetically, from a to z.

Let's arrange the genes in `dexp`.

```
dexp %>% arrange(transcript)
```

```
## # A tibble: 15,926 × 10
##    feature   albut transcript ref_genome .abundant   logFC logCPM
##    <chr>     <chr> <chr>      <chr>      <lgl>       <dbl>  <dbl>
##  1 ENSG0000… untrt A1BG-AS1   hg38       TRUE        0.513   1.02
##  2 ENSG0000… untrt A2M        hg38       TRUE        0.528  10.1
##  3 ENSG0000… untrt A2M-AS1    hg38       TRUE       -0.337   0.308
##  4 ENSG0000… untrt A4GALT     hg38       TRUE        0.519   5.89
##  5 ENSG0000… untrt AAAS       hg38       TRUE       -0.0254  5.12
##  6 ENSG0000… untrt AACS       hg38       TRUE       -0.191   4.06
##  7 ENSG0000… untrt AADAT      hg38       TRUE       -0.642   2.67
##  8 ENSG0000… untrt AAGAB      hg38       TRUE       -0.165   5.08
##  9 ENSG0000… untrt AAK1       hg38       TRUE       -0.188   3.82
## 10 ENSG0000… untrt AAMDC      hg38       TRUE        0.447   2.42
## # ℹ 15,916 more rows
## # ℹ 1 more variable: FDR <dbl>
```

Let's arrange `logFC` from smallest to largest.

```
dexp %>% arrange(logFC)
```

```
## # A tibble: 15,926 × 10
##    feature     albut transcript ref_genome .abundant logFC  logCPM
##    <chr>       <chr> <chr>      <chr>      <lgl>     <dbl>   <dbl>
##  1 ENSG000002… untrt LINC00906  hg38       TRUE      -4.59  0.473
##  2 ENSG000001… untrt LRRTM2     hg38       TRUE      -4.00  1.24
##  3 ENSG000001… untrt VASH2      hg38       TRUE      -3.95  0.017
##  4 ENSG000001… untrt VCAM1      hg38       TRUE      -3.66  4.60
##  5 ENSG000001… untrt SLC14A1    hg38       TRUE      -3.63  1.38
##  6 ENSG000002… untrt FER1L6     hg38       TRUE      -3.13  3.53
##  7 ENSG000001… untrt SMTNL2     hg38       TRUE      -3.12  1.46
##  8 ENSG000001… untrt WNT2       hg38       TRUE      -3.07  3.99
```

```
##  9 ENSG000001… untrt EGR2        hg38        TRUE       -3.04 -0.141
## 10 ENSG000001… untrt SLITRK6     hg38        TRUE       -3.03  1.16
## # ℹ 15,916 more rows
## # ℹ 1 more variable: FDR <dbl>
```

What if we want to arrange from largest to smallest? We can use `desc()`.

```
dexp %>% arrange(desc(logFC))
```

```
## # A tibble: 15,926 × 10
##    feature    albut transcript ref_genome .abundant logFC  logCPM
##    <chr>      <chr> <chr>      <chr>      <lgl>     <dbl>   <dbl>
##  1 ENSG00000… untrt ALOX15B    hg38       TRUE      10.1   1.62
##  2 ENSG00000… untrt ZBTB16     hg38       TRUE       7.15  4.15
##  3 ENSG00000… untrt <NA>       <NA>       TRUE       6.17  1.35
##  4 ENSG00000… untrt ANGPTL7    hg38       TRUE       5.68  3.51
##  5 ENSG00000… untrt STEAP4     hg38       TRUE       5.22  3.66
##  6 ENSG00000… untrt PRODH      hg38       TRUE       4.85  1.29
##  7 ENSG00000… untrt FAM107A    hg38       TRUE       4.74  2.78
##  8 ENSG00000… untrt LGI3       hg38       TRUE       4.68 -0.0503
##  9 ENSG00000… untrt SPARCL1    hg38       TRUE       4.56  5.53
## 10 ENSG00000… untrt KLF15      hg38       TRUE       4.48  4.69
## # ℹ 15,916 more rows
## # ℹ 1 more variable: FDR <dbl>
```

Acknowledgments

Some material from this lesson was either taken directly or adapted from the Intro to R and RStudio for Genomics lesson provided by datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html)*. Additional content was inspired by Chapter 3, Wrangling Data in the Tidyverse, *(https://jhudatascience.org/tidyversecourse/wrangle-data.html#filtering-data)* from *Tidyverse Skills for Data Science* and Suzan Baert's dplyr tutorials *(https://suzan.rbind.io/categories/tutorial/)*.

`dplyr`: joining, tranforming, and summarizing data frames

Objectives

Today we will continue to wrangle data using the tidyverse package, `dplyr`. We will learn:

1. how to join data frames using `dplyr`
2. how to transform and create new variables using `mutate()`
3. how to summarize variables using `group_by()` and `summarize()`

Loading `dplyr`

In this lesson, we are continuing with the package `dplyr`. We do not need to load the `dplyr` package separately, as it is a core `tidyverse` package. Again, if you need to install and load only `dplyr`, use `install.packages("dplyr")` and `library(dplyr)`.

Load the package:

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidy
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.4     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyverse_
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to f
```

Data

Let's load in some data to work with. In this lesson, we will continue to use sample metadata, raw count data, and differential expression results from the `airway` RNA-Seq project.

Load the data:

```
#sample information
smeta<-read_delim("./data/airway_sampleinfo.txt")
```

```
## Rows: 8 Columns: 9
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (8): SampleName, cell, dex, albut, Run, Experiment, Sample, B
## dbl (1): avgLength
##
## ℹ Use `spec()` to retrieve the full column specification for this
## ℹ Specify the column types or set `show_col_types = FALSE` to quie
```

```
smeta
```

```
## # A tibble: 8 × 9
##   SampleName cell    dex   albut Run       avgLength Experiment Sa
##   <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <c
## 1 GSM1275862 N61311  untrt untrt SRR10395…       126 SRX384345  SR
## 2 GSM1275863 N61311  trt   untrt SRR10395…       126 SRX384346  SR
## 3 GSM1275866 N052611 untrt untrt SRR10395…       126 SRX384349  SR
## 4 GSM1275867 N052611 trt   untrt SRR10395…        87 SRX384350  SR
## 5 GSM1275870 N080611 untrt untrt SRR10395…       120 SRX384353  SR
## 6 GSM1275871 N080611 trt   untrt SRR10395…       126 SRX384354  SR
## 7 GSM1275874 N061011 untrt untrt SRR10395…       101 SRX384357  SR
## 8 GSM1275875 N061011 trt   untrt SRR10395…        98 SRX384358  SR
```

```
#raw count data
acount<-read_csv("./data/airway_rawcount.csv") %>%
  dplyr::rename("Feature" = "...1")
```

```
## New names:
## Rows: 64102 Columns: 9
## ── Column specification
## ──────────────────────────────────────────────────────── Delimiter
## (1): ...1 dbl (8): SRR1039508, SRR1039509, SRR1039512, SRR1039513
## SRR1039...
## ℹ Use `spec()` to retrieve the full column specification for this
## Specify the column types or set `show_col_types = FALSE` to quiet
## • `` -> `...1`
```

```
acount
```

```
## # A tibble: 64,102 × 9
##    Feature     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR103
##    <chr>            <dbl>      <dbl>      <dbl>      <dbl>      <
##  1 ENSG000000…        679        448        873        408
##  2 ENSG000000…          0          0          0          0
##  3 ENSG000000…        467        515        621        365
##  4 ENSG000000…        260        211        263        164
##  5 ENSG000000…         60         55         40         35
##  6 ENSG000000…          0          0          2          0
##  7 ENSG000000…       3251       3679       6177       4252
##  8 ENSG000000…       1433       1062       1733        881
##  9 ENSG000000…        519        380        595        493
## 10 ENSG000000…        394        236        464        175
## # ℹ 64,092 more rows
## # ℹ 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

```
#differential expression results
dexp<-read_delim("./data/diffexp_results_edger_airways.txt")
```

```
## Rows: 15926 Columns: 10
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (4): feature, albut, transcript, ref_genome
## dbl (5): logFC, logCPM, F, PValue, FDR
## lgl (1): .abundant
##
## ℹ Use `spec()` to retrieve the full column specification for this
## ℹ Specify the column types or set `show_col_types = FALSE` to quie
```

```
dexp
```

```
## # A tibble: 15,926 × 10
##    feature  albut transcript ref_genome .abundant   logFC logCPM
##    <chr>    <chr> <chr>      <chr>      <lgl>       <dbl>  <dbl>
##  1 ENSG000… untrt TSPAN6     hg38       TRUE      -0.390    5.06
##  2 ENSG000… untrt DPM1       hg38       TRUE       0.198    4.61
##  3 ENSG000… untrt SCYL3      hg38       TRUE       0.0292   3.48
##  4 ENSG000… untrt C1orf112   hg38       TRUE      -0.124    1.47
```

```
##  5 ENSG000… untrt CFH        hg38       TRUE       0.417    8.09 7
##  6 ENSG000… untrt FUCA2      hg38       TRUE      -0.250    5.91 1
##  7 ENSG000… untrt GCLC       hg38       TRUE      -0.0581   4.84
##  8 ENSG000… untrt NFYA       hg38       TRUE      -0.509    4.13 4
##  9 ENSG000… untrt STPG1      hg38       TRUE      -0.136    3.12
## 10 ENSG000… untrt NIPAL3     hg38       TRUE      -0.0500   7.04
## # ℹ 15,916 more rows
## # ℹ 1 more variable: FDR <dbl>
```

Joining data frames

Often related data is stored across multiple data frames. In such cases, while each data frame likely contains different types of data, an identifier column or key (e.g., sampleID) can be used to unite or combine aspects of the data.

There are a series of functions from `dplyr` devoted to the purpose of joining data frames. There are two types of joins: mutating joins *(https://dplyr.tidyverse.org/reference/mutate-joins.html)* and filtering joins *(https://dplyr.tidyverse.org/reference/filter-joins.html)*.

Mutating joins

Imagine we have two data frames `x` and `y`. A mutating join will keep all columns from `x` and `y` by adding columns from `y` to `x`.

`left_join()` - Output contains all rows from x

> return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*

`right_join()` - Output contains all rows from y

> return all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*

`inner_join()` - Output contains matched rows from x

> return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*

Unmatched values from x will be dropped.

`full_join()` - Output contains all rows from x and y

> return all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*

Note

The R documentation for dplyr has been updated with dplyr v1.0.9. However, these descriptions still stand and are clearer (in my opinion) than the new documentation.

The most common type of join is the `left_join()`. Let's see this in action:

```
#reshape acount
acount_smeta<-acount %>% pivot_longer(where(is.numeric),names_to ="Sa
                    values_to= "Count") %>% left_join(smeta, by=c("Sa
acount_smeta
```

```
## # A tibble: 512,816 × 11
##    Feature         Sample Count SampleName cell  dex   albut avgLer
##    <chr>           <chr>  <dbl> <chr>      <chr> <chr> <chr>     <d
##  1 ENSG000000000… SRR10…   679 GSM1275862 N613… untrt untrt
##  2 ENSG000000000… SRR10…   448 GSM1275863 N613… trt   untrt
##  3 ENSG000000000… SRR10…   873 GSM1275866 N052… untrt untrt
##  4 ENSG000000000… SRR10…   408 GSM1275867 N052… trt   untrt
##  5 ENSG000000000… SRR10…  1138 GSM1275870 N080… untrt untrt
##  6 ENSG000000000… SRR10…  1047 GSM1275871 N080… trt   untrt
##  7 ENSG000000000… SRR10…   770 GSM1275874 N061… untrt untrt
##  8 ENSG000000000… SRR10…   572 GSM1275875 N061… trt   untrt
##  9 ENSG000000000… SRR10…     0 GSM1275862 N613… untrt untrt
## 10 ENSG000000000… SRR10…     0 GSM1275863 N613… trt   untrt
## # ℹ 512,806 more rows
## # ℹ 2 more variables: Sample.y <chr>, BioSample <chr>
```

Notice the use of `by` in `left_join`. The argument `by` requires the column or columns that we want to join by. If the column we want to join by has a different name, we can use the notation above, which says to match `Sample` from `acount` to `Run` from `smeta`.

Filtering joins

Filtering joins result in filtered `x` data based on matching or non-matching with `y`. These joins do not add columns from `y` to `x`.

`semi_join()`

> return all rows from x where there are matching values in y, keeping just columns from x.
> A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*

`anti_join()`

> return all rows from x where there are not matching values in y, keeping just columns from x. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*

Let's see a brief example of semi-join:

```
#reshape acount
smeta_f<-smeta %>% filter(Run %in% c("SRR1039512","SRR1039508"))

acount_L<-acount %>% pivot_longer(where(is.numeric),names_to ="Sample
                     values_to= "Count")

semi_join(acount_L,smeta_f, by=c("Sample"="Run"))
```

Note

This example does not use the `%>%`. This was simply to demonstrate the different "strategies" that can be used to set up and run code.

```
## # A tibble: 128,204 × 3
##    Feature         Sample     Count
##    <chr>           <chr>      <dbl>
##  1 ENSG00000000003 SRR1039508   679
##  2 ENSG00000000003 SRR1039512   873
##  3 ENSG00000000005 SRR1039508     0
##  4 ENSG00000000005 SRR1039512     0
##  5 ENSG00000000419 SRR1039508   467
##  6 ENSG00000000419 SRR1039512   621
##  7 ENSG00000000457 SRR1039508   260
##  8 ENSG00000000457 SRR1039512   263
##  9 ENSG00000000460 SRR1039508    60
## 10 ENSG00000000460 SRR1039512    40
## # ℹ 128,194 more rows
```

In this case, we could have used filter. However, if we were interested in filtering by multiple variables simultaneously, it would be easier to employ a `semi-join`.

Transforming variables

Data wrangling often involves transforming one variable to another. For example, we may be interested in log transforming a variable or adding two variables to create a third. In `dplyr` this can be done with `mutate()` and `transmute()`. These functions allow us to create a new variable from existing variables.

`mutate()`

> mutate() adds new variables and preserves existing ones; transmute() adds new variables and drops existing ones. New variables overwrite existing variables of the same name. --- dplyr.tidyverse.org *(https://dplyr.tidyverse.org/reference/mutate.html)*

Let's create a column in our original differential expression data frame denoting significant transcripts (those with an FDR corrected p-value less than 0.05 and a log fold change greater than or equal to 2).

```
dexp_sigtrnsc<-dexp %>% mutate(Significant= FDR<0.05 & abs(logFC) >=2
head(dexp_sigtrnsc$Significant)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

This creates a column named `Significant` that contains TRUE values where the expression above was true (meaning significant in this case) and FALSE where the expression was FALSE.

Let's look at another example. This time let's log transform our FDR corrected p-values.

```
exmut<-dexp %>% mutate(logFDR = log10(FDR))
exmut["logFDR"]
```

```
## # A tibble: 15,926 × 1
##     logFDR
##      <dbl>
##  1 -2.55
##  2 -1.11
##  3 -0.0735
##  4 -0.166
```

```
##  5 -2.42
##  6 -1.73
##  7 -0.100
##  8 -2.90
##  9 -0.320
## 10 -0.158
## # ℹ 15,916 more rows
```

Mutating several variables at once

What if we want to transform all of our counts spread across multiple columns in `acount` using `scale()`, which applies a z-score transformation? In this case we use `across()` within `mutate()`, which has replaced the scoped verbs (`mutate_if`,`mutate_at`, and `mutate_all`).

Z-score

A z score tells us the number of standard deviations from the mean of a given value. This can be achieved by `scale(x, center = TRUE, scale = TRUE)`.

Let's see this in action.

```
acount %>% mutate(across(where(is.numeric),scale))
```

```
## # A tibble: 64,102 × 9
##    Feature         SRR1039508[,1] SRR1039509[,1] SRR1039512[,1] SI
##    <chr>                    <dbl>          <dbl>          <dbl>
##  1 ENSG00000000003         0.103          0.0527         0.0991
##  2 ENSG00000000005        -0.0929        -0.100         -0.0821
##  3 ENSG00000000419         0.0418         0.0756         0.0468
##  4 ENSG00000000457        -0.0179        -0.0281        -0.0275
##  5 ENSG00000000460        -0.0756        -0.0814        -0.0738
##  6 ENSG00000000938        -0.0929        -0.100         -0.0817
##  7 ENSG00000000971         0.845          1.16           1.20
##  8 ENSG00000001036         0.321          0.262          0.278
##  9 ENSG00000001084         0.0568         0.0295         0.0414
## 10 ENSG00000001167         0.0208        -0.0196         0.0142
## # ℹ 64,092 more rows
## # ℹ 4 more variables: SRR1039516 <dbl[,1]>, SRR1039517 <dbl[,1]>,
## #   SRR1039520 <dbl[,1]>, SRR1039521 <dbl[,1]>
```

For further information on `across` *(https://dplyr.tidyverse.org/articles/colwise.html)*, check out this great tutorial here *(https://www.rebeccabarter.com/blog/2020-07-09-across/)*.

Coercing variables with `mutate`

Mutate can also be used to coerce variables. Again, we need to use `across()` and `where()`. Let's also check out the difference between `mutate()` and `transmute()`.

