Introductory R for Novices

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

Table of Contents

Welcome

Getting Started with R

Intro to Data Wrangling

Practice Exercises

Additional Resources

Introductory R for Novices

Course Description

This course, designed for novices, will introduce the foundational skills necessary to begin to analyze and visualize data in R. The content for this course is similar to past introductory R courses, but the pace of the course will be much slower to benefit novices.

Why learn R? R is a great resource for statistical analysis, data visualization, and report generation. R also provides packages and functions specific to the analysis of -omics data through efforts like Bioconductor.

This course includes 3-parts:

Part 1: Getting Started with R

- Topics covered in Part 1 will focus on the basics of R Programming including getting started with R and RStudio, creating and manipulating R objects, and understanding and manipulating vectors and other data structures.

Part 2: Introduction to Data Wrangling

- Now that you have an understanding of the basics, Part 2 will show you how to work with tabular data. Topics covered include filtering, transforming, summarizing, and reshaping data using the Tidyverse suite of packages.

Part 3: Introduction to Data Visualization

- In Part 3, you will learn to visualize your data. Though multiple R graphics systems will be introduced, Part 3 will focus exclusively on visualizing data using `ggplot2`.

Course Materials

This course will be taught using R and RStudio on Biowulf. To use R on Biowulf, you must have an NIH HPC account *(https://hpc.nih.gov/docs/accounts.html)*. If you do not have Biowulf, this course can be taken using a local R installation.

R Installation Instructions

- **Macbook**: Follow these instructions *(https://posit.co/download/rstudio-desktop/)*.
- **Windows**: R and RStudio installation on Windows requires administrative privileges. NCI researchers can request installation from service.cancer.gov *(https://service.cancer.gov/ncisp)*.

This is not required if you have a Biowulf account.

Lesson Recordings

Video recordings of BTEP Coding Club events can be found in the BTEP Video Archive *(https://bioinformatics.ccr.cancer.gov/btep/btep-video-archive-of-past-classes/)* 24-48 hours following any given event.

Getting Started with R

Getting Started with R

This course is the first part of a larger 3-part course designed for novices.

Material covered in Part 1 focuses on the basics of R Programming including getting started with R and RStudio, creating and manipulating R objects, and understanding and manipulating vectors and other data structures.

Lessons

1. April 22, 2025 - Introduction to R and RStudio
2. April 24, 2025 - Basics of R Programming: R Objects and Data Types
3. April 29, 2025 - Basics of R Programming: Vectors
4. May 1, 2025 - Introduction to R Data Structures: Data Import
5. May 6, 2025 - R Data Structures: Data Frames

Required Course Materials

This course will use R on Biowulf. To use R on Biowulf, you must have an NIH HPC account. However, if you do not have Biowulf, this course can be taken using a local R installation.

Lesson 1: Introduction to R and RStudio IDE

Learning Objectives

To understand:
1. the difference between R and RStudioIDE.
2. how to work within the RStudio environment including:

- creating an Rproject and Rscript
- navigating between directories
- using functions
- obtaining help

By the end of this section, you should be able to easily navigate and explore your RStudio environment.

What is R?

R is both a computational language and environment for statistical computing and graphics. It is open-source and widely used by scientists and non-scientists, not just bioinformaticians. Base packages of R are built into your initial installation, but R functionality is greatly improved by installing other packages. R as a programming language is based on the S language, developed by Bell laboratories. R is maintained by a network of collaborators from around the world, and core contributors are known as the *R Core team* (Term used for citations). However, R is also a resource for and by scientists, and R functionality makes it easy to develop and share packages on any topic. Check out more about R on The R Project for Statistical Computing *(https://www.r-project.org/about.html)* website.

Why R?

R is a particularly great resource for statistical analyses, plotting, and report generating. The fact that it is widely used means that users do not need to reinvent the wheel. There is a package available for most types of analyses, and if users need help, it is only a Google search away. As of now, CRAN houses +22,000 available packages. There are also many field specific packages, including those useful in the -omics (genomics, transcriptomics, metabolomics, etc.). For example, the latest version of Bioconductor (v 3.20) includes 2,289 software packages, 431 experiment data packages, 928 annotation packages, 30 workflows, and 5 books.

Where do we get R packages?

To take full advantage of R, you need to install R packages. R packages are loadable extensions that contain code, data, documentation, and tests in a standardized, easy to share

format that can easily be installed by R users. The primary repository for R packages is the Comprehensive R Archive Network (CRAN). CRAN *(https://cran.r-project.org/#:~:text=CRAN%20is%20a%20network%20of,you%20to%20minimize%20network%20load.)* 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("packageName")`. 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 packages use `library(devtools)` followed by `install_github()`. Many genomics and other packages useful to biologists / molecular biologists can be found on Bioconductor *(https://www.bioconductor.org/)*. Bioconductor and Bioconductor packages use BiocManager for installation; see here *(https://www.bioconductor.org/install/)*.

METACRAN *(https://www.r-pkg.org/)* is a useful database that allows you to search and browse CRAN/R packages.

Ways to run R

R is a programming language and it "comes with an environment or console that can read and execute your code" *(https://www.r-bloggers.com/2022/01/how-to-install-and-update-r-and-rstudio/)*. R can be used via command line interactively, command line using a script *(https://support.rstudio.com/hc/en-us/articles/218012917-How-to-run-R-scripts-from-the-command-line)*, or interactively through an environment. This course will demonstrate the utility of the RStudio integrated development environment (IDE).

What is RStudio?

RStudio *(https://posit.co/products/open-source/rstudio/)* is an integrated development environment for R, and now python. RStudio includes a console, editor, and tools for plotting, history, debugging, and work space management. It provides a graphic user interface for working with R, thereby making R more user friendly. RStudio is open-source and can be installed locally or used through a browser (RStudio Server or Posit Cloud). We will be showcasing RStudio Server on Biowulf *(https://hpc.nih.gov/apps/RStudio.html)* via HPC Open OnDemand *(https://hpc.nih.gov/ondemand/index.html)*, but we highly encourage new users to install R and RStudio locally to their PC or macbook.

What is Posit?

Posit *(https://posit.co/)* is a company that creates and maintains a variety of software products (some free and others proprietary) including the RStudio IDE.

Installing R and RStudio

Macbook: Follow these instructions *(https://posit.co/download/rstudio-desktop/)*.
Windows: Request installation from service.cancer.gov *(https://service.cancer.gov/ncisp)*.

> Check out this blog *(https://www.r-bloggers.com/2022/01/how-to-install-and-update-r-and-rstudio/)* for information related to updating R and RStudio.

There is also an RStudio User Guide *(https://docs.posit.co/ide/user/)*.

Getting Started with R and R Studio

This tutorial closely follows the "Intro to R and RStudio for Genomics" lesson provided by datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/index.html)*.

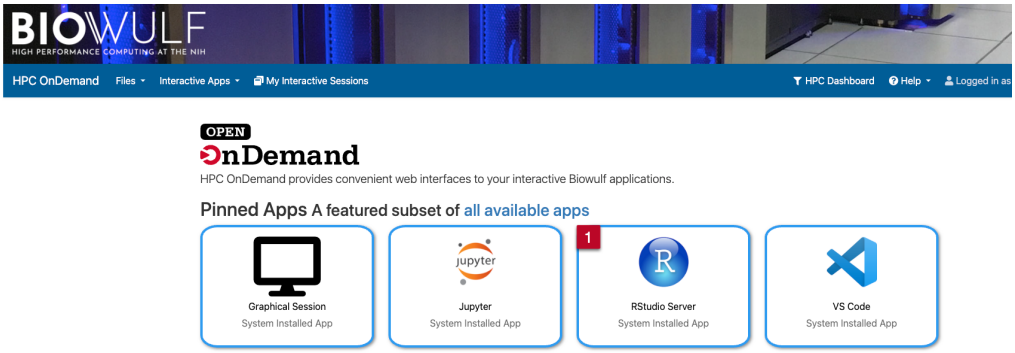
Connect to RStudio on NIH HPC Open OnDemand

NIH HPC Open OnDemand *(https://hpc.nih.gov/ondemand/index.html)* provides an online dashboard for users to easily access command line interactive sessions, graphical linux desktop environments, and interactive applications including RStudio, MATLAB, IGV, iDEP, VS Code, and Jupyter Notebook. To use NIH HPC Open OnDemand, you must have an NIH HPC account *(https://hpc.nih.gov/docs/accounts.html)*. If you are interested in bioinformatics, an NIH HPC account is highly recommended. These accounts are available for a nominal fee of $40 per month.

To connect to Open OnDemand make sure you are on the NIH Network and click on the following link: https://hpcondemand.nih.gov *(https://hpcondemand.nih.gov)*. This will take you to the HPC Open OnDemand dashboard.

From there you will need to:

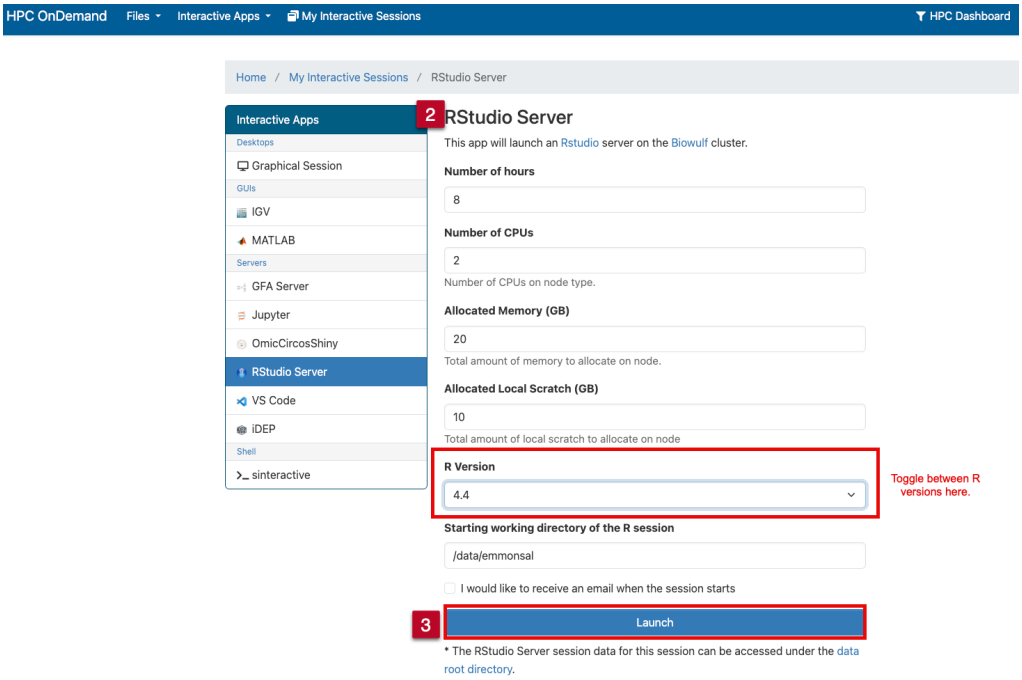
1. Select RStudio Server.



Step 1: Select RStudio Server from the selection of pinned applications.

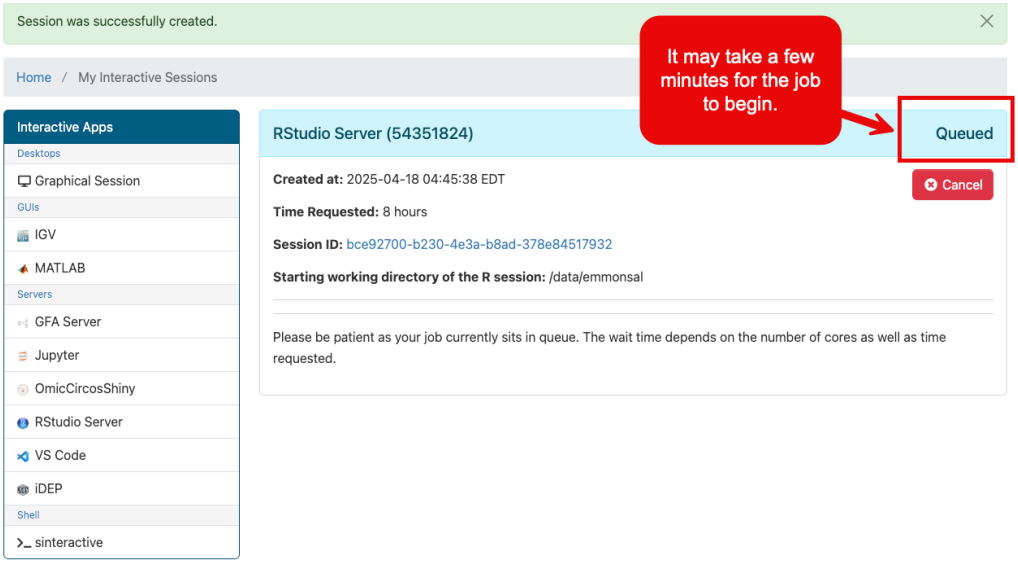
2. Select parameters for your RStudio session including the version of R you want to use.

3. Click "Launch" to start the session.



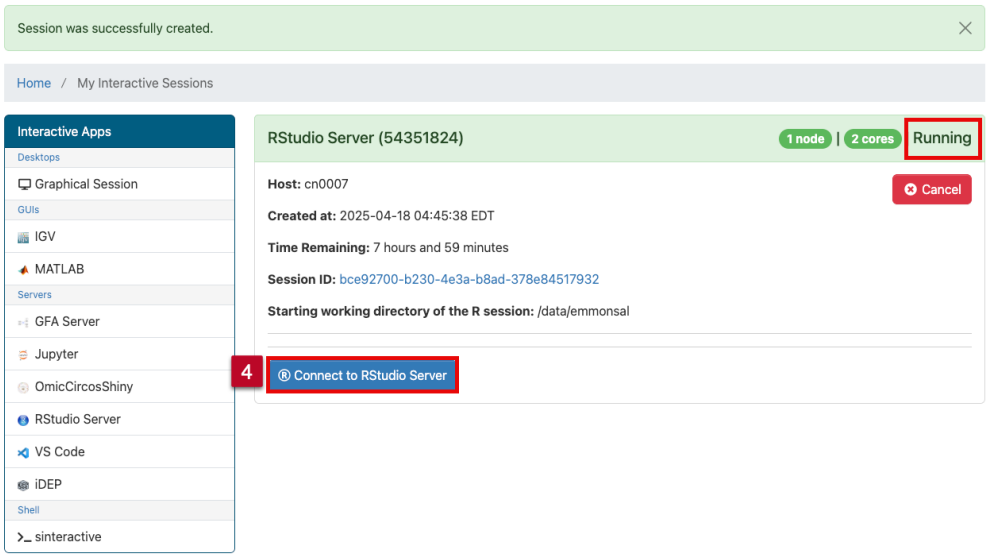
Step 2, 3: Alter any job parameters as you see fit and launch the session.

Your session will be queued, and it may take a few minutes to shift to "Running".



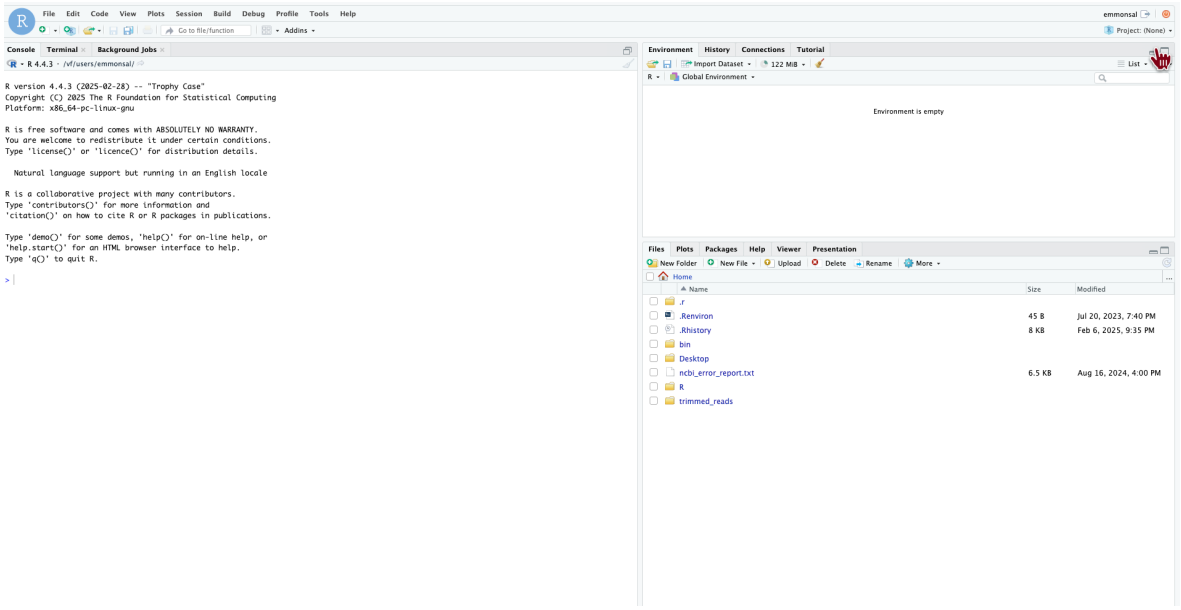
Session is queued.

4. When the session switches to "Running", click "Connect to RStudio Server".



Step 4: Connect to RStudio Server.

Congratulations! You are now connected.



RStudio Server on Biowulf

Using RStudio Server on Biowulf will allow you to 1. interact with your files on Biowulf, 2. use HPC resources (CPUs, RAM, etc.), and 3. also interact with local files.

Creating an R project

If you intend to use R for upcoming analysis projects, you will want to create R projects. R projects automatically set your working directory to the directory specified for a given project. R projects are beneficial because they "keep all the files associated with a given project (input

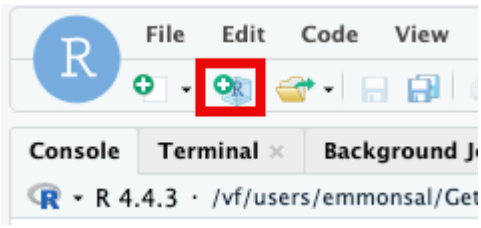
data, R scripts, analytical results, and figures) together in one directory" *(https://r4ds.hadley.nz/workflow-scripts.html#rstudio-projects)*.

Creating an R project *(https://docs.posit.co/ide/user/ide/guide/code/projects.html)* for each project you are working on facilitates organization and scientific reproducibility.

> An RStudio project allows you to more easily:
>
> - Save data, files, variables, packages, etc. related to a specific analysis project
> - Restart work where you left off
> - Collaborate, especially if you are using version control such as git. --- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html)*

R projects simplify data reproducibility by allowing us to use relative file paths that will translate well when sharing the project.

To start a new R project, select `File > New Project...` or use the R project button (See image below).



A New project wizard will appear. Click `New Directory` and `New Project`. Choose a new directory name....perhaps **"Getting_Started_with_R"**?

While we will not select `renv` today, this option will make a project more reproducible. See below. To make your project more reproducible, consider clicking the option box for `renv`.

The R project file ends in .Rproj. "This file contains various project options and can also be used as a shortcut for opening the project directly from the filesystem." *(https://docs.posit.co/ide/user/ide/guide/code/projects.html)*

Why renv?

R projects allow us to easily share data, code, and other related information, but this only scratches the surface of what is required for true data analysis reproducibility.

Too often an R script will fail simply due to a clash in package dependencies. Versions are important. R versions change over time; Bioconductor versions evolve, and R packages change. While we can include session info using the `sessionInfo()` function (more on functions later) at the end of a script or markdown file, this in no way facilitates our ability to truly

replicate the infrastructure surrounding our code. Thankfully, there are R packages available that help us do just that.

"The renv package helps you create reproducible environments for your R projects" *(https://rstudio.github.io/renv/index.html)*, primarily by tracking and managing package dependencies.

Read more about `renv` here *(https://rstudio.github.io/renv/articles/renv.html)*.

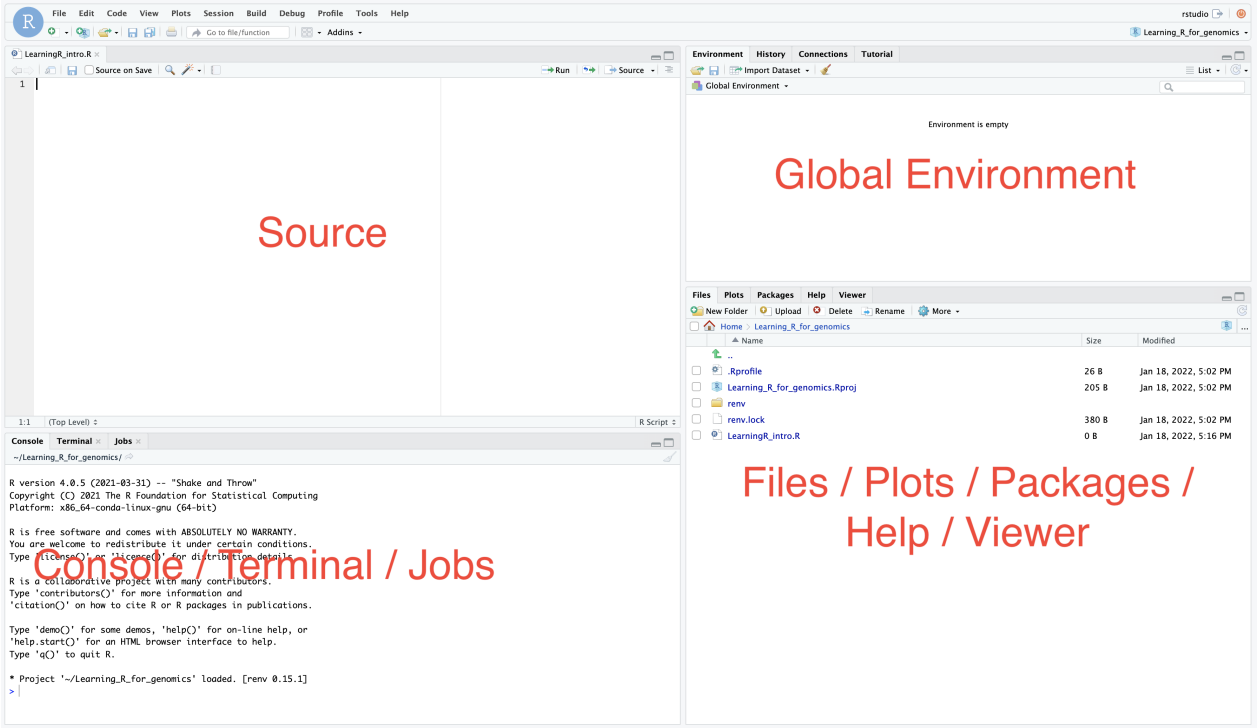
> **Reproducibility**
>
> There is even more that can be done to make projects reproducible beyond R Projects and `renv`. For example, you can use version control (git), R packages, and containerization (e.g., Singularity, Docker).

Creating an R script

As we learn more about R and start learning 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, from R console using `source()` and from linux command line using `Rscript`.

To create an R script, click `File > New File > R Script`. You can save your script by clicking on the floppy disk icon. You can name your script whatever you want, perhaps "Lesson_1". R scripts end in `.R`. Save your R script to your working directory, which will be the default location on RStudio Server.

Introduction to the RStudio layout



Let's look a bit into our RStudio layout.

> **Source**: This pane is where you will write/view R scripts. Some outputs (such as if you view a dataset using `View()`) will appear as a tab here.
> **Console/Terminal/Jobs**: This is actually where you see the execution of commands. This is the same display you would see if you were using R at the command line without RStudio. You can work interactively (i.e. enter R commands here), but for the most part we will run a script (or lines in a script) in the source pane and watch their execution and output here. The "Terminal" tab give you access to the BASH terminal (the Linux operating system, unrelated to R). RStudio also allows you to run jobs (analyses) in the background. This is useful if some analysis will take a while to run. You can see the status of those jobs in the background.
> **Environment/History**: Here, RStudio will show you what datasets and objects (variables) you have created and which are defined in memory. You can also see some properties of objects/datasets such as their type and dimensions. The "History" tab contains a history of the R commands you've executed.
> **Files/Plots/Packages/Help/Viewer**: This multi-purpose pane will show you the contents of directories on your computer. You can also use the "Files" tab to navigate and set the working directory. The "Plots" tab will show the output of any plots generated. In "Packages" you will see what packages are actively loaded, or you can attach installed packages. "Help" will display help files for R functions and packages. "Viewer" will allow you to view local web content (e.g. HTML outputs).
> ---datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/00-introduction.html)*

Look under the files tab

You can already see our R project and R script file in our project directory under the `Files` tab. If you chose to use `renv` you will also see some files and directories related to that.

Additional panes may show up depending on what you are doing in RStudio. For example, you may notice a `Render` tab in the Console/Terminal/Jobs pane when working with Rmarkdown (.Rmd) or Quarto (.qmd) files.

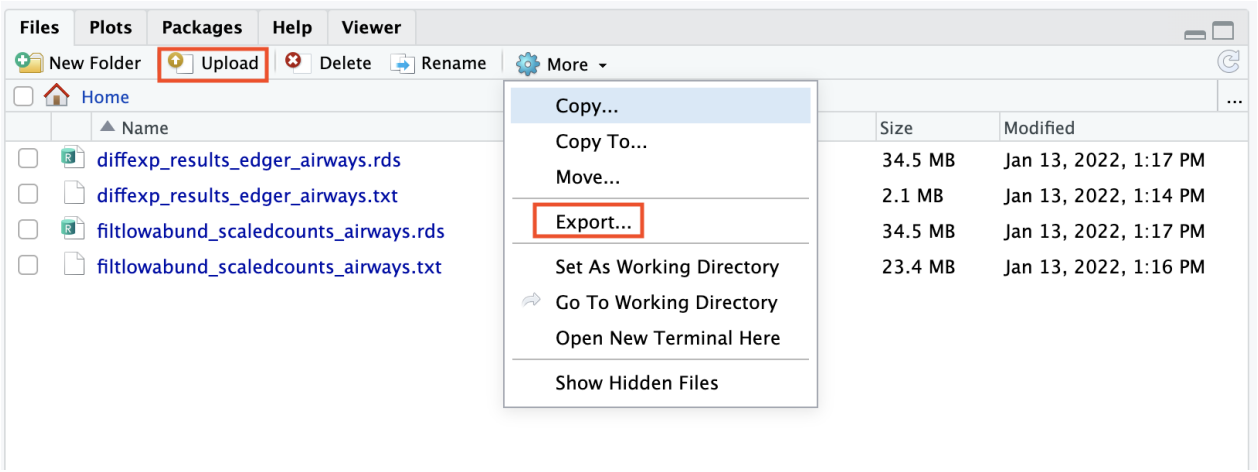
Also, you can change your RStudio layout. See this blog *(https://www.r-bloggers.com/2018/05/a-perfect-rstudio-layout/)* if you are interested. **For simplicity, please do NOT change the layout during this course.**

When to use `Source` vs `Console`?

We will use the `Source` pane to keep a record of the code that we run. However, at times, we may want to do quick testing without keeping a record. This is the scenario in which you would use the `Console`.

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 files to your local computer.



Data Management

Data organization is extremely important to reproducible science. Consider organizing your project directory in a way that facilitates reproducibility. All inputs and outputs (where possible) should be contained within the project directory, and a consistent directory structure should be created. For example, you may want directories for data, docs, outputs, figures, and scripts. See additional details here *(https://bioinformatics.ccr.cancer.gov/docs/reproducible-r-on-biowulf/L3_PackageManagement/)*. How you organize project directories is up to you, but consistency is fairly important for reproducibility. We will discuss more on this subject when introducing data frames.

> **Use relative file paths**
>
> Do not use absolute file paths in scripts. These will cause the script to fail unexpectedly for other users.

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()`. RData files are often not visible in a directory. You can see them using `ls -a` from the terminal. RData files within a working directory associated with a given project will launch automatically under the default option **Restore .RData into workspace at startup**. You may also load .Rdata by using `load()`.

Restoring your R environment

If you are working with significantly large datasets, you may not want to automatically save and restore .RData. To turn this off, go to Tools -> Global Options -> deselect "Restore .RData into workspace at startup" and choose "Never" for "Save workspace to .RData on exit". It is usually recommended not to restore the `.RData` file *(https://r4ds.hadley.nz/workflow-scripts.html#what-is-the-source-of-truth)* at the beginning of a session.

Another file to be aware of is the `.Rhistory` file. The R history file contains a list of commands from your previous R sessions.

What is a function?

Now we are ready to work with some of our first R commands. In R, commands are generally called 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.github.io/genomics-r-intro/00-introduction.html#using-functions-in-r-without-needing-to-master-them)*.

There are thousands of available functions to use in R, and if there isn't a function available for a specific task, you can write your own. **We will be using many more functions, so there will be many more opportunities to learn the syntax.**

We are going to run commands directly from our R script rather than typing into the R console.

Our first function will be `getwd()`. This simply prints your working directory and is the R equivalent of `pwd` (if you know Unix coding).

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

To run this function, 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 + return` 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.

```
[1] "/vf/users/emmonsal/Getting_Started_with_R"
```

It is good practice to annotate your code using a comment. We can denote comments with #.

We designated or set our working directory when we created our R project, but if for some reason we needed to set our working directory, we can do this with `setwd()`. There is no need to run currently. However, if you were to run it, you would use the following notation:

```
setwd("path_to_your_directory")
```

The path should be in quotes. You can use tab completion to fill in the path.

What is a path?

According to Wikipedia, a path is "a string of characters used to uniquely identify a location in a directory structure."

Therefore, a file path simply tells us where a file or files are located. You will need to direct R to the location of files that you want to work with or output that you create.

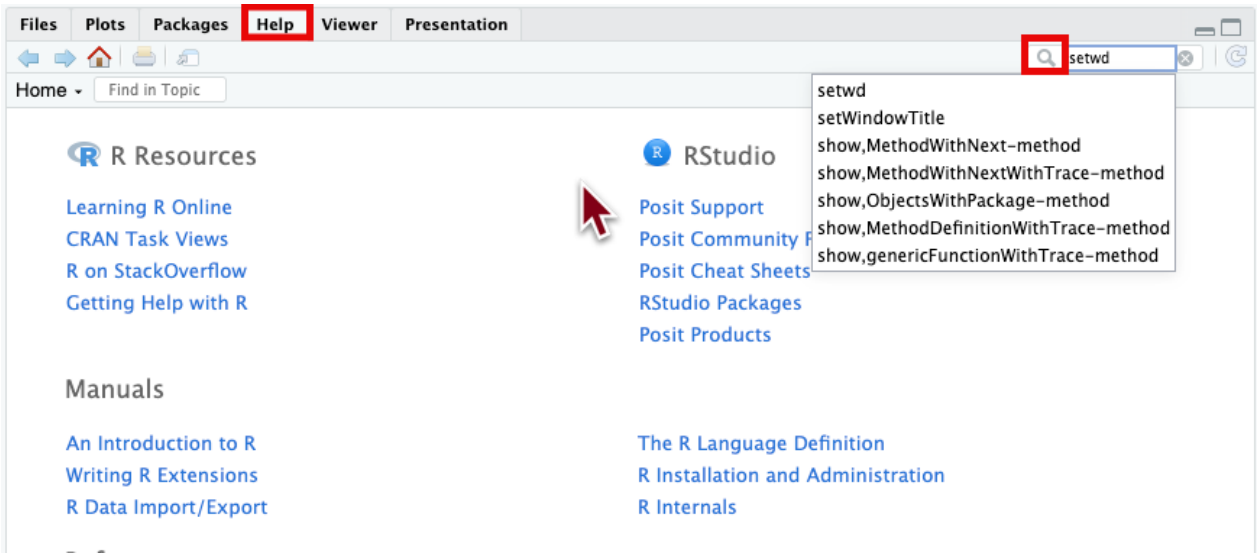
The working directory is the location in your file system that you are currently working in. In other words, it is the default location that R will look for input files and write output files.

> **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. You can use tab completion to help you fill in directory information.

Getting help

Now we know a bit about using functions, but what if I had no idea what the function `setwd()` was used for or what arguments to provide?

Getting help in R is fairly easy. In the pane to the bottom right, you should see a `Help` tab. You can search for help regarding a specific topic using the search field (look for the magnifying glass).



Alternatively, you can search directly for help in the console using `?setwd()` or `??setwd()`. `help.search()` or `??` can be used to search for a function using a keyword and will also work for unloaded packages; for example, you may try `help.search("anova")`.

R help pages provide a lot of information. The description and argument sections are likely where you will want to start. If you are still unsure how to use the function, scroll down and check out the examples section of the documentation. Consider testing some of the examples yourself and applying to your own data.

Many R packages also include more detailed help documentation known as a vignette. To see a package vignette, use `browseVignettes()` (e.g., `browseVignettes(package="dplyr")`).

To see a function's arguments, you can use `args()`.

```
args(setwd)
```

```
function (dir)
NULL
```

Because `setwd(dir)` is used to set the working directory to `dir`, it requires only a single argument (`dir`).

> **Note**
>
> R arguments can be specified by name with `argument_name= ____", by position, or by partial name. More on this later.

Additional Sources for help

Try googling your problem or using some other search engine. `rseek` *(https://rseek.org/)* is an R specific search engine that searches several R related sites. If using Google or other major search engine directly, make sure you use R to tag your search.

Stack Overflow is a particularly great resource for finding help. If you post a question, you will need to make a reproducible example (reprex) and be as descriptive as possible regarding the problem. For this purpose, you may find the `reprex` *(https://reprex.tidyverse.org/)* package particularly useful.

To provide details about your R session, use

```
sessionInfo()
```

```
R version 4.5.0 (2025-04-11)
Platform: aarch64-apple-darwin20
Running under: macOS Sequoia 15.4

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/
LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: Europe/Berlin
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.5.0    fastmap_1.2.0     cli_3.6.4         tools_4.5
 [5] htmltools_0.5.8.1 rstudioapi_0.17.1 yaml_2.3.10       rmarkdown_
 [9] knitr_1.50        jsonlite_2.0.0    xfun_0.52         digest_0.6
[13] rlang_1.1.6       evaluate_1.0.3
```

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-introduction/index.html)*. Material was also inspired by content from Introduction to data analysis with R and Bioconductor *(https://carpentries-incubator.github.io/bioc-intro/)*, which is part of the

Carpentries Incubator *(https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator)*.

Lesson 2: Basics of R Programming: R Objects and Data Types

Objectives

To understand some of the most basic features of the R language including:

- Creating and manipulating R objects.
- Understanding object types and classes.
- Using mathematical operations.

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here.

R objects

Objects (and functions) are key to understanding and using R programming.

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 objects. 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.

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".

> **Note**
>
> You can use `alt + -` on a PC to generate the `->` or `option + -` on a mac.

> **Using = for assignment?**

To improve the readability of your code, you should use the -> operator to assign values to objects rather than =. = has other purposes. For example, setting function arguments.

Let's create a simple object and run our code. There are a few methods to run code:

- The run button
- Key shortcuts (Windows: `ctrl+Enter`, Mac: `Command+Return`)
- Type directly into the console.

Use comments (#) to annotate your code for better reproducibility.

```
#Create an object called "a" assigned to a value of 1.
a <- 1

#Simply call the name of the object to print the value to the screen
a
```

```
[1] 1
```

In this example, "a" is the name of the object, 1 is the value, and <- is the assignment operator.

Now, if we use a in our code, R will replace it with its value during execution. Try the following:

```
a + 5
```

```
[1] 6
```

```
5 - a
```

```
[1] 4
```

```
a^2
```

```
[1] 1
```

```
a + a
```

```
[1] 2
```

Naming conventions and reproducibility

There are rules regarding the naming of objects.

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

   For example:

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

```
Error in parse(text = input): <text>:1:2: unexpected symbol
1: 1a
    ^
```

> **Note**
>
> It is generally a good habit to not begin sample names with a number.

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?

> **Strings**
>
> R recognizes different types of data (See below). We have used numbers above, but we can also use characters or strings. A string is a sequence of characters. It can contain letters, numbers, symbols and spaces, but to be recognized as a string it must be wrapped in quotes (either single or double). If a sequence of characters are not wrapped in quotes, R will try to interpret it as something other than a string like an R object.

3. Avoid common names with special meanings (See `?Reserved`) 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.

Object names should be short but informative. If you use `a`, `b`, `c`, you will likely forget what those object names represent. However, something like `This_is_my_scientific_data_from_blah_experiment` is far too long. Strike a nice balance.

Reassigning objects

To reassign an object, simply overwrite the object.

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

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

```
#Re-assign gene_name to a different gene
gene_name<-"GH1"
gene_name
```

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

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

Deleting objects

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

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

```
Error: object 'gene_name' not found
```

Object data types

> Data types are familiar in many programming languages, but also in natural language where we refer to them as the parts of speech, e.g. nouns, verbs, adverbs, etc. Once you know if a word - perhaps an unfamiliar one - is a noun, you can probably guess you can count it and make it plural if there is more than one (e.g. 1 Tuatara, or 2 Tuataras). If something is a adjective, you can usually change it into an adverb by adding "-ly" (e.g. jejune vs. jejunely). Depending on the context, you may need to decide if a word is in one category or another (e.g "cut" may be a noun when it's on your finger, or a verb when you are preparing vegetables). These concepts have important analogies when working with R objects.
> --- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html)*

The type and class of an R object affects how that object can be used or will behave. Examples of base R data types include **double**, **integer**, **complex**, **character**, and **logical**.

R objects can also have certain assigned attributes like class (e.g., data frame, factor, date), and these attributes will be important for how they interact with certain methods / functions. Ultimately, understanding the type and class of an object will be important for how an object can be used in R. When the type (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.

The type of an object can be examined using `typeof()`, while the class of an object can be viewed using `class()`. `typeof()` returns the storage mode of any object. Here, I am using `mode` and `type` interchangeably but they do differ. To find out more check out the help docs: `?mode()` or `?typeof`.

We now know what data types are, but what is a class?

> 'class' is a property assigned to an object that determines how generic functions operate with it. It is not a mutually exclusive classification. If an object has no specific class assigned to it, such as a simple numeric vector, it's class is usually the same as its mode, by convention. ---stackexchange *(https://stats.stackexchange.com/questions/3212/mode-class-and-type-of-r-objects#:~:text=class%20is%20an%20attribute%20of,physical%20characteristic%20of%20an%20obje*

It is often most useful to use `class()` and `typeof()` to find out more about an object or `str()` (more on this function later).

Let's create some objects and determine their types and classes.

```
chromosome_name <- 'chr02'
typeof(chromosome_name)
## [1] "character"
class(chromosome_name)
## [1] "character"

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

df<-head(iris)
typeof(df)
## [1] "list"
class(df)
## [1] "data.frame"


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

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

There are also functions that can gauge types directly, for example, `is.numeric()`, `is.character()`, `is.logical()`. And, there are functions for explicit coercion from one type to another: `as.double()`, `as.integer()`, `as.factor()`, `as.character()`, etc.

If an object has a class attribute, there is likely an associated "constructor function", or function used to build an object of that class. For example, `?data.frame()`, `?factor()`. We will discuss both data frames and factors in a later lesson.

Special null-able values

There are also special use, null-able values that you should be aware of. Read more to learn about NULL, NA, NaN, and Inf *(https://www.r-bloggers.com/2018/07/r-null-values-null-na-nan-inf/)*.

Mathematical operations

As mentioned, an object's type/mode can be used to understand the methods that can be applied to it. Objects of mode numeric can be treated as such, meaning mathematical operators can be used directly with those objects.

This chart from datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html)* shows many of the 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)

`()` are additionally used to establish the order of operations.

Let's see this in practice.

```
#create an object storing the number of human chromosomes (haploid)
human_chr_number<-23
#let's check the type of this object
typeof(human_chr_number)
```

```
[1] "double"
```

```
#Now, lets get the total number of human chromosomes (diploid)
human_chr_number * 2 #The output is 46!
```

```
[1] 46
```

Moreover, we do not need an object to perform mathematical computations. R can be used like a calculator.

For example,

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

```
[1] 1.618034
```

A function is an object.

R functions are saved as objects, and if we type the name of the function, we can see the value of the object (i.e., the code underlying the function). Functions are important to R programming, as anything that happens in R is due to the use of a function.

Looking up Compiled Code

> When looking at R source code, sometimes calls to one of the following functions show up: .C(), .Call(), .Fortran(), .External(), or .Internal() and .Primitive(). These functions are calling entry points in compiled code such as shared objects, static libraries or dynamic link libraries. Therefore, it is necessary to look into the sources of the compiled code, if complete understanding of the code is required. --- RNews 2006 *(https://cran.r-project.org/doc/Rnews/Rnews_2006-4.pdf)*

We have used some R functions in Lesson 1 (e.g. `getwd()` and `setwd()`)! Let's look at another example using the `round()` function.
`round()` "rounds the values in its first argument to the specified number of decimal places (default 0)" --- R help.

Consider

```
round(5.65) #can provide a single number
```

```
[1] 6
```

```
round(c(5.65,7.68,8.23)) #can provide a vector
```

```

```

```
[1] 6 8 8
```

In this example, we only provided the required argument in this case, which was any numeric or complex vector. We can see that two arguments can be included by the context prompt while typing (See below image). The optional second argument (i.e., digits) indicates the number of decimal places to round to. Contextual help is generally provided as you type the name of a function in RStudio.

```
#provide an additional argument rounding to the tenths place
round(5.65,digits=1)
```

```
[1] 5.7
```

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()`.

The pipe (|>, %>%).

Functions can be chained together using a pipe (`|>`, `%>%`). The pipe improves the readability of the code by minimizing nesting.

For example,

```
ex<- -5.679

ex |> round() |> abs()
```

```
[1] 6
```

We will talk about the pipe more in part 2 and 3 of this series. For now, it is helpful to know that it exists and what it is doing.

> **Differences between `|>` and `%>%`**
>
> There are some crucial differences between the native pipe `|>` and the `maggitr` pipe (`%>%`). Check out this blog *(https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/)* for details.

Pre-defined objects

Base R comes with a number of built-in functions, vectors, data frames, and other objects. You can view all using the function, `builtins()`. If you are interested in built-in datasets, check out `help(package="datasets")`.

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-introduction/index.html)*.

Lesson 3: Basics of R Programming: Vectors

Objectives

To understand some of the most basic features of the R language including creating, modifying, sub-setting, and exporting vectors.

As with previous lessons, to get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here.

Vectors

> Vectors are probably the most 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 are vectors. One of the most common ways to create a vector is to use the `c()` function - the "concatenate" or "combine" function. Inside the function you may enter one or more values; for multiple values, separate each value with a comma. --- datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/01-r-basics.html)*.

Creating vectors

```
#create a vector of gene names
transcript_names <- c("TSPAN6", "TNMD", "SCYL3", "GCLC")
transcript_names
```

```
[1] "TSPAN6" "TNMD"   "SCYL3"  "GCLC"
```

Let's check out the type of data within the vector. What do you think?

```
typeof(transcript_names)
```

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

Another property of vectors worth exploring is their length. Try `length()`

```
length(transcript_names)
```

```
[1] 4
```

In addition, you can assess the underlying structure of the object (vector in this case) by using `str()`. `str()` will be invaluable for understanding more complicated data structures such as matrices and data frames, which will be discussed later.

```
# this will return properties of the object's underlying structure
# in this case, the length and type
str(transcript_names)
```

```
 chr [1:4] "TSPAN6" "TNMD" "SCYL3" "GCLC"
```

Here, the length and type of data in the vector are returned, as well as a summary of the data.

```
#We know this is a vector from the length but you could always check
is.vector(transcript_names)
```

```
[1] TRUE
```

Vectors can also have a names attribute.

```
counts<-c("TSPAN6"= 679, "TNMD" = 0, "SCYL3" = 467)
counts
```

```
TSPAN6   TNMD  SCYL3
   679      0    467
```

```
names(counts)
```

```
[1] "TSPAN6" "TNMD"   "SCYL3"
```

Creating, modifying, sub-setting exporting

Let's learn how to further work with vectors, including creating, sub-setting, modifying, and saving. First, we will create a few vectors. Again, the `c()` vector is necessary for this task.

```
#Some possible RNASeq data
cell_line<- c("N052611", "N061011", "N080611", "N61311" )
sample_id <- c("SRR1039508", "SRR1039509", "SRR1039512",
               "SRR1039513", "SRR1039516", "SRR1039517",
               "SRR1039520", "SRR1039521")
transcript_counts <- c(679, 0, 467, 260,  60,   0)
```

Creating vectors with functions

Vectors can also be created with different functions. Some common functions used to create vectors include `seq()` and `rep()`.

Vector operations

If our vectors are numeric, we can apply mathematic operations and arithmetic expressions.

```
# Apply some basic math
transcript_counts + 10
```

```
[1] 689  10 477 270  70  10
```

```
transcript_counts^2 +100
```

```
[1] 461141    100 218189  67700   3700    100
```

```
# Transform the data using a log 10 transformation
log10(transcript_counts + 1)
```

```
[1] 2.832509 0.000000 2.670246 2.416641 1.785330 0.000000
```

```
# Add two vectors together
transcript_counts + rep(2,times=6)
## [1] 681   2 469 262  62   2

#Add different sized vectors
transcript_counts + c(0,1)
## [1] 679   1 467 261  60   1
```

```
transcript_counts + c(0,1,0,1)
## Warning in transcript_counts + c(0, 1, 0, 1): longer object length is not a
## multiple of shorter object length
## [1] 679   1 467 261  60   1
```

Some things to note here:

1. With vectors of the same length, we can add, subtract, multiply, etc., but operations are performed on elements in the same position of each vector.
2. With vectors of different lengths, the shorter vector will be recyled *(https://www.geeksforgeeks.org/vector-recycling-in-r/)* until the operation is complete. If the larger vector is not a multiple of the shorter vector, a warning will be thrown.

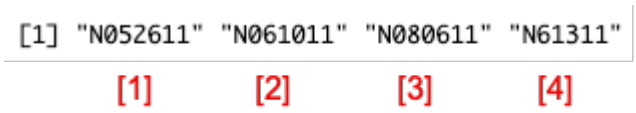
Vector sub-setting

There may be moments where you want to retrieve a specific value or values from a vector. To do this, we use bracket notation sub-setting (`[]`).In bracket notation, you call the name of the vector followed by brackets. The brackets contain an index for the value that we want. The index is the numerical position of the value in the vector. For example, take a look at `cell_line`.

```
cell_line
```

```
[1] "N052611" "N061011" "N080611" "N61311"
```

The first position `[1]` is held by "N052611". The next position is 2 followed by 3, etc.



Index positions in `cell_line`.

With numerical indexing, we can access a given value from the vector using `name[index]`, where `name` is the name of the vector, and `index` is the numerical position within the vector.

Let's get the second value from `cell_types`.

```
cell_line[2]
```

```
[1] "N061011"
```

In R vector indices start with 1 and end with length(vector). This is important and can differ based on programming language.

For example:

> Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.---bioc-intro *(https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r.html)*.

So to extract the last element in a vector, you could use the following annotation:

```
#retrieve the last element in the sample_id vector
sample_id[length(sample_id)]
```

```
[1] "SRR1039521"
```

This is the same as:

```
#retrieve the last element in the sample_id vector
sample_id[8]
```

```
[1] "SRR1039521"
```

You may also want to subset a range of values. In R, use a colon (`:`) to represent a range.

```
#Retrieve the 2nd and 3rd value from cell_line
cell_line[2:3]
```

```
[1] "N061011" "N080611"
```

```
#Retrieve the 1st, 4th, 5th, and 6th values from transcript_counts
transcript_counts[c(1,4:6)]
```

```
[1] 679 260  60   0
```

The combine function `c()` can also be used to add 1 or more elements to a vector. To be overwritten the object has to be reassigned.

```
#Lets add two genes to transcript_names
transcript_names <- c(transcript_names, "ANAPC10P1", "ABCD1")
transcript_names
## [1] "TSPAN6"    "TNMD"      "SCYL3"     "GCLC"      "ANAPC10P1" "A
```

Subtraction can be used to remove a value.

```
#Let's remove "SCYL3"
transcript_names <- transcript_names[-3]
transcript_names
```

```
[1] "TSPAN6"    "TNMD"      "GCLC"      "ANAPC10P1" "ABCD1"
```

We can rename a value by

```
#Let's rename "GCLC"
transcript_names[3] <- "NNAME"
transcript_names
```

```
[1] "TSPAN6"    "TNMD"      "NNAME"     "ANAPC10P1" "ABCD1"
```

We can use the names attribute to query or subset a vector.

```
counts["SCYL3"]
```

```
SCYL3
  467
```

We can also call a value directly; More on this below.

```
#Rename "ABCD1" to "NEW"
transcript_names[transcript_names == "ABCD1"]  <- "NEW"
transcript_names
```

```
[1] "TSPAN6"    "TNMD"      "NNAME"     "ANAPC10P1" "NEW"
```

Logical subsetting

It is also possible to subset in R using logical evaluation or numerical comparison. To do this, we use comparison operators, as we did in the last example. See the table below for a list of operators.

Comparison Operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	Not equal
==	equal
a | b	a or b
a & b	a and b

So if, for example, we wanted a subset of all transcript counts greater than 260, we could use indexing combined with a comparison operator:

```
transcript_counts[transcript_counts > 260]
```

```
[1] 679 467
```

Why does this work? Let's break down the code.

```
transcript_counts > 260
```

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

This returns a logical vector. We can see that positions 1 and 3 are TRUE, meaning they are greater than 260. Therefore, the initial sub-setting above is asking for a subset based on TRUE values. Here is the equivalent:

```
transcript_counts[c( TRUE, FALSE, TRUE, FALSE, FALSE, FALSE)]
```

```
[1] 679 467
```

You can also use this functionality to do a kind of find and replace. Perhaps we want to find zero values and replace them with NAs. We could use:

```
transcript_counts[transcript_counts==0]<-NA
```

> **Note**
>
> if you instead ran `transcript_counts[transcript_counts==0]<-"NA"`, you would coerce this vector to a character vector.

Now, if we want to return only values that aren't NAs, we can use

```
transcript_counts[!is.na(transcript_counts)] #values that aren't NAs
```

```
[1] 679 467 260  60
```

```
is.na(transcript_counts) #if you simply want to know if there are NAs
```

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

```
which(is.na(transcript_counts)) #if you want the indices of those NAs
```

```
[1] 2 6
```

Other ways to handle missing data

Other functions you may find useful when working with NAs inclue `na.omit()` and `complete.cases()`.

`na.omit()` removes the NAs from a vector.

```
na.omit(transcript_counts)
```

```
[1] 679 467 260  60
attr(,"na.action")
[1] 2 6
attr(,"class")
[1] "omit"
```

`complete.cases()` creates a logical vector that you can use for subs-etting based on the absence of NAs.

```
transcript_counts[complete.cases(transcript_counts)]
```

```
[1] 679 467 260  60
```

Many functions will also have an `na.rm` argument. For example, see ?mean.

Using objects to store thresholds

To make scripting reproducible, you could avoid calling a specific number directly and use objects in logical evaluations like those above. If we use an object, the value itself could easily be replaced with whatever value is needed. For example:

```
trnsc_cutoff <- 260
#note: this will also include NAs in the output
transcript_counts[transcript_counts>trnsc_cutoff]
```

```
[1] 679  NA 467  NA
```

```
#if we want to exclude possible NAs, something like this will work
transcript_counts[!is.na(transcript_counts) & transcript_counts>trnsc
```

```
[1] 679 467
```

Using the `%in%` operator.

There may be a time you want to know whether there are specific values in your vector. To do this, we can use the `%in%` operator (`?match()`). This operator returns TRUE for any value that is in your vector and can be used for sub-setting. It makes more sense to use this with data frames but we can see how this works here.

For example:

```
# have a look at transcript_names
transcript_names
```

```
[1] "TSPAN6"    "TNMD"      "NNAME"     "ANAPC10P1" "NEW"
```

```
# test to see if "NNAME" and "ANAPC10P1" are in this vector
# if you are looking for more than one value, you must pass this as a

c("NNAME","ANAPC10P1") %in% transcript_names
```

```
[1] TRUE TRUE
```

```
#We could also save the search vector to an object and search that wa
find_transcripts<-c("NNAME","ANAPC10P1")
find_transcripts %in% transcript_names
```

```
[1] TRUE TRUE
```

```
#to use this for subetting the vector lengths should match
transcript_names[transcript_names %in% find_transcripts]
```

```
[1] "NNAME"     "ANAPC10P1"
```

Saving and loading objects

We discussed saving the R workspace (.RData), but what if we simply want to save a single object. In such a case, we can use `saveRDS()`.

Let's save our transcript_counts vector to our working directory.

```
saveRDS(transcript_counts,"transcript_counts.rds")
```

Check the `Files` pane for your newly created file. Make sure you are viewing the contents of your working directory (`getwd()`).

To load the object back into your R workspace, use `readRDS()`.

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-introduction/index.html)*. Material was also inspired by content from Introduction to data analysis with R and Bioconductor *(https://carpentries-incubator.github.io/bioc-intro/)*, which is part of the Carpentries Incubator *(https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator)*.

Lesson 4: Introduction to R Data Structures - Data Import

Learning Objectives

1. Learn about data structures including factors, lists, matrices, and data frames.
2. Learn how to import data in a tabular format (data frames)
3. Learn to write out (export) data from the R environment

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here.

Installing and Loading Packages

In this lesson, we will learn how to import data with different file extensions, including Excel files. We will make use of Base R functions for data import as well as popular functions from `readr` *(https://readr.tidyverse.org/)* and `readxl` *(https://readxl.tidyverse.org/)*.

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")`).

Packages are installed into your file system at a given location denoted by `.libPaths()`. This is your **R library**, a directory of installed R packages. To use one or more packages, you have to load it within your R session. **This has to be done with each new R session.**

Key functions:

- `install.packages()` install packages from CRAN.
- `library()` load packages in R session.

Load the libraries:

```
library(readxl)
library(readr)
```

> **Tip**
>
> It is good practice to load libraries needed for a script at the beginning of the script.

Data Structures

Data structures are objects that store data.

Previously, we learned that **vectors** are collections of values of the same type *(https://datacarpentry.org/genomics-r-intro/01-r-basics.html#vectors)*. A vector is also one of the most basic data structures.

Other common data structures in R include:

- **factors**
- **lists**
- **data frames**
- **matrices**

What are factors?

Factors are an important data structure in statistical computing. They are specialized vectors (ordered or unordered) for the storage of categorical data (data with fixed values). While they appear to be character vectors, data in factors are stored as integers. These integers are associated with pre-defined levels, which represent the different groups or categories in the vector.

Reference level

Generally for statistical models, the reference or control level is set to level 1. You can reorder the levels using `factor()` or `forcats::relevel()`.

Important functions

- `factor()` - to create a factor and reorder levels
- `as.factor()` - to coerce to a factor
- `levels()` - view and / or rename the levels of a factor
- `nlevels()` - return the number of levels

For example:

```
sex <- factor(c("M","F","F","M","M","M"))
levels(sex)
```

```
[1] "F" "M"
```

Check out the package `forcats` *(https://forcats.tidyverse.org/)* for managing and reordering factors.

Note

R will organize factor levels alphabetically by default. This will be especially noticeable when plotting.

Warning

Pay attention when coercing from a factor to a numeric. To do this, you should first convert to a character vector. Otherwise, the numbers that you want to be numeric (the factor level names) will be returned as integers.

See more about working with factors here *(https://r4ds.had.co.nz/factors.html#factors)*.

Lists

Unlike an atomic vector, a list can contain multiple elements of different types, (e.g., character vector, numeric vector, list, data frame, matrix). Lists are not the focus of this lesson, but you should be aware of them, as you will likely come across them at some point, as many functions, including those specific to bioinformatics, may output data in the form of a list.

Important functions

- `list()` - create a list

- `names()` - create named elements (Also useful for vectors)
- `lapply()`, `sapply()` - for looping over elements of the list

Example

```
#Create a list
My_exp <- list(c("N052611", "N061011", "N080611", "N61311" ),
               c("SRR1039508", "SRR1039509", "SRR1039512",
                 "SRR1039513", "SRR1039516", "SRR1039517",
                 "SRR1039520", "SRR1039521"),c(100,200,300,400))

#Look at the structure
str(My_exp)
```

```
List of 3
 $ : chr [1:4] "N052611" "N061011" "N080611" "N61311"
 $ : chr [1:8] "SRR1039508" "SRR1039509" "SRR1039512" "SRR1039513" .
 $ : num [1:4] 100 200 300 400
```

```
#Name the elements of the list
names(My_exp)<-c("cell_lines","sample_id","counts")
#See how the structure changes
str(My_exp)
```

```
List of 3
 $ cell_lines: chr [1:4] "N052611" "N061011" "N080611" "N61311"
 $ sample_id : chr [1:8] "SRR1039508" "SRR1039509" "SRR1039512" "SRR1
 $ counts    : num [1:4] 100 200 300 400
```

```
#Subset the list
My_exp[[1]][2]
```

```
[1] "N061011"
```

```
My_exp$cell_lines[2]
```

```
[1] "N061011"
```

```
#Apply a function (remove the first index from each vector)
lapply(My_exp,function(x){x[-1]})
```

```
$cell_lines
[1] "N061011" "N080611" "N61311"

$sample_id
[1] "SRR1039509" "SRR1039512" "SRR1039513" "SRR1039516" "SRR1039517"
[6] "SRR1039520" "SRR1039521"

$counts
[1] 200 300 400
```

We are not going to spend a lot of time on lists, but you should consider learning more about them in the future, as you may receive output at some point in the form of a list. For a brief introduction to lists, see the following resources:

- R4DS *(https://r4ds.had.co.nz/vectors.html#lists)*
- UVA list tutorial *(https://bioconnector.github.io/workshops/r-lists.html)*
- Steve's Data Tips and Tricks *(https://www.spsanderson.com/steveondata/posts/2024-10-29/)*

Data Matrices

Another important data structure in R is the data matrix. Data frames and data matrices are similar in that both are tabular in nature and are defined by dimensions (i.e., rows (m) and columns (n), commonly denoted m x n). However, a matrix contains only values of a single type (i.e., numeric, character, logical, etc.).

> **Note**
>
> A vector can be viewed as a 1 dimensional matrix.

Elements in a matrix and a data frame can be referenced by using their row and column indices (for example, a[1,1] references the element in row 1 and column 1).

Below, we create the object a1, a 3-row by 4-column matrix.

```
a1 <- matrix(c(3,4,2,4,6,3,8,1,7,5,3,2), ncol=4)
a1
```

```
     [,1] [,2] [,3] [,4]
[1,]    3    4    8    5
[2,]    4    6    1    3
[3,]    2    3    7    2
```

Using the typeof() and class() command, we see that the elements in a1 are double and a1 a matrix, respectively.

```
typeof(a1)
```

```
[1] "double"
```

```
class(a1)
```

```
[1] "matrix" "array"
```

Similar to lists, we aren't going to focus much on matrices.

Data Frames: Working with Tabular Data

In genomics, we work with a lot of tabular data - data organized in rows and columns. The data structure that stores this type of data is a **data frame**. Data frames are collections of vectors of the same length but can be of different types. Because we often have data of multiple types, it is natural to examine that data in a data frame.

You may be tempted to open and manually work with these data in excel. However, there are a number of reasons why this can be to your detriment. First, it is very easy to make mistakes when working with large amounts of tabular data in excel. Have you ever mistakenly left out a column or row while sorting data? Second, many of the files that we work with are so large (big data) that excel and your local machine do not have the bandwidth to handle them. Third, you will likely need to apply analyses that are unavailable in excel. Lastly, it is difficult to keep track of any data manipulation steps or analyses in a point and click environment like excel.

R, on the other hand, can make analyzing tabular data more efficient and reproducible. But before getting into working with this data in R, let's review some best practices for data management.

Best Practices for organizing genomic data

1. "Keep raw data separate from analyzed data" -- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes.html)*

For large genomic data sets, you may want to include a project folder with two main subdirectories (i.e., raw_data and data_analysis). You may even consider changing the permissions (check out the unix command `chmod` *(https://www.howtogeek.com/437958/how-to-use-the-chmod-command-on-linux/)*) in your raw directory to make those files *read only*. Keeping raw data separate is not a problem in R, as one must explicitly import and export data.

1. "Keep spreadsheet data Tidy" -- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes.html)*

Data organization can be frustrating, and many scientists devote a great deal of time and energy toward this task. Keeping data tidy, can make data science more efficient, effective, and reproducible. There is a collection of packages in R that embrace the philosophy of tidy data and facilitate working with data frames. That collection is known as the tidyverse *(https://www.tidyverse.org/)*.

1. "Trust but verify" -- datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes.html)*

R makes data analysis more reproducible and can eliminate some mistakes from human error. However, you should approach data analysis with a plan, and make sure you understand what a function is doing before applying it to your data. Often using small subsets of data can be used as a form of data debugging to make sure the expected result materialized.

Some functions for creating practice data include: `data.frame()`, `rep()`, `seq()`, `rnorm()`, `sample()` and others. See some examples here *(https://ademos.people.uic.edu/Chapter7.html#32_b_using_the_rep_function_to_create_data_frames)*.

Let's use some of these to create a data frame.

