

R Introductory Series



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

Table of Contents

Course Overview

● Course Overview	10
● Welcome to the R Introductory Series!	10
● A series of introductory lessons in R for scientists.	10
● Course Expectations	10
● Content Organization	10
● Introduction to R and RStudio	10
● The Basics of R Programming	10
● R Data Structures: Introducing Data Frames	11
● Data Frames and Data Wrangling (part 1)	11
● Data Frames and Data Wrangling (part 2)	11
● Introduction to Data Visualization with R (part 1)	11
● Introduction to Data Visualization with R (Part 2)	11
● Introduction to Bioconductor and report generation with R	11
● Required Course Materials	11

Introduction to R and RStudio

● Learning Objectives	13
● What is R?	13
● Why R?	13
● Where do we get R packages?	14
● Ways to run R	14
● What is RStudio?	14

● Getting Started with R and R Studio	15
● Creating an R project	15
● Creating an R script	16
● Introduction to the RStudio layout	16
● When to use Source vs Console?	17
● Uploading and exporting files from RStudio Server	17
● Data Management	18
● Saving your R environment (.Rdata)	18
● Navigating directories	19
● What is a path?	19
● Using functions	20
● Getting help	21
● Additional Sources for help	23
● Test your learning	24
● Acknowledgments	25

Basics of R Programming

● Objectives	26
● R objects	26
● Creating and deleting objects	26
● Naming conventions and reproducibility	27
● Reassigning objects	28
● Deleting objects	28
● Object data types	28
● Mathematical operations	31
● Vectors	32
● Test your learning	33

● Creating, subsetting, modifying, exporting	33
● Logical subsetting	36
● Other ways to handle missing data	38
● Using objects to store thresholds	38
● Using the %in% operator.	39
● Test your learning	39
● Saving and loading objects	40
● Exporting your R project	41
● Acknowledgments	41
● Additional Resources	41

R Data Structures: Introducing Data Frames

● Learning Objectives	42
● Data Structures	42
● What are factors?	42
● Important functions	42
● Lists	43
● Important functions	43
● Example	43
● Data Frames: Working with Tabular Data	45
● Best Practices for organizing genomic data	45
● Introducing the airway data	46
● Importing / exporting data	47
● Examining and summarizing data frames	49
● What is the length of our data.frame? What are the dimensions?	51
● Other useful functions for inspecting data frames	51

● Data frame coercion and accessors	51
● Using colnames() to rename columns	53
● Test your learning	54
● Exporting Data (Save the data frame to a file)	54
● Data Matrices	55
● Acknowledgements	58
● Resources	58

Data Frames and Data Wrangling (Part 1)

● Learning Objectives	59
● Best Practices for organizing genomic data	59
● Introducing tidy data	59
● What is tidy data?	59
● What is messy data?	60
● Tools for working with tidy data	64
● Load the core tidyverse packages	64
● Load the data	65
● Subsetting data frames with base R	67
● Using %in%	68
● Tips to remember for subsetting	68
● Data wrangling with tidyverse	69
● Subsetting with dplyr	69
● Selecting columns	69
● Test your learning	70
● Filtering by row	71
● Test your learning	73
● Acknowledgements	73

● Resources	73
-------------	----

Data Frames and Data Wrangling (Part 2)

● Learning Objectives	74
● Load the tidyverse	74
● Re-load the data	74
● Introducing the pipe	75
● Running code one step at a time	75
● Nesting code	75
● Using the Pipe	76
● We can pipe from the beginning to the end.	76
● Test your learning	77
● Mutate	78
● Create a new column using existing columns	78
● Coerce variables	78
● More examples	79
● Test your learning	80
● Arrange, group_by, summarize	80
● Using arrange()	81
● The slice functions	82
● Sample sizes (counts and tallies) and missing data	83
● Test your learning	85
● Data Reshaping	86
● Pivot wider	87
● Coerce to a matrix	87
● Pivot longer	88
● Reshaping for plotting	88

● Acknowledgements	90
● Resources	90

Introduction to Data Visualization with R (Part 1)

● Data visualization with ggplot2	92
● Objectives	92
● Why use R for Data Visualization?	92
● Introducing ggplot2	92
● The ggplot2 template	93
● Geom functions	97
● Mapping and aesthetics (aes())	98
● R objects can also store figures	102
● Colors	102
● Facets	108
● Using multiple geoms per plot	112
● Other data visualization options in R	116
● R base graphics	116
● Lattice	118
● Resource list	118
● Acknowledgements	119

Introduction to Data Visualization with R (Part 2)

● Objectives	120
● Our grammar of graphics template	120

● Loading the libraries	121
● Importing the data	121
● Statistical transformations	122
● Coordinate systems	126
● Labels, legends, scales, and themes	127
● Create a custom theme to use with multiple figures.	133
● Saving plots (ggsave())	135
● Nice plot example	135
● Recommendations for creating publishable figures	137
● Complementary packages	138
● Acknowledgements	138

Introduction to Bioconductor and report generation with R

● Objectives	139
● Introducing Bioconductor	139
● What types of packages are available in Bioconductor?	139
● Bioconductor versions and install	140
● Bioconductor release schedule	140
● How to install a Bioconductor package?	141
● How to find Bioconductor packages of interest?	141
● Bioconductor education and communication	142
● Resources for learning	142
● Communication	143
● Introduction to report generation with R.	143
● What is Quarto?	143
● Why use Quarto	144
● Gallery of examples	144

● Getting Started	145
● Open a new .qmd file	145
● Don't know markdown? No problem. Use the Visual editor.	147
● Anatomy of Quarto document	148
● Additional Resources	149
● Acknowledgements	149

Additional Exercises

Base R: Objects, vectors, and data types 151

-
- Lesson 2 Exercise Questions: Base R syntax, objects, and data types 151

Base R and data frames 154

-
- Lesson 3 Exercise Questions: BaseR dataframe manipulation and factors 154

Practicing the Tidyverse (Part 1) 157

-
- Lesson 4 Exercise Questions: Tidyverse 157

Practicing the Tidyverse (Part 2) 160

-
- Lesson 5 Exercise Questions: Tidyverse 160

ggplot2: Changing plot types 162

-
- Lesson 5 Exercise Questions: ggplot2 162

ggplot2: Making Pretty Plots 165

-
- Lesson 6 Exercise Questions: ggplot2 165

- Start by plotting `Petal.Length` on the x-axis and `Petal.Width` on the y-axis. 165

- Fix the axes so that the dimensions on the x-axis and the y-axis are equal. 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. 166

-
- Change to color of the points by species to be color blind friendly, and change the legend title to "Iris Species". Label the x and y axis to eliminate the variable names and add unit information. 167
 - Play with the theme to make this a bit nicer. 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. 169
 - Now, save your plot using ggsave. 170
-

Getting the Data

- Data Access 172

Getting help

- Need help? 174

References

- For Further Reading 176

-
- Books and / or Book Chapters of Interest 176
 - R Cheat Sheets 176
 - Other Resources 176

Course Overview

Welcome to the R Introductory Series!

A series of introductory lessons in R for scientists.

This course will include a series of lessons for individuals **new to R** or with **limited R experience**. The purpose of this course is to introduce the foundational skills necessary to begin to analyze and visualize data in R. This course is not designed for those with intermediate R experience and is not tailored to any one specific type of analysis.

Course Expectations

The course will include a series of eight lessons taught in 1-hour blocks over four weeks. Lessons will be on Tuesdays and Thursdays at 1 pm. Each lesson will be followed by an optional 1-hour help session. Content has been adapted from material provided by Data Carpentry [Intro to R and RStudio for Genomics](https://datacarpentry.org/genomics-r-intro/) (<https://datacarpentry.org/genomics-r-intro/>) (Link to the license (<https://creativecommons.org/licenses/by/4.0/>)) as well as [R for Data Science](https://r4ds.had.co.nz/index.html) (<https://r4ds.had.co.nz/index.html>).

Content Organization

Introduction to R and RStudio

This lesson will serve as a general introduction to R and RStudio. Attendees will explore the RStudio interactive development environment (IDE) and learn to create R projects and scripts, navigate between directories, use functions, and obtain help.

The Basics of R Programming

In this lesson, attendees will learn the most basic features of the R programming language including:

- R syntax
- Creating R objects
- Data types
- Using mathematical operations
- Using comparison operators
- Creating, subsetting, and modifying vectors

R Data Structures: Introducing Data Frames

This lesson will introduce data structures with a focus on data frames. Attendees will learn how to import, summarize, and explore data stored in data frames.

Data Frames and Data Wrangling (part 1)

This lesson will introduce data wrangling with R. Attendees will learn to filter data using base R and tidyverse (dplyr) functionality.

Data Frames and Data Wrangling (part 2)

In this lesson, attendees will learn how to transform, summarize, and reshape data using functions from the tidyverse.

Introduction to Data Visualization with R (part 1)

This lesson will introduce prominent ways to visualize data with R. The majority of the lesson will be devoted to learning how to create publishable figures using the ggplot2 package.

Introduction to Data Visualization with R (Part 2)

In this lesson, attendees will continue learning how to plot publishable figures with ggplot2.

Introduction to Bioconductor and report generation with R

This lesson will be divided into two parts. Part 1 will introduce Bioconductor, an R package repository for the analysis of biological data. Part 2 will introduce RMarkdown and Quarto for report generation with R.

Required Course Materials

To participate in this class you will need your government-issued computer and a reliable internet connection. You do not need to download or install any software to participate in the class. However, at the end of the class, we will provide instruction on installing R and R Studio on your local machine.

This class will be taught on the DNAnexus platform. Every learner will need to create a [DNAnexus account \(https://dnanexus.com\)](https://dnanexus.com).

DNAnexus Accounts

If you are not taking the live iteration of this course and you are following this documentation on your own, you do not need a DNAnexus account. DNAnexus is only accessible to course registrants during class times.

Video Recordings

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

Introduction to R and RStudio

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

 - how R can enhance data analysis reproducibility

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, 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 \(https://www.r-project.org/about.html\)](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 +20,000 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 3.18) includes 2,266 software packages, 429 experiment data packages, 920 annotation packages, 30 workflows, and 4 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 shareable format that can easily be installed by R users. The primary repository for R packages is the Comprehensive R Archive Network (CRAN). [CRAN \(https://cran.r-project.org/#:~:text=CRAN%20is%20a%20network%20of,you%20to%20minimize%20network%20load.\)](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 \(https://www.bioconductor.org/\)](https://www.bioconductor.org/) - more on this later.

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

Ways to run R

R can be used via command line interactively, [command line using a script \(https://support.rstudio.com/hc/en-us/articles/218012917-How-to-run-R-scripts-from-the-command-line\)](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 \(https://posit.co/products/open-source/rstudio/\)](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, but we highly encourage new users to install R and RStudio locally to their PC or macbook.

Note

RStudio the company is now [Posit \(https://posit.co/\)](https://posit.co/).

Installing R and RStudio

Macbook: Follow [these instructions \(https://posit.co/download/rstudio-desktop/\)](https://posit.co/download/rstudio-desktop/).

Windows: Request installation from [service.cancer.gov \(https://service.cancer.gov/ncisp\)](https://service.cancer.gov/ncisp).

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.org/genomics-r-intro/00-introduction.html) (<https://datacarpentry.org/genomics-r-intro/00-introduction.html>).

Creating an R project

Because we are working on DNAnexus, and our files will not remain at the end of each class, we aren't going to use a R project for all lessons. However, it is worth creating an R project and discussing the benefits here.

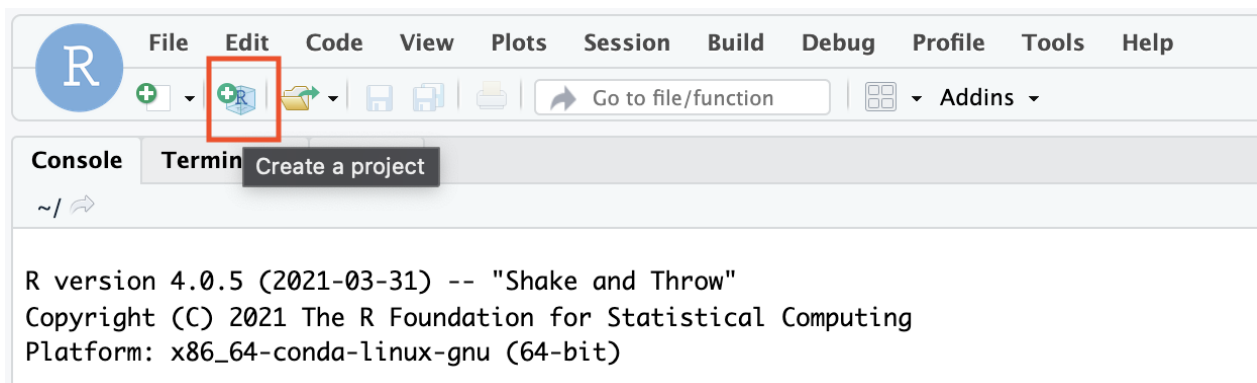
Creating an R project for each project you are working on facilitates organization and scientific reproducibility.

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/00-introduction.html) (<https://datacarpentry.org/genomics-r-intro/00-introduction.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 "LearningR"? To make your project more reproducible, consider clicking the option box for `renv`. The R project file ends in `.Rproj`.

One of the most wonderful and also frustrating aspects of working with R is managing packages. Unfortunately it is very common that you may run into versions of R and/or R packages that are not compatible. This may make it difficult

for someone to run your R script using their version of R or a given R package, and/or make it more difficult to run their scripts on your machine. `renv` is an RStudio add-on that will associate your packages and project so that your work is more portable and reproducible. To turn on `renv` click on the Tools menu and select Project Options. Under Environments check off "Use `renv` with this project" and follow any installation instructions. ---datacarpentry.org (<https://datacarpentry.org/genomics-r-intro/00-introduction.html>)

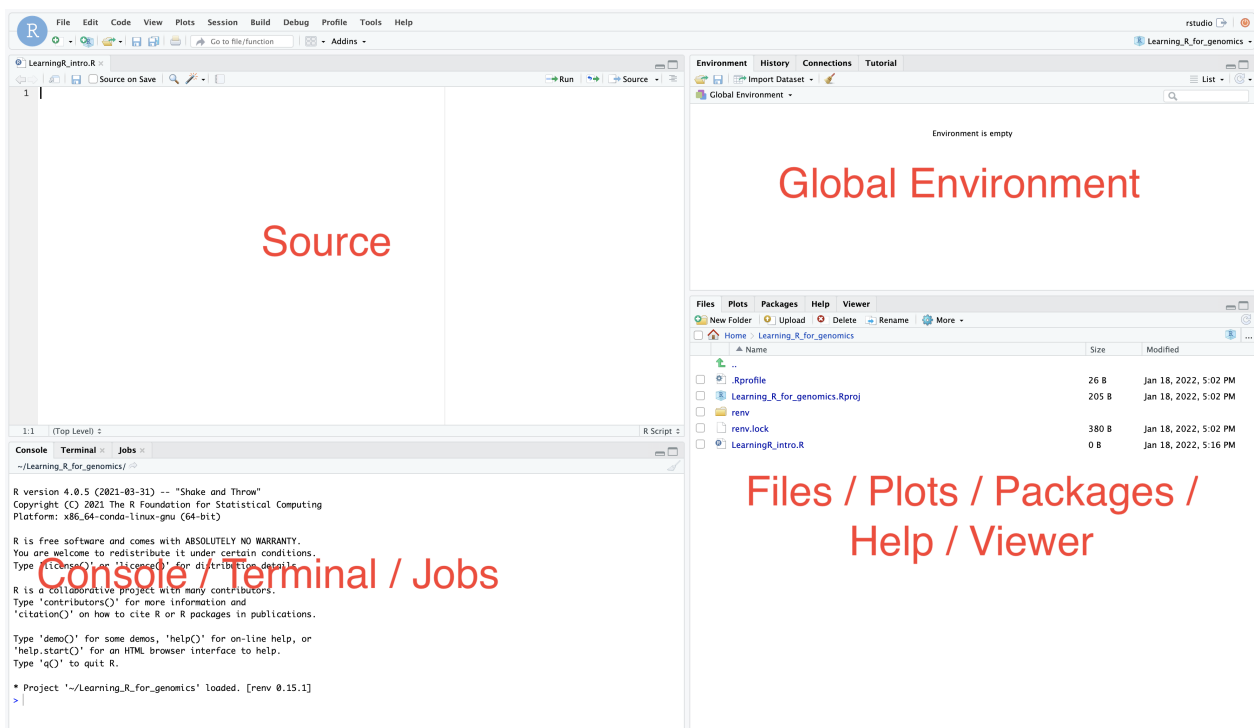
Read more about `renv` [here](https://rstudio.github.io/renv/articles/renv.html) (<https://rstudio.github.io/renv/articles/renv.html>).

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 "LearningR_intro". 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 R.

Files/Plots/Packages/Help/Viewer: This multipurpose 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.org/genomics-r-intro/00-introduction.html>)

Note

You can already see your R project and R script file in your 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 files (`.Rmd`).

Also, you can change your RStudio layout. See this [blog \(https://www.r-bloggers.com/2018/05/a-perfect-rstudio-layout/\)](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.

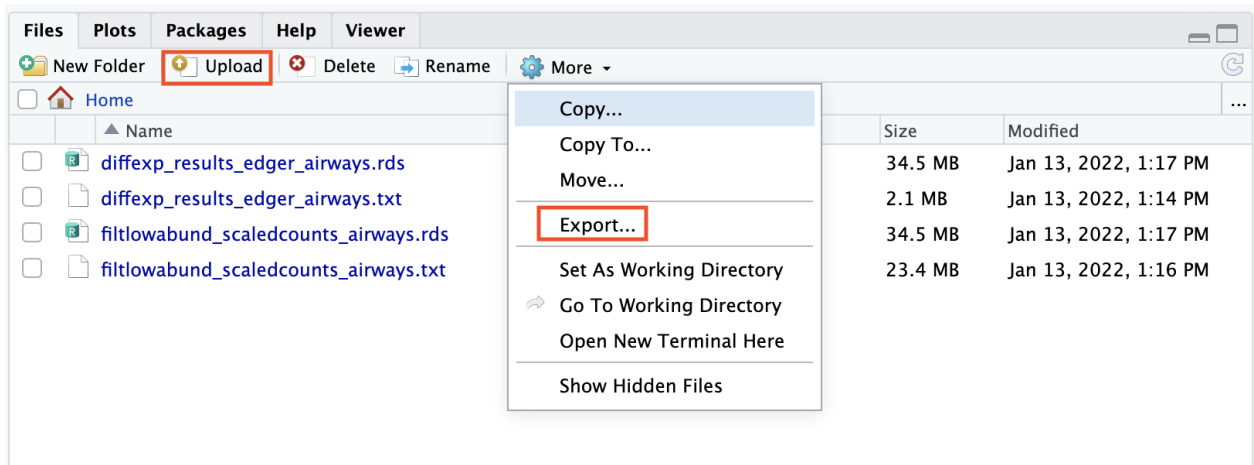
When to use Source vs Console?

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 the files created in the RStudio environment.



Warning

Your files will not remain when you exit the RStudio server session. If you want to keep notes or other files, you will need to export them.

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 \(https://bioinformatics.ccr.cancer.gov/docs/reproducible-r-on-biowulf/L3_PackageManagement/\)](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.

Tip

Do not use absolute file paths in scripts.

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()`.

Tip

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".

Navigating directories

Now we are ready to work with some of our first R commands. We are going to run commands directly from our R script rather than typing into the R console.

Our first command 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 command, 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] "/home/rstudio/LearningR"
```

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

We 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("/home/rstudio/LearningR")
```

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.

Using functions

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

An R function has three key properties:

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

We have already used some R functions (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. We will discuss other types of help in a moment.

```
round(x, digits = 0)
round(5.65, digits=1)
```

```
#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()`.

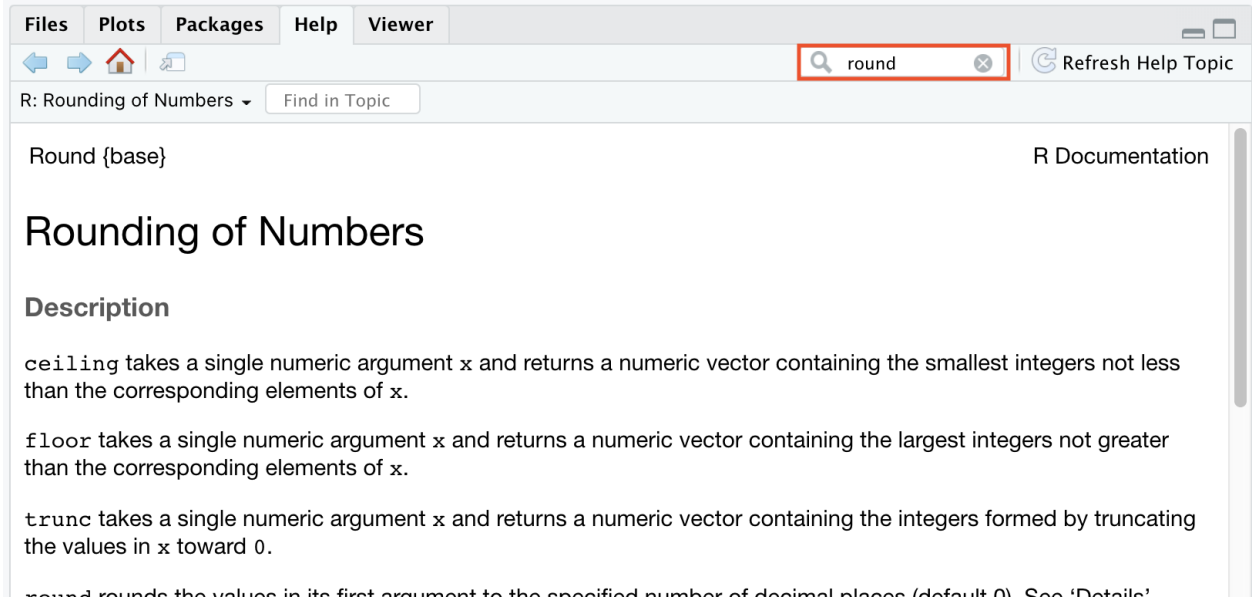
Info

See this [R reference card \(https://cran.r-project.org/doc/contrib/Short-refcard.pdf\)](https://cran.r-project.org/doc/contrib/Short-refcard.pdf) for a list of useful functions to get to know.

Getting help

Now we know a bit about using functions, but what if I had no idea what the function `round()` 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).



The screenshot shows the RStudio Help Viewer window. The search bar at the top contains the word "round". The main content area displays the R documentation for the `round()` function, including the title "Rounding of Numbers", a "Description" section, and the start of the "Arguments" section.

Files Plots Packages Help Viewer

← → Home Refresh Help Topic

R: Rounding of Numbers Find in Topic

Round {base} R Documentation

Rounding of Numbers

Description

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers formed by truncating the values in `x` toward 0.

`round` rounds the values in its first argument to the specified number of decimal places (default 0). See 'Details'

Alternatively, you can search directly for help in the console using `?round()` or `??round()`. `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(round)
```

```
## function (x, digits = 0)
## NULL
```

`round()` takes two arguments, `x`, which is the number to be rounded, and a `digits` argument. The `=` sign indicates that a default (in this case 0) is already set. Since `x` is not set, `round()` requires we provide it, in contrast to `digits` where R will use the default value 0 unless you explicitly provide a different value. --- [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html\)](https://datacarpentry.org/genomics-r-intro/01-introduction/index.html)

R arguments are also positional, so instead of including `digits=1` in our above use of `round()`, we could instead do the following:


```
round(5.65, 1)
```

```
## [1] 5.7
```

Additional Sources for help

Try googling your problem or using some other search engine. [rseek \(https://rseek.org/\)](https://rseek.org/) is an R specific search engine that searches several R related sites. If using google 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 \(https://reprex.tidyverse.org/\)](https://reprex.tidyverse.org/) package particularly useful.

To provide details about your R session, use

```
sessionInfo()
```

```

## R version 4.3.2 (2023-10-31)
## Platform: x86_64-apple-darwin20 (64-bit)
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resou
## LAPACK: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resou
##
## 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: America/New_York
## tzcode source: internal
##
## attached base packages:
## [1] stats      graphics  grDevices  utils      datasets  methods    ba
##
## loaded via a namespace (and not attached):
## [1] digest_0.6.33      R6_2.5.1          fastmap_1.1.1      xfun_0
## [5] cachem_1.0.8       knitr_1.45        htmltools_0.5.7    rmarkdc
## [9] cli_3.6.1          sass_0.4.7        jquerylib_0.1.4    compile
## [13] rstudioapi_0.15.0  tools_4.3.2       evaluate_0.23      bslib_0
## [17] yaml_2.3.7         rlang_1.1.2       jsonlite_1.8.7

```

Test your learning

- Which of the following functions is used to print your working directory in R?
 - pwd
 - Setwd()
 - getwd()
 - wkdir()

{{Sdet}}

Solution{{Esum}}

C

{{Edet}}

- Which of the following can be used to learn more regarding an R function?
 - ?function()
 - ??function()
 - args(function)
 - All of the above

{{Sdet}}

```
Solution{{Esum}}
```

```
D
```

```
{{Edet}}
```



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) (<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/) (<https://carpentries-incubator.github.io/bioc-intro/>), which is part of the [Carpentries Incubator](https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator) (<https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator>).

Basics of R Programming

Objectives

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

- Creating R objects and understanding object types
- Using mathematical operations
- Using comparison operators
- Creating, subsetting, and modifying vectors

By the end of this section, you should understand what an object and vector is and how to access and work with objects and vectors.

R objects

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

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.