```
#compare transmute to mutate
ex_coerce<-acount_smeta %>% transmute(across(where(is.character),as.f
glimpse(ex_coerce)
```

```
## Rows: 512,816
## Columns: 9
## $ Feature    <fct> ENSG00000000003, ENSG00000000003, ENSG00000000(
## $ Sample     <fct> SRR1039508, SRR1039509, SRR1039512, SRR1039513
## $ SampleName <fct> GSM1275862, GSM1275863, GSM1275866, GSM1275867
## $ cell       <fct> N61311, N61311, N052611, N052611, N080611, N08(
## $ dex        <fct> untrt, trt, untrt, trt, untrt, trt, untrt, trt
## $ albut      <fct> untrt, untrt, untrt, untrt, untrt, untrt, untrt
## $ Experiment <fct> SRX384345, SRX384346, SRX384349, SRX384350, SRX
## $ Sample.y   <fct> SRS508568, SRS508567, SRS508571, SRS508572, SRS
## $ BioSample  <fct> SAMN02422669, SAMN02422675, SAMN02422678, SAMN(
```

```
#mutate
ex_coerce<-acount_smeta %>% mutate(across(where(is.character),as.fact
glimpse(ex_coerce)
```

```
## Rows: 512,816
## Columns: 11
## $ Feature    <fct> ENSG00000000003, ENSG00000000003, ENSG00000000(
## $ Sample     <fct> SRR1039508, SRR1039509, SRR1039512, SRR1039513
## $ Count      <dbl> 679, 448, 873, 408, 1138, 1047, 770, 572, 0, 0
## $ SampleName <fct> GSM1275862, GSM1275863, GSM1275866, GSM1275867
## $ cell       <fct> N61311, N61311, N052611, N052611, N080611, N08(
## $ dex        <fct> untrt, trt, untrt, trt, untrt, trt, untrt, trt
## $ albut      <fct> untrt, untrt, untrt, untrt, untrt, untrt, untrt
## $ avgLength  <dbl> 126, 126, 126, 87, 120, 126, 101, 98, 126, 126
## $ Experiment <fct> SRX384345, SRX384346, SRX384349, SRX384350, SRX
## $ Sample.y   <fct> SRS508568, SRS508567, SRS508571, SRS508572, SRS
## $ BioSample  <fct> SAMN02422669, SAMN02422675, SAMN02422678, SAMN(
```

Notice that `transmute` dropped all columns that were not mutated.

Using `rowwise()` and `mutate()`

What if we wanted a new column that stored the mean of each row in our data frame?

Let's create a small data frame, and use `mutate()` to get the `mean()`. What happens when we use `mean` as is?

```
df<-data.frame(A=c(1,2,3),B=c(4,5,6),C=c(7,8,9))
df
```

```
##   A B C
## 1 1 4 7
## 2 2 5 8
## 3 3 6 9
```

```
df %>% mutate(D= mean(c(A,B,C)))
```

```
##   A B C D
## 1 1 4 7 5
## 2 2 5 8 5
## 3 3 6 9 5
```

```
df %>% mutate(D = (A+B+C)/3)
```

```
##   A B C D
## 1 1 4 7 4
## 2 2 5 8 5
## 3 3 6 9 6
```

The first example simply gives us the mean of A, B, and C (not row wise). The second example gave us what we wanted, but was more complicated. The alternative is to first group by row using `rowwise()` and then use `mutate()`.

```
df %>% rowwise() %>% mutate(D= mean(c(A,B,C)))
```

```
## # A tibble: 3 × 4
## # Rowwise:
##       A     B     C     D
```

```
##   <dbl> <dbl> <dbl> <dbl>
## 1     1     4     7     4
## 2     2     5     8     5
## 3     3     6     9     6
```

See more uses of `rowwise()` operations here *(https://dplyr.tidyverse.org/articles/rowwise.html)*.

Group_by and summarize

There is an approach to data analysis known as "split-apply-combine", in which the data is split into smaller components, some type of analysis is applied to each component, and the results are combined. The `dplyr` functions including `group_by()` and `summarize()` are key players in this type of workflow.

`group_by()` allows us to group a data frame by a categorical variable so that a given operation can be performed per group / category.

Let's get the top five genes with the greatest median raw counts by treatment.

```
#Call the data
acount_smeta %>%
  # group_by dex and Feature (Feature nested within treatment)
  group_by(dex,Feature) %>%
  #for each group calculate the median value of raw counts
  summarize(median_counts=median(Count)) %>%
  #arrange in descending order
  arrange(desc(median_counts),.by_group = TRUE) %>%
  #return the top 5 values for each group
  slice_head(n=5)
```

```
## `summarise()` has grouped output by 'dex'. You can override using
## argument.
```

```
## # A tibble: 10 × 3
## # Groups:   dex [2]
##    dex   Feature         median_counts
##    <chr> <chr>                   <dbl>
##  1 trt   ENSG00000115414       322164
##  2 trt   ENSG00000011465       263587
##  3 trt   ENSG00000156508       239676.
##  4 trt   ENSG00000198804       230992
##  5 trt   ENSG00000116260       187288.
##  6 untrt ENSG00000011465       336076
```

```
##  7 untrt ENSG00000115414       302956.
##  8 untrt ENSG00000156508       294097
##  9 untrt ENSG00000164692       249846
## 10 untrt ENSG00000198804       249206
```

```
#can skip arrange and use slice_max
acount_smeta %>%
  group_by(dex,Feature) %>%
  summarize(median_counts=median(Count))  %>%
  slice_max(n=5, order_by=median_counts) #notice use of slice_max
```

```
## `summarise()` has grouped output by 'dex'. You can override using
## argument.
```

```
## # A tibble: 10 × 3
## # Groups:   dex [2]
##    dex   Feature         median_counts
##    <chr> <chr>                   <dbl>
##  1 trt   ENSG00000115414       322164
##  2 trt   ENSG00000011465       263587
##  3 trt   ENSG00000156508       239676.
##  4 trt   ENSG00000198804       230992
##  5 trt   ENSG00000116260       187288.
##  6 untrt ENSG00000011465       336076
##  7 untrt ENSG00000115414       302956.
##  8 untrt ENSG00000156508       294097
##  9 untrt ENSG00000164692       249846
## 10 untrt ENSG00000198804       249206
```

How many rows per sample are in the acount_smeta data frame? We can use `count()` or `summarize()` paired with other functions (e.g., `n()`,`tally()`).

```
acount_smeta  %>%
  count(dex, Sample)
```

```
## # A tibble: 8 × 3
##   dex   Sample         n
##   <chr> <chr>      <int>
## 1 trt   SRR1039509 64102
## 2 trt   SRR1039513 64102
## 3 trt   SRR1039517 64102
```

```
## 4 trt   SRR1039521 64102
## 5 untrt SRR1039508 64102
## 6 untrt SRR1039512 64102
## 7 untrt SRR1039516 64102
## 8 untrt SRR1039520 64102
```

```
acount_smeta  %>%
  group_by(dex, Sample) %>%
  summarize(n=n()) #there are multiple functions that can be used her
```

```
## `summarise()` has grouped output by 'dex'. You can override using
## argument.
```

```
## # A tibble: 8 × 3
## # Groups:   dex [2]
##   dex   Sample         n
##   <chr> <chr>      <int>
## 1 trt   SRR1039509 64102
## 2 trt   SRR1039513 64102
## 3 trt   SRR1039517 64102
## 4 trt   SRR1039521 64102
## 5 untrt SRR1039508 64102
## 6 untrt SRR1039512 64102
## 7 untrt SRR1039516 64102
## 8 untrt SRR1039520 64102
```

This output makes sense, as there are 64,102 unique Ensembl ids (`n_distinct(acount_smeta$Feature)`)

Missing Values

> By default, all [built in] R functions operating on vectors that contain missing data will return NA. It's a way to make sure that users know they have missing data, and make a conscious decision on how to deal with it. When dealing with simple statistics like the mean, the easiest way to ignore NA (the missing data) is to use na.rm = TRUE (rm stands for remove). ---datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/05-dplyr/index.html)*

Let's see this in practice

```
#This is used to get the same result
#with a pseudorandom number generator like sample()
set.seed(138)
```

```
#make mock data frame
fun_df<-data.frame(genes=rep(c("A","B","C","D"), each=3),
                   counts=sample(1:500,12,TRUE)) %>%
  #Assign NAs if the value is less than 100. This is arbitrary.
  mutate(counts=replace(counts, counts<100, NA))

#let's view
fun_df
```

```
##    genes counts
## 1      A     NA
## 2      A    214
## 3      A     NA
## 4      B    352
## 5      B    256
## 6      B     NA
## 7      C    400
## 8      C    381
## 9      C    250
## 10     D    278
## 11     D     NA
## 12     D    169
```

```
#Summarize mean, median, min, and max
fun_df %>%
  group_by(genes) %>%
  summarize(
    mean_count = mean(counts),
    median_count = median(counts),
    min_count = min(counts),
    max_count = max(counts))
```

```
## # A tibble: 4 × 5
##   genes mean_count median_count min_count max_count
##   <chr>      <dbl>        <int>     <int>     <int>
## 1 A             NA           NA        NA        NA
## 2 B             NA           NA        NA        NA
## 3 C            344.         381       250       400
## 4 D             NA           NA        NA        NA
```

```
#use na.rm
```

```
fun_df %>%
  group_by(genes) %>%
  summarize(
    mean_count = mean(counts, na.rm=TRUE),
    median_count = median(counts, na.rm=TRUE),
    min_count = min(counts, na.rm=TRUE),
    max_count = max(counts, na.rm=TRUE))
```

```
## # A tibble: 4 × 5
##   genes mean_count median_count min_count max_count
##   <chr>      <dbl>        <dbl>     <int>     <int>
## 1 A            214          214       214       214
## 2 B            304          304       256       352
## 3 C            344.         381       250       400
## 4 D            224.         224.      169       278
```

Similar to mutate, we can summarize across multiple columns at once using `across()`. For example, let's get the mean of `logFC` and `logCPM`.

```
dexp %>%
  summarize(across(starts_with("Log"), mean))
```

```
## # A tibble: 1 × 2
##      logFC logCPM
##      <dbl>  <dbl>
## 1 -0.00859   3.71
```

Exporting files using the `write` functions

We have learned how to import data using the read functions, but how can we export / write out data? We can use a series of write functions. Some examples from `readr` include `write_csv()`, `write_delim()`, `write_tsv()`. Some examples from base R include `write.csv()`, `write.table()`, `writeLines()`.

Let's export a tab delimited file containing `acount_smeta`.

```
write_tsv(acount_smeta, "countsANDmeta.txt", quote="none")
```

Acknowledgments

Some material from this lesson was either taken directly or adapted from the Intro to R and RStudio for Genomics lesson provided by datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html)*. Additional content was inspired by Chapter 13, Relational Data, *(https://r4ds.had.co.nz/relational-data.html)* from *R for Data Science* and Suzan Baert's dplyr tutorials *(https://suzan.rbind.io/categories/tutorial/)*.

Lesson 7: Introduction to Bioconductor -omics classes (containers)

Objectives

1. To explore Bioconductor, a repository for R packages related to biological data analysis.
2. To better understand S4 objects as they relate to the Bioconductor core infrastructure.
3. To learn more about a popular Bioconductor S4 class, `SummarizedExperiment`.

What is Bioconductor?

Bioconductor is a repository for R packages related to biological data analysis, primarily bioinformatics and computational biology, and as such it is a great place to search for -omics packages and pipelines.

Bioconductor packages fit within four main categories:

- software
- annotation data
- experiment data
- workflows.

As of November 2023, Bioconductor v3.18 included 2,266 software packages, 429 experiment data packages, 920 annotation packages, 30 workflows, and 4 books.
To browse these packages, use BiocViews *(https://www.bioconductor.org/packages/release/BiocViews.html)*.

Important things to know about Bioconductor

- Bioconductor is a package repository, like CRAN
- All Bioconductor packages must be installed following the instructions here: https://bioconductor.org/install
- Bioconductor packages are linked in their versions, both to each other and to the version of R
- Bioconductor's installation function will look up your version of R and give you the appropriate versions of Bioconductor packages
- If you want the latest version of Bioconductor, you need to use the latest version of R
  --- Michael Love *(https://biodatascience.github.io/compbio/bioc/objects.html)*

Core infrastructure

An important goal of the Bioconductor project is interoperability, or the ability of packages to work together using shared data classes and methods. This is achieved through the use of common data structures. These common data structures are "implemented as classes in the S4 object-oriented system of the R language" *(https://www.nature.com/articles/nmeth.3252)*.

What is object oriented programming?

R is generally thought of as a functional programming language, where the focus is on the functions rather than the object, and the output of the function is always the same given the same inputs. This is a particular useful way of approaching problems, especially when the focus is on data analysis. This differs from an object oriented language, where the functions are built around an object (e.g., a data structure), and the output of a function can change based on the attributes of the object. This can also be a useful way to analyze data. As you may have guessed, these programming paradigms are not mutually exclusive and both are used with R programming.

Comparing functional programming vs object oriented programming

For more information comparing these paradigms, check out this brief article *(https://medium.com/@shaistha24/functional-programming-vs-object-oriented-programming-oop-which-is-better-82172e53a526AZ)* from *Medium*.

Object-oriented programming allows the same function to be used for many different types of input (e.g., `summary()`) (Advanced R).

Note

We work with S3 objects *(https://adv-r.hadley.nz/s3.html)* all the time in R and these are prominent when using base R programming.

Terms to know for object oriented programming with R

- class - object type
- method - implementation for a class; describes what an object can do
- method dispatch - finds the correct method for a class

> Classes are organised in a hierarchy so that if a method does not exist for one class, its parent's method is used, and the child is said to inherit behaviour. For example, in R, an ordered factor inherits from a regular factor, and a generalised linear model inherits from a linear model. --- Advanced R *(https://adv-r.hadley.nz/oo.html)*

S4 objects and Bioconductor

-omics data can be fairly complex, and data structures are a useful way of organizing and working with complex data. Many data structures are used or extended across multiple Bioconductor packages, thereby allowing users to approach and use new packages with less of a learning hurdle.

Understanding S4 classes

This lesson will not go into too much detail regarding the S4 class system. You do not need to know that much about S4 systems to use S4 objects. However, some of the basics will be described below. If you would like to learn more about S4 classes, see this lesson *(https://carpentries-incubator.github.io/bioc-project/05-s4.html)* with *The Carpentries* and this chapter *(https://adv-r.hadley.nz/s4.html)* of *Advanced R*.

What is the S4 system?

> S4 is a rigorous system that forces you to think carefully about program design. It's particularly well-suited for building large systems that evolve over time and will receive contributions from many programmers. *(https://adv-r.hadley.nz/oo.html)* --- *Advanced R*

S4 classes create the data structures that store complex information (e.g., biological assay data and metadata) in one or more slots. The entire structure can then be assigned to an R object. The types of information in each slot of the object is tightly controlled. S4 generics and methods define functions that can be applied to these objects. See this lesson *(https://carpentries-incubator.github.io/bioc-project/05-s4.html#s4-classes-and-methods)* from The Carpentries, authored by the Bioconductor Project community, for more information.

In brief, S4 objects have slots that store defined types of information. The use of the object can be extended by defining a new class, which will inherit the attributes of the old class, including all of the slots, and add new slots.

Info

You can find a list of common classes used by the Bioconductor Project here *(https://contributions.bioconductor.org/important-bioconductor-package-development-features.html?q=classes#reusebioc)*.

Let's take a look at the `SummarizedExperiment`, a commonly used class in many Bioconductor packages.

Introducing the *SummarizedExperiment*

The `SummarizedExperiment` class and the inherited class `RangedSummarizedExperiment` are available in the R package `SummarizedExperiment`.

SummarizedExperiment is a matrix-like container where rows represent features of interest (e.g. genes, transcripts, exons, etc.) and columns represent samples. The objects contain one or more assays, each represented by a matrix-like object of numeric or other mode. --- SummarizedExperiment vignette *(https://bioconductor.org/packages/devel/bioc/vignettes/SummarizedExperiment/inst/doc/SummarizedExperiment.html)*

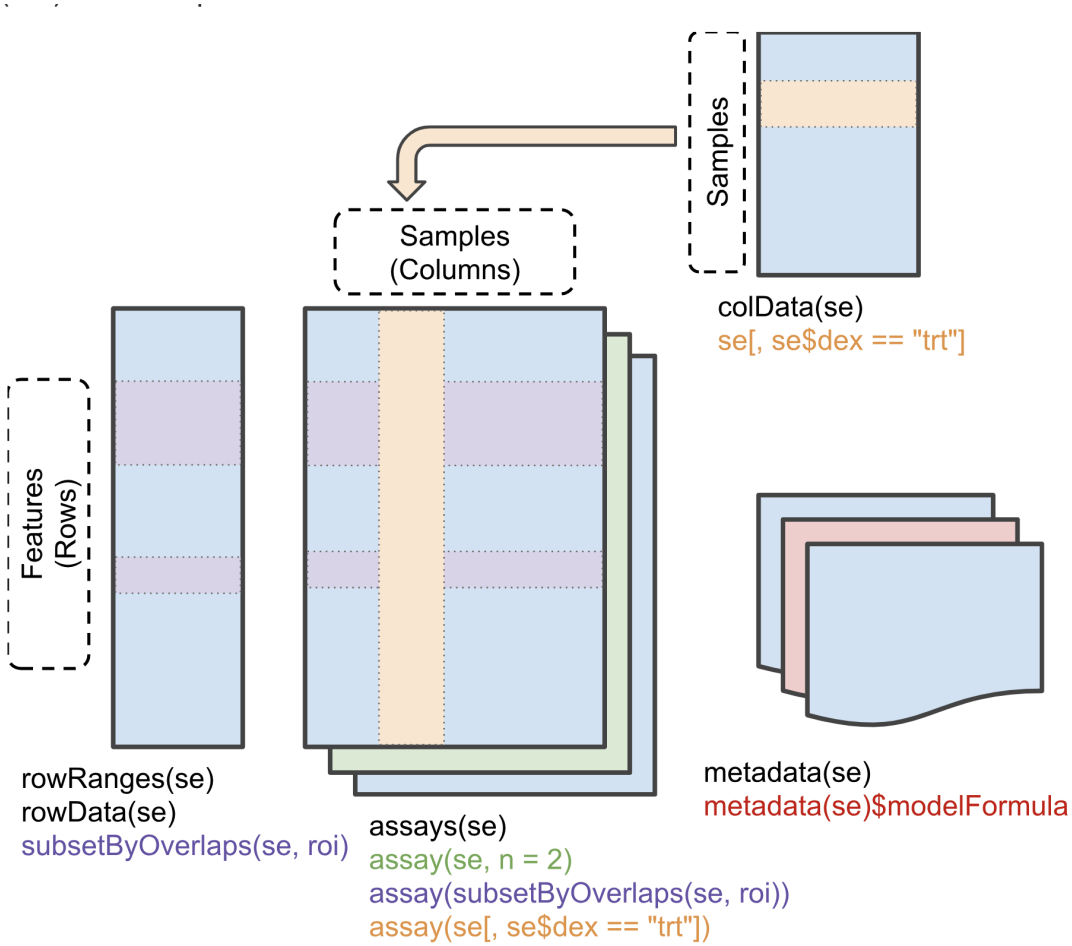


Image from the SummarizedExperiment vignette.

The advantage of using a `SummarizedExperiment` is that when manipulating the object, the data elements in each slot maintain consistency, meaning if a sample is removed, the corresponding sample column in the expression data would also be removed. This keeps you from accidentally including data that should have been excluded or vice versa.

Working with a summarized experiment.

We can use the `airway` package to see how this container works, including how to access and subset the data.

What is the `airway` package?

There are data sets available in R to practice with or showcase different packages. The airway data is from Himes et al. (2014) *(https://pubmed.ncbi.nlm.nih.gov/24926665/)* . These data, which are contained within a `RangedSummarizedExperiment` object, are from a bulk RNAseq experiment. In the experiment, the authors "characterized transcriptomic changes in four primary human ASM cell lines that were treated with dexamethasone," a common therapy for asthma. The airway package includes RNAseq count data from 8 airway smooth muscle cell samples.

The `airway` data is packaged in a `RangedSummarizedExperiment`. A `RangedSummarizedExperiment` inherits the features of a `SummarizedExperiment` but instead of only including feature metadata for the rows, it also includes genomic position information.

Note

Other popular classes that inherit from the `SummarizedExperiment` include the `DESeqDataSet` *(https://bioconductor.org/packages/devel/bioc/vignettes/DESeq2/inst/doc/DESeq2.html#the-deseqdataset)* and the `SingleCellExperiment` *(https://bioconductor.org/packages/release/bioc/vignettes/SingleCellExperiment/inst/doc/intro.html)*.