```
df<-data.frame(Samples=c(1:10),Counts=sample(1:5000, size=10, replace
df
```

```
   Samples Counts Treatment
1        1   4939   control
2        2    191   control
3        3   3697   control
4        4   4933   control
5        5   2938   control
6        6   1721   treated
7        7    214   treated
```

```
8         8   2999   treated
9         9   2084   treated
10       10   2196   treated
```

Example Data

There are data sets available in R to practice with or showcase different packages; for example, `library(help = "datasets")`. For the next two lessons, we will use data derived from the Bioconductor package `airway` *(https://bioconductor.org/packages/release/data/experiment/html/airway.html)* as well as data internal to or derived from Base R and packages within the tidyverse. Check out the Acknowledgements section for additional data sources.

Obtaining the data

- To download the data used in this lesson to your local computer, click here. You can then move the downloaded directory to your working directory in R.
- To use the data on Biowulf, open your Terminal in R and follow these steps:

```
cd /data/$USER/Getting_Started_with_R
wget https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting
unzip data.zip
```

Note

"Getting_Started_with_R" is the name of the project directory I created in Lesson 1. If you do not have this directory, make sure you change directories to your working directory in R.

Importing 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.

Pay close attention to the import functions and their arguments. Using the import arguments correctly can save you 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.

Row names

`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).

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

> **Compared to the corresponding base functions, readr functions:**
>
> Use a consistent naming scheme for the parameters (e.g. col_names and col_types not header and colClasses).
>
> Are generally much faster (up to 10x-100x) depending on the dataset.
>
> Leave strings as is by default, and automatically parse common date/time formats.
>
> Have a helpful progress bar if loading is going to take a while.
>
> All functions work exactly the same way regardless of the current locale. To override the US-centric defaults, use locale(). - readr.tidyverse.org *(https://readr.tidyverse.org/#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. We loaded this above.

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/4.5-arm64/Resources/lib
```

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: 32 × 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# ℹ 22 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 Transcri
 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 these 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
```

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 RNA-Seq 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    S
1 GSM1275862  N61311 untrt untrt SRR1039508       126  SRX384345 SRS!
2 GSM1275863  N61311   trt untrt SRR1039509       126  SRX384346 SRS!
3 GSM1275866 N052611 untrt untrt SRR1039512       126  SRX384349 SRS!
4 GSM1275867 N052611   trt untrt SRR1039513        87  SRX384350 SRS!
5 GSM1275870 N080611 untrt untrt SRR1039516       120  SRX384353 SRS!
6 GSM1275871 N080611   trt untrt SRR1039517       126  SRX384354 SRS!
     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, BioS
dbl (1): avgLength

ℹ Use `spec()` to retrieve the full column specification for this da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

```
smeta2
```

```
# A tibble: 8 × 9
  SampleName cell    dex   albut Run       avgLength Experiment Samp
  <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <chr
1 GSM1275862 N61311  untrt untrt SRR10395…       126 SRX384345  SRS5(
2 GSM1275863 N61311  trt   untrt SRR10395…       126 SRX384346  SRS5(
3 GSM1275866 N052611 untrt untrt SRR10395…       126 SRX384349  SRS5(
4 GSM1275867 N052611 trt   untrt SRR10395…        87 SRX384350  SRS5(
5 GSM1275870 N080611 untrt untrt SRR10395…       120 SRX384353  SRS5(
6 GSM1275871 N080611 trt   untrt SRR10395…       126 SRX384354  SRS5(
7 GSM1275874 N061011 untrt untrt SRR10395…       101 SRX384357  SRS5(
8 GSM1275875 N061011 trt   untrt SRR10395…        98 SRX384358  SRS5(
```

What if we want to retain row names?

Let's load in a count matrix from `airway`.

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

```
                      X Accession.SRR1039508 Accession.SRR1039509
1   ENSG00000000003.TSPAN6                679                   448
2     ENSG00000000005.TNMD                  0                     0
3     ENSG00000000419.DPM1                467                   515
4    ENSG00000000457.SCYL3                260                   211
5 ENSG00000000460.C1orf112                 60                    55
6      ENSG00000000938.FGR                  0                     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. Remember, `readr` is a part of the tidyverse and does not play well with row names. Therefore, we will use `read.delim()` withe the argument `row.names`.

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.SRR1039509
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.SRR1039513
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.SRR1039517
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.SRR1039521
ENSG00000000003.TSPAN6                  770                  572
ENSG00000000005.TNMD                      0                    0
ENSG00000000419.DPM1                    417                  508
ENSG00000000457.SCYL3                   233                  229
ENSG00000000460.C1orf112                 76                   60
ENSG00000000938.FGR                       0                    0
```

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 we
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, weigh

ℹ Use `spec()` to retrieve the full column specification for this da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

```
cexamp2
```

```
# A tibble: 35,549 × 9
   record_id month   day  year plot_id species_id sex   hindfoot_leng
       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl
 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** by Carrie Wright, Shannon E. Ellis, Stephanie C. Hicks and Roger D. Peng.

Data Export.

To export data to file, you will use similar functions (`write.table()`,`write.csv()`,`saveRDS()`, etc.).

For example, let's save `df` to a csv file.

```
write_csv(df,"./data/small_df_example.csv")
```

Acknowledgements

Some material from this lesson was either taken directly or adapted from *Intro to R and RStudio for Genomics (https://datacarpentry.github.io/genomics-r-intro/03-basics-factors-dataframes.html)* provided by datacarpentry.org. Other 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 this lesson was taken from datacarpentry.org *(https://datacarpentry.org/R-ecology-lesson/index.html)*.

Lesson 5: R Data Structures - Data Frames

Learning Objectives

This is the last lesson in Part 1 of *Introductory R for Novices: Getting Started with R*. This lesson will focus exclusively on working with data frames. Attendees will learn how to examine, summarize, and access data in data frames.

Specific learning objectives include:

1. Review data import.
2. Learn how to view and summarize data in a data frame.
3. Learn how to use data accessors.
4. Learn the syntax for sub-setting a data frame.

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here.

Load the libraries

This lesson will use some functions from the `tidyverse`.

```
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyver
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.2     ✔ tibble    3.2.1
✔ lubridate 1.9.4     ✔ tidyr     1.3.1
✔ purrr     1.0.4
── Conflicts ────────────────────────────────────────── tidyverse_con
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to forc
```

Examining and summarizing data frames

All of the objects we imported in the previous lesson, were data frames. In this lesson, we will learn how to view and find out more information regarding the data stored in a data frame. Let's use the R object, `smeta` as an example.

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

```
  SampleName    cell   dex albut        Run avgLength Experiment    S
1 GSM1275862  N61311 untrt untrt SRR1039508       126  SRX384345 SRS
2 GSM1275863  N61311   trt untrt SRR1039509       126  SRX384346 SRS
3 GSM1275866 N052611 untrt untrt SRR1039512       126  SRX384349 SRS
4 GSM1275867 N052611   trt untrt SRR1039513        87  SRX384350 SRS
5 GSM1275870 N080611 untrt untrt SRR1039516       120  SRX384353 SRS
6 GSM1275871 N080611   trt untrt SRR1039517       126  SRX384354 SRS
     BioSample
1 SAMN02422669
2 SAMN02422675
3 SAMN02422678
4 SAMN02422670
5 SAMN02422682
6 SAMN02422673
```

We can view these data by clicking on the name of the object in the `Environment` pane or by using `View()`.

To understand more about the underlying structure of our data, we can use `str()` or a similar function `dplyr::glimpse`.

```
str(smeta)
```

```
'data.frame':   8 obs. of  9 variables:
 $ SampleName: chr  "GSM1275862" "GSM1275863" "GSM1275866" "GSM12758
 $ cell      : chr  "N61311" "N61311" "N052611" "N052611" ...
 $ dex       : chr  "untrt" "trt" "untrt" "trt" ...
 $ albut     : chr  "untrt" "untrt" "untrt" "untrt" ...
 $ Run       : chr  "SRR1039508" "SRR1039509" "SRR1039512" "SRR10395
 $ avgLength : int  126 126 126 87 120 126 101 98
 $ Experiment: chr  "SRX384345" "SRX384346" "SRX384349" "SRX384350"
```

```
 $ Sample    : chr  "SRS508568" "SRS508567" "SRS508571" "SRS508572"
 $ BioSample : chr  "SAMN02422669" "SAMN02422675" "SAMN02422678" "SAI
```

`str()` shows us that we are looking at a data frame object with 8 rows by 9 columns. The column names are to the far left preceded by a `$`. This is a data frame accessor, and we will see how this works later. We can also see the data types (e.g., character, integer, logical, double) after the column name. This will help us understand how we can transform and visualize the data in these columns.

We can also get an overview of summary statistics of this data frame using `summary()`.

```
summary(smeta)
```

```
 SampleName           cell              dex                albut
Length:8           Length:8           Length:8           Length:8
Class :character   Class :character   Class :character   Class :chai
Mode  :character   Mode  :character   Mode  :character   Mode  :chai



    Run             avgLength      Experiment           Sample
Length:8           Min.   : 87.0   Length:8           Length:8
Class :character   1st Qu.:100.2   Class :character   Class :charact
Mode  :character   Median :123.0   Mode  :character   Mode  :charact
                   Mean   :113.8
                   3rd Qu.:126.0
                   Max.   :126.0
 BioSample
Length:8
Class :character
Mode  :character
```

Our data frame has 9 variables, so we get 9 fields that summarize the data. The only column with numerical data is `avgLength`, for which we can see summary statistics on the min and max values as well as mean, median, and interquartile ranges.

Using `summary()` with factors

`summary()` is also useful for obtaining quick information about a categorial (factor) variable, answering how many groups and the sample size of each group.

```
smeta2 <- smeta %>% mutate(dex = as.factor(dex))
summary(smeta2)
```

```
   SampleName           cell              dex       albut
   Length:8           Length:8           trt  :4   Length:8
   Class :character   Class :character   untrt:4   Class :character
   Mode  :character   Mode  :character              Mode  :character



       Run              avgLength      Experiment           Sample
   Length:8           Min.   : 87.0   Length:8           Length:8
   Class :character   1st Qu.:100.2   Class :character   Class :character
   Mode  :character   Median :123.0   Mode  :character   Mode  :character
                      Mean   :113.8
                      3rd Qu.:126.0
                      Max.   :126.0
   BioSample
   Length:8
   Class :character
   Mode  :character
```

What is the length of our data.frame? What are the dimensions?

Other attributes we may want to know regarding our data frame include the number of columns (`ncol()`, `length()`) and the dimensions (`dim()`).

```
#length returns the number of columns
length(smeta)
```

```
[1] 9
```

```
#dimensions, returns the row and column numbers
dim(smeta)
```

```
[1] 8 9
```

Other useful functions for inspecting data frames

Size:

`nrow()` - number of rows

`ncol()` - number of columns

Content:
`head()` - returns first 6 rows by default
`tail()` - returns last 6 rows by default

Names:
`colnames()` - returns column names
`rownames()` - returns row names

Section content from "Starting with Data", *Introduction to data analysis with R and Bioconductor (https://carpentries-incubator.github.io/bioc-intro/25-starting-with-data.html)*.

Data frame coercion and accessors

Let's pretend that the sample IDs were numeric rather than of type character.

```
smeta$SampleID <- c(1:nrow(smeta))
smeta
```

```
  SampleName    cell   dex albut        Run avgLength Experiment    S
1 GSM1275862  N61311 untrt untrt SRR1039508       126  SRX384345 SRS!
2 GSM1275863  N61311   trt untrt SRR1039509       126  SRX384346 SRS!
3 GSM1275866 N052611 untrt untrt SRR1039512       126  SRX384349 SRS!
4 GSM1275867 N052611   trt untrt SRR1039513        87  SRX384350 SRS!
5 GSM1275870 N080611 untrt untrt SRR1039516       120  SRX384353 SRS!
6 GSM1275871 N080611   trt untrt SRR1039517       126  SRX384354 SRS!
7 GSM1275874 N061011 untrt untrt SRR1039520       101  SRX384357 SRS!
8 GSM1275875 N061011   trt untrt SRR1039521        98  SRX384358 SRS!
     BioSample SampleID
1 SAMN02422669        1
2 SAMN02422675        2
3 SAMN02422678        3
4 SAMN02422670        4
5 SAMN02422682        5
6 SAMN02422673        6
7 SAMN02422683        7
8 SAMN02422677        8
```

Unless stated otherwise, "SampleID" will be treated as numeric rather than as a character vector. If we intend to work with this column and treat it as an ID, we will need to convert it or coerce it to a character or factor vector.

We can access a column of our data frame using `[]`, `[[]]`, or using the `$` *(http://adv-r.had.co.nz/Subsetting.html)*. These behave slightly differently, as we will see.

Let's access "SampleID" from `smeta`.

```
#Using $
smeta$SampleID
```

```
[1] 1 2 3 4 5 6 7 8
```

```
#Using []
smeta["SampleID"]
```

```
  SampleID
1        1
2        2
3        3
4        4
5        5
6        6
7        7
8        8
```

```
#Using [[]]
smeta[["SampleID"]]
```

```
[1] 1 2 3 4 5 6 7 8
```

Notice that `$` and `[[]]` behave similarly. These return a vector, while `[]` maintains the original structure, in this case a data frame.

Let's convert the "SampleID" column from an integer to a character vector. This is known as **coercion**.

```
#We can see that sample is being treated as numeric
is.numeric(smeta$SampleID)
```

```
[1] TRUE
```

```
#let's convert it to a character vector
```

```
smeta$SampleID<-as.character(smeta$SampleID)
#check this
is.character(smeta$SampleID)
```

```
[1] TRUE
```

```
#check this
is.numeric(smeta$SampleID)
```

```
[1] FALSE
```

See other related functions (e.g., `as.factor()`,`as.numeric()`).

Be careful with data coercion. What happens if we change a character vector into a numeric?

```
#A warning is thrown and the entire column is filled with NA
head(as.numeric(smeta$Run))
```

```
Warning in head(as.numeric(smeta$Run)): NAs introduced by coercion
```

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

> Some helpful things to remember
>
> - When you explicitly coerce one data type into another (this is known as explicit coercion), be careful to check the result. Ideally, you should try to see if it's possible to avoid steps in your analysis that force you to coerce.
> - R will sometimes coerce without you asking for it. This is called (appropriately) implicit coercion. For example [if you try] to create a vector with multiple data types, R [will choose] one type through implicit coercion.
> - Check the structure (`str()`) of your data frames before working with them! ---datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/05-dplyr/index.html)*

Using `colnames()` to rename columns

`colnames()` will return a vector of column names from our data frame. We can use this vector and `[]` sub-setting to modify our column names.

For example, let's rename the column "SampleID" to "ID".

```
#Let's rename "SampleID" to "ID"
 colnames(smeta)[10] <- "ID"

#if unsure of the index of a column, you could use which()
which(colnames(smeta)=="ID")
```

```
[1] 10
```

```
#or something like this
colnames(smeta)[colnames(smeta) ==
                          "ID"] <- "SampleID"
```

Subsetting data frames with base R

The tidyverse package `dplyr` makes it easy to subset data frames with `select()`, `filter()`, and `slice()`; however, it is still worth knowing how to subset data frames using Base R brackets.

Subsetting a data frame is similar to subsetting a vector; we can use bracket notation `[]`. However, a data frame is two dimensional with both rows and columns, so we can specify either one argument or two arguments (e.g., `df[row,column]`) depending. If you provide one argument, columns will be assumed. This is because a data frame has characteristics of both a list and a matrix.

For now, let's focus on providing two arguments to subset. (Note when a data frame structure is returned)

```
smeta[2,4] #Returns the value in the 4th column and 2nd row
```

```
[1] "untrt"
```

```
smeta[2, ] #Returns a df with row two
```

```
  SampleName   cell dex albut        Run avgLength Experiment    Samp
2 GSM1275863 N61311 trt untrt SRR1039509       126  SRX384346 SRS508!
```

```
     BioSample SampleID
2 SAMN02422675        2
```

```
smeta[-1, ] #Returns a df without row 1
```

```
  SampleName    cell   dex albut        Run avgLength Experiment    S
2 GSM1275863  N61311   trt untrt SRR1039509       126  SRX384346 SRS
3 GSM1275866 N052611 untrt untrt SRR1039512       126  SRX384349 SRS
4 GSM1275867 N052611   trt untrt SRR1039513        87  SRX384350 SRS
5 GSM1275870 N080611 untrt untrt SRR1039516       120  SRX384353 SRS
6 GSM1275871 N080611   trt untrt SRR1039517       126  SRX384354 SRS
7 GSM1275874 N061011 untrt untrt SRR1039520       101  SRX384357 SRS
8 GSM1275875 N061011   trt untrt SRR1039521        98  SRX384358 SRS
     BioSample SampleID
2 SAMN02422675        2
3 SAMN02422678        3
4 SAMN02422670        4
5 SAMN02422682        5
6 SAMN02422673        6
7 SAMN02422683        7
8 SAMN02422677        8
```

```
smeta[1:4,1] #returns a vector of rows 1-4 of column 1
```

```
[1] "GSM1275862" "GSM1275863" "GSM1275866" "GSM1275867"
```

```
#call names of columns directly
smeta[1:5,c("Sample","avgLength")]
```

```
     Sample avgLength
1 SRS508568       126
2 SRS508567       126
3 SRS508571       126
4 SRS508572        87
5 SRS508575       120
```

```
#use comparison operators
smeta[smeta$SampleID == "2",]
```

```
  SampleName   cell dex albut        Run avgLength Experiment    Samp
2 GSM1275863 N61311 trt untrt SRR1039509       126  SRX384346 SRS508
     BioSample SampleID
2 SAMN02422675        2
```

Subsetting Tibbles

Tibbles behave differently than data frames using base R accessors. See here *(https://tibble.tidyverse.org/reference/subsetting.html)* for more information.

What happens when we provide a single argument?

```
#notice the difference here
smeta[,2] #returns column two
```

```
[1] "N61311"  "N61311"  "N052611" "N052611" "N080611" "N080611" "N06
[8] "N061011"
```

```
#treated similar to a matrix
#does not return a df if the output is a single column

smeta[2] #returns column two
```

```
     cell
1  N61311
2  N61311
3 N052611
4 N052611
5 N080611
6 N080611
7 N061011
8 N061011
```

```
#treated similar to a list; maintains the df structure.
```

Note

We can also use `[[]]` or `$` for selecting specific columns.

Using `%in%`

`%in%` "returns a logical vector indicating if there is a match or not for its left operand". This logical vector can then be used to filter the data frame to only matched values.

Perhaps we only want to return a data frame with the following samples: "SRR1039508", "SRR1039513", "SRR1039520".

Using == is a bit tedious.

```
smeta[smeta$Run == "SRR1039508" | smeta$Run == "SRR1039513" |
        smeta$Run == "SRR1039520",]
```

```
  SampleName    cell   dex albut        Run avgLength Experiment    S
1 GSM1275862  N61311 untrt untrt SRR1039508       126  SRX384345 SRS!
4 GSM1275867 N052611   trt untrt SRR1039513        87  SRX384350 SRS!
7 GSM1275874 N061011 untrt untrt SRR1039520       101  SRX384357 SRS!
     BioSample SampleID
1 SAMN02422669        1
4 SAMN02422670        4
7 SAMN02422683        7
```

Instead, we can create a vector of values to keep.

```
s_keep<- c("SRR1039508", "SRR1039513", "SRR1039520")
s_keep
```

```
[1] "SRR1039508" "SRR1039513" "SRR1039520"
```

We can then see where the values in our vector match values in our column `smeta$Run`.

```
smeta$Run %in% s_keep
```

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

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

We can further use this logical vector to filter our data frame by true values.

```
smeta[smeta$Run %in% s_keep, ]
```

```
  SampleName    cell   dex albut        Run avgLength Experiment    S
1 GSM1275862  N61311 untrt untrt SRR1039508       126  SRX384345 SRS!
4 GSM1275867 N052611   trt untrt SRR1039513        87  SRX384350 SRS!
7 GSM1275874 N061011 untrt untrt SRR1039520       101  SRX384357 SRS!
     BioSample SampleID
1 SAMN02422669        1
4 SAMN02422670        4
7 SAMN02422683        7
```

`%in%` can also be used with `dplyr::filter()` and `subset()`.

Tips to remember for subsetting

- Typically provide two values separated by commas: data.frame[row, column]
- In cases where you are taking a continuous range of numbers use a colon between the numbers (start:stop, inclusive)
- For a non continuous set of numbers, pass a vector using c()
- Index using the name of a column(s) by passing them as vectors using c() ---datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html)*

Info

Subsetting including simplifying vs preserving can get confusing. Here *(http://adv-r.had.co.nz/Subsetting.html)* is a great chapter - though, a bit more advanced - that may clear things up if you are confused.

Data Wrangling

Part 2 of this course will focus on Data Wrangling. Learn how to filter, modify, summarize, and reshape your data. Check the BTEP calendar *(https://bioinformatics.ccr.cancer.gov/btep/)* for updates on upcoming classes / courses.

Acknowledgements

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

Intro to Data Wrangling

Introduction to Data Wrangling

This course is the second part of a larger 3-part course designed for novices.

Topics covered herein focus on wrangling data stored in data frames or tibbles and include concepts such as reshaping, subsetting, summarizing, mutating, and joining data.

Lessons

1. June 17, 2025 - Introduction to Data Wrangling with R
2. June 24, 2025 - Introducing Tidyr for Reshaping and Formatting Data
3. July 1, 2025 - Subsetting Data with dplyr
4. July 8, 2025 - Summarizing Data with dplyr
5. July 15, 2025 - Joining and Transforming Data with dplyr

Prerequisites

This course is recommended for attendees familiar with the skills learned in Part 1: Getting Started with R.

Course materials

This course will use R on Biowulf to avoid issues with R and package installations. To use R on Biowulf, you must have a NIH HPC account.

If you do not have a NIH HPC (Biowulf) account, this course can be taken using a local R installation. However, we will not be able to troubleshoot package installation issues during class. Additionally, because we will use packages belonging to the tidyverse *(https://www.tidyverse.org/)*, you will need to install these packages using `install.packages("tidyverse")` prior to the first lesson if you are not using R on Biowulf.

Introduction to Data Wrangling

Introducing Tidyr for Reshaping and Formatting Data

Lesson Objectives

1. Briefly review how to import data
2. Data reshape with `tidyr`: `pivot_longer()`, `pivot_wider()`, `separate()`, and `unite()`

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

Load the tidyverse

We will use core packages from the tidyverse for our data wrangling needs. Data reshaping primarily involves the tidyverse package, `tidyr`, but we will use additional packages as well, such as `tibble`.

Packages must be loaded with each new R session.

```
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyver
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.2     ✔ tibble    3.3.0
✔ lubridate 1.9.4     ✔ tidyr     1.3.1
✔ purrr     1.0.4
── Conflicts ────────────────────────────────────────── tidyverse_con
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to forc
```

Importing data

Before we can do anything with our data, we need to first import it into R.

We can either use the data import from the RStudio drop-down menu (`File > Import Dataset`), or we can use R functions for reading in data (Recommended). These functions generally start with `read`. The Base R read functions are followed by a `.`, while the `readr` functions are followed by an `_`. `readr` functions are from the `readr` package, which is a part of the tidyverse. `readr` functions are typically faster, more reproducible and consistent, and are better at recognizing certain types of data (e.g., dates). However, they also result in `tibbles` rather than data frames, and are not row name friendly.

Info

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.

Some different import functions

Import Excel files:
- `readxl::read_excel()`. - `readxl::read_xls()`. - `readxl::read_xlsx()`

Import tab-delimited files (.tsv, .txt):
- `read.delim()`
- `read.table()`. - `readr::read_delim()`. - `readr::read_tsv()`
- `readr::read_table()`

Comma separated files (.csv):
- `read.csv()`
- `readr::read_csv()`

The most important argument of all of these functions is the `file path`.

File paths

A file path tells us the location of a file or folder (a.k.a., directory). Because it is a character string, it must be surrounded by quotes. Each directory is separated by a `/`. It is best practice to work in R projects and use relative file paths to make scripts more reproducible.

Genomic Data:
- For genomic specific formats, you will likely need to install specific packages; check out Bioconductor *(https://bioconductor.org/)* for packages relevant to bioinformatics.

Other:
- 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* by Carrie Wright, Shannon E. Ellis, Stephanie C. Hicks and Roger D. Peng.

Load the lesson data

For today's lesson, we will work with data available from R (Base R and the tidyverse) as well as an example RNA-Seq count matrix. The count matrix is currently in the format "genes x samples", with the gene IDs, which are a combination of Ensembl IDs and gene symbols, as row names.

Get the Data

To download the data used in this lesson and future lessons to your local computer, click here. You can then move the downloaded directory to your working directory in R.

To use the data on Biowulf, open your Terminal in R and follow these steps:

```
# change to your working directory
cd /data/$USER/Data_Wrangling_with_R
# use wget to grab the zipped directory
wget https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Intro_t
# unzip the data
unzip -d data data.zip
```

Alternatively, you can download the zip to local and upload to RStudio Server.

Load the Data

Let's use `read.delim` to load the data.

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

```
                      Accession.SRR1039508 Accession.SRR1039509
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.SRR1039513
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.SRR1039517
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.SRR1039521
ENSG00000000003.TSPAN6                  770                  572
ENSG00000000005.TNMD                      0                    0
ENSG00000000419.DPM1                    417                  508
ENSG00000000457.SCYL3                   233                  229
ENSG00000000460.C1orf112                 76                   60
ENSG00000000938.FGR                       0                    0
```

The first thing we should do following data import is to examine the data. We need to know what is included in this data frame. What are the dimensions? What types of data are stored in each column?

How can we examine these data further?

```
str(aircount)
```

```
'data.frame':   50 obs. of  8 variables:
 $ Accession.SRR1039508: int  679 0 467 260 60 0 3251 1433 519 394 .
 $ Accession.SRR1039509: int  448 0 515 211 55 0 3679 1062 380 236 .
 $ Accession.SRR1039512: int  873 0 621 263 40 2 6177 1733 595 464 .
 $ Accession.SRR1039513: int  408 0 365 164 35 0 4252 881 493 175 ..
 $ Accession.SRR1039516: int  1138 0 587 245 78 1 6721 1424 820 658
 $ Accession.SRR1039517: int  1047 0 799 331 63 0 11027 1439 714 584
 $ Accession.SRR1039520: int  770 0 417 233 76 0 5176 1359 696 360 .
 $ Accession.SRR1039521: int  572 0 508 229 60 0 7995 1109 704 269 .
```

```
glimpse(aircount)
```

```
Rows: 50
Columns: 8
$ Accession.SRR1039508 <int> 679, 0, 467, 260, 60, 0, 3251, 1433, 51
$ Accession.SRR1039509 <int> 448, 0, 515, 211, 55, 0, 3679, 1062, 38
$ Accession.SRR1039512 <int> 873, 0, 621, 263, 40, 2, 6177, 1733, 59
$ Accession.SRR1039513 <int> 408, 0, 365, 164, 35, 0, 4252, 881, 493
$ Accession.SRR1039516 <int> 1138, 0, 587, 245, 78, 1, 6721, 1424, 8
$ Accession.SRR1039517 <int> 1047, 0, 799, 331, 63, 0, 11027, 1439, 
$ Accession.SRR1039520 <int> 770, 0, 417, 233, 76, 0, 5176, 1359, 69
$ Accession.SRR1039521 <int> 572, 0, 508, 229, 60, 0, 7995, 1109, 70
```

Data reshape

Now that we have some data to work with, let's learn how we can reshape it. Recall how we defined tidy data.

Specifically, tidy data has 3 components:

1. Each column is a **variable**, a quantity, quality, or property that can be collected or measured.
2. Each row is an **observation**, or set of values collected under similar conditions.
3. Each cell is a **value**, or state of a variable when you measure it. --- r4ds *(https://r4ds.hadley.nz/data-visualize.html#:~:text=A%20variable%20is%20a%20quantity,change%20from%20measurement*

We can organize data in many different ways. Some of these ways will be easier to work with, generally the tidy way.

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 (e.g., analysis goals).

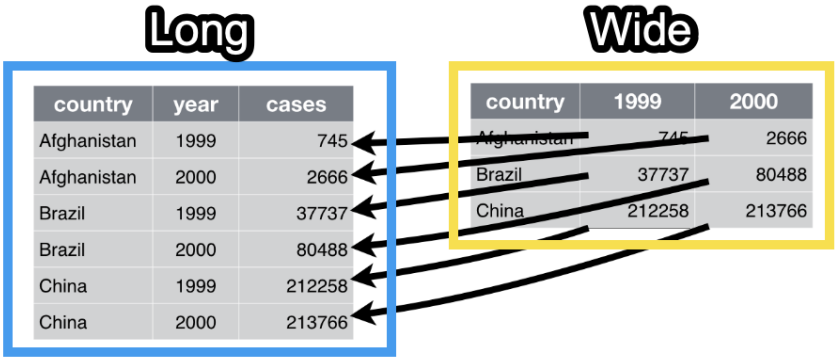
> **Note**

> 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:



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?

Wide format:

```
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
```

Long format:

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

With the long format it is easier to summarize information about the properties of the flowers but in the wide format it is easier to explore relationships between these properties.

For example, this code is simpler

```
iris_long %>%
  group_by(Species, Flower_property) %>%
  summarize(mean= mean(Measurement), sd = sd(Measurement))
```

```
`summarise()` has grouped output by 'Species'. You can override using
`.groups` argument.
```

```
# A tibble: 12 × 4
# Groups:   Species [3]
   Species    Flower_property  mean    sd
   <fct>      <chr>           <dbl> <dbl>
 1 setosa     Petal.Length    1.46  0.174
 2 setosa     Petal.Width     0.246 0.105
 3 setosa     Sepal.Length    5.01  0.352
 4 setosa     Sepal.Width     3.43  0.379
 5 versicolor Petal.Length    4.26  0.470
 6 versicolor Petal.Width     1.33  0.198
 7 versicolor Sepal.Length    5.94  0.516
 8 versicolor Sepal.Width     2.77  0.314
 9 virginica  Petal.Length    5.55  0.552
10 virginica  Petal.Width     2.03  0.275
11 virginica  Sepal.Length    6.59  0.636
12 virginica  Sepal.Width     2.97  0.322
```

than this

```
iris %>% group_by(Species) %>%
  summarize(across(where(is.numeric),list(mean = mean, sd=sd)))
```

```
# A tibble: 3 × 9
  Species    Sepal.Length_mean Sepal.Length_sd Sepal.Width_mean Sepa
  <fct>                  <dbl>           <dbl>            <dbl>
1 setosa                  5.01           0.352             3.43
2 versicolor              5.94           0.516             2.77
3 virginica               6.59           0.636             2.97
# ℹ 4 more variables: Petal.Length_mean <dbl>, Petal.Length_sd <dbl>
#   Petal.Width_mean <dbl>, Petal.Width_sd <dbl>
```

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, but this again, will depend on your task.***

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.

Pivot_longer

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

For the `cols` argument, we can select columns using the same arguments we would use with `select()`, including column names, indices, or the `select` helper functions, for example,

`contains()`,
`starts_with()`,
`ends_with()`,
etc.

Columns in the Tidyverse

In Base R, we often have to refer to data variables (columns) directly using an accessor like `$`. However, this is not the case in the tidyverse. In the tidyverse, columns that exist generally do not need quotes, while columns that do not yet exist do need quotes. This difference has important implications for creating for loops and functions. Learn more about tidy evaluation here *(https://dplyr.tidyverse.org/articles/programming.html)*.

Let's pivot `aircount`.

```
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
```

Pivot_wider

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. **Because these columns already exist, we do not need to put them in quotes.**

Let's pivot the data from long to wide.

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

```
# A tibble: 6 × 9
  Feature         Accession.SRR1039508 Accession.SRR1039509 Accessior
  <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.SRR10395
#   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.

Test our knowledge

What function would we use to transform table A to table B?

Table A:

```
# A tibble: 19 × 12
   fish  Release I80_1 Lisbon  Rstr Base_TD   BCE   BCW  BCE2  BCW2
   <fct>   <int> <int>  <int> <int>   <int> <int> <int> <int> <int> <
 1 4842        1     1      1     1       1     1     1     1     1
 2 4843        1     1      1     1       1     1     1     1     1
 3 4844        1     1      1     1       1     1     1     1     1
 4 4845        1     1      1     1       1    NA    NA    NA    NA
 5 4847        1     1      1    NA      NA    NA    NA    NA    NA
 6 4848        1     1      1     1      NA    NA    NA    NA    NA
 7 4849        1     1     NA    NA      NA    NA    NA    NA    NA
 8 4850        1     1     NA     1       1     1     1    NA    NA
 9 4851        1     1     NA    NA      NA    NA    NA    NA    NA
10 4854        1     1     NA    NA      NA    NA    NA    NA    NA
11 4855        1     1      1     1       1    NA    NA    NA    NA
12 4857        1     1      1     1       1     1     1     1     1
13 4858        1     1      1     1       1     1     1     1     1
14 4859        1     1      1     1       1    NA    NA    NA    NA
15 4861        1     1      1     1       1     1     1     1     1
16 4862        1     1      1     1       1     1     1     1     1
17 4863        1     1     NA    NA      NA    NA    NA    NA    NA
18 4864        1     1     NA    NA      NA    NA    NA    NA    NA
19 4865        1     1      1    NA      NA    NA    NA    NA    NA
```

Table B:

```
# A tibble: 114 × 3
   fish  station  seen
   <fct> <fct>   <int>
 1 4842  Release     1
 2 4842  I80_1       1
 3 4842  Lisbon      1
 4 4842  Rstr        1
 5 4842  Base_TD     1
 6 4842  BCE         1
 7 4842  BCW         1
 8 4842  BCE2        1
```

```
 9 4842  BCW2         1
10 4842  MAE          1
# ℹ 104 more rows
```

Solution

pivot_longer

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.

Separate

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 `.`. This requires the `data`, the column we want to separate (`col`), and the names of the new variables to create from the separated column (`into`). The default separator to split the columns is `"[^[:alnum:]]+"`. This is a regular expression that matches 1 or more non-alphanumeric values (i.e., characters that are neither alphabetical (a-z) nor numerical(0-9)).

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

separate_wider_position() and separate_wider_delim()

`separate()` has been superseded in favor of `separate_wider_position()`, `separate_wider_delim()`, and `separate_wider_regex()`. "A superseded function has a known better alternative, but the function itself is not going away." *(https://cran.r-project.org/web/packages/lifecycle/vignettes/stages.html)*

`separate_wider_delim()` - splits by delimiter.
`separate_wider_position()` - splits at fixed widths.
`separate_wider_regex()` - splits with regular expression matches.

Unite

`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. 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.

The Janitor package.

Check out the `janitor` *(https://sfirke.github.io/janitor/index.html)* package for additional functions for exploring and cleaning messy data.

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/)*.

Resources

readr / readxl cheatsheet *(https://rstudio.github.io/cheatsheets/html/data-import.html?_gl=1*1cexwpw*_ga*MTY1MjAxMTE4My4xNzUwNjY5NzMy*_ga_2C0WZ1JHG0*czE3NTA3MDgwNjgkbzIkZz*
Tidyr cheatsheet *(https://rstudio.github.io/cheatsheets/html/tidyr.html?_gl=1*4wx4lc*_ga*MTY1MjAxMTE4My4xNzUwNjY5NzMy*_ga_2C0WZ1JHG0*czE3NTA3MDgwNjgkbzIkZzA*
Stringr / regex cheatsheet *(https://rstudio.github.io/cheatsheets/html/strings.html?_gl=1*1cexwpw*_ga*MTY1MjAxMTE4My4xNzUwNjY5NzMy*_ga_2C0WZ1JHG0*czE3NTA3MDgwNjgkbzIkZz*

Subsetting Data with dplyr

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 (`%>%` or `|>`) operator to link functions

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

What is `dplyr`?

> dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:
>
> mutate() adds new variables that are functions of existing variables
> select() picks variables based on their names.
> filter() picks cases based on their values.
> summarise() reduces multiple values down to a single summary.
> arrange() changes the ordering of the rows. - dplyr.tidyverse.org *(https://dplyr.tidyverse.org/index.html)*

dplyr is also used to combine data tables sharing common IDs and to manipulate data in data backends.

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 ──────────────────────── tidyver
✔ dplyr     1.1.4     ✔ readr     2.1.5
```

```
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.2     ✔ tibble    3.3.0
✔ lubridate 1.9.4     ✔ tidyr     1.3.1
✔ purrr     1.0.4
── Conflicts ──────────────────────────────────────────── tidyverse_cor
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to for
```

Importing data

For this lesson, we will use sample metadata and differential expression results derived from the `airway` RNA-Seq project. See here *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Intro_to_Data_Wrangling/Lesson2/#get-the-data)* for instructions on accessing the data.

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, BioSa
dbl (1): avgLength

ℹ Use `spec()` to retrieve the full column specification for this da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

```
smeta
```

```
# A tibble: 8 × 9
  SampleName cell    dex   albut Run       avgLength Experiment Samp
  <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <chr
1 GSM1275862 N61311  untrt untrt SRR10395…       126 SRX384345  SRS5(
2 GSM1275863 N61311  trt   untrt SRR10395…       126 SRX384346  SRS5(
3 GSM1275866 N052611 untrt untrt SRR10395…       126 SRX384349  SRS5(
4 GSM1275867 N052611 trt   untrt SRR10395…        87 SRX384350  SRS5(
5 GSM1275870 N080611 untrt untrt SRR10395…       120 SRX384353  SRS5(
6 GSM1275871 N080611 trt   untrt SRR10395…       126 SRX384354  SRS5(
7 GSM1275874 N061011 untrt untrt SRR10395…       101 SRX384357  SRS5(
8 GSM1275875 N061011 trt   untrt SRR10395…        98 SRX384358  SRS5(
```

```
#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 da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

```
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 32.8
 2 ENSG000… untrt DPM1       hg38       TRUE       0.198    4.61  6.9
 3 ENSG000… untrt SCYL3      hg38       TRUE       0.0292   3.48  0.(
 4 ENSG000… untrt C1orf112   hg38       TRUE      -0.124    1.47  0.3
 5 ENSG000… untrt CFH        hg38       TRUE       0.417    8.09 29.3
 6 ENSG000… untrt FUCA2      hg38       TRUE      -0.250    5.91 14.9
 7 ENSG000… untrt GCLC       hg38       TRUE      -0.0581   4.84  0.3
 8 ENSG000… untrt NFYA       hg38       TRUE      -0.509    4.13 44.9
 9 ENSG000… untrt STPG1      hg38       TRUE      -0.136    3.12  1.(
10 ENSG000… untrt NIPAL3     hg38       TRUE      -0.0500   7.04  0.3
```

```
# 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", "GSM127
$ cell       <chr> "N61311", "N61311", "N052611", "N052611", "N08061
$ dex        <chr> "untrt", "trt", "untrt", "trt", "untrt", "trt", "
$ albut      <chr> "untrt", "untrt", "untrt", "untrt", "untrt", "unt
$ Run        <chr> "SRR1039508", "SRR1039509", "SRR1039512", "SRR103
$ avgLength  <dbl> 126, 126, 126, 87, 120, 126, 101, 98
$ Experiment <chr> "SRX384345", "SRX384346", "SRX384349", "SRX384350
$ Sample     <chr> "SRS508568", "SRS508567", "SRS508571", "SRS508572
$ BioSample  <chr> "SAMN02422669", "SAMN02422675", "SAMN02422678", "
```

```
glimpse(dexp)
```

```
Rows: 15,926
Columns: 10
$ feature    <chr> "ENSG00000000003", "ENSG00000000419", "ENSG000000
$ albut      <chr> "untrt", "untrt", "untrt", "untrt", "untrt", "unt
$ transcript <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH", "FU
$ ref_genome <chr> "hg38", "hg38", "hg38", "hg38", "hg38", "hg38", "h
$ .abundant  <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TR
$ logFC      <dbl> -0.390100222, 0.197802179, 0.029160865, -0.124382
$ logCPM     <dbl> 5.059704, 4.611483, 3.482462, 1.473375, 8.089146,
$ F          <dbl> 3.284948e+01, 6.903534e+00, 9.685073e-02, 3.77213
$ PValue     <dbl> 0.0003117656, 0.0280616149, 0.7629129276, 0.55469
$ FDR        <dbl> 0.002831504, 0.077013489, 0.844247837, 0.68232661
```

Always know how your data is structured.

Before you do anything with your data, always check out the structure of your data to avoid surprises.

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

Subsetting data in base R

If you remember back to "Getting Started with R" *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting_Started_with_R/Lesson5/)*, 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. Just as in Base R subsetting, the order of the columns will determine the order of the columns in the new data frame.

Let's select the transcript, logFC, and FDR corrected p-value columns:

```
#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.

The syntax to rename is `new_name = old_name`.

```
#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
```

new name	old name
gene	transcript
logFoldChange	logFC
FDRpvalue	FDR

Using `rename()` or `rename_with()`

If you want to retain all columns, you could also use `rename()` *(https://dplyr.tidyverse.org/reference/rename.html)* from `dplyr` to rename columns.
For example, let's rename only `transcript` to `gene` from `dexp`.

```
rename(dexp, gene=transcript)
```

```
# A tibble: 15,926 × 10
   feature       albut gene  ref_genome .abundant   logFC logCPM
   <chr>         <chr> <chr> <chr>      <lgl>       <dbl>  <dbl>   <
 1 ENSG00000000… untrt TSPA… hg38       TRUE      -0.390    5.06 32.8
 2 ENSG00000000… untrt DPM1  hg38       TRUE       0.198    4.61  6.9
 3 ENSG00000000… untrt SCYL3 hg38       TRUE       0.0292   3.48  0.0
 4 ENSG00000000… untrt C1or… hg38       TRUE      -0.124    1.47  0.3
 5 ENSG00000000… untrt CFH   hg38       TRUE       0.417    8.09 29.3
 6 ENSG00000001… untrt FUCA2 hg38       TRUE      -0.250    5.91 14.9
 7 ENSG00000001… untrt GCLC  hg38       TRUE      -0.0581   4.84  0.1
 8 ENSG00000001… untrt NFYA  hg38       TRUE      -0.509    4.13 44.9
 9 ENSG00000001… untrt STPG1 hg38       TRUE      -0.136    3.12  1.0
10 ENSG00000001… untrt NIPA… hg38       TRUE      -0.0500   7.04  0.3
# ℹ 15,916 more rows
# ℹ 1 more variable: FDR <dbl>
```

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    PV
```

```
   <chr> <chr>      <chr>      <lgl>       <dbl>  <dbl>   <dbl>    <d
 1 untrt TSPAN6     hg38       TRUE       -0.390    5.06 32.8    0.000
 2 untrt DPM1       hg38       TRUE        0.198    4.61  6.90   0.028
 3 untrt SCYL3      hg38       TRUE        0.0292   3.48  0.0969 0.763
 4 untrt C1orf112   hg38       TRUE       -0.124    1.47  0.377  0.555
 5 untrt CFH        hg38       TRUE        0.417    8.09 29.3    0.000
 6 untrt FUCA2      hg38       TRUE       -0.250    5.91 14.9    0.004
 7 untrt GCLC       hg38       TRUE       -0.0581   4.84  0.167  0.692
 8 untrt NFYA       hg38       TRUE       -0.509    4.13 44.9    0.000
 9 untrt STPG1      hg38       TRUE       -0.136    3.12  1.04   0.335
10 untrt NIPAL3     hg38       TRUE       -0.0500   7.04  0.350  0.569
# ℹ 15,916 more rows
```

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

```
# A tibble: 15,926 × 9
   albut transcript ref_genome .abundant   logFC logCPM       F   PVa
   <chr> <chr>      <chr>      <lgl>       <dbl>  <dbl>   <dbl>    <d
 1 untrt TSPAN6     hg38       TRUE       -0.390    5.06 32.8    0.000
 2 untrt DPM1       hg38       TRUE        0.198    4.61  6.90   0.028
 3 untrt SCYL3      hg38       TRUE        0.0292   3.48  0.0969 0.763
 4 untrt C1orf112   hg38       TRUE       -0.124    1.47  0.377  0.555
 5 untrt CFH        hg38       TRUE        0.417    8.09 29.3    0.000
 6 untrt FUCA2      hg38       TRUE       -0.250    5.91 14.9    0.004
 7 untrt GCLC       hg38       TRUE       -0.0581   4.84  0.167  0.692
 8 untrt NFYA       hg38       TRUE       -0.509    4.13 44.9    0.000
 9 untrt STPG1      hg38       TRUE       -0.136    3.12  1.04   0.335
10 untrt NIPAL3     hg38       TRUE       -0.0500   7.04  0.350  0.569
# 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    0.000312 0
 2 DPM1       hg38        TRUE        0.198    4.61  6.90   0.0281   0
 3 SCYL3      hg38        TRUE        0.0292   3.48  0.0969 0.763    0
 4 C1orf112   hg38        TRUE       -0.124    1.47  0.377  0.555    0
 5 CFH        hg38        TRUE        0.417    8.09 29.3    0.000463 0
 6 FUCA2      hg38        TRUE       -0.250    5.91 14.9    0.00405  0
 7 GCLC       hg38        TRUE       -0.0581   4.84  0.167  0.692    0
 8 NFYA       hg38        TRUE       -0.509    4.13 44.9    0.000100 0
 9 STPG1      hg38        TRUE       -0.136    3.12  1.04   0.335    0
10 NIPAL3     hg38        TRUE       -0.0500   7.04  0.350  0.569    0
# ℹ 15,916 more rows
```

```
#Note: this also would have excluded the feature column
```

If we are interested in moving a column without selection, we can use `relocate()`. We should include the columns we want to move and where we would like to put them.

```
relocate(dexp, transcript, .before=feature)
```

```
# A tibble: 15,926 × 10
   transcript feature  albut ref_genome .abundant   logFC logCPM
   <chr>      <chr>    <chr> <chr>      <lgl>       <dbl>  <dbl>   <d
 1 TSPAN6     ENSG000… untrt hg38       TRUE      -0.390    5.06 32.8
 2 DPM1       ENSG000… untrt hg38       TRUE       0.198    4.61  6.9
 3 SCYL3      ENSG000… untrt hg38       TRUE       0.0292   3.48  0.0
 4 C1orf112   ENSG000… untrt hg38       TRUE      -0.124    1.47  0.3
 5 CFH        ENSG000… untrt hg38       TRUE       0.417    8.09 29.3
 6 FUCA2      ENSG000… untrt hg38       TRUE      -0.250    5.91 14.9
 7 GCLC       ENSG000… untrt hg38       TRUE      -0.0581   4.84  0.1
 8 NFYA       ENSG000… untrt hg38       TRUE      -0.509    4.13 44.9
 9 STPG1      ENSG000… untrt hg38       TRUE      -0.136    3.12  1.0
10 NIPAL3     ENSG000… untrt hg38       TRUE      -0.0500   7.04  0.3
# ℹ 15,916 more rows
# ℹ 1 more variable: FDR <dbl>
```

Note

By default, relocate() will move columns to the left-hand side of the data frame.

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()`, and operators (`!`, &, `|`) for combining selections.

```
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
```

```
#or
select(dexp, transcript, starts_with("log") | ends_with("r"))
```

```
# 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()`) or this reference *(https://tidyselect.r-lib.org/reference/language.html)* from `tidyselect`.

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
```

```
# Not recommended
select_if(dexp, is.numeric) #scoped verbs are superseded
```

```
# 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
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 Samp
  <chr>      <chr>  <chr> <chr> <chr>          <dbl> <chr>      <chr
1 GSM1275863 N61311 trt   untrt SRR1039509       126 SRX384346  SRS5(
```

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 Samp
  <chr>      <chr>  <chr> <chr> <chr>          <dbl> <chr>      <chr
1 GSM1275863 N61311 trt   untrt SRR1039509       126 SRX384346  SRS5(
```

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 Samp
  <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <chr
1 GSM1275862 N61311  untrt untrt SRR10395…       126 SRX384345  SRS5(
2 GSM1275863 N61311  trt   untrt SRR10395…       126 SRX384346  SRS5(
3 GSM1275870 N080611 untrt untrt SRR10395…       120 SRX384353  SRS5(
4 GSM1275871 N080611 trt   untrt SRR10395…       126 SRX384354  SRS5(
```

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 Samp
  <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <chr
1 GSM1275866 N052611 untrt untrt SRR10395…       126 SRX384349  SRS5(
2 GSM1275867 N052611 trt   untrt SRR10395…        87 SRX384350  SRS5(
3 GSM1275874 N061011 untrt untrt SRR10395…       101 SRX384357  SRS5(
4 GSM1275875 N061011 trt   untrt SRR10395…        98 SRX384358  SRS5(
```

Including multiple phrases

We can use multiple expressions in a single call to `filter()`. For example, let's filter `dexp` to include only named transcripts (i.e.,no NAs), values of |log fold change| is greater than 2, and either a p-value or FDR corrected p_value is less than or equal to 0.01.

```
#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))
sig_annot_transcripts
```

```
# A tibble: 178 × 10
   feature     albut transcript ref_genome .abundant logFC  logCPM
   <chr>       <chr> <chr>      <chr>      <lgl>     <dbl>   <dbl> <(
 1 ENSG000000… untrt PDK4       hg38       TRUE       2.55  5.41    3
 2 ENSG000000… untrt SLC7A14    hg38       TRUE      -2.89  3.54   27
 3 ENSG000000… untrt NPC1L1     hg38       TRUE      -2.61 -0.0372  6
 4 ENSG000000… untrt CHDH       hg38       TRUE      -2.01  2.14   13
 5 ENSG000000… untrt HSD17B6    hg38       TRUE      -2.03  3.02    9
 6 ENSG000000… untrt POU2F2     hg38       TRUE      -2.06  0.835  10
 7 ENSG000000… untrt GPM6B      hg38       TRUE       2.43  5.67   17
 8 ENSG000000… untrt PER3       hg38       TRUE      -2.21  3.22    8
 9 ENSG000000… untrt COL11A1    hg38       TRUE       2.41  4.06   40
10 ENSG000000… untrt FGFR2      hg38       TRUE      -2.26  0.499   6
# ℹ 168 more rows
# ℹ 1 more variable: FDR <dbl>
```

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. Let's return only rows with values of less than 0.05 in the columns `PValue` and `FDR`.

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

```
# A tibble: 4,967 × 10
   feature     albut transcript ref_genome .abundant  logFC logCPM
   <chr>       <chr> <chr>      <chr>      <lgl>      <dbl>  <dbl> <
 1 ENSG000000… untrt TSPAN6     hg38       TRUE      -0.390   5.06 3
 2 ENSG000000… untrt CFH        hg38       TRUE       0.417   8.09 2
 3 ENSG000000… untrt FUCA2      hg38       TRUE      -0.250   5.91 1
 4 ENSG000000… untrt NFYA       hg38       TRUE      -0.509   4.13 4
 5 ENSG000000… untrt SEMA3F     hg38       TRUE      -0.259   4.81 1
 6 ENSG000000… untrt ANKIB1     hg38       TRUE      -0.236   6.38 1
 7 ENSG000000… untrt RAD52      hg38       TRUE      -0.319   3.13  
 8 ENSG000000… untrt LASP1      hg38       TRUE       0.388   8.39 2
 9 ENSG000000… untrt SNX11      hg38       TRUE       0.395   3.56 1
10 ENSG000000… untrt TMEM176A   hg38       TRUE       0.357   4.65 1
# ℹ 4,957 more rows
# ℹ 1 more variable: FDR <dbl>
```

Anonymous function

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 Stack Overflow post *(https://stackoverflow.com/questions/56532119/dplyr-piping-data-difference-between-and-x)* useful.
Therefore, the above line could have been written as follows: This function could be written like this:

```
my_func <- function(x) {
  x < 0.05
}
```

```
filter(dexp, if_all(PValue:FDR, my_func))
```

```
# A tibble: 4,967 × 10
  feature     albut transcript ref_genome .abundant  logFC logCPM     F  PValue
  <chr>       <chr> <chr>      <chr>      <lgl>      <dbl>  <dbl> <dbl>   <dbl>
1 ENSG000000… untrt TSPAN6     hg38       TRUE      -0.390   5.06 32.8  3.12e-4
2 ENSG000000… untrt CFH        hg38       TRUE       0.417   8.09 29.3  4.63e-4
3 ENSG000000… untrt FUCA2      hg38       TRUE      -0.250   5.91 14.9  4.05e-3
4 ENSG000000… untrt NFYA       hg38       TRUE      -0.509   4.13 44.9  1.00e-4
5 ENSG000000… untrt SEMA3F     hg38       TRUE      -0.259   4.81 12.3  6.98e-3
6 ENSG000000… untrt ANKIB1     hg38       TRUE      -0.236   6.38 14.5  4.41e-3
7 ENSG000000… untrt RAD52      hg38       TRUE      -0.319   3.13  9.03 1.53e-2
8 ENSG000000… untrt LASP1      hg38       TRUE       0.388   8.39 22.7  1.11e-3
9 ENSG000000… untrt SNX11      hg38       TRUE       0.395   3.56 18.7  2.05e-3
10 ENSG000000… untrt TMEM176A   hg38       TRUE       0.357   4.65 12.1  7.30e-3
# ℹ 4,957 more rows
# ℹ 1 more variable: FDR <dbl>
```

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.

Step by Step

```
#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"
```

Nesting Code

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

Using the pipe (%>%, |>)

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*. You can use either the `magrittr` pipe (`%>%`), which loads with the tidyverse, or the native R pipe (`|>`, R version +4.1).

!!! info `%>%` vs `|>` These pipes behave in largely the same way. However, `%>%` does have some special behaviors. You can read more 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

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

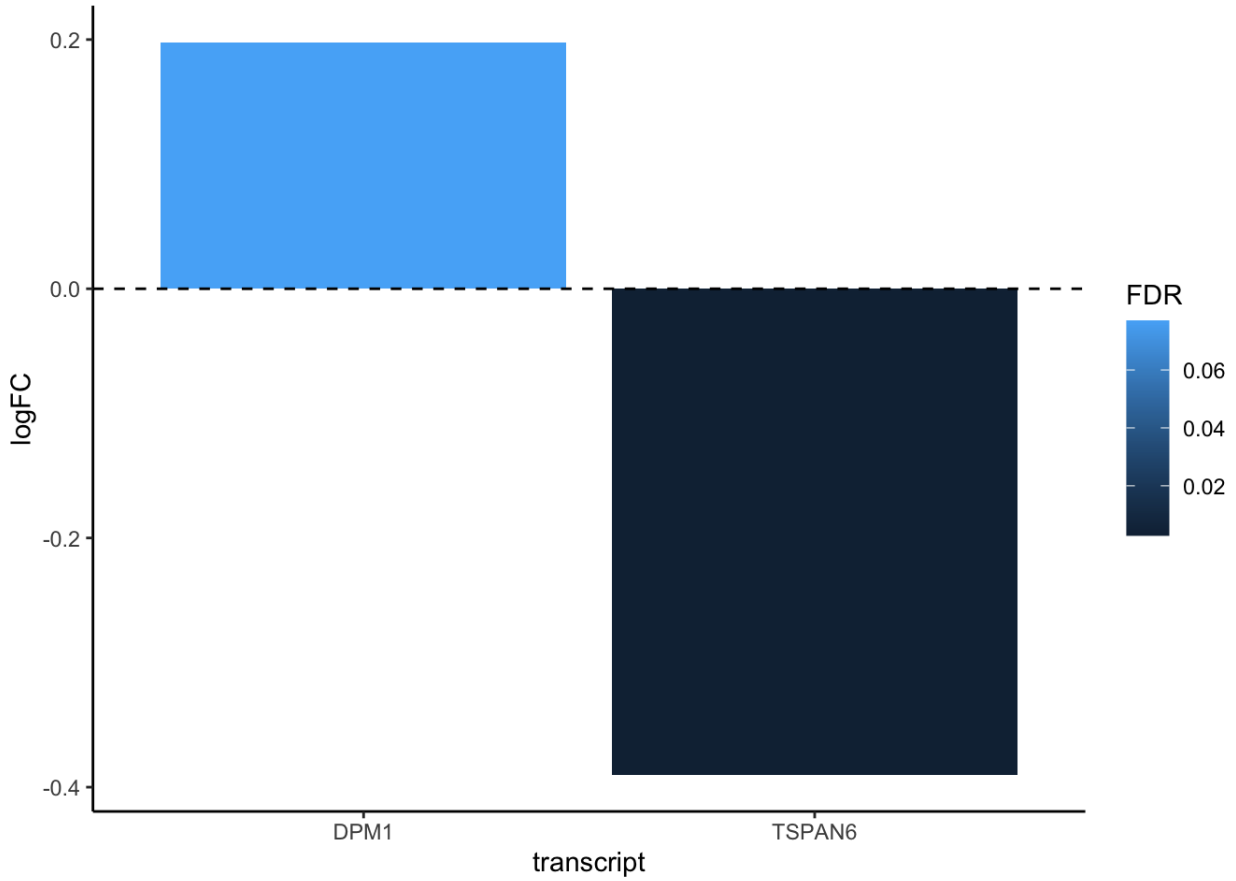
```
# A tibble: 2 × 3
  transcript  logFC     FDR
  <chr>       <dbl>   <dbl>
1 TSPAN6     -0.390 0.00283
2 DPM1        0.198 0.0770
```

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 da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```



Note

ggplot2 will be covered in Part 3 of this course.

> 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)*

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.github.io/genomics-r-intro/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://github.com/suzanbaert/Dplyr_Tutorials/tree/master)*.

Summarizing Data with dplyr

Objectives.

1. This lesson will introduce the "split-apply-combine" approach to data analysis and the key players in the `dplyr` package used to implement this type of workflow:
    - `group_by()`
    - `summarize()`
2. We will also learn about other useful dplyr functions including
    - `arrange()`
    - `distinct()`

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

Load Tidyverse

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 ──────────────────────── tidyver
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.2     ✔ tibble    3.3.0
✔ lubridate 1.9.4     ✔ tidyr     1.3.1
✔ purrr     1.0.4
── Conflicts ────────────────────────────────────────── tidyverse_con
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to for
```

Load the 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 derived from the airway RNA-Seq project.

Get the sample metadata:

```
#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, BioSa
dbl (1): avgLength

ℹ Use `spec()` to retrieve the full column specification for this da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

Get the raw counts:

```
#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, SR
SRR1039...
ℹ Use `spec()` to retrieve the full column specification for this da
Specify the column types or set `show_col_types = FALSE` to quiet th
• `` -> `...1`
```

Here we used `read_csv` and `rename` to load the raw count data. Remember, `rename` allows us to rename any column without selection.

Get the differential expression results:

```
#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 da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

Group_by and summarize

There is an approach to data analysis known as "split-apply-combine", in which the data are 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.

Before diving into this further, let's create some more interesting data to work with by merging our count matrix with our sample metadata.

```
acount_smeta <- acount %>%
  pivot_longer(where(is.numeric), names_to ="Sample",
               values_to= "Count") %>% #reshape the data
  left_join(smeta, by=c("Sample"="Run")) #join with meta data

acount_smeta
```

```
# A tibble: 512,816 × 11
   Feature        Sample Count SampleName cell  dex   albut avgLengtl
   <chr>          <chr>  <dbl> <chr>      <chr> <chr> <chr>     <dbl>
 1 ENSG000000000… SRR10…   679 GSM1275862 N613… untrt untrt       12(
 2 ENSG000000000… SRR10…   448 GSM1275863 N613… trt   untrt       12(
 3 ENSG000000000… SRR10…   873 GSM1275866 N052… untrt untrt       12(
 4 ENSG000000000… SRR10…   408 GSM1275867 N052… trt   untrt        8
 5 ENSG000000000… SRR10…  1138 GSM1275870 N080… untrt untrt       12(
 6 ENSG000000000… SRR10…  1047 GSM1275871 N080… trt   untrt       12(
 7 ENSG000000000… SRR10…   770 GSM1275874 N061… untrt untrt       10
 8 ENSG000000000… SRR10…   572 GSM1275875 N061… trt   untrt        9
 9 ENSG000000000… SRR10…     0 GSM1275862 N613… untrt untrt       12(
10 ENSG000000000… SRR10…     0 GSM1275863 N613… trt   untrt       12(
```

```
# ℹ 512,806 more rows
# ℹ 2 more variables: Sample.y <chr>, BioSample <chr>
```

`left_join()`

`left_join()` is a mutating join function from `dplyr`. We will learn more about this function in the next lesson. Don't dwell on the code too much here.

Key Functions

Here we are interested in functions that allow us to summarize our data. These include.

- `group_by()` - group a data frame by a categorical variable so that a given operation can be performed per group / category. The data frame will not be reorganized, but it will have a grouping attribute, which will impact how tidyverse functions interact with it.
- `summarize()` - computes summary statistics (1 or more) in a data frame. This function creates a new data frame, returning one row for each combination of grouping variables. If there are no grouping variables, the output will have a single row summarizing all observations in the input. See `?summarize`.

  The syntax:
  `summarize(new_column = operations_on_existing_columns)`

  where `new_column` *is the name of the new column to appear in the resulting summary table, and* `operations_on_existing_columns` *is where we apply summary functions to an existing column to create what will go in* `new_column`*. This should return a single value. To return more than one value per group, see* `?reframe()`*.*

  `summarize` may include multiple `new_column = operations_on_existing_columns` statements, with each statement separated by `,`. **We will see a similar syntax with** `mutate`**.**
- `count()` - computes groupwise counts. This does not require `group_by`.
- `ungroup()` - removes the grouping criteria set by `group_by()`. This is useful for performing additional operations that you do not want applied by group.

`.by`

`summarize()` can provide results by group without `group_by` using the `.by` argument.

For example,

Let's compute the median raw counts for each gene by treatment.