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`), or type directly into the console).

```
#You can and should annotate your code with comments 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.

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 an error  
1a
```

```
## Error: <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?

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

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

How do I know what objects have been created?

To view a list of the objects you have created, use `ls()` or look at your global environment pane.***

Reassigning objects

To reassign an object, simply overwrite the object.

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

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

```
#if instead we want to reassign gene_name to a different gene,  
#we would use:  
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 in eval(expr, envir, enclos): object 'gene_name' not found
```

Object data types

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

have certain assigned attributes (related to class), and these attributes will be important for how they interact with certain methods / functions. Ultimately, understanding the mode / type and class of an object will be important for how an object can be used in R. When the mode of an object is changed, we call this "coercion". You may see a coercion warning pop up when working with objects in the future.

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

The most common modes (from datacarpentry.org); Other examples: complex, raw, etc. (See ? typeof()).

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 an 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 \(https://datacarpentry.org/genomics-r-intro/01-r-basics.html\)](https://datacarpentry.org/genomics-r-intro/01-r-basics.html)

The mode or type of an object can be examined using `mode()` or `typeof()`, while the class of an object can be viewed using `class()`.

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

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

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

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

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

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

As you can see, the output of `mode()` and `typeof()` is largely the same but `typeof()` does differ in some cases and is based on the storage mode. So numeric types can be stored in memory differently, with doubles taking up more memory than an integer, for example. If this is confusing, you can always read the documentation `?mode()` and `?typeof()`. Searching for help provided this nifty R explanation for mode vs type names.

Mode names

Modes have the same set of names as types (see [typeof](#)) except that

- types "integer" and "double" are returned as "numeric".
- types "special", "builtin" and "closure" are returned as "function".
- type "symbol" is called mode "name".
- type "language" is returned as "(" or "call".

On the other hand,

'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, its 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%20obj>)

There are also functions that can gauge types directly, for example, `is.numeric()`, `is.character()`, `is.logical()`. It is often most useful to use `class()` and `typeof()` to find out more about an object or `str()` (more on this function later).

There are some special use, null-able values. Read more to learn about [NULL](https://www.r-bloggers.com/2018/07/r-null-values-null-na-nan-inf/), [NA](https://www.r-bloggers.com/2018/07/r-null-values-null-na-nan-inf/), [NaN](https://www.r-bloggers.com/2018/07/r-null-values-null-na-nan-inf/), and [Inf](https://www.r-bloggers.com/2018/07/r-null-values-null-na-nan-inf/) (<https://www.r-bloggers.com/2018/07/r-null-values-null-na-nan-inf/>).

Mathematical operations

As mentioned, an object's 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/01-r-basics.html) (<https://datacarpentry.org/genomics-r-intro/01-r-basics.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)

Let's see this in practice.

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

```
## [1] "numeric"
```

```
#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
```

Vectors

Vectors are probably the most used commonly used object type in R. A vector is a collection of values that are all of the same type (numbers, characters, etc.). The columns that make up a data frame 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 \(https://datacarpentry.org/genomics-r-intro/01-r-basics.html\)](https://datacarpentry.org/genomics-r-intro/01-r-basics.html).

```
#create a vector of gene names
transcript_names<-c("TSPAN6","TNMD","SCYL3","GCLC")
#Let's check out the mode. What do you think?
mode(transcript_names)
## [1] "character"
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.

```
str(transcript_names) #this will return properties of the object's un
```

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

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

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

Test your learning

Given the following R code:

```
numbers<- c("1", "2.56", "83", "678")
```

What type of data is stored in this vector?

- a. double
- b. character
- c. logical
- d. complex

{{Sdet}}

Solution{{Esum}}

B

{{Edet}}

Creating, subsetting, modifying, exporting

Let's learn how to further work with vectors, including creating, sub-setting, modifying, and saving.

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

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.

```
#Get the second value from the vector 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://datacarpentry.org/genomics-r-intro/01-r-basics.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.

```
#Retrieve the second and third value from cell_types
cell_line[2:3]
```

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

```
#Retrieve the first, fifth, and sixth values from transcript_counts
transcript_counts[c(1,5:6)]
```

```
## [1] 679 60 0
```

The combine function `c()` can be used to add an element to a vector.

```
#Lets add a gene to transcript_names
transcript_names<-c(transcript_names,"ANAPC10P1","ABCD1")
#The object will not be overwritten without assigning it to a name
transcript_names
## [1] "TSPAN6" "TNMD" "SCYL3" "GCLC" "ANAPC10P1" "ABCD1"
```

Indexing 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 also call a value directly
#Rename "ABCD1" to "NEW"; more on this to come
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 (See table below).

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 subsetting 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 include `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 subsetting based on the absence of NAs.

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

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

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  
transcript_counts[transcript_counts>trnsc_cutoff] #note this will also
```

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

```
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 subsetting. 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
# character vector
c("NNAME","ANAPC10P1") %in% transcript_names
```

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

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

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

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

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

This type of searching will come in handy when we discuss filtering in Lesson 2.

Test your learning

Given the following R code:

```
fruit<-c("apples", "bananas", "oranges", "grapes","kiwi","kumquat")
```

What does `fruit[5]<-"mango"` do?

- renames the object "fruit" to "mango"
- adds "mango" to an existing vector named "fruit"
- replaces "bananas" with "mango"
- replaces "kiwi" with "mango"

{{Sdet}}

Solution{{Esum}}

D

{{Edet}}

Given the following R code:

```
Total_subjects <- c(23, 4, 679, 3427, 12, 890, 654)
```

Which of the following could be used to return all values less than 678 in the vector "Total_subjects"?

- `Total_subjects < 678`
- `Total_subjects[> 678]`
- `Total_subjects(Total_subjects < 678)`
- `Total_subjects[Total_subjects < 678]`

{{Sdet}}

Solution{{Esum}}

D

{{Edet}}

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()`.

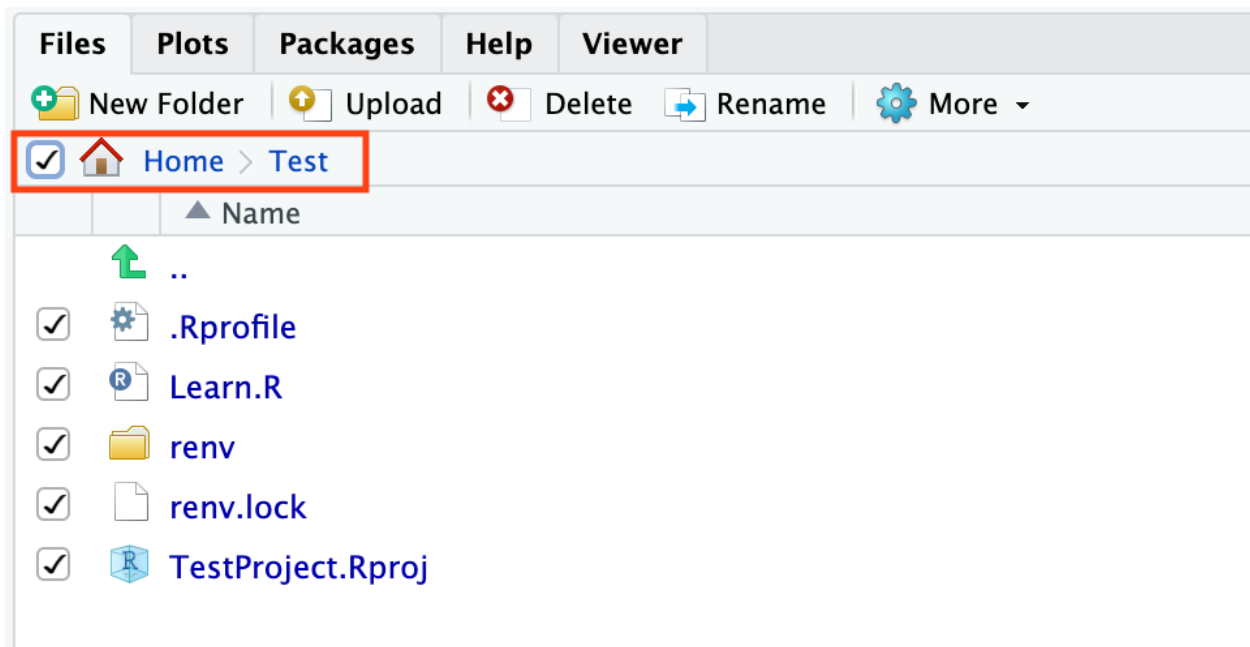
Let's save our `transcript_counts` vector to our working directory.

```
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()`).

Exporting your R project

To use the materials you generated on the RServer on DNAnexus on your local computer, let's export our files. To do this, let's select all files in our working directory. This will export a zipped file with the contents of your working directory.



If you plan to use these files again on DNAnexus, simply use Upload. To upload a directory, the contents must be zipped. To zip a directory on a mac, simply right click on the directory and select Compress "directory_name". To zip a directory on a PC, right click the folder and choose "Send to: Compressed (zipped) folder".

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) (<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/) (<https://carpentries-incubator.github.io/bioc-intro/>), which is part of the [Carpentries Incubator](https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator) (<https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator>).

Additional Resources

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

R Data Structures: Introducing Data Frames

Learning Objectives

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

Data Structures

Data structures are objects that store data.

Previously, we learned that **vectors** are [collections of values of the same type \(https://datacarpentry.org/genomics-r-intro/01-r-basics.html#vectors\)](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. 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.

Important functions

- `factor()` - to create a factor and reorder levels
- `as.factor()` - to coerce to a factor
- `levels()` - view 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.

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.

Lists

Unlike an atomic vector, a list can contain multiple elements of different types, (e.g., character vector, numeric vector, list, data frame, matrix).

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" "SRR:
 $ 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>)

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 \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes.html\)](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 \(https://www.howtogeek.com/437958/how-to-use-the-chmod-command-on-linux/\)](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.

2. "Keep spreadsheet data Tidy" -- [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes.html\)](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, which we will talk about more next lesson, can make data science more efficient, effective, and reproducible.

3. "Trust but verify" -- [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes.html\)](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. Hopefully, today's lesson will help with this. 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 \(https://ademos.people.uic.edu/Chapter7.html#32_b_using_the_rep_function_to_create_data_frames\)](https://ademos.people.uic.edu/Chapter7.html#32_b_using_the_rep_function_to_create_data_frames).

Introducing the airway data

There are data sets available in R to practice with or showcase different packages. For today's lesson and the remainder of this course, we will use data from the Bioconductor package `airway` (<https://bioconductor.org/packages/release/data/experiment/html/airway.html>) to showcase tools used for data wrangling and visualization. The use of this data was inspired by a 2021 workshop entitled *Introduction to Tidy Transcriptomics* (https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html) by Maria Doyle and Stefano Mangiola. Code has been adapted from this workshop to explore tidyverse functionality.

The airway data is from [Himes et al. \(2014\) \(https://pubmed.ncbi.nlm.nih.gov/24926665/\)](https://pubmed.ncbi.nlm.nih.gov/24926665/). These data, which are contained within a `RangedSummarizedExperiment` object are from a bulk RNAseq experiment. In the experiment, the authors "characterized transcriptomic changes in four primary human ASM cell lines that were treated with dexamethasone," a common therapy for asthma. The `airway` package includes RNAseq count data from 8 airway smooth muscle cell samples. Each cell line includes a treated and untreated negative control.

Note

Current recommendations indicate that there should be 3-5 sample replicates for an RNAseq experiment.

Do not worry about the `RangedSummarizedExperiment`. The data we will use today and next week have been provided to you in the following files:

- `filtnlowabund_scaledcounts_airways.txt` - Includes scaled transcript count data.
- `diffexp_results_edger_airways.txt` - Includes results from differential expression analysis using EdgeR.

Object (`.rds`) files have also been included.

Note

Bioconductor will be discussed further in Lesson 8.

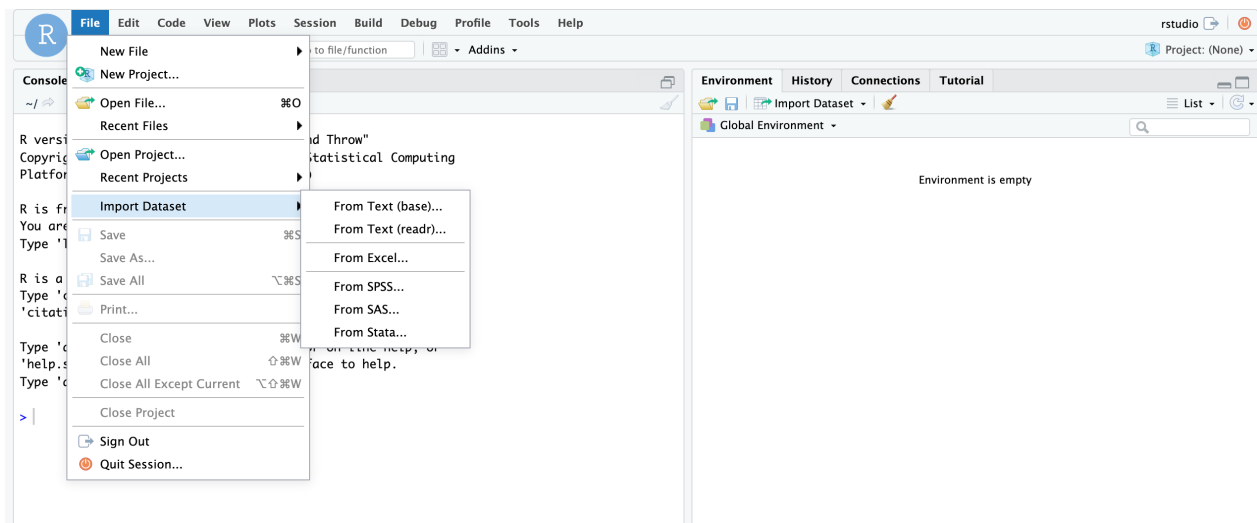
Importing / exporting data

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

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

Note

`readr` is a tidyverse package, but it isn't necessary for import. You can read more about `readr` and its advantages [here \(https://readr.tidyverse.org/\)](https://readr.tidyverse.org/).



Let's focus on the base R import functions. These include `read.csv()`, `read.table()`, `read.delim()`, etc. You should examine the function arguments (e.g., `?read.delim()`) to get an idea of what is happening at import and ensure that your data is being parsed correctly.

```
#Let's import our data and save to an object called scaled_counts
scaled_counts<-read.delim(
  "./data/filtlowabund_scaledcounts_airways.txt", as.is=TRUE)
```

We can now see this object in our RStudio environment pane.

The screenshot shows the RStudio Environment pane. At the top, there are tabs for 'Environment', 'History', 'Connections', and 'Tutorial'. Below the tabs, there are icons for 'Import Dataset', '676 MiB', and a search icon. The 'Environment' pane shows a data object named 'scaled_counts' with 127408 observations and 18 variables. The data object is highlighted in blue.

This object can be viewed by clicking on it in the environment pane. Alternatively, you can use `View(scaled_counts)`.

feature	sample	counts	SampleName	cell	dex	albut	Run	avgLength	Experiment	Sample	BioSamp	
1	ENSG00000000003	508	679	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
2	ENSG000000000419	508	467	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
3	ENSG000000000457	508	260	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
4	ENSG000000000460	508	60	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
5	ENSG000000000971	508	3251	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
6	ENSG00000001036	508	1433	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
7	ENSG00000001084	508	519	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
8	ENSG00000001167	508	394	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
9	ENSG00000001460	508	172	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
10	ENSG00000001461	508	2112	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
11	ENSG00000001497	508	524	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
12	ENSG00000001561	508	71	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
13	ENSG00000001617	508	555	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
14	ENSG00000001629	508	1660	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
15	ENSG00000001630	508	59	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
16	ENSG00000001631	508	729	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
17	ENSG00000002016	508	201	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
18	ENSG00000002330	508	206	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
19	ENSG00000002549	508	1459	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
20	ENSG00000002586	508	7507	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
21	ENSG00000002746	508	151	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO
22	ENSG00000002822	508	411	GSM1275862	N61311	untrt	untrt	SRR1039508	126	SRX384345	SRS508568	SAMNO

Showing 1 to 22 of 127,408 entries, 18 total columns

To import an existing object, we use `readRDS()`.

```
#Let's import our data from the .rds file
#and save to an object called scaled_counts_rds
scaled_counts_rds<-
  data.frame(readRDS("./data/filtlowabund_scaledcounts_airways.rds"))
```

Note

Using RStudio functionality, you can navigate to the files tab and click on the `.rds` file of interest. You will receive a prompt asking if you would like to load the object into R.

To export data to file, you will use similar functions (`write.table()`, `write.csv()`, `saveRDS()`, etc.). We will show how these work later in the lesson.

Examining and summarizing data frames

The object that we imported, `scaled_counts`, is a data frame. Let's learn a bit more about our data frame. First, we can learn more about the structure of our data using `str()`. We have seen this function in use previously.

```
str(scaled_counts)
```

```
'data.frame': 127408 obs. of 18 variables:
 $ feature      : chr  "ENSG000000000003" "ENSG000000000419" "ENSG000000000003" ...
 $ sample      : int  508 508 508 508 508 508 508 508 508 508 ...
 $ counts      : int  679 467 260 60 3251 1433 519 394 172 2112 ...
 $ SampleName  : chr  "GSM1275862" "GSM1275862" "GSM1275862" "GSM1275862" ...
 $ cell       : chr  "N61311" "N61311" "N61311" "N61311" ...
 $ dex        : chr  "untrt" "untrt" "untrt" "untrt" ...
 $ albut      : chr  "untrt" "untrt" "untrt" "untrt" ...
 $ Run       : chr  "SRR1039508" "SRR1039508" "SRR1039508" "SRR1039508" ...
 $ avgLength  : int  126 126 126 126 126 126 126 126 126 126 ...
 $ Experiment : chr  "SRX384345" "SRX384345" "SRX384345" "SRX384345" ...
 $ Sample     : chr  "SRS508568" "SRS508568" "SRS508568" "SRS508568" ...
 $ BioSample  : chr  "SAMN02422669" "SAMN02422669" "SAMN02422669" "SAMN02422669" ...
 $ transcript : chr  "TSPAN6" "DPM1" "SCYL3" "C1orf112" ...
 $ ref_genome : chr  "hg38" "hg38" "hg38" "hg38" ...
 $ .abundant  : logi TRUE TRUE TRUE TRUE TRUE TRUE ...
 $ TMM       : num  1.06 1.06 1.06 1.06 1.06 ...
 $ multiplier : num  1.42 1.42 1.42 1.42 1.42 ...
 $ counts_scaled: num  960.9 660.9 367.9 84.9 4600.7 ...
```

`str()` shows us that we are looking at a data frame object with 127,408 observations in 18 variables (or 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 type (character, integer, logical, numeric) 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(scaled_counts)
```

```

  feature          sample      counts      SampleName
Length:127408   Min.   :508.0   Min.    :    0   Length:127408
Class :character 1st Qu.:511.2   1st Qu.:   66   Class :character
Mode  :character Median :514.5   Median :   310   Mode  :character
                Mean  :514.5   Mean   :  1376
                3rd Qu.:517.8   3rd Qu.:   960
                Max.  :521.0   Max.   :513766

  cell          dex          albut          Run
Length:127408   Length:127408   Length:127408   Length:127408
Class :character Class :character Class :character Class :character
Mode  :character Mode  :character Mode  :character Mode  :character

  avgLength      Experiment      Sample      BioSample
Min.   : 87.0     Length:127408   Length:127408   Length:127408
1st Qu.:100.2    Class :character Class :character Class :character
Median :123.0    Mode  :character Mode  :character Mode  :character
Mean   :113.8
3rd Qu.:126.0
Max.   :126.0

  transcript      ref_genome      .abundant      TMM
Length:127408    Length:127408    Mode:logical    Min.   :0.9512
Class :character Class :character TRUE:127408     1st Qu.:0.9706
Mode  :character Mode  :character                    Median :1.0052
                                           Mean   :1.0006
                                           3rd Qu.:1.0257
                                           Max.   :1.0553

  multiplier      counts_scaled
Min.   :1.026     Min.   :    0.0
1st Qu.:1.230     1st Qu.:   95.4
Median :1.467     Median :  445.8
Mean   :1.466     Mean   : 1933.7
3rd Qu.:1.581     3rd Qu.: 1369.6
Max.   :2.136     Max.   :632885.3

```

Our data frame has 18 variables, so we get 18 fields that summarize the data. Counts, avgLength, TMM, multiplier, and counts_scaled are numerical data and so we get summary statistics on the min and max values for these columns, as well as mean, median, and interquartile ranges.

Tip

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

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

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

```
[1] 18
```

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

```
[1] 127408    18
```

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](https://carpentries-incubator.github.io/bioc-intro/25-starting-with-data.html)", *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

Notice that "sample" was treated as numeric, rather than as a character vector. If we intend to work with this column, 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 "sample" from `scaled_counts`. We use `head()` to limit the printed output.

```
#Using $  
head(scaled_counts$sample)
```

```
[1] 508 508 508 508 508 508
```

```
#Using []  
head(scaled_counts["sample"])
```

```
  sample  
1    508  
2    508  
3    508  
4    508  
5    508  
6    508
```

```
#Using [[]]  
head(scaled_counts[["sample"]])
```

```
[1] 508 508 508 508 508 508
```

Let's convert the "sample" 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(scaled_counts$sample)
```

```
[1] TRUE
```

```
#let's convert it to a character vector  
scaled_counts$sample<-as.character(scaled_counts$sample)  
#check this  
is.character(scaled_counts$sample)
```

```
[1] TRUE
```

```
#check this
is.numeric(scaled_counts$sample)
```

```
[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(scaled_counts$Sample))
```

```
Warning in head(as.numeric(scaled_counts$Sample)): NAs introduced by
```

```
[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 its 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 when we tried to create a vector with multiple data types, R chose one type through implicit coercion.
- Check the structure (`str()`) of your data frames before working with them! --- [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/05-dplyr/index.html\)](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 `[]` subsetting to easily modify column names.

For example, let's rename the column "Sample" to "Accession".

```
#Let's rename "Sample" to "Accession"
colnames(scaled_counts)[11]<-"Accession"

#if unsure of the index of the "Sample" column, you could use which()
which(colnames(scaled_counts)=="Sample")
```

```
#or you could get the indices in a data frame
data.frame(colnames(scaled_counts))

#or something like this
colnames(scaled_counts)[colnames(scaled_counts) ==
                        "Sample"] <- "Accession"
```

Test your learning

Which of the following will NOT print the "Run" column from scaled_counts?

- a. scaled_counts\$Run
- b. scaled_counts["Run"]
- c. scaled_counts[8,]
- d. scaled_counts[8]

{{Sdet}}

Solution{{Esum}}

C

{{Edet}}

What is the column index for "avgLength" from the scaled_counts df?

- a. 3
- b. 8
- c. 12
- d. 9

{{Sdet}}

Solution{{Esum}}

D {{Edet}}

Exporting Data (Save the data frame to a file)

If we want to export our df (scaled_counts) to use with another program, we can write out to a file.

```
write.table(scaled_counts,
            file = "scaled_counts_mod.txt",
            quote=FALSE, row.names=FALSE, sep="\t")
```

If you are unsure what these arguments mean, use ?write.table().

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 $m \times n$). 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"
```

Earlier, we mentioned that elements in a matrix can be referenced by their row and column number. Below, we extract the element in the 3rd row and 4th column of `a1` (which is 2)

```
a1[3,4] ## returns 2
```

```
[1] 2
```

We can assign column and row names to a matrix.

```
colnames(a1) <- c("control1", "control2", "tumor1", "tumor2")
rownames(a1) <- c("ADA", "AMPD2", "HPRT")
a1
```

```
      control1 control2 tumor1 tumor2
ADA           3         4         8         5
AMPD2          4         6         1         3
HPRT           2         3         7         2
```

But, we cannot reference columns using \$.

```
a1$control1
```

```
Error in a1$control1: $ operator is invalid for atomic vectors
```

We can create matrices mixed with words and numbers (see a2).

```
a2 <- matrix(c("apples", "pears", "oranges", 50, 25, 75), ncol=2)
a2
```

```
      [,1]      [,2]
[1,] "apples" "50"
[2,] "pears"  "25"
[3,] "oranges" "75"
```

But, R will coerce all of the elements to the same type, in this case character.

```
typeof(a2)
```

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

```
typeof(a2[,2])
```

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

```
class(a2)
```

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

We can also perform mathematical operations on matrices.

```
a3 <- 5
a3
```

```
[1] 5
```

Below we multiply every element in a1 by a3 and store in a4. Note, we are still left with a 3 by 4 matrix except the values have been multiplied by the value assigned to a3 (5).