```
library(SummarizedExperiment)
library(airway)
data(airway, package="airway")
se <- airway
se
```

```
## class: RangedSummarizedExperiment
## dim: 63677 8
## metadata(1): ''
## assays(1): counts
## rownames(63677): ENSG00000000003 ENSG00000000005 ... ENSG00000273
##   ENSG00000273493
## rowData names(10): gene_id gene_name ... seq_coord_system symbol
## colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
## colData names(9): SampleName cell ... Sample BioSample
```

Accessors

As you can see from the image, there are several accessor functions to access the data from the object:

- `assays()` - access matrix-like experimental data (e.g., count data). Rows are genomic features (e.g., transcripts) and columns are samples.

  This can hold more than one assay, and each assay should be called directly with the `$` accessor.

```
assays(se) #we see one assay listed
```

```
## List of length 1
## names(1): counts
```

```
head(assays(se)$counts) #calling the counts with $
```

```
##                 SRR1039508 SRR1039509 SRR1039512 SRR1039513 S
## ENSG00000000003        679        448        873        408
## ENSG00000000005          0          0          0          0
## ENSG00000000419        467        515        621        365
## ENSG00000000457        260        211        263        164
## ENSG00000000460         60         55         40         35
## ENSG00000000938          0          0          2          0
##                 SRR1039517 SRR1039520 SRR1039521
## ENSG00000000003       1047        770        572
## ENSG00000000005          0          0          0
## ENSG00000000419        799        417        508
## ENSG00000000457        331        233        229
## ENSG00000000460         63         76         60
## ENSG00000000938          0          0          0
```

- `colData()` - access sample metadata, as a `DataFrame`

```
colData(se)
```

```
## DataFrame with 8 rows and 9 columns
##             SampleName     cell      dex    albut        Run a
##               <factor> <factor> <factor> <factor>   <factor> <
## SRR1039508 GSM1275862  N61311     untrt    untrt SRR1039508
## SRR1039509 GSM1275863  N61311     trt      untrt SRR1039509
## SRR1039512 GSM1275866  N052611    untrt    untrt SRR1039512
## SRR1039513 GSM1275867  N052611    trt      untrt SRR1039513
## SRR1039516 GSM1275870  N080611    untrt    untrt SRR1039516
## SRR1039517 GSM1275871  N080611    trt      untrt SRR1039517
## SRR1039520 GSM1275874  N061011    untrt    untrt SRR1039520
## SRR1039521 GSM1275875  N061011    trt      untrt SRR1039521
##             Experiment    Sample    BioSample
##               <factor>  <factor>     <factor>
## SRR1039508  SRX384345 SRS508568 SAMN02422669
```

```
## SRR1039509  SRX384346 SRS508567 SAMN02422675
## SRR1039512  SRX384349 SRS508571 SAMN02422678
## SRR1039513  SRX384350 SRS508572 SAMN02422670
## SRR1039516  SRX384353 SRS508575 SAMN02422682
## SRR1039517  SRX384354 SRS508576 SAMN02422673
## SRR1039520  SRX384357 SRS508579 SAMN02422683
## SRR1039521  SRX384358 SRS508580 SAMN02422677
```

- `rowData()` - access feature metadata (e.g., differential expression results)

```
rowData(se)
```

```
## DataFrame with 63677 rows and 10 columns
##                         gene_id     gene_name  entrezid   gen
##                     <character>   <character> <integer>    <c
## ENSG00000000003 ENSG00000000003        TSPAN6        NA prote
## ENSG00000000005 ENSG00000000005          TNMD        NA prote
## ENSG00000000419 ENSG00000000419          DPM1        NA prote
## ENSG00000000457 ENSG00000000457         SCYL3        NA prote
## ENSG00000000460 ENSG00000000460      C1orf112        NA prote
## ...                        ...           ...       ...
## ENSG00000273489 ENSG00000273489 RP11-180C16.1        NA
## ENSG00000273490 ENSG00000273490        TSEN34        NA prote
## ENSG00000273491 ENSG00000273491  RP11-138A9.2        NA
## ENSG00000273492 ENSG00000273492    AP000230.1        NA
## ENSG00000273493 ENSG00000273493  RP11-80H18.4        NA
##                 gene_seq_start gene_seq_end             seq_
##                      <integer>    <integer>           <charac
## ENSG00000000003       99883667     99894988
## ENSG00000000005       99839799     99854882
## ENSG00000000419       49551404     49575092
## ENSG00000000457      169818772    169863408
## ENSG00000000460      169631245    169823221
## ...                        ...          ...
## ENSG00000273489      131178723    131182453
## ENSG00000273490       54693789     54697585 HSCHR19LRC_LRC_J_
## ENSG00000273491      130600118    130603315          HG1308_P
## ENSG00000273492       27543189     27589700
## ENSG00000273493       58315692     58315845
##                 seq_coord_system        symbol
##                        <integer>   <character>
## ENSG00000000003               NA        TSPAN6
## ENSG00000000005               NA          TNMD
## ENSG00000000419               NA          DPM1
## ENSG00000000457               NA         SCYL3
```

```
## ENSG00000000460              NA      C1orf112
## ...                         ...           ...
## ENSG00000273489              NA RP11-180C16.1
## ENSG00000273490              NA        TSEN34
## ENSG00000273491              NA  RP11-138A9.2
## ENSG00000273492              NA    AP000230.1
## ENSG00000273493              NA  RP11-80H18.4
```

- `rowRanges(se)` - because this is a `RangedSummarizedExperiment`, there is more information about the genomic ranges spanning the exons of each transcript

```
rowRanges(se)$ENSG00000000003
```

```
## GRanges object with 17 ranges and 2 metadata columns:
##        seqnames            ranges strand |   exon_id
##           <Rle>         <IRanges>  <Rle> | <integer>
##    [1]        X 99883667-99884983      - |    667145 ENSI
##    [2]        X 99885756-99885863      - |    667146 ENSI
##    [3]        X 99887482-99887565      - |    667147 ENSI
##    [4]        X 99887538-99887565      - |    667148 ENSI
##    [5]        X 99888402-99888536      - |    667149 ENSI
##    ...      ...                 ...    ... .       ...
##   [13]        X 99890555-99890743      - |    667156 ENSI
##   [14]        X 99891188-99891686      - |    667158 ENSI
##   [15]        X 99891605-99891803      - |    667159 ENSI
##   [16]        X 99891790-99892101      - |    667160 ENSI
##   [17]        X 99894942-99894988      - |    667161 ENSI
##   -------
##   seqinfo: 722 sequences (1 circular) from an unspecifie
```

- `metadata()` - access experiment wide unstructured metadata (e.g., experimental methods, publication references)

```
metadata(se)
```

```
## [[1]]
## Experiment data
##   Experimenter name: Himes BE
##   Laboratory: NA
##   Contact information:
##   Title: RNA-Seq transcriptome profiling identifies CRISPLD2
##   URL: http://www.ncbi.nlm.nih.gov/pubmed/24926665
```

```
##   PMIDs: 24926665
##
##   Abstract: A 226 word abstract is available. Use 'abstract'
```

```

> **Note**
>
> You can use `str()` to get an idea of how to access information from each slot. Notice the use of `@`.

Now that we know how to access the information stored in the object `se`, how can we subset it?

#### Subsetting and manipulating `SummarizedExperiments`

First, notice that you can easily access columns from the sample metadata (`colData()`) using `$`.

Using brackets to subset:

```
se$SampleName
```

```
## [1] GSM1275862 GSM1275863 GSM1275866 GSM1275867 GSM1275870 GSM127
## [8] GSM1275875
## 8 Levels: GSM1275862 GSM1275863 GSM1275866 GSM1275867 ... GSM1275
```

```
se$dex
```

```
## [1] untrt trt   untrt trt   untrt trt   untrt trt
## Levels: trt untrt
```

```
levels(se$dex)
```

```
## [1] "trt"   "untrt"
```

We can also use 2D subsetting like we would for a data frame (i.e., `[]` notation), where left of the comma applies to the rows (features) and right of the comma applies to the columns (samples).

If we only wanted the first 10 transcripts and the first three samples, we could use
```

```
se[1:10,1:3]
```

```
## class: RangedSummarizedExperiment
## dim: 10 3
## metadata(1): ''
## assays(1): counts
## rownames(10): ENSG00000000003 ENSG00000000005 ... ENSG00000001084
##   ENSG00000001167
## rowData names(10): gene_id gene_name ... seq_coord_system symbol
## colnames(3): SRR1039508 SRR1039509 SRR1039512
## colData names(9): SampleName cell ... Sample BioSample
```

We can also apply conditions using `[]`.

For example, if we only want to include treated samples.

```
se[ ,se$dex=="trt"] #notice the dimensions of the data have changed
```

```
## class: RangedSummarizedExperiment
## dim: 63677 4
## metadata(1): ''
## assays(1): counts
## rownames(63677): ENSG00000000003 ENSG00000000005 ... ENSG00000273
##   ENSG00000273493
## rowData names(10): gene_id gene_name ... seq_coord_system symbol
## colnames(4): SRR1039509 SRR1039513 SRR1039517 SRR1039521
## colData names(9): SampleName cell ... Sample BioSample
```

```
se[ ,se$dex=="trt"]$dex #we can check to see that only trt samples re
```

```
## [1] trt trt trt trt
## Levels: trt untrt
```

We can do the same with feature information. For example, if we only want to include `protein_coding` transcripts:

```
se[rowData(se)$gene_biotype == "protein_coding",]
```

```
## class: RangedSummarizedExperiment
## dim: 22810 8
## metadata(1): ''
## assays(1): counts
## rownames(22810): ENSG00000000003 ENSG00000000005 ... ENSG00000273
##   ENSG00000273490
## rowData names(10): gene_id gene_name ... seq_coord_system symbol
## colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
## colData names(9): SampleName cell ... Sample BioSample
```

OR

We could filter using the count assay. For example, keeping only transcripts that have a count of at least 10 across a minimum of 4 samples.

```
transcripts_to_keep <- rowSums(assays(se)$counts >= 10) >= 4
se[transcripts_to_keep,]
```

```
## class: RangedSummarizedExperiment
## dim: 16139 8
## metadata(1): ''
## assays(1): counts
## rownames(16139): ENSG00000000003 ENSG00000000419 ... ENSG00000273
##   ENSG00000273487
## rowData names(10): gene_id gene_name ... seq_coord_system symbol
## colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
## colData names(9): SampleName cell ... Sample BioSample
```

Check out the `SummarizedExperiment` vignette *(https://bioconductor.org/packages/devel/bioc/vignettes/SummarizedExperiment/inst/doc/SummarizedExperiment.html)* for more information!

Using `tidySummarizedExperiment`

Why did we focus so heavily on the tidyverse if it can't be used to manipulate Bioconductor objects? Well, for one, regardless of whether you are a user of Bioconductor packages, you will often manipulate data frames, which is where the tidyverse collection of packages is super useful. Second, you can actually use tidyverse commands in certain contexts. The R package, `tidySummarizedExperiment` allows you to view a `SummarizedExperiment` object as a tibble (a tidyverse data frame) and provides other tidyverse compatible functions from the packages `dplyr`, `tidyr`, `ggplot`, and `plotly`. *(https://stemangiola.github.io/tidySummarizedExperiment/)*

```
library(tidySummarizedExperiment)

# filter to samples in the untreated condition
se %>% filter(dex == "untrt")
```

```
## # A SummarizedExperiment-tibble abstraction: 254,708 × 23
## # [90mFeatures=63677 | Samples=4 | Assays=counts[0m
##    .feature        .sample   counts SampleName cell  dex   albut
##    <chr>           <chr>      <int> <fct>      <fct> <fct> <fct>
##  1 ENSG00000000003 SRR10395…    679 GSM1275862 N613… untrt untrt
##  2 ENSG00000000005 SRR10395…      0 GSM1275862 N613… untrt untrt
##  3 ENSG00000000419 SRR10395…    467 GSM1275862 N613… untrt untrt
##  4 ENSG00000000457 SRR10395…    260 GSM1275862 N613… untrt untrt
##  5 ENSG00000000460 SRR10395…     60 GSM1275862 N613… untrt untrt
##  6 ENSG00000000938 SRR10395…      0 GSM1275862 N613… untrt untrt
##  7 ENSG00000000971 SRR10395…   3251 GSM1275862 N613… untrt untrt
##  8 ENSG00000001036 SRR10395…   1433 GSM1275862 N613… untrt untrt
##  9 ENSG00000001084 SRR10395…    519 GSM1275862 N613… untrt untrt
## 10 ENSG00000001167 SRR10395…    394 GSM1275862 N613… untrt untrt
## # ℹ 40 more rows
## # ℹ 14 more variables: Experiment <fct>, Sample <fct>, BioSample <
## #   gene_id <chr>, gene_name <chr>, entrezid <int>, gene_biotype <
## #   gene_seq_start <int>, gene_seq_end <int>, seq_name <chr>, seq_
## #   seq_coord_system <int>, symbol <chr>, GRangesList <list>
```

```
# total over all genes, per condition
se %>%
  group_by(dex) %>%
  summarize(total=sum(counts))
```

```
## # A tibble: 2 × 2
##   dex      total
##   <fct>    <int>
## 1 trt   85955244
## 2 untrt 89561179
```

```
# the features where mean expression > 0
se %>%
  group_by(.feature) %>%
  mutate(mean_exprs = mean(counts)) %>%
  filter(mean_exprs > 0)
```

```
## # A tibble: 267,752 × 24
## # Groups:   .feature [33,469]
##    .feature        .sample   counts SampleName cell  dex   albut F
##    <chr>           <chr>      <int> <fct>      <fct> <fct> <fct> <
##  1 ENSG00000000003 SRR10395…    679 GSM1275862 N613… untrt untrt S
##  2 ENSG00000000419 SRR10395…    467 GSM1275862 N613… untrt untrt S
##  3 ENSG00000000457 SRR10395…    260 GSM1275862 N613… untrt untrt S
##  4 ENSG00000000460 SRR10395…     60 GSM1275862 N613… untrt untrt S
##  5 ENSG00000000938 SRR10395…      0 GSM1275862 N613… untrt untrt S
##  6 ENSG00000000971 SRR10395…   3251 GSM1275862 N613… untrt untrt S
##  7 ENSG00000001036 SRR10395…   1433 GSM1275862 N613… untrt untrt S
##  8 ENSG00000001084 SRR10395…    519 GSM1275862 N613… untrt untrt S
##  9 ENSG00000001167 SRR10395…    394 GSM1275862 N613… untrt untrt S
## 10 ENSG00000001460 SRR10395…    172 GSM1275862 N613… untrt untrt S
## # ℹ 267,742 more rows
## # ℹ 15 more variables: Experiment <fct>, Sample <fct>, BioSample <
## #   gene_id <chr>, gene_name <chr>, entrezid <int>, gene_biotype <
## #   gene_seq_start <int>, gene_seq_end <int>, seq_name <chr>, seq_
## #   seq_coord_system <int>, symbol <chr>, GRangesList <list>, mean
```

Other packages uniting bioinformatics and the tidyverse

The following is not a comprehensive list:

1. `tidySingleCellExperiment` *(https://stemangiola.github.io/tidySingleCellExperiment/)* - for `SingleCellExperiment` *(https://bioconductor.org/packages/release/bioc/vignettes/SingleCellExperiment/inst/doc/intro.html)* objects
2. `tidyseurat` *(https://stemangiola.github.io/tidyseurat/)* - for `Seurat` objects

> **Note**
>
> `Seurat` is not a Bioconductor package. It is a CRAN package.`tidyseurat` is also a CRAN package.

3. `tidybulk` *(https://stemangiola.github.io/tidybulk/)* - for analyzing RNA-seq data
4. `tidyHeatmap` *(https://stemangiola.github.io/tidyHeatmap/)* - make heatmaps from tidy data - a CRAN package

Saving an R object

S4 objects store complex information that isn't necessarily simple to save. If you intend to work with the object further, try using `saveRDS`.

```
saveRDS(se, "airways.rds") #save the object
```

Note

`saveRDS()` is used to save a single object. To save multiple objects, use `save()`. To save your entire R environment, use `save.image()`.

To load the object, back to your R environment, use `readRDS()`.