```
#Call the data
medcount<- 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))
```

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

```
medcount
```

```
# A tibble: 128,204 × 3
# Groups:   dex [2]
   dex   Feature         median_counts
   <chr> <chr>                   <dbl>
 1 trt   ENSG00000000003         510
 2 trt   ENSG00000000005           0
 3 trt   ENSG00000000419         512.
 4 trt   ENSG00000000457         220
 5 trt   ENSG00000000460          57.5
 6 trt   ENSG00000000938           0
 7 trt   ENSG00000000971        6124.
 8 trt   ENSG00000001036        1086.
 9 trt   ENSG00000001084         598.
10 trt   ENSG00000001167         252.
# ℹ 128,194 more rows
```

Using `summarize()`, by default the output is grouped by every grouping column except the last (e.g., here, no longer grouped by "Feature"), which is helpful for performing additional operations at higher levels of grouping (e.g., "dex").

Now, let's obtain the top five genes with the greatest median raw counts by treatment using `slice_max`. Remember, `medcount` has grouped output by `dex`. This grouping is maintained unless `ungroup` was applied.

```
medcount  %>%
  slice_max(n=5, order_by=median_counts) #notice use of slice_max
```

```
# 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
```

Often we are interested in knowing more about sample sizes and including that information in summary output. For example, 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 the
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 (See `n_distinct(acount_smeta$Feature)`).

na.rm

> 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
```

Lastly, similar to mutate, we can summarize across multiple columns at once using `across()`. We will focus more heavily on `across()` next lesson. 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
```

Additional Examples

Let's use `penguins` for additional practice.

The penguins data contains

> Data on adult penguins covering three species found on three islands in the Palmer Archipelago, Antarctica, including their size (flipper length, body mass, bill dimensions), and sex. - penguins docs

Let's summarize these data by finding the mean penguin body mass by penguin species. Remember to include `na.rm = TRUE` to exclude missing values.

```
penguins %>%
  group_by(species) %>%
  summarize(mean_mass = mean(body_mass, na.rm = TRUE))
```

```
# A tibble: 3 × 2
  species   mean_mass
  <fct>         <dbl>
1 Adelie        3701.
2 Chinstrap     3733.
3 Gentoo        5076.
```

What if we also want to include the standard deviation by species?

```
penguins %>%
  group_by(species) %>%
  summarize(mean_mass = mean(body_mass, na.rm = TRUE),
            sd_mass = sd(body_mass, na.rm = TRUE))
```

```
# A tibble: 3 × 3
  species   mean_mass sd_mass
  <fct>         <dbl>   <dbl>
1 Adelie        3701.    459.
2 Chinstrap     3733.    384.
3 Gentoo        5076.    504.
```

Looking for more functions to use with summarize? Here *(https://r4ds.had.co.nz/transform.html?q=summar#summarise-funs)* are some useful summary functions. However, the use of `summarize()` is not limited to these suggestions.

Reordering rows with arrange()

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. This is in ascending order.

`arrange()` will break ties using additionally supplied columns for ordering. It will also mostly ignore grouping. To order by group, use `.by_group = TRUE`.

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   9
 2 ENSG0000… untrt A2M        hg38       TRUE        0.528  10.1    3
 3 ENSG0000… untrt A2M-AS1    hg38       TRUE       -0.337   0.308  2
 4 ENSG0000… untrt A4GALT     hg38       TRUE        0.519   5.89  24
 5 ENSG0000… untrt AAAS       hg38       TRUE       -0.0254  5.12   0
 6 ENSG0000… untrt AACS       hg38       TRUE       -0.191   4.06   5
 7 ENSG0000… untrt AADAT      hg38       TRUE       -0.642   2.67  16
 8 ENSG0000… untrt AAGAB      hg38       TRUE       -0.165   5.08   5
 9 ENSG0000… untrt AAK1       hg38       TRUE       -0.188   3.82   2
10 ENSG0000… untrt AAMDC      hg38       TRUE        0.447   2.42   8
```

```
# 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  1
 2 ENSG000001… untrt LRRTM2     hg38       TRUE      -4.00  1.24   1
 3 ENSG000001… untrt VASH2      hg38       TRUE      -3.95  0.0171 1
 4 ENSG000001… untrt VCAM1      hg38       TRUE      -3.66  4.60   5
 5 ENSG000001… untrt SLC14A1    hg38       TRUE      -3.63  1.38    
 6 ENSG000002… untrt FER1L6     hg38       TRUE      -3.13  3.53   2
 7 ENSG000001… untrt SMTNL2     hg38       TRUE      -3.12  1.46   1
 8 ENSG000001… untrt WNT2       hg38       TRUE      -3.07  3.99   5
 9 ENSG000001… untrt EGR2       hg38       TRUE      -3.04 -0.141   
10 ENSG000001… untrt SLITRK6    hg38       TRUE      -3.03  1.16   1
# ℹ 15,916 more rows
# ℹ 1 more variable: FDR <dbl>
```

What if we want to arrange from largest to smallest (in descending order)? 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    5
 2 ENSG00000… untrt ZBTB16     hg38       TRUE       7.15  4.15   14
 3 ENSG00000… untrt <NA>       <NA>       TRUE       6.17  1.35    3
 4 ENSG00000… untrt ANGPTL7    hg38       TRUE       5.68  3.51    4
 5 ENSG00000… untrt STEAP4     hg38       TRUE       5.22  3.66    4
 6 ENSG00000… untrt PRODH      hg38       TRUE       4.85  1.29    2
 7 ENSG00000… untrt FAM107A    hg38       TRUE       4.74  2.78    6
 8 ENSG00000… untrt LGI3       hg38       TRUE       4.68 -0.0503  1
 9 ENSG00000… untrt SPARCL1    hg38       TRUE       4.56  5.53    7
10 ENSG00000… untrt KLF15      hg38       TRUE       4.48  4.69    4
# ℹ 15,916 more rows
# ℹ 1 more variable: FDR <dbl>
```

Note

If you include more than one column to order by descending values, each column needs to be wrapped with `desc()`.

Additional useful functions

- `distinct()` - return distinct combinations of values

```
acount_smeta %>% distinct(Sample)
```

```
# A tibble: 8 × 1
  Sample
  <chr>
1 SRR1039508
2 SRR1039509
3 SRR1039512
4 SRR1039513
5 SRR1039516
6 SRR1039517
7 SRR1039520
8 SRR1039521
```

- `n_distinct()` - "counts the number of unique/distinct combinations in a set of **one or more vectors**."

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 Suzan Baert's dplyr tutorials *(https://github.com/suzanbaert/Dplyr_Tutorials)* and Allison Horst's tutorial "Wrangling penguins: some basic data wrangling in R with dplyr" *(https://allisonhorst.shinyapps.io/dplyr-learnr/#section-dplyrgroup_by-summarize)*.

Joining and Transforming Data with dplyr

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()`

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

Loading Tidyverse

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 ──────────────────────── tidyver
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.5.2     ✔ tibble    3.3.0
✔ lubridate 1.9.4     ✔ tidyr     1.3.1
✔ purrr     1.0.4
── Conflicts ────────────────────────────────────────── tidyverse_co
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to for
```

Load the 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 derived from the airway RNA-Seq project.

Get the sample metadata:

```
#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, BioS
dbl (1): avgLength

ℹ Use `spec()` to retrieve the full column specification for this da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

Get the raw counts:

```
#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, SR
SRR1039...
ℹ Use `spec()` to retrieve the full column specification for this da
Specify the column types or set `show_col_types = FALSE` to quiet th
• `` -> `...1`
```

Get the differential expression results:

```
#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 da
ℹ Specify the column types or set `show_col_types = FALSE` to quiet
```

Joining data frames

Any given project will often include multiple sets of data from different sources. These related data are generally 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, which is useful depending on your analysis goal(s).

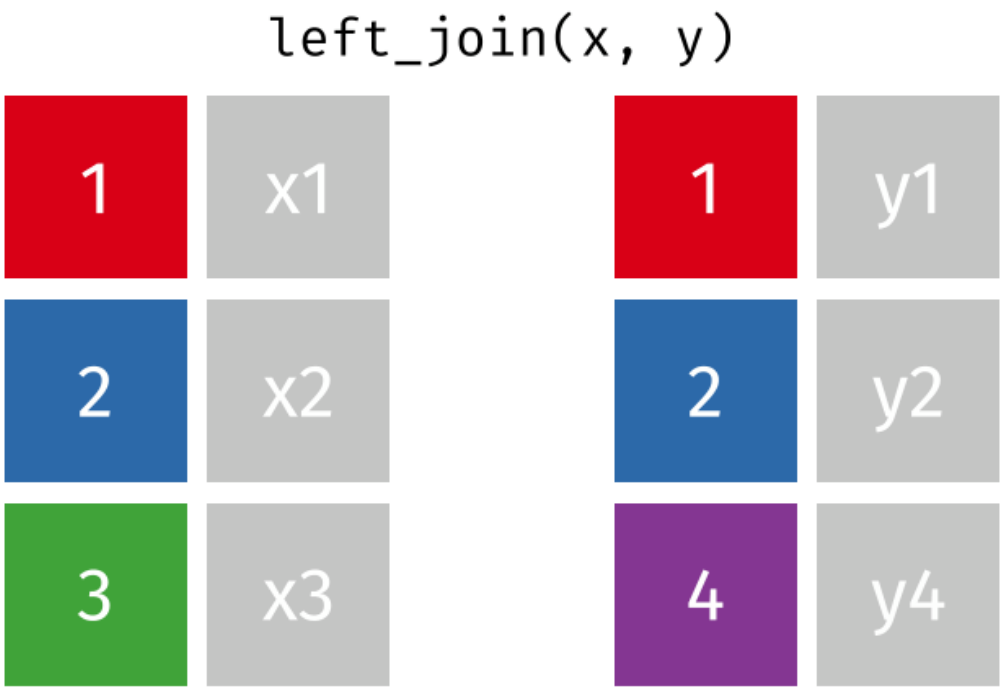
There are a series of functions from `dplyr` devoted to the purpose of joining data frames. There are two types of joins: mutating joins *(https://dplyr.tidyverse.org/reference/mutate-joins.html)* and filtering joins *(https://dplyr.tidyverse.org/reference/filter-joins.html)*.

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`

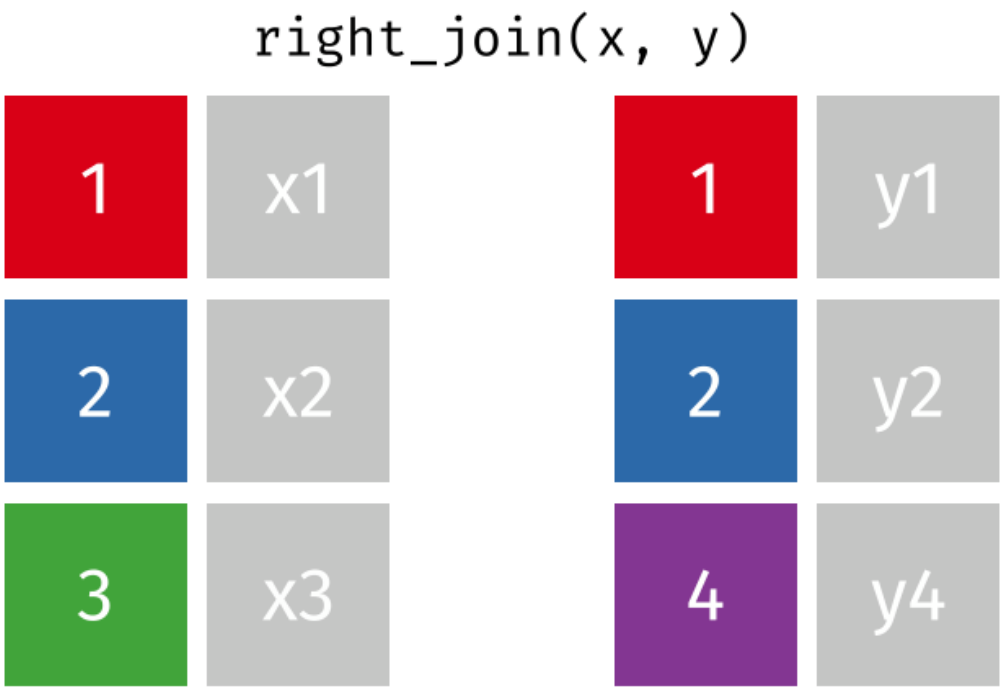
> return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*



Animation from Tidyexplain, Garrick Aden-Buie *(https://github.com/gadenbuie/tidyexplain)*

`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)*.

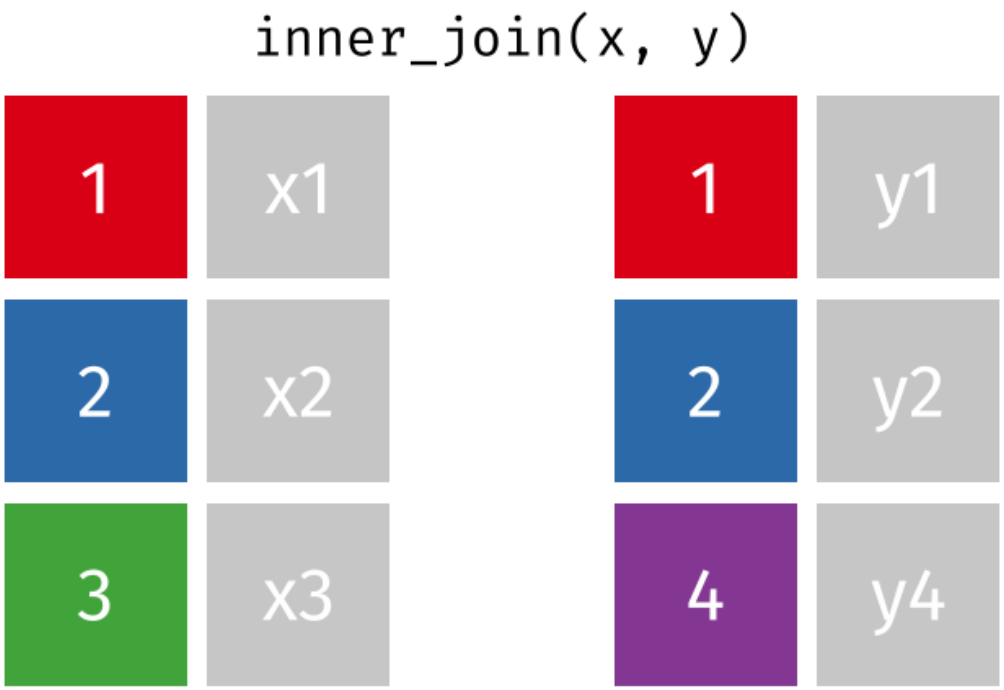


Animation from Tidyexplain, Garrick Aden-Buie *(https://github.com/gadenbuie/tidyexplain)*

`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)*

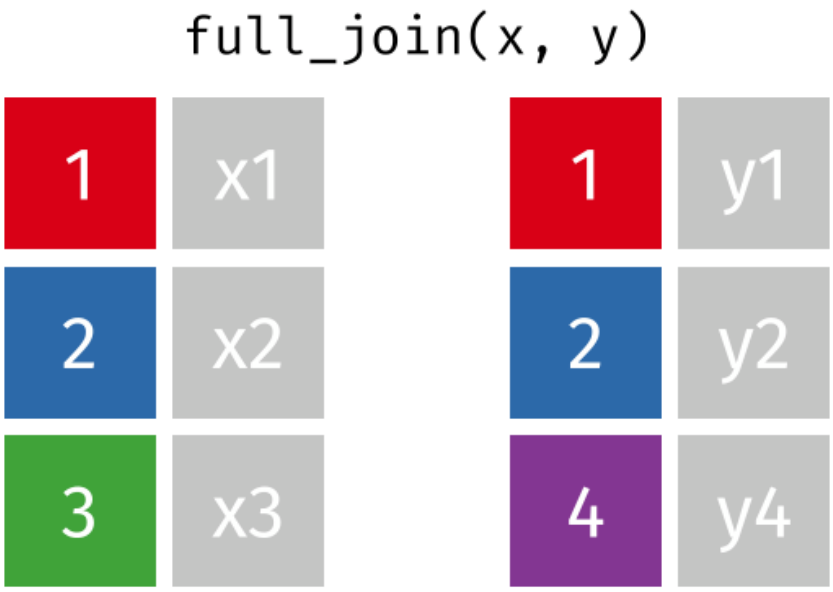
Unmatched values from x and unmatched values from y will be dropped. So use caution, as it is easy to lose observations with an inner join.



Animation from Tidyexplain, Garrick Aden-Buie *(https://github.com/gadenbuie/tidyexplain)*

`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)*.



Animation from Tidyexplain, Garrick Aden-Buie *(https://github.com/gadenbuie/tidyexplain)*

Note

The R documentation for dplyr was 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 avgLengtl
   <chr>           <chr>  <dbl> <chr>      <chr> <chr> <chr>     <dbl>
 1 ENSG000000000… SRR10…   679 GSM1275862 N613… untrt untrt       126
 2 ENSG000000000… SRR10…   448 GSM1275863 N613… trt   untrt       126
 3 ENSG000000000… SRR10…   873 GSM1275866 N052… untrt untrt       126
 4 ENSG000000000… SRR10…   408 GSM1275867 N052… trt   untrt        87
 5 ENSG000000000… SRR10…  1138 GSM1275870 N080… untrt untrt       120
 6 ENSG000000000… SRR10…  1047 GSM1275871 N080… trt   untrt       126
 7 ENSG000000000… SRR10…   770 GSM1275874 N061… untrt untrt       101
```

```
  8 ENSG000000000… SRR10…   572 GSM1275875 N061… trt   untrt        98
  9 ENSG000000000… SRR10…     0 GSM1275862 N613… untrt untrt       126
 10 ENSG000000000… SRR10…     0 GSM1275863 N613… trt   untrt       126
 # ℹ 512,806 more rows
 # ℹ 2 more variables: Sample.y <chr>, BioSample <chr>
```

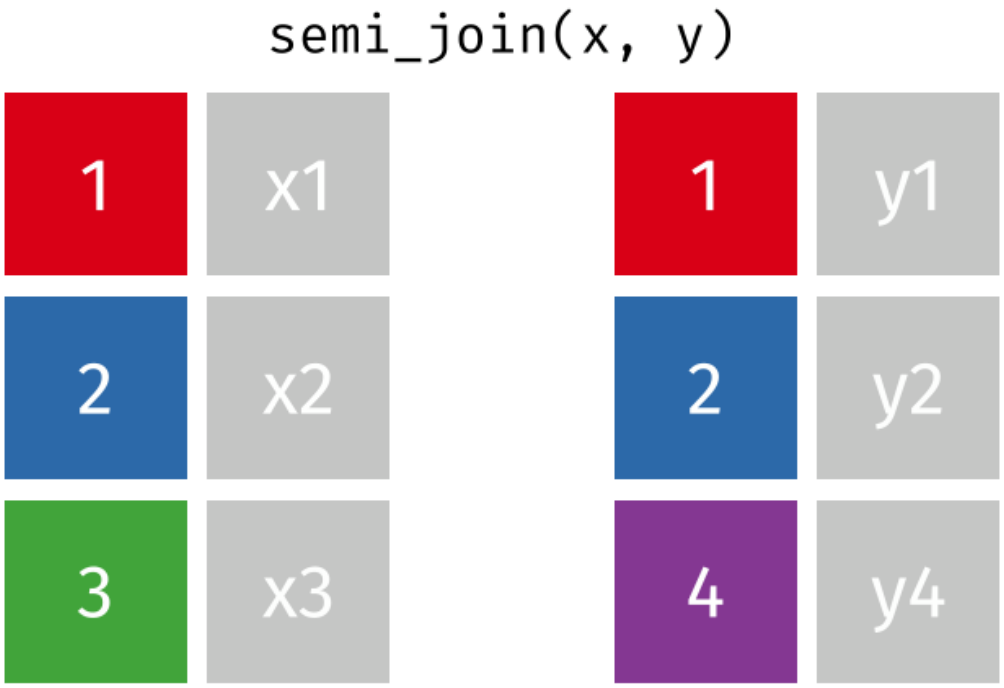
Notice the use of `by` in `left_join`. The argument `by` requires the column or columns that we want to join by. If the column we want to join by has a different name, we can use the notation above, which says to match `Sample` from `acount` to `Run` from `smeta`.

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()`

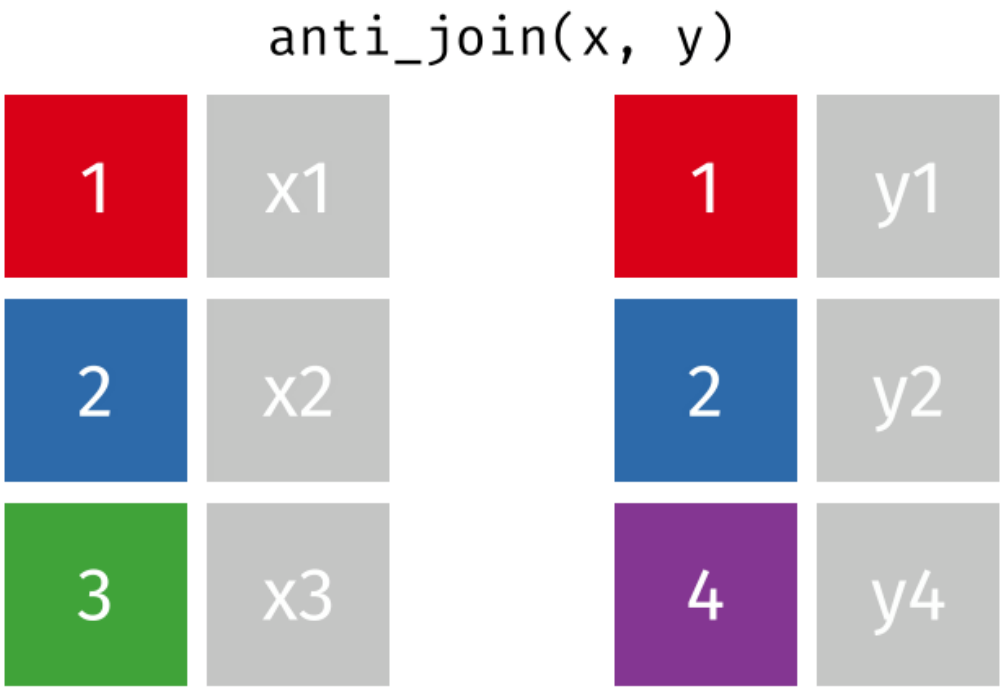
> return all rows from x where there are matching values in y, keeping just columns from x. --- R documentation, dplyr (version 0.7.8) *(https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join)*



> Animation from Tidyexplain, Garrick Aden-Buie *(https://github.com/gadenbuie/tidyexplain)*

`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)*



Animation from Tidyexplain, Garrick Aden-Buie *(https://github.com/gadenbuie/tidyexplain)*

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"))
```

```
# 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, it is easier to use a filtering join if we know we want to save elements from another table. This saves us from having to determine the filtering criteria for use with `filter()`.

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()`. `mutate()` allows us to create a new variable from existing variables.

`mutate()`

> mutate() creates new columns that are functions of existing variables. It can also modify (if the name is the same as an existing column) and delete columns (by setting their value to NULL). --- 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.

.keep

You can control which columns from the data are included in your output using `.keep`.
- `"all"` - default - keep all columns.
- `"used"` - keeps the transformed columns and new columns.

- `"unused"` - keeps only unused column and new columns.
- `"none"` - keeps the new columns and grouping variables.

Recoding variables based on values

`dplyr` offers functions for recoding variables: `if_else()` and `case_when()`.

if_else - uses two logical conditions

```
dexp_sigtrnsc2<- dexp %>%
  mutate(Significant= if_else(FDR<0.05 & abs(logFC) >=2,
                              "Significant", "Not Significant"))
```

case_when - uses multiple logical conditions. Case_when uses a series of formulas (Syntax: `logical_test ~ Value_if_True`).

```
dexp_sigtrnsc3<- dexp %>%
  mutate(Significant=
           case_when(FDR<0.05 & logFC >=2 ~ "Up",
                     FDR<0.05 & logFC <=-2 ~ "Down",
                     .default = "Not Significant")
  )
```

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

```
dexp %>% mutate(logFDR = log10(FDR), .keep="none")
```

```
# 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
```

Here, `.keep="none"` resulted in retaining only a single column ("logFDR").

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`).

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] SRR1(
   <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()`.

```
#convert character vectors to factors
ex_coerce<-acount_smeta %>% mutate(across(where(is.character),as.fac
```

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

`mutate()` works across columns, and it is not as easy to apply operations across rows for some functions (e.g., `mean`).

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 due to vectorization (Read more on vectorization in references listed here *(https://stackoverflow.com/questions/49967559/why-is-r-dplyrmutate-inconsistent-with-custom-functions)*).

For the first example to work as expected, we can 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)*.

What's next?

Now that you know the basics of working with R and the key operations to wrangle your data, it is time to learn how to visualize your data. Part 3 of this course will introduce data visualization with `ggplot2`. Stay tuned for upcoming course dates.

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://github.com/suzanbaert/Dplyr_Tutorials)*.

Introduction to Data Visualization

Introduction to Data Visualization

This course is the third and final part of a larger 3-part course designed for novices:

This course focuses on the basics of ggplot2, a tidyverse package for data visualization. Attendees will learn the building blocks needed to create publishable figures as well as tips and tricks to make plotting easier.

Lessons

1. January 6, 2026 - Introduction to ggplot2 for R Data Visualization
2. January 8, 2026 - Plot Customization with ggplot2
3. January 13, 2026 - From Data to Display: Crafting a Publishable Plot
4. January 15, 2026 - Recommendations and Tips for Creating Effective Plots with ggplot2

Prerequisites

This course is recommended for attendees familiar with the skills learned in Part 1: Getting Started with R *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting_Started_with_R/)*. Attendees will also benefit from skills learned in Part2: Introduction to Data Wrangling *(https://bioinformatics.ccr.cancer.gov/btep/courses/introductory-r-for-novices-introduction-to-data-wrangling)*.

Course materials

We will use R on Biowulf for this course to avoid issues with R and package installations. To use R on Biowulf, you must have a NIH HPC account. If you do not have a NIH HPC (Biowulf) account, this course can be taken using a local R installation. However, we will not be able to troubleshoot package installation issues during class. Additionally, because we will use packages belonging to the tidyverse *(https://www.tidyverse.org/)*, you will need to install these packages using `install.packages("tidyverse")` prior to the first lesson if you are not using R on Biowulf.

Get the Data

The data used in this course can be downloaded here. To use these files, you should unzip data.zip and add it to your working directory.

Introduction to ggplot2 for R Data Visualization

Learning Objectives

1. Identify and describe the core components of a ggplot2 plot, including data, aesthetics, and geometric layers.
2. Learn the grammar of graphics for plot construction.
3. Construct basic plots in ggplot2 by mapping variables to aesthetics and adding simple geometric layers.

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here *(https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

Why use R for Data Visualization?

Learning R and associated plotting packages is a great way to generate publishable figures in a reproducible fashion.

With R you can:
1. Create simple or complex figures.
2. Create high resolution figures.
3. Generate scripts that can be reused to create the same or similar plot.

Why not use Excel for data visualization?

Excel is a great program for managing data in a spreadsheet. However, it isn't great for working with "big data". Large data sets are difficult to work with, and resulting plots are generally not publishable due to a low resolution. Learning R and associated plotting packages is a great way to generate publishable figures in a reproducible fashion. Using R will not only keep you from accidentally editing your data, but it will also allow you to generate scripts that can be viewed later or reused to generate the same plot using different data. This will keep you from having to rely on your memory when wondering what data was used or how a plot was generated.

`ggplot2` is an R graphics package from the tidyverse collection. It 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). 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 frames (tidy data), which you should already be accustomed to.

To begin plotting, let's load our `tidyverse` library.

```
#load libraries
library(tidyverse) # Tidyverse automatically loads ggplot2
```

```
-- Attaching core tidyverse packages ----------------------- tidyver
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.2     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.4
-- Conflicts ------------------------------------------- tidyverse_con
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to forc
```

Example Data

We also need some data to plot, so if you haven't already, let's load the data we will need for this lesson.

Getting the Data

If you have not already done so, please download the data for this course from here and unzip it to your working directory.

If you are using RStudio on Biowulf, you can use the following steps to download and unzip the data directly to your working directory.

- Open the "Terminal" in RStudio (See the tab next to "Console").
- Make sure you are in your working directory. You can check this by typing `pwd` and hitting enter. If you are not in your working directory, you can change to it using the `cd` command. For example, if your working directory is `/data/username/`, you would type `cd /data/username/` and hit enter.
- Download the data using the `wget` command:

```
wget https://bioinformatics.ccr.cancer.gov/docs/r_for_novices/Data_Visualizat
```

- Unzip the data using the `unzip` command:

```
unzip data.zip
```

Alternatively, you can download the data to your local machine and then upload it to your working directory in RStudio using the "Upload" button in the "Files" tab.

```
#scaled_counts data
scaled_counts<-
  read_delim("./data/filtlowabund_scaledcounts_airways.txt")
```

```
Rows: 127408 Columns: 18
-- Column specification --------------------------------------------
Delimiter: "\t"
chr (11): feature, SampleName, cell, dex, albut, Run, Experiment, Sar
dbl  (6): sample, counts, avgLength, TMM, multiplier, counts_scaled
lgl  (1): .abundant

i Use `spec()` to retrieve the full column specification for this dat
i Specify the column types or set `show_col_types = FALSE` to quiet t
```

```
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 dat
i Specify the column types or set `show_col_types = FALSE` to quiet t
```

The example data we will use for today's lesson were generated from data available in the Bioconductor package `airway` *(https://bioconductor.org/packages/release/data/experiment/html/airway.html)*, which "provides a RangedSummarizedExperiment object of read counts in genes for an RNA-Seq experiment on four human airway smooth muscle cell lines treated with dexamethasone" *(https://bioconductor.org/packages/release/data/experiment/html/airway.html)* and reported in Himes et al. (2014) *(https://pubmed.ncbi.nlm.nih.gov/24926665/)*.

In this experiment, the authors compared transcriptomic differences in primary human airway smooth muscle cell lines 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.

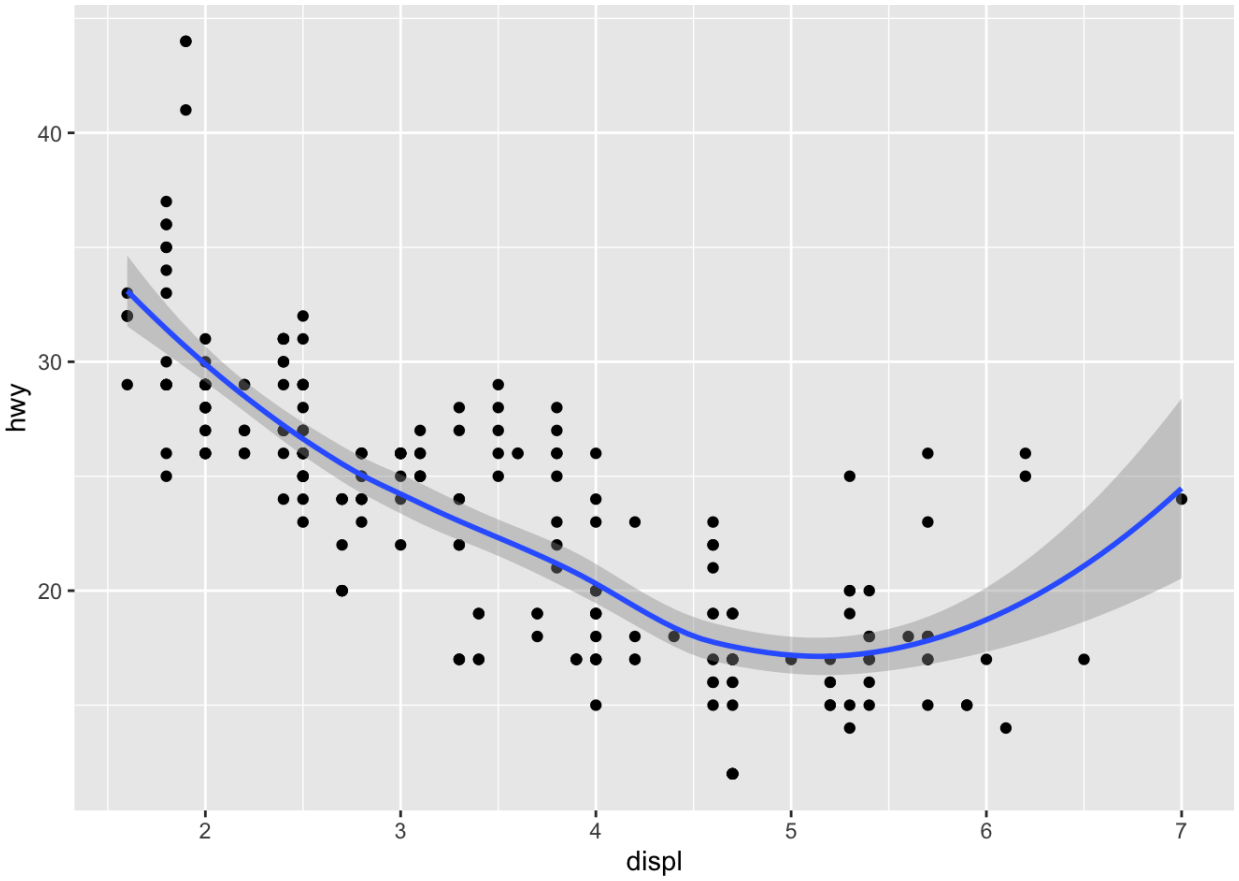
Practice Data

There are a number of built-in data sets available for practicing with ggplot2. Check these out here *(https://ggplot2.tidyverse.org/reference/#data)*!

For example, `mtcars` is commonly used in ggplot2 documentation:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth()
```

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



Occasionally, I will pull in practice data to demonstrate specific aspects of `ggplot2`.

The ggplot2 template

The following represents the basic ggplot2 template.

```

```

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

We need three basic components to create a plot:

- **data we want to plot**
- **geom function(s)**
- **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.

We will examine the relationship between the total transcript sums per sample (total reads) and the number of recovered transcripts per sample.

We can generate these data using

```
sc <- scaled_counts |> group_by(dex, SampleName) |>
  summarize(Num_transcripts=sum(counts>100),TotalCounts=sum(counts))
```

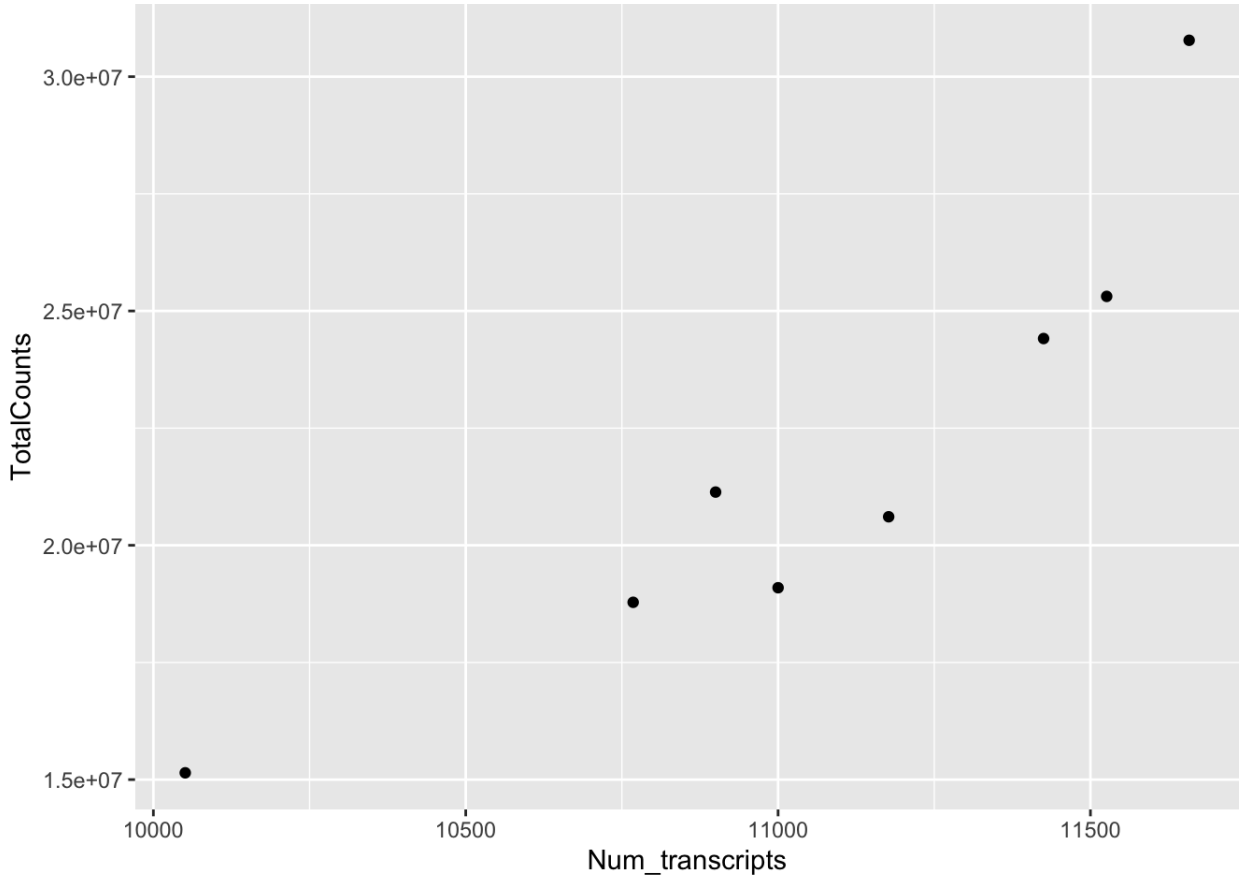
```
`summarise()` has grouped output by 'dex'. You can override using the
argument.
```

```
sc
```

```
# A tibble: 8 x 4
# Groups:   dex [2]
  dex   SampleName Num_transcripts TotalCounts
  <chr> <chr>                <int>       <dbl>
1 trt   GSM1275863           10768    18783120
2 trt   GSM1275867           10051    15144524
3 trt   GSM1275871           11658    30776089
4 trt   GSM1275875           10900    21135511
5 untrt GSM1275862           11177    20608402
6 untrt GSM1275866           11526    25311320
7 untrt GSM1275870           11425    24411867
8 untrt GSM1275874           11000    19094104
```

Let's plot

```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts))
```



We can easily see that there is a relationship between the number of reads 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.

What did we do in the above code?

The first step to 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 these need 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 `sc`, 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.

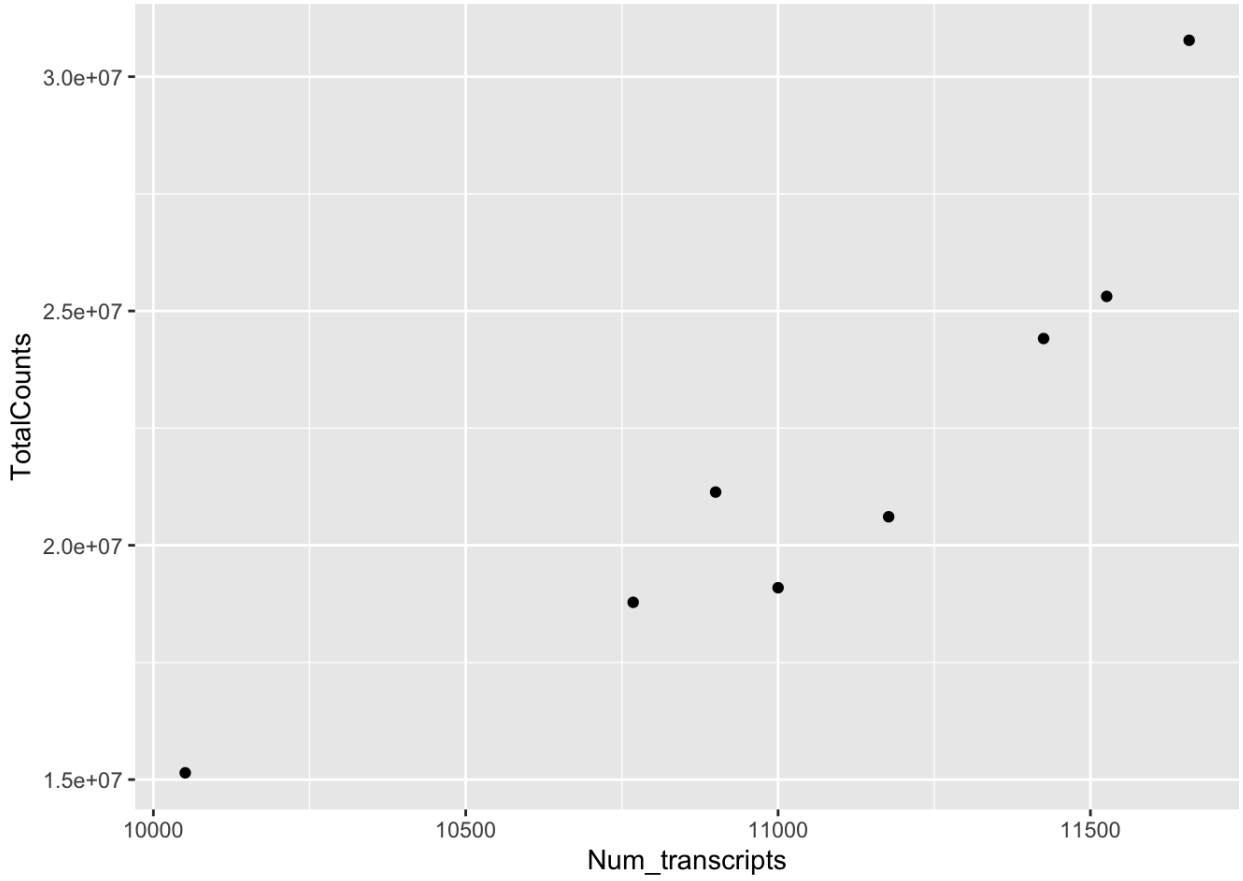
```
#What does running ggplot() do?
ggplot(data=sc)
```



```
#What about just running a geom function?
geom_point(data=sc,aes(x=Num_transcripts, y = TotalCounts))
```

```
mapping: x = ~Num_transcripts, y = ~TotalCounts
geom_point: na.rm = FALSE
stat_identity: na.rm = FALSE
position_identity
```

```
#what about this
ggplot() +
geom_point(data=sc,aes(x=Num_transcripts, y = TotalCounts))
```



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.

- 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()`) (Tip: Use `ComplexHeatmap` or `pheatmap`),
- geographic maps (`geom_polygon()`), etc.

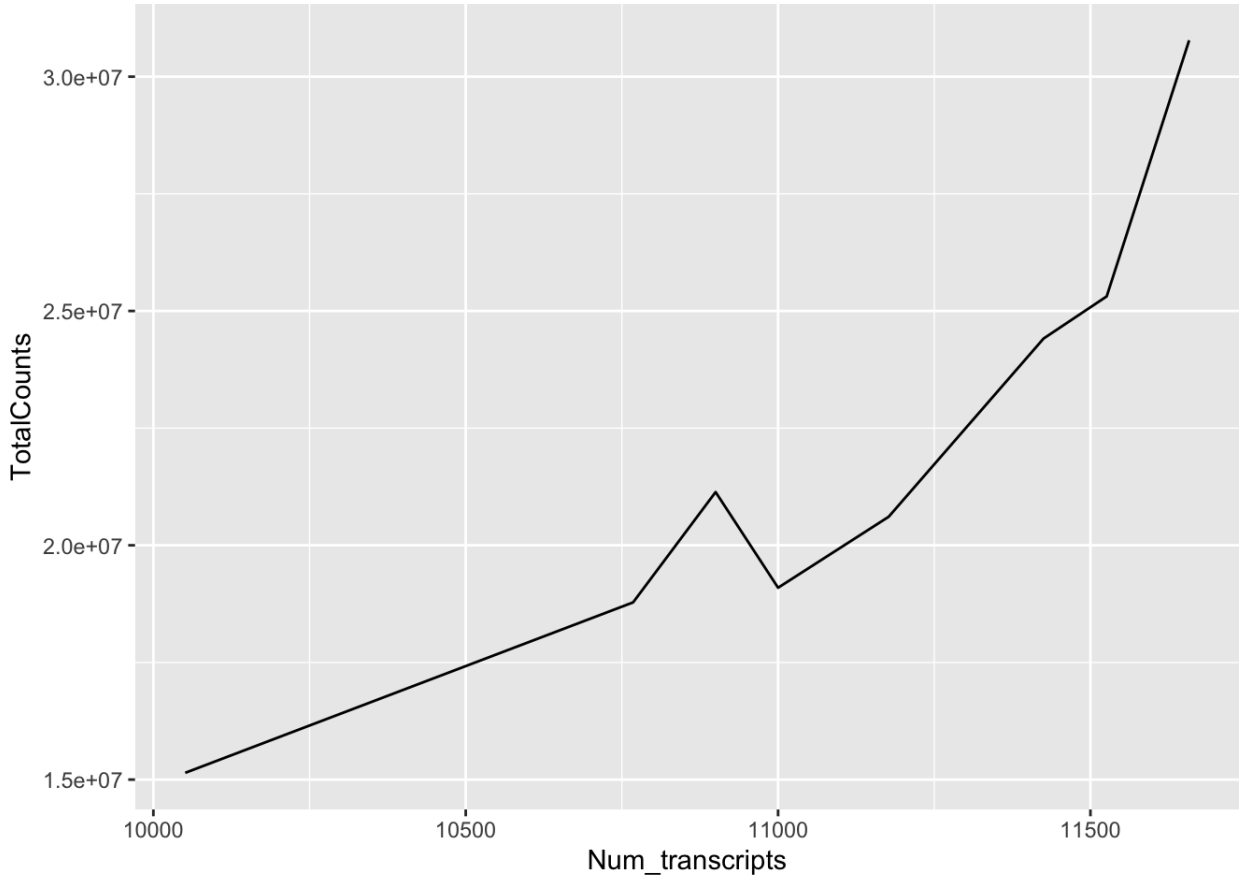
> ggplot2 provides over 40 geoms, and extension packages provide even more (see https://exts.ggplot2.tidyverse.org/gallery/ *(https://exts.ggplot2.tidyverse.org/gallery/)* for a sampling). The best way to get a comprehensive overview is the ggplot2 cheatsheet, which you can find at https://posit.co/resources/cheatsheets/ *(https://posit.co/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. For more detailed reference pages and examples, see the ggplot2 website reference pages *(https://ggplot2.tidyverse.org/reference/index.html)*.

Create a line plot

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=sc) +
  geom_line(aes(x=Num_transcripts, y = TotalCounts))
```

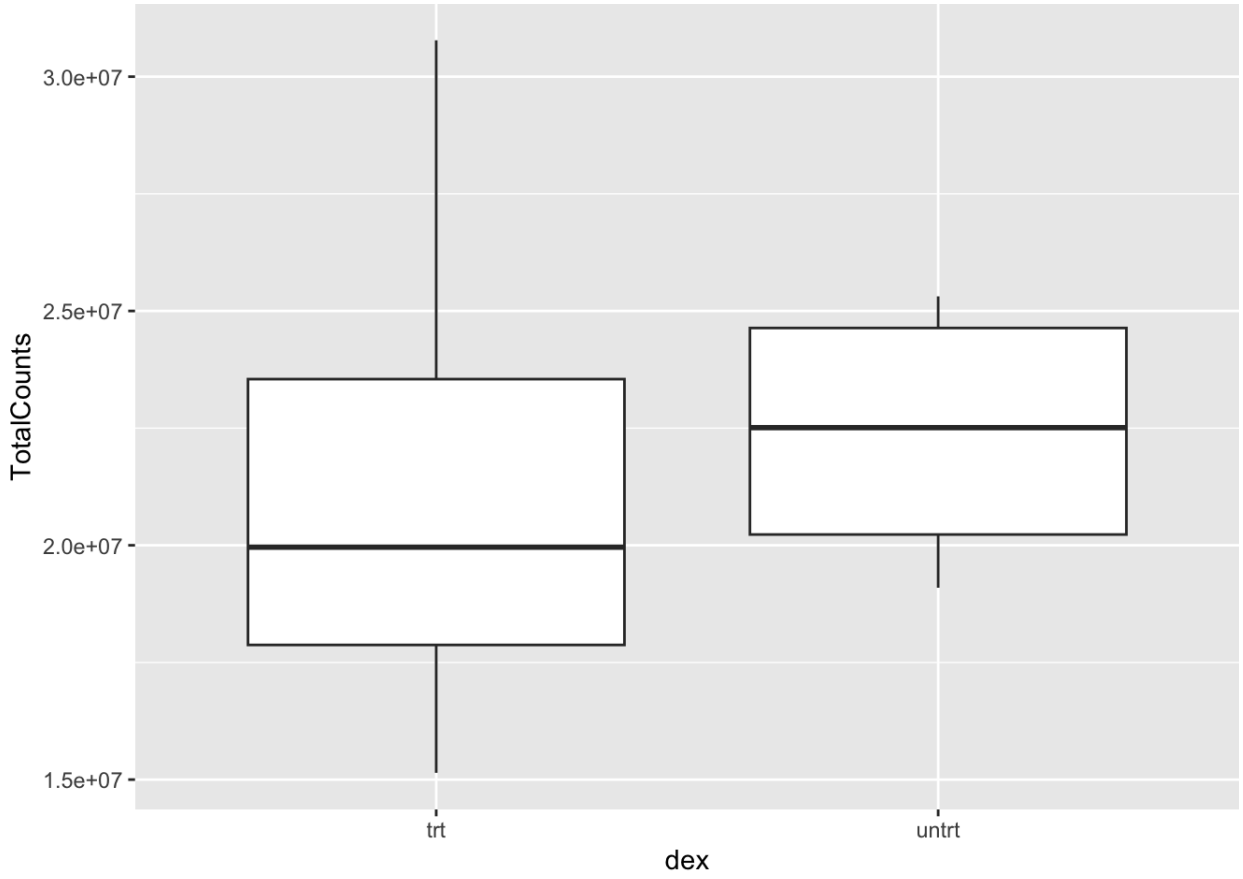


Create a box plot

Let's plot the same data using `geom_boxplot()`. A boxplot *(https://www.data-to-viz.com/caveat/boxplot.html)* can be used to summarize the distribution of a numeric variable across groups.

```
ggplot(data=sc) +
```

```
  geom_boxplot(aes(x=dex, y = TotalCounts))
```



Note

This time we also modified the x argument.

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

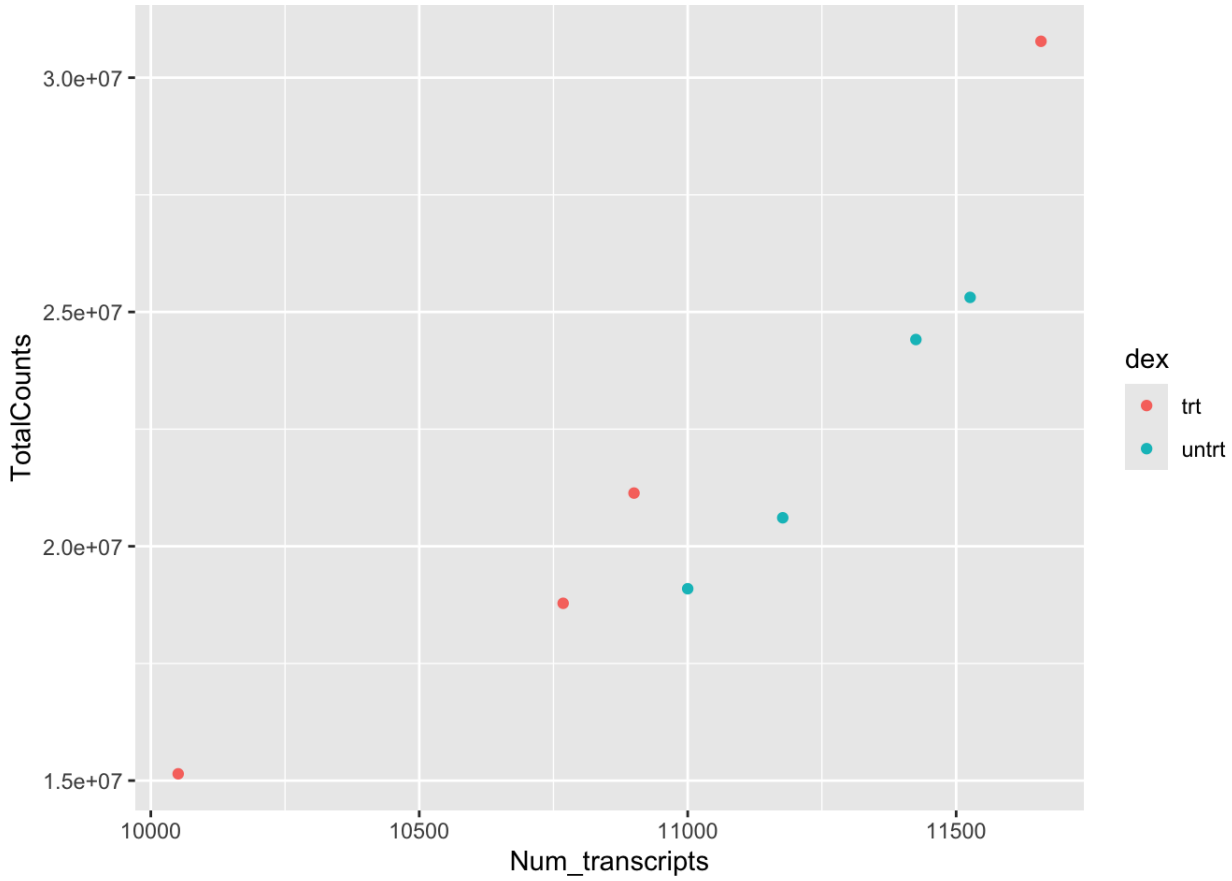
6. linetype
7. alpha

This is not an all encompassing list. You can add multiple aesthetics to a plot to represent different variables.

Map a Color to a Variable

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(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,color=dex))
```



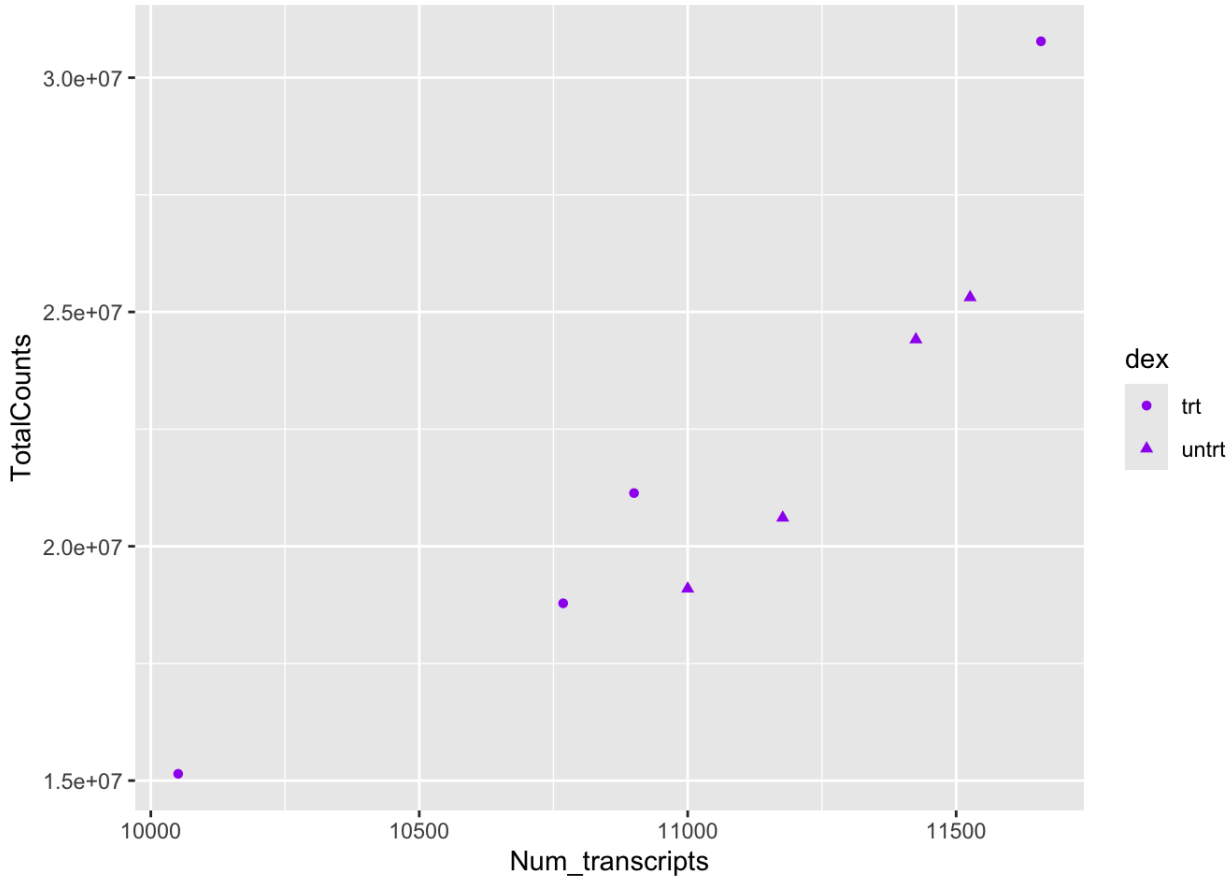
There is potentially a relationship. ASM cells treated with dexamethasone in general have lower total numbers of transcripts and lower total counts.

Notice how we changed the color of our points to represent a variable, in this case. To do this, we set color equal to 'dex' within the `aes()` function. This mapped our aesthetic, color, to a variable we were interested in exploring ("dex"). **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(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,shape=dex),
             color="purple")
```

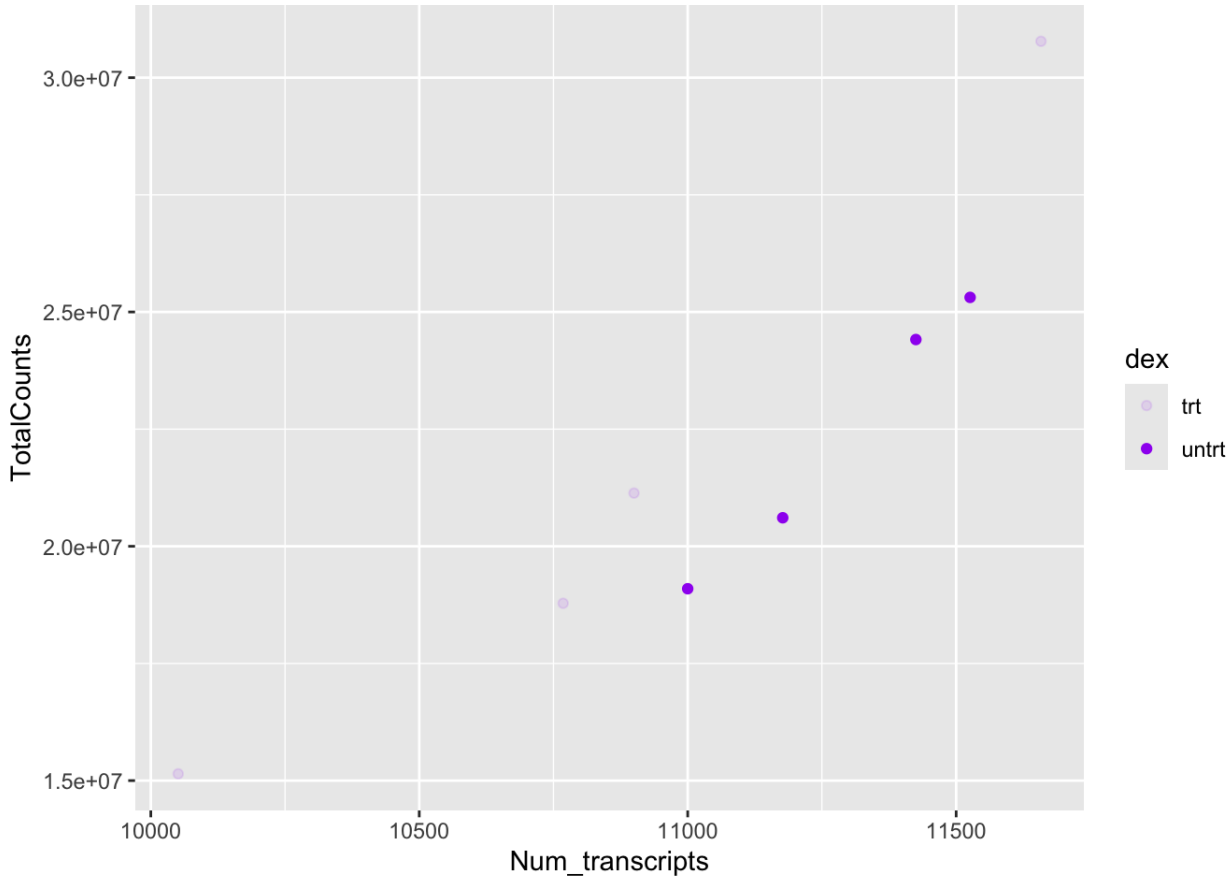


We can also see from this that 'dex' could be mapped to other aesthetics. In the above example, we see it mapped to shape rather than color. **By default, ggplot2 will only map six shapes at a time, and if your number of categories goes beyond 6, the remaining groups will go unmapped.** This is by design 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.htmls)*.