```
a4 <- a1*a3
a1
```

	control1	control2	tumor1	tumor2
ADA	3	4	8	5
AMPD2	4	6	1	3
HPRT	2	3	7	2

```
a4
```

	control1	control2	tumor1	tumor2
ADA	15	20	40	25
AMPD2	20	30	5	15
HPRT	10	15	35	10

Here are some similarities and differences between matrices and data frames:

	Characteristic	Matrix	Data.frame
1	is rectangular data table	yes	yes
2	can perform math operations	yes	yes
3	needs homogenous data type	yes	no
4	can have heterogeneous data type	no	yes
5	can reference using row and column number	yes	yes
6	can reference column using \$	no	yes
7	can use for plotting	yes	yes

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) (<https://datacarpentry.org/genomics-r-intro/aio.html>) and from a 2021 workshop entitled *Introduction to Tidy Transcriptomics* (https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html) by Maria Doyle and Stefano Mangiola.

Resources

1. [BaseR cheatsheet](#)

Data Frames and Data Wrangling (Part 1)

This lesson will introduce data wrangling with R. Attendees will learn to filter data using base R and tidyverse (dplyr) functionality.

Learning Objectives

- Understand the concept of tidy data.
- Become familiar with the tidyverse packages.
- Be able to filter a data frame by rows and columns using base R and dplyr.

Best Practices for organizing genomic data

Let's refer back to our best practices for organizing genomic data.

1. "Keep raw data separate from analyzed data" -- [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html\)](https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html)
2. "Keep spreadsheet data Tidy" -- [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html\)](https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html)

But, what is tidy data???

3. "Trust but verify" -- [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html\)](https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html)

Introducing tidy data

What is tidy data?

Tidy data is an approach (or philosophy) to data organization and management. **There are three rules to tidy data:** (1) each variable forms its own column, (2) each observation forms a row, and (3) each value has its own cell. One advantage to following these rules is that the data structure remains consistent, making it easier to understand the tools that work well with the underlying structure, and there are a lot of tools in R built specifically to interact with tidy data. Equipped with the right tools will make data analysis more efficient. See the infographics below.

“**TIDY DATA** is a standard way of mapping the meaning of a dataset to its structure.”

—HADLEY WICKHAM

In tidy data:

- each **variable** forms a **column**
- each **observation** forms a **row**
- each **cell** is a **single measurement**

each column a variable

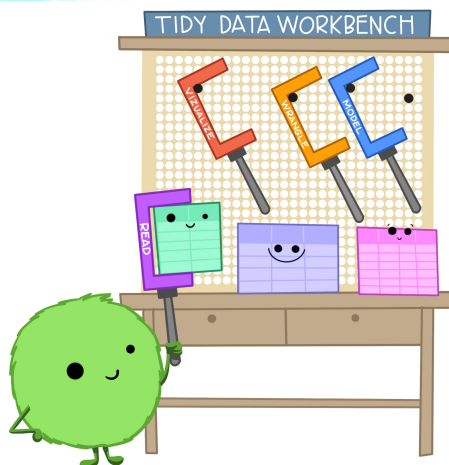
id	name	color
1	floof	gray
2	max	black
3	cat	orange
4	donut	gray
5	merlin	black
6	panda	calico

each row an observation

Wickham, H. (2014). Tidy Data. Journal of Statistical Software 59 (10). DOI: 10.18637/jss.v059.i10

Image from Lowndes and Horst 2020: Tidy Data for Efficiency, Reproducibility, and Collaboration (<https://www.openscapes.org/blog/2020/10/12/tidy-data/>)

When working with tidy data, we can use the **same tools** in **similar ways** for different datasets...



...but working with untidy data often means reinventing the wheel with **one-time approaches** that are **hard to iterate or reuse**.

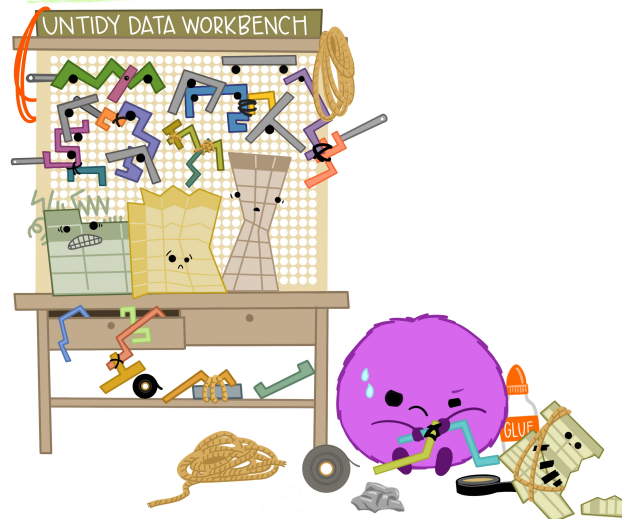


Image from Lowndes and Horst 2020: Tidy Data for Efficiency, Reproducibility, and Collaboration (<https://www.openscapes.org/blog/2020/10/12/tidy-data/>)

What is messy data?

“Tidy datasets are all alike, but every messy dataset is messy in its own way.” — Hadley Wickham.

Messy data sets tend to share five common problems:

1. Column headers are values, not variable names.
2. Multiple variables are stored in one column.
3. Variables are stored in both rows and columns.
4. Multiple types of observational units are stored in the same table.
5. A single observational unit is stored in multiple tables. --- [Hadley Wickham, Tidy Data](http://vita.had.co.nz/papers/tidy-data.pdf) (<http://vita.had.co.nz/papers/tidy-data.pdf>)

Let's look at a few examples of untidy data presented in [Tidyverse Skills for Data Science](https://jhubdatascience.org/tidyversecourse/intro.html#examples-of-untidy-data) (<https://jhubdatascience.org/tidyversecourse/intro.html#examples-of-untidy-data>).

Remember our data frame `scaled_counts`? `scaled_counts` is a tidy data frame. Let's take a moment to envision an **untidy data frame** containing the same data. Again, remember, there are infinite possibilities for messy data, but here is one example.

The original data frame:

```
#import the data and save to an object called scaled_counts
scaled_counts<-read.delim(
  "./data/filtflowabund_scaledcounts_airways.txt", as.is=TRUE)
head(scaled_counts)
```

```
      feature sample counts SampleName cell dex albut
1 ENSG00000000003    508    679 GSM1275862 N61311 untrt untrt SRR1039
2 ENSG000000000419    508    467 GSM1275862 N61311 untrt untrt SRR1039
3 ENSG000000000457    508    260 GSM1275862 N61311 untrt untrt SRR1039
4 ENSG000000000460    508     60 GSM1275862 N61311 untrt untrt SRR1039
5 ENSG000000000971    508   3251 GSM1275862 N61311 untrt untrt SRR1039
6 ENSG00000001036    508   1433 GSM1275862 N61311 untrt untrt SRR1039
  avgLength Experiment      Sample      BioSample transcript ref_genome
1      126   SRX384345   SRS508568   SAMN02422669      TSPAN6      hg38
2      126   SRX384345   SRS508568   SAMN02422669      DPM1      hg38
3      126   SRX384345   SRS508568   SAMN02422669      SCYL3      hg38
4      126   SRX384345   SRS508568   SAMN02422669   C1orf112      hg38
5      126   SRX384345   SRS508568   SAMN02422669      CFH      hg38
6      126   SRX384345   SRS508568   SAMN02422669      FUCA2      hg38
  TMM multiplier counts_scaled
1 1.055278 1.415149 960.88642
2 1.055278 1.415149 660.87475
3 1.055278 1.415149 367.93883
4 1.055278 1.415149 84.90896
5 1.055278 1.415149 4600.65058
6 1.055278 1.415149 2027.90904
```

An untidy version (subset) of `scaled_counts`.

```

                    508
cell                N61311
dex                untrt
SampleName         GSM1275862
Run / Experiment / Accession SRR1039508;SRX384345;SRS508568
TSPAN6            960.886417275434
DPM1              660.874752382368
SCYL3            367.938834302817
C1orf112         84.9089617621886
CFH              4600.65057814792

                    509
cell                N61311
dex                trt
SampleName         GSM1275863
Run / Experiment / Accession SRR1039509;SRX384346;SRS508567
TSPAN6            716.779730254346
DPM1              823.976698841491
SCYL3            337.590453311757
C1orf112         87.9975115267612
CFH              5886.23354376281

                    512
cell                N052611
dex                untrt
SampleName         GSM1275866
Run / Experiment / Accession SRR1039512;SRX384349;SRS508571
TSPAN6            1075.40953718585
DPM1              764.982041915709
SCYL3            323.977901809712
C1orf112         49.2742055984354
CFH              7609.16919953838

                    513
cell                N052611
dex                trt
SampleName         GSM1275867
Run / Experiment / Accession SRR1039513;SRX384350;SRS508572
TSPAN6            871.667100344899
DPM1              779.800224573256
SCYL3            350.375991315107
C1orf112         74.7753640001752
CFH              9084.13850653557

                    516
cell                N080611
dex                untrt
SampleName         GSM1275870

```



```

Run / Experiment / Accession SRR1039516;SRX384353;SRS508575
TSPAN6                1392.02747542151
DPM1                  718.031746988071
SCYL3                 299.689570719041
C1orf112              95.4113735350417
CFH                   8221.28001960276
                        517
cell                   N080611
dex                   trt
SampleName            GSM1275871
Run / Experiment / Accession SRR1039517;SRX384354;SRS508576
TSPAN6                1074.3148432507
DPM1                  819.844851726182
SCYL3                 339.635351591197
C1orf112              64.6435865566326
CFH                   11314.6798247617
                        520
cell                   N061011
dex                   untrt
SampleName            GSM1275874
Run / Experiment / Accession SRR1039520;SRX384357;SRS508579
TSPAN6                1212.77045557059
DPM1                  656.786077886933
SCYL3                 366.981189802531
C1orf112              119.702018991383
CFH                   8152.33750393948
                        521
cell                   N061011
dex                   trt
SampleName            GSM1275875
Run / Experiment / Accession SRR1039521;SRX384358;SRS508580
TSPAN6                868.672540272425
DPM1                  771.478409892293
SCYL3                 347.772747766408
C1orf112              91.1194972313732
CFH                   12141.6730060805

```

Tools for working with tidy data



The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. ---[tidyverse.org](https://www.tidyverse.org/) (<https://www.tidyverse.org/>)

The core packages within tidyverse include `dplyr`, `ggplot2`, `forcats`, `tibble`, `readr`, `stringr`, `tidyr`, and `purrr`, some of which we will use in this lesson and future lessons.

Load the core tidyverse packages

The tidyverse includes a collection of packages. To use these packages, we need to load the packages using the `library()` function.

```
library(tidyverse)
```

Load the data

To explore tidyverse functionality, let's read in some data and take a look.

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

We've already learned `str()`, but let's check out the tidyverse equivalent, `glimpse()`.

```
str(dexp, give.attr=FALSE)
```

```
tibble [15,926 × 10] (S3: tbl_df/tbl/data.frame)
 $ feature      : chr [1:15926] "ENSG00000000003" "ENSG00000000419" "ENSG00000000419" ...
 $ albut       : Factor w/ 1 level "untrt": 1 1 1 1 1 1 1 1 1 1 ...
 $ transcript   : chr [1:15926] "TSPAN6" "DPM1" "SCYL3" "C1orf112" ...
 $ ref_genome  : chr [1:15926] "hg38" "hg38" "hg38" "hg38" ...
 $ .abundant   : logi [1:15926] TRUE TRUE TRUE TRUE TRUE TRUE ...
 $ logFC       : num [1:15926] -0.3901 0.1978 0.0292 -0.1244 0.4173 ...
 $ logCPM     : num [1:15926] 5.06 4.61 3.48 1.47 8.09 ...
 $ F           : num [1:15926] 32.8495 6.9035 0.0969 0.3772 29.339 ...
 $ PValue     : num [1:15926] 0.000312 0.028062 0.762913 0.554696 0.000312 ...
 $ FDR        : num [1:15926] 0.00283 0.07701 0.84425 0.68233 0.00376
```

```
glimpse(dexp)
```

```
Rows: 15,926
Columns: 10
 $ feature      <chr> "ENSG00000000003", "ENSG00000000419", "ENSG00000000419", ...
 $ albut       <fct> untrt, untrt, untrt, untrt, untrt, untrt, untrt, untrt, untrt, untrt, ...
 $ transcript   <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH", "FUC1", "FUC2", "FUC3", ...
 $ ref_genome  <chr> "hg38", "hg38", "hg38", "hg38", "hg38", "hg38", "hg38", "hg38", "hg38", ...
 $ .abundant   <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, ...
 $ logFC       <dbl> -0.390100222, 0.197802179, 0.029160865, -0.124382031, 0.417300000, ...
 $ logCPM     <dbl> 5.059704, 4.611483, 3.482462, 1.473375, 8.089146, 4.611483, 3.482462, ...
 $ F          <dbl> 3.284948e+01, 6.903534e+00, 9.685073e-02, 3.772134e-02, 29.339000000, ...
 $ PValue     <dbl> 0.0003117656, 0.0280616149, 0.7629129276, 0.5546960000, 0.0003117656, ...
 $ FDR        <dbl> 0.002831504, 0.077013489, 0.844247837, 0.682326610, 0.003760000, ...
```

We can see that `glimpse()` is a little more succinct and clean and that `str()` will show attributes. These attributes were ignored above using `give.attr=FALSE` to get around package dependencies.

sscaled

We will also use a subset version of `scaled_counts` that includes the columns "sample", "cell", "dex", "transcript", "counts", and "counts_scaled".

```
scaled_counts[
  c("sample", "cell", "dex", "transcript", "counts", "counts_scaled")]
```

We can use import functions from the tidyverse to load `sscaled`, which are a bit more efficient and create a data frame like object, known as a tibble.

```
#import sscaled
sscaled<-read_delim("data/sscaled.txt")
```

```
Rows: 127408 Columns: 6
— Column specification —————
Delimiter: "\t"
chr (3): cell, dex, transcript
dbl (3): sample, counts, counts_scaled
```

```
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
```

sscaled

```
# A tibble: 127,408 × 6
  sample cell dex transcript counts counts_scaled
  <dbl> <chr> <chr> <chr> <dbl> <dbl>
1 508 N61311 untreated TSPAN6 679 961.
2 508 N61311 untreated DPM1 467 661.
3 508 N61311 untreated SCYL3 260 368.
4 508 N61311 untreated Clorf112 60 84.9
5 508 N61311 untreated CFH 3251 4601.
6 508 N61311 untreated FUCA2 1433 2028.
7 508 N61311 untreated GCLC 519 734.
8 508 N61311 untreated NFYA 394 558.
9 508 N61311 untreated STPG1 172 243.
```

```
10      508 N61311 untreated NIPAL3      2112      2989.
# i 127,398 more rows
```

Notice the tibble output. **Tibbles** easily show you the data types in each column and limit output to the screen (i.e., first 10 rows; only the columns that fit). They also do not force syntactic variable names.

Subsetting data frames with base R

Before diving into subsetting with `dplyr`, let's take a step back and learn to subset with base R. **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 df structure is returned)

```
scaled_counts[2,4] #Returns the value in the 4th column and 2nd row
scaled_counts[2, ] #Returns a df with row two
scaled_counts[-1, ] #Returns a df without row 1
scaled_counts[1:4,1] #returns a vector of rows 1-4 of column 1
#call names of columns directly
scaled_counts[1:10,c("sample","counts")]
#use comparison operators
scaled_counts[scaled_counts$sample == "508",]
```

What happens when we provide a single argument?

```
#notice the difference here
scaled_counts[,2] #returns column two
#treated similar to a matrix
#does not return a df if the output is a single column

scaled_counts[2] #returns column two
#treated similar to a list; maintains the df structure

#You can preserve the structure of the data frame while subsetting
```

```
# use drop = F
scaled_counts[,2,drop=F]
```

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 dataframe to only matched values.

For example, perhaps we have 4 transcripts that we are interested in exploring further. We can assign those transcripts to a vector.

```
keep_t<-c("CPD", "EXT1", "MCL1", "LASP1")
```

We can then see where those transcripts match transcripts in `sscaled$transcript`.

```
head(sscaled$transcript %in% keep_t)
```

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

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

```
#Let's filter our data to only include 4 transcripts of interest
interesting_trnsc<-sscaled[sscaled$transcript %in% keep_t,]
```

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 \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html\)](https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html)

Info

Subsetting including simplifying vs preserving can get confusing. [Here \(http://adv-r.had.co.nz/Subsetting.html\)](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 with tidyverse

While bracket notation 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.

The package `dplyr` is a fairly new (2014) package that tries to provide easy tools for the most common data manipulation tasks. It was built to work directly with data frames. The thinking behind it was largely inspired by the package `plyr` which has been in use for some time but suffered from being slow in some cases. `dplyr` addresses this by porting much of the computation to C++. An additional feature is the ability to work with data stored directly in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query returned. This addresses a common problem with R in that all operations are conducted in memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can have a database that is over 100s of GB, conduct queries on it directly and pull back just what you need for analysis in R. --- [datacarpentry.com \(https://datacarpentry.org/genomics-r-intro/05-dplyr.html\)](https://datacarpentry.org/genomics-r-intro/05-dplyr.html)

We do not need to load the `dplyr` package, as it is included in `library(tidyverse)`, which we have already installed and loaded. However, if you need to install and load on your local machine you would use the following:

```
install.packages("dplyr")
library("dplyr")
```

Subsetting with dplyr

We've seen how to select columns and rows using base R, but now let's look at a more intuitive way with functions (`select()` and `filter()`) from the tidyverse package `dplyr`.

Selecting columns

`select()` requires the data frame followed by the columns that we want to select or deselect as arguments.

```
#select the gene / transcript, logFC, and FDR corrected p-value
```

```
#first argument is the df followed by columns to select
dexp_s<-select(dexp, transcript, logFC, FDR)
```

We can also select all columns, leaving out ones that do not interest us using - or !. This is helpful if the columns to keep far outweigh those to exclude.

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

For readability we should move the transcript column to the front. We can select a range of columns using the `:`.

```
#you can reorder columns and call a range of columns using select().
df_exp<-select(df_exp, transcript:FDR,albut)
#Note: this also would have excluded the feature column
```

We can also include helper functions such as `starts_with()` and `ends_with()`. See more helper functions with `?select()`.

```
select(df_exp, 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
```

Test your learning

1. From the `interesting_trnsc` data frame select the following columns and save to an object: `sample`, `dex`, `transcript`, `counts_scaled`.

```
{{Sdet}}
```


Possible Solution{{Esum}}



```
interesting_trnsc_s<- select(interesting_trnsc, sample, dex, tra
{{Edet}}
```

2. From the `interesting_trnsc` data frame select all columns except `cell` and `counts`.

```
{{Sdet}}
```

Possible Solution{{Esum}}

```
interesting_trnsc_s2<-select(interesting_trnsc,!c(cell,counts))
{{Edet}}
```

Filtering by row

Now let's filter the rows based on a condition. Let's look at only the treated samples in `scaled_counts` using the function `filter()`. `filter()` requires the `df` as the first argument followed by the filtering conditions.

```
filter(sscaled, dex == "treated") #we've seen == notation before
```

We can also filter using `%in%`

```
#filter for two cell lines
f_sscaled<-filter(sscaled,cell %in% c("N061011", "N052611"))
#let's check that this worked
levels(factor(f_sscaled$cell))
```

```
[1] "N052611" "N061011"
```

```
#let's filter by keep_t from above
filter(f_sscaled,transcript %in% keep_t)
```

```
# A tibble: 16 × 6
  sample cell      dex      transcript counts counts_scaled
  <dbl> <chr>   <chr>   <chr>         <dbl>         <dbl>
1     512 N052611 untreated LASP1          7831          9647.
```

2	512	N052611	untreated	CPD	8270	10187.
3	512	N052611	untreated	MCL1	5170	6369.
4	512	N052611	untreated	EXT1	8503	10474.
5	513	N052611	treated	LASP1	5809	12411.
6	513	N052611	treated	CPD	7638	16318.
7	513	N052611	treated	MCL1	5153	11009.
8	513	N052611	treated	EXT1	2317	4950.
9	520	N061011	untreated	LASP1	5766	9082.
10	520	N061011	untreated	CPD	7067	11131.
11	520	N061011	untreated	MCL1	4410	6946.
12	520	N061011	untreated	EXT1	6925	10907.
13	521	N061011	treated	LASP1	7825	11884.
14	521	N061011	treated	CPD	10091	15325.
15	521	N061011	treated	MCL1	7338	11144.
16	521	N061011	treated	EXT1	3242	4923.

And we can filter using numeric columns. There are lots of options for filtering so explore the functionality a bit when you get a chance.

```
#get only results from counts greater than or equal to 20k
#use head to get only the first handful of rows
head(filter(f_sscales, counts_scaled >= 20000))
```

```
# A tibble: 6 × 6
  sample cell    dex      transcript counts counts_scaled
  <dbl> <chr> <chr> <chr> <dbl> <dbl>
1     512 N052611 untreated CSDE1    19863    24468.
2     512 N052611 untreated MRC2     23978    29537.
3     512 N052611 untreated DCN     422752   520769.
4     512 N052611 untreated VIM     37558    46266.
5     512 N052611 untreated CD44    25453    31354.
6     512 N052611 untreated VCL     17309    21322.
```

```
#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(df_exp, !is.na(transcript),
         abs(logFC) > 2, (PValue | FDR <= 0.01))
```

Test your learning



Filter the `interesting_trnsc` data frame to only include the following genes: MCL1 and EXT1.

{{Sdet}}

Possible Solution{{Esum}}

```
interesting_trnsc_f<-filter(interesting_trnsc, transcript %in% c("MCL1", "EXT1"))
interesting_trnsc_f<-filter(interesting_trnsc, transcript == "MCL1")
```

{{Edet}}

Acknowledgements

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/05-dplyr.html) (<https://datacarpentry.org/genomics-r-intro/05-dplyr.html>) and from a 2021 workshop entitled *Introduction to Tidy Transcriptomics* (https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html) by Maria Doyle and Stefano Mangiola. This lesson was also inspired by material from "Manipulating and analyzing data with dplyr", *Introduction to data analysis with R and Bioconductor* (<https://carpentries-incubator.github.io/bioc-intro/30-dplyr.html>).

Resources

1. R for Data Science (<https://r4ds.had.co.nz/index.html>)
2. Statistical Inference via Data Science: A ModernDive into R and the tidyverse (<https://moderndive.com/3-wrangling.html>)
3. dplyr cheatsheet
4. tidy cheatsheet
5. Other cheatsheets (<https://www.rstudio.com/resources/cheatsheets/>)

Data Frames and Data Wrangling (Part 2)

In this lesson, attendees will learn how to transform, summarize, and reshape data using functions from the tidyverse.

Learning Objectives

Continue to wrangle data using tidyverse functionality. To this end, you should understand:

1. how to use common dplyr functions (e.g., `group_by()`, `arrange()`, `mutate()`, and `summarize()`).
2. how to employ the pipe (`|>`) operator to link functions.
3. how to perform more complicated wrangling using the split, apply, combine concept.
4. how to tidy (reshape) data using `tidyr`.

Load the tidyverse

```
library(tidyverse)
```

Remember that this loads the core tidyverse packages. There are other packages that you may be interested in; see [here \(https://www.tidyverse.org/packages/\)](https://www.tidyverse.org/packages/).

Re-load the data

We will continue working with the `airway` data for this lesson. Let's import the data.

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

sscaled <- select(scaled_counts, sample,
  cell, dex, transcript, counts, counts_scaled)

dexp <- read_delim("./data/diffexp_results_edger_airways.txt")
```

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.

Running code one step at a time

```
#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
tspanDpm<- filter(dexp_s, transcript == "TSPAN6" | transcript=="DPM1")

#Print
tspanDpm
```

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

Or we could nest a function within a function.

Nesting code

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

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

But, these affect code readability and clutter our work space, making it difficult to follow what we or someone else did.

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*.

The `magrittr` pipe (`%>%`)

Prior to R version 4.1.0, the native R pipe did NOT exist. Use of the pipe came from the `magrittr` package, which is a dependency of the `tidyverse`. While the native pipe (`|>`) and the `magrittr` pipe (`%>%`) are fairly similar. They are not identical. You can read more about the differences [here \(https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/\)](https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/).

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

Let's see how this works

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

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

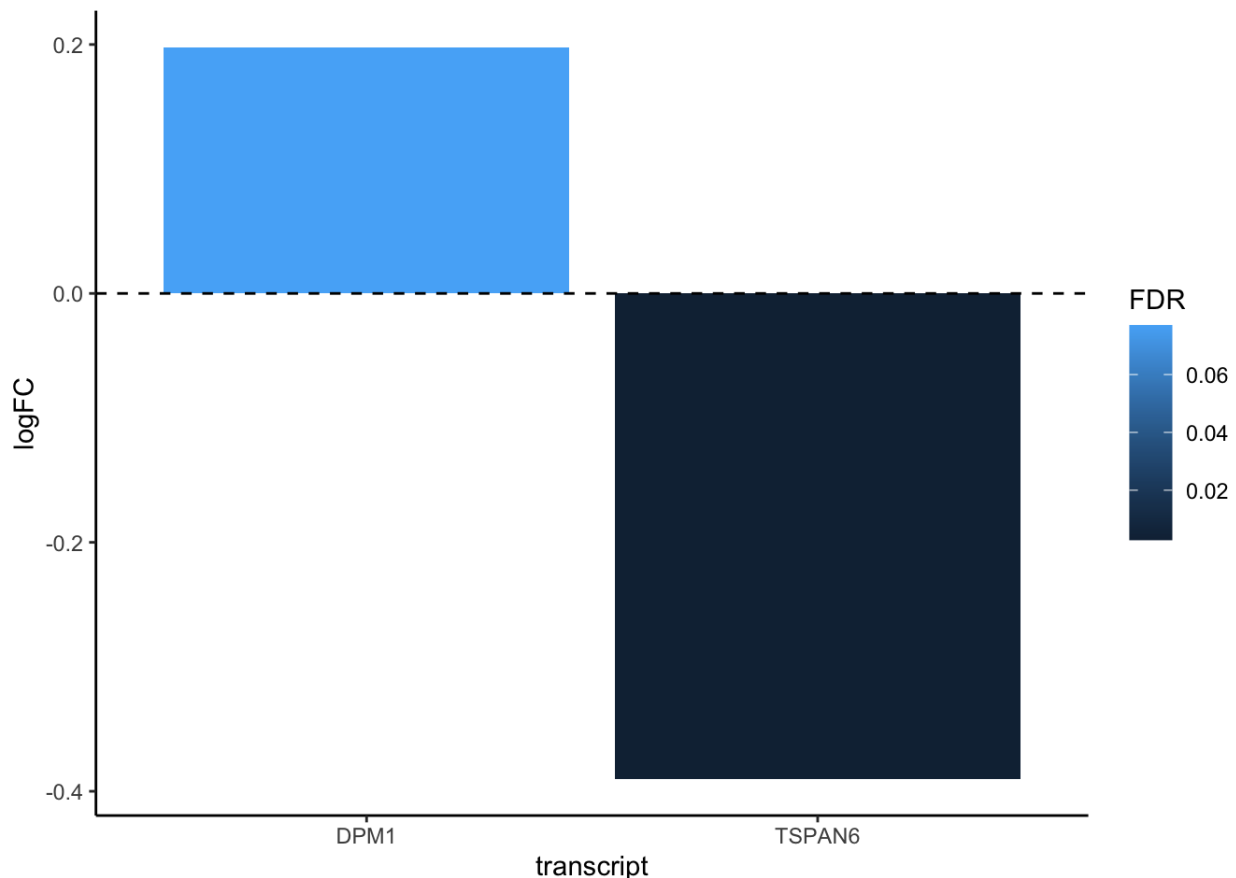
Notice that the data argument has been dropped from `select()` and `filter()`. This is because the pipe passes the object from the left to the right.

Important

The `|>` must be at the end of each line.

We can pipe from the beginning to the end.

```
readRDS("./data/diffexp_results_edger_airways.rds") |> #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")
```



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 \(https://datacarpentry.org/genomics-r-intro/05-dplyr.html\)](https://datacarpentry.org/genomics-r-intro/05-dplyr.html)

Test your learning

Using what you have learned about `select()` and `filter()`, use the pipe (`|>`) to create a subset data frame from `scaled_counts` that only includes the columns 'sample', 'cell', 'dex', 'transcript', and 'counts_scaled' and only rows that include the treatment "untrt" and the transcripts "ACTN1" and "ANAPC4"?

```
{{Sdet}}
```

Possible Solution{{Esum}}



```
scaled_counts |> select(sample,cell,dex,transcript,counts_scaled) |>
  filter(dex=="untrt", transcript %in% c("ACTN1","ANAPC4"))
```

{{Edet}}

Mutate

Another useful data manipulation function from `dplyr` is `mutate()`. `mutate()` allows you to create a new variable from existing variables. Perhaps you want to know the ratio of two columns or convert the units of a variable. That can be done with `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 \(https://dplyr.tidyverse.org/reference/mutate.html\)](https://dplyr.tidyverse.org/reference/mutate.html)

Create a new column using existing columns

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"])
```

```
# A tibble: 6 × 1
  Significant
  <lgl>
1 FALSE
2 FALSE
3 FALSE
4 FALSE
5 FALSE
6 FALSE
```

The conditional evaluates to a logical vector, containing `TRUE` or `FALSE` values.

Coerce variables

We can also use `mutate` to coerce variables.

To mutate across multiple columns, we need to use the function `across()` with the select helper `where()`.