```
rm(se) # remove the object from the environment
readRDS("./airways.rds") #load the object from file
```

```
## # A SummarizedExperiment-tibble abstraction: 509,416 × 23
## # ⓲[90mFeatures=63677 | Samples=8 | Assays=counts⓳[0m
##    .feature        .sample   counts SampleName cell  dex   albut F
##    <chr>           <chr>      <int> <fct>      <fct> <fct> <fct> <
##  1 ENSG00000000003 SRR10395…    679 GSM1275862 N613… untrt untrt S
##  2 ENSG00000000005 SRR10395…      0 GSM1275862 N613… untrt untrt S
##  3 ENSG00000000419 SRR10395…    467 GSM1275862 N613… untrt untrt S
##  4 ENSG00000000457 SRR10395…    260 GSM1275862 N613… untrt untrt S
##  5 ENSG00000000460 SRR10395…     60 GSM1275862 N613… untrt untrt S
##  6 ENSG00000000938 SRR10395…      0 GSM1275862 N613… untrt untrt S
##  7 ENSG00000000971 SRR10395…   3251 GSM1275862 N613… untrt untrt S
##  8 ENSG00000001036 SRR10395…   1433 GSM1275862 N613… untrt untrt S
##  9 ENSG00000001084 SRR10395…    519 GSM1275862 N613… untrt untrt S
## 10 ENSG00000001167 SRR10395…    394 GSM1275862 N613… untrt untrt S
## # ℹ 40 more rows
## # ℹ 14 more variables: Experiment <fct>, Sample <fct>, BioSample <
## #   gene_id <chr>, gene_name <chr>, entrezid <int>, gene_biotype <
## #   gene_seq_start <int>, gene_seq_end <int>, seq_name <chr>, seq_
## #   seq_coord_system <int>, symbol <chr>, GRangesList <list>
```

Acknowledgements

Content from this lesson was inspired by or taken from the following sources:

1. https://bioconductor.org/packages/devel/bioc/vignettes/SummarizedExperiment/inst/doc/SummarizedExperiment.html *(https://bioconductor.org/packages/devel/bioc/vignettes/SummarizedExperiment/inst/doc/SummarizedExperiment.html)*
2. https://carpentries-incubator.github.io/bioc-project/05-s4.html#s4-classes-and-methods *(https://carpentries-incubator.github.io/bioc-project/05-s4.html#s4-classes-and-methods)*

3. https://carpentries-incubator.github.io/bioc-intro/60-next-steps.html *(https://carpentries-incubator.github.io/bioc-intro/60-next-steps.html)*
4. https://biodatascience.github.io/compbio/bioc/objects.html *(https://biodatascience.github.io/compbio/bioc/objects.html)*

Lesson 8: Data Wrangling Review and Practice

Objectives

1. Review important data wrangling functions
2. Put our wrangling skills to use on a realistic RNA-Seq data set

Data Wrangling Review

Important functions by topic

Importing / Exporting Data

Importing and exporting data into the R environment is done using base R and `readR` (`readxl` in the case of excel files) functions. Most of these functions begin with `read.` and `read_` for importing, or `write.` and `write_` for exporting. You can use tab complete to find the appropriate function.

Data reshape (`library(tidyr)` *(https://tidyr.tidyverse.org/)*)

`pivot_wider()` - Makes a long data set wide.

`pivot_longer()` - Makes a wide data set long.

`separate()` - Splits column content across multiple columns.

`unite()` - Condenses content across multiple columns into a single column.

Dealing with row names *(https://tibble.tidyverse.org/reference/rownames.html)* (`library(tibble)` *(https://tibble.tidyverse.org/)*)

`rownames_to_column()` - Creates a column in your data frame from existing row names.

`column_to_rownames()` - Creates row names from a column in your data frame.

Data Visualization (`library(ggplot2)` *(https://ggplot2.tidyverse.org/)*

There are 3 components required for all ggplot2 plots: DATA, GEOM_FUNCTION, and `aes` MAPPINGS.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>))
```

Subsetting Data (`library(dplyr)` *(https://dplyr.tidyverse.org/)*)

`select()` - Filters data by column.

- Check out associated helper functions:
- Select specific columns:
    - `everything()`
    - `last_col()`
- Select columns by matching some aspect of the column name:
    - `starts_with()`
    - `ends_with()`
    - `contains()`
    - `matches()`
    - `num_range()`
- Select columns named in a character vector:
    - `all_of()`
    - `any_of()`
- Select a column using a function
    - `where()`

`rename()` - rename a column without selecting

`filter()` - Filters data by row.

- `filter()` often uses relational operators. Type `?Comparison` and `?match` in the console for more information.
- For filtering across multiple rows, check out `if_all()` and `if_any()`.

`slice()` - positional subsetting

Reordering rows (`library(dplyr)`)

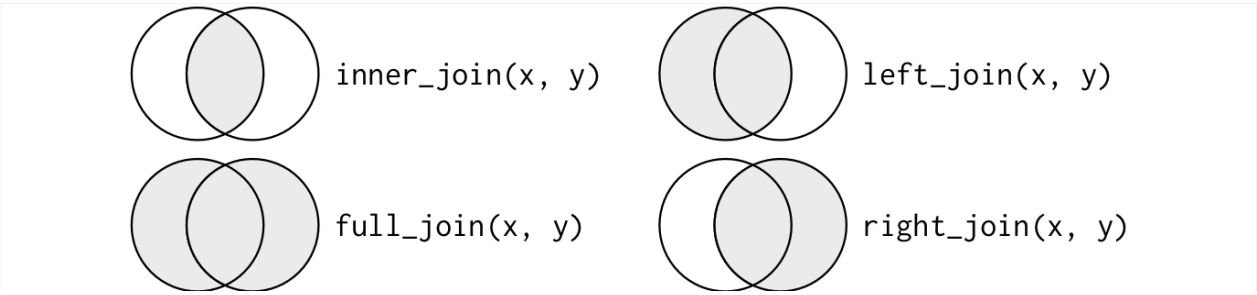
`arrange()` - orders rows by the values in specific columns

- Commonly used with `desc()`, which sorts values in descending order.

Joining Data frames (`library(dplyr)`)

Mutating Joins

`left_join()` - Output contains all rows from x
`right_join()` - Output contains all rows from y
`inner_join()` - Output contains matched rows from x
`full_join()` - Output contains all rows from x and y



Filtering joins

`semi_join()` - Outputs columns from x and rows from x that match y.
`anti_join()` - Outputs columns from x and rows from x that DO NOT match y.

Transforming Variables (`library(dplyr)`)

`mutate()` - Create new columns while keeping all existing columns

`transmute()` - Create new columns while dropping all existing columns.

To mutate across multiple columns using a single phrase use `across()`. Because `across()` requires you to select which columns you want to mutate, it can be used with the `select()` helper functions.

Split, apply, combine (`library(dplyr)`)

To apply some function to smaller subsets (groups) within your data, and return a summarized data frame, use `group_by()` with `summarize()`.

`group_by()` - group a data frame by a categorical variable so that a given operation can be performed per group / category.

`summarize()` - reduces multiple values down to a single summary *(https://dplyr.tidyverse.org/)* using specified functions (e.g., mean, median, standard deviation, etc.).

Other useful `tidyverse` packages

Working with dates

Check out `library(lubridate)` *(https://lubridate.tidyverse.org/)*.

Working with factors

Check out `library(forcats)` *(https://forcats.tidyverse.org/)*.

Looking for a `for loop` alternative

Check out `library(purr)` *(https://purrr.tidyverse.org/)*.

Using R on Biowulf

If you wish to use R on Biowulf, you can view modules available using `module -r avail '^R$'`. Loading the module requires an interactive session, `sinteractive` (unless submitting a job).

- Example of module load `module load R/3.5`
- After loading the module, to begin using R, type `R`.

For a list of currently installed packages in the default R environment, see https://hpc.nih.gov/apps/R.html#packages *(https://hpc.nih.gov/apps/R.html#packages)*.

For instructions on using R interactively or via a batch job, see the R Biowulf help page *(https://hpc.nih.gov/apps/R.html)*. You may also find our course documentation *(https://bioinformatics.ccr.cancer.gov/docs/reproducible-r-on-biowulf/)* for *Toward Reproducibility with R on Biowulf* helpful.

For information on obtaining a Biowulf account, click here *(https://hpc.nih.gov/docs/accounts.html)*.

Wrangling a realistic dataset

Provided Data

The provided data set is an example of a real count matrix returned from the NCI CCR Sequencing Facility (CCR-SF). The provided file (`./data/SF_example_RNASeq_1.txt`) contains RNAseq data for two sets of samples (Cell1 and Cell2) run in triplicate.

Data Challenges

There are some immediate problems with the data.

1. The column names begin with numbers, which are not syntactic with R.
2. The gene names are hybrids of Ensembl ID and gene symbols and will match neither if needed in the future. Therefore, these should be split into their own columns.
3. The values of the gene counts are in decimals. Most differential expression packages will expect integers.

Our challenge

First, we will want to fix the three immediate problems discussed above. Once those have been fixed, we are going to create a bar plot, using `ggplot2` to plot the total gene counts per sample. With that accomplished we are then going to format the data so that it can be used to construct a `DESeqDataSet` object. This object is necessary to apply different functions from `DESeq2` including differential expression analysis.

Note: While the instructions below include some guidance on the different functions required, you may need to include helper functions or modify function arguments to complete each task. Expected output will be shown below each prompt.

Step 1: Load the data.

Begin by loading the data and saving to an object named `dmat`.

{{Sdet}}

Solution}

```
library(tidyverse)
dmat<-read_delim("./data/SF_example_RNASeq_1.txt",col_names =TRUE)
```

{{Edet}}

Great! You should have a 10,000 x 7 tibble if you used a `readr` function for data loading.

```
dmat
```

```
## # A tibble: 10,000 × 7
##    gene_id           `1_Cell1_Rep1` `2_Cell1_Rep2` `3_Cell1_Rep3`
##    <chr>                      <dbl>          <dbl>          <dbl>
##  1 ENSG00000001630.…          6877.          6614           7058.
##  2 ENSG00000002016.…           283.           287.           287.
##  3 ENSG00000002330.…          1946           1662           2121
##  4 ENSG00000002834.…         17636          19333          18917
##  5 ENSG00000003056.…          3874           4107           4005
##  6 ENSG00000003393.…          2041           2150           2141
##  7 ENSG00000003989.…           279            345            305
##  8 ENSG00000004534.…          2695.          3031           2871
##  9 ENSG00000004838.…            42             52             39
## 10 ENSG00000004848.…             1              0              0
## # ℹ 9,990 more rows
## # ℹ 2 more variables: `5_Cell2_Rep2` <dbl>, `6_Cell2_Rep3` <dbl>
```

Step 2: Rename the Samples (column names)

Notice that the column names each begin with a number (e.g., 1_MB231_RNA1). Place an "S" at the beginning of each sample name using `rename_with()`, which is similar to `rename()`, but it uses a function to rename the columns. Look up the function `?paste()` to use with `rename()`. Overwrite `dmat` with `dmat`.

{{Sdet}}

Solution}

```
#The easiest way to do this is with the function paste
dmat<-rename_with(dmat,~ paste0("S",.x),contains("Cell"))

#for a gsub solution
#rename_with(dmat,~ gsub("^", "S", .x), contains("Cell")) #^ is a reg
```

{{Edet}}

```
colnames(dmat)
```

```
## [1] "gene_id"       "S1_Cell1_Rep1" "S2_Cell1_Rep2" "S3_Cell1_Rep3
## [5] "S4_Cell2_Rep1" "S5_Cell2_Rep2" "S6_Cell2_Rep3"
```

Step 3: Separate the gene abbreviation from the Ensembl ID

Separate the gene abbreviation from the Ensembl ID in column 1 using `separate()`. Save the output to a new object named `dmat2`.

{{Sdet}}

Solution}.

```
#separate gene abbreviation from ensembl id
dmat2<-separate(dmat, gene_id, into=c("Ensembl_ID","gene_abb"),sep="_
```

{{Edet}}

Store the newly separated columns (`Ensembl_ID` and `gene_abb`) in a new data frame named `gene_names` and drop the gene abbreviations from our working data frame (`dmat2`) because we ultimately want this to be a data matrix (overwrite `dmat2` with `dmat2`).

{{Sdet}}

Solution}

```
#separate gene abbreviation from ensembl id
gene_names<- dmat2 %>% select(1:2)

dmat2<-dmat2 %>% select(!gene_abb)
```

{{Edet}}

```
gene_names
```

```
## # A tibble: 10,000 × 2
##    Ensembl_ID         gene_abb
##    <chr>              <chr>
##  1 ENSG00000001630.11 CYP51A1
##  2 ENSG00000002016.12 RAD52
##  3 ENSG00000002330.9  BAD
##  4 ENSG00000002834.13 LASP1
##  5 ENSG00000003056.3  M6PR
##  6 ENSG00000003393.10 ALS2
##  7 ENSG00000003989.12 SLC7A2
##  8 ENSG00000004534.10 RBM6
##  9 ENSG00000004838.9  ZMYND10
```

```
## 10 ENSG00000004848.6  ARX
## # ℹ 9,990 more rows
```

```
dmat2
```

```
## # A tibble: 10,000 × 7
##    Ensembl_ID         S1_Cell1_Rep1 S2_Cell1_Rep2 S3_Cell1_Rep3 S4
##    <chr>                      <dbl>         <dbl>         <dbl>
##  1 ENSG00000001630.11         6877.         6614          7058.
##  2 ENSG00000002016.12          283.          287.          287.
##  3 ENSG00000002330.9          1946          1662          2121
##  4 ENSG00000002834.13        17636         19333         18917
##  5 ENSG00000003056.3          3874          4107          4005
##  6 ENSG00000003393.10         2041          2150          2141
##  7 ENSG00000003989.12          279           345           305
##  8 ENSG00000004534.10         2695.         3031          2871
##  9 ENSG00000004838.9            42            52            39
## 10 ENSG00000004848.6             1             0             0
## # ℹ 9,990 more rows
## # ℹ 2 more variables: S5_Cell2_Rep2 <dbl>, S6_Cell2_Rep3 <dbl>
```

Step 4: Convert the gene counts to integers

Many of the packages that handle RNASeq count data do not work correctly with decimal numbers. We need to convert these numbers to integers using `mutate()`. Save your transformed data frame to an object named `dmat3`.
{{Sdet}}

Solution}

```
dmat3<-dmat2 %>% mutate(across(where(is.numeric),~as.integer(round(.)
```

{{Edet}}

```
dmat3
```

```
## # A tibble: 10,000 × 7
##    Ensembl_ID         S1_Cell1_Rep1 S2_Cell1_Rep2 S3_Cell1_Rep3 S4
##    <chr>                      <int>         <int>         <int>
##  1 ENSG00000001630.11          6877          6614          7058
##  2 ENSG00000002016.12           283           287           287
```

```
##  3 ENSG00000002330.9           1946          1662          2121
##  4 ENSG00000002834.13         17636         19333         18917
##  5 ENSG00000003056.3           3874          4107          4005
##  6 ENSG00000003393.10          2041          2150          2141
##  7 ENSG00000003989.12           279           345           305
##  8 ENSG00000004534.10          2695          3031          2871
##  9 ENSG00000004838.9             42            52            39
## 10 ENSG00000004848.6              1             0             0
## # ℹ 9,990 more rows
## # ℹ 2 more variables: S5_Cell2_Rep2 <int>, S6_Cell2_Rep3 <int>
```

Step 5: Create a bar plot, using `ggplot2` to plot the total gene counts per sample.

In order to create a ggplot2 bar plot we will need to reshape the data. The sample names should be in a single column named `Sample` and the gene counts in a single column named `Count`. Use `pivot_longer()` and save the reshaped data to an object named `ldmat`.

{{Sdet}}

Solution}

```
ldmat<-pivot_longer(dmat3,starts_with("S"),names_to =c("Sample"),valu
```

{{Edet}}

```
ldmat
```

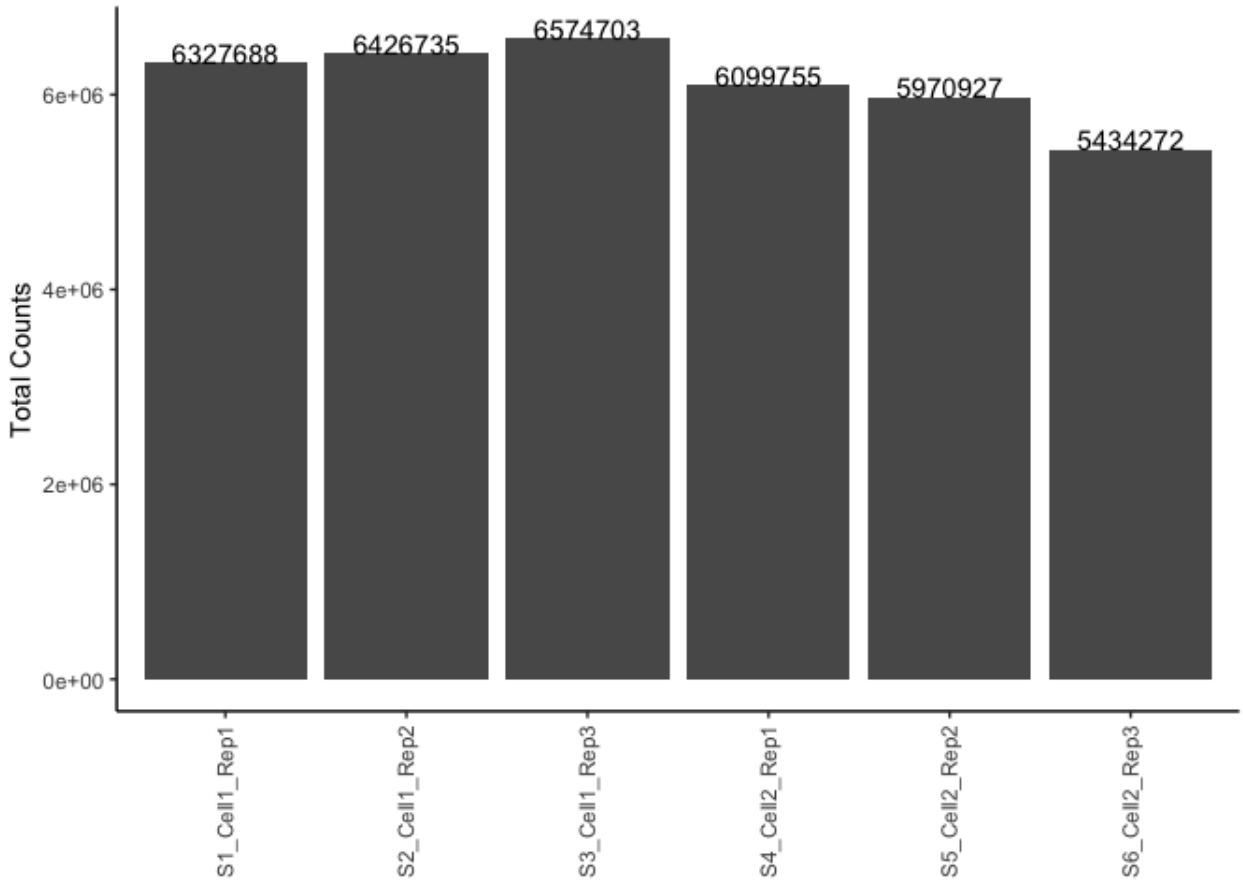
```
## # A tibble: 60,000 × 3
##    Ensembl_ID         Sample        Count
##    <chr>              <chr>         <int>
##  1 ENSG00000001630.11 S1_Cell1_Rep1  6877
##  2 ENSG00000001630.11 S2_Cell1_Rep2  6614
##  3 ENSG00000001630.11 S3_Cell1_Rep3  7058
##  4 ENSG00000001630.11 S4_Cell2_Rep1 11305
##  5 ENSG00000001630.11 S5_Cell2_Rep2 10761
##  6 ENSG00000001630.11 S6_Cell2_Rep3 10047
##  7 ENSG00000002016.12 S1_Cell1_Rep1   283
##  8 ENSG00000002016.12 S2_Cell1_Rep2   287
##  9 ENSG00000002016.12 S3_Cell1_Rep3   287
## 10 ENSG00000002016.12 S4_Cell2_Rep1   265
## # ℹ 59,990 more rows
```

Now create the plot.