We could have just as easily mapped it to alpha, which adds a gradient to the point visibility by category.

```
#map the alpha aesthetic to the variable "dex"
#use the color purple across all points (NOT mapped to a variable)
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,alpha=dex),
             color="purple") #note the warning.
```

```
Warning: Using alpha for a discrete variable is not advised.
```



Or we could map it to size. There are multiple options, so feel free to explore a little with your plots.

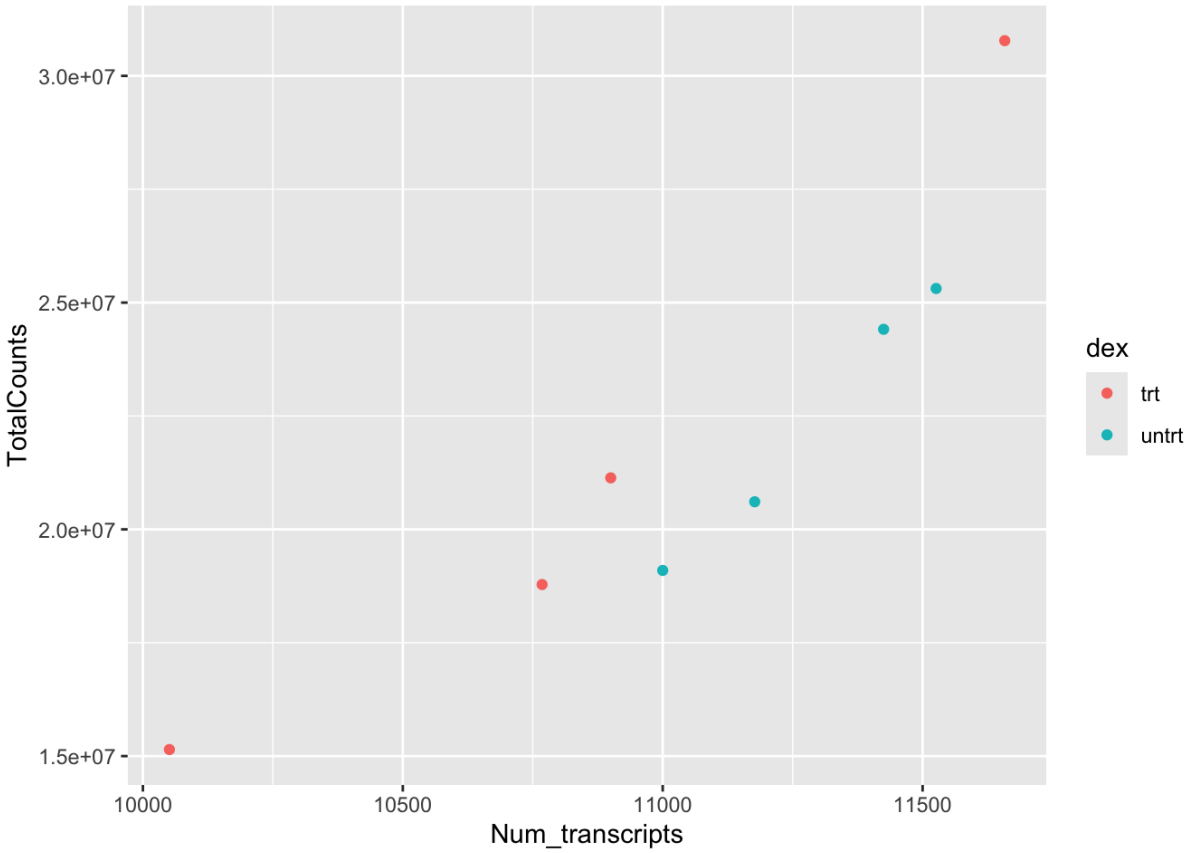
Defaults

Notice that 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. Most of what we see on this plot is auto generated with defaults (e.g., Assigned colors, legend, axis titles, plot background, tick marks and labels) and we can change these defaults, for example, what colors are used, by adding additional layers to our code.

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 created with `ggplot2`.

```
scatter_plot<-ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,
                 color=dex))

scatter_plot
```



We can add additional layers directly to our object.

How can we modify colors?

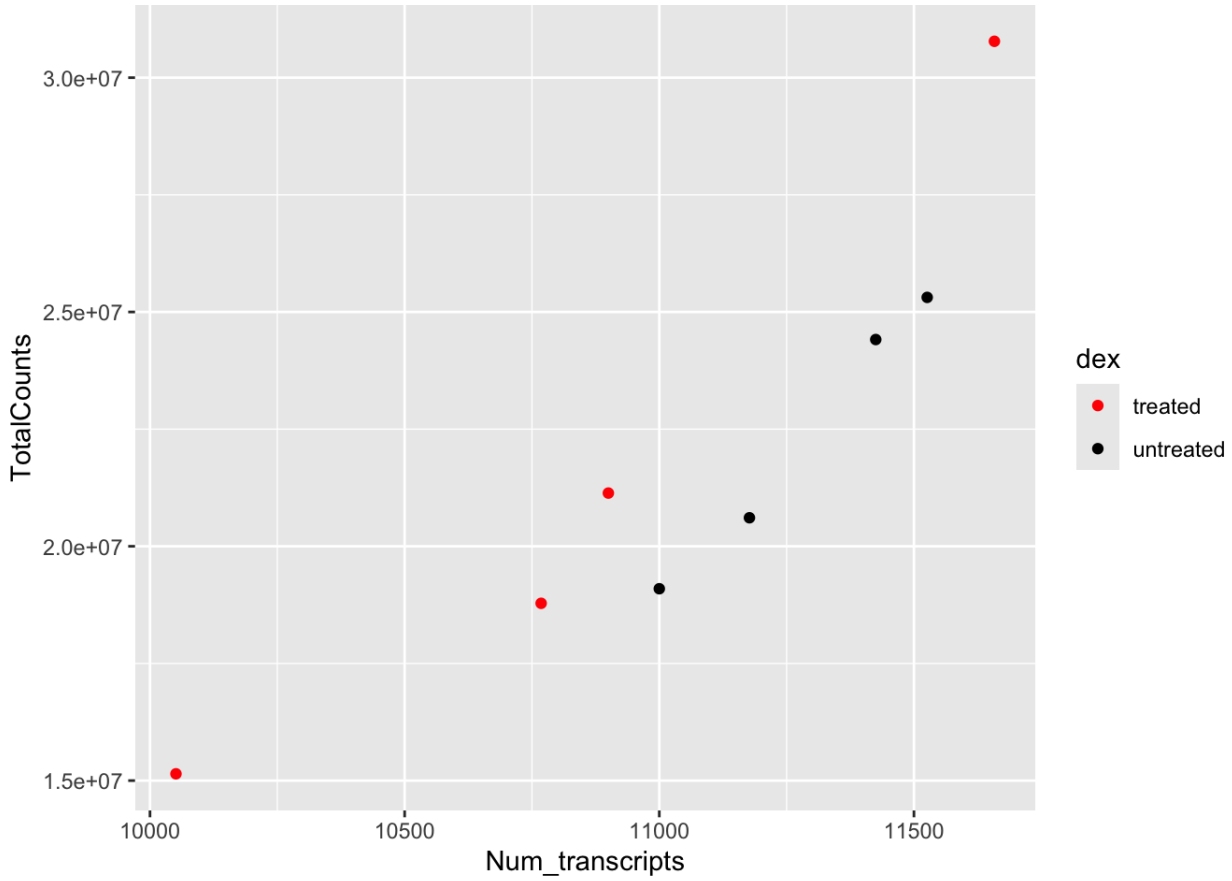
Colors are assigned to the fill and color aesthetics in `aes()`. We can change the default colors by providing an additional layer to our figure. To change the color, we use the `scale_color` functions:

- `scale_color_manual()`,
- `scale_color_brewer()` *(https://r-graph-gallery.com/38-rcolorbrewers-palettes.html)*,
- `scale_color_grey()`, etc.

Example:

```
ggplot(sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,
```

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



Similarly, if we want to change the `fill`, we would use the `scale_fill` options. To modify shapes, use `scale_shape` options.

Additional arguments

We can modify the behavior of any function by adding additional arguments (if available). Here we changed the color labels in the legend using the `labels` argument. The labels must be in the correct order. You do not want to mislabel the legend.

Order of Categorical Variables

By default, `ggplot2` will alphabetize categorical variables. If you want to change the order of a categorical variable, you can do so by converting the variable to a factor and specifying the levels in the order you want them to appear. The package `forcats` has a number of functions to help you work with factors. See the forcats documentation *(https://forcats.tidyverse.org/)* for more information.

More on Colors

There are a number of ways to specify the color argument including by name, number, and hex code. Here *(https://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.

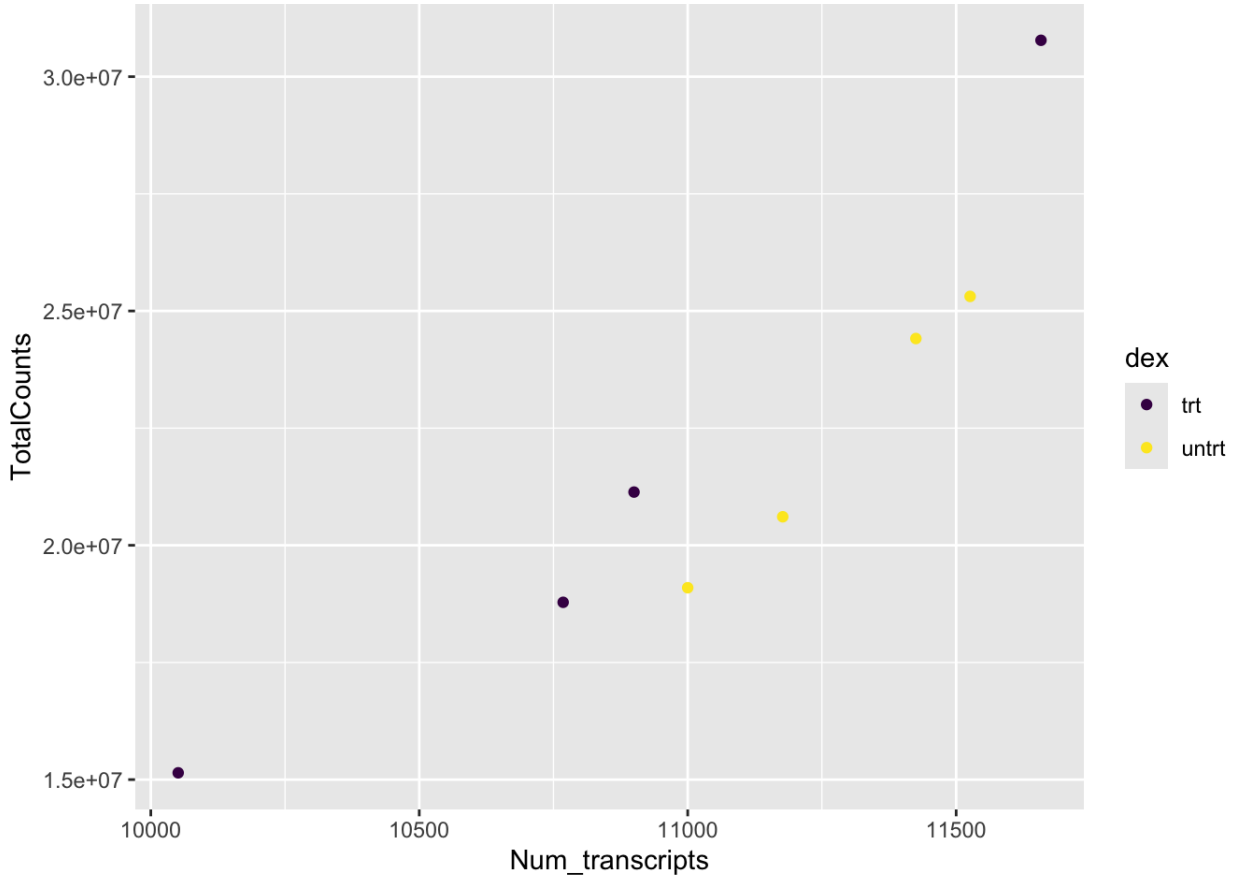
There are also a number of complementary packages in R that expand our color options.

- `viridis` *(https://cran.r-project.org/web/packages/viridis/index.html)* - provides colorblind friendly palettes.
- `randomcoloR` *(https://cran.r-project.org/web/packages/randomcoloR/index.html)* - generates large numbers of random colors.
- `Paletteer` *(https://github.com/EmilHvitfeldt/paletteer)* - contains a comprehensive set of color palettes to load the palettes from multiple packages all at once.

```
library(viridis)
```

```
Loading required package: viridisLite
```

```
ggplot(sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,
                 color=dex)) +
scale_color_viridis(discrete=TRUE, option="viridis")
```



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). In newer versions of ggplot2, you can additionally use `vars()` to select variables for faceting. See `?facet_wrap()` for more information.

Using ~ in ggplot2

The ~ is used in R formulas to split the dependent or response variable from the independent variable(s). For more information, see this explanation here. *(https://medium.com/anu-perumalsamy/what-does-mean-in-r-18cecd1b223f#:~:text='~(tilde)'%20is%20an%20operator%20that%20splits%20the%20left,the%20set%20of%20fea {target=_blank})*

In `facet_wrap()` / `facet_grid()` the ~ is used to generate a formula specifying rows by columns.

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.

Let's import and examine the data with `head()`.

```
density_data<-read.csv("./data/density_data.csv",
                       stringsAsFactors=TRUE)

head(density_data)
```

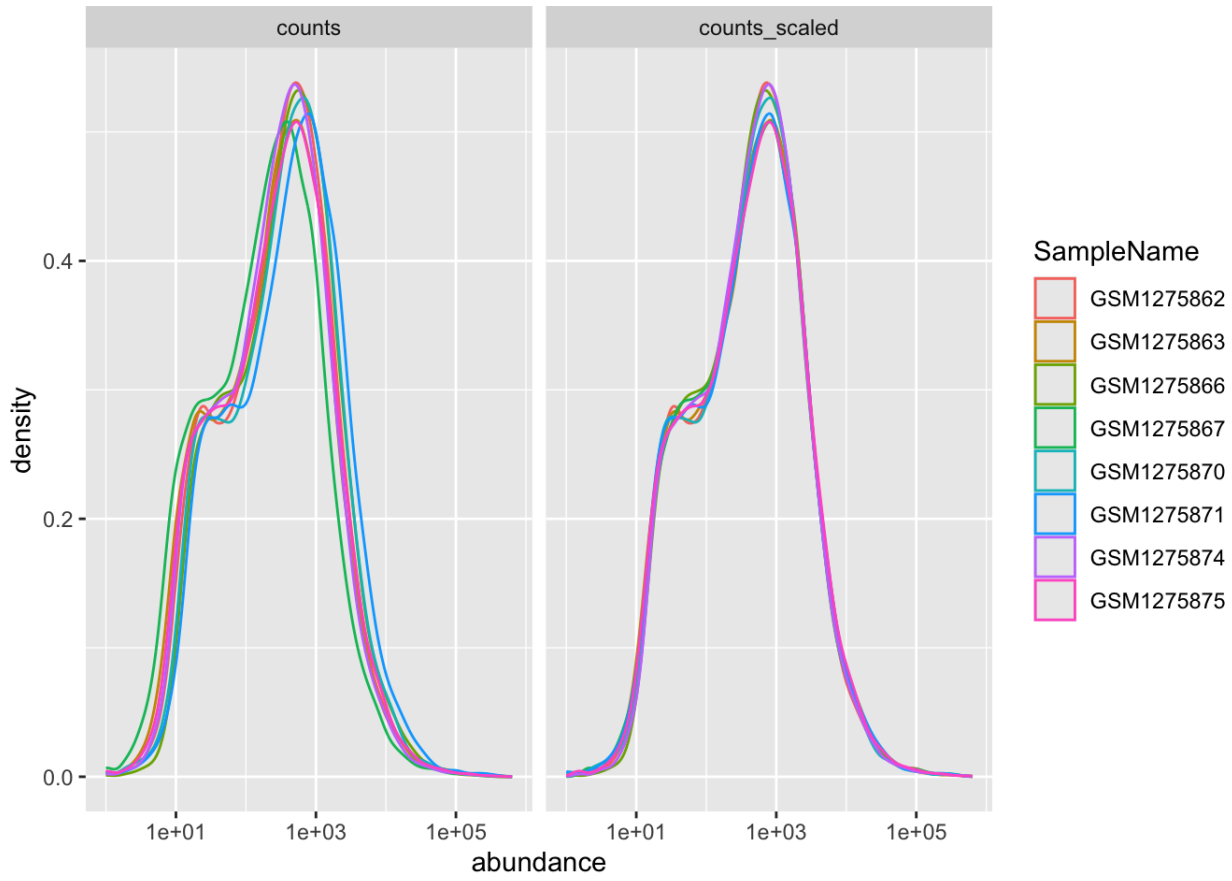
```
          feature sample SampleName   cell   dex albut        Run avg
1 ENSG00000000003    508 GSM1275862 N61311 untrt untrt SRR1039508
2 ENSG00000000003    508 GSM1275862 N61311 untrt untrt SRR1039508
3 ENSG00000000419    508 GSM1275862 N61311 untrt untrt SRR1039508
4 ENSG00000000419    508 GSM1275862 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 .abundant
1  SRX384345 SRS508568 SAMN02422669     TSPAN6       hg38      TRUE 1
2  SRX384345 SRS508568 SAMN02422669     TSPAN6       hg38      TRUE 1
3  SRX384345 SRS508568 SAMN02422669       DPM1       hg38      TRUE 1
4  SRX384345 SRS508568 SAMN02422669       DPM1       hg38      TRUE 1
5  SRX384345 SRS508568 SAMN02422669      SCYL3       hg38      TRUE 1
6  SRX384345 SRS508568 SAMN02422669      SCYL3       hg38      TRUE 1
  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
```

Notice the `source` column, which indicates whether the counts are scaled or unscaled. These data are in long vs wide format. You may need to reshape the data to represent the information in a specific way with ggplot2. Here, we can use this variable to facet our density plot.

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

```
Warning in scale_x_log10(): log-10 transformation introduced infinite
```

```
Warning: Removed 140 rows containing non-finite outside the scale ran
(`stat_density()`).
```



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.

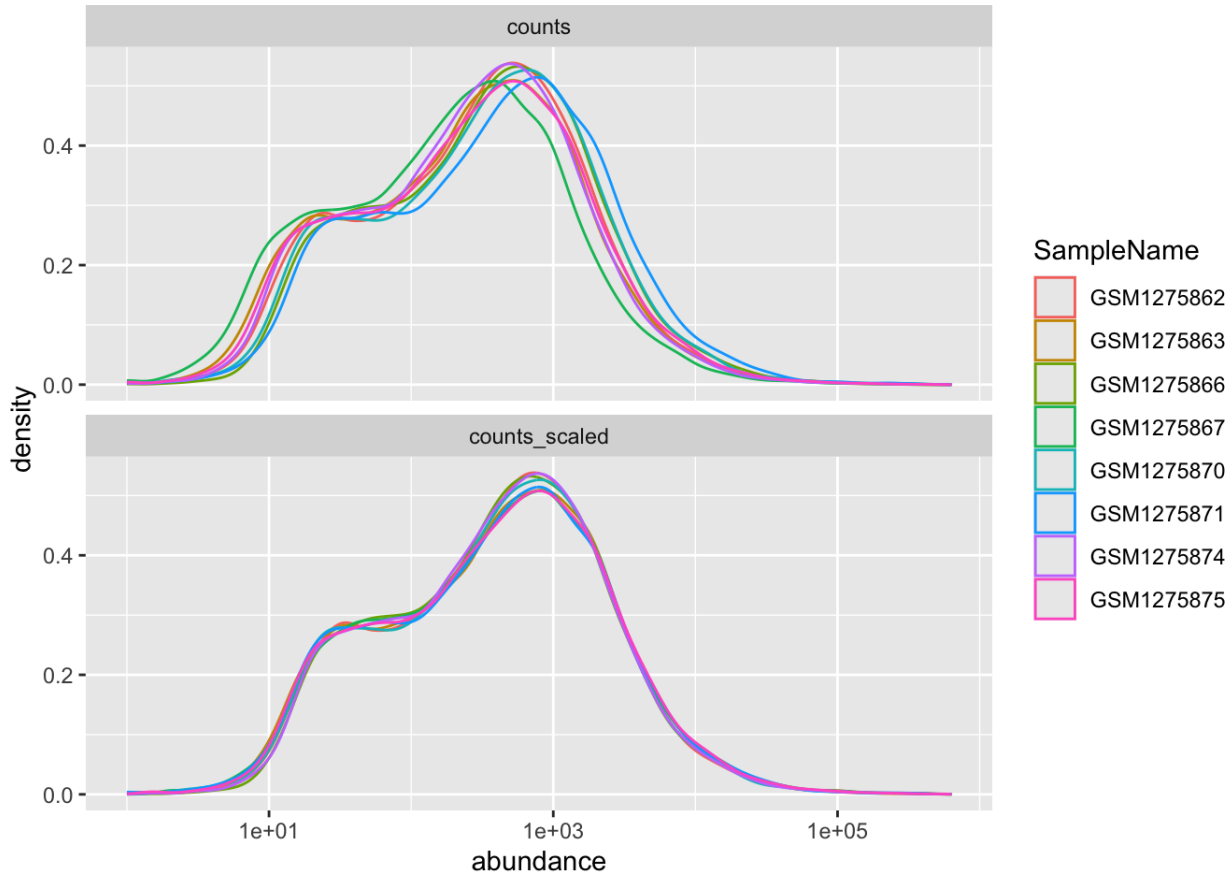
Here, faceting allowed us to visualize multiple features of our data. We were able to see count distributions by sample as well as normalized vs non-normalized counts.

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_grid(~source, ncol=1) + #use the ncol argument
  scale_x_log10()
```

```
Warning in scale_x_log10(): log-10 transformation introduced infinite
```

```
Warning: Removed 140 rows containing non-finite outside the scale ran
(`stat_density()`).
```



Building upon our template

This is the grammar of graphics. Adding layers to create unique figures.

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

Note that there are a lot of invisible (default) layers that often go into each ggplot2, and there are ways to customize these layers. See this chapter *(https://r4ds.had.co.nz/data-visualisation.html#the-layered-grammar-of-graphics)* from R for Data Science for more information on the grammar of graphics.

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, scale the axes, and customize the legends and theme *(https://ggplot2.tidyverse.org/reference/theme.html)*. We will be working with these additional plot features in the weeks to come, so stay tuned.

Resource list

1. ggplot2 cheatsheet *(https://ggplot2.tidyverse.org/index.html#cheatsheet)*
2. The R Graph Gallery *(https://www.r-graph-gallery.com/)*
3. The R Graphics Cookbook *(https://r-graphics.org/recipe-quick-bar)*
4. ggplot2 extensions *(https://exts.ggplot2.tidyverse.org/gallery/)*
5. From Data to Viz *(https://www.data-to-viz.com/)*
6. Other Resources *(https://ggplot2.tidyverse.org/index.html#learning-ggplot2)*
7. ggplot2: Elegant Graphics for Data Analysis *(https://ggplot2-book.org/index.html)*

Acknowledgements

Material from this lesson was inspired by Chapter 3 of R for Data Science *(https://r4ds.had.co.nz/data-visualisation.html)* and from "Data Visualization", *Introduction to data analysis with R and Bioconductor (https://carpentries-incubator.github.io/bioc-intro/40-visualization/index.html)*, which is part of the Carpentries Incubator.

Plot Customization with ggplot2

Learning Objectives

1. Review the grammar of graphics template.
2. Understand the statistical transformations inherent to geoms.
3. Customize figures with labels, legends, scales, and themes.
4. Save plots with `ggsave()`.

Our grammar of graphics template

Last lesson we discussed the three basic components of creating a `ggplot2` plot: the **data**, **one or more geoms**, and **aesthetic mappings**.

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

But, we also learned of other features that greatly improve our figures (e.g., facets), and today we will be expanding our `ggplot2` template even further to include:

- 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**,
- **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)*)

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

Loading the libraries

To begin plotting, let's load our `tidyverse` library. This includes `ggplot2`, which we will be using for plotting.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyver
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.1     v stringr   1.5.2
v ggplot2   4.0.0     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.1.0
-- Conflicts ------------------------------------------ tidyverse_con
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to forc
```

Importing the data

We also need some data to plot, so if you haven't already, let's load the data we will need for this lesson.

```
#scaled_counts
scaled_counts <-
  read.delim("./data/filtlowabund_scaledcounts_airways.txt",
             as.is=TRUE)

#differential expression results
dexp <- read.delim("./data/diffexp_results_edger_airways.txt",
                   as.is=TRUE)


```

```
#transcript counts greater than 100
sc <- read.csv("./data/sc.csv")
```

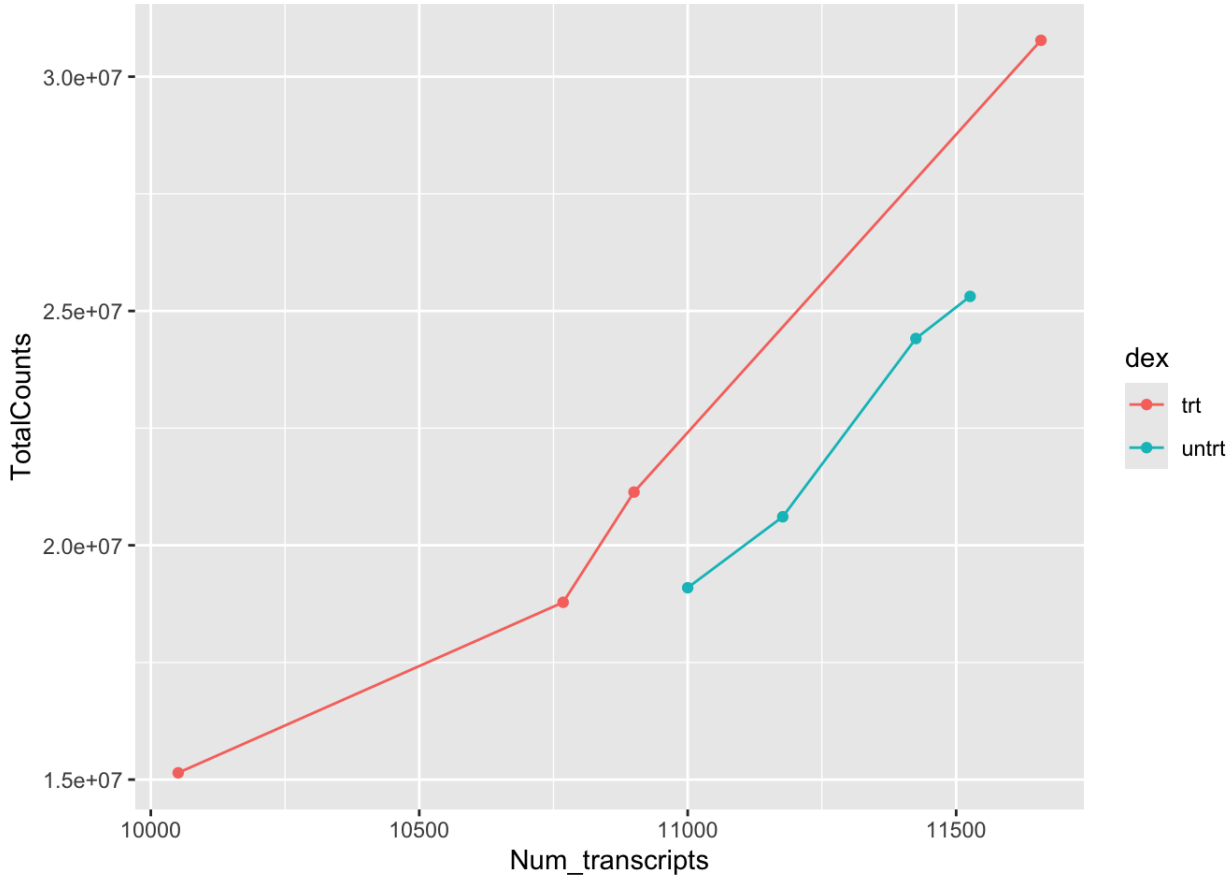
Using Multiple Geoms per Plot

In Lesson 1, we discovered that a geom, the geometrical representation of the plot, is required to create a visualization with ggplot2. This is true, but keep in mind that we can use **1 or more** geoms to build our plot.

Because we build plots using layers in ggplot2. We can add multiple geoms to a plot to represent the data in unique ways. Let's see how this works.

Let's combine a scatter plot with a line plot.

```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,color=dex)) +
  geom_line(aes(x=Num_transcripts, y = TotalCounts,color=dex))
```



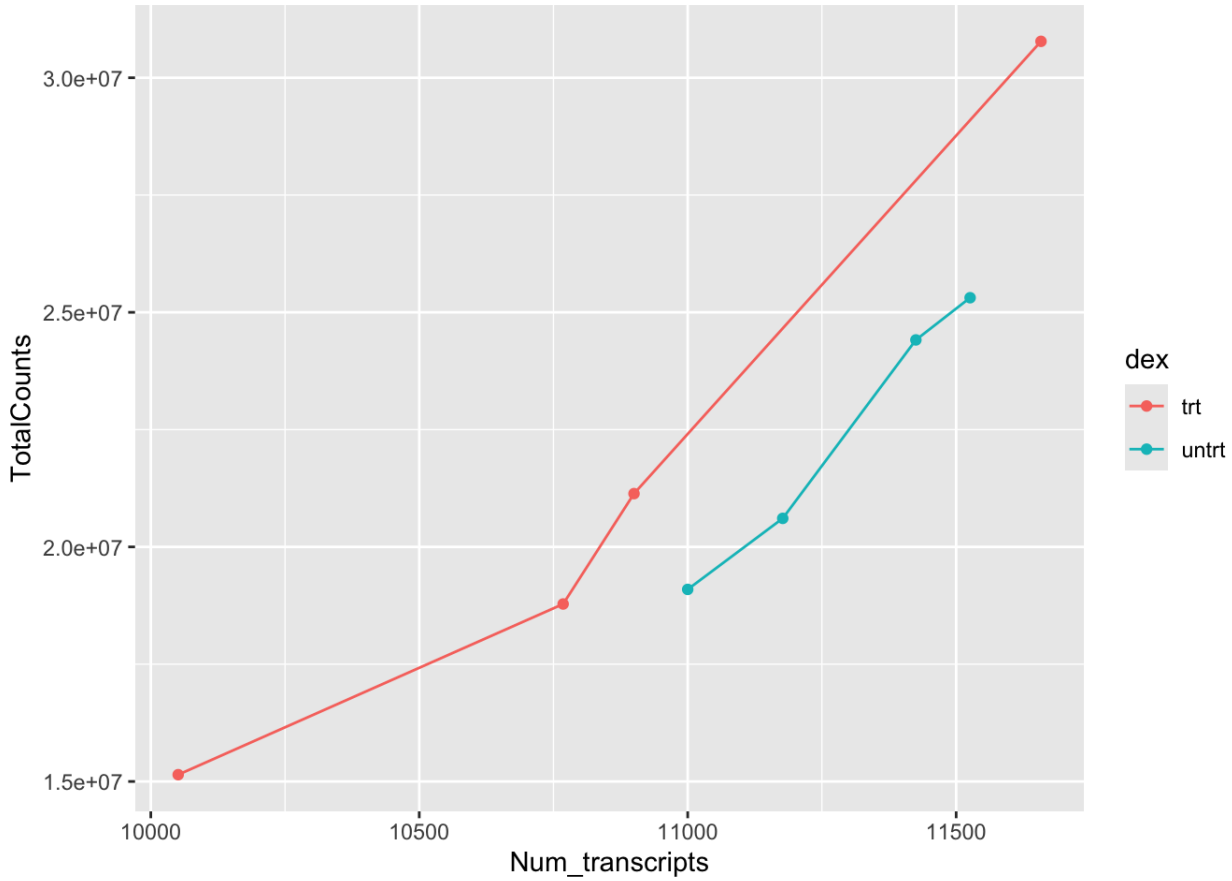
As you can see, we simply add a new geom, `geom_line()` to add a line plot.

Global vs local aesthetics

To make our code more effective, we can put shared aesthetics in the ggplot function (`ggplot()`). Aesthetics in the `ggplot()` function are global aesthetics, and will be applied to all geoms in the plot. Aesthetics in the geom functions are local aesthetics, and will only be applied to that specific geom.

Setting global aesthetics

```
ggplot(data=sc, aes(x=Num_transcripts, y = TotalCounts,color=dex)) +
  geom_point() +
  geom_line()
```



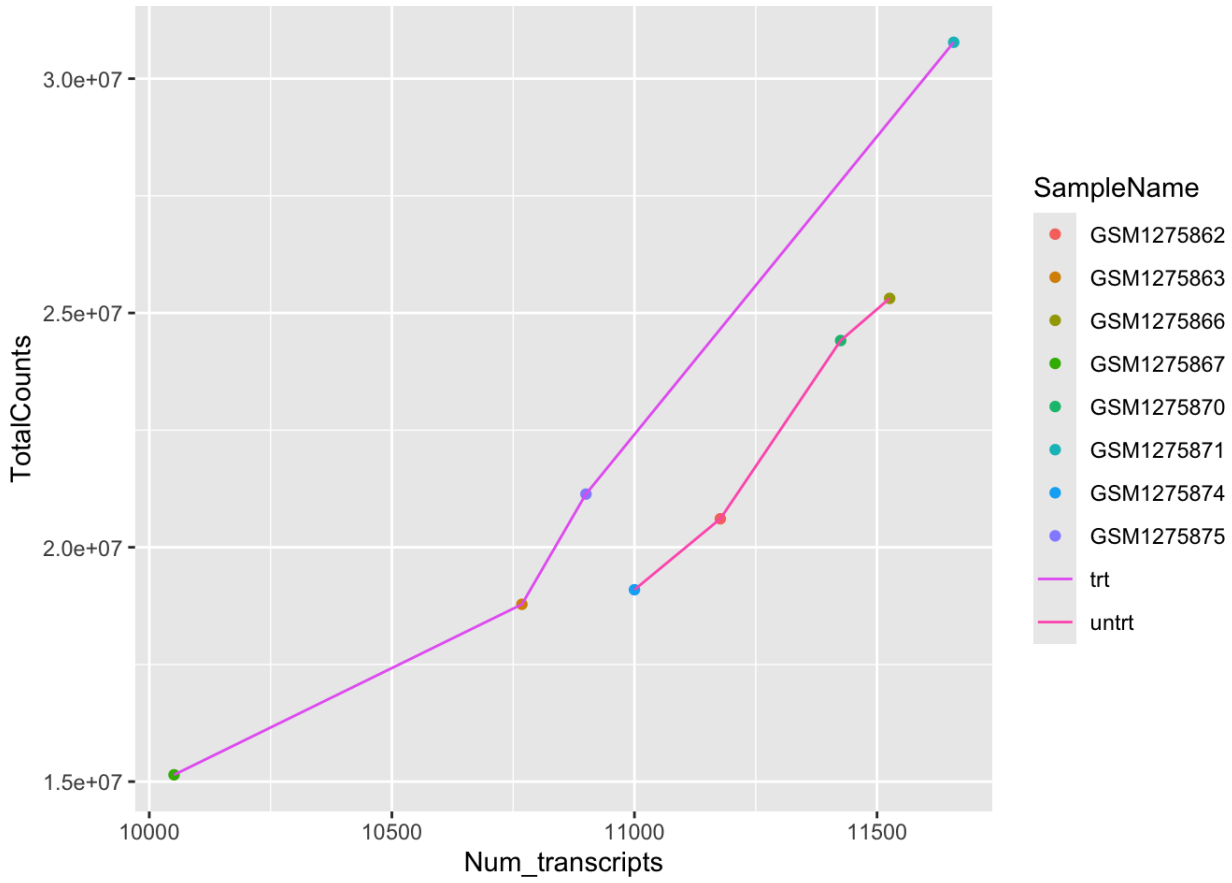
Geoms can be added in many different ways to create unique representations. Remember, that the layers are ordered, and the order matters for adding new geoms.

Setting local aesthetics

We can plot different aesthetics per geom.

```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,
                 color=SampleName)) +
```

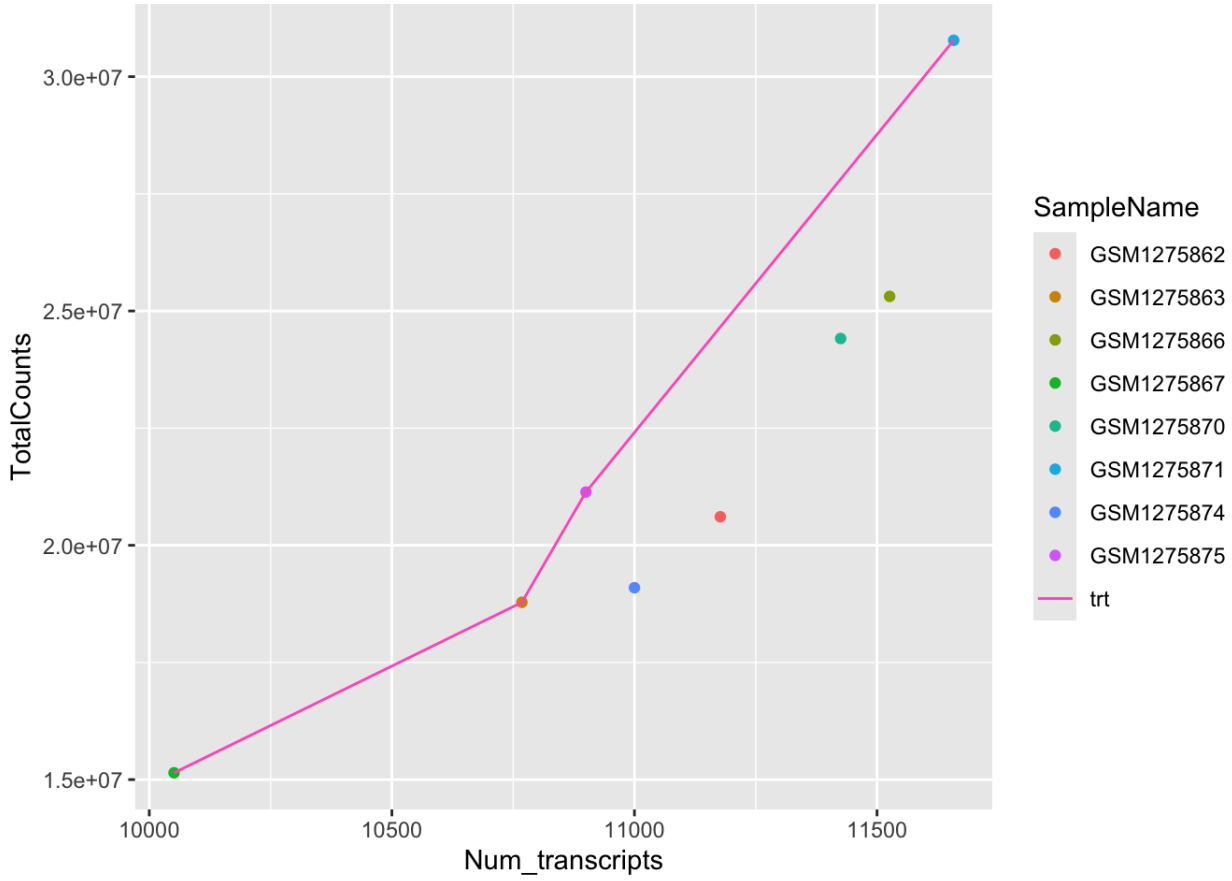
```
  geom_line(aes(x=Num_transcripts, y = TotalCounts,color=dex))
```



Subsetting data per geom

We can represent only a subset of data in one geom and not the other.

```
ggplot(data=sc) +
 geom_point(aes(x=Num_transcripts, y = TotalCounts,
                color=SampleName)) +
 geom_line(data=filter(sc,dex=="trt"),
           aes(x=Num_transcripts, y = TotalCounts,color=dex))
```



To get multiple legends for the same aesthetic, check out the CRAN package `ggnewscale` *(https://eliocamp.github.io/ggnewscale/)*. Whereas, legends for different aesthetics can easily be controlled with the `scale` and `guide` functions.

Statistical transformations

> Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:
>
> - bar charts, histograms, and frequency polygons bin your data and then plot bin counts, the number of points that fall in each bin.
> - smoothers fit a model to your data and then plot predictions from the model.
> - boxplots compute a robust summary of the distribution and then display a specially formatted box. The algorithm used to calculate new values for a graph is called a stat, short for statistical transformation. --- R4DS *(https://r4ds.had.co.nz/data-visualisation.html#statistical-transformations)*

Let's plot a bar graph using the data (`sc`).

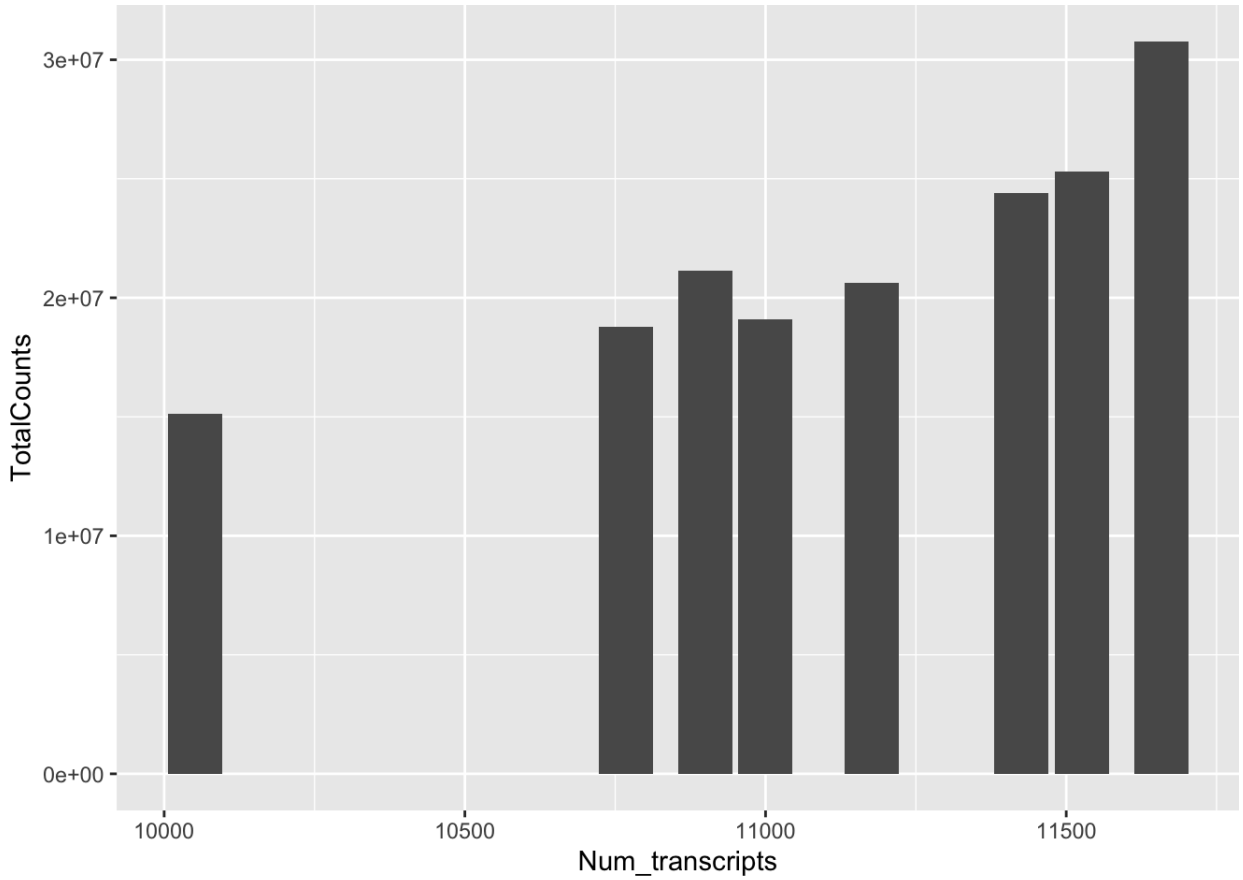
```
#returns an error message. What went wrong?
ggplot(data=sc) +
```

```
  geom_bar( aes(x=Num_transcripts, y = TotalCounts))
```

```
Error in `geom_bar()`:
! Problem while computing stat.
i Error occurred in the 1st layer.
Caused by error in `setup_params()`:
! `stat_count()` must only have an x or y aesthetic.
```

An error was returned. What's the difference between stat identity and stat count?

```
ggplot(data=sc) +
  geom_bar( aes(x=Num_transcripts, y = TotalCounts), stat="identity")
```



As we can see, `stat="identity"` returns the raw data, `stat="count"` "counts the number of cases at each x position". You should be aware of the default statistic used by a geom.

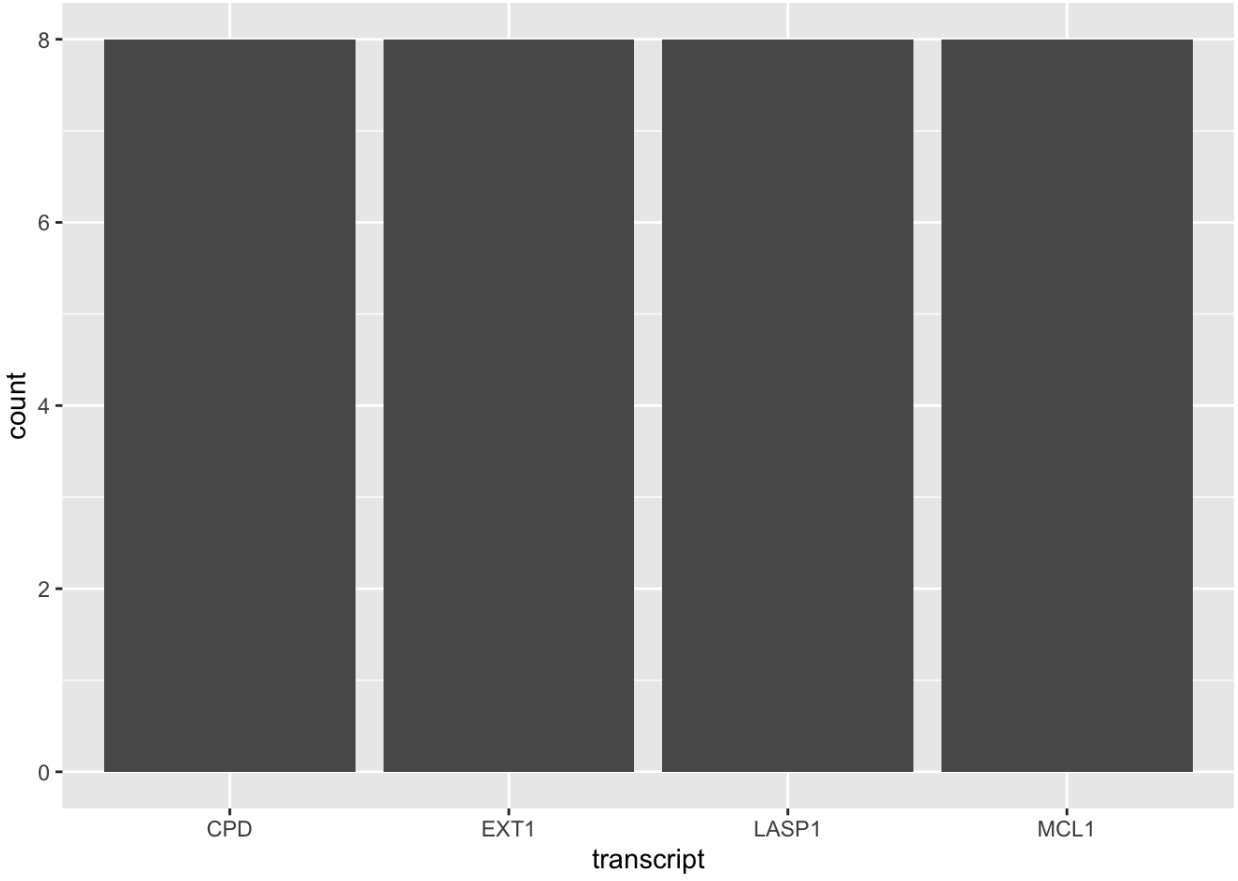
Let's look at another example. Here, we are looking at 4 genes of interest from our scaled counts.

```
#filter our data to include 4 transcripts of interest
keep_t<-c("CPD","EXT1","MCL1","LASP1")
```

```
interesting_trnsc<-scaled_counts %>%
  filter(transcript %in% keep_t)

#the default here is `stat_count()`, which requires only an x aesthet
ggplot(data = interesting_trnsc) +
  geom_bar(mapping = aes(x = transcript, y=counts_scaled))
```

```
Error in `geom_bar()`:
! Problem while computing stat.
i Error occurred in the 1st layer.
Caused by error in `setup_params()`:
! `stat_count()` must only have an x or y aesthetic.
```

```
#remove the y aesthetic
ggplot(data = interesting_trnsc) +
  geom_bar(mapping = aes(x = transcript))
```

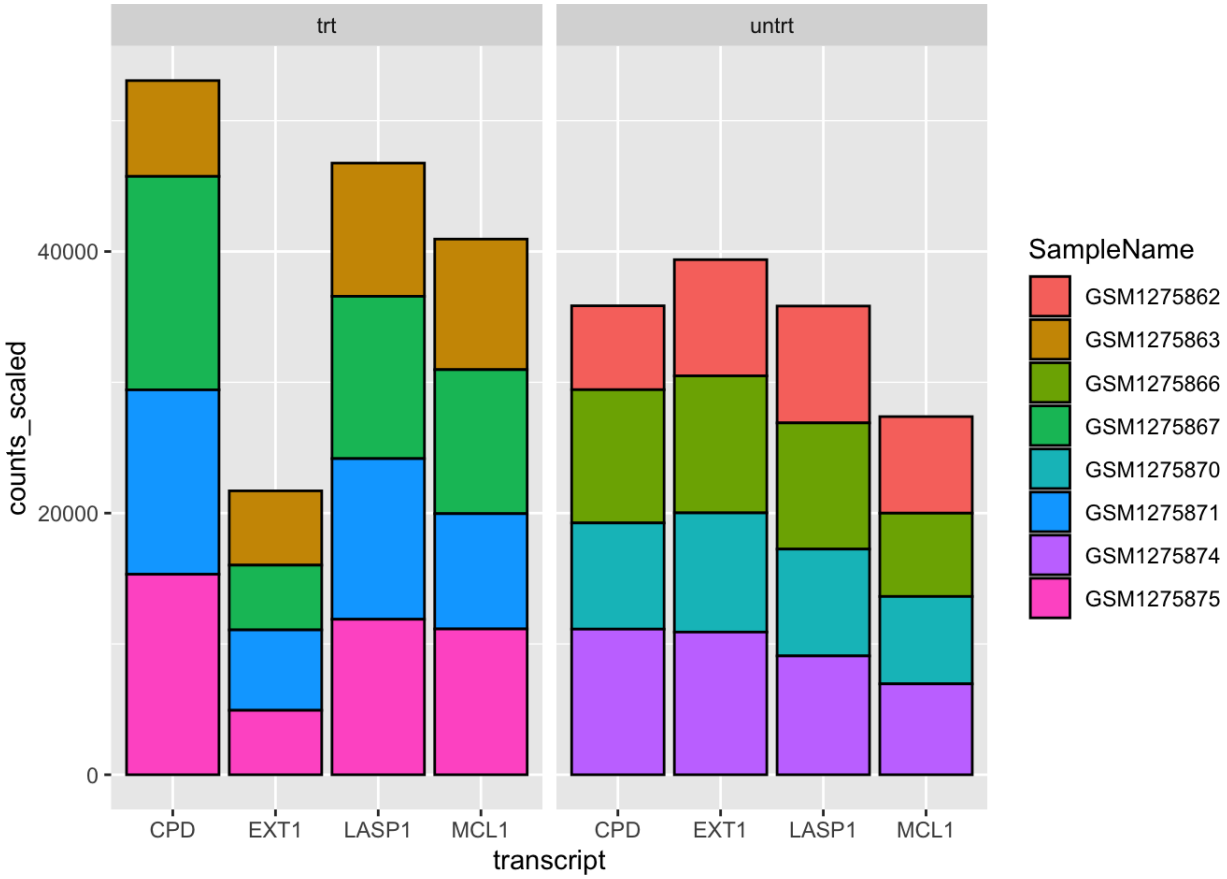


This is not a very useful figure, and probably not worth plotting. We could have gotten this info using `str()`, as we know we only have 8 samples. However, the point here is that there are default statistical transformations occurring with many geoms, and you can specify alternatives.

Let's change the stat parameter to "identity", and set a fill aesthetic to `SampleName`. This will plot the raw values of the normalized counts rather than how many rows are present for each transcript.

> **Note**
>
> Setting the color aesthetic in a bar plot results in a colored outline around the bar.

```
#defaulted to a stacked barplot
ggplot(data = interesting_trnsc) +
  geom_bar(mapping = aes(x = transcript,y=counts_scaled,
                         fill=SampleName),
           stat="identity",color="black") +
  facet_wrap(~dex)
```

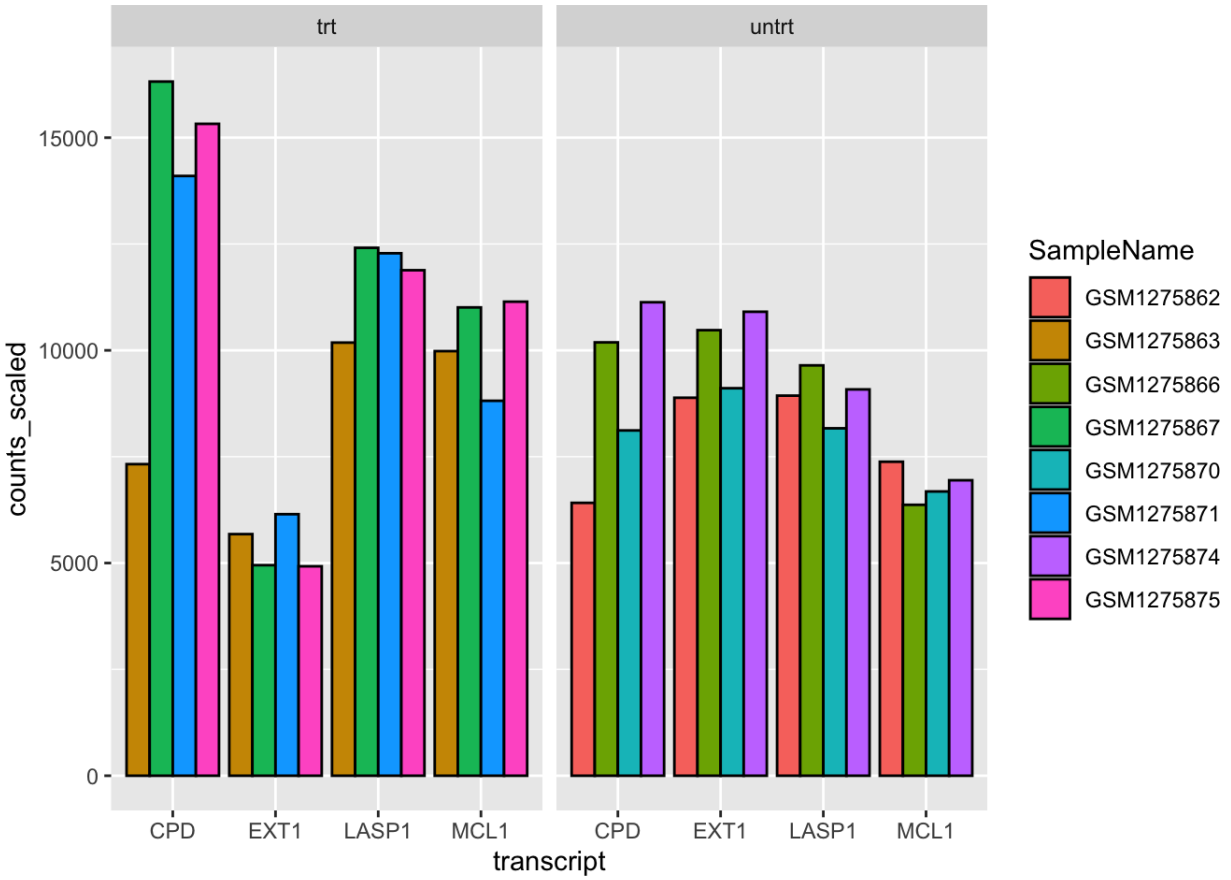


Notice that the output is stacked. What if we wanted the columns side by side?

We can again refer to our function arguments. In this case, we can modify `position` and set to "dodge" (`position="dodge"`). We can add facets to additionally view by treatment ("dex").

```
#introducing the position argument, position="dodge"
ggplot(data = interesting_trnsc) +
```

```
  geom_bar(mapping = aes(x = transcript,y=counts_scaled,
                         fill=SampleName),
           stat="identity",color="black",position="dodge") +
  facet_wrap(~dex)
```



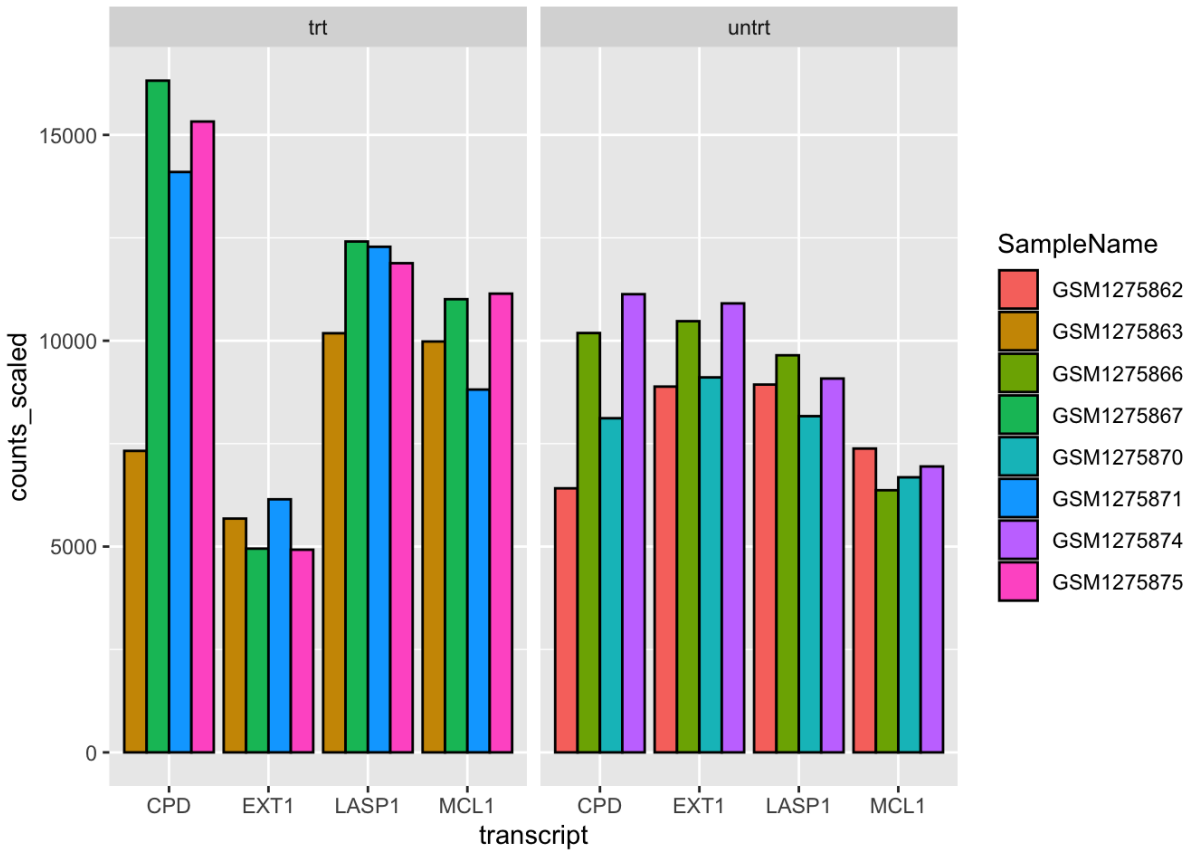
How do we know what the default stat is for `geom_bar()`? Well, we could read the documentation, `?geom_bar()`. This is true of multiple geoms. The statistical transformation can often be customized, so if the default is not what you need, check out the documentation to learn more about how to make modifications. For example, you could provide custom mapping for a box plot. To do this, see the examples section of the `geom_boxplot()` documentation.

geom_col()

If we read the documentation for `geom_bar()`, we see that there is an alternative function for when we want `stat="identity"` instead of `stat="count"`. That function is `geom_col()`. By using `geom_col`, instead of `geom_bar`, we avoid many of the problems we saw above.