```
#view sscaled
glimpse(sscaled)
```

```
Rows: 127,408
Columns: 6
$ sample      <dbl> 508, 508, 508, 508, 508, 508, 508, 508, 508, 50
$ cell        <chr> "N61311", "N61311", "N61311", "N61311", "N61311
$ dex         <chr> "untrt", "untrt", "untrt", "untrt", "untrt", "u
$ transcript  <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH", '
$ counts      <dbl> 679, 467, 260, 60, 3251, 1433, 519, 394, 172, ;
$ counts_scaled <dbl> 960.88642, 660.87475, 367.93883, 84.90896, 460(
```

```
#use mutate with across and select helpers
ex_coerce<-sscaled |> mutate(across(where(is.character),as.factor))
glimpse(ex_coerce)
```

```
Rows: 127,408
Columns: 6
$ sample      <dbl> 508, 508, 508, 508, 508, 508, 508, 508, 508, 50
$ cell        <fct> N61311, N61311, N61311, N61311, N61311, N61311
$ dex         <fct> untrt, untrt, untrt, untrt, untrt, untrt, untrt
$ transcript  <fct> TSPAN6, DPM1, SCYL3, C1orf112, CFH, FUCA2, GCLC
$ counts      <dbl> 679, 467, 260, 60, 3251, 1433, 519, 394, 172, ;
$ counts_scaled <dbl> 960.88642, 660.87475, 367.93883, 84.90896, 460(
```

Note

`across()` has superseded the use of `mutate_if`, `_at`, `_all`. For more information on `across()` see [this reference article \(https://dplyr.tidyverse.org/reference/across.html\)](https://dplyr.tidyverse.org/reference/across.html).

More examples

Check out more examples using `mutate()` [here \(https://dplyr.tidyverse.org/reference/mutate.html#ref-examples\)](https://dplyr.tidyverse.org/reference/mutate.html#ref-examples)

Test your learning ?

Using `mutate` apply a base 10 logarithmic transformation to the `counts_scaled` column of `sscaled`. Save the resulting data frame to an object called `log10counts`. Hint: see the function `log10()`.

```
{{Sdet}}
```

Possible Solution{{Esum}}

```
log10counts<-sscaled |> mutate(logCounts=log10(counts_scaled))
```

```
{{Edet}}
```

Arrange, group_by, summarize

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

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

Let's get the median `counts_scaled` by transcript within a treatment. When you reduce the size of a data set through a calculation, you need to use `summarize()`.

```
scaled_counts |> #Call the data
  group_by(dex,transcript) |> # group_by treatment and transcript
  #(transcript nested within treatment)
  summarize(median_counts=median(counts_scaled))
```

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

```
# A tibble: 29,152 × 3
# Groups:   dex [2]
  dex transcript median_counts
<chr> <chr>          <dbl>
1 trt  A1BG-AS1          80.2
2 trt  A2M                37821.
3 trt  A2M-AS1           24.2
```

```

4 trt    A4GALT          2043.
5 trt    AAAS           1086.
6 trt    AACS            481.
7 trt    AADAT           154.
8 trt    AAGAB           984.
9 trt    AAK1            399.
10 trt   AAMDC           181.
# i 29,142 more rows

```

The output is a grouped data frame.

Note

`group_by()` can also be used with `mutate()`.

Using `arrange()`

Now, if we want the top five transcripts with the greatest median scaled counts by treatment, we need to organize our data frame and then return the top rows. We can use `arrange()` to arrange our data frame by `median_counts`. If we want to arrange from highest to lowest value, we will additionally need to use `desc()`. The `.by_group` allows us to arrange by median counts within a grouping. By including `slice_head()` we can return the top five values by group.

```

scaled_counts |> #Call the data
  group_by(dex,transcript) |> # group_by treatment and transcript
  #(transcript nested within treatment)
  summarize(median_counts=median(counts_scaled)) |> #for each group
  #calculate the median value of scaled counts
  arrange(desc(median_counts),.by_group = TRUE) |>
  #arrange in descending order
  slice_head(n=5) #return the top 5 values for each group

```

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

```

# A tibble: 10 × 3
# Groups:   dex [2]
  dex transcript median_counts
  <chr> <chr>          <dbl>
1 trt   FN1             486430.
2 trt   DCN             389306.

```

```

3 trt    MT-C01      369456.
4 trt    EEF1A1      346869.
5 trt    QSOX1       284100.
6 untrt  FN1        456360.
7 untrt  DCN        439781.
8 untrt  EEF1A1      404269.
9 untrt  MT-C01      346974.
10 untrt  COL1A2     331816.

```

The slice functions

We could have skipped `arrange()` and used `slice_max()`. `slice_max()` returns the rows with the greatest values from each group.

```

scaled_counts |>
  group_by(dex,transcript) |>
  summarize(median_counts=median(counts_scaled)) |>
  slice_max(median_counts, n=5) #notice use of slice_max

```

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

```

# A tibble: 10 × 3
# Groups:   dex [2]
  dex transcript median_counts
  <chr> <chr>          <dbl>
1 trt   FN1            486430.
2 trt   DCN            389306.
3 trt   MT-C01         369456.
4 trt   EEF1A1         346869.
5 trt   QSOX1          284100.
6 untrt FN1        456360.
7 untrt DCN        439781.
8 untrt EEF1A1      404269.
9 untrt MT-C01     346974.
10 untrt COL1A2    331816.

```

The other slice functions include:

`slice_head(n = 1)` takes the first row from each group.

`slice_tail(n = 1)` takes the last row in each group.

`slice_min(x, n = 1)` takes the row with the smallest value of column x.

`slice_sample(n = 1)` takes one random row. --- R4DS (https://r4ds.hadley.nz/data-transform.html#the-slice_-functions).

Sample sizes (counts and tallies) and missing data

How many rows per sample are in the `scaled_counts` data frame?

```
scaled_counts |>
  group_by(dex, sample) |>
  summarize(n=n()) #there are multiple functions that can be used here
```

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

```
# A tibble: 8 × 3
# Groups:   dex [2]
  dex  sample     n
  <chr> <dbl> <int>
1 trt     509 15926
2 trt     513 15926
3 trt     517 15926
4 trt     521 15926
5 untrt   508 15926
6 untrt   512 15926
7 untrt   516 15926
8 untrt   520 15926
```

```
#See count(); can also use tally()
scaled_counts |>
  count(dex, sample)
```

```
# A tibble: 8 × 3
  dex  sample     n
  <chr> <dbl> <int>
1 trt     509 15926
2 trt     513 15926
3 trt     517 15926
4 trt     521 15926
5 untrt   508 15926
6 untrt   512 15926
```

```
7 untrt      516 15926
8 untrt      520 15926
```

Note

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 \(https://datacarpentry.org/genomics-r-intro/05-dplyr.html\)](https://datacarpentry.org/genomics-r-intro/05-dplyr.html)

Let's see this in practice

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

#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.
fun_df<-fun_df |> mutate(counts=ifelse(counts<100,NA,count))

fun_df #view
```

```
  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
```

```
#We should get NAs returned for some of our genes
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
```

```
#Now let's 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
```

Test your learning

Create a data frame summarizing the mean counts_scaled by sample from the scaled_counts data frame.

```
{{Sdet}}
```

Possible Solution{{Esum}}

```
scaled_counts |> group_by(sample) |>
  summarize(Mean_counts_scaled=mean(counts_scaled))
```

```
{{Edet}}
```

Data Reshaping

Tidy data implies that we have one observation per row and one variable per column. This generally means data is in a long format. However, whether data is tidy or not will depend on what we consider a variable versus an observation, so wide data sets may also be tidy. Often if data is in a wide format, data related to the same measurement is distributed in different columns. This at times will mean the data looks more like a data matrix; though, it may not necessarily be a matrix.

In genomics, we often receive data in wide format. For example, you may be given RNAseq data from a colleague with the first column being sampleIDs and all additional columns being genes. The data frame itself holds count data.

For example:

```
# A tibble: 3 × 5
  sampleid      A      B      C      D
  <chr>      <dbl> <dbl> <dbl> <dbl>
1 A1          0    352   400   278
2 B1         214    256   381     0
3 C1          0      0    250   169
```

You may recognize these data. These are from our `fun_df`, which was originally in long format. Above we added a `sampleid` column and replaced the NAs with 0s and converted to wide format.

```
#Add sample id; replace NAs with 0s
fun_df <- data.frame(sampleid=rep(c("A1","B1","C1"),4),fun_df) |>
  mutate(counts= ifelse(is.na(counts), 0, counts))
fun_df
```

```
  sampleid genes counts
1      A1    A      0
2      B1    A    214
3      C1    A      0
4      A1    B    352
5      B1    B    256
6      C1    B      0
7      A1    C    400
8      B1    C    381
9      C1    C    250
```


10	A1	D	278
11	B1	D	0
12	C1	D	169

Pivot wider

We can convert to wide format using `pivot_wider()`, which takes three main arguments:

1. the data we are reshaping
2. the column that includes the new column names - `names_from`
3. the column that includes the values that will fill our new columns - `values_from`

```
fun_df_w <- fun_df |>
  pivot_wider(names_from = genes, values_from = counts)
fun_df_w
```

```
# A tibble: 3 × 5
  sampleid     A     B     C     D
  <chr>    <dbl> <dbl> <dbl> <dbl>
1 A1         0   352   400   278
2 B1       214   256   381     0
3 C1         0     0   250   169
```

This resembles a matrix. At times we may need to work with a matrix while at others we may need a data frame. We can coerce to a matrix by sending the sample ids to the row names using `column_to_rownames()` and `as.matrix()`. Remember, our `as.` functions are usually reserved for coercing.

Coerce to a matrix

```
fun_mat <- fun_df_w |> column_to_rownames("sampleid") |>
  as.matrix(rownames.force = TRUE)
fun_mat
```

```
      A  B  C  D
A1   0 352 400 278
B1 214 256 381  0
C1   0  0 250 169
```

Now, we can apply functions that require data matrices.

Pivot longer

We can convert back to long / tidy format using `pivot_longer()`.

`pivot_longer()` takes four main arguments:

1. the data we want to transform
2. the columns we want to pivot longer
3. the column we want to create to store the column names - `names_to`
4. the column we want to create to store the values associated with the column names - `quantity`

```
fun_df_w |>
  pivot_longer(where(is.numeric), names_to="genes", values_to="counts")
```

```
# A tibble: 12 × 3
  sampleid genes counts
  <chr>    <chr>  <dbl>
1 A1      A         0
2 A1      B        352
3 A1      C        400
4 A1      D        278
5 B1      A        214
6 B1      B        256
7 B1      C        381
8 B1      D         0
9 C1      A         0
10 C1     B         0
11 C1     C        250
12 C1     D        169
```

Reshaping for plotting

There are other reasons you may be interested in using `pivot_wider` or `pivot_longer`. In my experience, most uses revolve around plotting criteria. For example, you may want to plot two different but related measurements on the same plot. You could `pivot_longer` so that those two measurements are now in the same column, stored as a categorical variable.

Let's see how this might work with our `scaled_counts` data. We want to plot both "counts" and "counts_scaled" together in a density plot to understand the distribution of the data. Did scaling the counts improve the distribution?

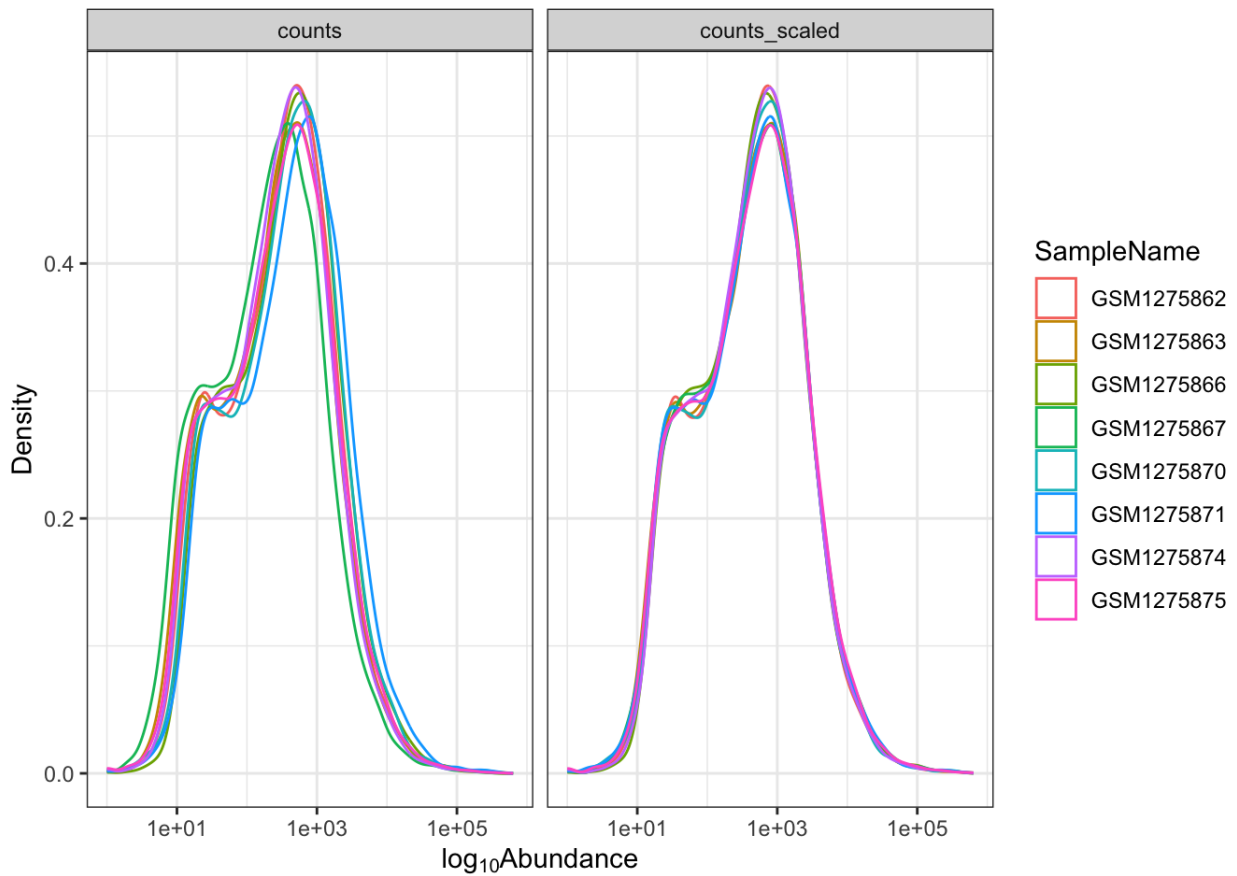
We can place "counts" and "counts_scaled" into a column named "source" and place their values in a column named "abundance".

```
#getting the data
counts_long<- scaled_counts |>
  #pivot
  pivot_longer(cols = c("counts", "counts_scaled"),
               names_to = "source", values_to = "abundance")
#View
counts_long |> select(source, abundance) |> head()
```

```
# A tibble: 6 × 2
  source      abundance
  <chr>      <dbl>
1 counts      679
2 counts_scaled 961.
3 counts      467
4 counts_scaled 661.
5 counts      260
6 counts_scaled 368.
```

Now let's create a density plot using ggplot2.

```
ggplot(data=counts_long, aes(x = abundance + 1, color = SampleName))
  geom_density() +
  facet_wrap(~source) +
  ggplot2::scale_x_log10() +
  theme_bw()+
  ylab("Density")+
  xlab(expression('log' [10]*'Abundance'))
```



Note

It's not important to understand the code here. This will make more sense in the next lesson.

Acknowledgements

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/05-dplyr.html) (<https://datacarpentry.org/genomics-r-intro/05-dplyr.html>) and from a 2021 workshop entitled *Introduction to Tidy Transcriptomics* (https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html) by Maria Doyle and Stefano Mangiola. This lesson was also inspired by material from "Manipulating and analyzing data with dplyr", *Introduction to data analysis with R and Bioconductor* (<https://carpentries-incubator.github.io/bioc-intro/30-dplyr.html>).

Resources

1. R for Data Science (<https://r4ds.had.co.nz/index.html>)
2. Statistical Inference via Data Science: A ModernDive into R and the tidyverse (<https://moderndive.com/3-wrangling.html>)
3. dplyr cheatsheet
4. tidyr cheatsheet

-
5. Other cheatsheets (<https://www.rstudio.com/resources/cheatsheets/>)

Data visualization with ggplot2

Objectives

To learn how to create publishable figures using the `ggplot2` package in R.

By the end of this lesson, learners should be able to create simple, pretty, and effective figures.

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.

Introducing ggplot2

`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 ————— tidy
## ✓ dplyr      1.1.3      ✓ readr      2.1.4
## ✓ forcats   1.0.0      ✓ stringr    1.5.0
## ✓ ggplot2   3.4.4      ✓ tibble     3.2.1
## ✓ lubridate 1.9.3      ✓ tidyr      1.3.0
```

```
## ✓ purrr      1.0.2
## — Conflicts ————— tidyverse_
## ✘ dplyr::filter() masks stats::filter()
## ✘ dplyr::lag()   masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to f
```

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
#We used this in lesson 2 so you may not need to reload
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,
## dbl (6): sample, counts, avgLength, TMM, multiplier, counts_scale
## lgl (1): .abundant
##
## i Use `spec()` to retrieve the full column specification for this
## i Specify the column types or set `show_col_types = FALSE` to quie
```

```
dexp<-read_delim("./data/diffexp_results_edger_airways.txt")
```

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

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

Let's see this template in practice.

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

```
#let's get some data
#we are only interested in transcript counts greater than 100
#read in the data
sc<-read_csv("./data/sc.csv")
```

```
## Rows: 8 Columns: 4
## — Column specification —————
## Delimiter: ","
## chr (2): dex, SampleName
## dbl (2): Num_transcripts, TotalCounts
##
## i Use `spec()` to retrieve the full column specification for this
## i Specify the column types or set `show_col_types = FALSE` to quiet
```

```
#let's view the data
sc
```

```
## # A tibble: 8 × 4
##   dex   SampleName Num_transcripts TotalCounts
##   <chr> <chr>           <dbl>         <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
```


These data include total transcript read counts summed by sample and the total number of transcripts recovered by sample that had at least 100 reads.

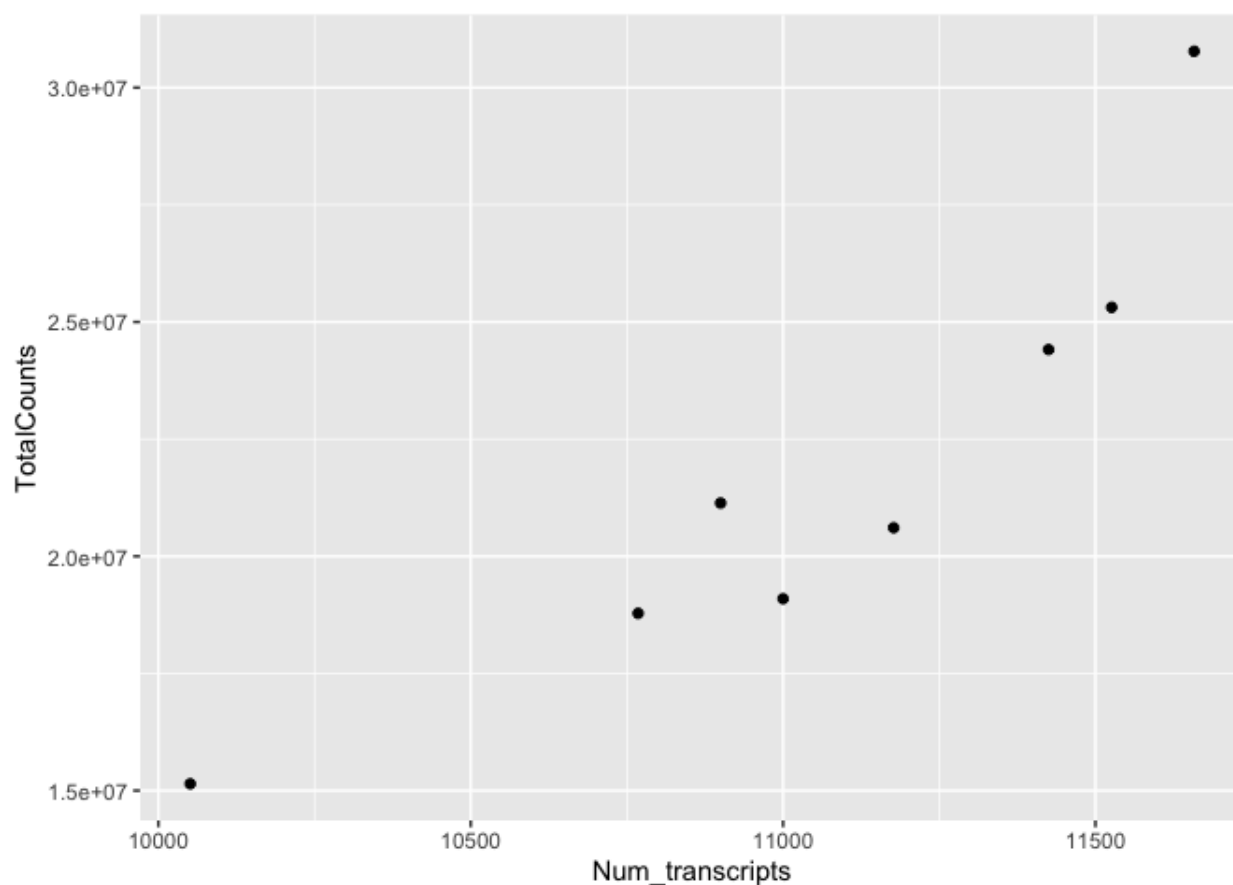
Note

These data can be generated using

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

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 transcripts per sample and the total transcripts recovered per sample. ggplot2 default parameters are great for exploratory data analysis. But, with only a few tweaks, we can make some beautiful, publishable figures.

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 this needs to actually be provided at the initialization phase, which creates the coordinate system from which we build our plot. But, typically, you should at least call the data at this point.

The data we called was from the data frame `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.

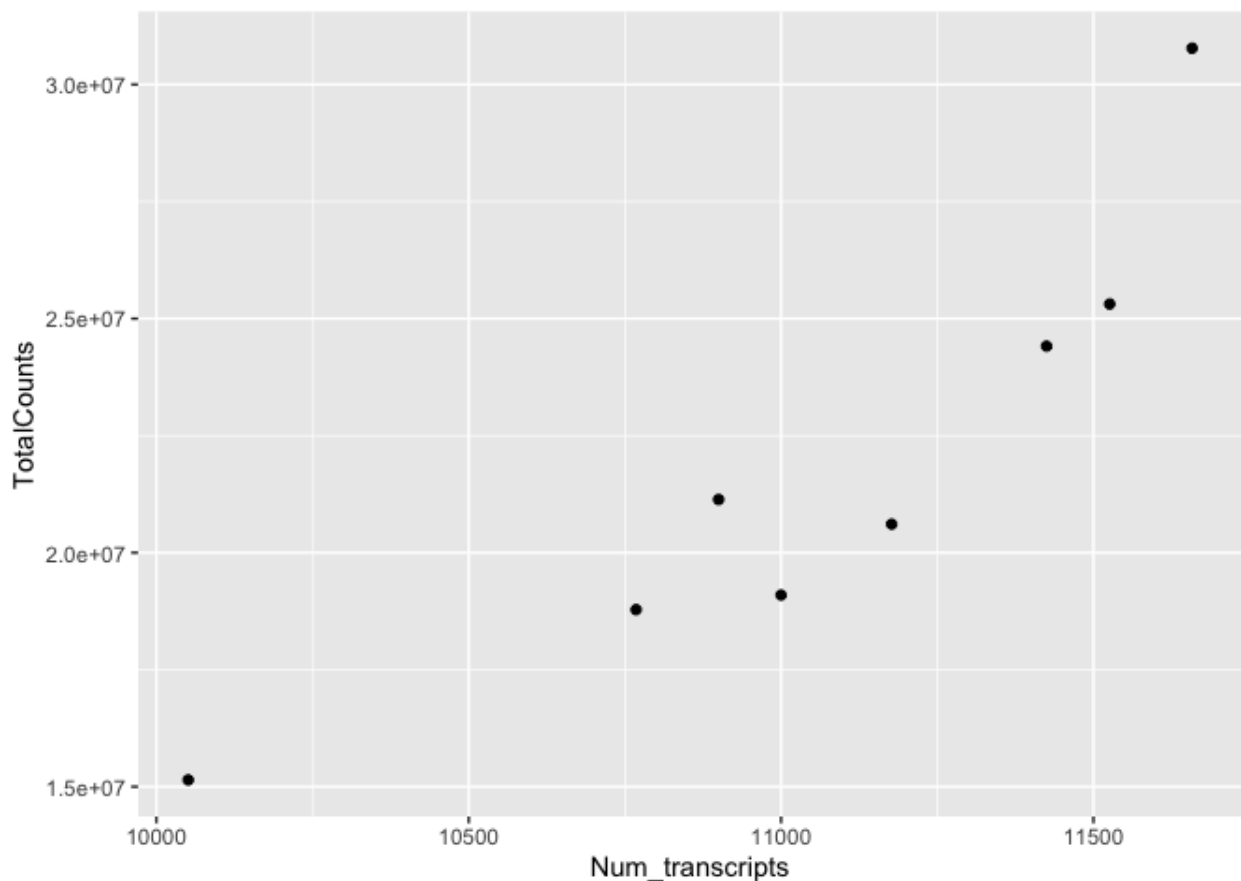
Let's break down the individual components of the code.

```
#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 \(https://r4ds.had.co.nz/data-visualisation.html#geometric-objects\)](https://r4ds.had.co.nz/data-visualisation.html#geometric-objects)

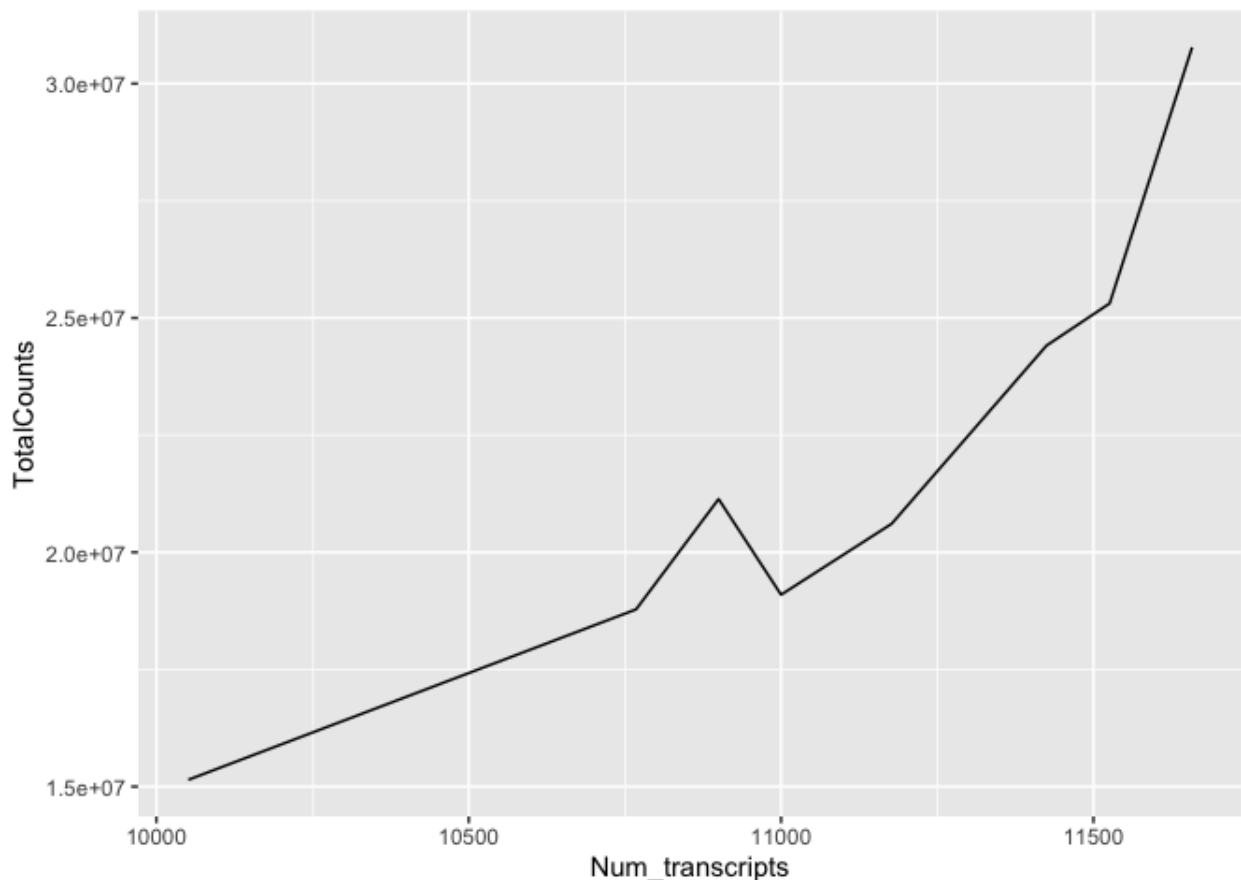
There are multiple geom functions that change the basic plot type or the plot representation. We can create scatter plots (`geom_point()`), line plots (`geom_line()`, `geom_path()`), bar plots (`geom_bar()`, `geom_col()`), line modeled to fitted data (`geom_smooth()`), heat maps (`geom_tile()`), geographic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

ggplot2 provides over 40 geoms, and extension packages provide even more (see <https://exts.ggplot2.tidyverse.org/gallery/> (<https://exts.ggplot2.tidyverse.org/gallery/>) for a sampling). The best way to get a comprehensive overview is the ggplot2 cheatsheet, which you can find at <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.

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))
```