{{Sdet}}

Solution}

```
ldmat %>%
  group_by(Sample) %>%
  summarize(Tcounts= sum(Count)) %>%
  ggplot() +
  geom_bar(aes(x=Sample,y=Tcounts),stat="identity") +
  geom_text(aes(x=Sample,y=Tcounts,label = Tcounts), vjust = 0)+
  theme_classic()+
  theme(axis.text.x = element_text(angle=90,vjust=0.5)) +
  labs(x=NULL,y="Total Counts")
```




```
#ggplot(ldmat) + #Shows the same but is less clean
#  geom_bar(aes(x=Sample,y=Count),stat="identity")
```

{{Edet}}

Step 6: Prepare objects needed to create a `DESeqDataSet` object.

> The object class used by the DESeq2 package to store the read counts and the intermediate estimated quantities during statistical analysis is the DESeqDataSet.
> --- Analyzing RNA-seq data with DESeq2 *(http://bioconductor.org/packages/devel/bioc/vignettes/DESeq2/inst/doc/DESeq2.html#the-deseqdataset)*

Constructing this object from a count matrix requires count data, sample information, and an experiment design. Our final objective is to prep a count matrix and sample information that can be used to create a `DESeqDataSet` object.

The count data currently stored in a data frame will need to be a matrix. To do this, first convert the current column Ensembl_ID to row names. Next, before converting to a matrix, filter genes with low expression. Keep only genes with 10 or more reads across all samples (Hint: check out the function `rowSums()`). Finally, convert the resulting data frame to a matrix (use `as.matrix()`). Save this object back to `dmat3`.

{{Sdet}}

Solution}

```
dmat3<-dmat3 %>% column_to_rownames("Ensembl_ID") %>%
  filter(rowSums(.) >= 10) %>%
  as.matrix()
```

{{Edet}}

```
head(dmat3)
```

```
##                    S1_Cell1_Rep1 S2_Cell1_Rep2 S3_Cell1_Rep3 S4_Ce
## ENSG00000001630.11          6877          6614          7058
## ENSG00000002016.12           283           287           287
## ENSG00000002330.9           1946          1662          2121
## ENSG00000002834.13         17636         19333         18917
## ENSG00000003056.3           3874          4107          4005
## ENSG00000003393.10          2041          2150          2141
##                    S5_Cell2_Rep2 S6_Cell2_Rep3
## ENSG00000001630.11         10761         10047
## ENSG00000002016.12           235           254
## ENSG00000002330.9            711           576
## ENSG00000002834.13          4464          3892
## ENSG00000003056.3           5703          4978
## ENSG00000003393.10          1624          1426
```

Next, we need to create some sample information to include with our count matrix. Because our RNAseq data includes two sets of samples run in triplicate for two cell lines, Cell1 and Cell2, we can create sample information from the sample names using the function `data.frame()`. The column names of our data matrix (`dmat3`) and the row names of our sample metadata (`samp_df`) should be in the same order.

{{Sdet}}

Solution}

```
samp_df<-data.frame(Sample=colnames(dmat3),Cell_line=rep(c("Cell1","C
```

{{Edet}}

```
samp_df
```

```
##                Cell_line Replicate
## S1_Cell1_Rep1     Cell1         1
## S2_Cell1_Rep2     Cell1         2
## S3_Cell1_Rep3     Cell1         3
## S4_Cell2_Rep1     Cell2         1
## S5_Cell2_Rep2     Cell2         2
## S6_Cell2_Rep3     Cell2         3
```

Now we have the files needed to create a `DESeqDataSet` object. Let's use the function `DESeqDataSetFromMatrix()`. The general template is as follows:

```
dds <- DESeqDataSetFromMatrix(countData = data_matrix,
                              colData = sample_info,
                              design = ~ condition)
```

For more information on constructing the `DESeqDataSet` object and using the package `DESeq2`, check out this vignette *(http://bioconductor.org/packages/devel/bioc/vignettes/DESeq2/inst/doc/DESeq2.html)*.

To install `DESeq2` use:

```
if (!require("BiocManager", quietly = TRUE))
    install.packages("BiocManager")

BiocManager::install("DESeq2")
```

And create the object

{{Sdet}}

Solution}

```
library(DESeq2)
dds <- DESeqDataSetFromMatrix(countData = dmat3,
                              colData = samp_df,
                              design = ~ Cell_line)
```

```
## Warning in DESeqDataSet(se, design = design, ignoreRank): some vai
## design formula are characters, converting to factors
```

```
dds
```

```
## class: DESeqDataSet
## dim: 3622 6
## metadata(1): version
## assays(1): counts
## rownames(3622): ENSG00000001630.11 ENSG00000002016.12 ...
##   ENSGR0000169100.8 ENSGR0000223773.2
## rowData names(0):
## colnames(6): S1_Cell1_Rep1 S2_Cell1_Rep2 ... S5_Cell2_Rep2
##   S6_Cell2_Rep3
## colData names(2): Cell_line Replicate
```

{{Edet}}

Getting the Data

Get the data

Lesson 1

No data available.

Lesson 2

No data available.

Lesson 3

Lesson 3 covered data import and reshaping. Data unavailable through base R or other R packages, can be downloaded here. The survey / species data sets were obtained from a data carpentry lesson Data Analysis and Visualization in R for Ecologists *(https://datacarpentry.org/R-ecology-lesson/02-starting-with-data.html)*.

Lesson 4

Lesson 4 focused on data visualization with ggplot2. A summary RNA-Seq file (`RNASeq_totalcounts_vs_totaltrans.xlsx`) was used for this lesson as well as the file `countB.csv`. Both files are also available in `Lesson3_data.zip` provided above.

Lesson 5

Lesson 5 introduced the package `dplyr` and the functions `select()`, `filter()`, and `arrange()`. Data used in this lesson derived from the RNA-Seq experiment `airway` and can be downloaded here

Lesson 6

Lesson 6 continued with `dplyr`, focusing on the functions `mutate()`, `group_by()`, and `summarize()`. Data used in this lesson derived from the RNA-Seq experiment `airway` and can be downloaded here

Lesson 7

No data available

Lesson 8

Data associated with the lesson 8 data wrangling challenge can be found here.

Practice Questions

Lesson 2: Help Session

This is our first coding help session. We have designed some practice problems to get you acquainted with using R before beginning to wrangle in our next lesson.

Practice problems

Which of the following will throw an error and why?

```
4_chr <- c('chr13', 'chr4','chr9','chr01')
chr.4 <- c('chr13', 'chr4','chr9','chr01')
.4chr <- c('chr13', 'chr4','chr9','chr01')
chr_4 <- c('chr13', 'chr4','chr9','chr01')
chr4 <- c('chr13', 'chr4','chr9','chr01')
```

{{Sdet}}

Solution}

```
4_chr <- c('chr13', 'chr4','chr9','chr01')
chr.4 <- c('chr13', 'chr4','chr9','chr01')
```

```
## Error: <text>:1:2: unexpected input
## 1: 4_
##      ^
```

```
.4chr <- c('chr13', 'chr4','chr9','chr01')
chr_4 <- c('chr13', 'chr4','chr9','chr01')
chr4 <- c('chr13', 'chr4','chr9','chr01')
```

```
## Error: <text>:1:3: unexpected symbol
## 1: .4chr
##       ^
```

{{Edet}}

Create the following objects; give each object an appropriate name (your best guess at what name to use is fine):

1. Create an object that has a value of your favorite gene name
2. Create an object containing a vector of numbers 8-16.
3. Create an object containing a vector of numbers 22-29.
4. Combine the vectors from question 4 and 5 and save to a new object.

{{Sdet}}

Solution}

```
(fav_gene<-"GH1")
## [1] "GH1"
(vec1<-c(8:16))
## [1]  8  9 10 11 12 13 14 15 16
(vec2<-c(22:29))
## [1] 22 23 24 25 26 27 28 29
(vec3<-c(vec1,vec2))
##  [1]  8  9 10 11 12 13 14 15 16 22 23 24 25 26 27 28 29
```

{{Edet}}

Create the following objects in R. What type of data are stored in these objects?

```
chromosome_name <- 'chr02'
ODval <- 0.47
chr_position <- '1001701'
question <- TRUE
irregular <- NA
```

{{Sdet}}

Solution}

```
typeof(chromosome_name)
## [1] "character"
typeof(ODval)
## [1] "double"
typeof(chr_position)
## [1] "character"
typeof(question)
## [1] "logical"
typeof(irregular)
## [1] "logical"
```

{{Edet}}

Here are some interesting vectors.

```
snps <- c('rs53576', 'rs1815739', 'rs6152', 'rs1799971')
snp_chromosomes <- c('3', '11', 'X', '6')
snp_positions <- c(8762685, 66560624, 67545785, 154039662)
```

If you combine, snp_positions with snp_chromosomes, what is the resulting data type?

{{Sdet}}

Solution}

```
typeof(c(snp_positions,snp_chromosomes))
```

```
## [1] "character"
```

{{Edet}}

Add 23792 to the snp_positions vector and overwrite the object.

{{Sdet}}

Solution}

```
(snp_positions <- c(snp_positions, 23792))
```

```
## [1]   8762685  66560624  67545785 154039662     23792
```

{{Edet}}

Use the following example data frame:

```
df<-data.frame(id=paste("Sample",1:10,sep="_"), cell=rep(factor(c("c
```

\
\

What are the column names? Rename the column "id" to "Sampleid". [You will likely need to look up how to do this. How can you find help?]

{{Sdet}}

Solution}

```
colnames(df)
```

```
## [1] "id"     "cell"   "counts"
```

```
colnames(df)[1] <- "Sampleid"
```

{{Edet}}

Save the column 'cell' from df to an object called 'cell_line'.

{{Sdet}}

Solution}

```
cell_line <- df$cell
cell_line
```

```
##  [1] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_A c
##  [7] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_B c
## [13] cell_line_B cell_line_B cell_line_B cell_line_B cell_line_B c
## [19] cell_line_B cell_line_B
## Levels: cell_line_A cell_line_B
```

{{Edet}}

Duplicate the column 'cell' and save to a new column of df called 'cell_line'.

{{Sdet}}

Solution}

```
(df$cell_line <- df$cell)
```

```
##  [1] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_A c
##  [7] cell_line_A cell_line_A cell_line_A cell_line_A cell_line_B c
## [13] cell_line_B cell_line_B cell_line_B cell_line_B cell_line_B c
## [19] cell_line_B cell_line_B
## Levels: cell_line_A cell_line_B
```

{{Edet}}

Look at the help documentation for `is.na()`. Use this function to determine whether there are NAs in the following vector:

```
vec_a<-c(88, 347, 53, 27, 94, NA,28, 409, NA)
```

{{Sdet}}

Solution}

```
is.na(vec_a) #returns logical vector
```

```
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

```
which(is.na(vec_a)) #provides index or location in vector
```

```
## [1] 6 9
```

{{Edet}}

Acknowledgements

Questions were inspired by or taken directly from datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html)*

Help Session Lesson 3

Loading data

1. Import data from the sheet "iris_data_long" from the excel workbook (file_path = "./data/iris_data.xlsx"). Make sure the column names are unique and do not contain spaces. Save the imported data to an object called `iris_long`.

   {{Sdet}}

   Solution}

```
iris_long<-readxl::read_excel("../data/iris_data.xlsx",sheet="ir
iris_long
```

```
## # A tibble: 600 × 4
##    Iris.ID Species Measurement.location Measurement
##      <dbl> <chr>   <chr>                       <dbl>
##  1       1 setosa  Sepal.Length                  5.1
##  2       1 setosa  Sepal.Width                   3.5
##  3       1 setosa  Petal.Length                  1.4
##  4       1 setosa  Petal.Width                   0.2
##  5       2 setosa  Sepal.Length                  4.9
##  6       2 setosa  Sepal.Width                   3
##  7       2 setosa  Petal.Length                  1.4
##  8       2 setosa  Petal.Width                   0.2
##  9       3 setosa  Sepal.Length                  4.7
## 10       3 setosa  Sepal.Width                   3.2
## # ℹ 590 more rows
```

   {{Edet}}

2. Import a tab delimited file (file_path= "./data/species_datacarpentry.txt"). Save the file to an object named `species`. `genus`,`species`, and `taxa` should be converted to factors upon import.

   {{Sdet}}

   Solution}

```

```

```
species<-readr::read_delim("../data/species_datacarpentry.txt",c
species
```

```
## # A tibble: 54 × 4
##    species_id genus            species         taxa
##    <chr>      <fct>            <fct>           <fct>
##  1 AB         Amphispiza       bilineata       Bird
##  2 AH         Ammospermophilus harrisi         Rodent
##  3 AS         Ammodramus       savannarum      Bird
##  4 BA         Baiomys          taylori         Rodent
##  5 CB         Campylorhynchus  brunneicapillus Bird
##  6 CM         Calamospiza      melanocorys     Bird
##  7 CQ         Callipepla       squamata        Bird
##  8 CS         Crotalus         scutalatus      Reptile
##  9 CT         Cnemidophorus    tigris          Reptile
## 10 CU         Cnemidophorus    uniparens       Reptile
## # ℹ 44 more rows
```

{{Edet}}

3. Load in a comma separated file with row names present (file_path= "./data/countB.csv") and save to an object named `countB`.

{{Sdet}}

Solution}

```
countB<-read.csv("../data/countB.csv",row.names=1)
head(countB)
```

```
##        SampleA_1 SampleA_2 SampleA_3 SampleB_1 SampleB_2 Samp
## Tspan6       703       567       867        71       970
## TNMD         490       482        18       342       935
## DPM1         921       797       622       661         8
## SCYL3        335       216       222       774       979
## FGR          574       574       515       584       941
## CFH          577       792       672       104       192
```

{{Edet}}

Challenge data load

1. Load in a tab delimited file (file_path= "./data/WebexSession_report.txt") using `read_delim()`. You will need to troubleshoot the error message and modify the function arguments as needed.

{{Sdet}}

Solution}

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────────
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.4     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyv
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>)
```

```
read_delim("../data/WebexSession_report.txt",delim="\t",locale =
```

```
## Rows: 10 Columns: 21
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr   (7): Name, Date, Invited, Registered, Duration, Network
## dbl   (1): Participant
## lgl  (11): Audio Type, Email, Company, Title, Phone Number, A
## time  (2): Start time, End time
##
## ℹ Use `spec()` to retrieve the full column specification for
## ℹ Specify the column types or set `show_col_types = FALSE` to
```

```
## # A tibble: 10 × 21
##    Participant `Audio Type` Name     Email Date  Invited Regi
##          <dbl> <lgl>        <chr>    <lgl> <chr> <chr>   <chr
##  1           1 NA           Partici… NA    6/8/… No      N/A
##  2           2 NA           Partici… NA    6/9/… <NA>    <NA>
```

```
##  3           3 NA           Partici… NA    6/10… No      N/A
##  4           4 NA           Partici… NA    6/11… <NA>    <NA>
##  5           5 NA           Partici… NA    6/12… No      N/A
##  6           6 NA           Partici… NA    6/13… <NA>    <NA>
##  7           7 NA           Partici… NA    6/14… No      N/A
##  8           8 NA           Partici… NA    6/15… <NA>    <NA>
##  9           9 NA           Partici… NA    6/16… Yes     N/A
## 10          NA NA           <NA>     NA    <NA>  <NA>    <NA>
## # ℹ 13 more variables: `End time` <time>, Duration <chr>, Com
## #   Title <lgl>, `Phone Number` <lgl>, `Address 1` <lgl>, `Ad
## #   City <lgl>, `State/Province` <lgl>, `Zip/Postal Code` <lg
## #   `Country/region` <lgl>, `Network joined from:` <chr>,
## #   `Internal Participant:` <chr>
```

```
head(read.delim("../data/WebexSession_report.txt",
                fileEncoding="UTF-16LE")) #via base R
```

```
##   All.sessions.in.Eastern.Daylight.Time..New.York..GMT.04.00.
## 1          Session detail for 'A Webex Meeting of some type':
## 2                                                  Participant
## 3                                                            1
## 4                                                            2
## 5                                                            3
## 6                                                            4
##             X.1   X.2     X.3     X.4        X.5        X.6
## 1
## 2          Name Email    Date Invited Registered Start time E
## 3 Participant 1  <NA>  6/8/22      No        N/A    1:00 PM
## 4 Participant 2  <NA>  6/9/22                       1:00 PM
## 5 Participant 3  <NA> 6/10/22      No        N/A   12:57 PM
## 6 Participant 4  <NA> 6/11/22                      12:57 PM
##       X.9  X.10         X.11      X.12      X.13 X.14
## 1
## 2 Company Title Phone Number Address 1 Address 2 City State/P
## 3
## 4
## 5
## 6
##              X.16           X.17                X.18
## 1
## 2 Zip/Postal Code Country/region Network joined from: Interna
## 3                                          External
## 4
```

```
## 5                                                External
## 6
```

{{Edet}}

Reshaping data

1. Reshape `iris_long` to a wide format. Your new column names will contain names from `Measurement.location`. Your wide data should look as follows:

```
## # A tibble: 150 × 6
##    Iris.ID Species Sepal.Length Sepal.Width Petal.Length Peta
##      <dbl> <chr>          <dbl>       <dbl>        <dbl>
##  1       1 setosa           5.1         3.5          1.4
##  2       2 setosa           4.9         3            1.4
##  3       3 setosa           4.7         3.2          1.3
##  4       4 setosa           4.6         3.1          1.5
##  5       5 setosa           5           3.6          1.4
##  6       6 setosa           5.4         3.9          1.7
##  7       7 setosa           4.6         3.4          1.4
##  8       8 setosa           5           3.4          1.5
##  9       9 setosa           4.4         2.9          1.4
## 10      10 setosa           4.9         3.1          1.5
## # ℹ 140 more rows
```

{{Sdet}}

Solution}

```
tidyr::pivot_wider(iris_long, names_from = Measurement.location,
```

{{Edet}}

2. Let's use `table4a` from the `tidyr` package. Use `pivot_longer()` to place the year columns in a column named `year` and their values in a column named `cases`.

```
library(tidyr)
data(table4a)
table4a
```

```
## # A tibble: 3 × 3
##   country     `1999` `2000`
```

```
##   <chr>        <dbl>  <dbl>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

{{Sdet}}

Solution}

```
pivot_longer(table4a,2:3, names_to = "year", values_to = "cases"
```

{{Edet}}

```
## # A tibble: 6 × 3
##   country     year   cases
##   <chr>       <chr>  <dbl>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

3. Separate the column `rate` from tidyr's table3 into two columns: `cases` and `population`.

```
data(table3)
table3
```

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

{{Sdet}}

Solution}

```
separate(table3, rate, into = c("cases", "population"))
```

{{Edet}}

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

Reshape challenge

1. Use `pivot_longer` to reshape countB. Your reshaped data should look the same as the data below.

{{Sdet}}

Solution}

```
library(tidyverse)
countB<-read.csv("../data/countB.csv",row.names=1) %>%
  rownames_to_column("Feature")

countB_l<-pivot_longer(countB,
         cols=2:length(countB),
         names_to = c(".value", "Replicate"),
         names_sep = "_") #reshaping data so that all replicates

tibble(countB_l)
```

{{Edet}}

```
## # A tibble: 27 × 4
##    Feature Replicate SampleA SampleB
##    <chr>   <chr>       <int>   <int>
##  1 Tspan6  1             703      71
##  2 Tspan6  2             567     970
##  3 Tspan6  3             867     242
##  4 TNMD    1             490     342
```

```
##  5 TNMD    2              482     935
##  6 TNMD    3               18     469
##  7 DPM1    1              921     661
##  8 DPM1    2              797       8
##  9 DPM1    3              622     500
## 10 SCYL3   1              335     774
## # ℹ 17 more rows
```

Help Session Lesson 4

Plotting with ggplot2

For the following plots, let's use the diamonds data (`?diamonds`).