For example,

```
ggplot(data = interesting_trnsc) +
  geom_col(mapping = aes(x = transcript,y=counts_scaled,
                         fill=SampleName),
           color="black",position="dodge") +
  facet_wrap(~dex)
```



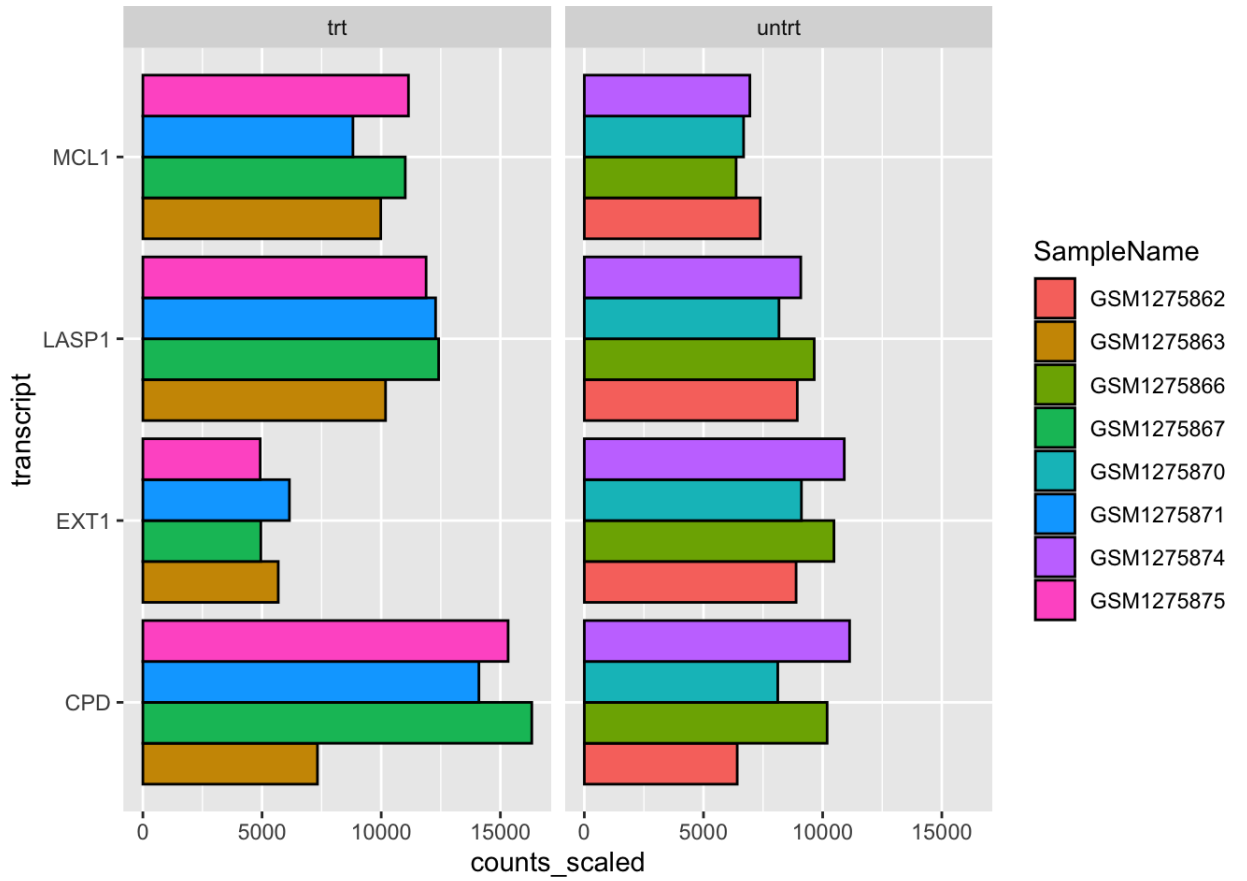
Coordinate systems

`ggplot2` uses a default coordinate system (the Cartesian coordinate system). This isn't super important until we want to do something like make a map (See `coord_quickmap()`) or create a pie chart (See `coord_polar()`).

When will we have to think about coordinate systems? We likely won't have to modify from default in too many cases (see those above). The most common circumstance in which we will likely need to change the coordinate system is in the event that we want to switch the x and y axes (`?coord_flip()`) or if we want to fix our aspect ratio (`?coord_fixed()`). Fixing the aspect ratio is useful when we want to ensure that one unit on the x-axis is the same length as one unit on the y-axis.

```
#let's return to our bar plot above
#get horizontal bars instead of vertical bars

ggplot(data = interesting_trnsc) +
  geom_bar(mapping = aes(x = transcript,y=counts_scaled,
                         fill=SampleName),
           stat="identity",color="black",position="dodge") +
  facet_wrap(~dex) +
```

```
  coord_flip()
```



Note

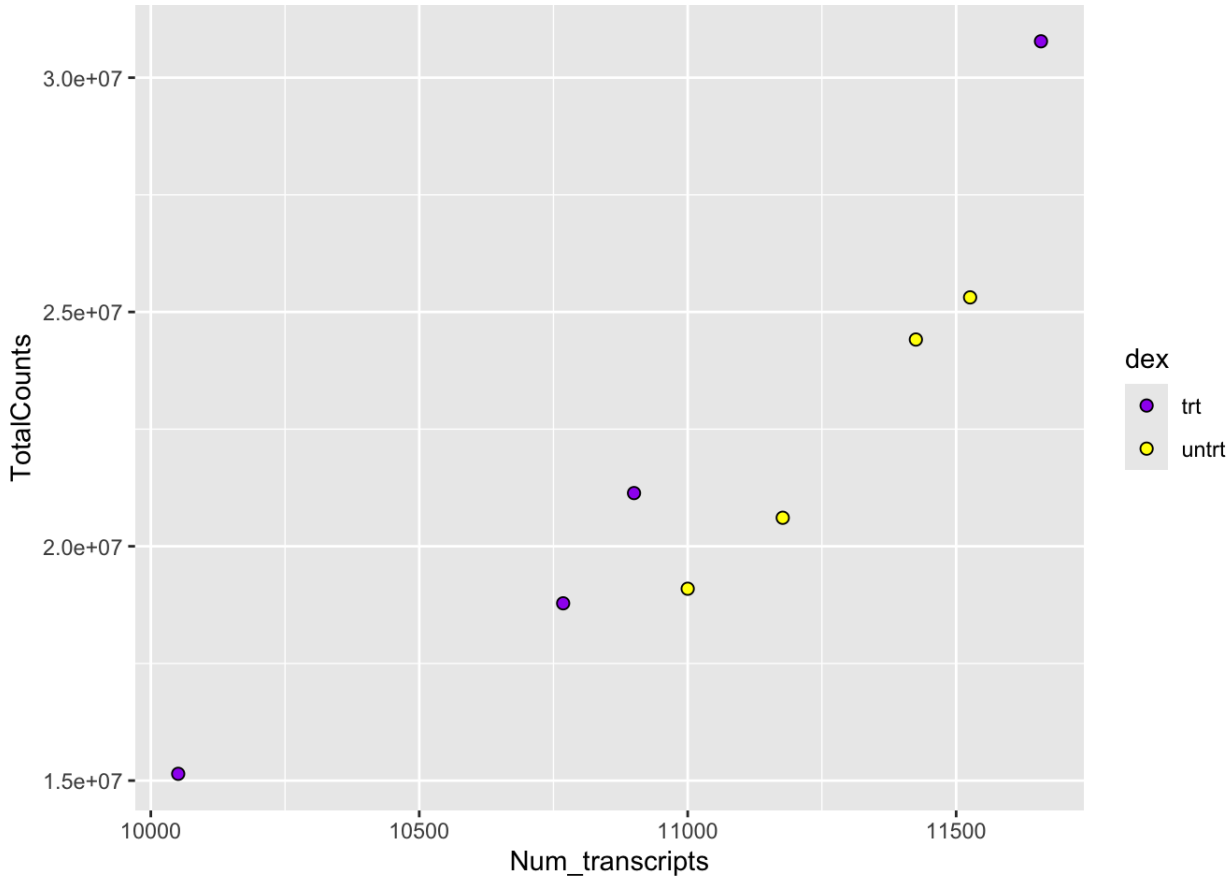
In the case of a bar plot, `coord_flip` is no longer required to get this effect. We could instead switch the `x` and `y` arguments. You may, however, be interested in using `coord_flip` with a different geom in the future, so it is nice to be aware of.

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, scale the axes, and customize the legends and theme *(https://ggplot2.tidyverse.org/reference/theme.html)*.

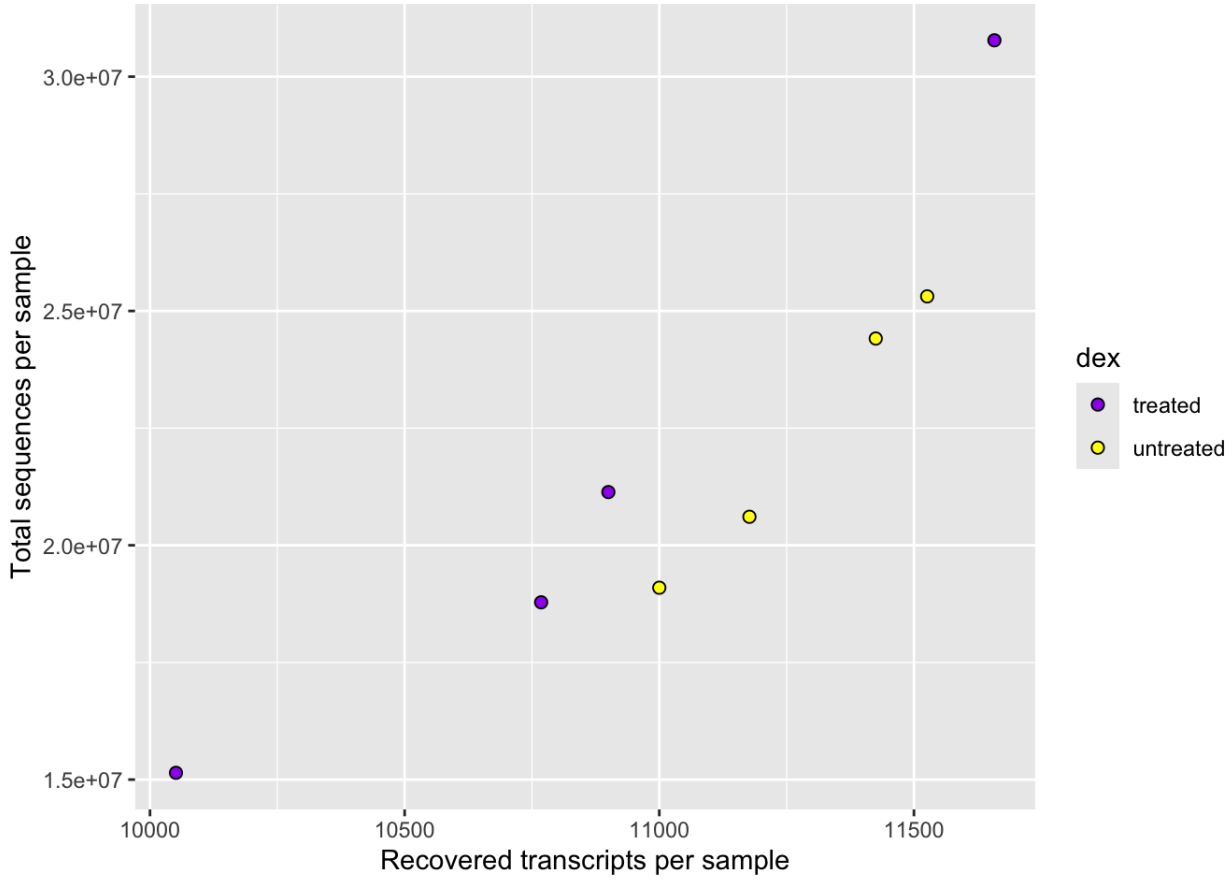
The default axes and legend titles come from the ggplot2 code. Let's return back to our simple data set, `sc`, to demonstrate.

```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
             shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"))
```



In the above plot, the y-axis label ("TotalCounts") is the variable name mapped to the y aesthetic, while the x-axis label ("Num_transcripts") is the variable name named to the x aesthetic. The fill aesthetic was set equal to "dex", and so this became the default title of the fill legend. We can change these labels using `ylab()`, `xlab()`, or `labs()`, and `guide()` for the legend.

```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
             shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"),
                    labels=c('treated','untreated'))+
  labs(x ="Recovered transcripts per sample",
       y="Total sequences per sample")#add x and y labels
```

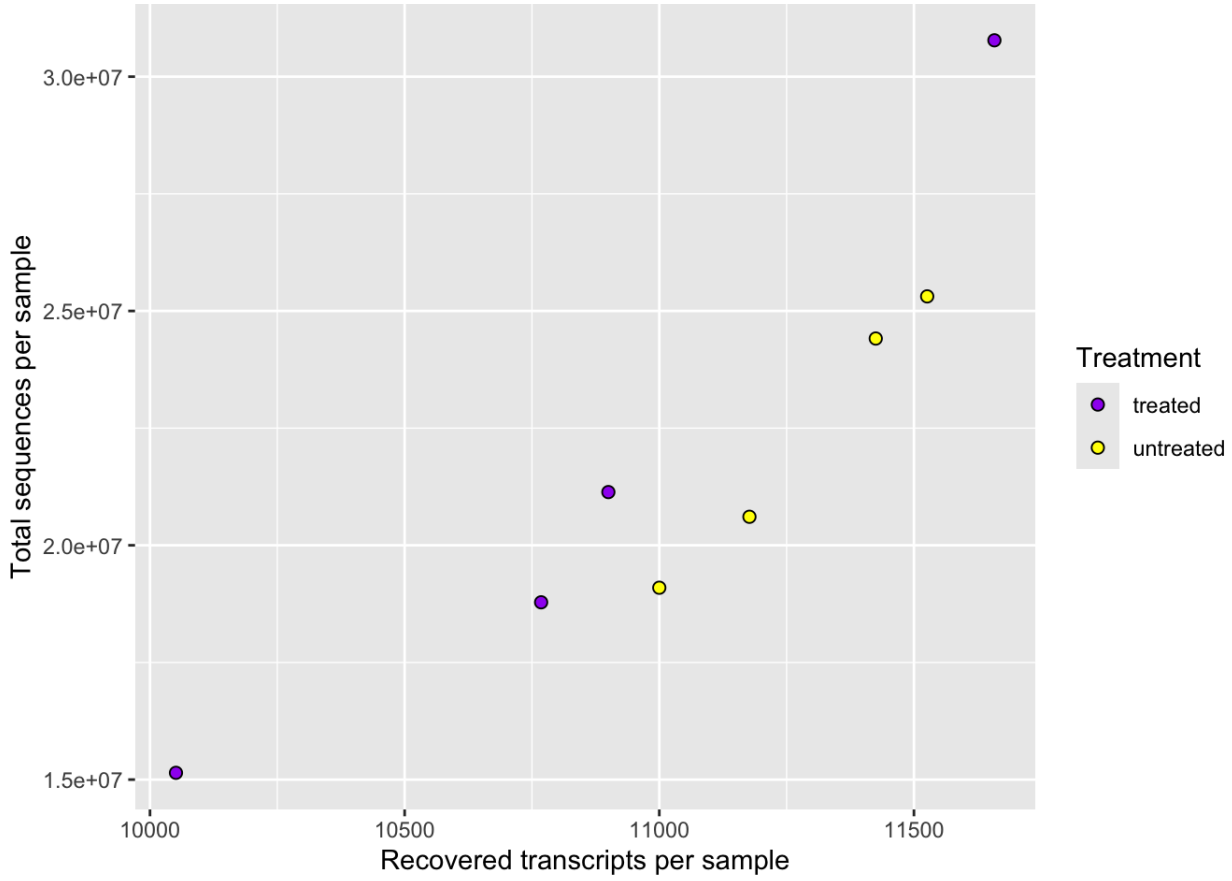


titles and subtitles

`labs()` can also be used to assign a title, subtitle, tags, and caption. See options with `?labs()`.

Let's change the legend title.

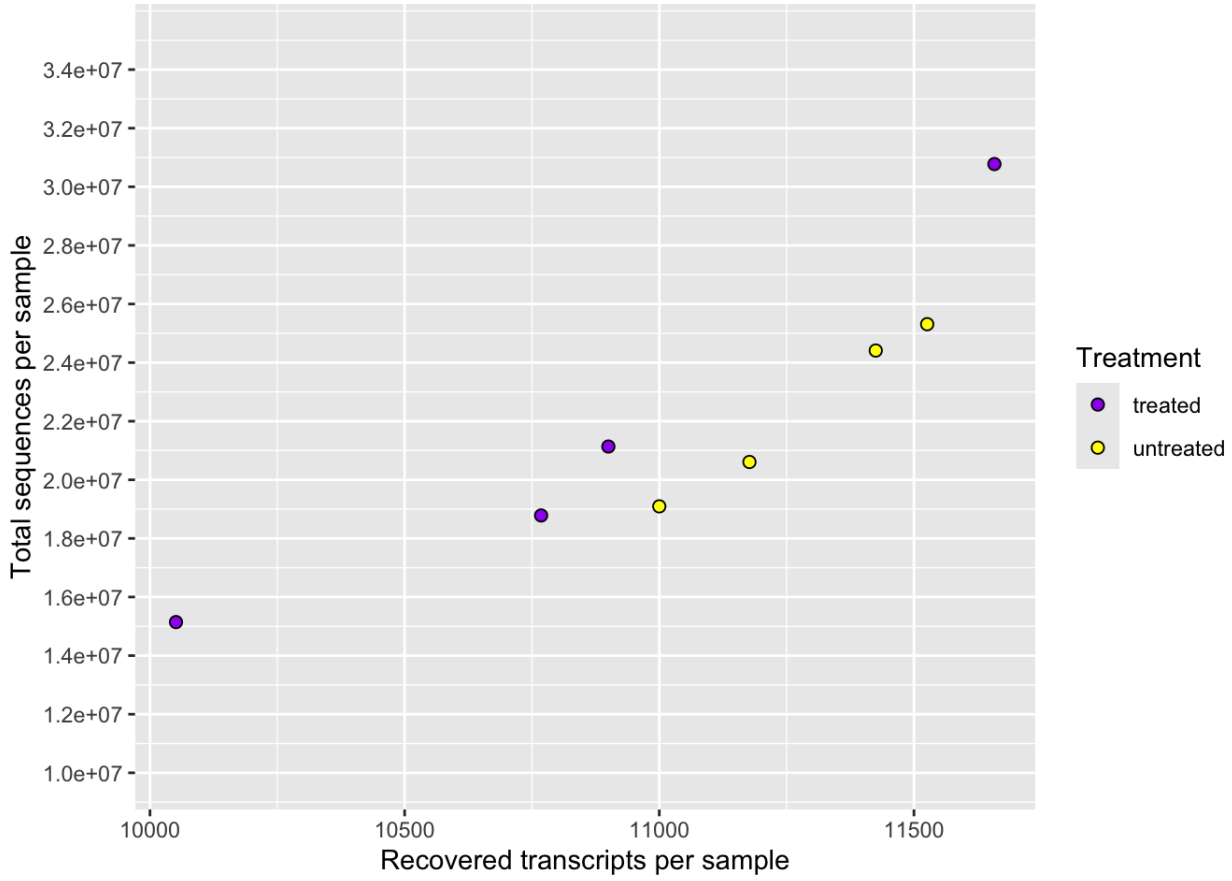
```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
             shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"),
                    labels=c('treated','untreated'))+
  labs(x ="Recovered transcripts per sample",
      y="Total sequences per sample") +
  guides(fill = guide_legend(title="Treatment"))
```



Legend titles can be modified with `guides()`, `labs()`, or within the `scale` function. For example, we could have also modified the legend title in `scale_fill_manual()` using the `name` argument.

We can modify the axes scales of continuous variables using `scale_x_continuous()` and `scale_y_continuous()`. Discrete (categorical variable) axes can be modified using `scale_x_discrete()` and `scale_y_discrete()`.

```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
             shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"),
                    labels=c('treated','untreated'))+
  labs(x ="Recovered transcripts per sample",
      y="Total sequences per sample") +
  guides(fill = guide_legend(title="Treatment")) + #label the legend
  scale_y_continuous(breaks=seq(1.0e7, 3.5e7, by = 2e6),
                     limits=c(1.0e7,3.5e7)) #change breaks and limits
```

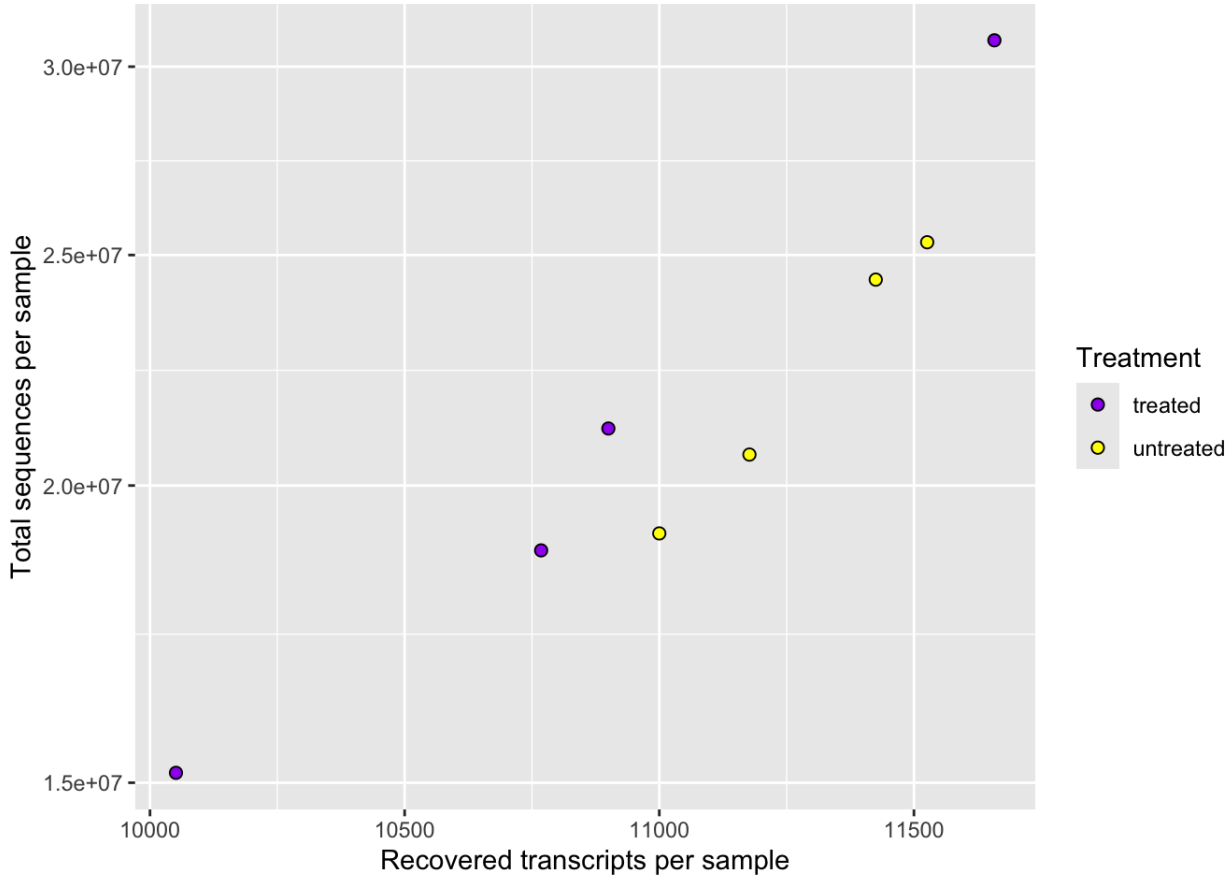



```
library(scales)
```

Check out the scales *(https://scales.r-lib.org/)* package to make nice axes labels.

Perhaps we want to represent these data on a logarithmic scale.

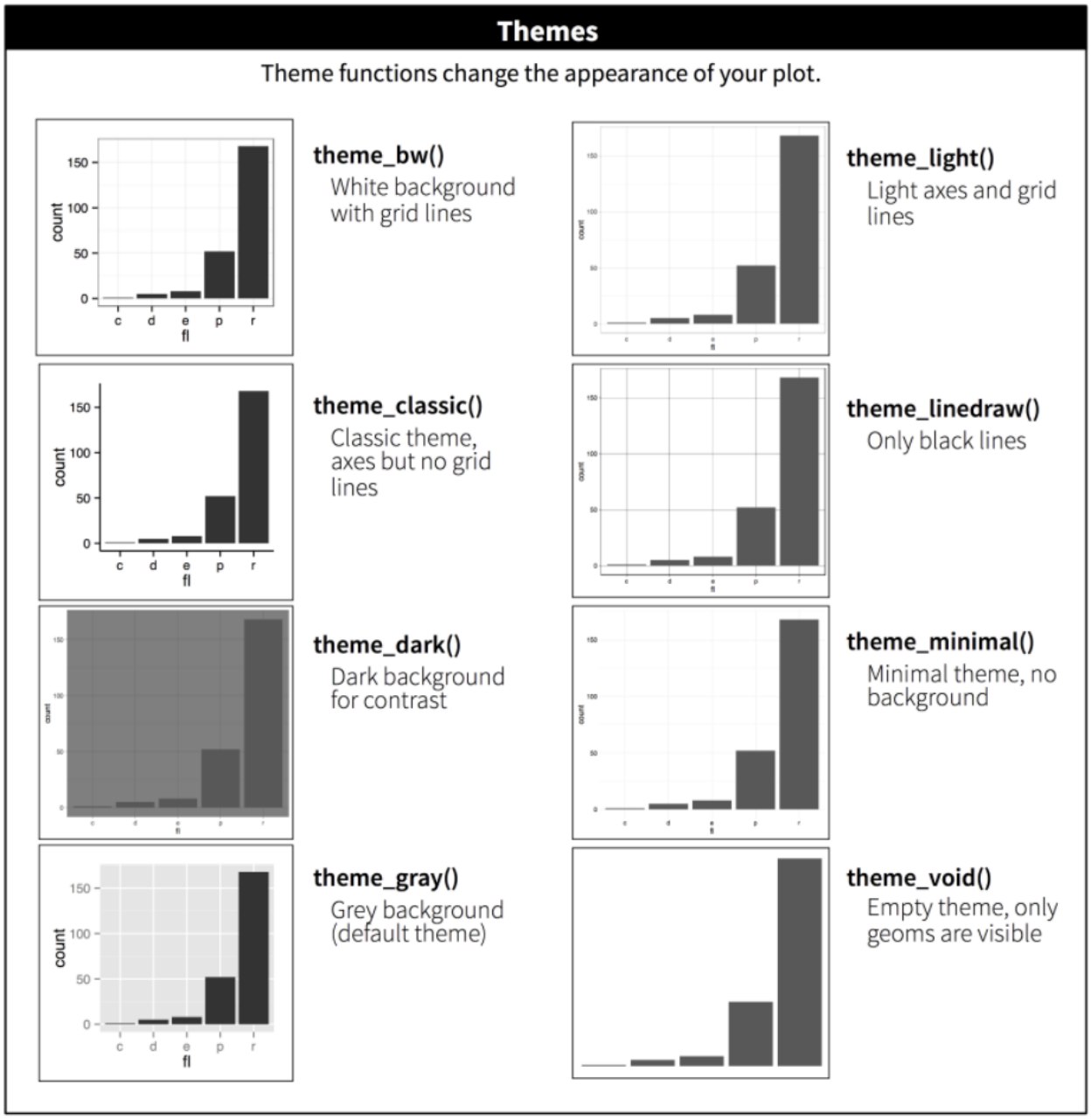
```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
             shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"),
                    labels=c('treated','untreated'))+
  labs(x ="Recovered transcripts per sample",
      y="Total sequences per sample") +
  guides(fill = guide_legend(title="Treatment")) + #label the legend
  scale_y_continuous(trans="log10") #use the trans argument
```



Note

You could manually transform the data without transforming the scales. The figures would be the same, excluding the axes labels. When you use the transformed scale (e.g., `scale_y_continuous(trans="log10")` or `scale_y_log10()`), the axis labels remain in the original data space. When the data is transformed manually, the labels will also be transformed.

Finally, we can change the overall look of non-data elements of our plot (titles, labels, fonts, background, grid lines, and legends) by customizing ggplot2 themes. Check out `?ggplot2::theme()`. For a list of available parameters. `ggplot2` provides 8 complete themes, with `theme_gray()` as the default theme.



You can also create your own custom theme and then apply it to all figures in a plot.

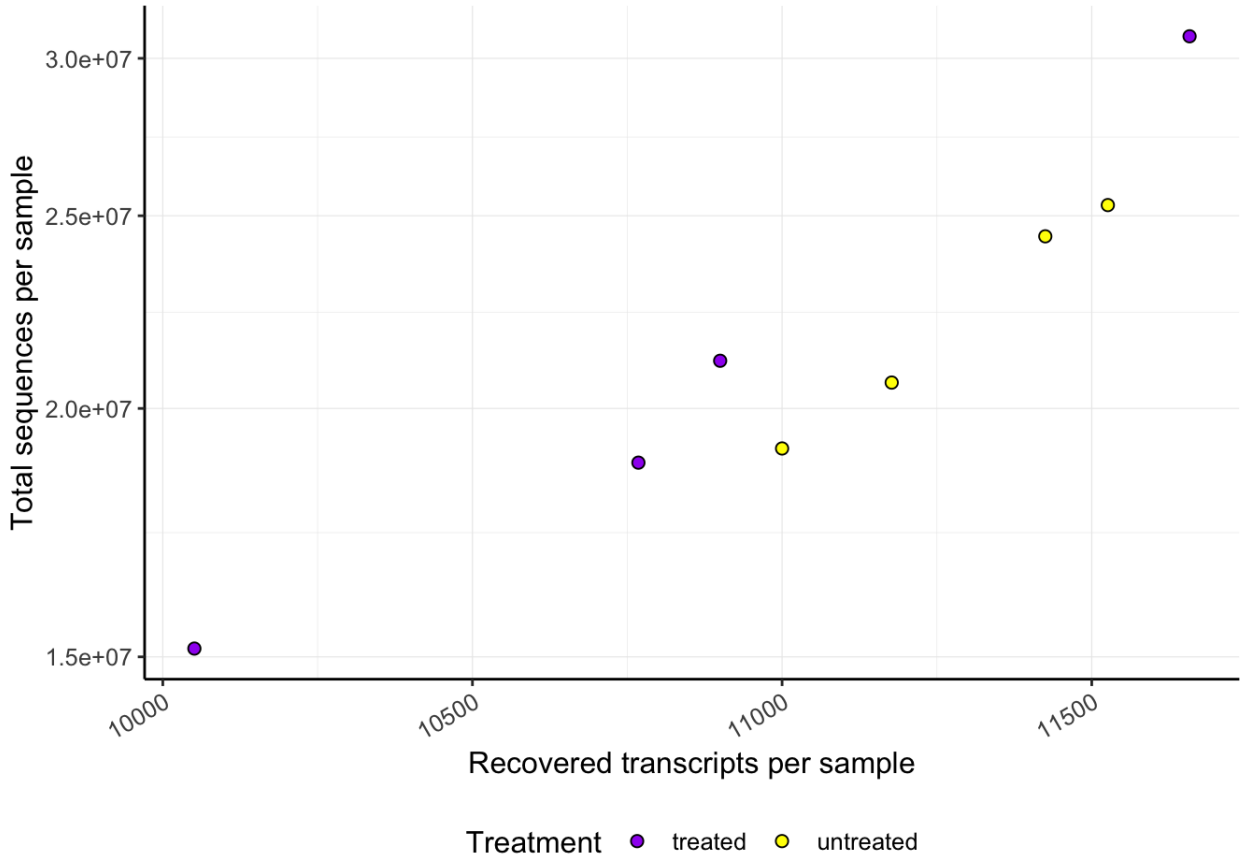
Create a custom theme to use with multiple figures.

```
#Setting a theme
my_theme <-
    theme_bw() +
      theme(
        #Remove the border around the plot
        panel.border = element_blank(),
        # Add the axis lines back in
        axis.line = element_line(),
        #resize the major and minor grid lines
        panel.grid.major = element_line(size = 0.2),
```

```
        panel.grid.minor = element_line(size = 0.1),
        #set the text size
        text = element_text(size = 12),
        #Move the legend to the bottom
        legend.position = "bottom",
        #Angle the x axis text
        axis.text.x = element_text(angle = 30, hjust = 1, vjust = 1)
      )
```

```
Warning: The `size` argument of `element_line()` is deprecated
as of ggplot2 3.4.0.
i Please use the `linewidth` argument instead.
```

```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
             shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"),
                    labels=c('treated','untreated'))+
  labs(x ="Recovered transcripts per sample",
       y="Total sequences per sample") +
  guides(fill = guide_legend(title="Treatment")) + #label the legend
  scale_y_continuous(trans="log10") + #use the trans argument
  my_theme
```



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)
```

Acknowledgements

Material from this lesson was inspired by 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.

From Data to Display: Crafting a Publishable Plot

Learning Objectives

1. Integrate previously learned `ggplot2`skills including data mapping, geoms, labels, scales, and themes to construct a complete visualization workflow.
2. Design and produce a polished, publication-ready plot from start to finish, making informed choices about plot type, aesthetics, and formatting.

For this exercise, we are going to use the information we have learned to create a volcano plot of our differential expression results.

Warning

This lesson requires audience participation.

Try not to cheat. Attempt to add the necessary code without referring to the documentation. To help you with this, code blocks are collapsed to hide the code.

> A volcano plot is a type of scatterplot that shows statistical significance (P-value) versus magnitude of change (fold change). It enables quick visual identification of genes with large fold changes that are also statistically significant. These may be the most biologically significant genes. --- Maria Doyle, 2021 *(https://training.galaxyproject.org/training-material/topics/transcriptomics/tutorials/rna-seq-viz-with-volcanoplot/tutorial.html)*

To generate a volcano plot, we need to know which genes were differentially expressed. Differential expression results can be obtained using a number of R packages (e.g., `limma`, `edgeR`, `DESeq2`). For today's lesson, we are using output generated from `edgeR` and available in the file, **"./data/diffexp_results_edger_airways.txt".**

Step 1: Load the required packages.

What package(s) do we need to create our plot?

Load the package(s)

```
library(tidyverse)
```

Primarily, we need `ggplot2`, but other packages from the `tidyverse` are useful for handling factors or wrangling the data as needed.

Step 2: Load and view the data.

For this lesson, we need to load the differential expression results. **How can we load the data and save to an object called `dexp`?** The data is at "./data/diffexp_results_edger_airways.txt".

Load the data

```
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 dat
i Specify the column types or set `show_col_types = FALSE` to quiet t
```

How can we further examine these data?

Examine the data

```
glimpse(dexp)
```

```
Rows: 15,926
Columns: 10
$ feature    <chr> "ENSG00000000003", "ENSG00000000419", "ENSG0000000
$ albut      <chr> "untrt", "untrt", "untrt", "untrt", "untrt", "untr
$ transcript <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH", "FUC
$ ref_genome <chr> "hg38", "hg38", "hg38", "hg38", "hg38", "hg38", "h
$ .abundant  <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TR
$ logFC      <dbl> -0.390100222, 0.197802179, 0.029160865, -0.1243820
$ logCPM     <dbl> 5.059704, 4.611483, 3.482462, 1.473375, 8.089146,
$ F          <dbl> 3.284948e+01, 6.903534e+00, 9.685073e-02, 3.772134
$ PValue     <dbl> 0.0003117656, 0.0280616149, 0.7629129276, 0.554695
$ FDR        <dbl> 0.002831504, 0.077013489, 0.844247837, 0.682326613
```

We can view the data using `View(dexp)` or select the data from the Global Environment pane.

To understand the structure of the data, use `dplyr::glimpse()` or `str()`.

Step 3: Define significance

The volcano plot helps us identify our significant genes. Generally, we are interested in identifying genes above or below certain thresholds for significance and log fold change. These thresholds can be fairly arbitrary. Here, we will define significance based on values with an FDR less than 0.01 and an absolute value of logFC of 1. Of note, logFC here is represented by log2 transformed values, so logFC = 1 corresponds to a fold change of 2.

Create a new column in `dexp` called "Significant" that contains TRUE values where genes were significantly differentially expressed based on the above thresholds and FALSE where they were not significant. Order the data frame by FDR and logFC. Save these transformed data to a new object called `dexp_sigtrnsc`. Unfamiliar with how to wrangling the data? Check out Part 2 of this Series, Introduction to Data Wrangling.

Wrangle the data

```
dexp_sigtrnsc <- dexp %>%
  mutate(Significant = FDR < 0.01 & abs(logFC) >= 1) %>% arrange(FDR, abs(logFC))
dexp_sigtrnsc[,-c(2,4,5)]
```

```
# A tibble: 15,926 x 8
   feature         transcript logFC logCPM     F    PValue       FDR
   <chr>           <chr>      <dbl>  <dbl> <dbl>     <dbl>     <dbl>
 1 ENSG00000165995 CACNB2      3.28   4.51 1575. 3.34 e-11   4.07e-7
 2 ENSG00000109906 ZBTB16      7.15   4.15 1429. 5.11 e-11   4.07e-7
 3 ENSG00000106976 DNM1       -1.76   5.38  646. 1.62 e- 9   2.57e-6
 4 ENSG00000162493 PDPN        1.88   5.68  768. 7.60 e-10   2.57e-6
 5 ENSG00000154930 ACSS1       1.89   4.96  657. 1.50 e- 9   2.57e-6
 6 ENSG00000157214 STEAP2      1.97   7.13  685. 1.25 e- 9   2.57e-6
 7 ENSG00000146250 PRSS35     -2.76   3.91  807. 6.16 e-10   2.57e-6
 8 ENSG00000120129 DUSP1       2.94   7.31  694. 1.18 e- 9   2.57e-6
 9 ENSG00000152583 SPARCL1     4.56   5.53  721. 1.000e- 9   2.57e-6
10 ENSG00000168309 FAM107A     4.74   2.78  656. 1.51 e- 9   2.57e-6
# i 15,916 more rows
```

Because we arranged the data by significance, we can create an object with the top 6 significant genes to highlight these in our volcano plot. Save the names of these genes to an object called `topgenes`.

Get 6 top significant genes

```
topgenes<-dexp_sigtrnsc$transcript[1:6]
topgenes
```

```
[1] "CACNB2" "ZBTB16" "DNM1"   "PDPN"   "ACSS1"  "STEAP2"
```

Step 4: Create the plot beginning with our 3 required entities.

What are the 3 required components needed to create a plot?

1. Data

   data - the data should include our differential expression results (`dexp_sigtrnsc`).

2. 1 or more geoms

   All data points are plotted using an x and y coordinate system. This requires `geom_point()`.

3. Mapping aesthetics

   x-axis - represents the logarithm of the fold change between two conditions *(https://en.wikipedia.org/wiki/Volcano_plot_(statistics))*.
   y-axis - represents the negative logarithm (base 10) of the p-value on the y-axis, ensuring that data points with lower p-values—indicative of higher statistical significance—are positioned toward the top of the plot *(https://en.wikipedia.org/wiki/Volcano_plot_(statistics))*.

   color - use color to differentiate between "significant" and "non-significant" genes.

Begin the plot

```
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant))
```

Step 5: Customize Our Figure

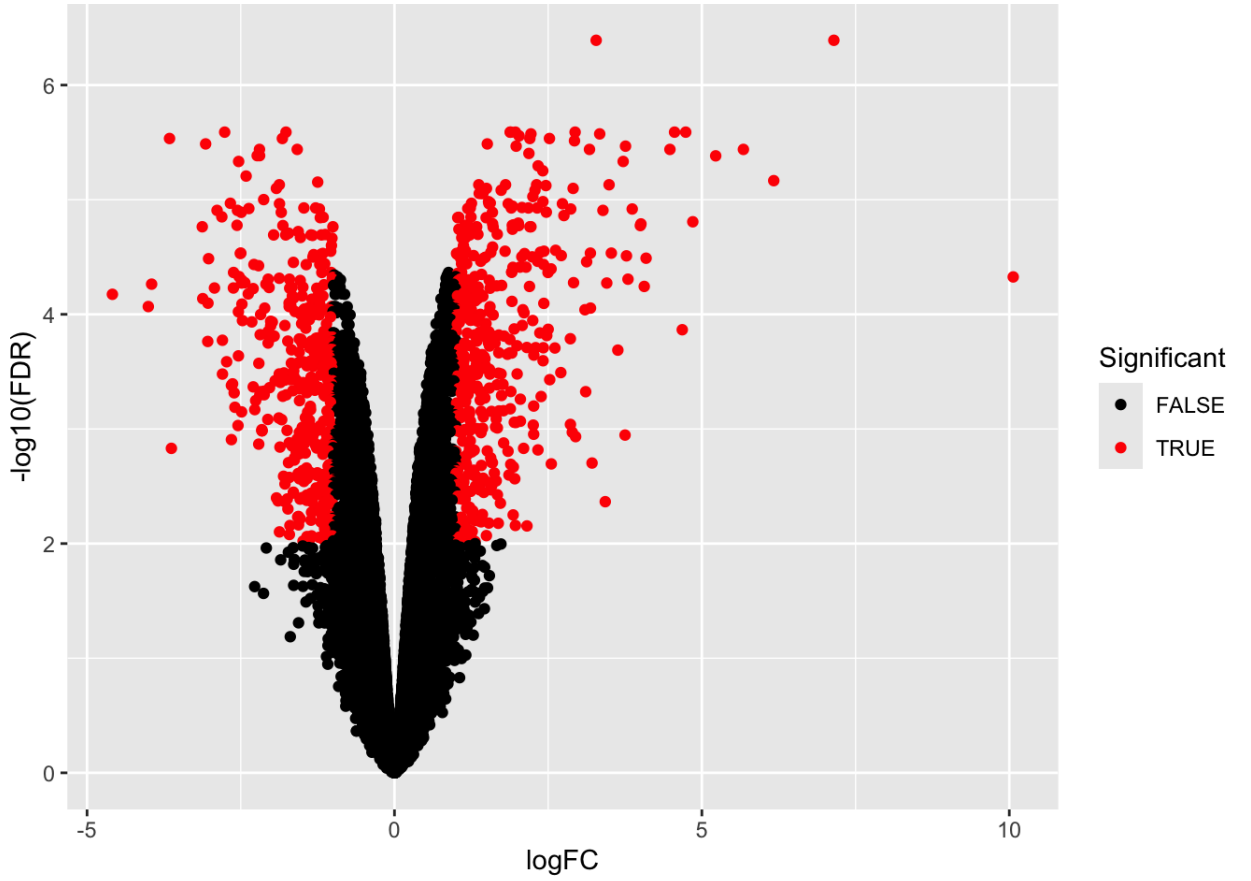
At this point, you have a relatively nice plot with just a couple of lines of code, but we really want our figure to shine for publication. Think about what changes can be made to make the plot nice but also effective!

Scale the Colors

How can we control the colors representing our TRUE / FALSE values? Assign "black" to FALSE and "red" to TRUE.

Scale the Colors

```
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant)) +
  scale_color_manual(values = c("black","red"))
```



Note

There are many scale functions and scale_color functions.

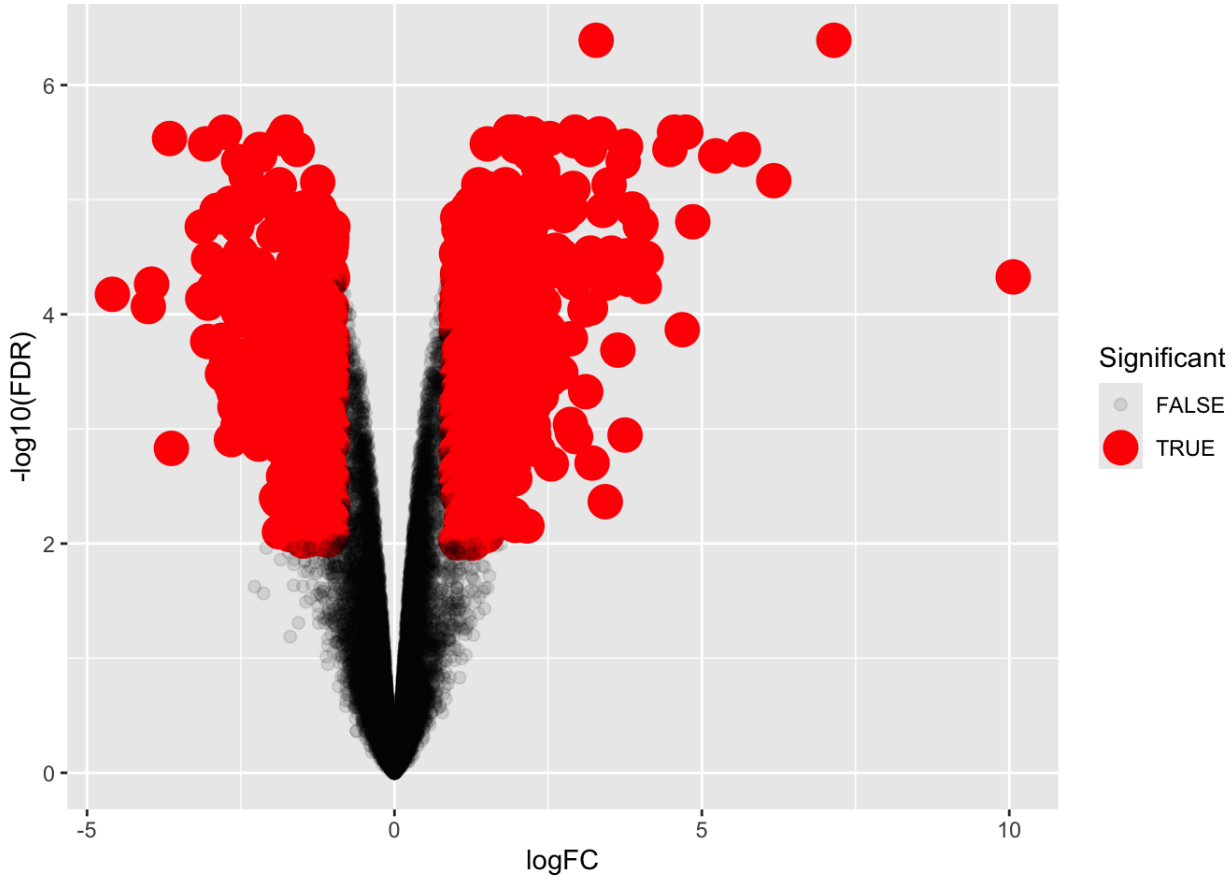
Add Size and Alpha attributes to our Mapping Aesthetics

The red and black colors nicely discriminate between significant and non-significant genes. However, we can make a few more changes to really highlight our "significant" genes. Two things come to mind. We can make the non-significant points less visible with `alpha` and `size`.

If we want to represent differences in a variable using `alpha` and `size`, where should we put these in our code?

Assign alpha and size aesthetics

```
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant, alpha =Significant,
  size = Significant)) +
  scale_color_manual(values = c("black","red"))
```



Warning messages

You will likely see the following warning messages:

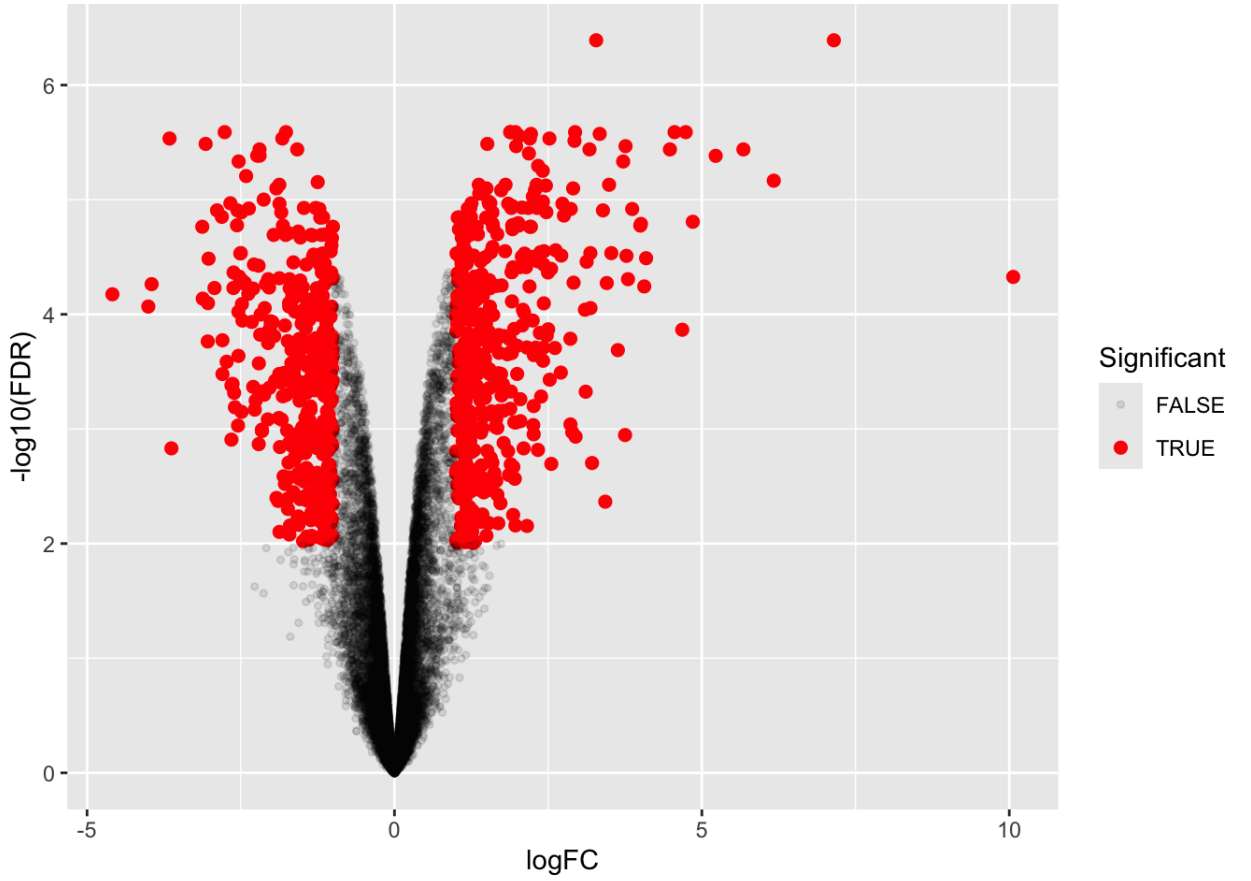
1: Using alpha for a discrete variable is not advised.
2: Using size for a discrete variable is not advised.

These are not errors, but you should consider what they mean for your plot. Make sure your choices are not misleading the audience.

The size mapping results in very large points for "Significant = TRUE". How can we fix this?

Scale the size aesthetic

```
# Use scale_size_discrete to set the range of sizes possible
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant, alpha =Significant,
  size = Significant)) +
  scale_color_manual(values = c("black","red")) +
  scale_size_discrete(range=c(1,2))
```



Again, scale can be applied to the parameters in our mapping aesthetics including the x and y axes.

Legends

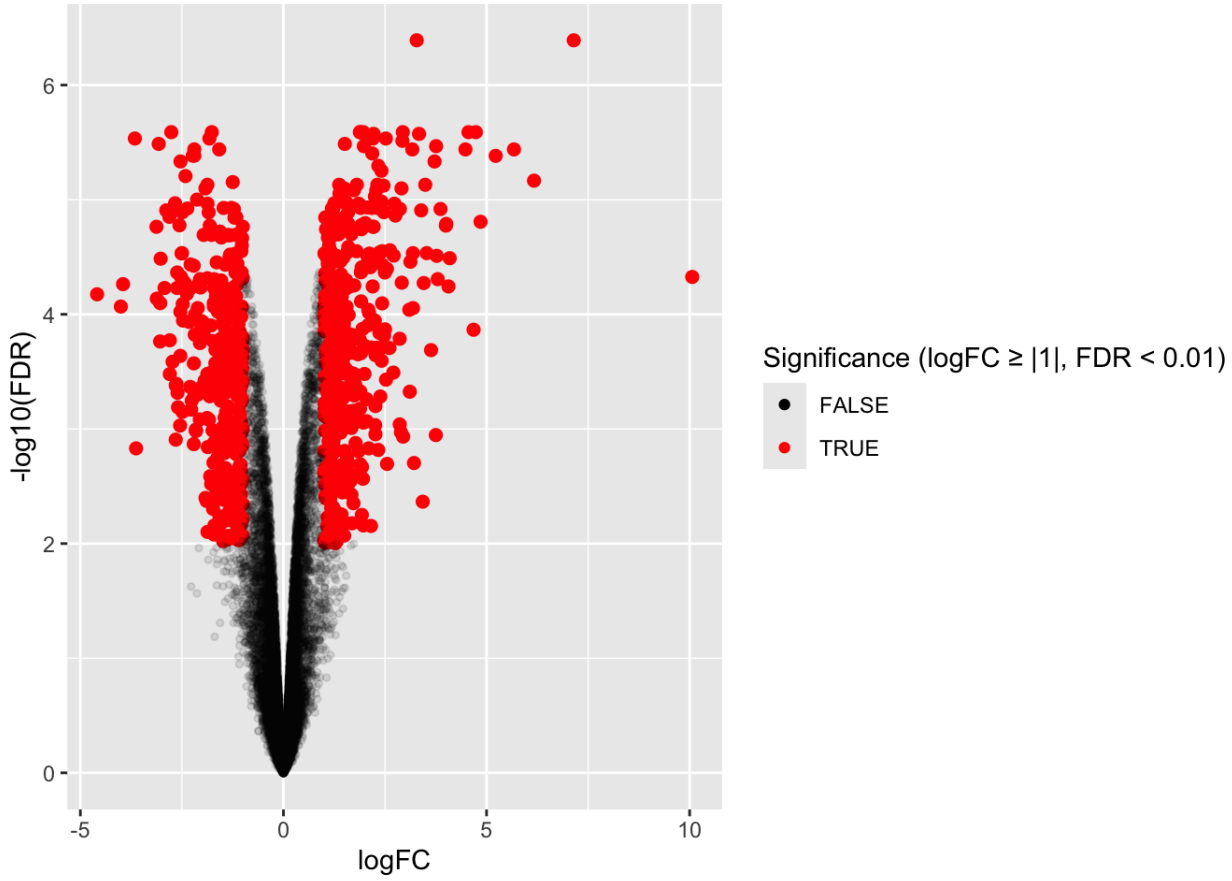
If we want separate legends for each aesthetic, we can set this using arguments in the `scale` functions. For example, see `guide` and `name`.

Fix the legend

The legend isn't great. It is neither informative nor visually appealing. **How can we modify the legend?**

Fix the legend

```
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant, alpha =Significant,
  size = Significant)) +
  scale_color_manual(values = c("black","red")) +
  scale_size_discrete(range=c(1,2)) +
    guides(color = guide_legend(
      "Significance (logFC \u2265 |1|, FDR < 0.01)"), size = "none", alpha= "none")
```



There are multiple ways to modify the legend, including using `guides()` and `theme`.

Adding mathematical expressions

There are multiple ways to add mathematical expressions to ggplot2 figures.

Here are some useful resouces:

- From ggplot2 docs: https://ggplot2.tidyverse.org/articles/faq-axes.html#how-can-i-add-superscripts-and-subscripts-to-axis-labels *(https://ggplot2.tidyverse.org/articles/faq-axes.html#how-can-i-add-superscripts-and-subscripts-to-axis-labels)*
- Guide on special symbols: https://steffilazerte.ca/posts/ggplot-symbols/#table *(https://steffilazerte.ca/posts/ggplot-symbols/#table)*
- Using `expression()`: https://library.virginia.edu/data/articles/mathematical-annotation-in-r *(https://library.virginia.edu/data/articles/mathematical-annotation-in-r)*
- `?plotmath` and `demo(plotmath)`

In this example, I used unicode, which is a universal character encoding standard assigning a unique number / code to every character, symbol, etc. In R and ggplot2, unicode can be used to display special symbols (like mathematical operators, Greek letters, or arrows) in plot labels, legends, and titles by using escape sequences such as \u2265 for "≥". However, it doesn't work with all graphic devices, so use caution.

As the references above suggest, we could have also used `bquote()` or `expression()`. For example, try the following code instead: `guides(color = guide_legend(bquote("Significance (logFC " >= " |1|, FDR < 0.01")), size = "none", alpha= "none")`. Or, make the x and y axis labels nicer: `labs(x=expression(paste(Log[2],"FC")),y=expression(paste(-Log[10],italic("P"))))`.

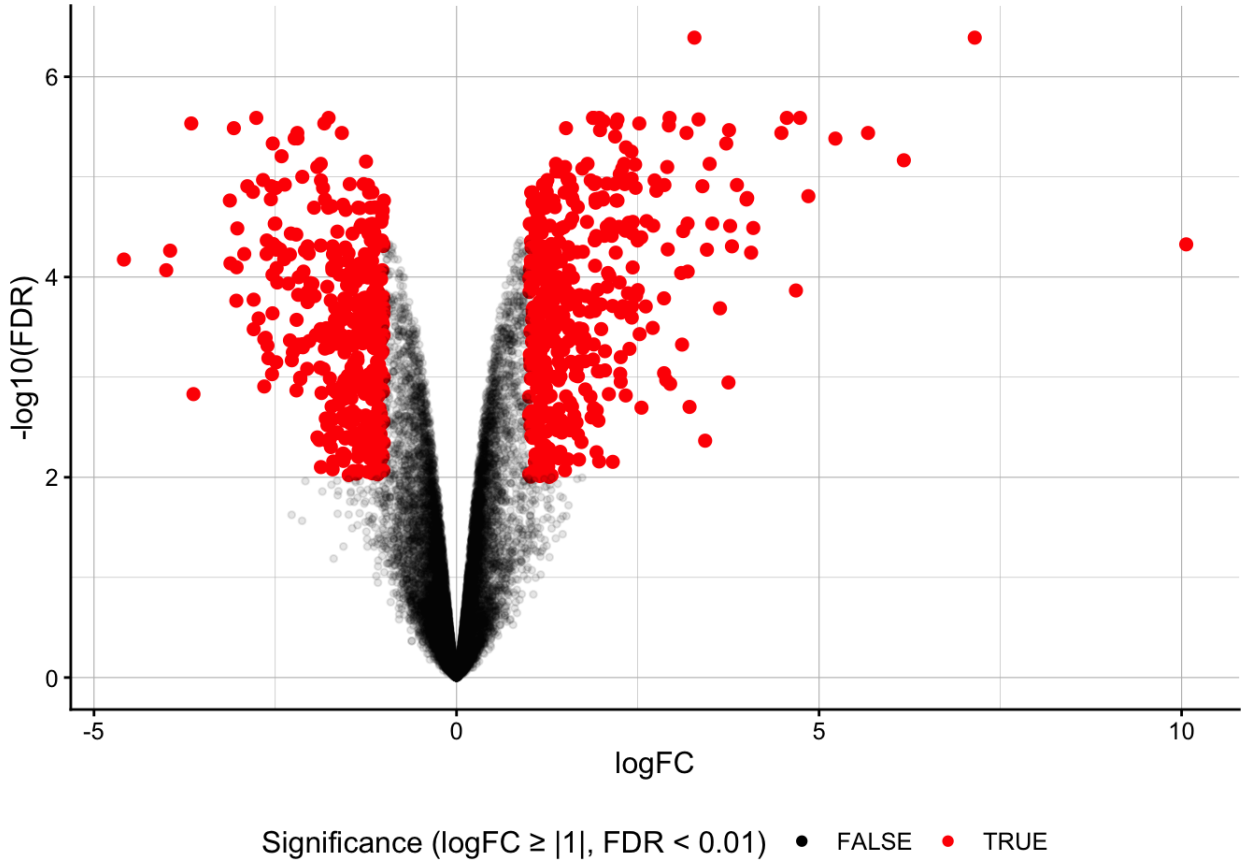
I would not necessarily memorize how to do this, but would look it up as needed.

Clean it up with `theme`

Let's make this nicer by customizing the background, grid lines, legend position, and text. **How can we modify theme elements?**

Set theme elements