Mapping and aesthetics (`aes()`)

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

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

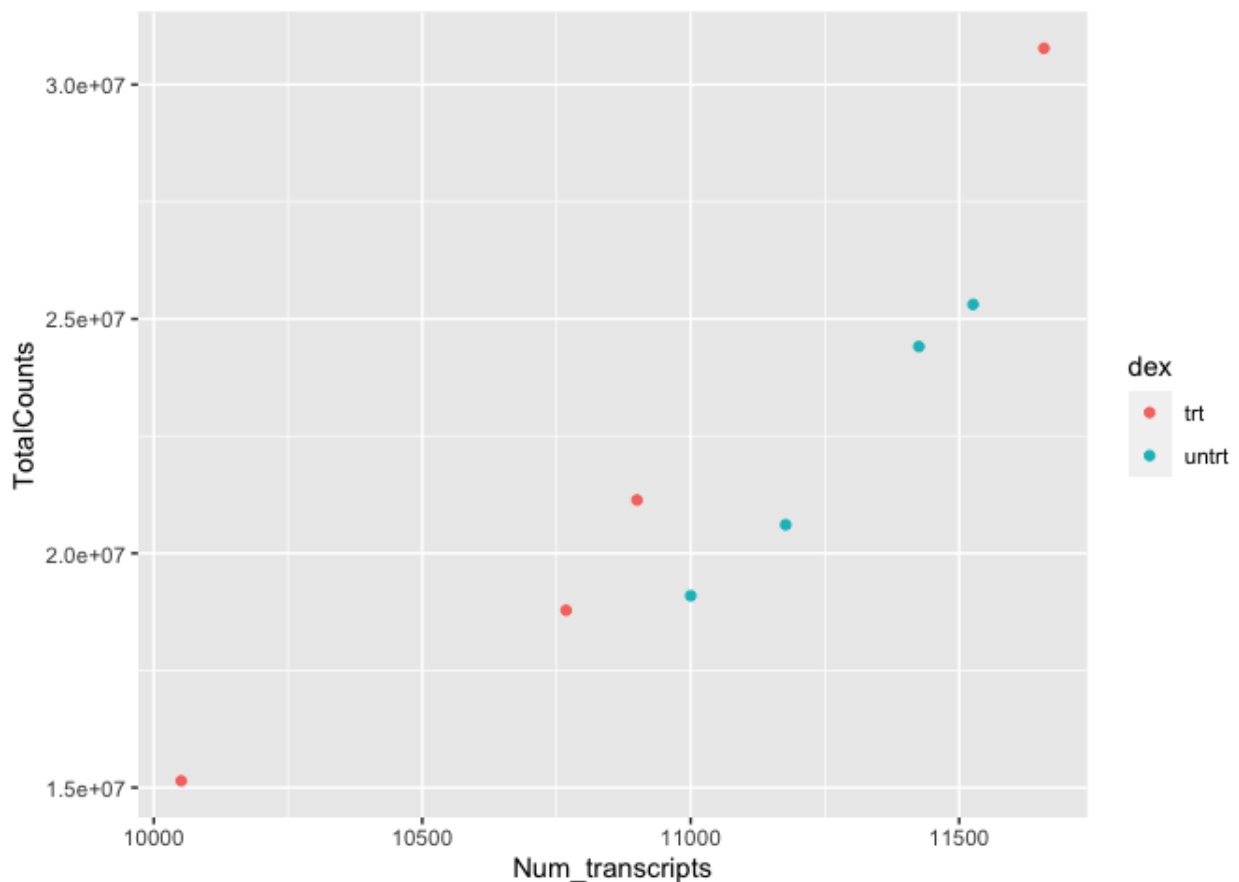
Mapping aesthetics include some of the following:

1. the x and y data arguments
2. shapes
3. color
4. fill
5. size
6. linetype
7. alpha

This is not an all encompassing list.

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

```
#adding the color argument to our mapping aesthetic
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,color=dex))
```

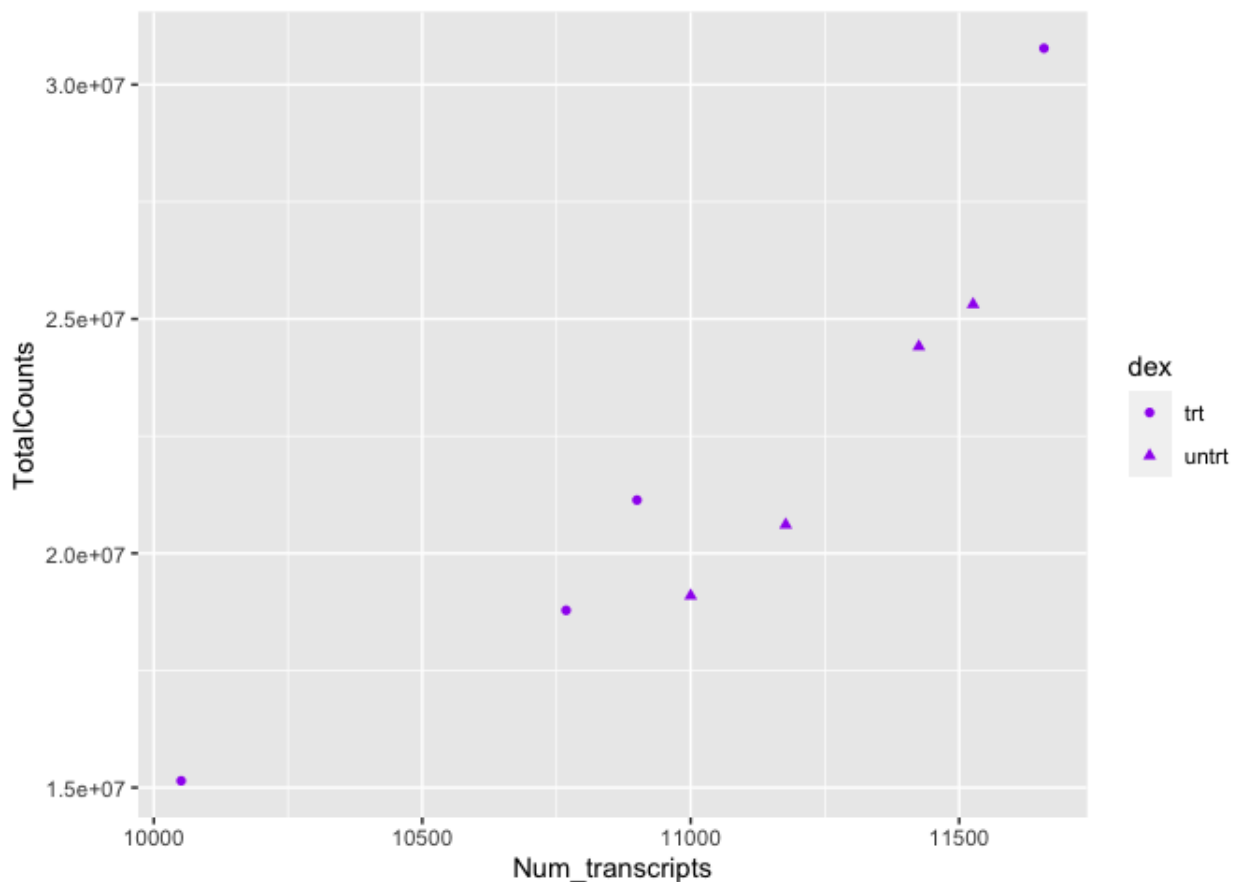


There is potentially a relationship. ASM cells treated with dexamethasone in general have lower total numbers of transcripts and lower total counts.

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. 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")
```

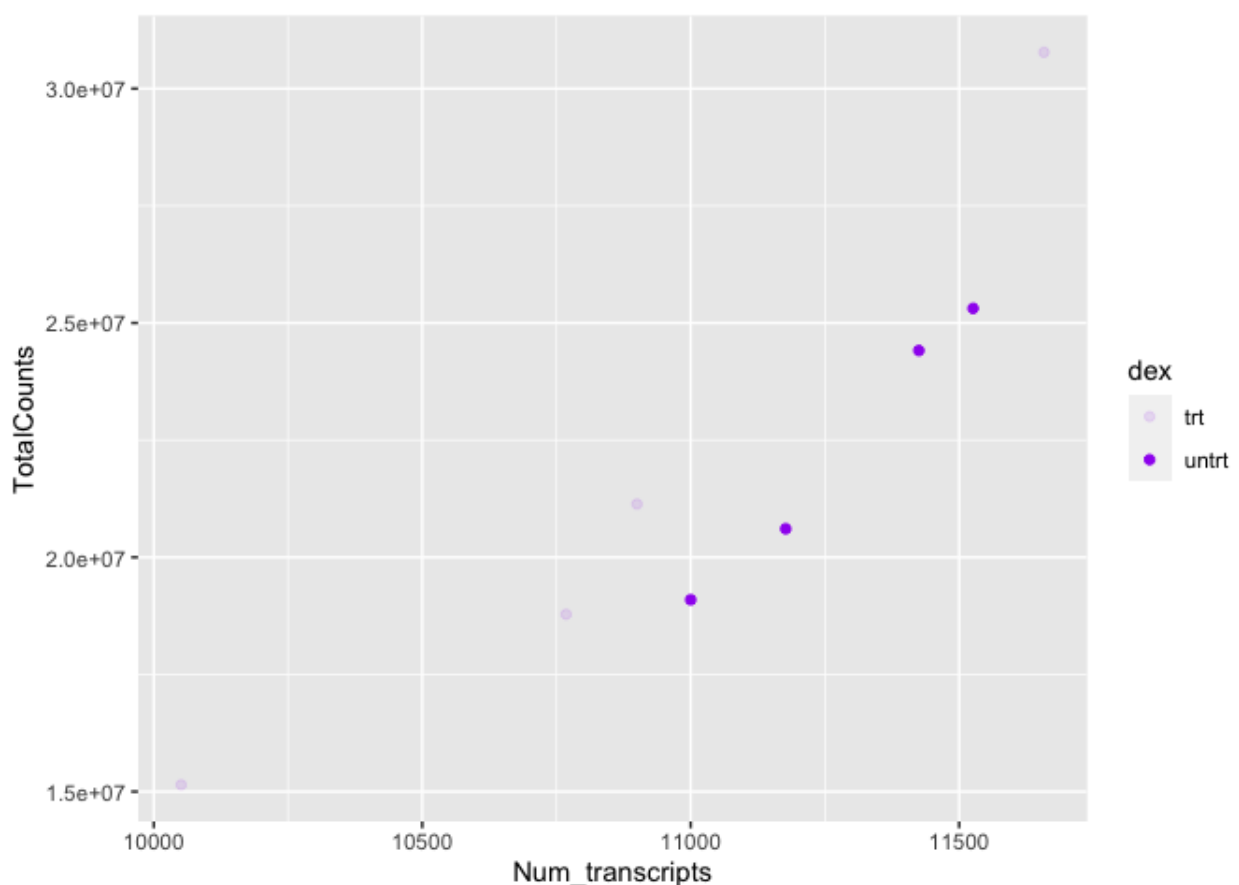


We can also see from this that 'dex' could be mapped to other aesthetics. In the above example, we see it mapped to shape rather than color. By default, ggplot2 will only map six shapes at a time, and if your number of categories goes beyond 6, the remaining groups will go unmapped. This is by design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

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

```
#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 explore a little with your plots.

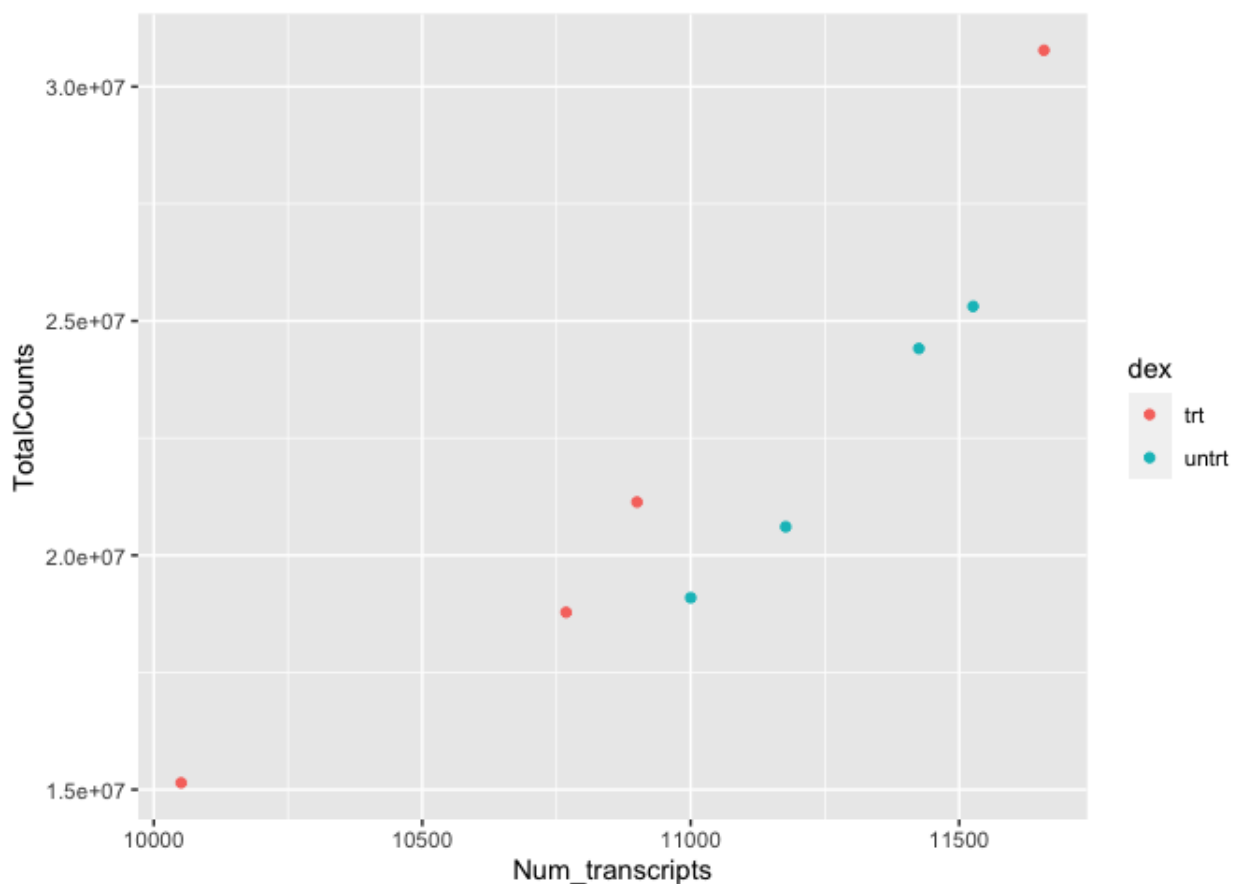
Other things to note:

The assignment of color, shape, or alpha to our variable was automatic, with a unique aesthetic level representing each category (i.e., 'trt', 'untrt') within our variable. You will also notice that ggplot2 automatically created a legend to explain the levels of the aesthetic mapped. We can change aesthetic parameters - what colors are used, for example - by adding additional layers to the plot. We will be adding layers throughout the tutorial.

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.

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

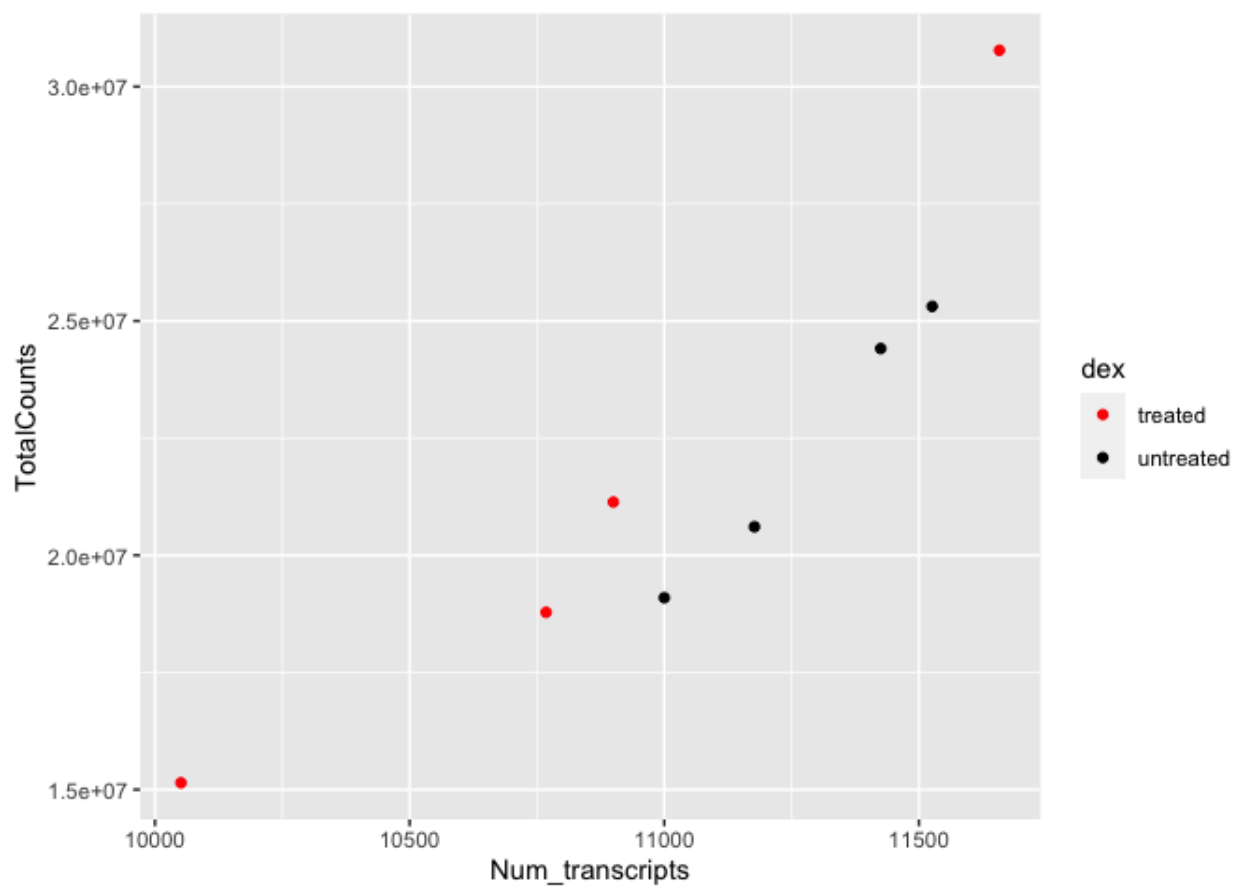


We can add additional layers directly to our object. We will see how this works by defining some colors for our 'dex' variable.

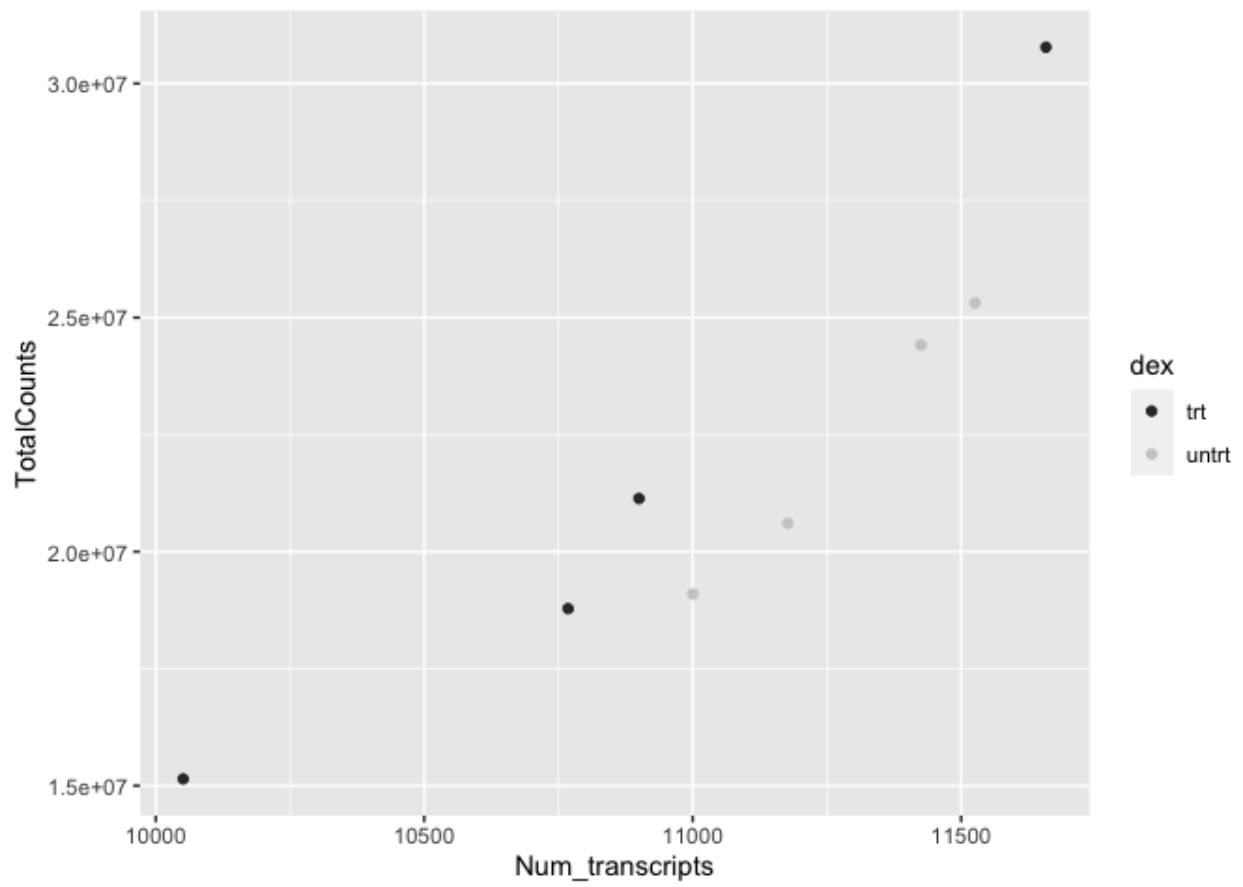
Colors

ggplot2 will automatically assign colors to the categories in our data. Colors are assigned to the fill and color aesthetics in aes(). We can change the default colors by providing an additional layer to our figure. To change the color, we use the scale_color functions: scale_color_manual(), scale_color_brewer(), scale_color_grey(), etc. We can also change the name of the color labels in the legend using the labels argument of these functions