> The diamonds dataset comes in ggplot2 and contains information about ~54,000 diamonds, including the price, carat, color, clarity, and cut of each diamond. --- R4DS *(https://r4ds.had.co.nz/data-visualisation.html#exercises-3)*

```
library(ggplot2)
data(diamonds)
diamonds
```

```
## # A tibble: 53,940 × 10
##    carat cut       color clarity depth table price     x     y
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dl
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2
## # ℹ 53,930 more rows
```

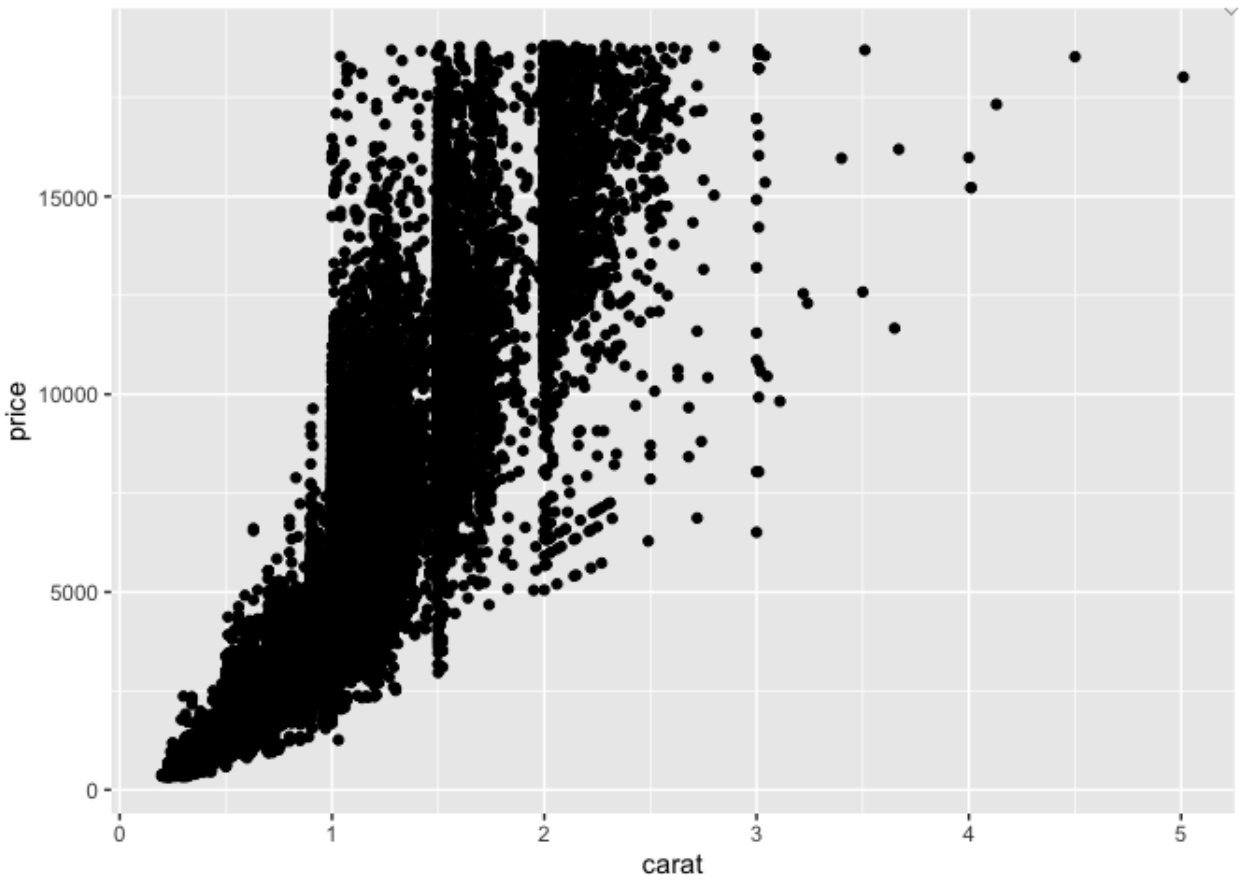
Create a scatter plot demonstrating how `carat` (x axis) relates to `price` (y axis).

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price))
```

{{Edet}}



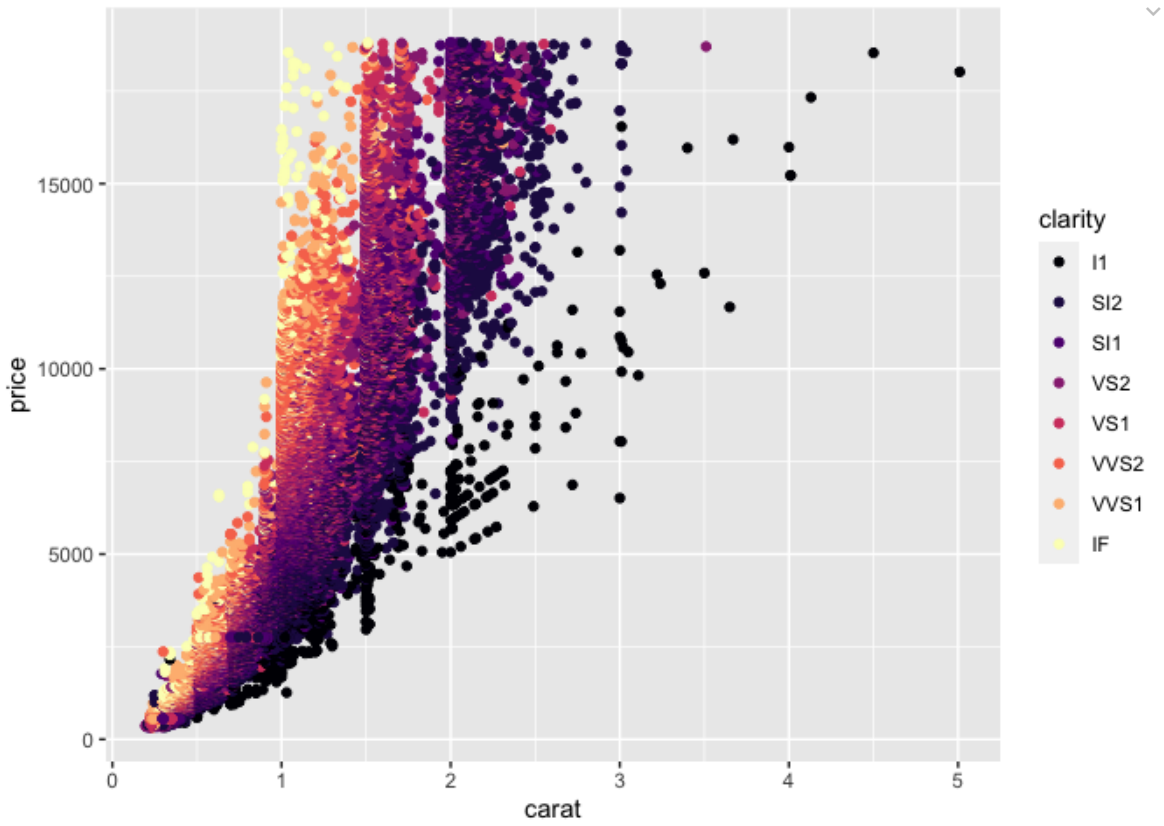
- Color the points above by `clarity` and scale the colors using the viridis package, option "magma".

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price,color=clarity))
  viridis::scale_color_viridis(discrete=TRUE,option="magma")
```

{{Edet}}



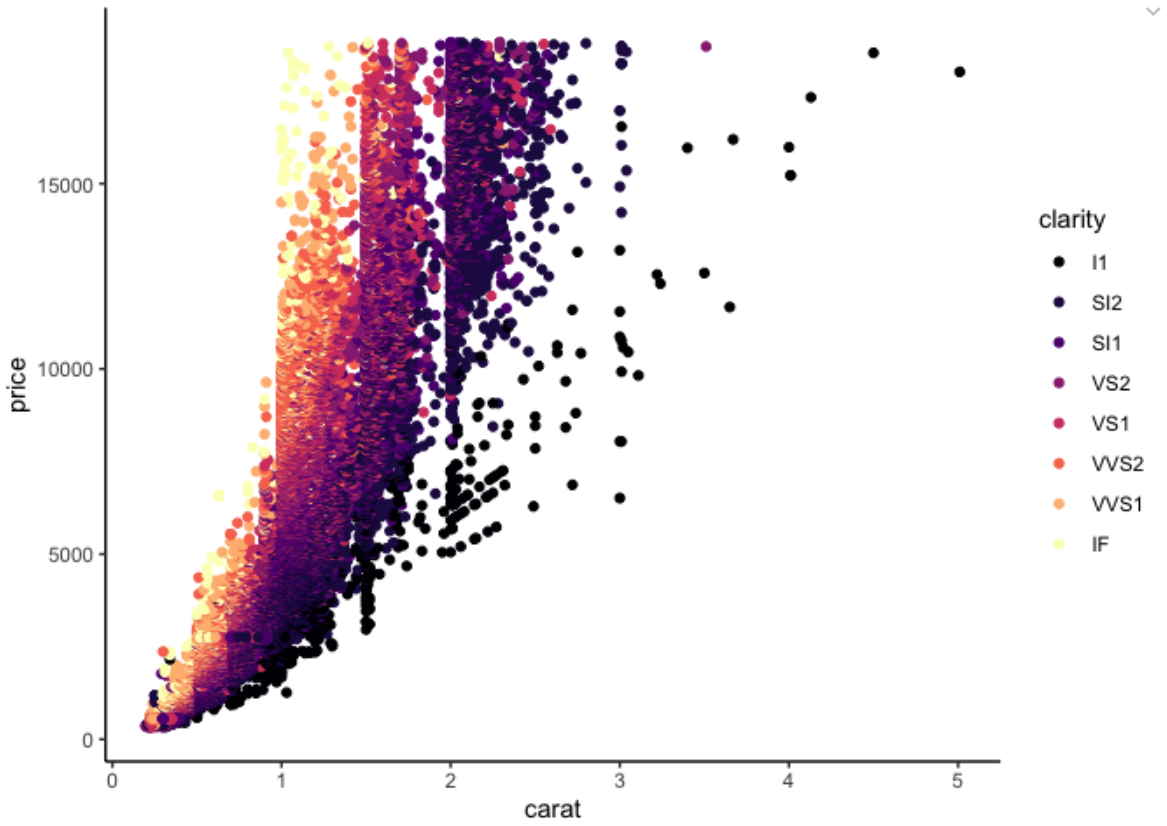
- Apply the complete theme, `theme_classic()`.

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price,color=clarity)) +
  viridis::scale_color_viridis(discrete=TRUE,option="magma") +
  theme_classic()
```

{{Edet}}



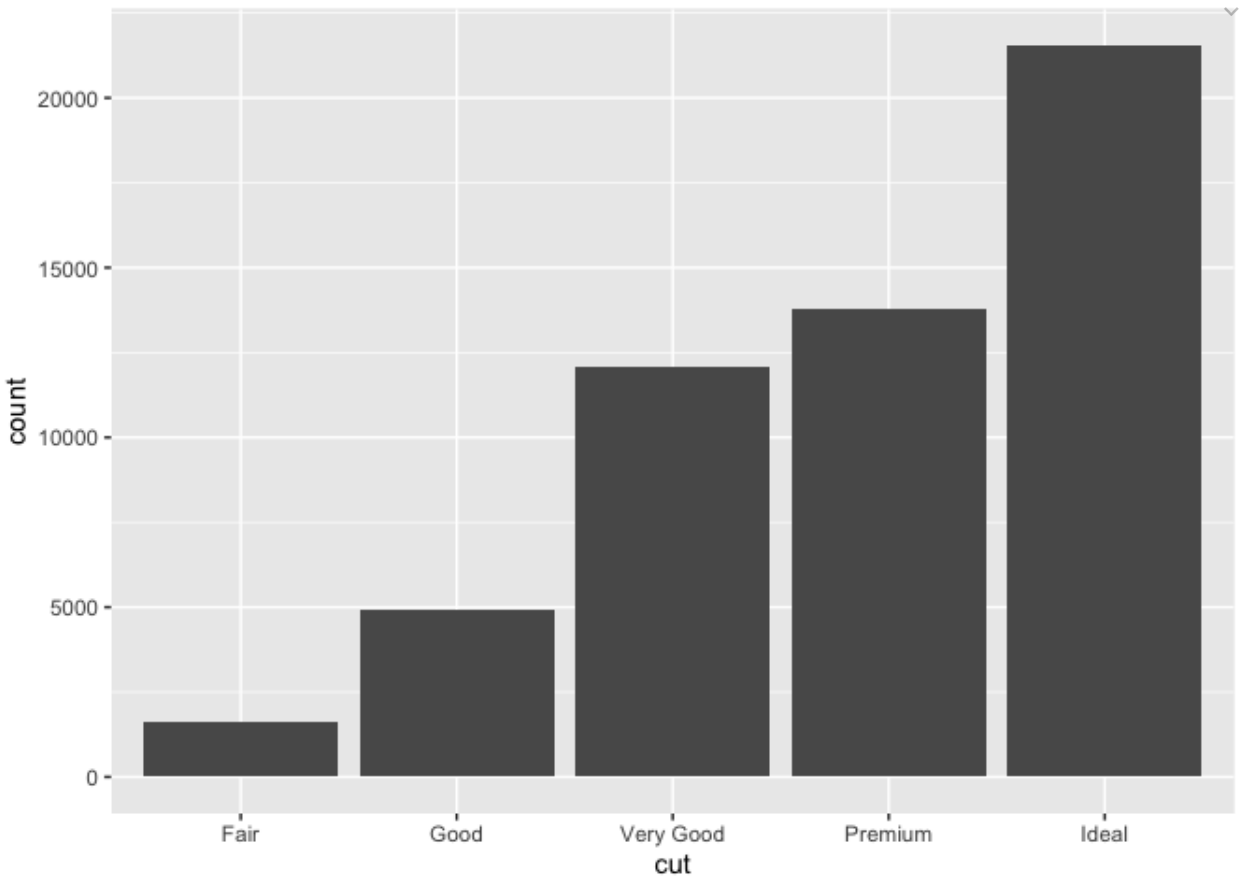
Create a bar chart displaying the number of diamonds per `cut`. Be sure to check out the help documentation for `geom_bar()`.

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```

{{Edet}}



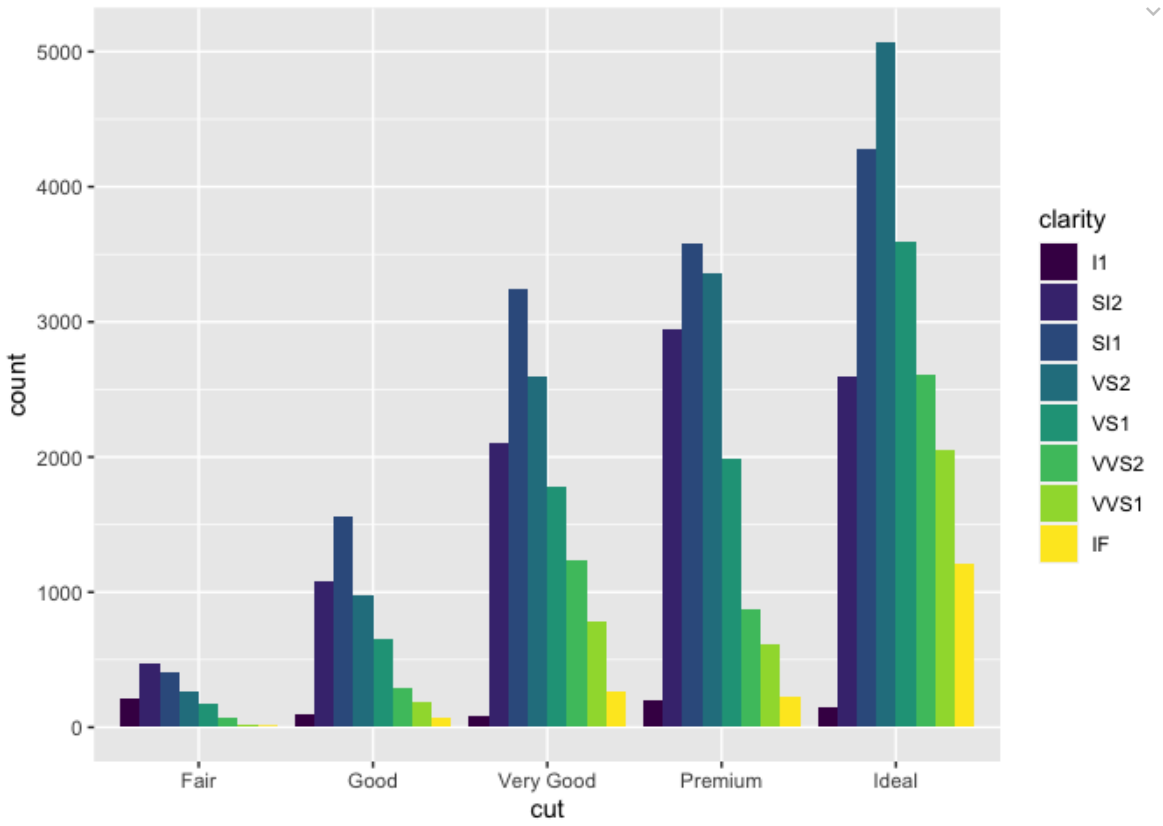
- Fill the bars by `clarity`. Modify the **position** of the bars so that they are set to dodge.

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill=clarity),position="dodge"
```