```
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant, alpha =Significant,
  size = Significant)) +
  scale_color_manual(values = c("black","red")) +
  scale_size_discrete(range=c(1,2)) +
    guides(color = guide_legend(
      "Significance (logFC \u2265 |1|, FDR < 0.01)"), size = "none", alpha= "none")
  theme_classic() +
  theme(panel.grid.major = element_line(size = 0.2, color="grey"),
        panel.grid.minor = element_line(size = 0.1, color="grey"),
        text = element_text(size = 12),
        legend.position = "bottom")
```



You are free to customize your plot however you see fit. Here, I decided to use the complete theme, `theme_classic()`. I then made some additional changes from there. For example, I added in major and minor grid lines, resized the text, and positioned the legend.

- Add major grid lines: `panel.grid.major = element_line(size = 0.2, color="grey")`
- Add minor grid lines: `panel.grid.minor = element_line(size = 0.1, color="grey")`
- Assign all text 12 point font: `text = element_text(size = 12)`
- Move the legend to the bottom of the plot: `legend.position = "bottom"`

Step 6: Label the most significant points.

How can we add text labels to some of our points?

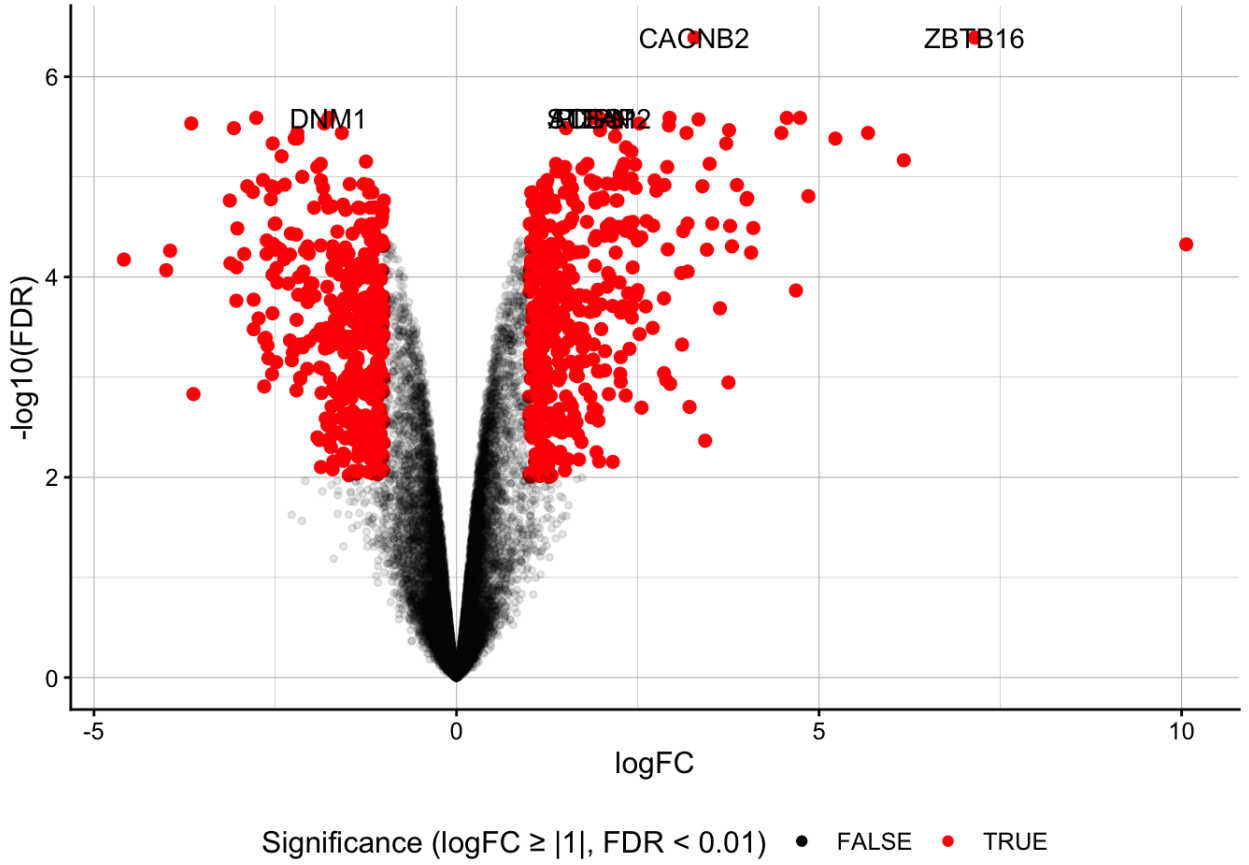
To label our top significant genes, we can add an additional geom. In this case, `geom_text()`.

Note

Here, we only want to plot labels for our significant genes. We can call these directly by filtering the data.

Add text labels

```
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant, alpha =Significant,
  size = Significant)) +
  scale_color_manual(values = c("black","red")) +
  scale_size_discrete(range=c(1,2)) +
    guides(color = guide_legend(
      "Significance (logFC \u2265 |1|, FDR < 0.01)"),
      size = "none", alpha= "none") +
  geom_text(data=dexp_sigtrnsc %>%
                   filter(transcript %in% topgenes), #filter the data
                 aes(label=transcript)) +
  theme_classic() +
  theme(panel.grid.major = element_line(size = 0.2, color="grey"),
        panel.grid.minor = element_line(size = 0.1, color="grey"),
        text = element_text(size = 12),
        legend.position = "bottom")
```



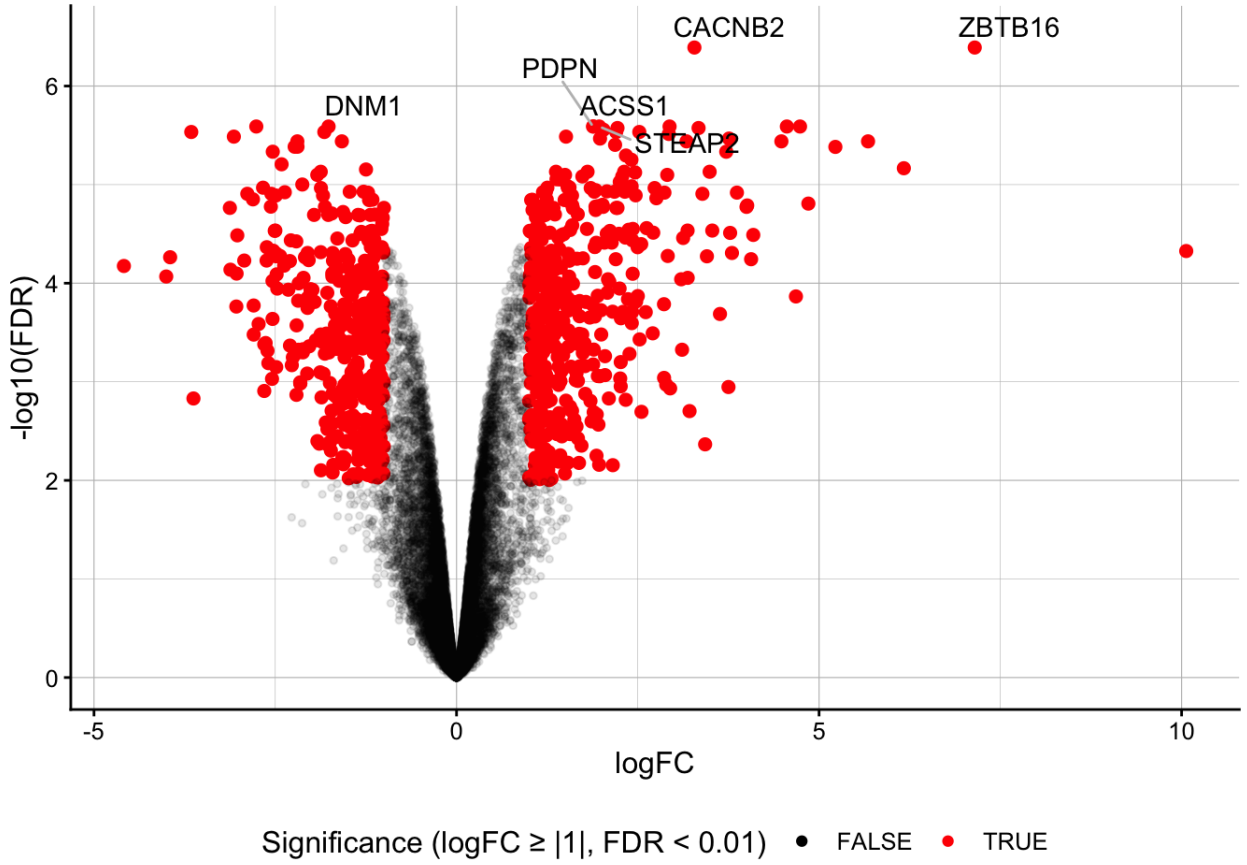
As we can see, `geom_text` results in overlapping labels. To avoid overlapping labels, we can use `check_overlap = TRUE` - feel free to try it. However, this will drop labels, and we want all 6 top genes to have labels.

To get around this, we can use a package called `ggrepel` *(https://ggrepel.slowkow.com/)*, which keeps the labels from overlapping.

Use ggrepel to avoid overlapping labels

```
# install the package with install.packages("ggrepel")
library(ggrepel)

ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = -log10(FDR))) +
  geom_point(aes( color = Significant, alpha =Significant,
  size = Significant)) +
  scale_size_discrete(range=c(1,2)) +
  scale_color_manual(values = c("black","red")) +
    guides(color = guide_legend(
      "Significance (logFC \u2265 |1|, FDR < 0.01)"),
      size = "none", alpha= "none") +
  geom_text_repel(data=dexp_sigtrnsc %>%
                    filter(transcript %in% topgenes),
                  aes(label=transcript),
                  nudge_y=0.1,nudge_x=0.2,direction="both",
                  segment.color="gray") +
  theme_classic() +
```

```
  theme(panel.grid.major = element_line(size = 0.2, color="grey"),
        panel.grid.minor = element_line(size = 0.1, color="grey"),
        text = element_text(size = 12),
        legend.position = "bottom")
```



Geom ordering

In `ggplot2`, the order in which you add layers (such as `geom_point()`, `geom_text()`, `geom_line()`, etc.) directly affects how your plot is rendered:

Layers added later are drawn on top of earlier layers. For example, if you add `geom_point()` first and then `geom_text()`, the text labels will appear on top of the points. If you reverse the order, the points may cover or obscure the text.

Using an External Package.

There are many packages external to `ggplot2` that can be used to create or enhance figures. We will learn about some of these in the next lesson. Such packages can save us a lot of time and energy. See the below example with `EnhancedVolcano`.

Note

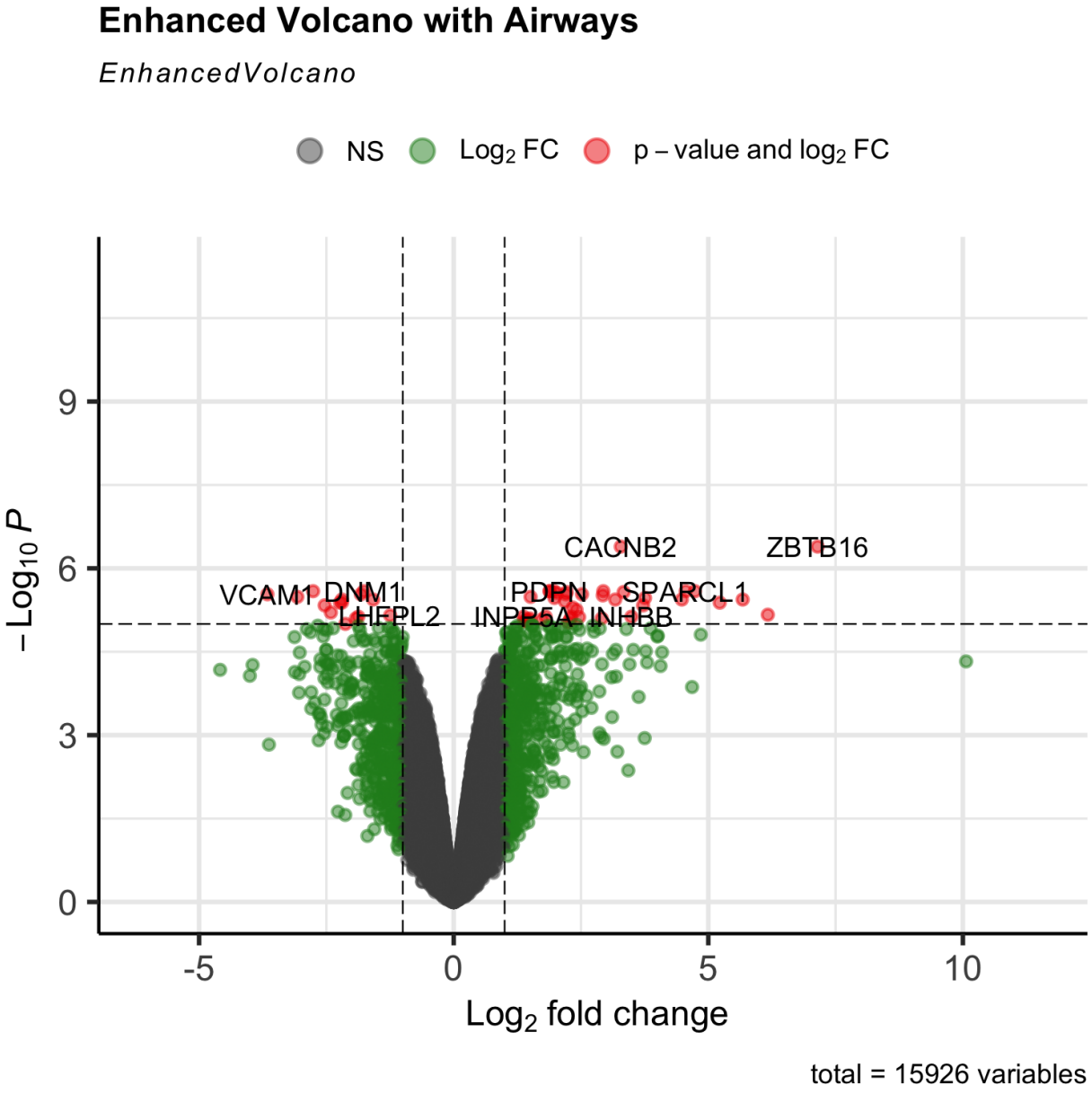
Search for packages using a dedicated R search Engine *(https://rseek.org/)*.

EnhancedVolcano

There is a dedicated Bioconductor package for creating volcano plots specifically called EnhancedVolcano *(https://bioconductor.org/packages/release/bioc/html/EnhancedVolcano.html)*. Plots created using this package can be customized using ggplot2 functions and syntax.

```
#The default cut-off for log2FC is >|2|
#the default cut-off for log10 p-value is 10e-6
library(EnhancedVolcano)
EnhancedVolcano(dexp_sigtrnsc,
                title = "Enhanced Volcano with Airways",
                lab = dexp_sigtrnsc$transcript,
                x = 'logFC',
                y = 'FDR')
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3
i Please use `linewidth` instead.
i The deprecated feature was likely used in the EnhancedVolcano packa
  Please report the issue to the authors.
```



This creates a very nice plot rather quickly.

Adding horizontal and vertical lines

The horizontal and vertical lines can be added to our ggplot2 figure using `geom_hline()` and `geom_vline()`, respectively.

Acknowledgements

The volcano plot code in this lesson was adapted from a 2021 workshop entitled *Introduction to Tidy Transciptomics (https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html)* by Maria Doyle and Stefano Mangiola.

Recommendations and Tips for Creating Effective Plots with ggplot2

Learning Objectives

1. Evaluate general principles and best practices for designing clear, publication-quality figures in ggplot2.
2. Construct multi-panel figures using tools such as patchwork.
3. Identify and explore specialized R packages that support particular plot types.
4. Write simple R functions that wrap ggplot2 code to streamline the creation of repeatable or customized plot templates.

In the previous lessons, we learned the basics of the grammar of graphics. In this lesson, we will focus on miscellaneous topics that will help you in your plot making journey.

Included topics:

- recommendations for publishable figures
- additional packages that enhance ggplot2 functionality (e.g., `patchwork`, `gghighlight`, `ggthemes`, `ggrepel`, `scales`)
- creating plotting functions
- resources for further learning

Recommendations for creating publishable figures

(Inspired by Visualizing Data in the Tidyverse, a Coursera lesson)

1. Consider whether the plot type you have chosen is the best way to convey your message
2. Make your plot visually appealing
    - Careful color selection - color blind friendly if possible (e.g., `library(viridis)`)
    - Eliminate unnecessary white space
    - Carefully choose themes including font types
3. Label all axes with concise and informative labels
    - These labels should be straight forward and adequately describe the data
4. Ask yourself "Does the data make sense?"
    - Does the data plotted address the question you are answering?

5. Try not to mislead the audience
    - Often this means starting the y-axis at 0
    - Keep axes consistent when arranging facets or multiple plots
    - Keep colors consistent across plots
6. Do not try to convey too much information in the same plot
    - Keep plots fairly simple

Complementary or Related Packages

There are many complementary R packages related to creating publishable figures using ggplot2. Check out ggplot2 extensions with the ggplot2 extensions - gallery *(https://exts.ggplot2.tidyverse.org/gallery/)*. By default, these are listed by popularity.

Here is a sampling of data visualization packages you may be interested in:

> **Warning**
>
> These packages do not exclusively use ggplot2 for graphic generation.

Genomics

1. gggenomes *(https://thackl.github.io/gggenomes/)* - extends the grammar of graphics for comparative genomics.
2. GViz *(https://bioconductor.org/packages/release/bioc/vignettes/Gviz/inst/doc/Gviz.html)* - Plotting data and annotation information along genomic coordinates
3. ComplexHeatmap *(https://bioconductor.org/packages/release/bioc/html/ComplexHeatmap.html)* - generate simple or complex heatmaps
4. EnhancedVolcano *(https://bioconductor.org/packages/release/bioc/vignettes/EnhancedVolcano/inst/doc/EnhancedVolcano.html)* - generate high quality, publication ready volcano plots
5. pcaExplorer *(https://www.bioconductor.org/packages/release/bioc/html/pcaExplorer.html)* - general-purpose interactive companion tool for RNA-seq analysis (uses a Shiny application)
6. OmicsCircos *(https://www.cancer.gov/about-nci/organization/cbiit/training/library/omnicircos)* - generate high quality circular plots for omics data.

You may also search for plots using "plot" or "visualization" using Bioconductor: https://bioconductor.org/packages/release/BiocViews.html#___Software *(https://bioconductor.org/packages/release/BiocViews.html#___Software)*

> **Can I add ggplot2 layers?**

There are many -omics related packages that include data visualization wrappers (e.g., DESeq2, Seurat, etc.). These are not visualization specific packages. Many of these functions can be customized by adding ggplot2 layers. How do we know if we can add ggplot layers? Try any / all of the following:

1. Check imports → does package depend on ggplot2? (e.g., `packageDescription("package")$Imports`)
2. Check the source code. Does it use `ggplot2`: (e.g., DESeq2::plotPCA)
    1. Call directly `DESeq2::plotPCA`
    2. showMethods(PlotPCA)
    3. getMethod("plotPCA", "DESeqTransform")
3. Inspect the output object → `class(x)` includes "gg" or "ggplot"?
4. Try adding a layer → does `+ theme_minimal()` work?
5. Read examples/vignettes → do they use + syntax?

Check out this BTEP tutorial *(https://bioinformatics.ccr.cancer.gov/docs/btep-coding-club/CC2023/complex_heatmap_enhanced_volcano/)* on `EnhancedVolcano` and `ComplexHeatmap`.

Statistics integration

1. ggpubr *(https://ggplot2.tidyverse.org/)* - generate out-of-the-box publication quality plots. Includes statistical integration.
    - Coding Club tutorial: https://bioinformatics.ccr.cancer.gov/docs/btep-coding-club/CC2024/ggpubr/Intro_to_ggpubr/ *(https://bioinformatics.ccr.cancer.gov/docs/btep-coding-club/CC2024/ggpubr/Intro_to_ggpubr/)*
2. ggfortify *(https://github.com/sinhrks/ggfortify)* - easily visualize statistical results including PCA.
3. factoextra *(https://rpkgs.datanovia.com/factoextra/index.html)* - visualize multivariate statistics (e.g., PCA).

Combining plots

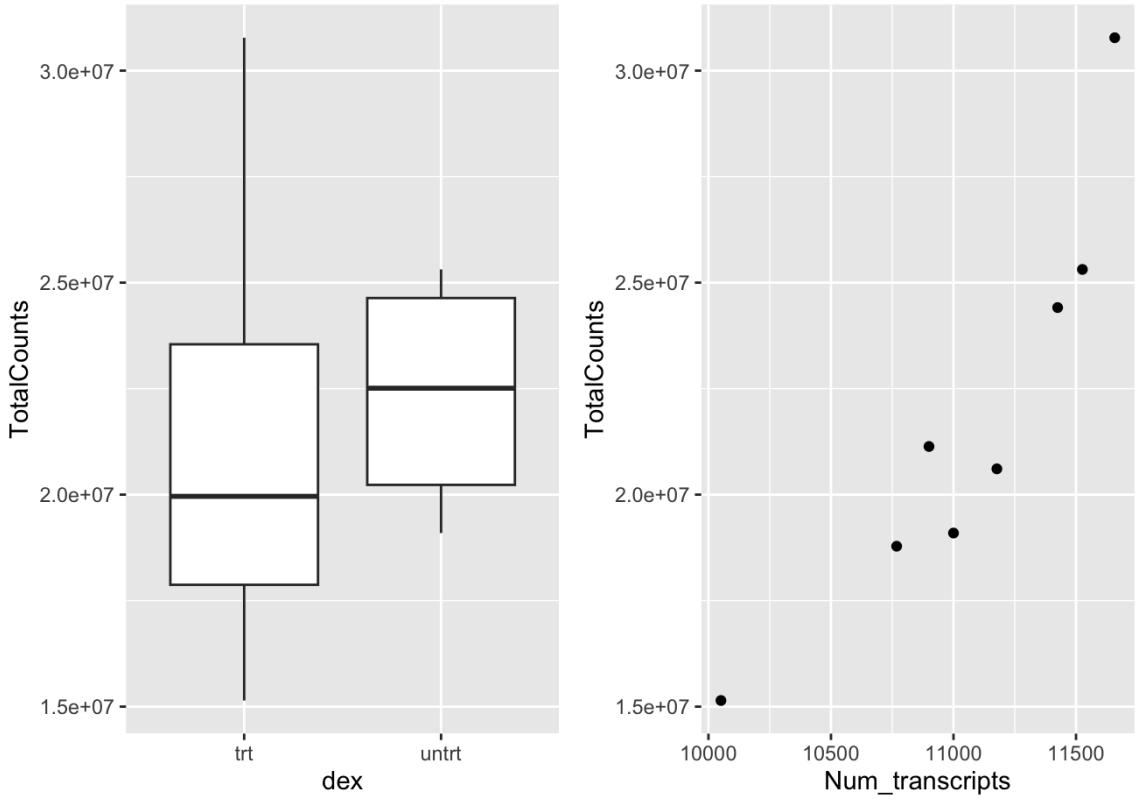
1. patchwork *(https://patchwork.data-imaginist.com/)* - the go-to package for combining plots.

    Example:

```
library(tidyverse)
```

```
library(patchwork)
```

```
sc <- read.csv("./data/sc.csv")

a <- ggplot(data=sc) +
geom_boxplot(aes(x=dex, y = TotalCounts))

b <- ggplot() +
    geom_point(data=sc,aes(x=Num_transcripts, y = TotalCounts))

a + b
```



2. cowplot *(https://wilkelab.org/cowplot/)* - also includes nice themes and annotation functions.

You may find this BTEP tutorial *(https://bioinformatics.ccr.cancer.gov/docs/data-visualization-with-r/Lesson6_V2/)* on combining R graphics useful.

Miscellaneous

1. gghighlight *(https://yutannihilation.github.io/gghighlight/)* - highlight specific points, lines, etc. in a plot
2. scales *(https://scales.r-lib.org/)* - tools for working with ggplot2 scaling infrastructure (funcitons involving `scale`).
3. ggthemes *(https://jrnold.github.io/ggthemes/index.html)* - extra geoms, scales, and themes for ggplot2.

4. ggrepel *(https://ggrepel.slowkow.com/)* - repel overlapping text labels.
5. plotly *(https://plotly.com/ggplot2/)* - create interactive plots (`ggplotly` to work with `ggplot2` plots).

> **Note**
>
> There are many more packages. Shop around, especially if you are interested in plotting a specific data type.

Using ggplot2 in a function

While we have learned how to use existing functions in R, we have not covered writing functions.

The Syntax

The syntax for writing a function is as follows:

```
function(x) {
    body # do something with x
}
```

where `function` is the function used to write the function,
`x` is one or more arguments,
and `body`is the code that performs the function task.

We would name the function by assigning it to an object using `function_name <-`.

Here is an example.

```
add5 <- function(x){
    x+5
}

add5(5)
```

```
[1] 10
```

This function named `add5` simply adds 5 to whatever number we include as an argument.

> **Note**

When you call a function in R, R evaluates all of the arguments before it passes them into the function body (unless you've deliberately delayed evaluation with special tricks like tidy evaluation). This has important implications.

Functions that use ggplot2

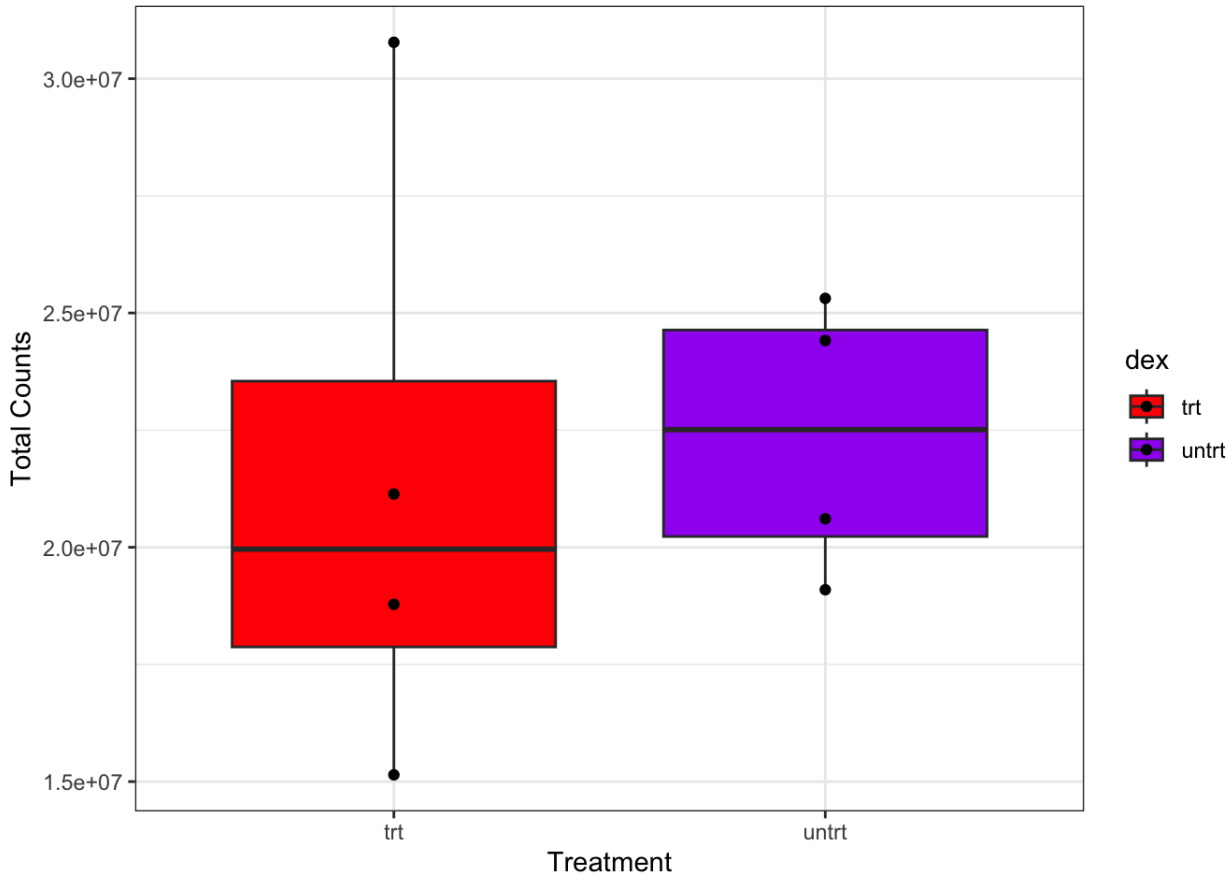
Now that you know the basics, you may be interested in creating a function that will plot different sets of data the same way using `ggplot2`.

However, tidyverse functions use something called "tidy evaluation to allow you to refer to the names of variables inside your data frame without any special treatment" *(https://r4ds.hadley.nz/functions.html#data-frame-functions)*. While there are two types of tidy evaluation to be aware of, data-masking and tidy-selection, these are generally beyond the scope of this lesson. You can learn more about tidy evaluation here *(https://dplyr.tidyverse.org/articles/programming.html)*.

What you really need to know is that when you pass expressions containing column names to a function using tidyverse verbs, including `aes()`. you need to use `{{}}`. Let's see why.

Let's use our data `sc` to create a function that makes a boxplot.

```
my_boxplot<- function(data){
  ggplot(data,aes(x=dex, y = TotalCounts, fill=dex)) +
    geom_boxplot() +
    geom_point() +
    scale_fill_manual(values=c("red","purple"))+
    theme_bw() +
    labs(x="Treatment",y="Total Counts")
}

my_boxplot(sc)
```



Here, we need to supply the data frame to use this function, and everything works fine.

But what if we intend to use this function on a data set where the x variable is not "dex". We want to supply the column name as an argument.

For example,

```
my_boxplot_x<- function(data,x){
  ggplot(data,aes(x=x, y = TotalCounts, fill=dex)) +
    geom_boxplot() +
    geom_point() +
    scale_fill_manual(values=c("red","purple"))+
    theme_bw() +
    labs(x="Treatment",y="Total Counts")


}

my_boxplot_x(sc, dex)
```

```
Error in `geom_boxplot()`:
! Problem while computing aesthetics.
```

```
i Error occurred in the 1st layer.
Caused by error:
! object 'dex' not found
```

We run into an error that says **"object 'dex' not found"**. We know "dex" is in `sc`, so what is happening?

When we run `my_boxplot_x(sc, dex)`, R tries to find an object called `dex` in our global environment, not in `sc`. Because `dex` is not in the global environment, an error is thrown. We need to tell our function to hold off on evaluating the argument right now, rather, capture it as an expression to be evaluated in the right context (inside `aes()`).

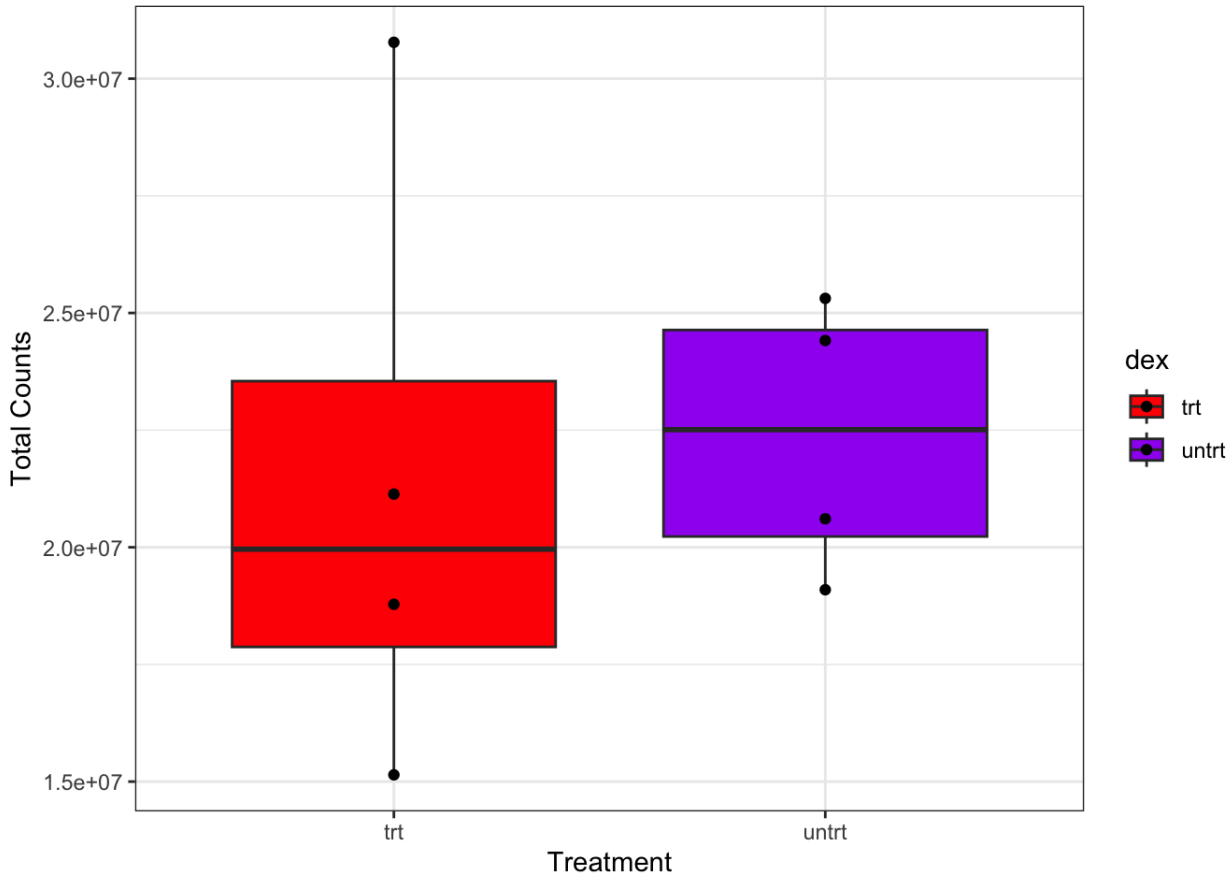
How do we fix this. We use something called embracing. "Embracing a variable means to wrap it in braces so (e.g.) `var` becomes `{{ var }}`. Embracing a variable tells the [Tidyverse] verb to use the value stored inside the argument, not the argument as the literal variable name." *(https://r4ds.hadley.nz/functions.html#indirection-and-tidy-evaluation)*

More on embracing `{{}}`

`{{x}}` is shorthand for `aes(x = !!enquo(x))`. `enquo(x)` captures the unevaluated argument as a quoted expression (a quosure), while `!!` is the unquote operator, which tells tidy evaluation to evaluate and insert that captured expression into the surrounding code.

Let's try embracing the x argument.

```
my_boxplot_x<- function(data,x){
  ggplot(data,aes(x={{x}}, y = TotalCounts, fill=dex)) +
    geom_boxplot() +
    geom_point() +
    scale_fill_manual(values=c("red","purple"))+
    theme_bw() +
    labs(x="Treatment",y="Total Counts")


}
my_boxplot_x(sc, dex)
```



To learn more about writing plotting functions with `ggplot2`, see this chapter *(https://r4ds.hadley.nz/functions.html#plot-functions)* in *R For Data Science* and this vignette *(https://ggplot2.tidyverse.org/articles/ggplot2-in-packages.html)*.

Tips on Saving and Scaling

`ggplot2` comes with its own function for simplified saving, `ggsave()`. When creating plots, we tend to work interactively and save interactively. While you may create the perfect figure at a width of 7 inches and height of 5 inches, this may not scale well (either smaller or larger). For example, you may notice the text becomes very small when the size of your image is scaled up. Text is set using an absolute point size. If you come across this issue, try the suggestions outlined here *(https://tidyverse.org/blog/2020/08/taking-control-of-plot-scaling/)*.

Tips for saving:

1. Use vector graphics (PDF, SVG) to save your figure. You can then scale the size of the image outside of R and maintain proportions and crispness.

2. If you need to use raster graphics (PNG, TIFF, JPEG), which suffer from blurring when resized, use the R package `ragg` for image resizing.

   For example,

```
volcano <- readRDS("./data/Volcano.rds")

volcano

ggsave("png_small.png",width=7,height=5, dpi=300,units="in")

ggsave("scale_png.png", volcano,
    device = ragg::agg_png,
    width = 21, height = 15, units = "in", res = 300,
    scaling = 3)
```

The arguments `res` and `scaling` are specific to `ragg:agg_png`.

Vector vs Raster Graphics

What are vector and raster graphics and why does this matter?

Raster and vector graphics differ in how they represent visual information, and that distinction directly affects how visualizations look and scale. In short, this means that the output format of a plot matters.

Raster graphics are made of a fixed grid of pixels, each storing a color value. Example formats include PNG, TIFF, JPEG. This type of graphic is generally great for photos or heatmaps, but suffers from blurring when resized. **Vector graphics** (e.g., PDF, SVG), in contrast, describe shapes, lines, and text mathematically, so they remain crisp at any zoom level and produce smaller files for simple plots. This matters for data visualization because the choice determines clarity and flexibility. Raster formats are better for complex, image-heavy displays or web use, while vector formats are ideal for reports, publications, and presentations where sharp text and scalable detail are essential.

Finding R packages for Beginners

1. Google Search
    1. Rseek *(https://rseek.org/)* - A special Google-powered search engine that searches R-related websites (CRAN, R-bloggers, Stack Overflow, GitHub, etc.).
2. Repository Search
    1. CRAN *(https://cran.r-project.org/web/packages/index.html)* - try CRAN Task Views
    2. Bioconductor *(https://bioconductor.org/packages/release/BiocViews.html#___Software)* - repository for bioinformatics, genomics, and clinical data analysis.
3. r-universe *(https://r-universe.dev/search)* - a modern R package ecosystem and discovery platform built by the rOpenSci *(https://ropensci.org/)* team. Publish, explore, and evaluate R packages (CRAN and other sources).

4. Blogs

    1. Posit *(https://posit.co/blog/)* - Highlights new Tidyverse and ecosystem tools.
    2. R bloggers *(https://www.r-bloggers.com/)* - aggregates posts from hundreds of R users and developers.
    3. R Weekly *(https://rweekly.org/)* - A weekly digest of new packages, tutorials, and news.

Resources for Further Learning

1. Official ggplot2 documentation - https://ggplot2.tidyverse.org/ *(https://ggplot2.tidyverse.org/)*

2. BTEP

    1. Coding Club *(https://bioinformatics.ccr.cancer.gov/docs/btep-coding-club/)*
    2. Data Visualization with R *(https://bioinformatics.ccr.cancer.gov/docs/data-visualization-with-r/index.html)*

3. Online books / tutorials

4. A self-learning platform (e.g., Coursera)

Practice Exercises

Part 1: Exercises

Exercise 1: Lesson2

Q1. What is the value of each object? Run the code and print the values.

```
mass <- 47.5            # mass?
age  <- 122             # age?
mass <- mass * 2.0      # mass?
age  <- age - 20        # age?
mass_index <- mass / age  # mass_index?
```

(Question taken from https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r/index.html)

Q1: Solution

```
mass <- 47.5            # mass?
mass
## [1] 47.5
age  <- 122             # age?
age
## [1] 122
mass <- mass * 2.0      # mass?
mass
## [1] 95
age  <- age - 20        # age?
age
## [1] 102
mass_index <- mass / age  # mass_index?
mass_index
## [1] 0.9313725
```

Q2. Create the following objects; give each object an appropriate name.

a. Create an object that has the value of the number of bones in the adult human body.

b. We can create a vector of values using `c()`. For example to create a vector of fruits, we could use the following: `fruit <- c("apples", "bananas", "mango", "kiwi")`. Use this information to create an object containing the names of four different bones. (We will learn more about vectors in Lesson 3.)

Q2: Solution

```
# a.
bone_num<- 206
bone_num
```

```
## [1] 206

# b.
bone_names<- c("talus","calcaneus","tibia","fibula")
bone_names
## [1] "talus"     "calcaneus" "tibia"     "fibula"
```

Q3. What types of data are stored in the objects created in question 2.

Q3: Solution

```
typeof(bone_num)
## [1] "double"
typeof(bone_names)
## [1] "character"
```

Q4. Modify `bone_num` to contain the number of bones in an adult human hand.

Q4: Solution

```
bone_num <- 27
bone_num
## [1] 27
```

Q5. Here is an object storing multiple values:

```
num_vec <- c(1:100)
```

What is the mean of this vector? How about the median? What functions can you use to find this information?

Q5: Solution

```
mean(num_vec)
## [1] 50.5
median(num_vec)
## [1] 50.5
```

Q6. What does the function `paste()` do? How can you find out? Can you use it to collapse `bone_names` into a string of length 1? **Hint: Read the help documentation closely.**

Q6: Solution

```
# To find help, use the ?
?paste

# To collapse the vector to length 1, check the collapse argument
paste(bone_names, collapse=", ")
## [1] "talus, calcaneus, tibia, fibula"
length(bone_names)
## [1] 4
```

Exercise 2: Lesson 3

Q1. Let's use some functions.

a. Use `sum()` to add the numbers from 1 to 10.

Q1a: Solution

```
sum(1:10)
## [1] 55
```

b. Compute the base 10 logarithm of the elements in the following vector and save to an object called `logvec`: `c(1:10)`.

Q1b: Solution

```
logvec<- log10(c(1:10))
```

c. Combine the following vectors and compute the mean.

```
a <- c(45, 67, 34, 82)
b <- c(90, 45, 62, 56, 54)
```

Q1c: Solution

```
mean(c(a,b))
## [1] 59.44444
```

d. What does the function `identical()` do? Use it to compare the following vectors.

```
c <- seq(2, 10, by=2)
d <- c(2, 4, 6, 8, 10)
```

Q1d: Solution

```
#tells us whether the two vectors are the same
identical(c, d)
## [1] TRUE
```

Q2. Vectors include data of a single type, so what happens if we mix different types? Use `typeof()` to check the data type of the following objects.

```
num_char <- c(1, 2, 3, "a")
num_logical <- c(1, 2, 3, TRUE, FALSE)
char_logical <- c("a", "b", "c", TRUE)
tricky <- c(1, 2, 3, "4")
```

Q2: Solution

```
#These were coerced into a single data type
typeof(num_char)
## [1] "character"
num_char
## [1] "1" "2" "3" "a"
typeof(num_logical)
## [1] "double"
num_logical
## [1] 1 2 3 1 0
typeof(char_logical)
## [1] "character"
char_logical
## [1] "a"    "b"    "c"    "TRUE"
typeof(tricky)
## [1] "character"
tricky
## [1] "1" "2" "3" "4"
```

(Question taken from https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r.html *(https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r.html)*)

Q3. `fruit` is a vector containing the common names of different types of fruit. Can you replace "kiwi" with "mango".

```
fruit<-c("apples", "bananas", "oranges", "grapes","kiwi","kumquat")
```

Q3: Solution

```
fruit[5] <- "mango"
fruit
## [1] "apples"  "bananas" "oranges" "grapes"  "mango"   "kumquat"
```

Q4. Given the following R code, return all values less than 678 in the vector "Total_subjects".

```
Total_subjects <- c(23, 4, 679, 3427, 12, 890, 654)
```

Q4: Solution

```
Total_subjects[Total_subjects < 678]
## [1]  23   4  12 654
```

Q5. This question uses the vectors created in Q2. Using indexing, create a new vector named `combined` that contains:

The 2nd and 3rd value of `num_char`.
The last value of `char_logical`.
The 1st value of `tricky`.

`combined` contains data of what type?

Q5: Solution

```
combined <- c(num_char[2:3], char_logical[length(char_logical)],
              tricky[1])
typeof(combined)
## [1] "character"
combined
## [1] "2"    "3"    "TRUE" "1"
```

Exercise 3: Lesson 4

Loading data

The data used in this practice exercise can be found here.

Q1. 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`.

Q1: Solution

```
iris_long<-readxl::read_excel("../data/iris_data.xlsx",sheet="iris_data_long",.name
## New names:
## • `Iris ID` -> `Iris.ID`
## • `Measurement location` -> `Measurement.location`
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
```

Q2. 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.

Q2: Solution

```
species<-readr::read_delim("../data/species_datacarpentry.txt",col_types="cfff")
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
```

Q3. Load in a comma separated file with row names present (file_path= "./data/countB.csv") and save to an object named `countB`.

Q3: Solution

```
countB<-read.csv("../data/countB.csv",row.names=1)
head(countB)
##        SampleA_1 SampleA_2 SampleA_3 SampleB_1 SampleB_2 SampleB_3
## Tspan6       703       567       867        71       970       242
## TNMD         490       482        18       342       935       469
## DPM1         921       797       622       661         8       500
## SCYL3        335       216       222       774       979       793
## FGR          574       574       515       584       941       344
## CFH          577       792       672       104       192       936
```

Challenge data load

Q4. 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.

Q4: Solution

```
library(tidyverse)
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.4     ✔ readr     2.1.5
## ✔ forcats   1.0.0     ✔ stringr   1.5.1
## ✔ ggplot2   3.5.2     ✔ tibble    3.2.1
## ✔ lubridate 1.9.4     ✔ tidyr     1.3.1
## ✔ purrr     1.0.4
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all confl
read_delim("../data/WebexSession_report.txt",delim="\t",locale = locale(encoding =
## Rows: 10 Columns: 21
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: "\t"
## chr   (7): Name, Date, Invited, Registered, Duration, Network joined from:, ...
## dbl   (1): Participant
## lgl  (11): Audio Type, Email, Company, Title, Phone Number, Address 1, Addre...
## time  (2): Start time, End time
```

```
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message
## # A tibble: 10 × 21
##    Participant `Audio Type` Name     Email Date  Invited Registered `Start time`
##          <dbl> <lgl>        <chr>    <lgl> <chr> <chr>   <chr>      <time>
##  1           1 NA           Partici… NA    6/8/… No      N/A        13:00
##  2           2 NA           Partici… NA    6/9/… <NA>    <NA>       13:00
##  3           3 NA           Partici… NA    6/10… No      N/A        12:57
##  4           4 NA           Partici… NA    6/11… <NA>    <NA>       12:57
##  5           5 NA           Partici… NA    6/12… No      N/A        12:55
##  6           6 NA           Partici… NA    6/13… <NA>    <NA>       12:55
##  7           7 NA           Partici… NA    6/14… No      N/A        12:32
##  8           8 NA           Partici… NA    6/15… <NA>    <NA>       12:32
##  9           9 NA           Partici… NA    6/16… Yes     N/A        12:42
## 10          NA NA           <NA>     NA    <NA>  <NA>    <NA>         NA
## # ℹ 13 more variables: `End time` <time>, Duration <chr>, Company <lgl>,
## #   Title <lgl>, `Phone Number` <lgl>, `Address 1` <lgl>, `Address 2` <lgl>,
## #   City <lgl>, `State/Province` <lgl>, `Zip/Postal Code` <lgl>,
## #   `Country/region` <lgl>, `Network joined from:` <chr>,
## #   `Internal Participant:` <chr>
```

Exercise 4: Lesson 5

For this exercise we will use `filtlowabund_scaledcounts_airways.txt`, which includes normalized and non-normalized transcript count data from an RNAseq experiment. You can read more about the experiment here *(https://pubmed.ncbi.nlm.nih.gov/24926665/)*. To obtain this file, click here.

The following questions synthesize several of the skills you have learned thus far. It may not be immediately apparent how you would go about answering these questions. Remember, the R community is expansive, and there are a number of ways to get help including but not limited to google search. These questions have multiple solutions, but you should try to stick to the tools you have learned to use thus far.

Q1. Import `filtlowabund_scaledcounts_airways.txt` into R and save to an R object named `transcript_counts`. Try not to use the drop-down menu for loading the data.

Q1 Solution

```
transcript_counts <-read.delim("../data/filtlowabund_scaledcounts_airways.txt")
```

Q2. What are the dimensions of `transcript_counts`?

Q2 Solution

```
dim(transcript_counts)
## [1] 127408     18
```

Q3. What are the column names?

Q3 Solution

```
colnames(transcript_counts)
##  [1] "feature"       "sample"        "counts"        "SampleName"
##  [5] "cell"          "dex"           "albut"         "Run"
##  [9] "avgLength"     "Experiment"    "Sample"        "BioSample"
## [13] "transcript"    "ref_genome"    ".abundant"     "TMM"
## [17] "multiplier"    "counts_scaled"
```

Q4. Is there a difference in the number of transcripts with greater than 0 normalized counts (counts_scaled) per sample? What commands did you use to answer this question.

Q4 Solution

```
#using table
table(transcript_counts[transcript_counts$counts_scaled>0,]$sample)
##
##   508   509   512   513   516   517   520   521
## 15921 15919 15923 15918 15913 15920 15914 15910

#alternative solution
summary(factor(transcript_counts[transcript_counts$counts_scaled>0,]$sample))
##   508   509   512   513   516   517   520   521
## 15921 15919 15923 15918 15913 15920 15914 15910

# or using the tidyverse
library(dplyr)
transcript_counts %>% filter(counts_scaled>0) %>% count(sample)
##   sample     n
## 1    508 15921
## 2    509 15919
## 3    512 15923
## 4    513 15918
## 5    516 15913
## 6    517 15920
## 7    520 15914
## 8    521 15910
```

Q5. How many categories of transcripts are there? Think about what you know regarding factors. Why is this number much smaller than the results of question 4?

Q5 Solution

```
nlevels(factor(transcript_counts$transcript, exclude = NULL))
## [1] 14576
```

Q6. Subset transcript_counts to only include the following columns: sample, cell, dex, transcript, avgLength, counts_scaled. Save this new dataframe to a new object called `transc_df`.

Q6 Solution

```
transc_df <- transcript_counts[c("sample","cell","dex",
                                 "transcript","avgLength",
                                 "counts_scaled")]
```

Q7. Using your new data frame from question six (`transc_df`), rename the column "sample" to "Sample".

Q7 Solution

```
colnames(transc_df)[1]<-"Sample"
```

Q8. What is the mean and standard deviation of "avgLength" across the entire `transc_df` data frame? Hint: Read the help documentation for `mean()` and `sd()`.

Q8 Solution

```
mean_avgLength<- mean(transc_df$avgLength)
sd_avgLength<- sd(transc_df$avgLength)
```

Q9. Make a data frame with the column names "Mean" and "Standard_Dev" that holds the values from question 8. Hint: check out the function `data.frame()`.

Q9 Solution

```
data.frame(Mean=mean_avgLength, Standard_Dev=sd_avgLength)
##     Mean Standard_Dev
## 1 113.75     14.85561
```

Part 2: Exercises

Data Reshape

Q1. 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`.

Q1 Solution

```
library(tidyverse)
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.4     ✔ readr     2.1.5
## ✔ forcats   1.0.0     ✔ stringr   1.5.1
## ✔ ggplot2   3.5.2     ✔ tibble    3.3.0
## ✔ lubridate 1.9.4     ✔ tidyr     1.3.1
## ✔ purrr     1.0.4
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all confl
iris_long<-readxl::read_excel("./data/iris_data.xlsx",sheet="iris_data_long",.name_
## New names:
## • `Iris ID` -> `Iris.ID`
## • `Measurement location` -> `Measurement.location`

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
```

Q2. 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 Petal.Width
     <dbl> <chr>          <dbl>       <dbl>        <dbl>       <dbl>
 1       1 setosa           5.1         3.5          1.4         0.2
 2       2 setosa           4.9         3            1.4         0.2
```

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

Q2 Solution

```
tidyr::pivot_wider(iris_long, names_from = Measurement.location, values_from = Meas
## # A tibble: 150 × 6
##    Iris.ID Species Sepal.Length Sepal.Width Petal.Length Petal.Width
##      <dbl> <chr>          <dbl>       <dbl>        <dbl>       <dbl>
##  1       1 setosa           5.1         3.5          1.4         0.2
##  2       2 setosa           4.9         3            1.4         0.2
##  3       3 setosa           4.7         3.2          1.3         0.2
##  4       4 setosa           4.6         3.1          1.5         0.2
##  5       5 setosa           5           3.6          1.4         0.2
##  6       6 setosa           5.4         3.9          1.7         0.4
##  7       7 setosa           4.6         3.4          1.4         0.3
##  8       8 setosa           5           3.4          1.5         0.2
##  9       9 setosa           4.4         2.9          1.4         0.2
## 10      10 setosa           4.9         3.1          1.5         0.1
## # ℹ 140 more rows
```

Q3. 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`.

```
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
```

The resulting data frame should appear as follows:

```
# 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
```

Q3 Solution

```
pivot_longer(table4a,2:3, names_to = "year", values_to = "cases")
## # 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
```

Q4. 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
```

The result should appear as follows:

```
# 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
```

Q4 Solution

```
separate(table3, rate, into = c("cases", "population"))
## # 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

Q5 Use `pivot_longer` to reshape countB. You will need to import countB (file_path = "./data/countB.csv"). Your reshaped data should look the same as the data below.

```
# 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
```

Q5 Solution

```
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 = "_")

tibble(countB_l)
## # 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
```

Select and Filter

All solutions use the pipe. Solutions have multiple possibilities.

Q1. Import the file "./data/filtlowabund_scaledcounts_airways.txt" and save to an object named `sc`. Create a 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"?

Q1 Solution

```
library(tidyverse)
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.4     ✔ readr     2.1.5
## ✔ forcats   1.0.0     ✔ stringr   1.5.1
## ✔ ggplot2   3.5.2     ✔ tibble    3.3.0
## ✔ lubridate 1.9.4     ✔ tidyr     1.3.1
## ✔ purrr     1.0.4
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all confl

sc <- read_delim("../data/filtlowabund_scaledcounts_airways.txt")
## Rows: 127408 Columns: 18
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: "\t"
## chr (11): feature, SampleName, cell, dex, albut, Run, Experiment, Sample, Bi...
## dbl  (6): sample, counts, avgLength, TMM, multiplier, counts_scaled
## lgl  (1): .abundant
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this messag

cnames <- c('sample', 'cell', 'dex', 'transcript', 'counts_scaled')

sc %>% select(all_of(cnames)) %>% filter(dex == "untrt" & (transcript %in% c("ACTN1
## # A tibble: 8 × 5
##   sample cell    dex   transcript counts_scaled
##    <dbl> <chr>   <chr> <chr>              <dbl>
## 1    508 N61311  untrt ANAPC4               777.
## 2    508 N61311  untrt ACTN1              14410.
## 3    512 N052611 untrt ANAPC4               786.
## 4    512 N052611 untrt ACTN1              16644.
## 5    516 N080611 untrt ANAPC4               709.
## 6    516 N080611 untrt ACTN1              15805.
## 7    520 N061011 untrt ANAPC4               827.
## 8    520 N061011 untrt ACTN1              16015.
```

Q2. 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()`.

Q2 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 this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message
top5<- dexp %>%
  dplyr::filter(abs(logFC) > 2) %>%
  slice_min(n=5,order_by=FDR, with_ties=FALSE)
top5
## # A tibble: 5 × 10
##   feature      albut transcript ref_genome .abundant logFC logCPM     F   PValue
##   <chr>        <chr> <chr>      <chr>      <lgl>     <dbl>  <dbl> <dbl>    <dbl>
## 1 ENSG0000010… untrt ZBTB16     hg38       TRUE       7.15   4.15 1429. 5.11e-11
## 2 ENSG0000016… untrt CACNB2     hg38       TRUE       3.28   4.51 1575. 3.34e-11
## 3 ENSG0000012… untrt DUSP1      hg38       TRUE       2.94   7.31  694. 1.18e- 9
## 4 ENSG0000014… untrt PRSS35     hg38       TRUE      -2.76   3.91  807. 6.16e-10
## 5 ENSG0000015… untrt SPARCL1    hg38       TRUE       4.56   5.53  721. 1.00e- 9
## # ℹ 1 more variable: FDR <dbl>
```

Q3. Filter `sc` to contain only the top 5 differentially expressed genes.

Q3 Solution

```
sc %>% dplyr::filter(transcript %in%  top5$transcript)
## # A tibble: 40 × 18
##    feature sample counts SampleName cell  dex   albut Run   avgLength Experiment
##    <chr>    <dbl>  <dbl> <chr>      <chr> <chr> <chr> <chr>     <dbl> <chr>
##  1 ENSG00…    508      4 GSM1275862 N613… untrt untrt SRR1…       126 SRX384345
##  2 ENSG00…    508    665 GSM1275862 N613… untrt untrt SRR1…       126 SRX384345
##  3 ENSG00…    508    330 GSM1275862 N613… untrt untrt SRR1…       126 SRX384345
##  4 ENSG00…    508     62 GSM1275862 N613… untrt untrt SRR1…       126 SRX384345
##  5 ENSG00…    508     80 GSM1275862 N613… untrt untrt SRR1…       126 SRX384345
##  6 ENSG00…    509    739 GSM1275863 N613… trt   untrt SRR1…       126 SRX384346
##  7 ENSG00…    509   5020 GSM1275863 N613… trt   untrt SRR1…       126 SRX384346
##  8 ENSG00…    509     41 GSM1275863 N613… trt   untrt SRR1…       126 SRX384346
##  9 ENSG00…    509   2040 GSM1275863 N613… trt   untrt SRR1…       126 SRX384346
## 10 ENSG00…    509    731 GSM1275863 N613… trt   untrt SRR1…       126 SRX384346
## # ℹ 30 more rows
## # ℹ 8 more variables: Sample <chr>, BioSample <chr>, transcript <chr>,
```

```
## #   ref_genome <chr>, .abundant <lgl>, TMM <dbl>, multiplier <dbl>,
## #   counts_scaled <dbl>
```

Q4. Select only columns of type character from `sc`.

Q4 Solution

```
sc %>% select(where(is.character))
## # A tibble: 127,408 × 11
##    feature         SampleName cell  dex   albut Run   Experiment Sample BioSample
##    <chr>           <chr>      <chr> <chr> <chr> <chr> <chr>      <chr>  <chr>
##  1 ENSG000000000… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  2 ENSG000000004… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  3 ENSG000000004… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  4 ENSG000000004… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  5 ENSG000000009… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  6 ENSG000000010… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  7 ENSG000000010… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  8 ENSG000000011… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
##  9 ENSG000000014… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
## 10 ENSG000000014… GSM1275862 N613… untrt untrt SRR1… SRX384345  SRS50… SAMN0242…
## # ℹ 127,398 more rows
## # ℹ 2 more variables: transcript <chr>, ref_genome <chr>
```

Q5. Select all columns from `dexp` except `.abundant` and `PValue`. Keep only rows with `FDR` less than or equal to 0.01.

Q5 Solution

```
dexp %>% select(-c(.abundant,PValue)) %>% filter(FDR <= 0.01)
## # A tibble: 2,763 × 8
##    feature         albut transcript ref_genome  logFC logCPM     F       FDR
##    <chr>           <chr> <chr>      <chr>       <dbl>  <dbl> <dbl>     <dbl>
##  1 ENSG00000000003 untrt TSPAN6     hg38       -0.390  5.06   32.8 0.00283
##  2 ENSG00000000971 untrt CFH        hg38        0.417  8.09   29.3 0.00376
##  3 ENSG00000001167 untrt NFYA       hg38       -0.509  4.13   44.9 0.00126
##  4 ENSG00000002834 untrt LASP1      hg38        0.388  8.39   22.7 0.00722
##  5 ENSG00000003096 untrt KLHL13     hg38       -0.949  4.16   84.8 0.000234
##  6 ENSG00000003402 untrt CFLAR      hg38        1.18   6.90  130.  0.0000800
##  7 ENSG00000003987 untrt MTMR7      hg38        0.993  0.341  24.7 0.00585
##  8 ENSG00000004059 untrt ARF5       hg38        0.358  5.84   30.9 0.00328
##  9 ENSG00000004487 untrt KDM1A      hg38       -0.308  5.86   23.5 0.00663
## 10 ENSG00000004700 untrt RECQL      hg38        0.360  5.60   22.7 0.00721
## # ℹ 2,753 more rows
```

Q6. Import the file "./data/airway_rawcount.csv". Use the function `rename()` to rename the first column. Use the pipe to import and rename successively without intermediate steps or function nesting. Save to an object named `acount`.

Q6 Solution

```
acount<-read_csv("../data/airway_rawcount.csv") %>%
  dplyr::rename(Feature = ...1)
## New names:
## Rows: 64102 Columns: 9
## ── Column specification
## ──────────────────────────────────────────────────────── Delimiter: "," chr
## (1): ...1 dbl (8): SRR1039508, SRR1039509, SRR1039512, SRR1039513, SRR1039516,
## SRR1039...
## ℹ Use `spec()` to retrieve the full column specification for this data. ℹ
## Specify the column types or set `show_col_types = FALSE` to quiet this message.
## • `` -> `...1`
acount
## # A tibble: 64,102 × 9
##    Feature     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR103951
##    <chr>            <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
##  1 ENSG000000…        679        448        873        408       1138       1047
##  2 ENSG000000…          0          0          0          0          0          0
##  3 ENSG000000…        467        515        621        365        587        799
##  4 ENSG000000…        260        211        263        164        245        331
##  5 ENSG000000…         60         55         40         35         78         63
##  6 ENSG000000…          0          0          2          0          1          0
##  7 ENSG000000…       3251       3679       6177       4252       6721      11027
##  8 ENSG000000…       1433       1062       1733        881       1424       1439
##  9 ENSG000000…        519        380        595        493        820        714
## 10 ENSG000000…        394        236        464        175        658        584
## # ℹ 64,092 more rows
## # ℹ 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

Q7. Use filter on the object `acount` to keep only genes that had a count greater than 10 in at least one sample.