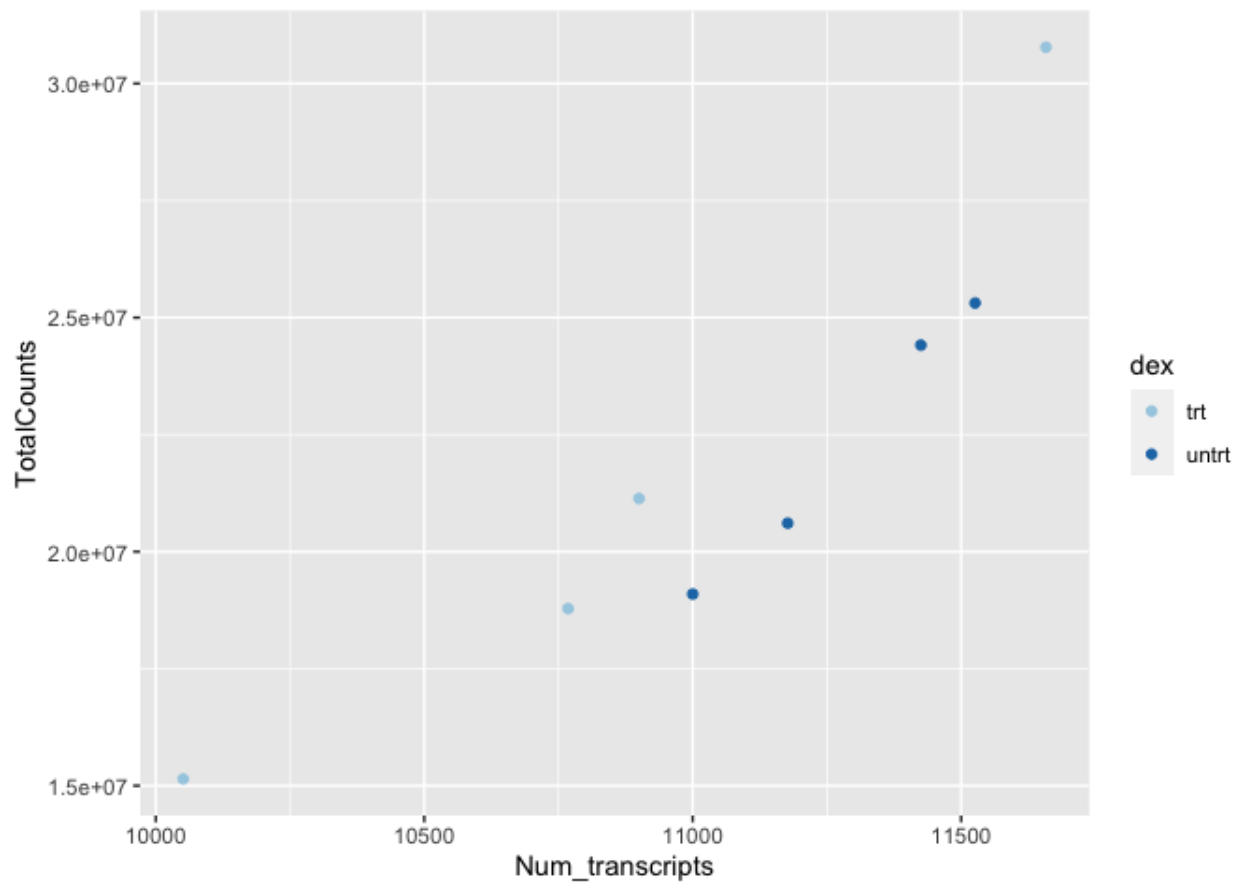

```
dot_plot +  
  scale_color_manual(values=c("red", "black"),  
                    labels=c('treated', 'untreated'))
```



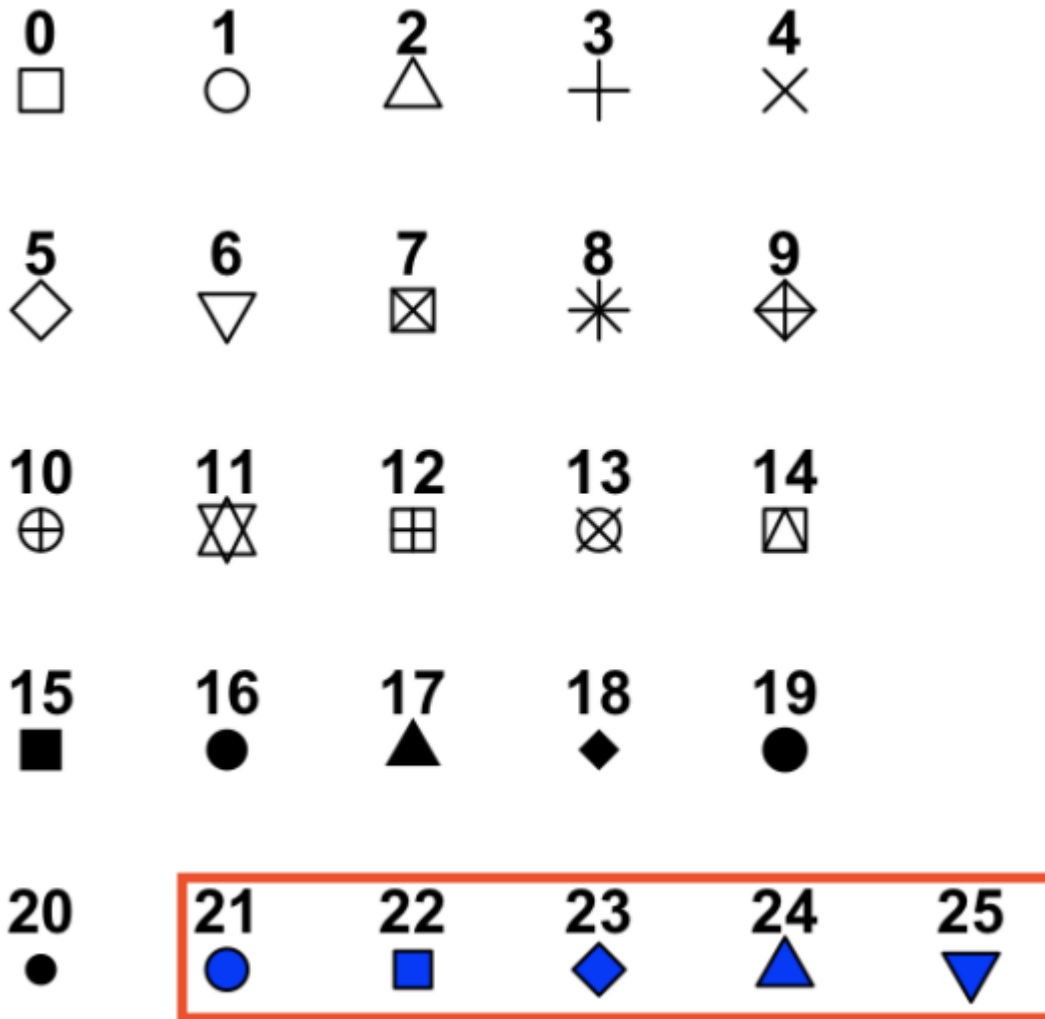
```
dot_plot +  
  scale_color_grey()
```



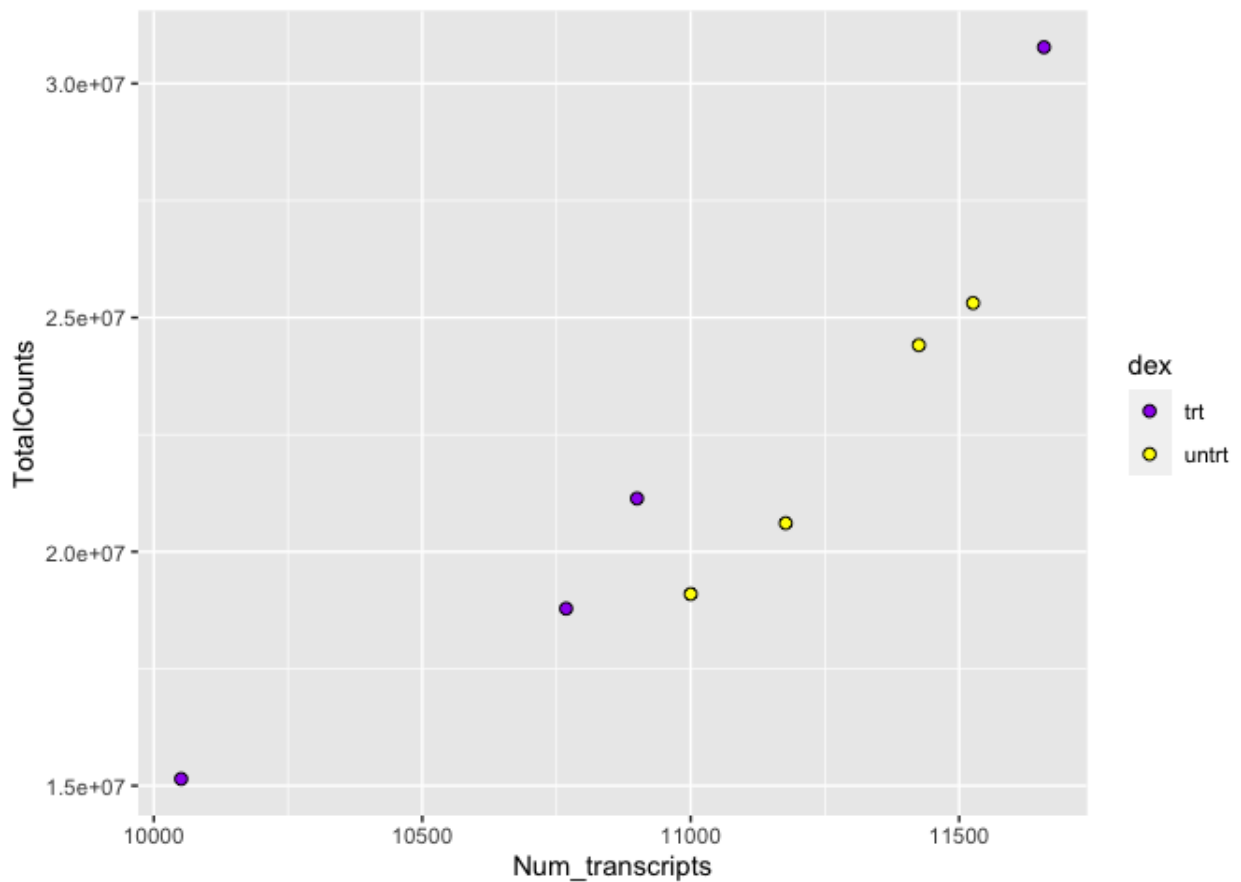
```
dot_plot +  
  scale_color_brewer(palette = "Paired")
```



Similarly, if we want to change the fill, we would use the `scale_fill` options. To apply `scale_fill` to shape, we will have to alter the shapes, as only some shapes take a fill argument.



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



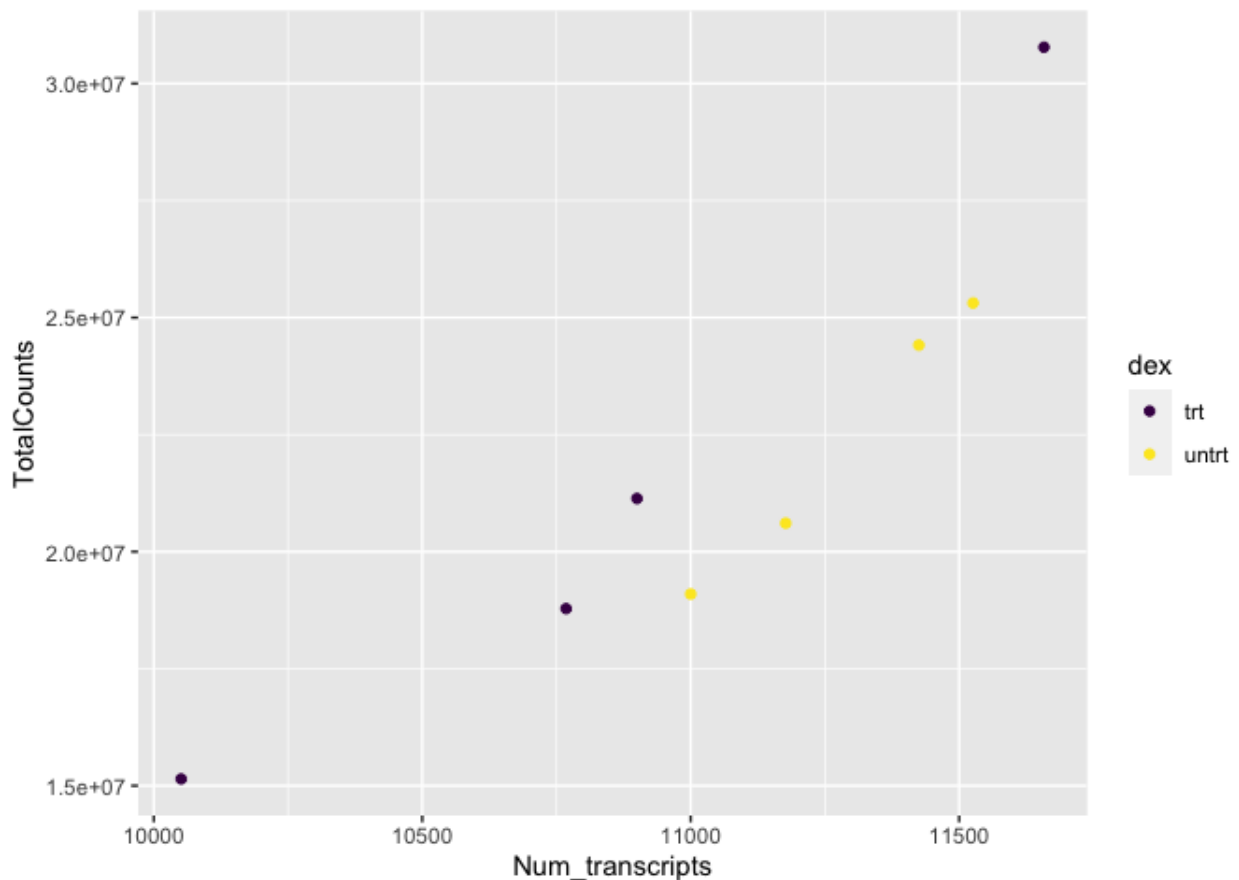
There are a number of ways to specify the color argument including by name, number, and hex code. [Here](https://www.r-graph-gallery.com/ggplot2-color.html) (<https://www.r-graph-gallery.com/ggplot2-color.html>) is a great resource from the [R Graph Gallery](https://www.r-graph-gallery.com/index.html)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

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

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

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

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



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

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).

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\)](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)
(<https://r-graph-gallery.com/density-plot.html>)