{{Edet}}



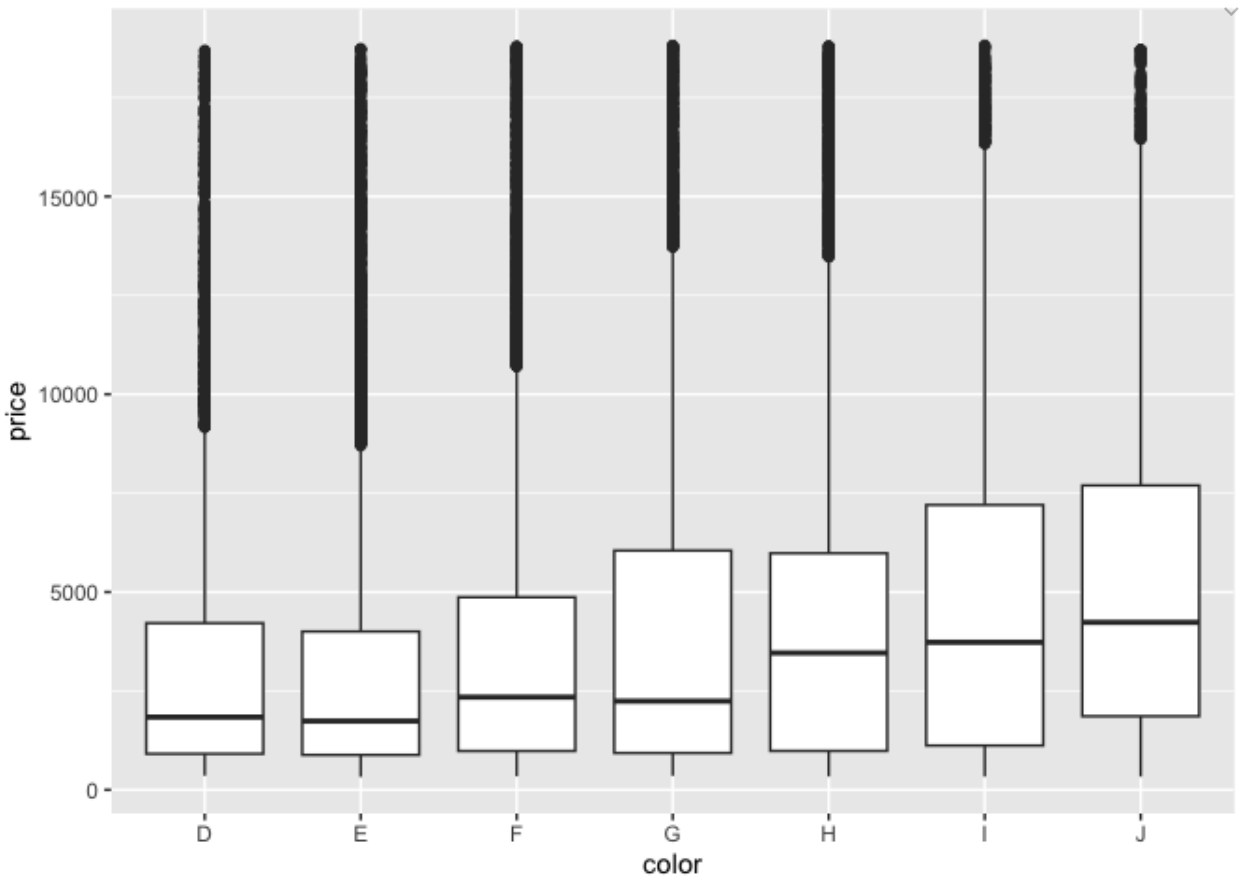
Examine how the `price` of a diamond changes across different diamond `color` categories using a boxplot.

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(x = color, y = price))
```

{{Edet}}



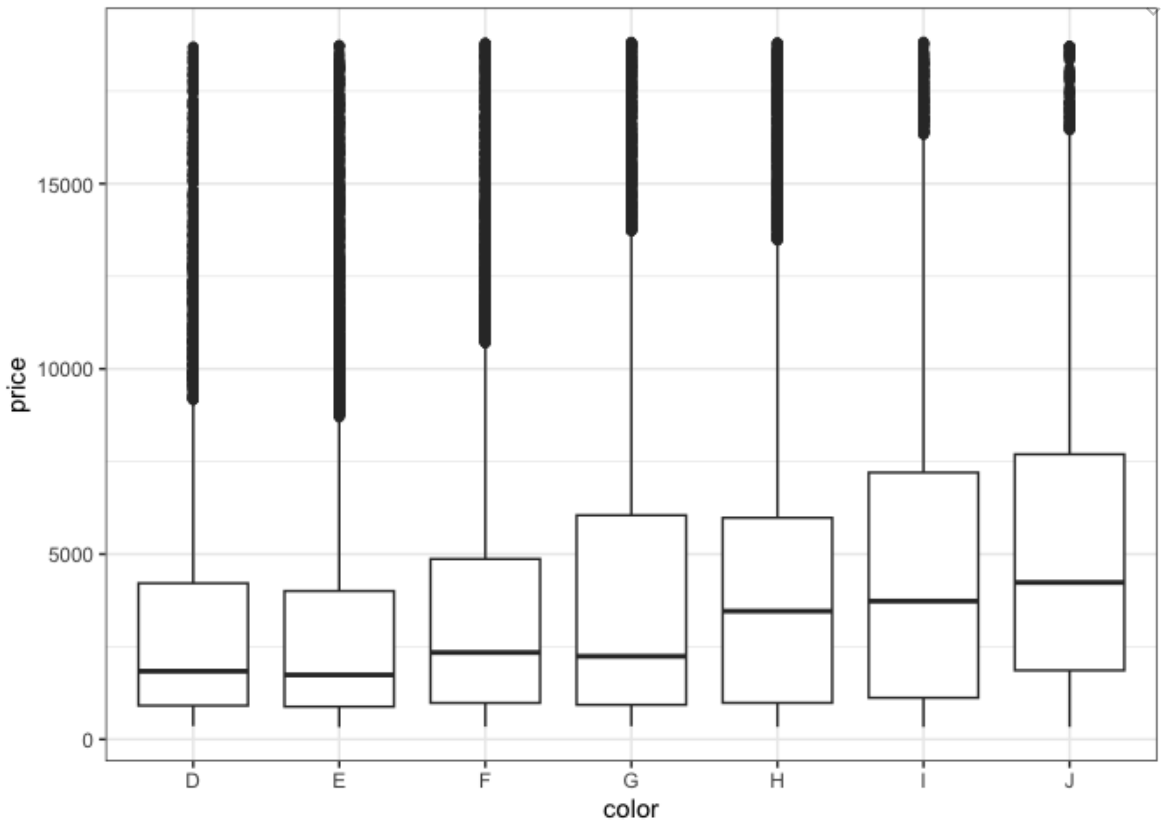
- Apply the complete theme, `theme_bw()`.

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(x = color, y = price))+
  theme_bw()
```

{{Edet}}



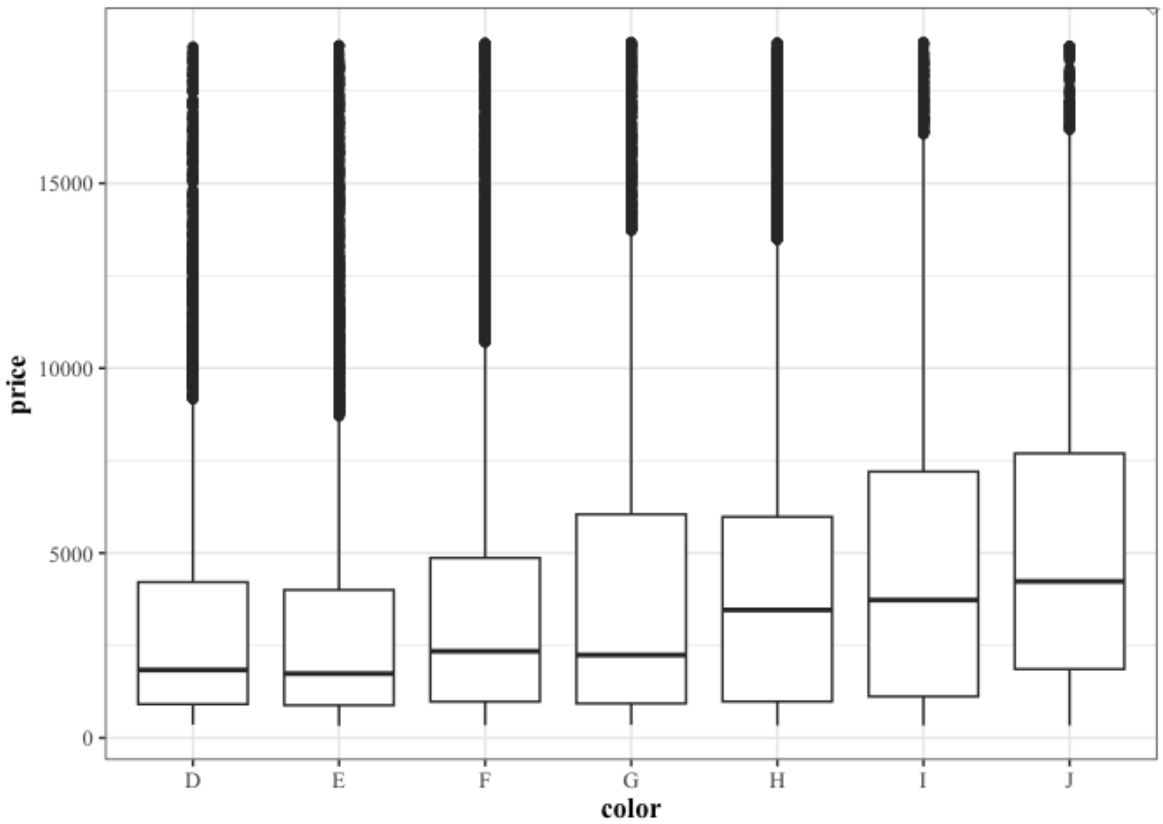
- Change the font of all text elements to "Times New Roman" and change the size of the font to 12. Bold the x and y axis labels.

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(x = color, y = price))+
  theme_bw()+
  theme(text=element_text(family="Times New Roman",size=12),
        axis.title = element_text(face="bold"))
```

{{Edet}}



Challenge Question

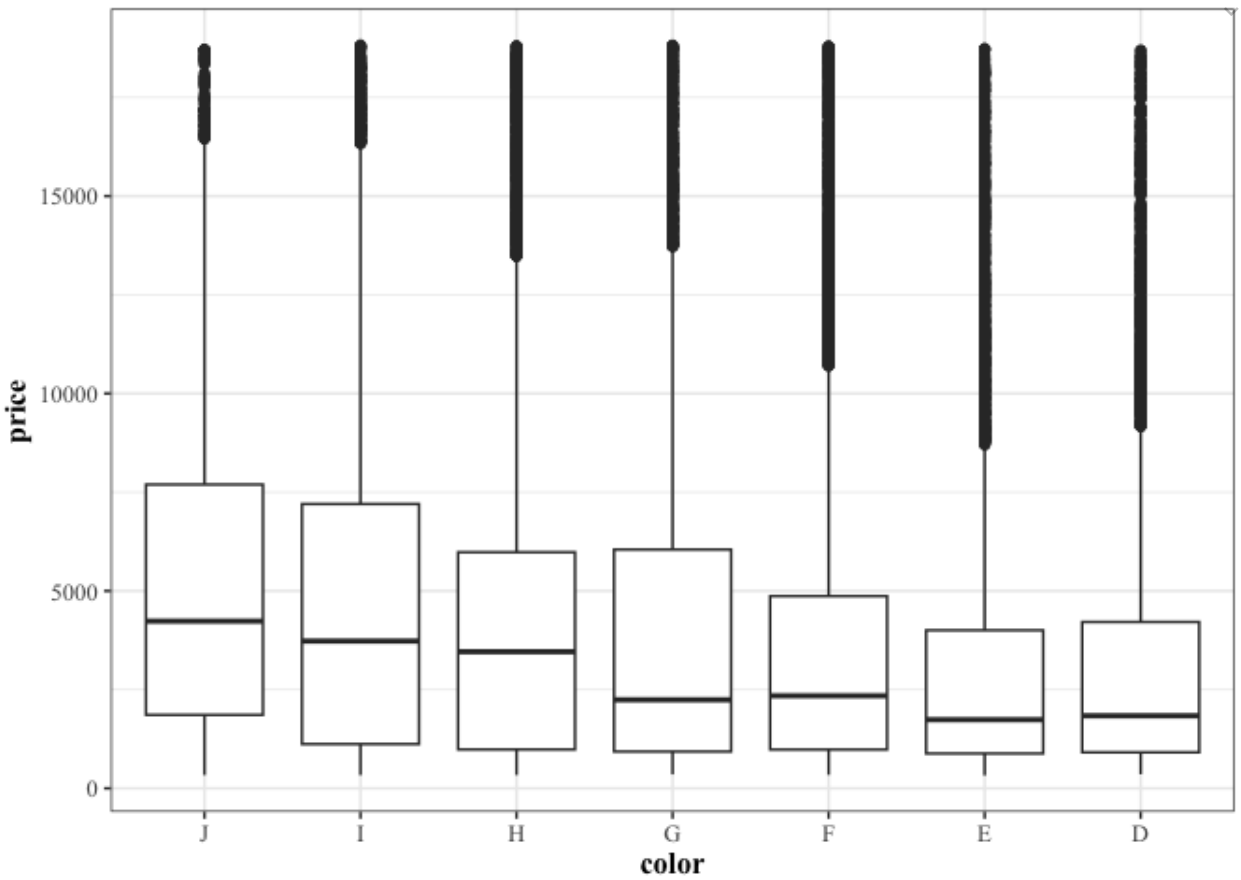
Using the boxplot you created above, reorder the x-axis so that color is organized from worst (J) to best (D). There are multiple possible solutions. **Hint:** Check out functions in the forcats package (a tidyverse core package)

{{Sdet}}

Solution}

```
ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(x = forcats::fct_rev(color), y = price))
  labs(x="color",y="price")+
  theme_bw()+
  theme(text=element_text(family="Times New Roman",size=12),
        axis.title = element_text(face="bold"))
```

{{Edet}}



Putting it all together

- Load in the comma separated file "./data/countB.csv" and save to an object named `gcounts`.

{{Sdet}}

Solution}

```
gcounts<-readr::read_csv("../data/countB.csv")
```

```
## New names:
## Rows: 9 Columns: 7
## ── Column specification
## ──────────────────────────────────────────────────────── Deli
## (1): ...1 dbl (6): SampleA_1, SampleA_2, SampleA_3, SampleB_1
## SampleB_3
## ℹ Use `spec()` to retrieve the full column specification for
## Specify the column types or set `show_col_types = FALSE` to q
## • `` -> `...1`
```

```
colnames(gcounts)[1]<-"Gene"
gcounts
```

```
## # A tibble: 9 × 7
##   Gene   SampleA_1 SampleA_2 SampleA_3 SampleB_1 SampleB_2 Sa
##   <chr>      <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 Tspan6       703       567       867        71       970
## 2 TNMD         490       482        18       342       935
## 3 DPM1         921       797       622       661         8
## 4 SCYL3        335       216       222       774       979
## 5 FGR          574       574       515       584       941
## 6 CFH          577       792       672       104       192
## 7 FUCA2        798       766       995        27       756
## 8 GCLC         822       874       923       705       667
## 9 NFYA         622       793       918       868       334
```

{{Edet}}

- Plot the values (gene counts) from Sample A on the y axis and sample B on the x axis. Hint: you will need to reshape the data to accomplish this task.

{{Sdet}}

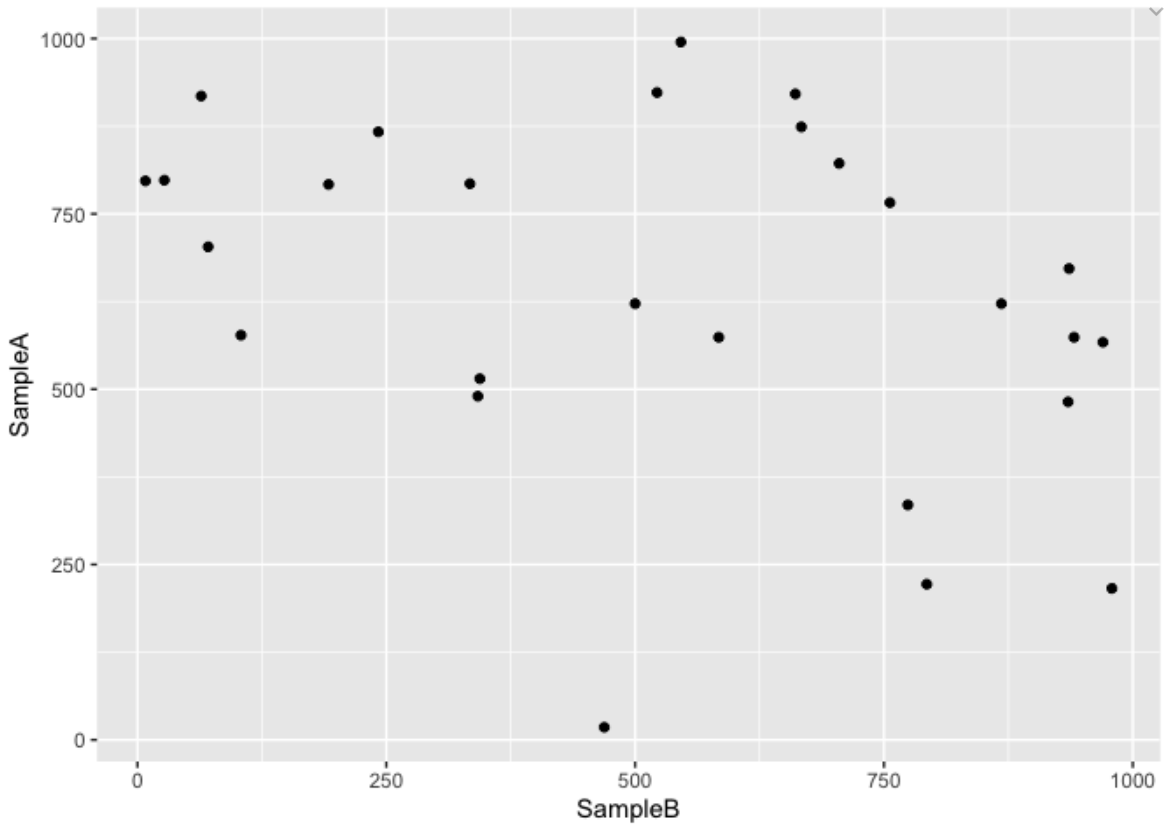
Solution}