Q7 Solution

```
acount %>%
  filter(if_any(where(is.numeric), ~.> 10))
## # A tibble: 17,792 × 9
##    Feature     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR103951
##    <chr>            <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
##  1 ENSG000000…        679        448        873        408       1138       1047
##  2 ENSG000000…        467        515        621        365        587        799
##  3 ENSG000000…        260        211        263        164        245        331
##  4 ENSG000000…         60         55         40         35         78         63
##  5 ENSG000000…       3251       3679       6177       4252       6721      11027
##  6 ENSG000000…       1433       1062       1733        881       1424       1439
##  7 ENSG000000…        519        380        595        493        820        714
##  8 ENSG000000…        394        236        464        175        658        584
##  9 ENSG000000…        172        168        264        118        241        216
## 10 ENSG000000…       2112       1867       5137       2657       2735       2751
## # ℹ 17,782 more rows
## # ℹ 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

Q8. 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()`. For an alternative solution, check out the docs from rowwise operations *(https://dplyr.tidyverse.org/articles/rowwise.html)*.

Q8 Solution

```
f_acount<- acount %>% column_to_rownames("Feature") %>% filter(rowSums(.) > 10)

# Alternatively

f_acount2<- acount %>% filter(rowSums(pick(where(is.numeric))) > 10)
```

Group_by, Summarize, Arrange

We will continue with `penguins` for this exercise. Questions and solutions (Q1-Q3) were taken from https://allisonhorst.shinyapps.io/dplyr-learnr/#section-dplyrgroup_by-summarize *(https://allisonhorst.shinyapps.io/dplyr-learnr/#section-dplyrgroup_by-summarize)*.

First, let's convert `penguins` to a tibble.

```
penguins <- dplyr::as_tibble(penguins)
```

Q1: Use `group_by()` and `summarize()` to obtain the mean and standard deviation of penguin bill length, grouped by penguin species and sex.

Q1: Solution

```
library(tidyverse)
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.4     ✔ readr     2.1.5
## ✔ forcats   1.0.0     ✔ stringr   1.5.1
## ✔ ggplot2   3.5.2     ✔ tibble    3.3.0
## ✔ lubridate 1.9.4     ✔ tidyr     1.3.1
## ✔ purrr     1.0.4
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all confl
penguins %>%
  group_by(species, sex) %>%
  summarize(bill_length_mean = mean(bill_len, na.rm = TRUE),
            bill_length_sd = sd(bill_len, na.rm = TRUE))
## `summarise()` has grouped output by 'species'. You can override using the
## `.groups` argument.
## # A tibble: 8 × 4
## # Groups:   species [3]
##   species   sex    bill_length_mean bill_length_sd
##   <fct>     <fct>             <dbl>          <dbl>
## 1 Adelie    female             37.3           2.03
## 2 Adelie    male               40.4           2.28
## 3 Adelie    <NA>               37.8           2.80
## 4 Chinstrap female             46.6           3.11
## 5 Chinstrap male               51.1           1.56
## 6 Gentoo    female             45.6           2.05
## 7 Gentoo    male               49.5           2.72
## 8 Gentoo    <NA>               45.6           1.37
```

Q2: Use `group_by()` and `summarize()` to prepare a summary table containing the maximum and minimum flipper length for male Adelie penguins, grouped by island.

Q2: Solution

```
penguins %>%
  filter(species == "Adelie", sex == "male") %>%
  group_by(island) %>%
  summarize(flip_max_length = max(flipper_len),
            flip_min_length = min(flipper_len))
## # A tibble: 3 × 3
##   island    flip_max_length flip_min_length
##   <fct>               <int>           <int>
## 1 Biscoe                203             180
## 2 Dream                 208             178
## 3 Torgersen             210             181
```

Q3: Starting with penguins, create a summary table containing the maximum and minimum length of flippers (call the columns "flip_max" and "flip_min") for chinstrap penguins, grouped by island.

Q3: Solution

```
penguins %>%
  filter(species == "Chinstrap") %>%
  group_by(island) %>%
  summarize(flip_max = max(flipper_len),
            flip_min = min(flipper_len))
## # A tibble: 1 × 3
##   island flip_max flip_min
##   <fct>     <int>    <int>
## 1 Dream       212      178
```

Q4. Create a data frame reordering `penguins` by year, island, and sex.

Q4: Solution

```
penguins %>% arrange(year, island, sex)
## # A tibble: 344 × 8
##    species island bill_len bill_dep flipper_len body_mass sex     year
##    <fct>   <fct>     <dbl>    <dbl>       <int>     <int> <fct>  <int>
##  1 Adelie  Biscoe     37.8     18.3         174      3400 female  2007
##  2 Adelie  Biscoe     35.9     19.2         189      3800 female  2007
##  3 Adelie  Biscoe     35.3     18.9         187      3800 female  2007
##  4 Adelie  Biscoe     40.5     17.9         187      3200 female  2007
##  5 Adelie  Biscoe     37.9     18.6         172      3150 female  2007
##  6 Gentoo  Biscoe     46.1     13.2         211      4500 female  2007
##  7 Gentoo  Biscoe     48.7     14.1         210      4450 female  2007
##  8 Gentoo  Biscoe     46.5     13.5         210      4550 female  2007
##  9 Gentoo  Biscoe     45.4     14.6         211      4800 female  2007
## 10 Gentoo  Biscoe     43.3     13.4         209      4400 female  2007
## # ℹ 334 more rows
```

Q5. Create a data frame containing male Adelie penguins reordered by body_mass in descending order.

Q5: Solution

```
penguins %>%
  filter(species == "Adelie", sex == "male") %>%
  arrange(desc(body_mass))
## # A tibble: 73 × 8
##    species island    bill_len bill_dep flipper_len body_mass sex    year
##    <fct>   <fct>        <dbl>    <dbl>       <int>     <int> <fct> <int>
##  1 Adelie  Biscoe        43.2     19           197      4775 male   2009
##  2 Adelie  Biscoe        41       20           203      4725 male   2009
##  3 Adelie  Torgersen     42.9     17.6         196      4700 male   2008
##  4 Adelie  Torgersen     39.2     19.6         195      4675 male   2007
##  5 Adelie  Dream         39.8     19.1         184      4650 male   2007
##  6 Adelie  Dream         39.6     18.8         190      4600 male   2007
##  7 Adelie  Biscoe        45.6     20.3         191      4600 male   2009
##  8 Adelie  Torgersen     42.5     20.7         197      4500 male   2007
##  9 Adelie  Dream         37.5     18.5         199      4475 male   2009
## 10 Adelie  Torgersen     41.8     19.4         198      4450 male   2008
## # ℹ 63 more rows
```

Mutate and Wrangle Challenge

Let's grab some data to work with.

```
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
```

Q1. 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`.

Q1: Solution

```
log10counts<- acount %>%
  mutate(across(where(is.numeric),~log10(.x+1)))
log10counts
## # A tibble: 64,102 × 9
##    Feature     SRR1039508 SRR1039509 SRR1039512 SRR1039513 SRR1039516 SRR1039517
##    <chr>            <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
##  1 ENSG000000…       2.83       2.65       2.94       2.61       3.06       3.02
##  2 ENSG000000…       0          0          0          0          0          0
##  3 ENSG000000…       2.67       2.71       2.79       2.56       2.77       2.96
##  4 ENSG000000…       2.42       2.33       2.42       2.22       2.39       2.52
##  5 ENSG000000…       1.79       1.75       1.61       1.56       1.90       1.81
##  6 ENSG000000…       0          0          0.477      0          0.301      0
##  7 ENSG000000…       3.51       3.57       3.79       3.63       3.83       4.04
##  8 ENSG000000…       3.16       3.03       3.24       2.95       3.15       3.16
##  9 ENSG000000…       2.72       2.58       2.78       2.69       2.91       2.85
## 10 ENSG000000…       2.60       2.37       2.67       2.25       2.82       2.77
## # ℹ 64,092 more rows
## # ℹ 2 more variables: SRR1039520 <dbl>, SRR1039521 <dbl>
```

Q2. Create a column in `dexp` 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".

Q2: Solution

```
dexp_new<-dexp %>%
  mutate(Expression=case_when(logFC < -1 ~ "Down-regulated",
                              logFC > 1 ~ "Up-regulated",
                              .default = "None")
  )
```

Challenge question:

Q3. Calculate the mean raw counts for each gene ("Feature") by treatment ("dex") in `acount_smeta`. Combine these results with the differential expression results. Your resulting data frame should resemble the following:

```
# A tibble: 15,926 × 12
   Feature         Mean_Counts_trt Mean_Counts_untrt albut transcript
   <chr>                     <dbl>             <dbl> <chr> <chr>
 1 ENSG00000000003           619.              865   untrt TSPAN6
 2 ENSG00000000419           547.              523   untrt DPM1
 3 ENSG00000000457           234.              250.  untrt SCYL3
 4 ENSG00000000460            53.2              63.5 untrt C1orf112
 5 ENSG00000000971          6738.             5331.  untrt CFH
 6 ENSG00000001036          1123.             1487.  untrt FUCA2
 7 ENSG00000001084           573.              658.  untrt GCLC
 8 ENSG00000001167           316               469   untrt NFYA
 9 ENSG00000001460           168.              208   untrt STPG1
10 ENSG00000001461          2545              3113.  untrt NIPAL3
# ℹ 15,916 more rows
# ℹ 6 more variables: .abundant <lgl>, logFC <dbl>, logCPM <dbl>, F
#   PValue <dbl>, FDR <dbl>
```

```
Rows: 15,926
Columns: 12
$ Feature           <chr> "ENSG00000000003", "ENSG00000000419", "ENS
$ Mean_Counts_trt   <dbl> 618.75, 546.75, 233.75, 53.25, 6738.25, 11
$ Mean_Counts_untrt <dbl> 865.00, 523.00, 250.25, 63.50, 5331.25, 14
$ albut             <chr> "untrt", "untrt", "untrt", "untrt", "untrt
$ transcript        <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH
$ ref_genome        <chr> "hg38", "hg38", "hg38", "hg38", "hg38", "hg
$ .abundant         <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,
```

```
$ logFC              <dbl> -0.390100222, 0.197802179, 0.029160865, -0
$ logCPM             <dbl> 5.059704, 4.611483, 3.482462, 1.473375, 8.(
$ F                  <dbl> 3.284948e+01, 6.903534e+00, 9.685073e-02, 1
$ PValue             <dbl> 0.0003117656, 0.0280616149, 0.7629129276, (
$ FDR                <dbl> 0.002831504, 0.077013489, 0.844247837, 0.68
```

Q3: 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 using the `.groups`
## argument.
```

Q4. **If you are interested in practicing data wrangling further, try this wrangling challenge *(https://bioinformatics.ccr.cancer.gov/docs/data-wrangle-with-r/Lesson8/#wrangling-a-realistic-dataset)*.**

Part 3: Exercises

Lesson 1 Exercise Questions: ggplot2 basics

These exercise questions should be attempted after completing Lesson 1: Introduction to ggplot2 for R Data Visualization.

Q1: What geoms would you use to draw each of the following named plots?

a. Scatterplot
b. Line chart
c. Histogram
d. Bar chart
e. Pie chart

(Question taken from https://ggplot2-book.org/individual-geoms.html *(https://ggplot2-book.org/individual-geoms.html)*.)

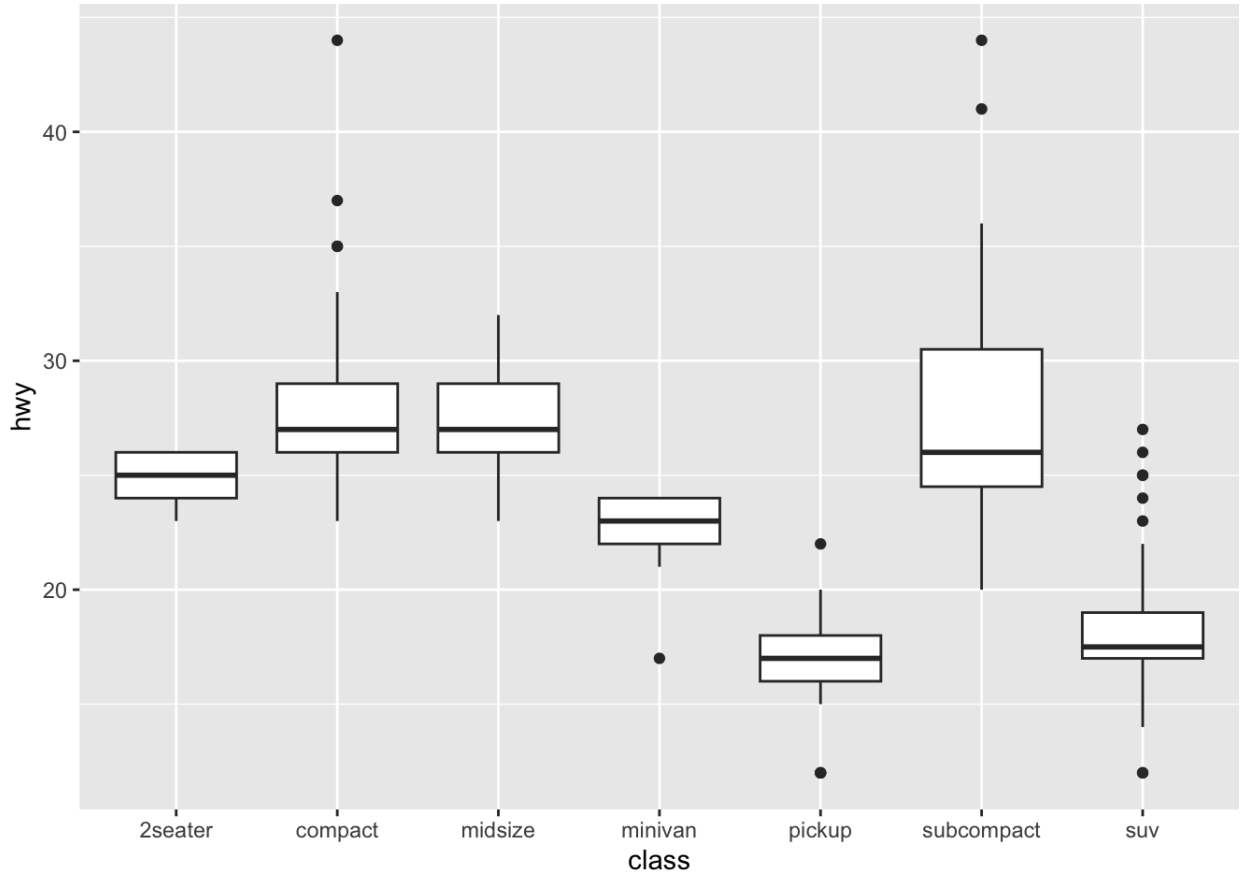
Q1: Solution

a. geom_point
b. geom_line
c. geom_histogram
d. geom_bar
c. geom_bar with coord_polar

Q2. **We will use the `mpg` data set for the remainder of the questions. Use `?mpg` to learn more about these data.** Visualize highway miles per gallon (`hwy`) by the `class` of car using a box plot.

Q2: Solution

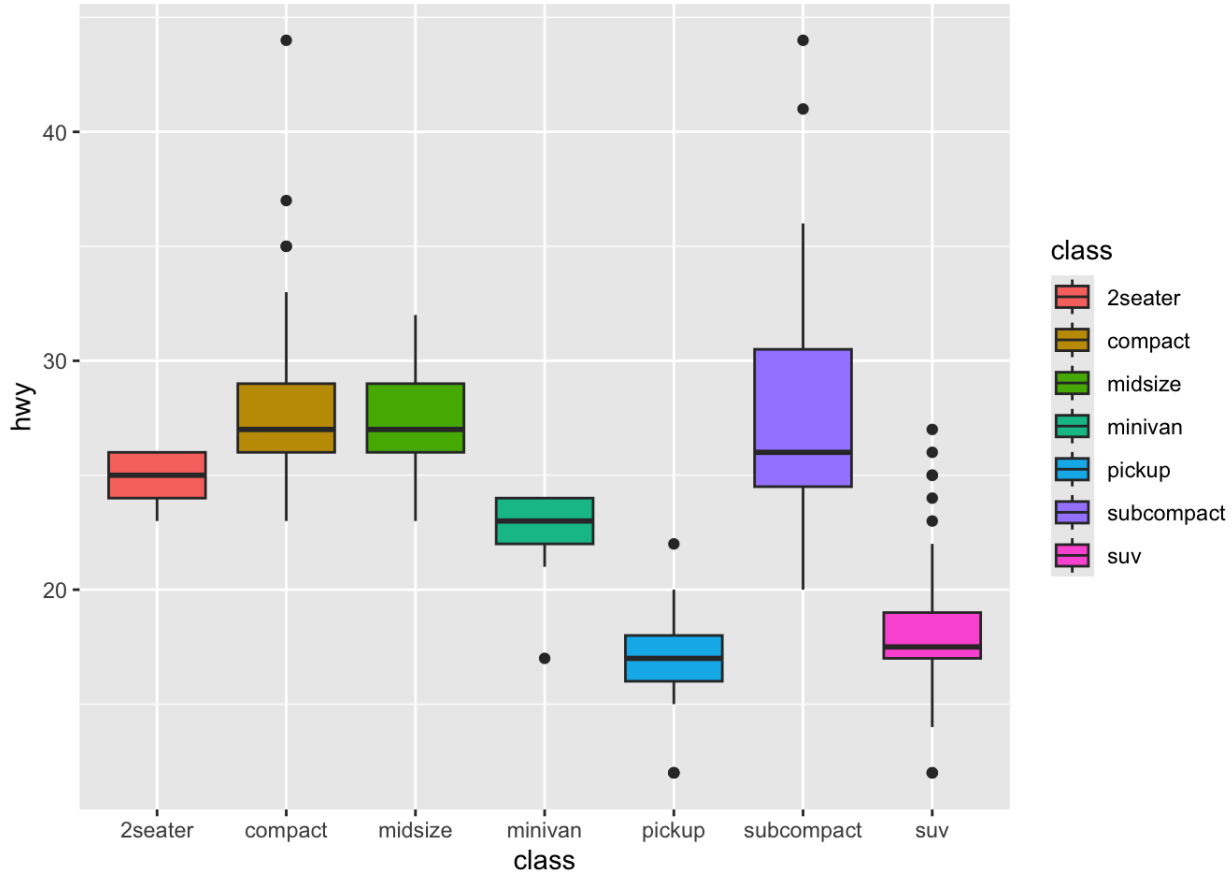
```
library(ggplot2)

ggplot(mpg)+
  geom_boxplot(aes(class,hwy))
```



Q3. Using the plot from Q2, fill each box with color by `class`.

Q3: Solution

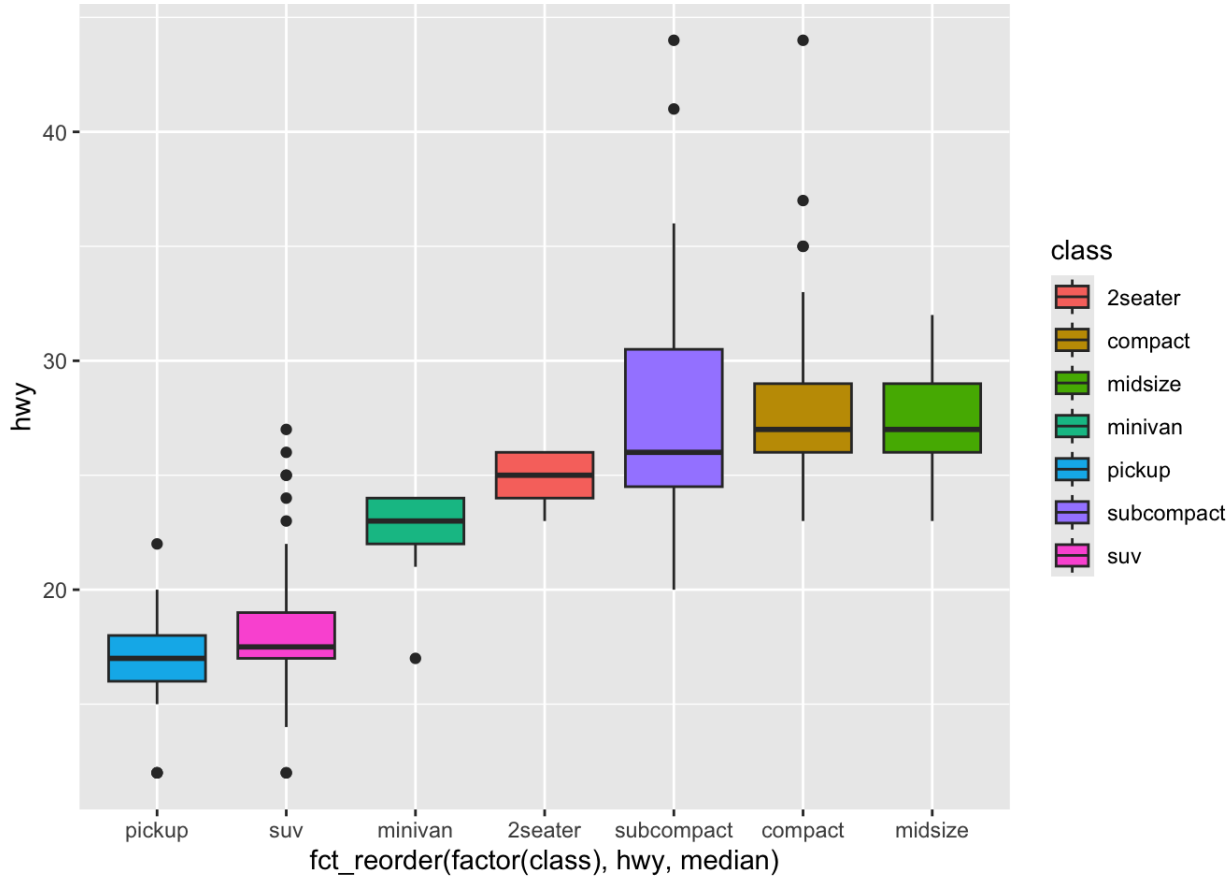
```
ggplot(mpg)+
  geom_boxplot(aes(class,hwy,fill=class))
```



Q4. Challenge Question: Using the plot from Q3, reorder the boxes by the median of `hwy`. Hint: See `fct_reorder()` from `forcats`.

Q4: Solution

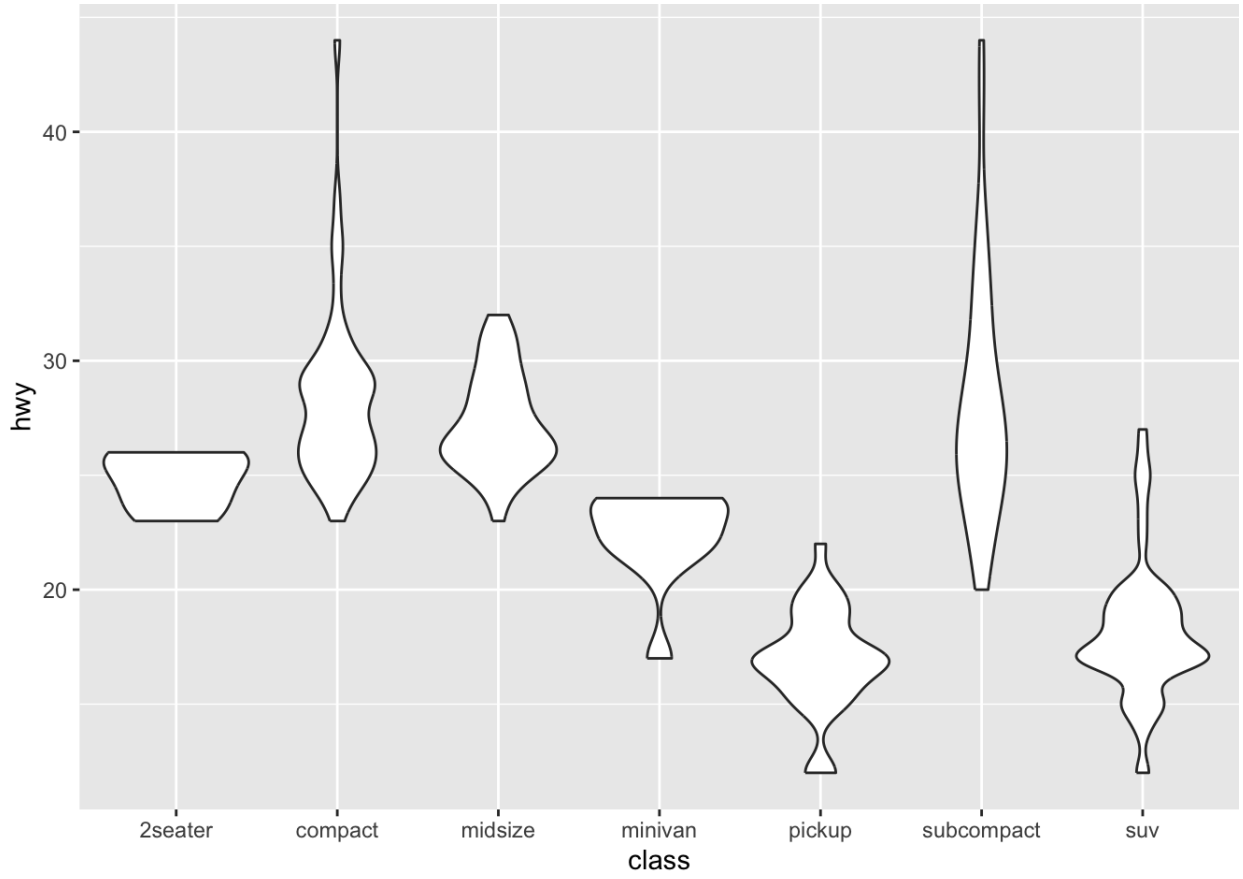
```
library(forcats)
ggplot(mpg)+
  geom_boxplot(aes(fct_reorder(factor(class),hwy,median),hwy,fill=class))
```



Q5. Visualize highway miles per gallon (`hwy`) by the `class` of car using a violin plot.

Q5: Solution

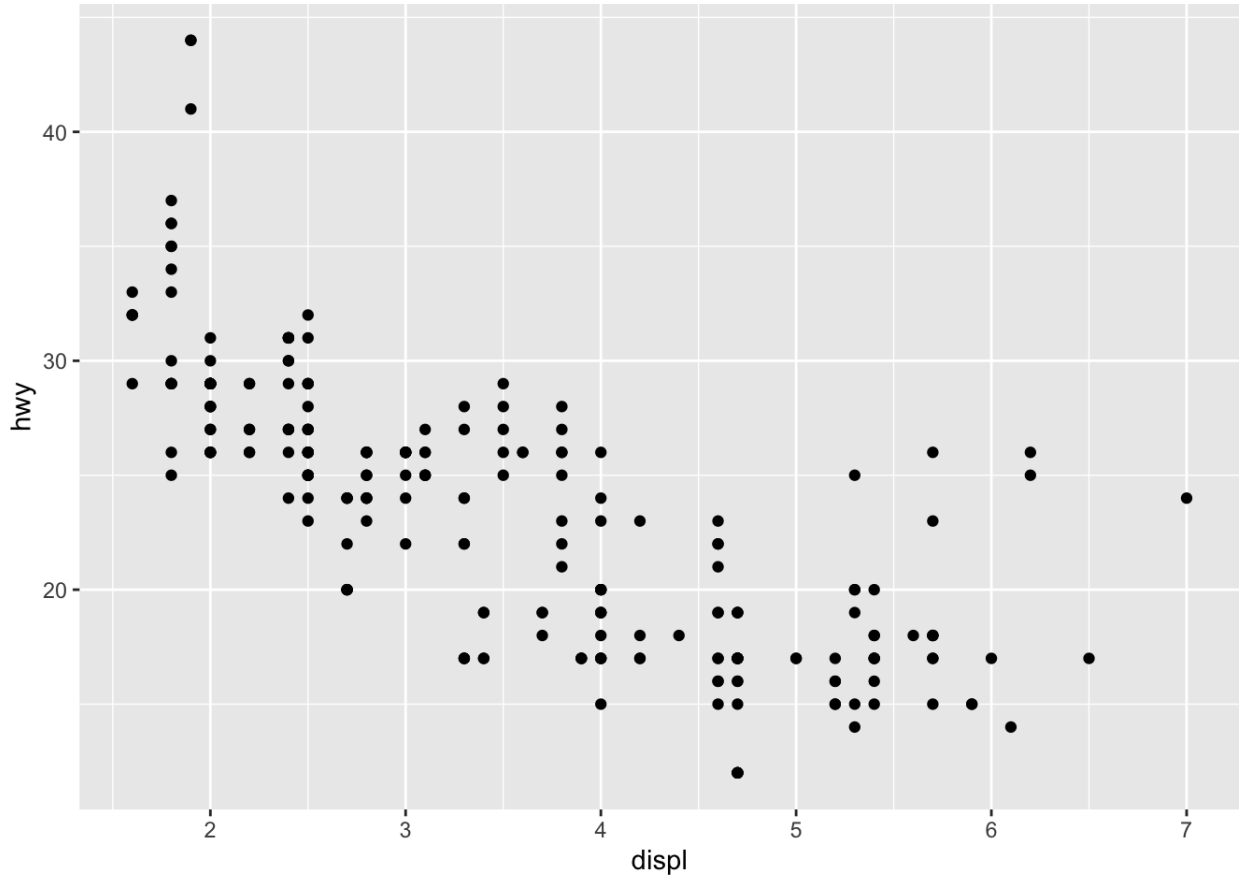
```
ggplot(mpg)+
  geom_violin(aes(class,hwy))
```



Q6. Visualize a cars engine size in liters (`displ`) versus fuel efficiency on the hwy (`hwy`) using a scatter plot.

Q6: Solution

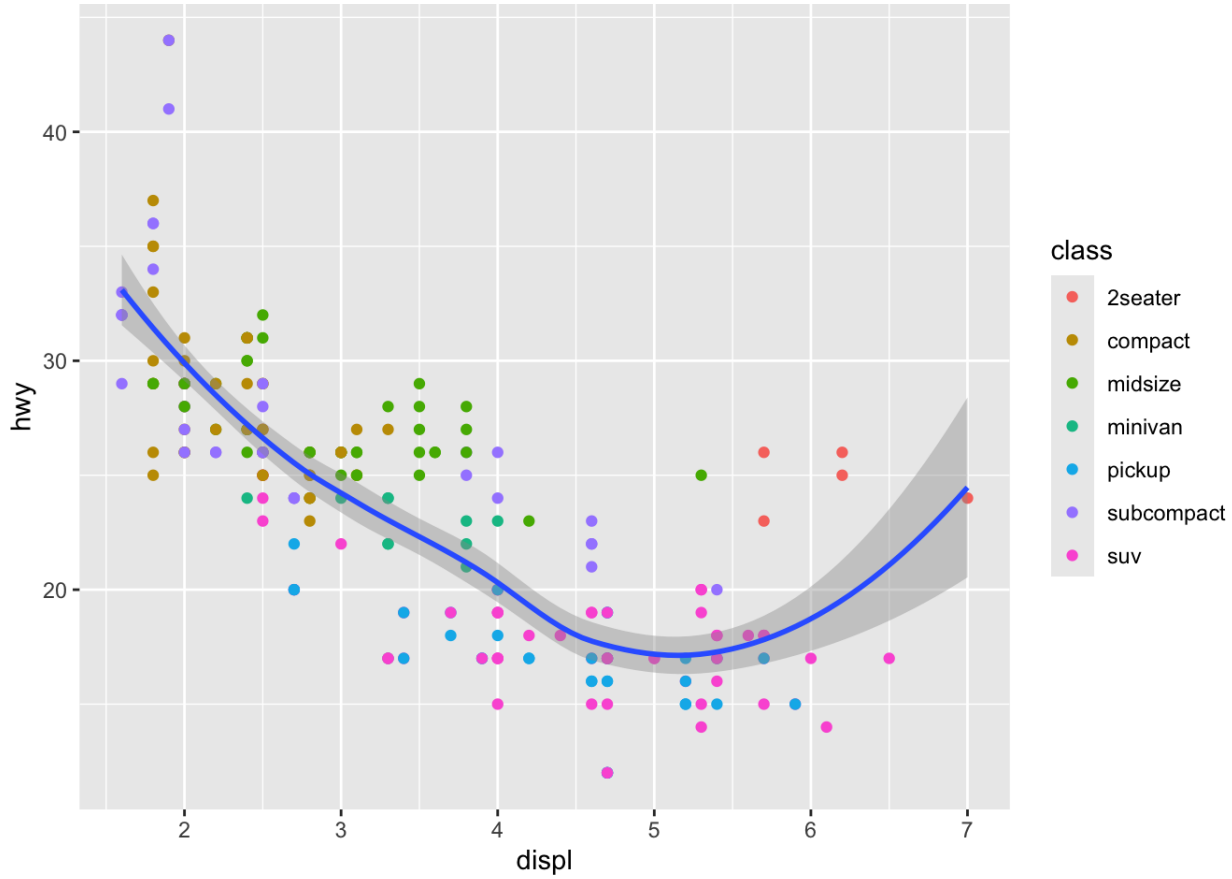
```
ggplot(mpg) +
  geom_point(aes(displ,hwy))
```



Q7. Using the plot generated in Q6, fit a smooth line (loess) to the data. Color the points by car `class`.

Q7: Solution

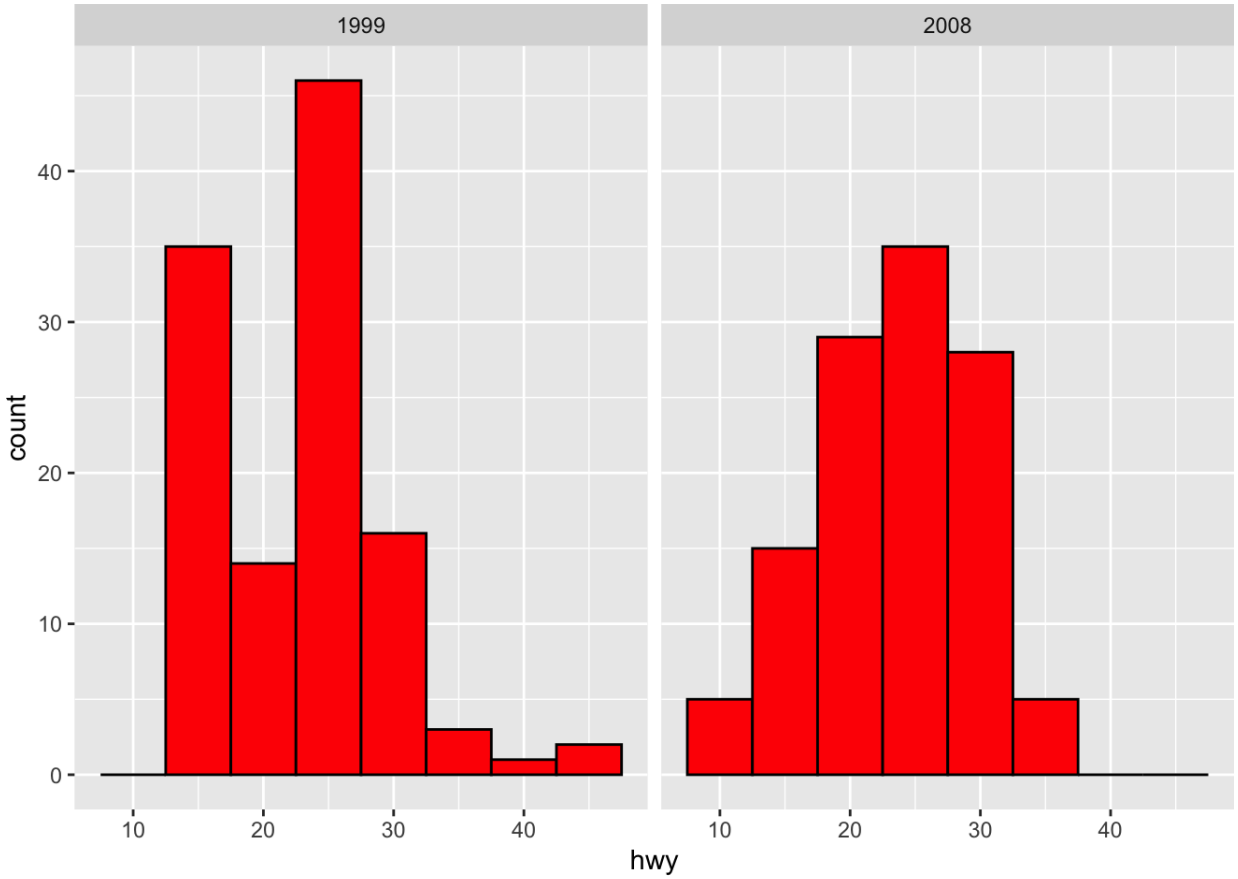
```
ggplot(mpg) +
  geom_point(aes(displ,hwy,color=class))+
  geom_smooth(aes(displ,hwy))
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



Q8. Visualize a histogram of `hwy` and facet by `year`. What is bindwidth (See `?geom_histogram`)? Explore the binwidth and color the bars red with a black outline.

Q8: Solution

```
ggplot(mpg)+
  geom_histogram(aes(hwy),fill="red",color="black", binwidth=5) +
  facet_wrap(~year)
```


1999
2008
count
0
10
20
30
40
10
20
30
40
hwy

Lesson 2 Exercise Questions: ggplot2 Plot Customization

This document contains practice questions on plot customization using **ggplot2**.

All questions use datasets available in base **R** or in **ggplot2**. Suggested workflow for students:

1. Attempt each question in your own script or console.
2. Only then consult the provided solution code.

> **Note**
>
> While one solution is provided per answer, multiple solutions are possible.

Start by loading `ggplot2`.

```
if (!requireNamespace("ggplot2", quietly = TRUE)) install.packages("g

library(ggplot2)
```

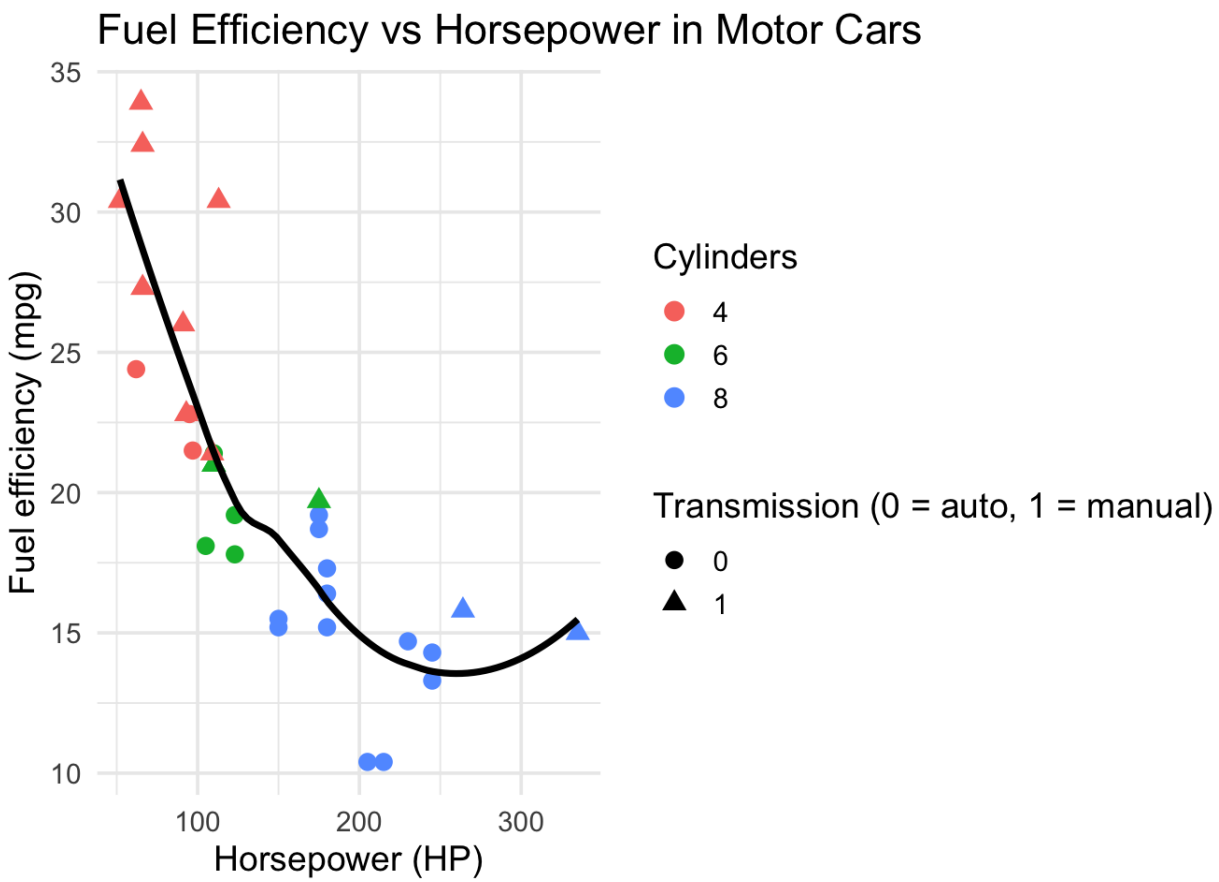
Q1. Using the `mtcars` dataset, create the following scatter plot:

- Set the x-axis to `hp` and the y-axis to `mpg`.
- Map `cyl` to **color** and `am` to **shape**.
- Increase point size and add a **smoothed line** (loess) in a different color, without a confidence band.
- Customize the following:
  - Make axes labels more informative (e.g., “Horsepower (HP)”).
  - Add a plot title (e.g., "Fuel Efficiency vs Horsepower in Motor Cars").
  - Set a minimal theme with a customized base font size = 14.

Q1: Solution

```
ggplot(mtcars, aes(x = hp, y = mpg)) +
  geom_point(aes(color = factor(cyl),
                 shape = factor(am)), size = 3) +
  geom_smooth(se = FALSE, method = "loess", color = "black") +
  labs(
    x = "Horsepower (HP)",
    y = "Fuel efficiency (mpg)",
    color = "Cylinders",
    shape = "Transmission (0 = auto, 1 = manual)",
    title = "Fuel Efficiency vs Horsepower in Motor Cars",
```

```
  ) +
  theme_minimal(base_size = 14)
```



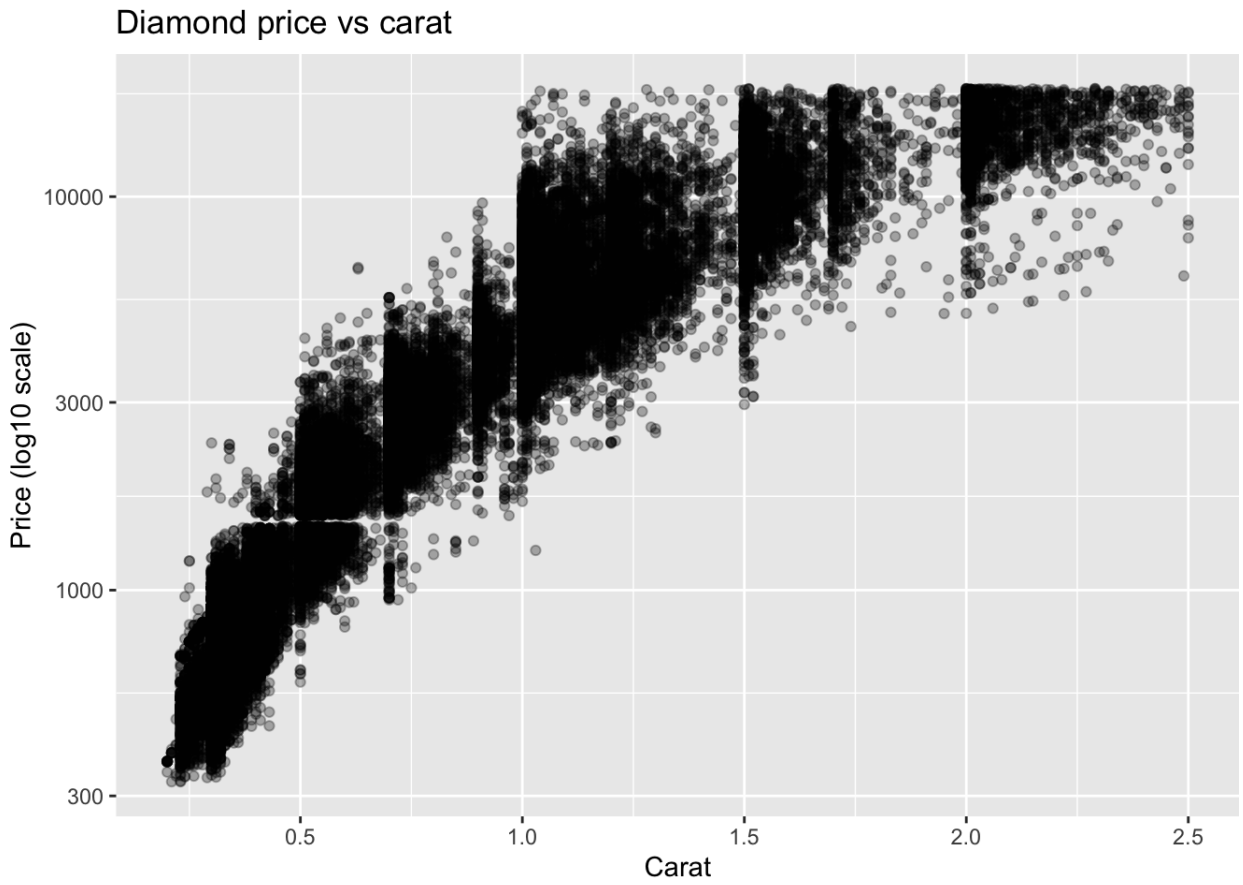
Q2: Using `diamonds` from `ggplot2`:

- Create a scatter plot of `carat` (x-axis) and `price` (y-axis); set the general transparency of the points to 0.3.
- Apply a **log10 transformation** to the y-axis, maintaining the original units on the axis (See `?scale_y_log10()`).
- Limit the x-axis to carats between **0.2 and 2.5**.
- Clean up the labels and add a title.

Q2: Solution

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(alpha = 0.3) +
  scale_y_log10() +
  scale_x_continuous(limits = c(0.2, 2.5))+
  labs(
    title = "Diamond price vs carat",
    x = "Carat",
```

```
    y = "Price (log10 scale)"
  )
```



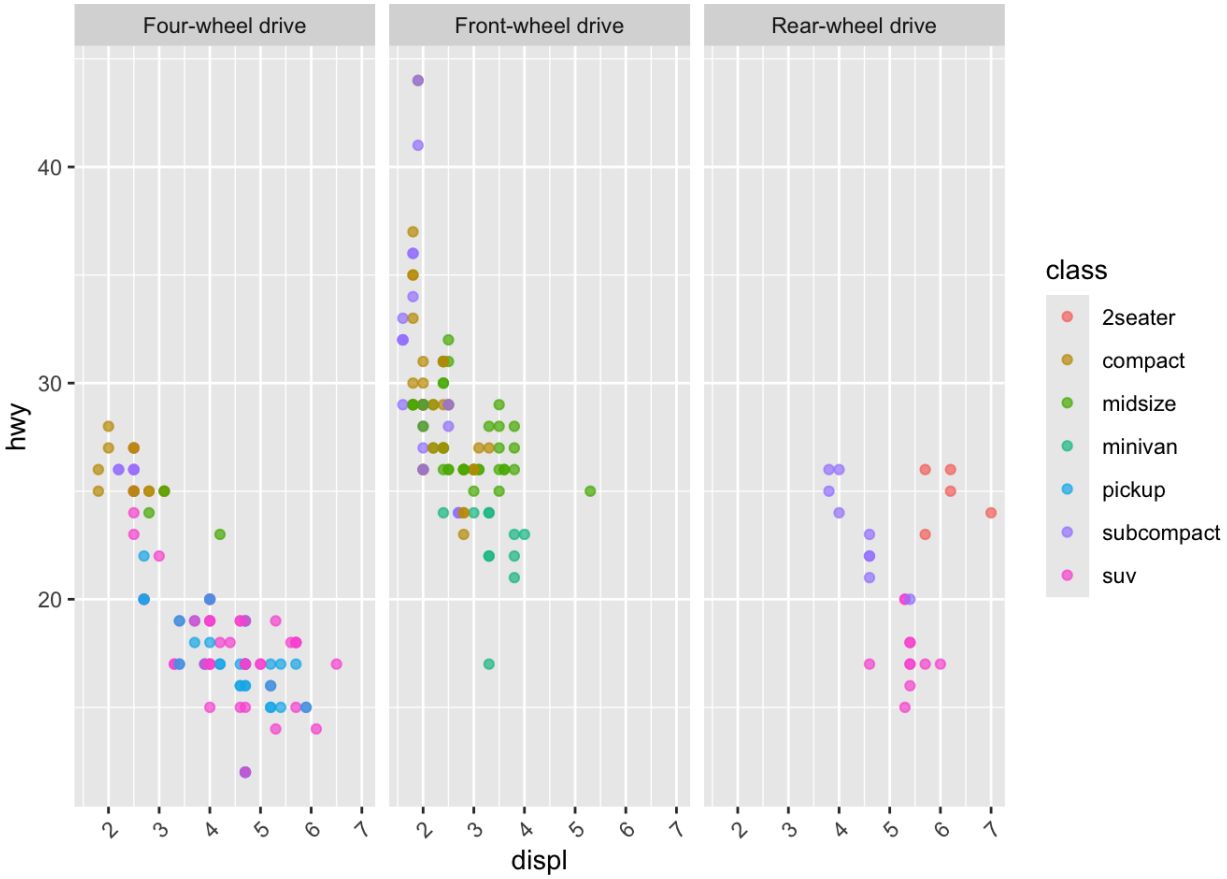
Q3: Using `ggplot2::mpg`, create a scatterplot of `displ` on the x-axis and `hwy` on the y-axis.

- Color points by `class`.
- Facet the plot by `drv` (front, rear, 4-wheel drive).
- Customize the facet labels using a `labeller` to replace `drv` values with more descriptive labels (e.g., "Four-wheel drive", "Front-wheel drive", and "Rear-wheel drive").
- Arrange the facets in a **single row**.
- Rotate the x-axis text by 45 degrees and right-align it.

Q3: Solution

```
drv_labels <- c(
  "4" = "Four-wheel drive",
  "f" = "Front-wheel drive",
  "r" = "Rear-wheel drive"
)

ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
  geom_point(alpha = 0.7) +
  facet_wrap(~ drv, nrow = 1,
```

```
            labeller = labeller(drv = drv_labels)) +
  theme(
    axis.text.x = element_text(angle = 45, hjust = 1)
  )
```



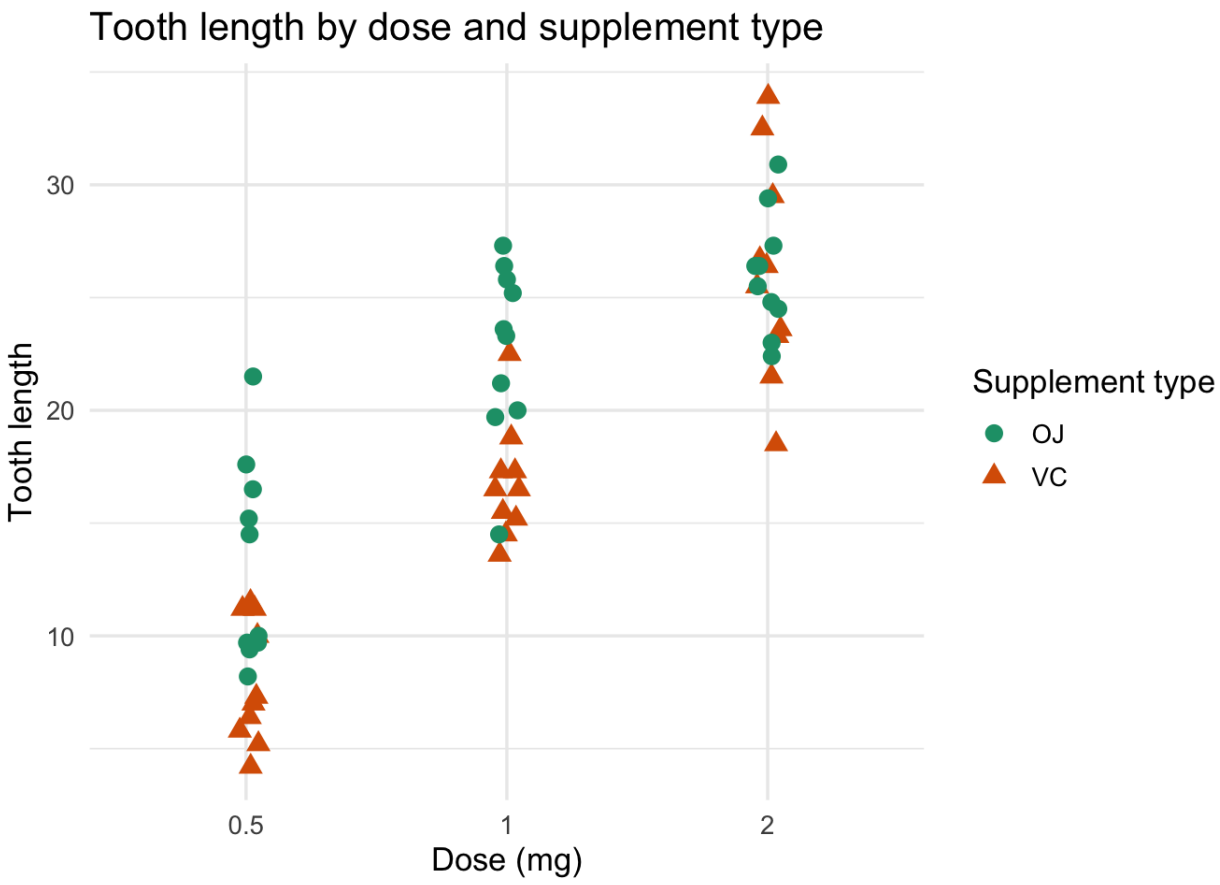
Q4. Using `ToothGrowth`:

- Convert `dose` to a factor.
- Plot `len` (tooth length) vs `dose`, with:
  - `dose` on the x-axis.
  - Points colored by `supp` and shape also mapped to `supp`.
- Customize the plot:
  - Use `scale_color_manual()` and `scale_shape_manual()` to assign specific colors and shapes to each supplement type.
  - Combine color and shape into a single **legend** with a custom title “Supplement type”.
  - Remove the legend background and legend key borders.

Q4: Solution

```
ToothGrowth$dose <- factor(ToothGrowth$dose)
```

```
ggplot(ToothGrowth,
       aes(x = dose, y = len,
           color = supp, shape = supp)) +
  geom_point(size = 3, position = position_jitter(width = 0.05, height = 0)) +
  scale_color_manual(
    name = "Supplement type",
    values = c("OJ" = "#1b9e77", "VC" = "#d95f02")
  ) +
  scale_shape_manual(
    name = "Supplement type",
    values = c("OJ" = 16, "VC" = 17)
  ) +
  labs(
    title = "Tooth length by dose and supplement type",
    x = "Dose (mg)",
    y = "Tooth length"
  ) +
  theme_minimal(base_size = 13) +
  theme(
    legend.background = element_blank(),
    legend.key = element_blank(),
    legend.position = "right"
  )
```

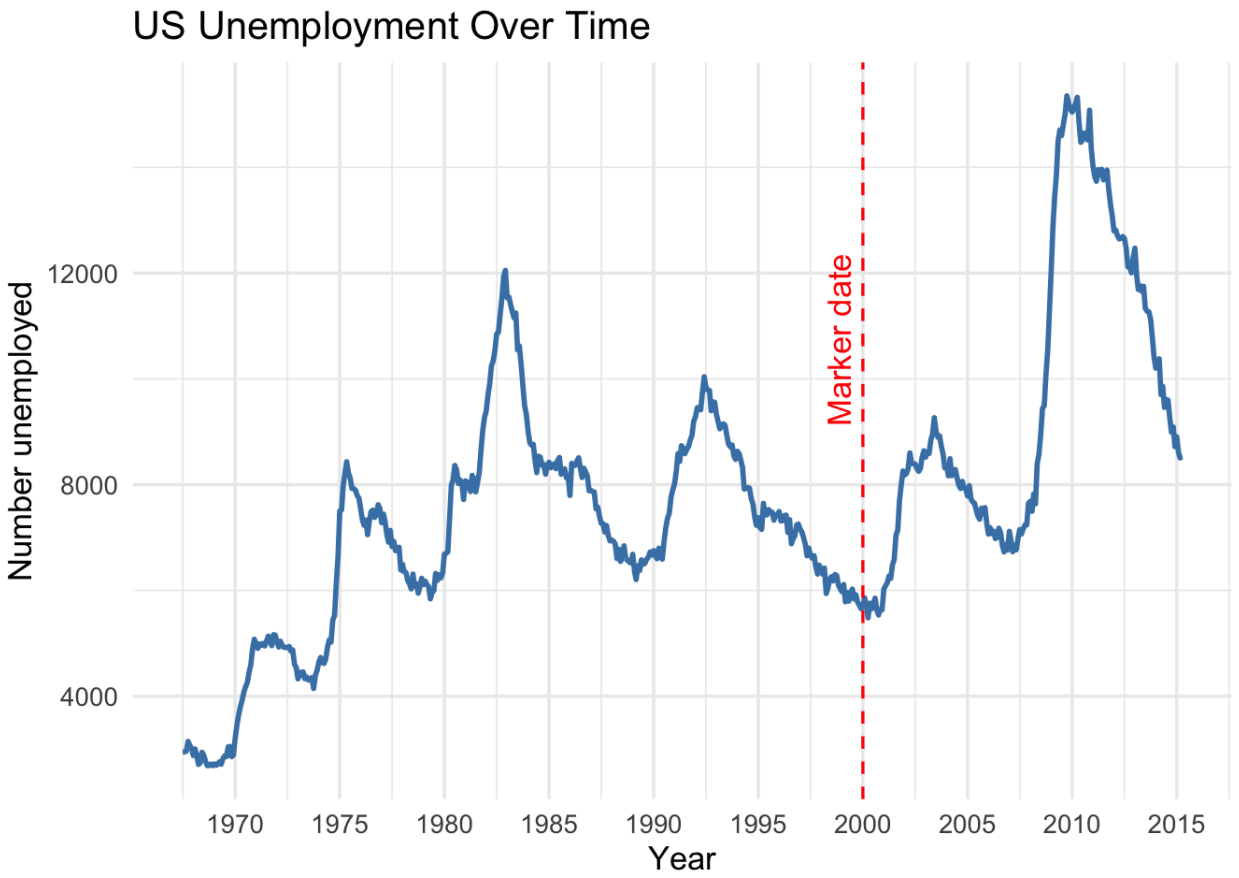


Q5: Using `ggplot2::economics`:

- Create a line plot of `date` on the x-axis and `unemploy` (number unemployed) on the y-axis.
- Add:
  - A **vertical line** at a chosen date (`lubrdiate::as_date("2000-01-01")`) using `geom_vline()`.
  - A **text annotation** near that line describing the event (See `annotate()`).
- Customize the plot:
  - Change the line color and size (within `geom_line()`).
  - Use a clean theme.
  - Adjust x-axis date breaks and labels (e.g., show ticks every 5 years).