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

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

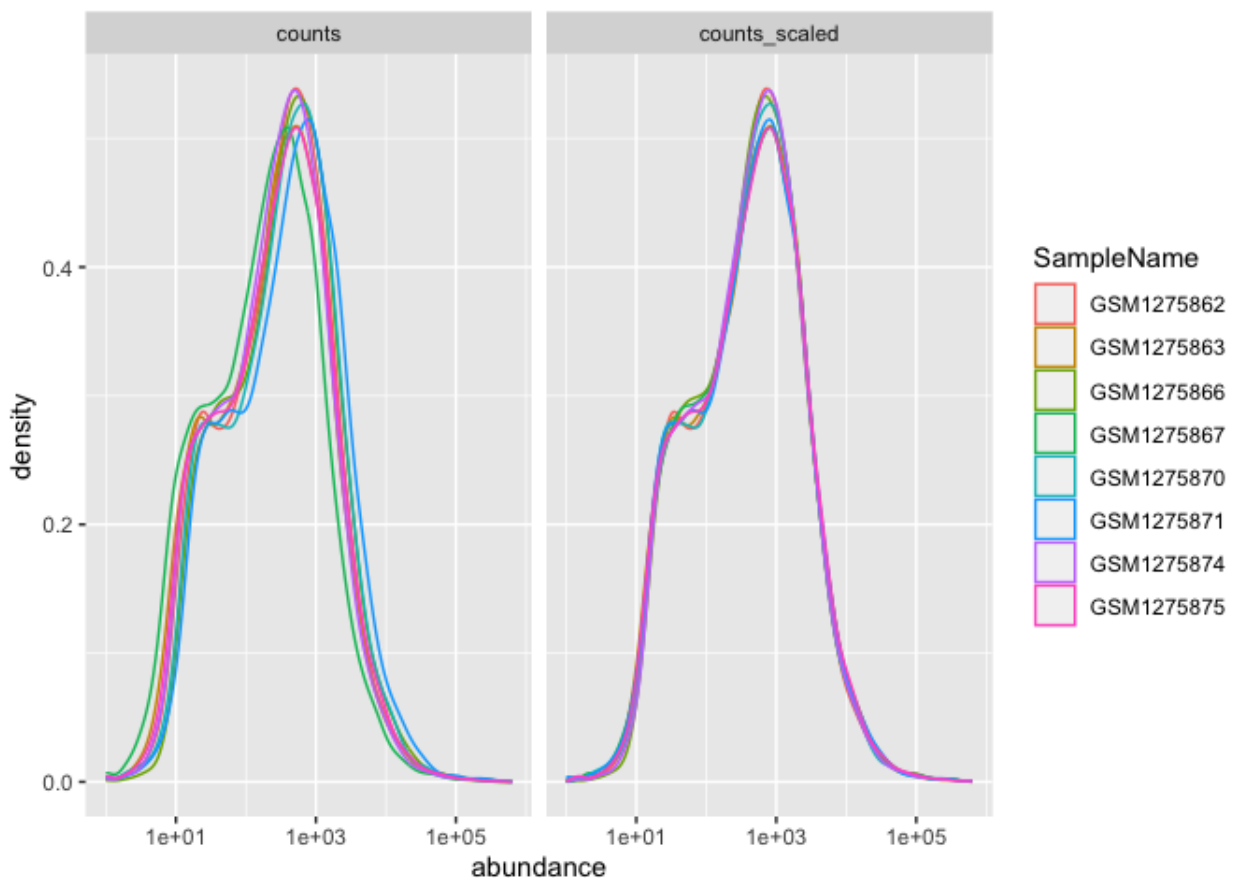
head(density_data)
```

```
##           feature sample SampleName   cell  dex albut      Run
## 1  ENSG00000000003     508  GSM1275862  N61311  untrt  untrt  SRR1039508
## 2  ENSG00000000003     508  GSM1275862  N61311  untrt  untrt  SRR1039508
## 3  ENSG000000000419     508  GSM1275862  N61311  untrt  untrt  SRR1039508
## 4  ENSG000000000419     508  GSM1275862  N61311  untrt  untrt  SRR1039508
## 5  ENSG000000000457     508  GSM1275862  N61311  untrt  untrt  SRR1039508
## 6  ENSG000000000457     508  GSM1275862  N61311  untrt  untrt  SRR1039508
##  Experiment      Sample      BioSample transcript ref_genome .abundanc
## 1  SRX384345  SRS508568  SAMN02422669      TSPAN6      hg38      TRI
## 2  SRX384345  SRS508568  SAMN02422669      TSPAN6      hg38      TRI
## 3  SRX384345  SRS508568  SAMN02422669      DPM1        hg38      TRI
## 4  SRX384345  SRS508568  SAMN02422669      DPM1        hg38      TRI
## 5  SRX384345  SRS508568  SAMN02422669      SCYL3       hg38      TRI
## 6  SRX384345  SRS508568  SAMN02422669      SCYL3       hg38      TRI
##  multiplier      source abundance
## 1  1.415149      counts  679.0000
## 2  1.415149 counts_scaled  960.8864
## 3  1.415149      counts  467.0000
## 4  1.415149 counts_scaled  660.8748
## 5  1.415149      counts  260.0000
## 6  1.415149 counts_scaled  367.9388
```

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

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

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

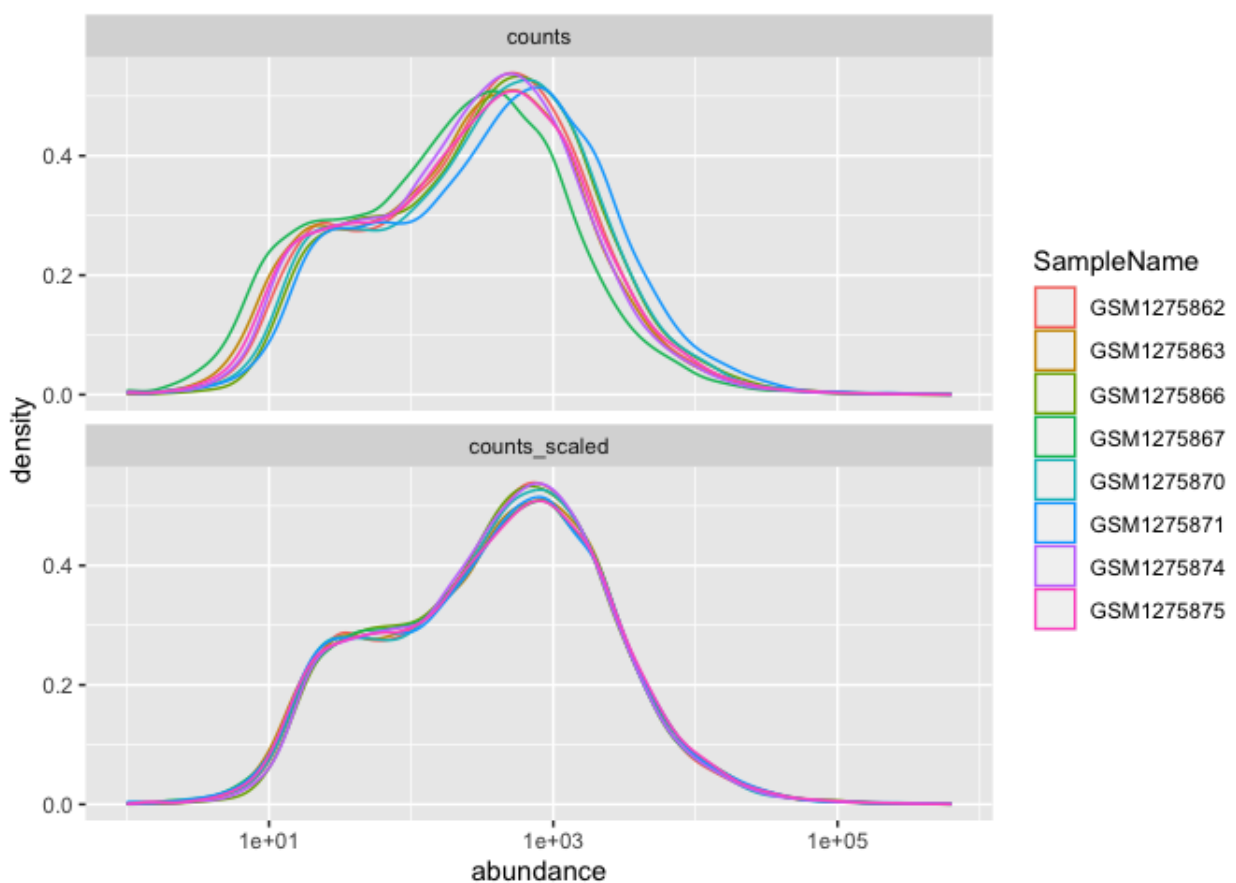
Here, faceting allowed us to visualize multiple features of our data. We were able to see count distributions by sample as well as normalized vs non-normalized counts.

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

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

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

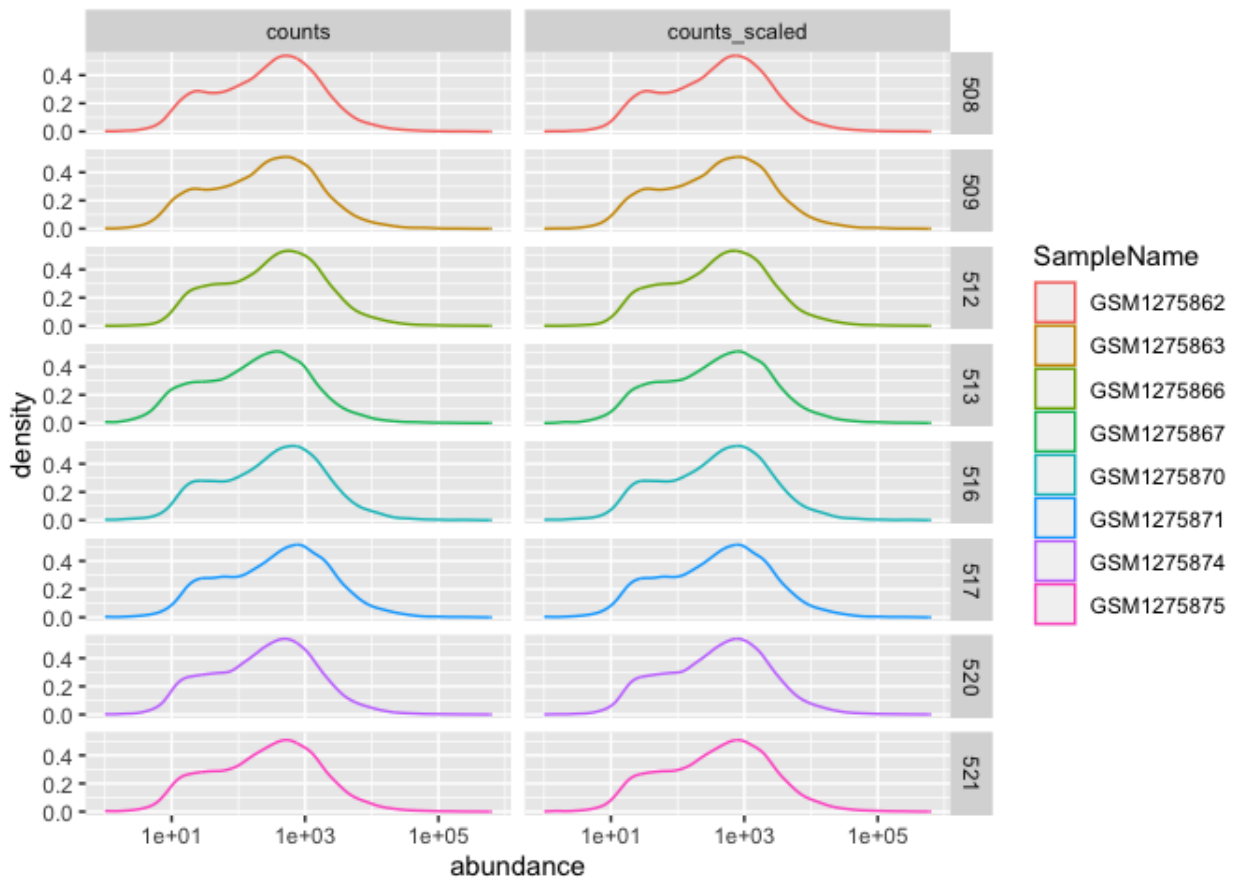


We could plot each sample individually using `facet_grid()`

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

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

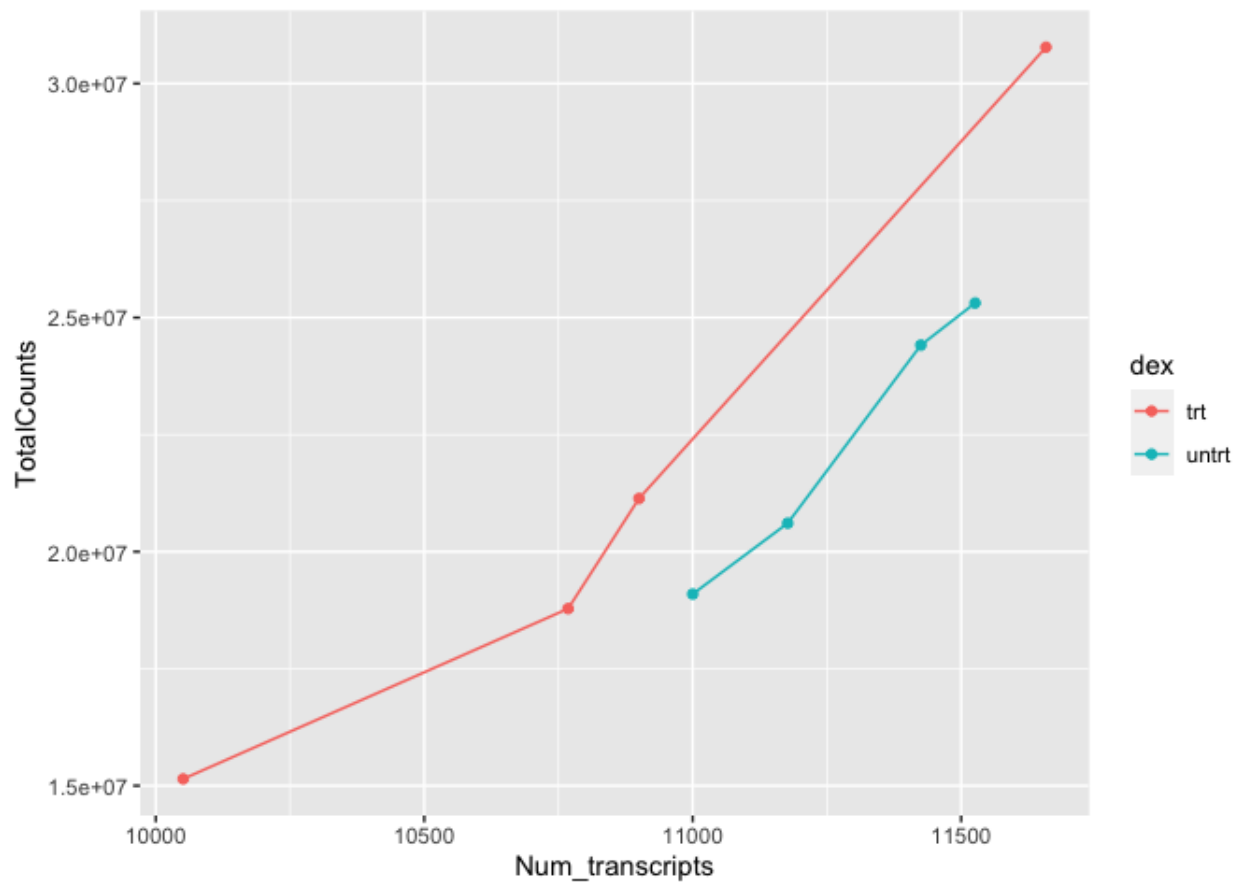
```
## Warning: Removed 140 rows containing non-finite values (`stat_dens
```



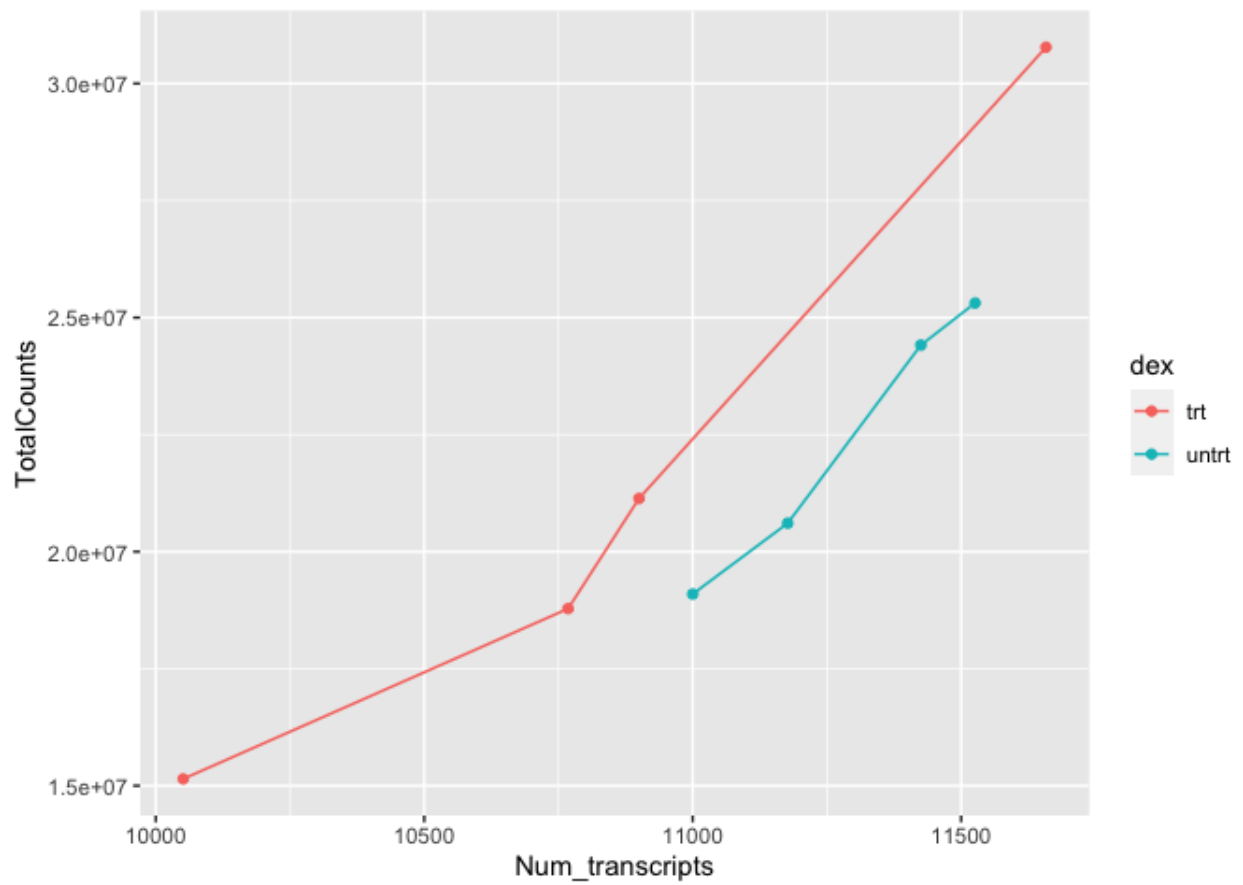
Using multiple geoms per plot

Because we build plots using layers in ggplot2. We can add multiple geoms to a plot to represent the data in unique ways.

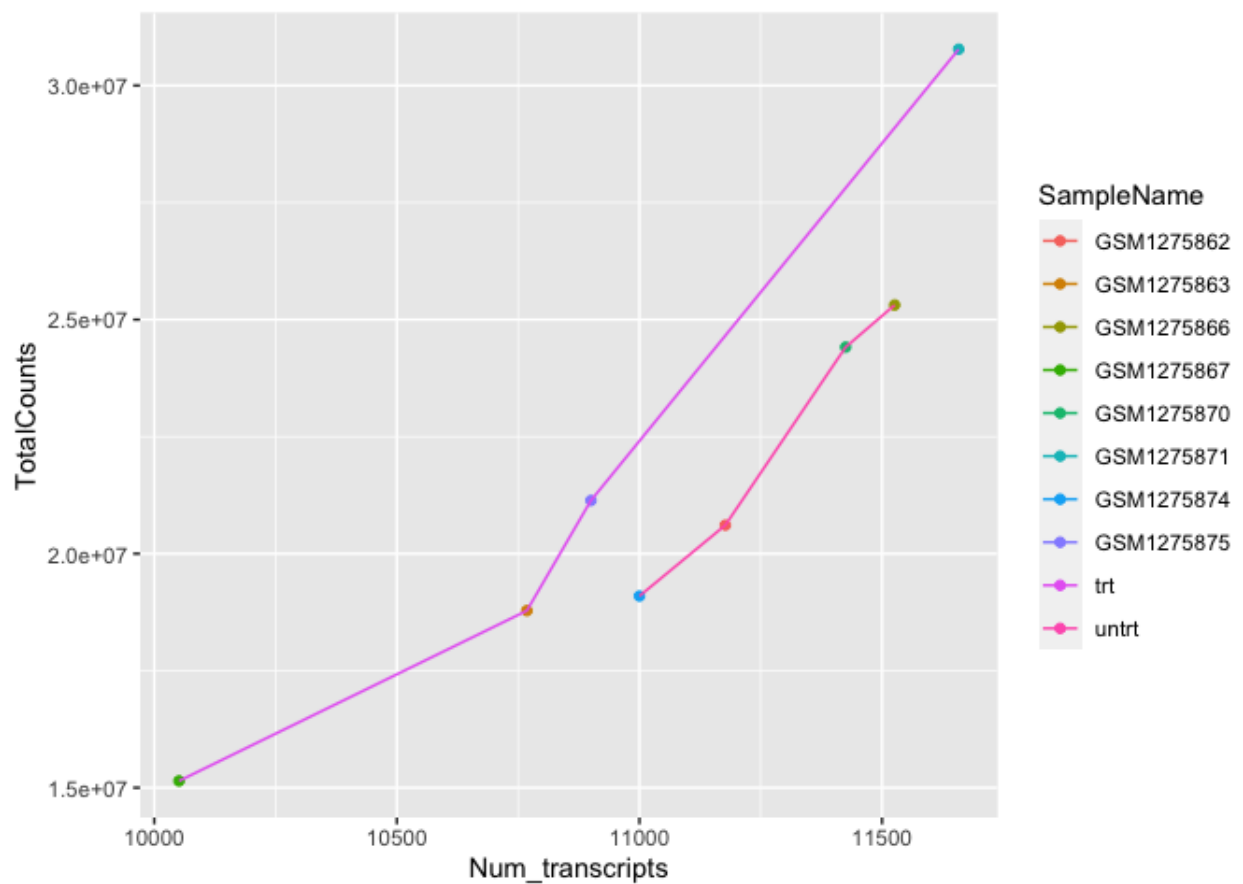
```
#We can combine geoms; here we combine a scatter plot with a
#add a line to our plot
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,color=dex)) +
  geom_line(aes(x=Num_transcripts, y = TotalCounts,color=dex))
```



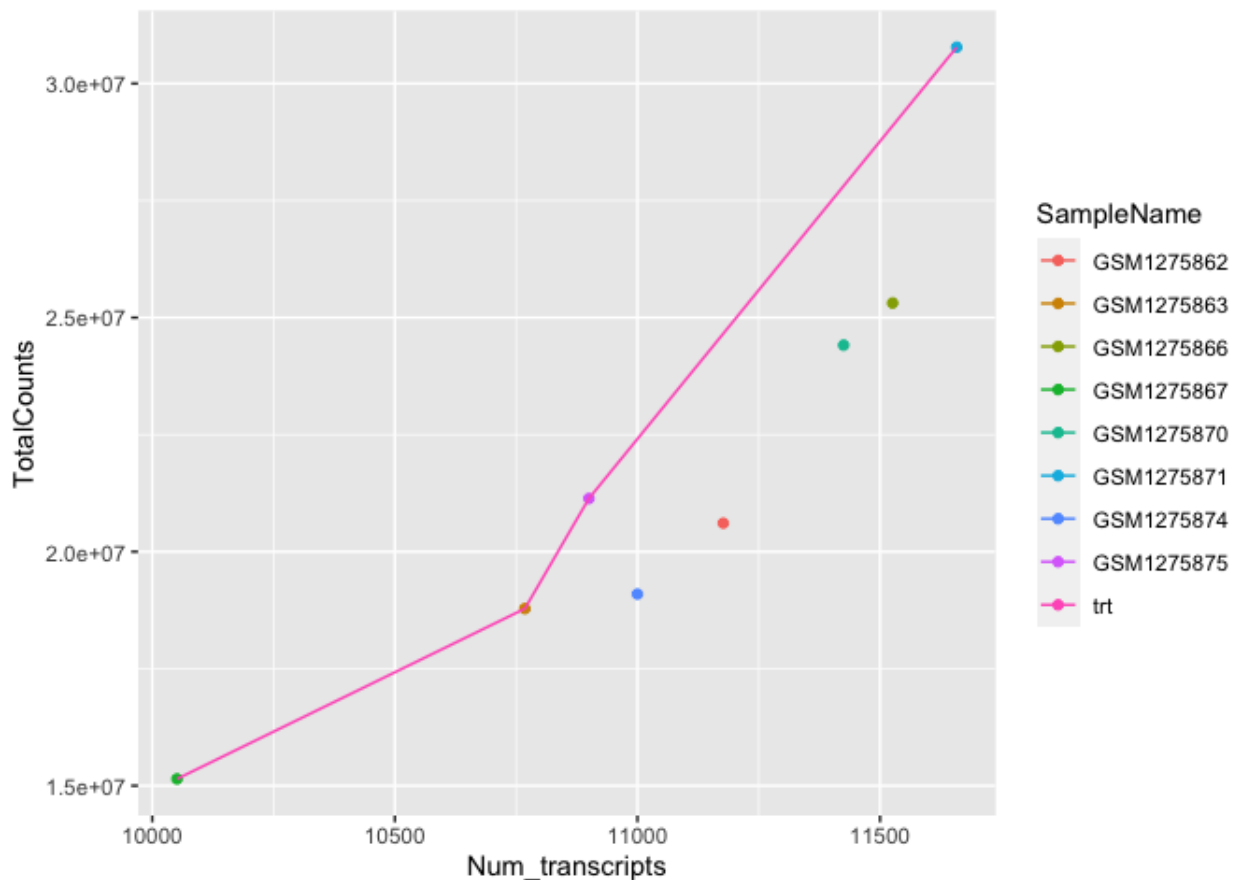
```
#to make our code more effective, we can put shared aesthetics in the  
#ggplot function  
ggplot(data=sc, aes(x=Num_transcripts, y = TotalCounts,color=dex)) +  
  geom_point() +  
  geom_line()
```



```
#or plot different aesthetics per layer
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,
                 color=SampleName)) +
  geom_line(aes(x=Num_transcripts, y = TotalCounts,color=dex))
```



```
#you can also add subsets of data in a new layer without overriding
#preceding layers
#let's only provide a line for the treated samples
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://cran.r-project.org/web/packages/ggnewscale/index.html) (<https://cran.r-project.org/web/packages/ggnewscale/index.html>).

Other data visualization options in R

We will continue with `ggplot2` in the next lesson, but before we get there, let's take a moment to discuss other R visualization options.

R base graphics

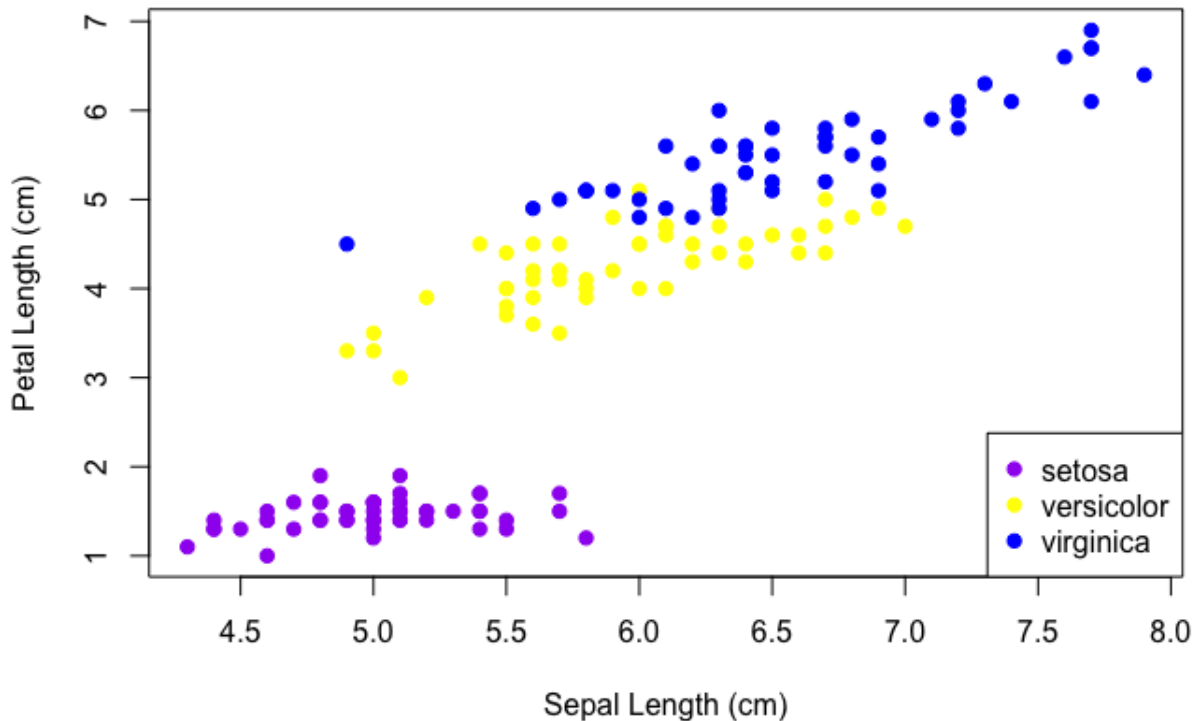
You do not need to load a package to visually explore data. Rather, you can use base R graphics for plotting (from the `graphics` package). This plotting is fairly different from `ggplot2`, which is based on the `grid` package. Unlike `ggplot2` the data does not need to be organized in a data frame to use base R graphics. The plots are built line-by-line using an "Artist's palette model" (<https://bookdown.org/rdpeng/exdata/plotting-systems.html>), and because of this, it is difficult to preserve plots from base R graphics as objects to be manipulated later, as you can with `ggplot2`.

You can obtain fairly nice figures using base R graphics; however, it often will take more lines of code.

The most common function from R base graphics is `plot()`. For a complete list of functions, use `library(help = "graphics")`.

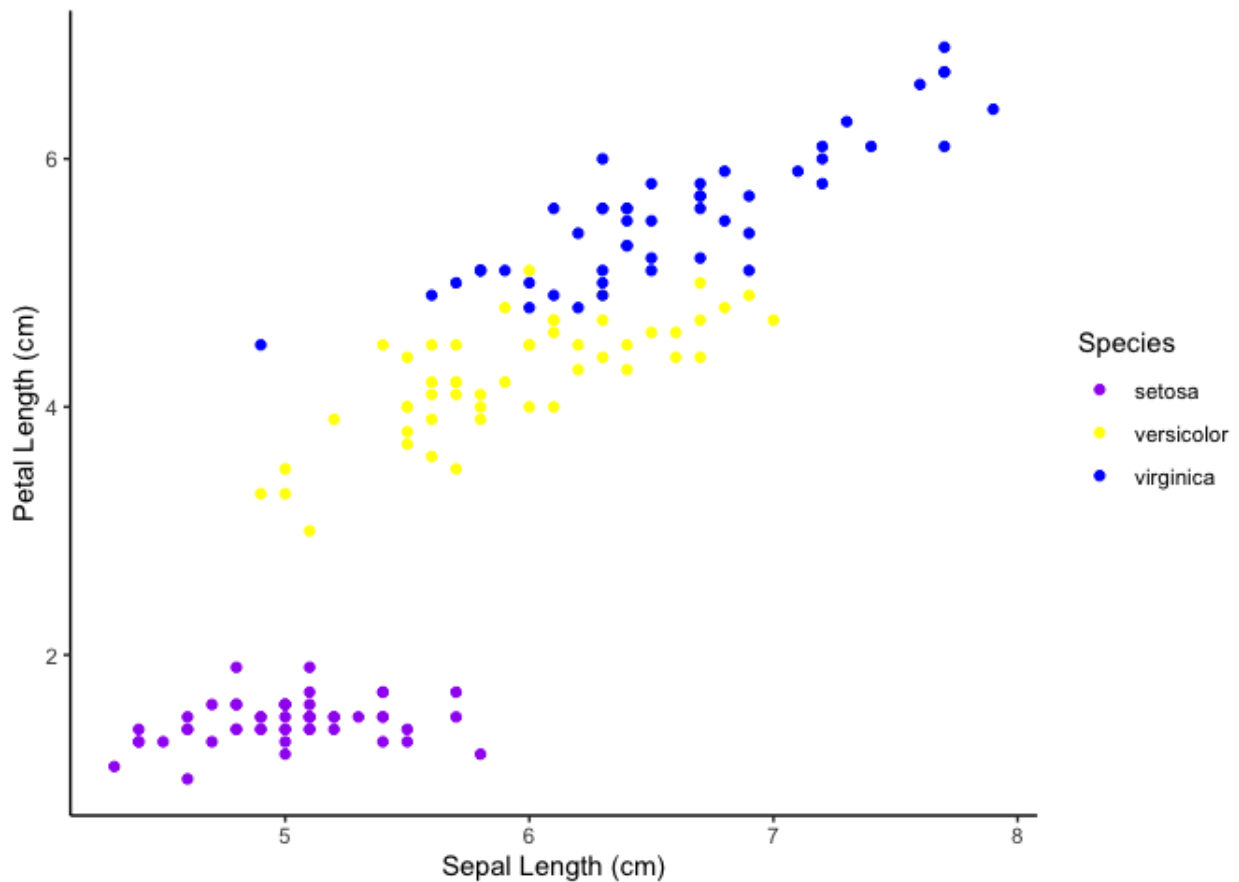
Base R graph

```
plot(iris$Sepal.Length, iris$Petal.Length, pch=19,
     col=c("purple", "yellow", "blue")[as.numeric(iris$Species)],
     xlab="Sepal Length (cm)", ylab="Petal Length (cm)")
legend("bottomright", legend=levels(iris$Species),
      col=c("purple", "yellow", "blue"), pch=19)
```



ggplot2 graph

```
ggplot(data=iris)+
  geom_point(aes(Sepal.Length, Petal.Length, color=Species))+
  scale_color_manual(values=c("purple", "yellow", "blue"))+
  theme_classic() +
  labs(x="Sepal Length (cm)", y="Petal Length (cm)")
```



Lattice

The `lattice` package is another prominent graphic system in R. Like `ggplot2` this is also based on the `grid` package.

For more information comparing the three plotting systems, see [this chapter \(https://bookdown.org/rdpeng/exdata/plotting-systems.html\)](https://bookdown.org/rdpeng/exdata/plotting-systems.html) from *Exploratory Data Analysis with R*.

Resource list

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

Acknowledgements

Material from this lesson was adapted from Chapter 3 of *R for Data Science* (<https://r4ds.had.co.nz/data-visualisation.html>) and from "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.

Introduction to Data Visualization with R (Part 2)

Objectives

1. Review the grammar of graphics template.
2. Learn about the statistical transformations inherent to geoms.
3. Learn more about fine tuning figures with labels, legends, scales, and themes.
4. Learn how to save plots with `ggsave()`.
5. Review general tips for creating publishable figures.

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, 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/Chap-Graphics.html>))

```
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.

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

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

Importing 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
#We used this in lesson 2 so you may not need to reload
scaled_counts<-
  read.delim("../data/filtlowabund_scaledcounts_airways.txt",
             as.is=TRUE)
```

```
dexp<-read.delim("../data/diffexp_results_edger_airways.txt",
                 as.is=TRUE)

#let's get some data
#we are only interested in transcript counts greater than 100
#read in the data
sc<-read.csv("../data/sc.csv")
```

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 \(https://r4ds.had.co.nz/data-visualisation.html#statistical-transformations\)](https://r4ds.had.co.nz/data-visualisation.html#statistical-transformations)

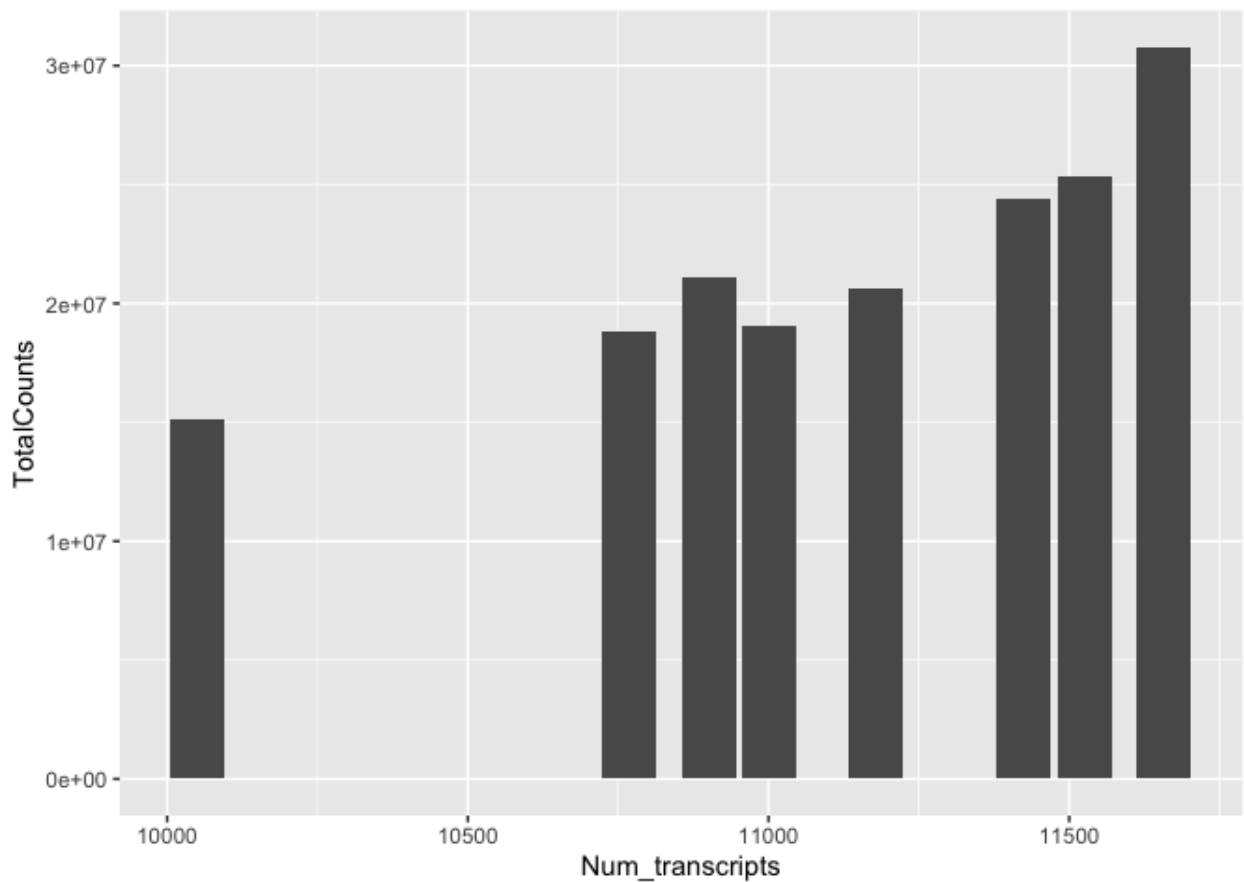
Let's plot a bar graph using the data (sc).

```
#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.
```

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.