```
library(tidyverse)

gcount2<-pivot_longer(gcounts,
  cols=2:length(gcounts),
  names_to = c(".value", "Replicate"),
  names_sep = "_"
) #reshaping data so that all replicates are stacked in a single

ggplot(data=gcount2) +
  geom_point(aes(x=SampleB,y=SampleA))
```

{{Edet}}



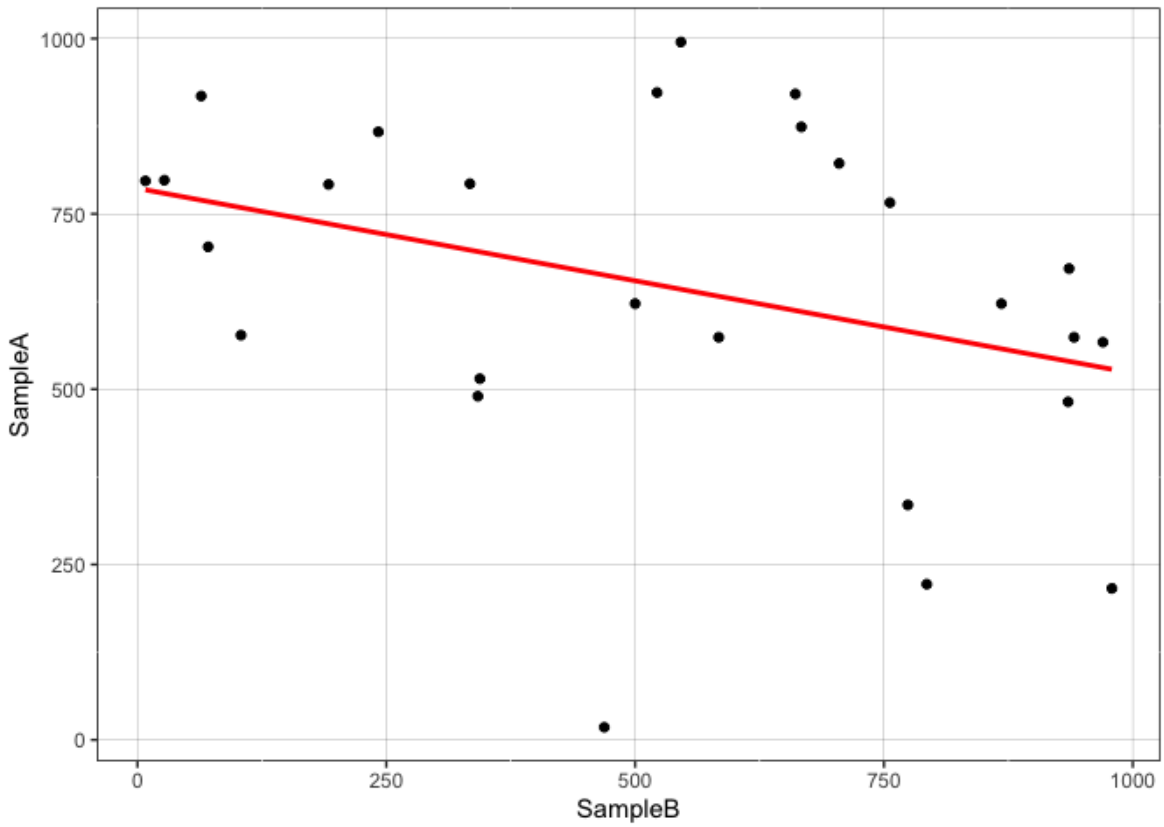
- Add a linear model to your scatter plot (See `geom_smooth()`). Also, the trend line should be red and the confidence interval around the trend line should NOT be visible. Change the panel background to white.

{{Sdet}}

Solution}

```
ggplot(data=gcount2,aes(x=SampleB,y=SampleA)) +
  geom_point() +
  geom_smooth(method="lm",color="red", se=FALSE)+
  theme(panel.background = element_rect(fill = "white", colour =
        panel.grid.major = element_line(color="black",size = 0.0
```

{{Edet}}


1000
750
500
250
0
SampleA
0
250
500
750
1000
SampleB

Help Session Lesson 5

All solutions should use the pipe.

1. Import the file "./data/filtlowabund_scaledcounts_airways.txt" and save to an object named `sc`. Create a subset data frame from `sc` that only includes the columns `sample`, `cell`, `dex`, `transcript`, and `counts_scaled` and only rows that include the treatment "untrt" and the transcripts "ACTN1" and "ANAPC4"?

   {{Sdet}}

   Solution}

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────────
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.4     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyv
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>)
```

```
sc<-read_delim("../data/filtlowabund_scaledcounts_airways.txt")
```

```
## Rows: 127408 Columns: 18
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (11): feature, SampleName, cell, dex, albut, Run, Experim
## dbl  (6): sample, counts, avgLength, TMM, multiplier, counts_
## lgl  (1): .abundant
##
## ℹ Use `spec()` to retrieve the full column specification for
## ℹ Specify the column types or set `show_col_types = FALSE` to
```

```
cnames<-c('sample', 'cell', 'dex', 'transcript', 'counts_scaled'
sc<-sc %>% select(all_of(cnames)) %>% filter(dex == "untrt" & (t
```

{{Edet}}

2. Using `dexp` ("./data/diffexp_results_edger_airways.txt") create a data frame containing the top 5 differentially expressed genes and save to an object named `top5`. Top genes in this case will have the smallest `FDR` corrected p-value and an absolute value of the log fold change greater than 2. See `dplyr::slice()`.

   {{Sdet}}

   Solution}

```
dexp<-read_delim("../data/diffexp_results_edger_airways.txt")
```

```
## Rows: 15926 Columns: 10
## ── Column specification ────────────────────────────────────────
## Delimiter: "\t"
## chr (4): feature, albut, transcript, ref_genome
## dbl (5): logFC, logCPM, F, PValue, FDR
## lgl (1): .abundant
##
## ℹ Use `spec()` to retrieve the full column specification for
## ℹ Specify the column types or set `show_col_types = FALSE` to
```

```
top5<- dexp %>% dplyr::filter(abs(logFC) > 2) %>% arrange(FDR) %
```

{{Edet}}

3. Filter `sc` to contain only the top 5 differentially expressed genes.

   {{Sdet}}

   Solution}

```
sc %>% dplyr::filter(transcript %in%  top5$transcript)
```

```
## # A tibble: 0 × 5
## # ℹ 5 variables: sample <dbl>, cell <chr>, dex <chr>, transcr
## #   counts_scaled <dbl>
```

{{Edet}}

4. Select only columns of type character from `sc`.

   {{Sdet}}

   Solution}

```
sc %>% select(where(is.character))
```

```
## # A tibble: 8 × 3
##   cell    dex   transcript
##   <chr>   <chr> <chr>
## 1 N61311  untrt ANAPC4
## 2 N61311  untrt ACTN1
## 3 N052611 untrt ANAPC4
## 4 N052611 untrt ACTN1
## 5 N080611 untrt ANAPC4
## 6 N080611 untrt ACTN1
## 7 N061011 untrt ANAPC4
## 8 N061011 untrt ACTN1
```

   {{Edet}}

5. Select all columns from `dexp` except `.abundant` and `PValue`. Keep only rows with `FDR` less than or equal to 0.01.

   {{Sdet}}

   Solution}

```
dexp %>% select(-c(.abundant,PValue)) %>% filter(FDR <= 0.01)
```

```
## # A tibble: 2,763 × 8
##    feature         albut transcript ref_genome  logFC logCPM
##    <chr>           <chr> <chr>      <chr>       <dbl>  <dbl>
##  1 ENSG00000000003 untrt TSPAN6     hg38       -0.390  5.06
##  2 ENSG00000000971 untrt CFH        hg38        0.417  8.09
##  3 ENSG00000001167 untrt NFYA       hg38       -0.509  4.13
##  4 ENSG00000002834 untrt LASP1      hg38        0.388  8.39
##  5 ENSG00000003096 untrt KLHL13     hg38       -0.949  4.16
##  6 ENSG00000003402 untrt CFLAR      hg38        1.18   6.90
##  7 ENSG00000003987 untrt MTMR7      hg38        0.993  0.341
##  8 ENSG00000004059 untrt ARF5       hg38        0.358  5.84
```

```
##  9 ENSG00000004487 untrt KDM1A      hg38        -0.308  5.86
## 10 ENSG00000004700 untrt RECQL      hg38         0.360  5.60
## # ℹ 2,753 more rows
```

{{Edet}}

6. Import the file "./data/airway_rawcount.csv". Use the function `rename()` to rename the first column. Use the pipe from `magrittr` to import and rename successively without intermediate steps or function nesting. Save to an object named `acount`.

{{Sdet}}

Solution}

```
acount<-read_csv("../data/airway_rawcount.csv") %>%
  dplyr::rename("Feature" = "...1")
```

```
## New names:
## Rows: 64102 Columns: 9
## ── Column specification
## ──────────────────────────────────────────────────────── Deli
## (1): ...1 dbl (8): SRR1039508, SRR1039509, SRR1039512, SRR103
## SRR1039...
## ℹ Use `spec()` to retrieve the full column specification for
## Specify the column types or set `show_col_types = FALSE` to q
## • `` -> `...1`
```

{{Edet}}

7. Use filter on the object `acount` to keep only genes that had a count greater than 10 in at least one sample

{{Sdet}}

Solution}

```
bcount<- acount %>%
  filter(if_any(where(is.numeric), ~.> 10))
```

{{Edet}}

8. Challenge Question: Filter genes from `acount` that had a total count less than ten across all samples. Hint: Use `column_to_rownames` and look up rowSums().

{{Sdet}}

Solution}

```
f_acount<-acount %>% column_to_rownames("Feature") %>% filter(ro
```

{{Edet}}

Help Session Lesson 6

Let's grab some data.

```
library(tidyverse)
acount_smeta<-read_tsv("../data/countsANDmeta.txt")
acount_smeta

#raw count data
acount<-read_csv("../data/airway_rawcount.csv") %>%
  dplyr::rename("Feature" = "...1")
acount


#differential expression results
dexp<-read_delim("../data/diffexp_results_edger_airways.txt")
dexp
```

All solutions should use the pipe.

1. Filter `acount` ("./data/airway_rawcount.csv") to include genes NOT found in our differential expression results (`dexp`).

   {{Sdet}}

   Solution}

   ```
   acount %>% filter(!Feature %in% dexp$feature)
   ```

   ```
   ## # A tibble: 48,176 × 9
   ##    Feature     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SR
   ##    <chr>            <dbl>      <dbl>      <dbl>      <dbl>
   ##  1 ENSG000000…          0          0          0          0
   ##  2 ENSG000000…          0          0          2          0
   ##  3 ENSG000000…         10          2          9          2
   ##  4 ENSG000000…          3          0          3          1
   ##  5 ENSG000000…          2          0          1          0
   ##  6 ENSG000000…          0          0          1          0
   ##  7 ENSG000000…          4          6         22         10
   ##  8 ENSG000000…          0          0          0          1
   ##  9 ENSG000000…          3          1          0          0
   ```

```
## 10 ENSG000000…          0          0          0          0
## # ℹ 48,166 more rows
## # ℹ 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

```
#OR

acount %>% anti_join(dexp, by=c("Feature"="feature"))
```

```
## # A tibble: 48,176 × 9
##    Feature     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SR
##    <chr>            <dbl>      <dbl>      <dbl>      <dbl>
##  1 ENSG000000…          0          0          0          0
##  2 ENSG000000…          0          0          2          0
##  3 ENSG000000…         10          2          9          2
##  4 ENSG000000…          3          0          3          1
##  5 ENSG000000…          2          0          1          0
##  6 ENSG000000…          0          0          1          0
##  7 ENSG000000…          4          6         22         10
##  8 ENSG000000…          0          0          0          1
##  9 ENSG000000…          3          1          0          0
## 10 ENSG000000…          0          0          0          0
## # ℹ 48,166 more rows
## # ℹ 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

{{Edet}}

2. From `acounts_smeta` ("./data/countsANDmeta.txt"), retrieve the bottom five genes with the lowest mean raw counts by `dex`. What are the dimensions of the resulting data frame? Why are there more than 5 rows?

{{Sdet}}

Solution}

```
acount_smeta %>%
  group_by(dex,Feature) %>%
  summarize(mean_counts=mean(Count))  %>%
  slice_min(n=5, order_by=mean_counts) %>%
  glimpse()
```

```
## `summarise()` has grouped output by 'dex'. You can override u
## argument.
```

```
## Rows: 67,285
## Columns: 3
## Groups: dex [2]
## $ dex         <chr> "trt", "trt", "trt", "trt", "trt", "trt",
## $ Feature     <chr> "ENSG00000000005", "ENSG00000000938", "EN
## $ mean_counts <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

{{Edet}}

3. Using mutate apply a base-10 logarithmic transformation to the numeric columns in `acount`; add a pseudocount of 1 prior to this transformation. Save the resulting data frame to an object called log10counts.

{{Sdet}}

Solution}

```
log10counts<- acount %>% mutate(across(where(is.numeric),~log10(
log10counts
```

```
## # A tibble: 64,102 × 9
##    Feature     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SR
##    <chr>            <dbl>      <dbl>      <dbl>      <dbl>
##  1 ENSG000000…       2.83       2.65       2.94       2.61
##  2 ENSG000000…       0          0          0          0
##  3 ENSG000000…       2.67       2.71       2.79       2.56
##  4 ENSG000000…       2.42       2.33       2.42       2.22
##  5 ENSG000000…       1.79       1.75       1.61       1.56
##  6 ENSG000000…       0          0          0.477      0
##  7 ENSG000000…       3.51       3.57       3.79       3.63
##  8 ENSG000000…       3.16       3.03       3.24       2.95
##  9 ENSG000000…       2.72       2.58       2.78       2.69
## 10 ENSG000000…       2.60       2.37       2.67       2.25
## # ℹ 64,092 more rows
## # ℹ 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

{{Edet}}

4. Create a column in `dexp` ("./data/diffexp_results_edger_airways.txt") called `Expression`. This column should say "Down-regulated" if `logFC` is less than -1 or "Up-regulated" if `logFC` is greater than 1. All other values should say "None". Hint: Look up help for `case_when()`.

{{Sdet}}

Solution}

```
dexp_new<-dexp %>% mutate(Expression=case_when(logFC < -1 ~ "Dow
                                               logFC > -1 & logFC < 1 ~
                                               logFC > 1 ~ "Up-regulate
```

{{Edet}}

Challenge question:

1. Calculate the mean raw counts for each gene by treatment in `acount_smeta`. Combine these results with the differential expression results. Your resulting data frame should resemble the following:

   {{Sdet}}

   Solution}

```
a<-acount_smeta %>%
  group_by(dex, Feature) %>%
  summarise(mean_count = mean(Count)) %>%
  pivot_wider(names_from=dex,values_from=mean_count,
              names_prefix="Mean_Counts_") %>%
  right_join(dexp, by=c("Feature" = "feature"))
```

```
## `summarise()` has grouped output by 'dex'. You can override u
## argument.
```

{{Edet}}

```
## # A tibble: 15,926 × 12
##    Feature         Mean_Counts_trt Mean_Counts_untrt albut tr
##    <chr>                     <dbl>             <dbl> <chr> <c
##  1 ENSG00000000003           619.               865   untrt TS
##  2 ENSG00000000419           547.               523   untrt DP
##  3 ENSG00000000457           234.               250.  untrt SC
##  4 ENSG00000000460            53.2               63.5 untrt C1
##  5 ENSG00000000971          6738.              5331.  untrt CF
##  6 ENSG00000001036          1123.              1487.  untrt FU
##  7 ENSG00000001084           573.               658.  untrt GC
##  8 ENSG00000001167           316                469   untrt NF
##  9 ENSG00000001460           168.               208   untrt ST
## 10 ENSG00000001461          2545               3113.  untrt NI
## # ℹ 15,916 more rows
```

```
## # ℹ 6 more variables: .abundant <lgl>, logFC <dbl>, logCPM <d
## #   PValue <dbl>, FDR <dbl>
```

Additional Resources

Additional Resources

Getting started with R

1. Hands on Programming with R *(https://rstudio-education.github.io/hopr/index.html)*
2. R for Data Science (R4DS) *(https://r4ds.had.co.nz/index.html)*

R Cheatsheets and references

1. Navigating RStudio cheatsheet
2. R reference card *(https://cran.r-project.org/doc/contrib/Short-refcard.pdf)*
3. readr / readxl cheatsheet
4. Tidyr (data reshaping) cheatsheet
5. Stringr / regex cheatsheet
6. Data Visualization (ggplot2) cheatsheet
7. Data Transformation (dplyr) cheatsheet
8. Factors with forcats cheatsheet
9. Working with Dates (lubridate) cheatsheet

Cheatsheets are reproduced here from https://www.rstudio.com/resources/cheatsheets/ *(https://www.rstudio.com/resources/cheatsheets/)*.

Other Resources

1. Helpful search engine for R: rseek *(https://rseek.org/)*
2. Test your regular expressions *(https://regex101.com/)*
3. Troubleshooting Errors *(https://bioinformatics.ccr.cancer.gov/docs/btep-coding-club/Troubleshooting_Rprog/)*