Q5: Solution

```
library(lubridate)
marker_date <- lubridate::as_date("2000-01-01")

ggplot(economics, aes(x = date, y = unemploy)) +
  geom_line(color = "steelblue", linewidth = 1) +
  geom_vline(xintercept = marker_date,
             linetype = "dashed",
             color = "red") +
  annotate("text",
           x = marker_date,
           y = max(economics$unemploy) * 0.7,
           label = "Marker date",
           color = "red",
           angle = 90,
           vjust = -0.5) +
  scale_x_date(date_breaks = "5 years", date_labels = "%Y") +
  labs(
    title = "US Unemployment Over Time",
    x = "Year",
    y = "Number unemployed"
  ) +
  theme_minimal(base_size = 13)
```



Q6. Using any plot you created above (for example, the `mtcars` plot from Question 1):

- Save the plot object to a variable, e.g., `p`.
- Use `ggsave()` to:
  - Export the plot as a PNG file.
  - Specify a custom width and height in inches.
  - Set a suitable DPI (e.g., 300).

Q6: Solution

```
# Create the plot and assign to p
p <- ggplot(mtcars, aes(x = hp, y = mpg)) +
  geom_point(aes(color = factor(cyl),
                 shape = factor(am)), size = 3) +
  geom_smooth(se = FALSE, method = "loess", color = "black") +
  labs(
    x = "Horsepower (HP)",
    y = "Fuel efficiency (mpg)",
    color = "Cylinders",
    shape = "Transmission (0 = auto, 1 = manual)",
    title = "Fuel Efficiency vs Horsepower in Motor Cars",
  ) +
  theme_minimal(base_size = 14)
```

```
# Save as high-resolution PNG
ggsave(
  filename = "mtcars_mpg_hp.png",
  plot = p,
  width = 6,        # inches
  height = 4,       # inches
  dpi = 300
)
```

Lesson 3 Exercise Questions: Building a Publication Quality Plot

Putting what we have learned to the test:
The following questions synthesize several of the skills you have learned thus far. It may not be immediately apparent how you would go about answering these questions. Remember, the R community is expansive, and there are a number of ways to get help including but not limited to google search. These questions have multiple solutions, but you should try to stick to the tools you have learned to use thus far.

Your mission is to make a publishable figure.

We will use the `iris` data set for this.

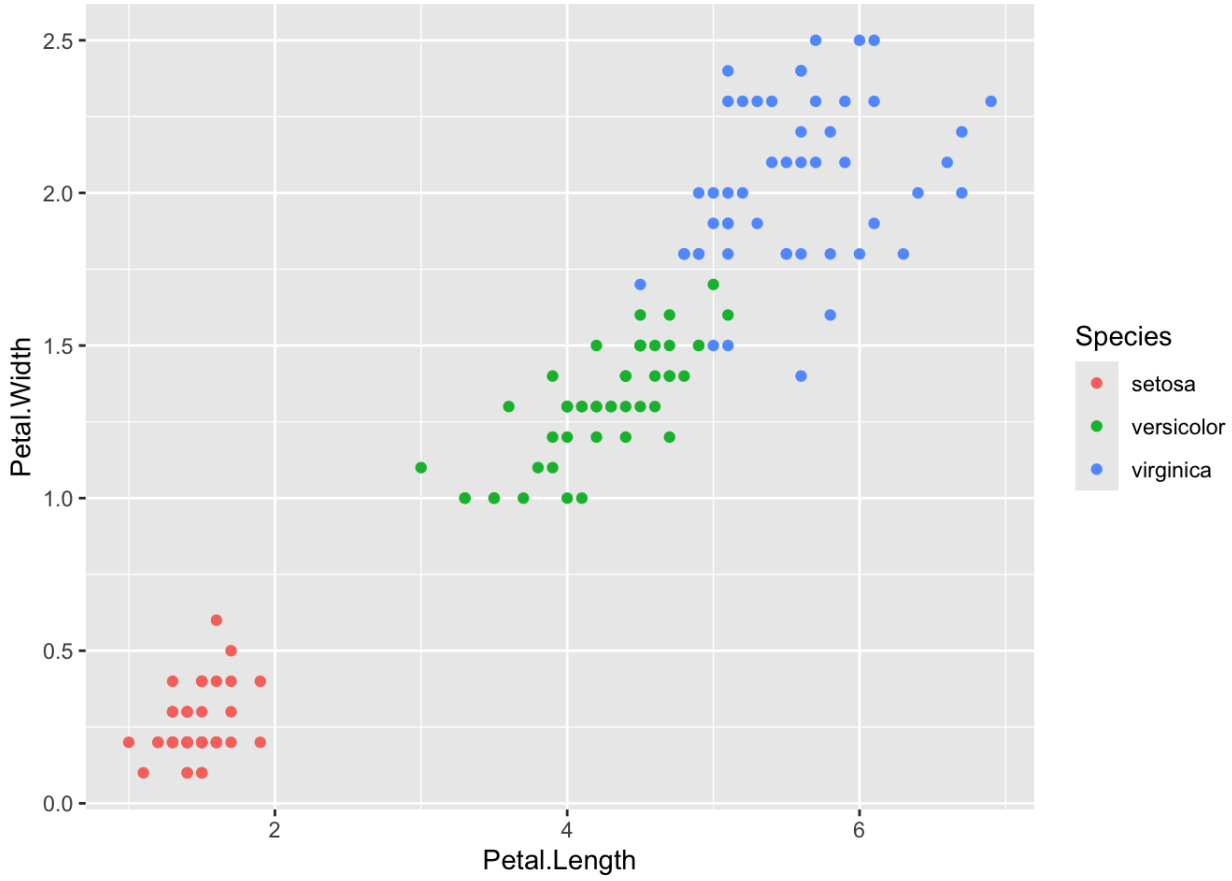
Start by loading `ggplot2`.

```
if (!requireNamespace("ggplot2", quietly = TRUE)) install.packages("g

library(ggplot2)
```

Q1. Start by creating a scatter plot of `iris` with `Petal.Length` on the x-axis and `Petal.Width` on the y-axis. Color the points by `Species`.

Q1: Solution

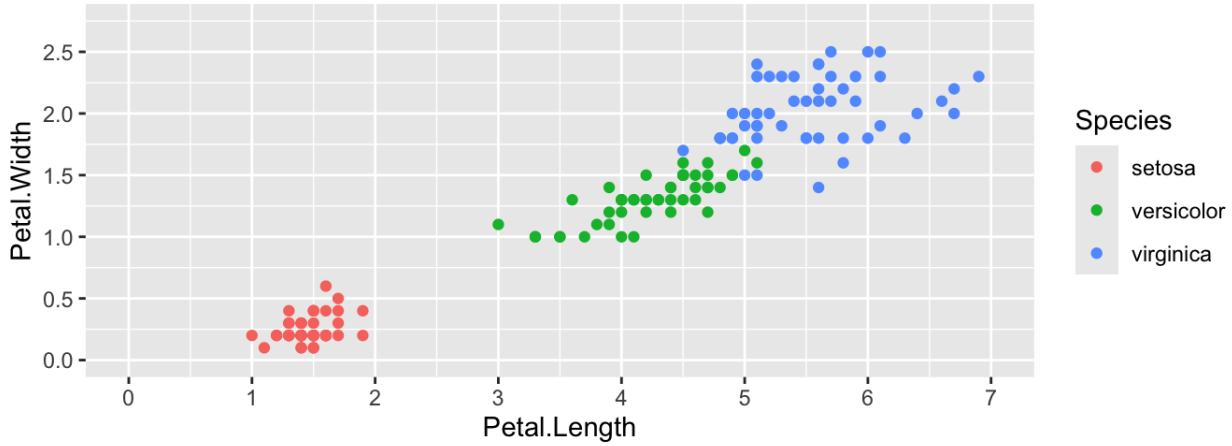
```
ggplot(iris)+
  geom_point(aes(Petal.Length,Petal.Width,color=Species))
```



Q2. Fix the axes so that the dimensions on the x-axis and the y-axis are equal (See ? `coord_fixed`). Both axes should start at 0. Label the axis breaks every 0.5 units on the y-axis and every 1.0 units on the x-axis.

Q2: Solution

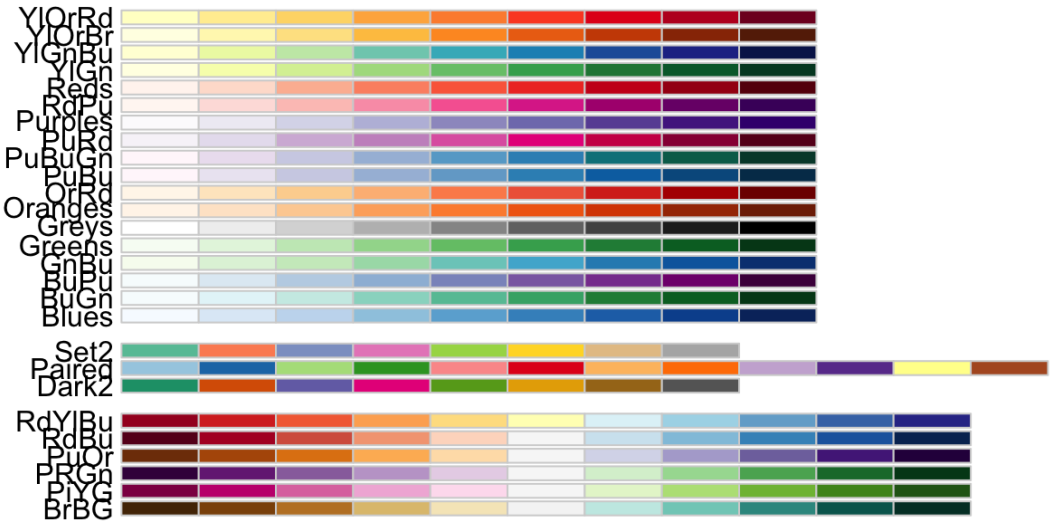
```
ggplot(iris)+
  geom_point(aes(Petal.Length,Petal.Width,color=Species))+
  coord_fixed(ratio=1,ylim=c(0,2.75),xlim=c(0,7)) +
  scale_y_continuous(breaks=seq(0,2.5, by=0.5)) +
  scale_x_continuous(breaks=0:7)
```



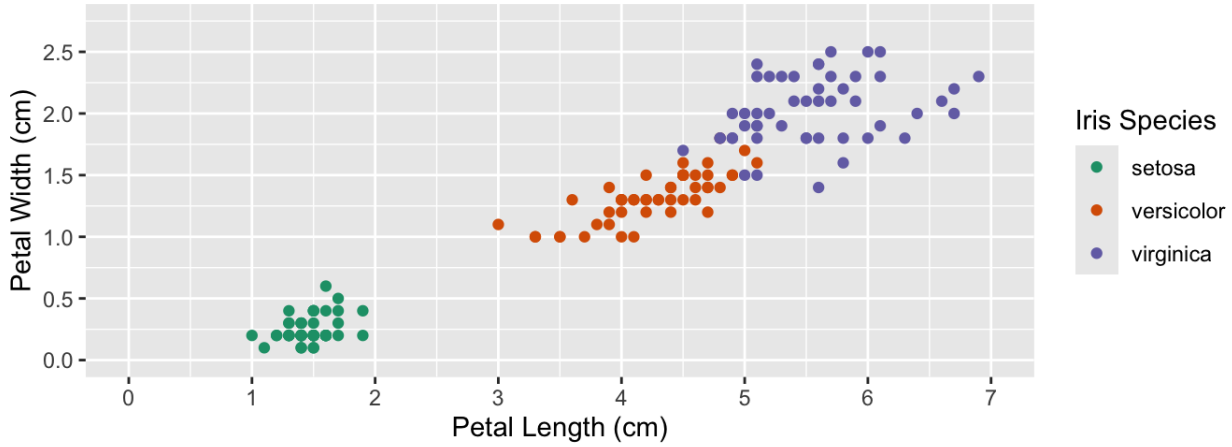
Q3. Assign a color blind friendly palette to the color of the points, and change the legend title to "Iris Species". Label the x and y axes to make the variable names visually appealing; include unit information.

Q3: Solution

```
#multiple ways to find color blind friendly palettes.
#using color brewer scales
RColorBrewer::display.brewer.all(colorblindFriendly=TRUE)
```



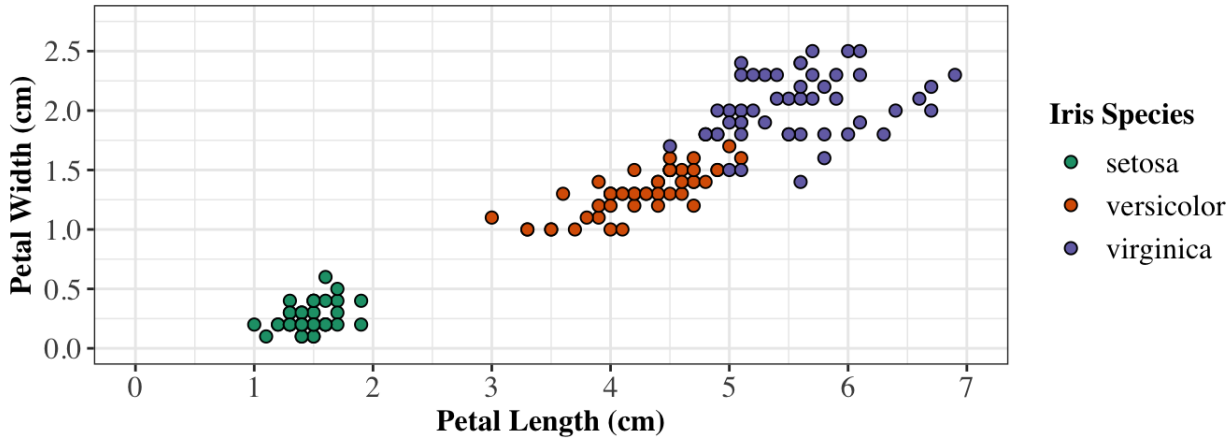

```
ggplot(iris)+
  geom_point(aes(Petal.Length,Petal.Width,color=Species))+
  coord_fixed(ratio=1,ylim=c(0,2.75),xlim=c(0,7)) +
  scale_y_continuous(breaks=seq(0,2.5, by=0.5)) +
  scale_x_continuous(breaks=0:7) +
  scale_color_brewer(palette = "Dark2",name="Iris Species") +
  labs(x="Petal Length (cm)", y= "Petal Width (cm)")
```



Q4. Play with the theme to make your plot nicer and more publishable. Change font style to "Times". Change all font sizes to 12 pt font. Bold the legend title and the axes titles. Increase the size of the points on the plot to 2. **Bonus: fill the points with color and have a black outline around each point.**

Q4: Solution

```
ggplot(iris)+
  geom_point(aes(Petal.Length,Petal.Width,fill=Species),size=2,shape=21)+
  coord_fixed(ratio=1,ylim=c(0,2.75),xlim=c(0,7)) +
  scale_y_continuous(breaks=seq(0,2.5, by=0.5)) +
  scale_x_continuous(breaks=0:7) +
  scale_fill_brewer(palette = "Dark2",name="Iris Species") +
  labs(x="Petal Length (cm)", y= "Petal Width (cm)") +
  theme_bw()+
  theme(axis.text=element_text(family="Times",size=12),
        axis.title=element_text(family="Times",face="bold",size=12),
        legend.text=element_text(family="Times",size=12),
        legend.title = (element_text(family="Times",face="bold",size=12))
        )
```



Q5. Save your plot using `ggsave`.

Q5: Solution

```
ggsave("iris.tiff", width=5.5, height=3.5,units="in")
```

Lesson 4 Exercise Questions: ggplot2

This exercise questions are meant to test your learning following Lesson 4: Recommendations and Tips for Creating Effective Plots with ggplot2. To approach these questions, you will need to understand how to find help.

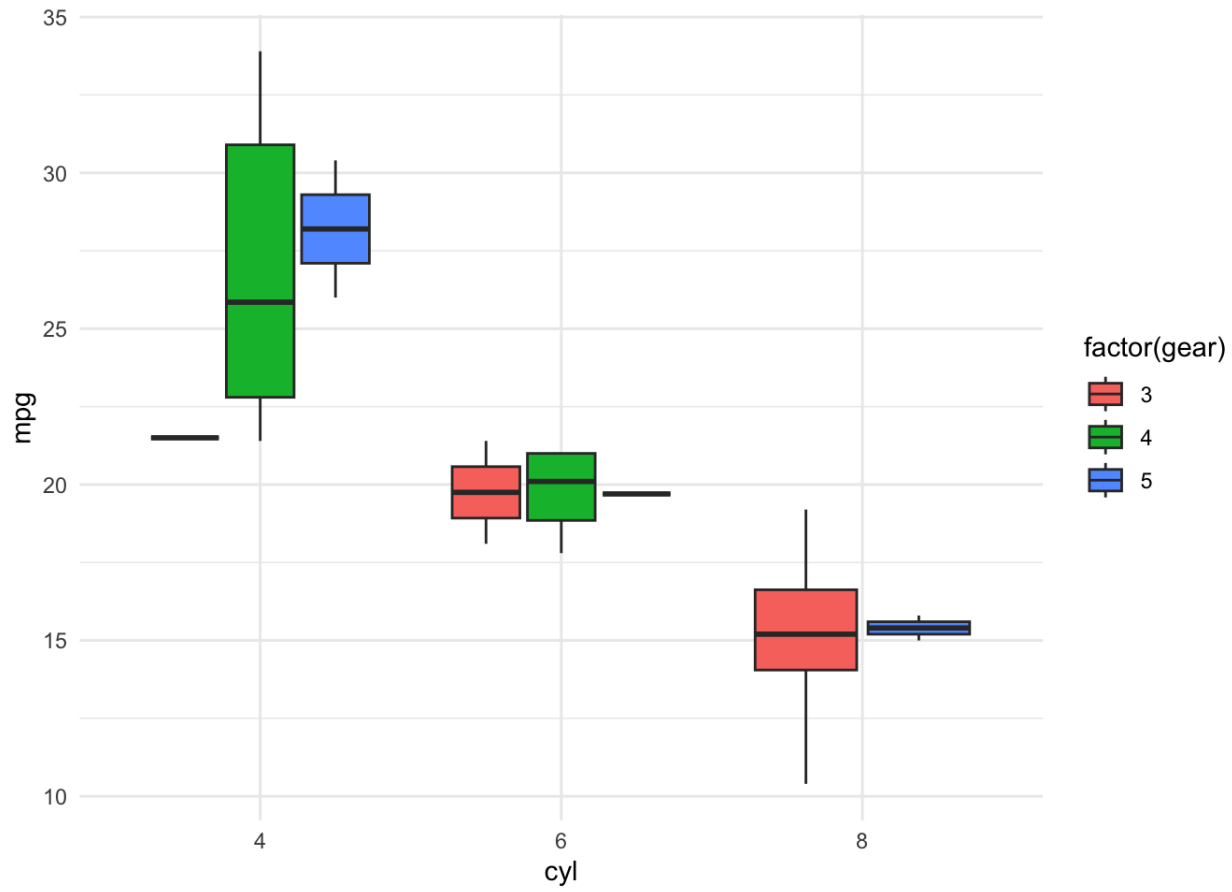
Start by loading `ggplot2` and `patchwork`.

```
if (!requireNamespace("ggplot2", quietly = TRUE)) install.packages("g
if (!requireNamespace("patchwork", quietly = TRUE)) install.packages

library(ggplot2)
library(patchwork)
```

Q1. Write a function `plot_mpg_by_cyl(df)` that plots a box plot of `mpg` vs `cyl` colored by `gear`. Test on `mtcars`.

Q1: Solution

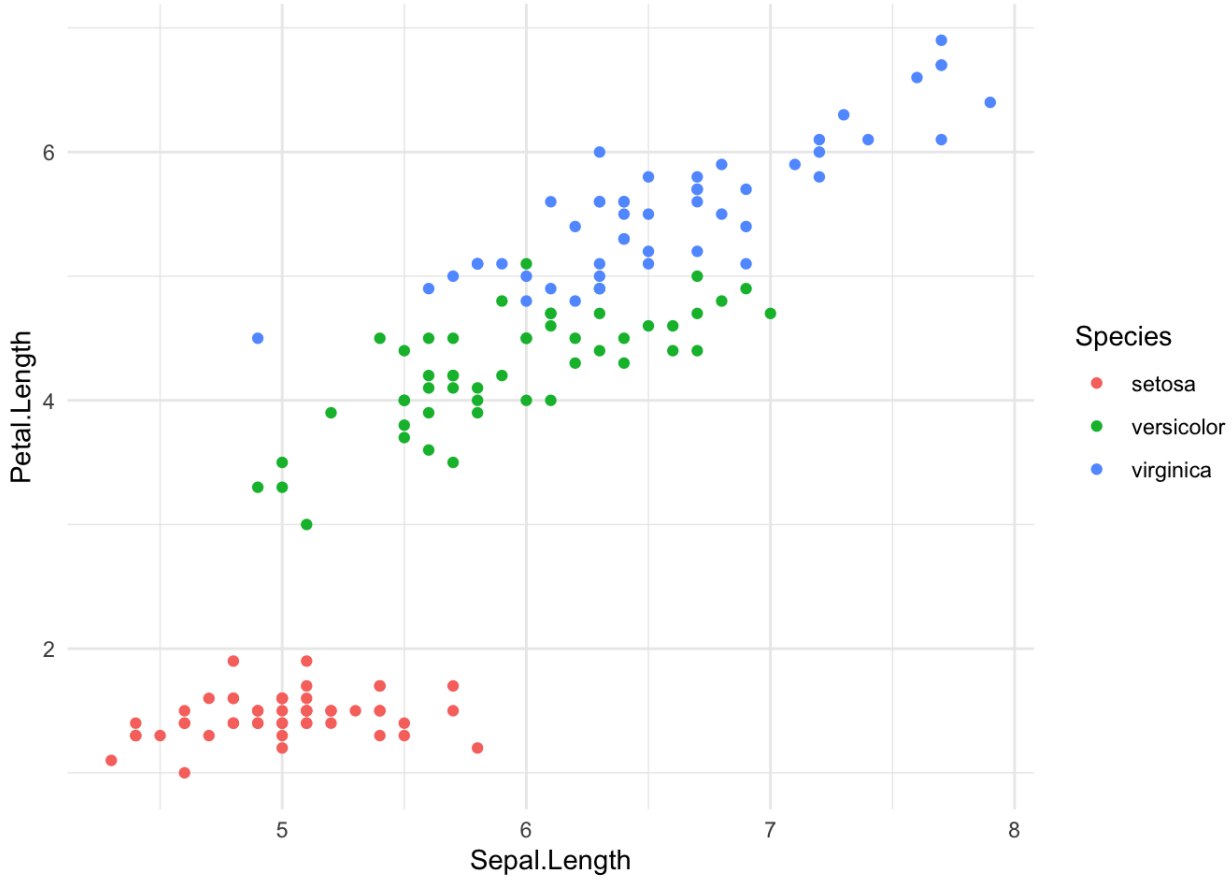
```
plot_mpg_by_cyl <- function(df) {
  ggplot(df) +
    geom_boxplot(aes(x = factor(cyl), y = mpg, fill = factor(gear))) +
    labs(x = "cyl", y = "mpg", color = "gear") +
    theme_minimal()
}

plot_mpg_by_cyl(mtcars)
```



Q2. Write `scatter_plot(df, x, y, color)` that creates a scatter plot using column names for `x`, `y`, and `color`. Test on `iris` with `Sepal.Length`, `Petal.Length`, `Species`.

Q2: Solution

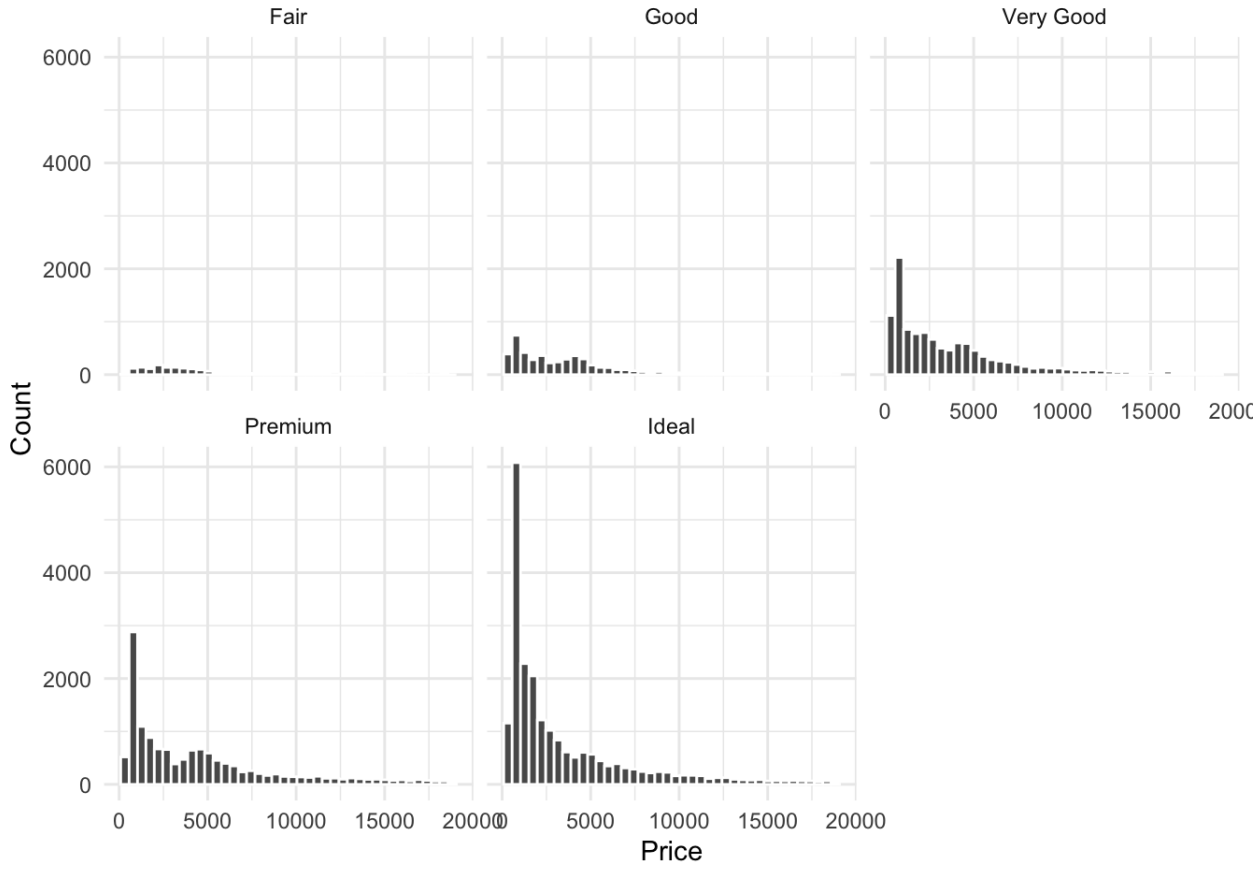
```
scatter_plot <- function(df, x, y, color) {
  ggplot(df) +
    geom_point(aes(x = {{x}}, y = {{y}}, color = {{color}})) +
    theme_minimal()
}

scatter_plot(iris, Sepal.Length, Petal.Length, Species)
```



Q3. Write `facet_histogram(df, variable, facet_by, bins = 30)` that draws a histogram of a numeric `variable` and `facet_wraps` by `facet_by`. The function should also accept a `bins` argument. Test on `diamonds` with `price` by `cut`.

Q3: Solution

```
facet_histogram <- function(df, variable, facet_by, bins = 30) {
  ggplot(df) +
    geom_histogram(aes(x = {{variable}}), bins = bins,
    color = "white") +
    facet_wrap(vars({{facet_by}})) +
    labs(x= stringr::str_to_sentence(rlang::englue("{{variable}}")),
    y = "Count") +
    theme_minimal()
}

facet_histogram(ggplot2::diamonds, price, cut, bins = 40)
```



Q4. Challenge Question: Define a function `plot_relationship(df, x, y, method = "point")` that:

- uses `geom_point()` when method = "point",
- uses `geom_point()` and `geom_smooth()` when method = "smooth",
- includes a custom `theme_minimal()` and title using `rlang::englue()`.

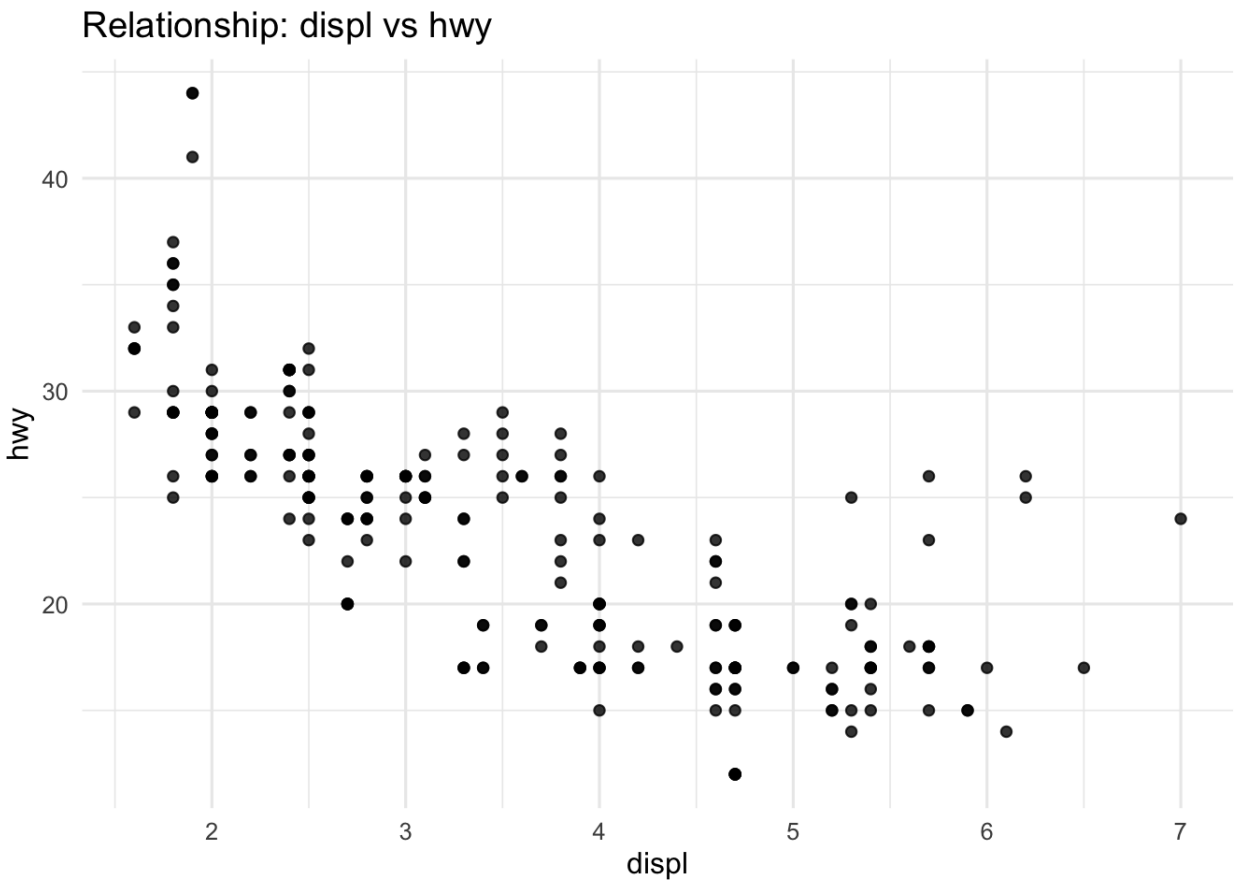
Test on `mpg` with both methods to compare relationships between `displ` and `hwy`.

Q4: Solution

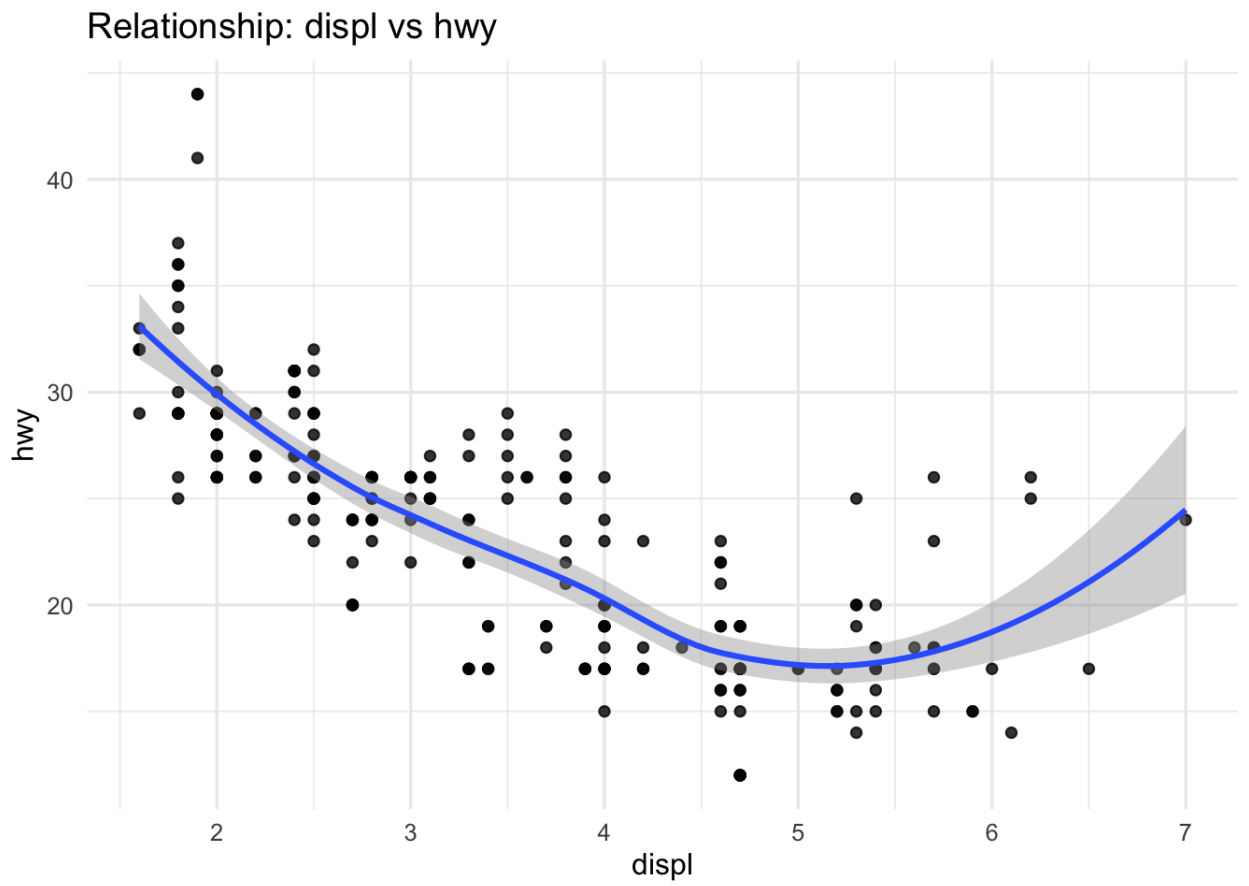
```
plot_relationship <- function(df, x, y, method = "point") {
  label <- rlang::englue("Relationship: {{x}} vs {{y}}")
  base <- ggplot(df, aes(x = {{x}}, y = {{y}})) +
    geom_point(alpha = 0.8) +
    labs(title = label) +
    theme_minimal(base_size = 12)

  if (method == "point") {
    base
  } else if (method == "smooth") {
    base + geom_smooth(se = TRUE)
  } else {
```

```
    stop("method must be 'point' or 'smooth'")
  }
}

# Examples using mpg
p4_point  <- plot_relationship(mpg, displ, hwy, method = "point")
p4_smooth <- plot_relationship(mpg, displ, hwy, method = "smooth")
```

Method = "point":



Method = "smooth":



Q5. Using `mpg`, build two scatter plots:

- `p1` - plot `displ` on the x-axis and `hwy` on the y-axis
- `p2` - plot `cty` on the x-axis and `hwy` on the y-axis

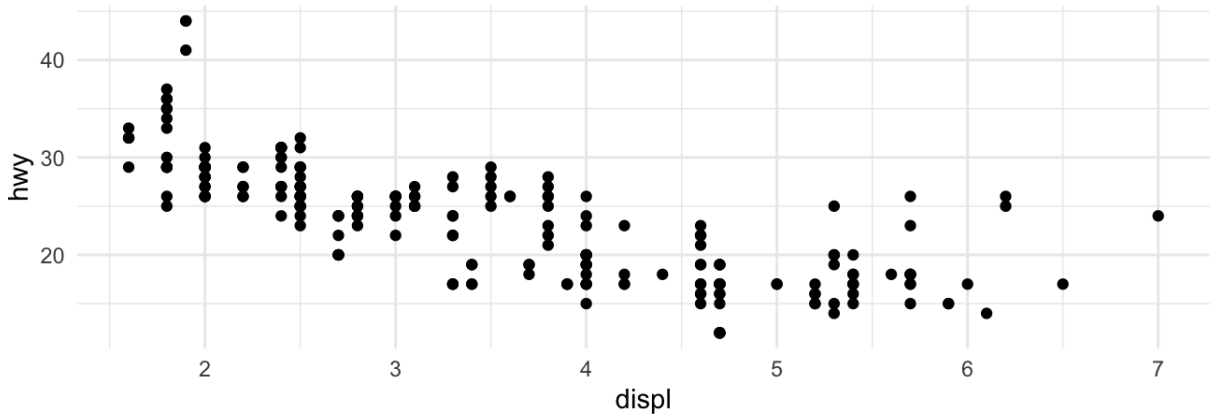
Include the complete theme, `theme_minimal()`. Stack the plots vertically and horizontally using `patchwork`.

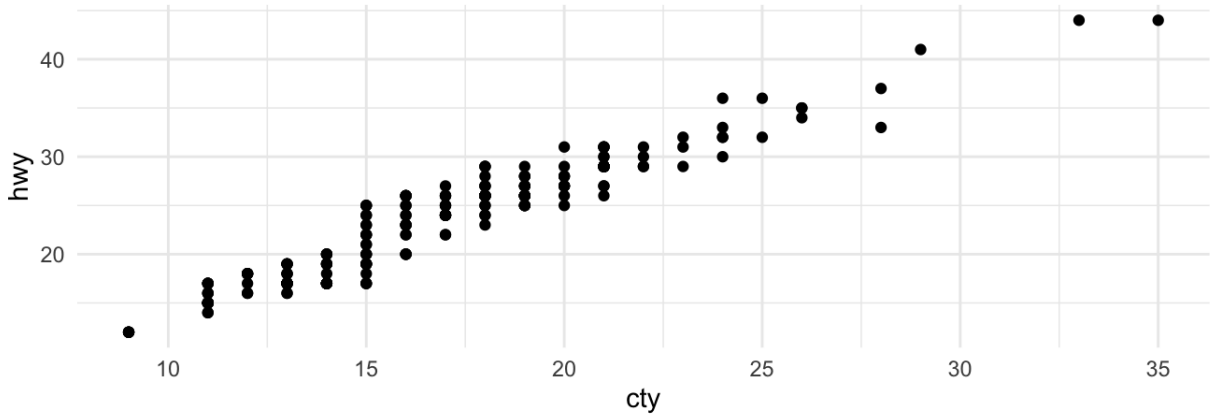
Q5: Solution

```
p1 <- ggplot(mpg) +
  geom_point(aes(displ, hwy)) + theme_minimal()

p2 <- ggplot(mpg) +
  geom_point(aes(cty, hwy)) + theme_minimal()

#vertically
p1 / p2
```

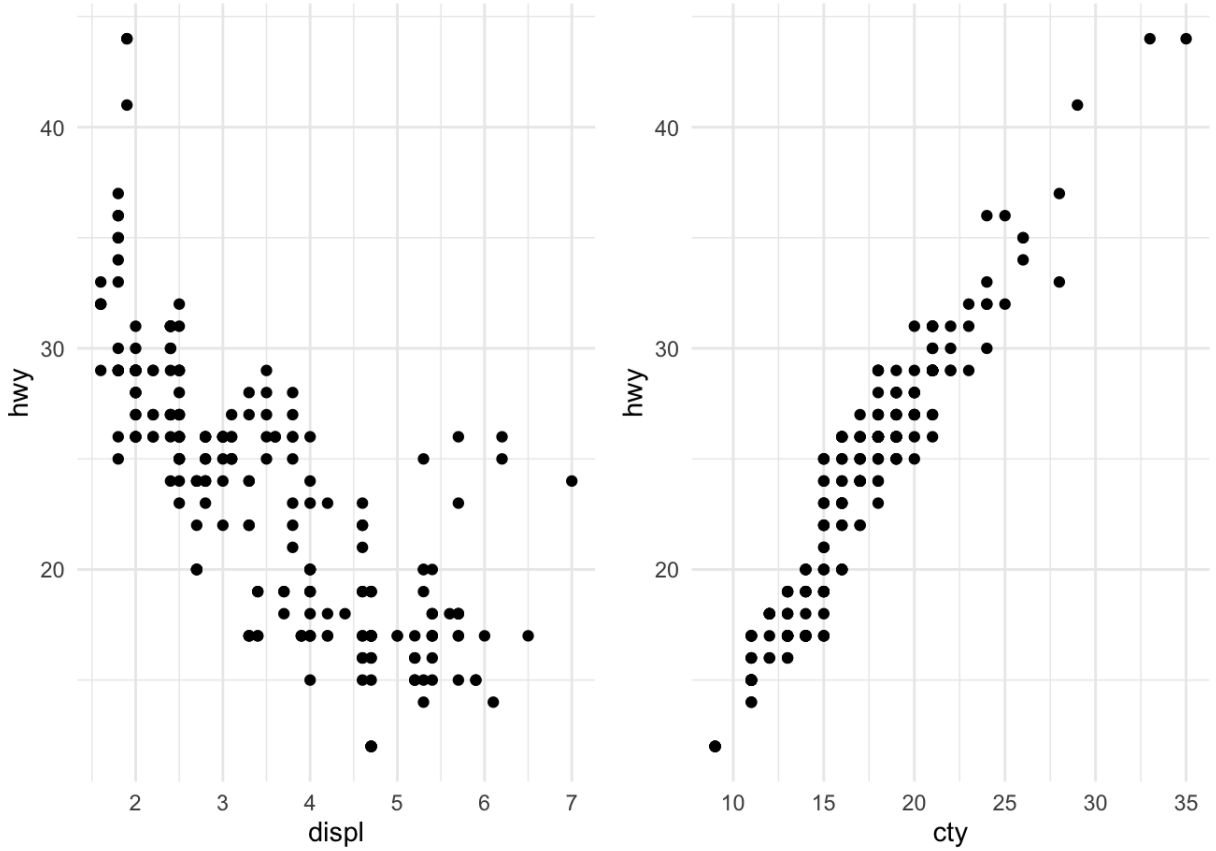
```
#horizontally
p1 | p2
```

Vertical:





Horizontal:



Q6. With the built-in data set `economics`, make 3 line plots using `geom_line()` and `theme_minimal()`:

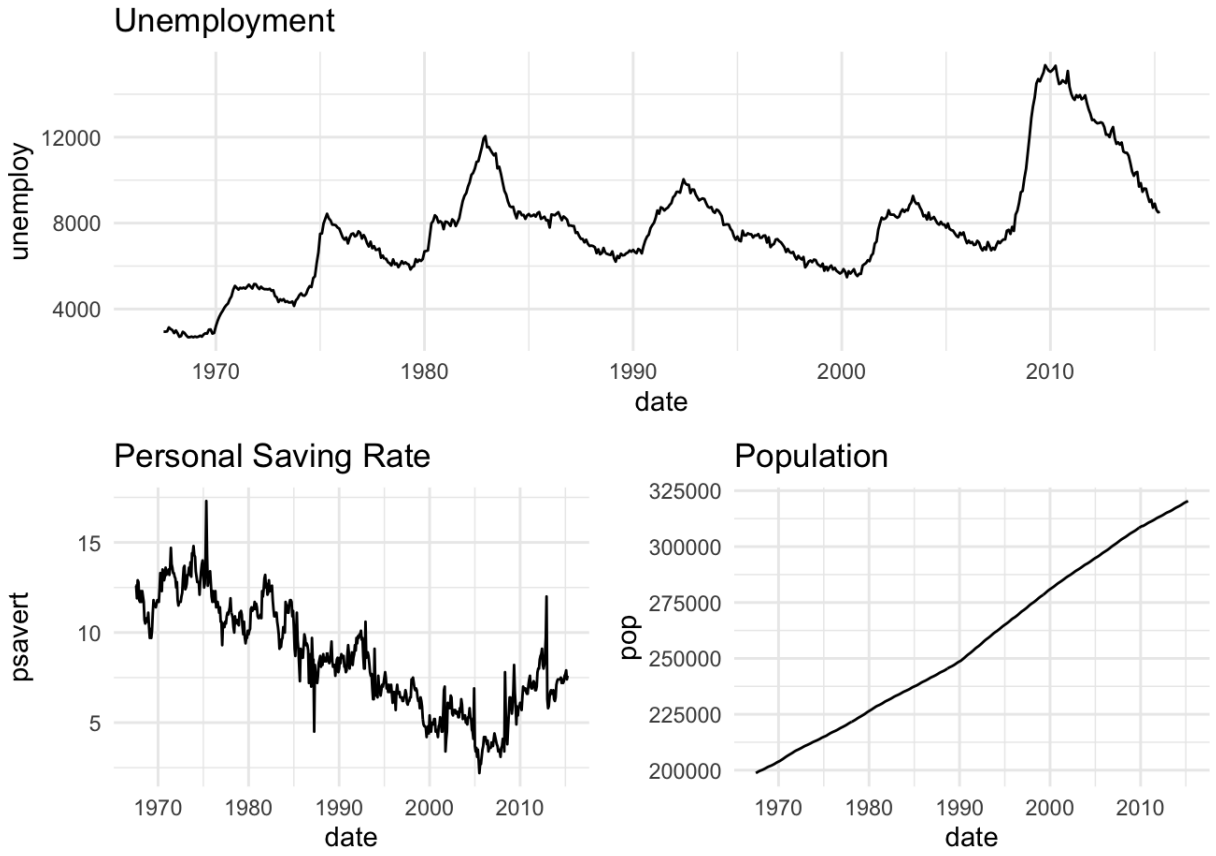
- `p1` - plot `date` on the x-axis and `unemploy` on the y-axis. Include a plot title (`title = "Unemployment"`).
- `p2` - plot `date` on the x-axis and `psavert` on the y-axis. Include a plot title (`title = "Personal Saving Rate"`).
- `p3` - plot `date` on the x-axis and `pop` on the y-axis. Include a plot title (`title = "Population"`).

Arrange the plots using patchwork. `p1` should be on the top (row 1), and `p2` and `p3` should be oriented horizontally on the bottom (row 2).

Q6: Solution

```
p1 <- ggplot(economics) +
  geom_line(aes(date, unemploy)) + labs(title = "Unemployment") + theme_minimal()

p2 <- ggplot(economics) +
  geom_line(aes(date, psavert)) + labs(title = "Personal Saving Rate") + theme_mini

p3 <- ggplot(economics) +
  geom_line(aes(date, pop)) + labs(title = "Population") + theme_minimal()
```

```
(p1 / (p2 | p3))
```



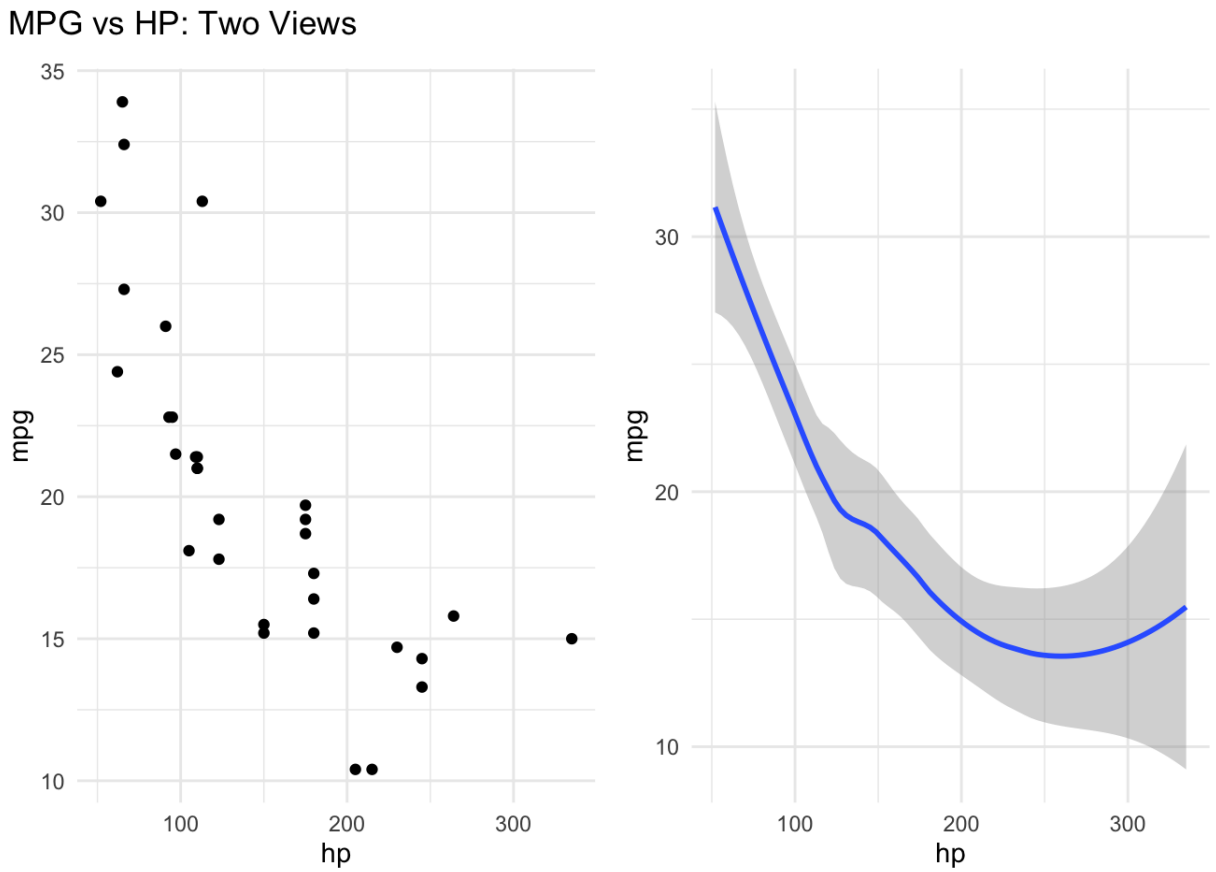
Q7. Using `mtcars`, create two plots:

- `a` - a scatter plot with `hp` on the x-axis and `mpg` on the y-axis
- `b` - a smooth plot with `hp` on the x-axis and `mpg` on the y-axis.

Combine the plots in a horizontal orientation (1 row, 2 columns) using patchwork. Use `plot_annotation()` to include a shared title.

Q7: Solution

```
a <- ggplot(mtcars) +
  geom_point(aes(hp, mpg)) + theme_minimal()

b <- ggplot(mtcars) +
  geom_smooth(aes(hp, mpg),se = TRUE) + theme_minimal()

(a | b) + plot_annotation(title = "MPG vs HP: Two Views")
```



Q8. Write a function, `compare_two_vars(df, x, y, group)` that makes two plots combined with `patchwork`.

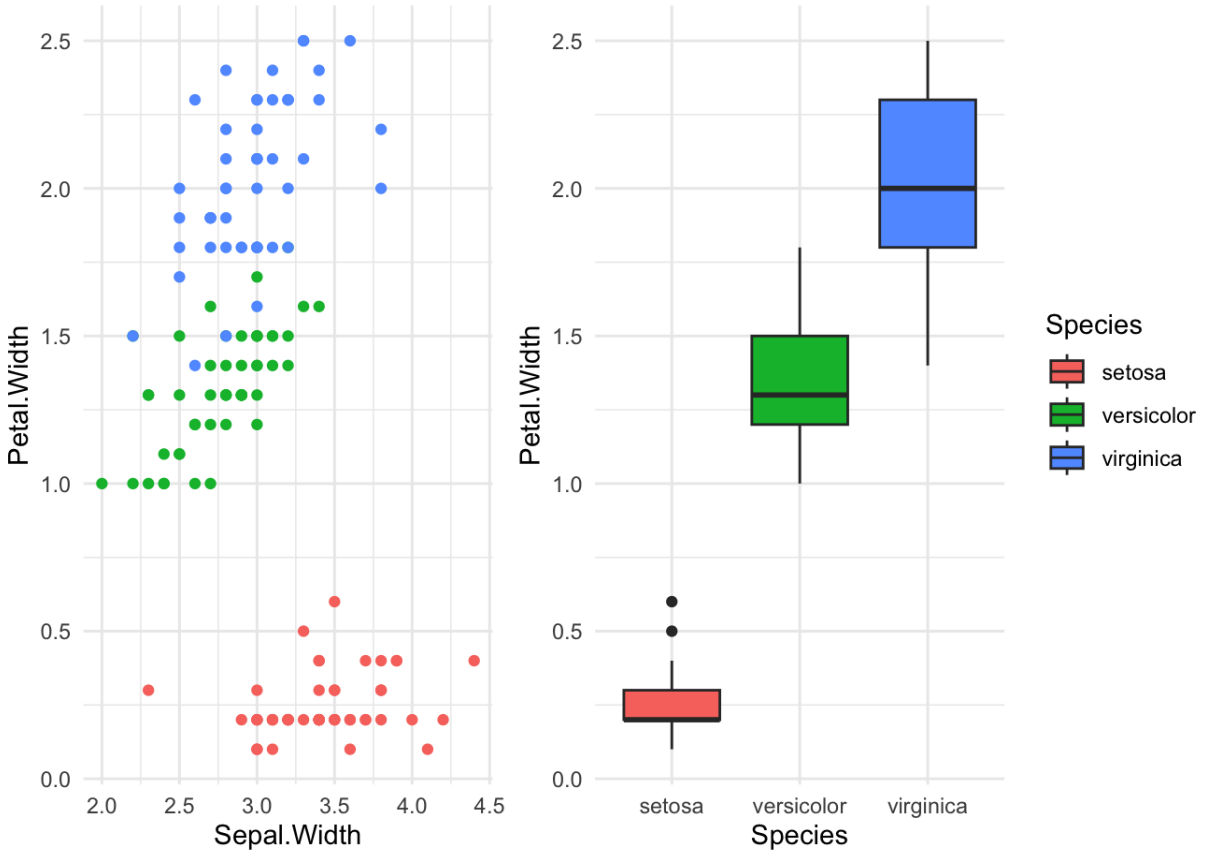
- `p1` - create a scatter plot taking two arguments, `x` and `y`, and color set to `group`,
- `p2` makes a boxplot of `y` by `group`, - combines them side-by-side with **patchwork**.

Test on `iris` with `Sepal.Width` (x), `Petal.Width` (y), grouped by `Species`.

Q8: Solution

```
compare_two_vars <- function(df, x, y, group) {
  p1 <- ggplot(df) +
    geom_point(aes(x = {{x}}, y = {{y}}, color = {{group}})) +
    theme_minimal() +
    theme(legend.position = "none")

  p2 <- ggplot(df) +
    geom_boxplot(aes(x = {{group}}, y = {{y}}, fill = {{group}})) +
    theme_minimal()


  p1 | p2
}
```

```
compare_two_vars(iris, Sepal.Width, Petal.Width, Species)
```



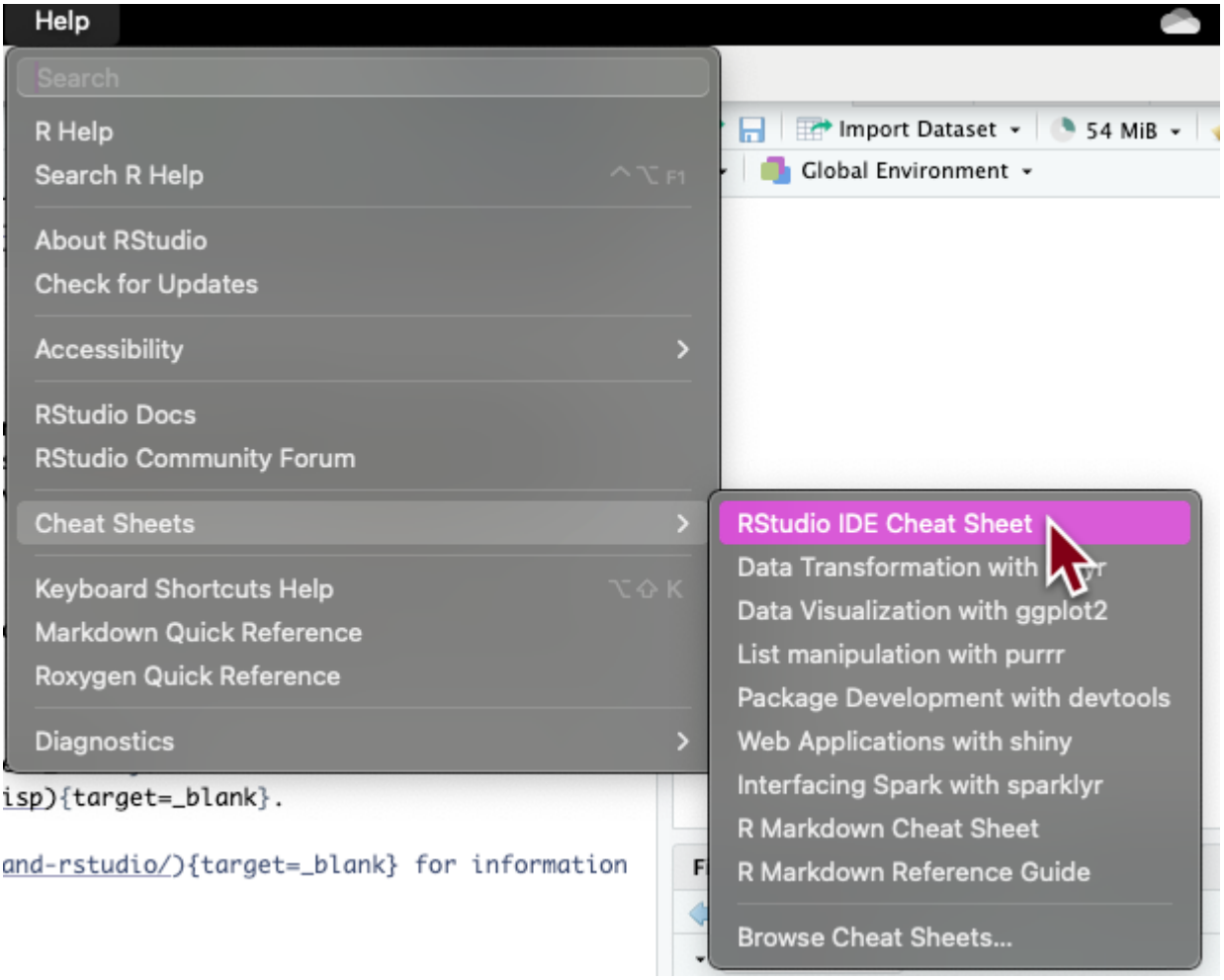
Additional Resources

Books and / or Book Chapters of Interest

1. R for Data Science *(https://r4ds.hadley.nz/)*
2. Hands-on Programming with R *(https://rstudio-education.github.io/hopr/)*
3. Statistical Inference via Data Science: A ModernDive into R and the Tidyverse *(https://moderndive.com/v2/preface.html#about-the-book)*
4. The R Graphics Cookbook *(https://r-graphics.org/index.html)*
5. ggplot2: Elegant Graphics for Data Analysis *(https://ggplot2-book.org/index.html)*
6. Advanced R *(https://adv-r.hadley.nz/)*
7. YaRrr! The Pirate's Guide to R *(https://bookdown.org/ndphillips/YaRrr/)*

R Cheat Sheets

Cheat sheets can be accessed directly using the Help tab within RStudio (Help > Cheat Sheets > Browse Cheat Sheets).



Other Resources

1. The R Graph Gallery *(https://www.r-graph-gallery.com/)*
2. From Data to Viz *(https://www.data-to-viz.com/)*
3. RMarkdown from RStudio *(https://rmarkdown.rstudio.com/lesson-1.html)*
4. Quarto for R *(https://quarto.org/docs/computations/r.html)*
5. Ten simple rules for teaching yourself R, Lawlor et al. 2022, *PLoS Comput Biol (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9436135/)*
6. Learn R *(https://www.learn-r.org/)*
7. Dplyr Learn R tutorial *(https://allisonhorst.shinyapps.io/dplyr-learnr/#section-welcome)*