Let's look at another example.

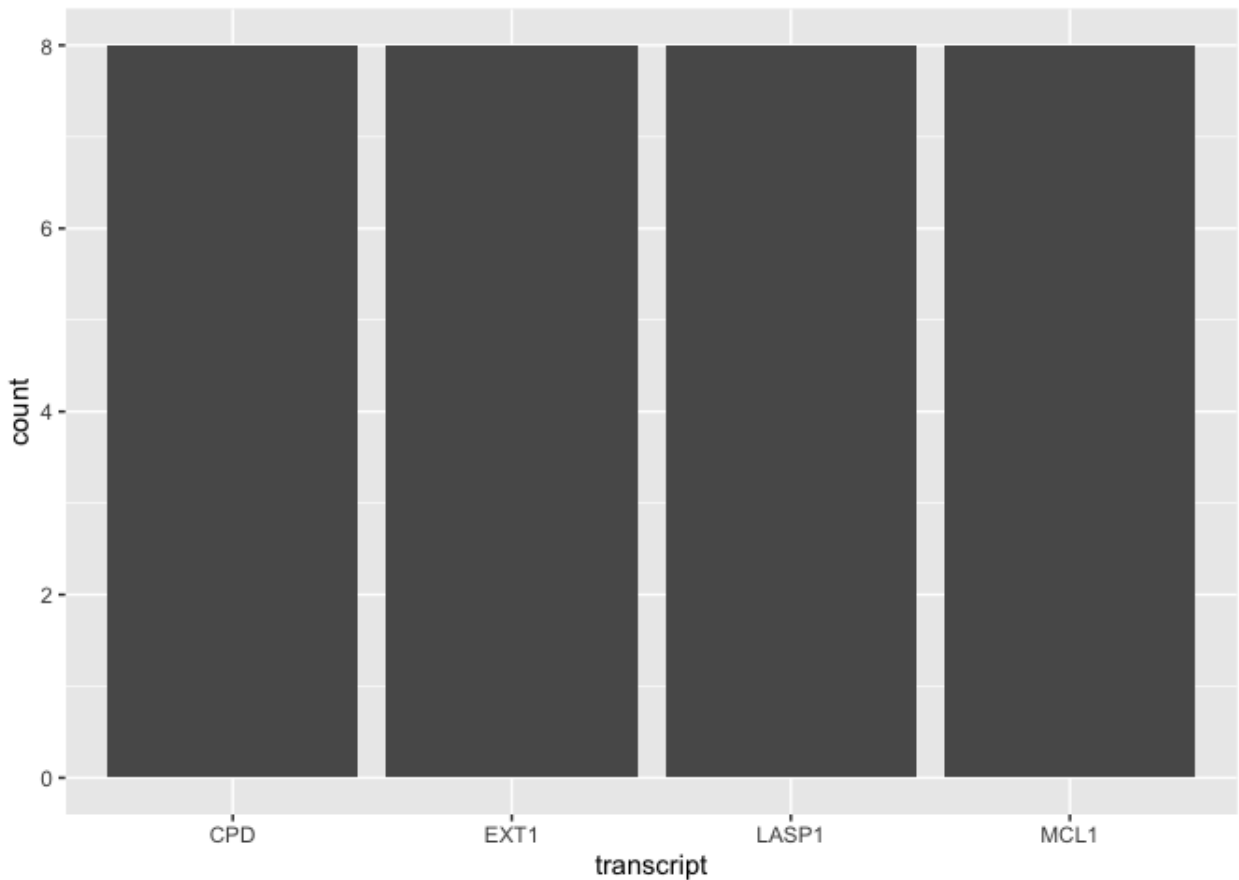
```
#Let's filter our data to only include 4 transcripts of interest
#We used this code in the tidyverse lesson
keep_t<-c("CPD","EXT1","MCL1","LASP1")
interesting_trnsc<-scaled_counts %>%
  filter(transcript %in% keep_t)

#the default here is `stat_count()`
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.
```

```
#Let's take away the y aesthetic
ggplot(data = interesting_trnsc) +
```

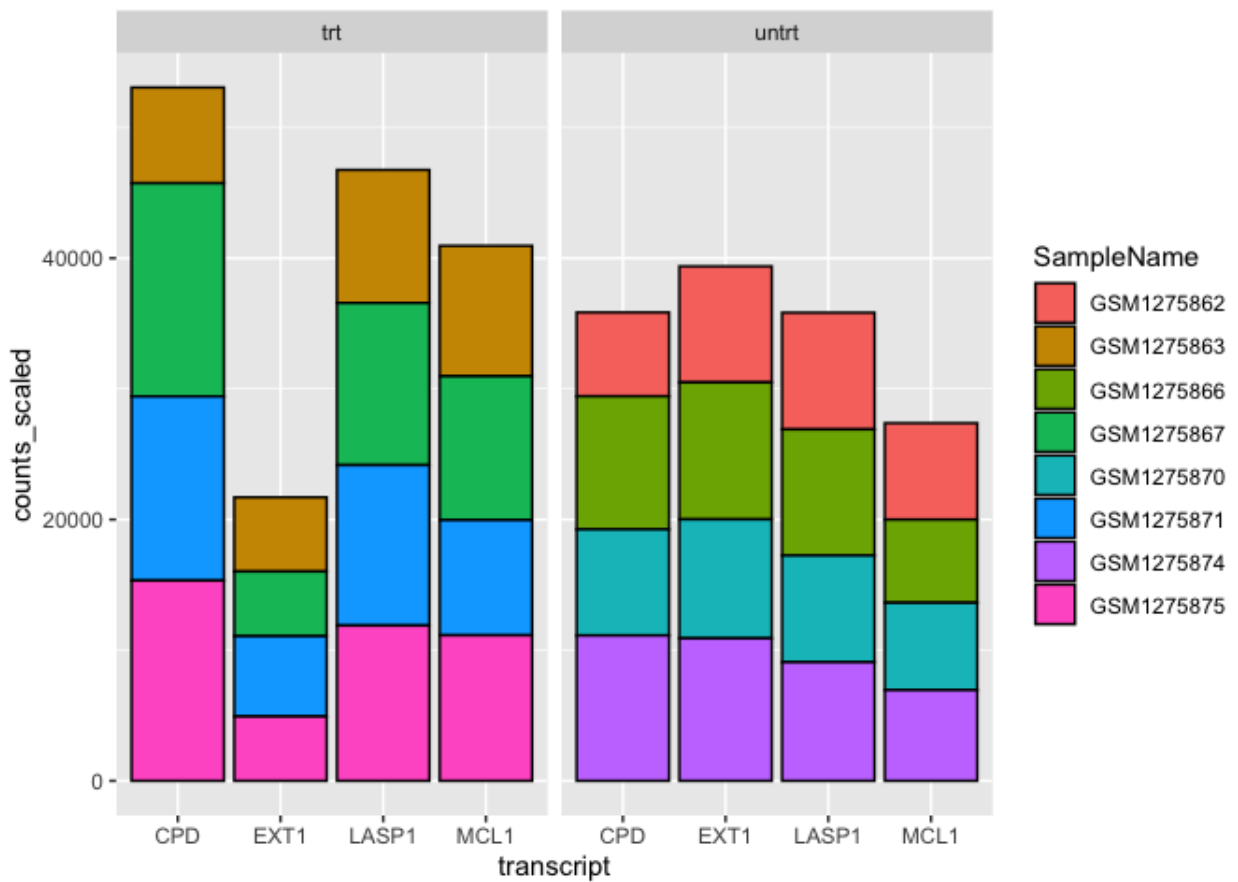
```
geom_bar(mapping = aes(x = transcript))
```



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

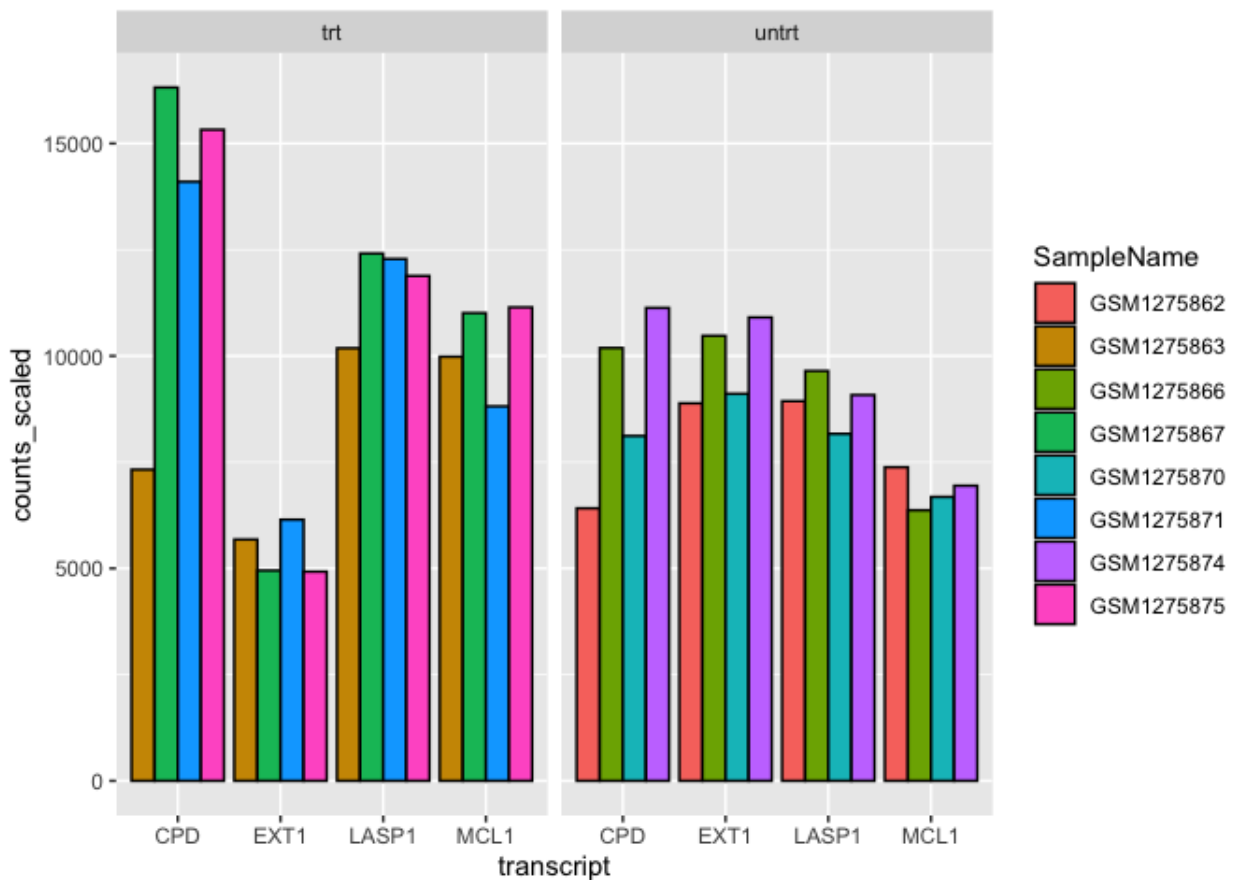
Let's change the `stat` parameter to "identity". This will plot the raw values of the normalized counts rather than how many rows are present for each transcript.

```
#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)
```



What if we wanted the columns side by side?

```
#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.

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()`).

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

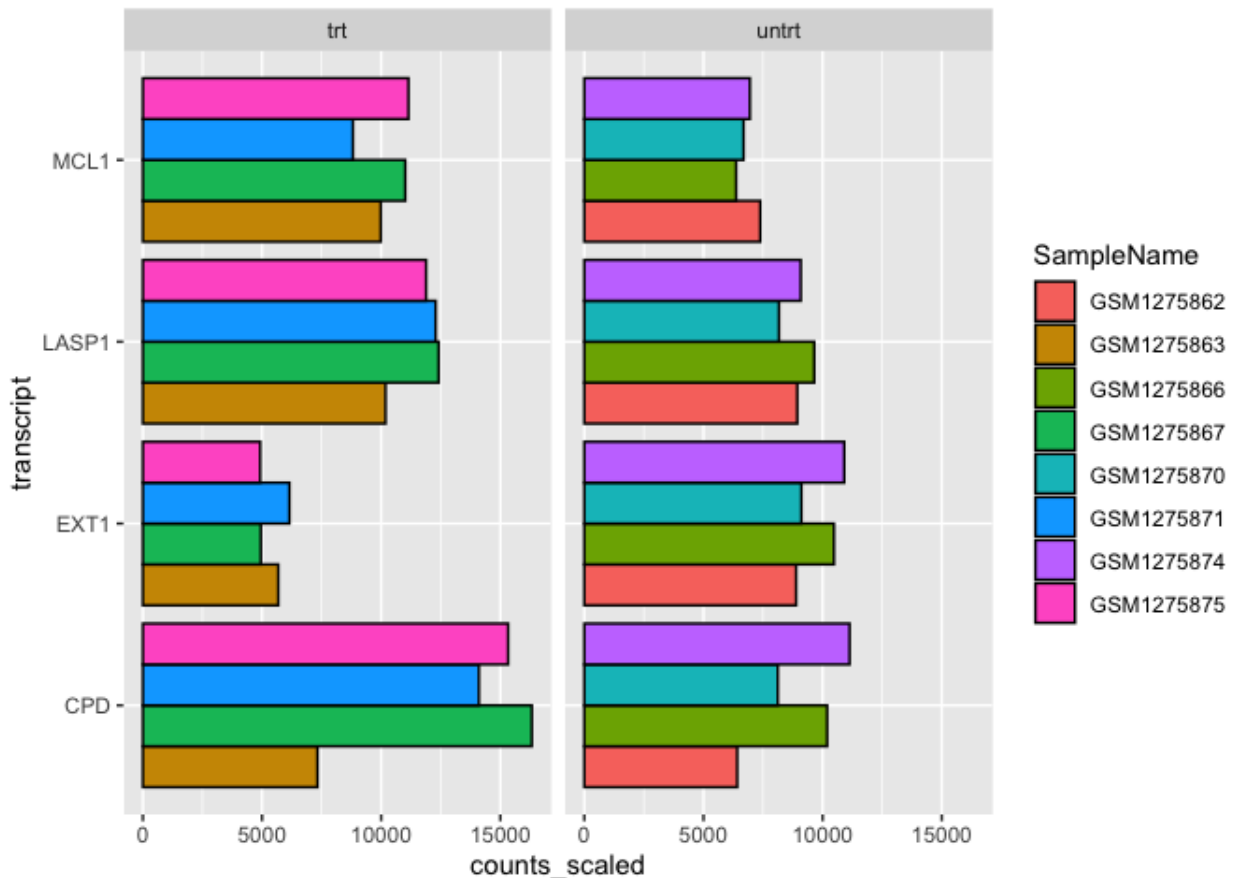
ggplot(data = interesting_trnsc) +
  geom_bar(mapping = aes(x = transcript,y=counts_scaled,
```



```

fill=SampleName),
  stat="identity",color="black",position="dodge") +
facet_wrap(~dex) +
coord_flip()

```



Labels, legends, scales, and themes

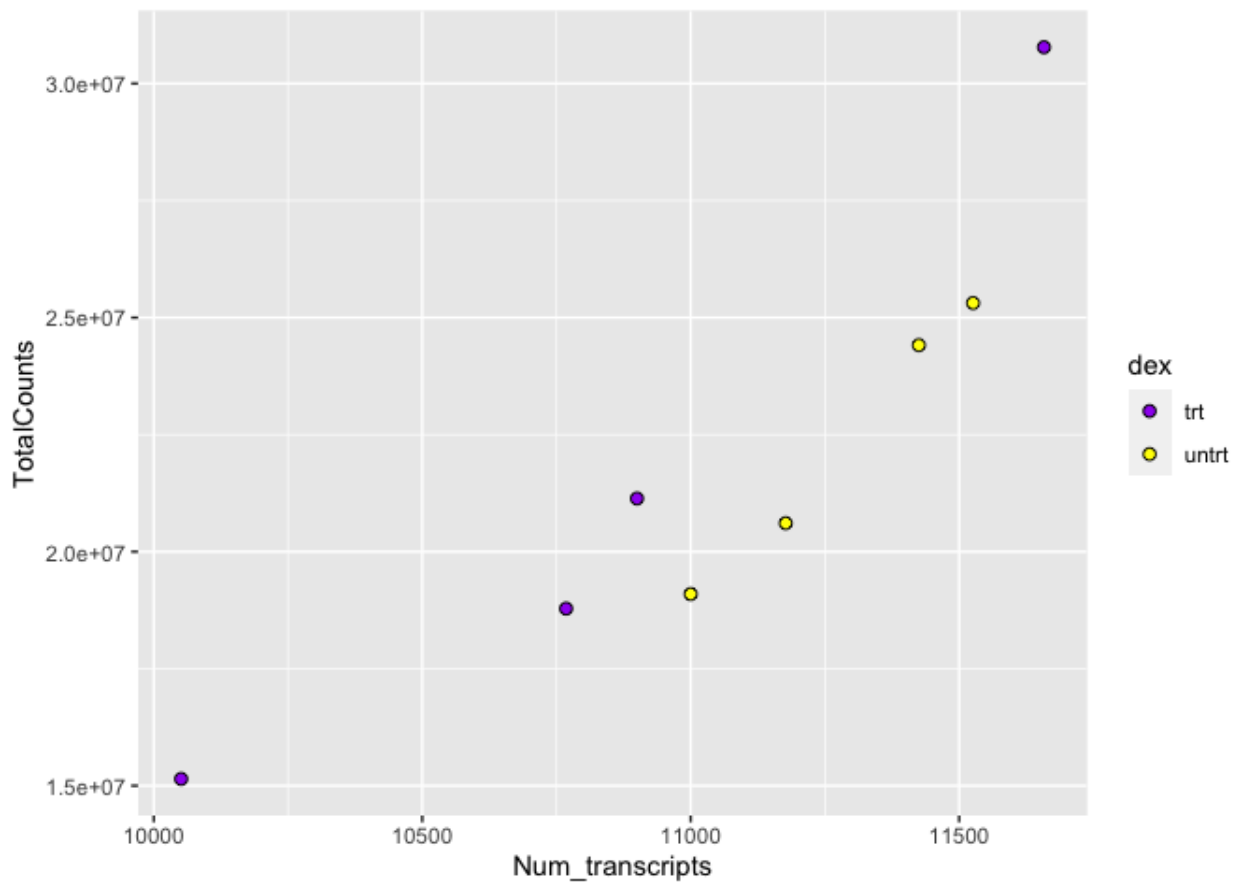
How do we ultimately get our figures to a publishable state? The bread and butter of pretty plots really falls to the additional non-data layers of our ggplot2 code. These layers will include code to label the axes, scale the axes, and customize the legends and [theme](https://ggplot2.tidyverse.org/reference/theme.html)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

The default axes and legend titles come from the ggplot2 code.

```

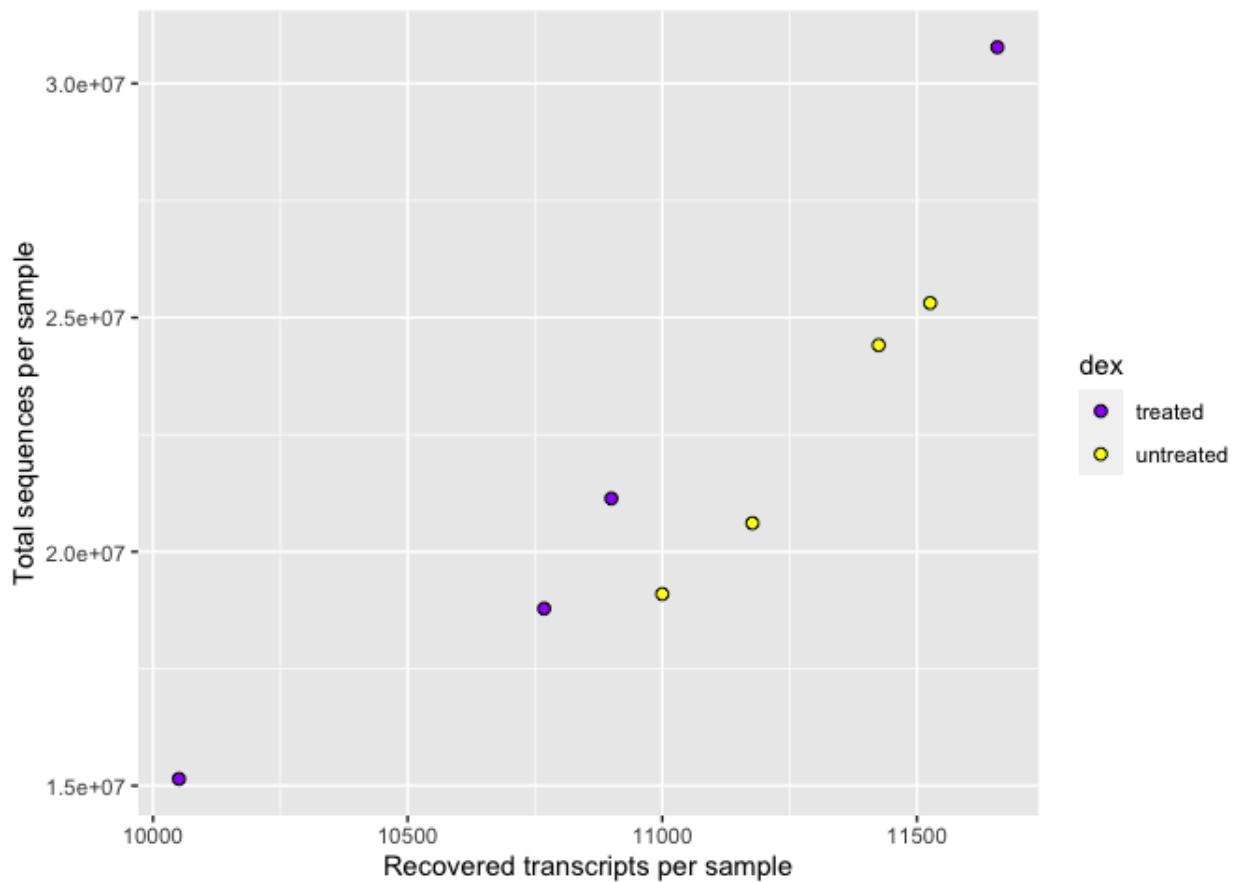
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
            shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"))

```



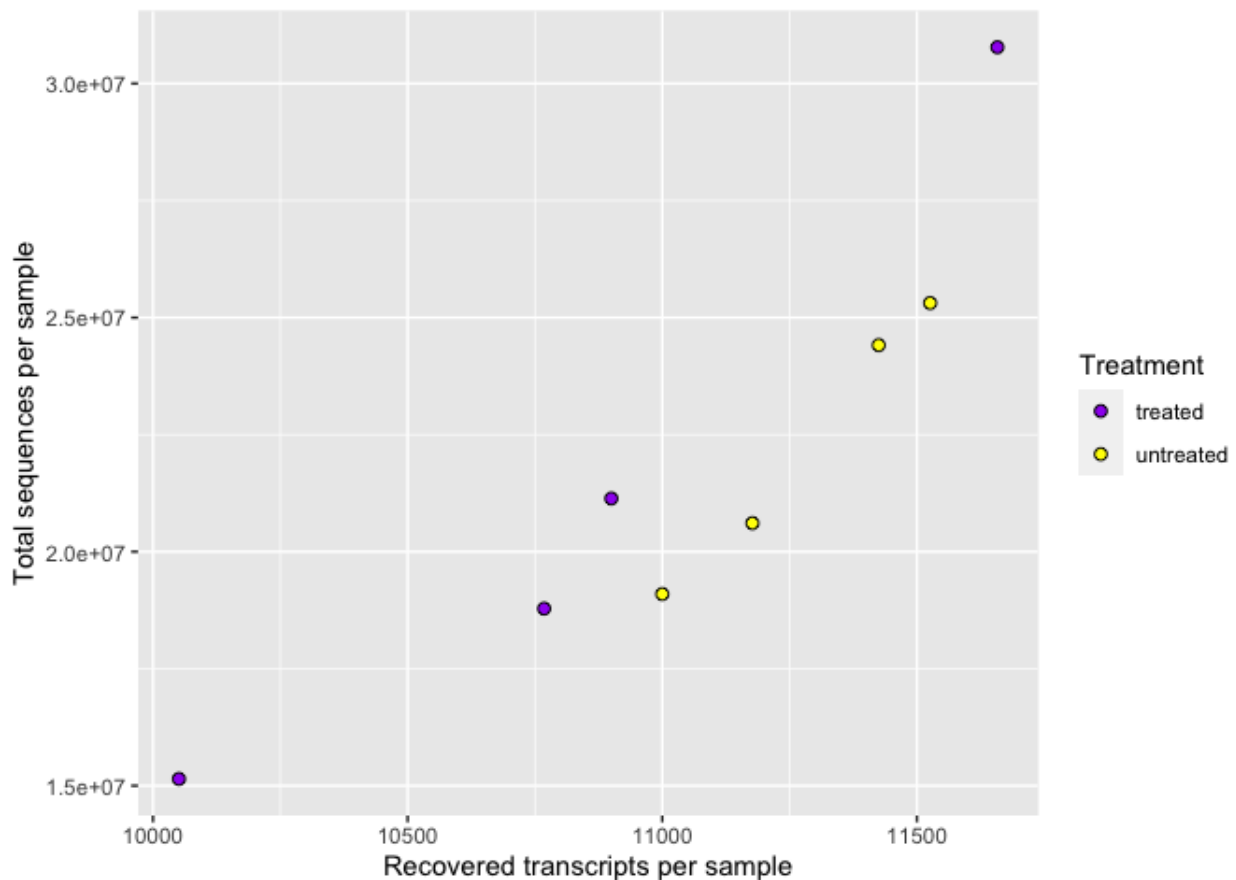
In the above plot, the y-axis label (TotalCounts) is the variable name mapped to the y aesthetic, while the x-axis label (Num_transcripts) is the variable name named to the x aesthetic. The fill aesthetic was set equal to "dex", and so this became the default title of the fill legend. We can change these 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'))+
  #can change labels of fill levels along with colors
  xlab("Recovered transcripts per sample") + #add x label
  ylab("Total sequences per sample") #add y label
```



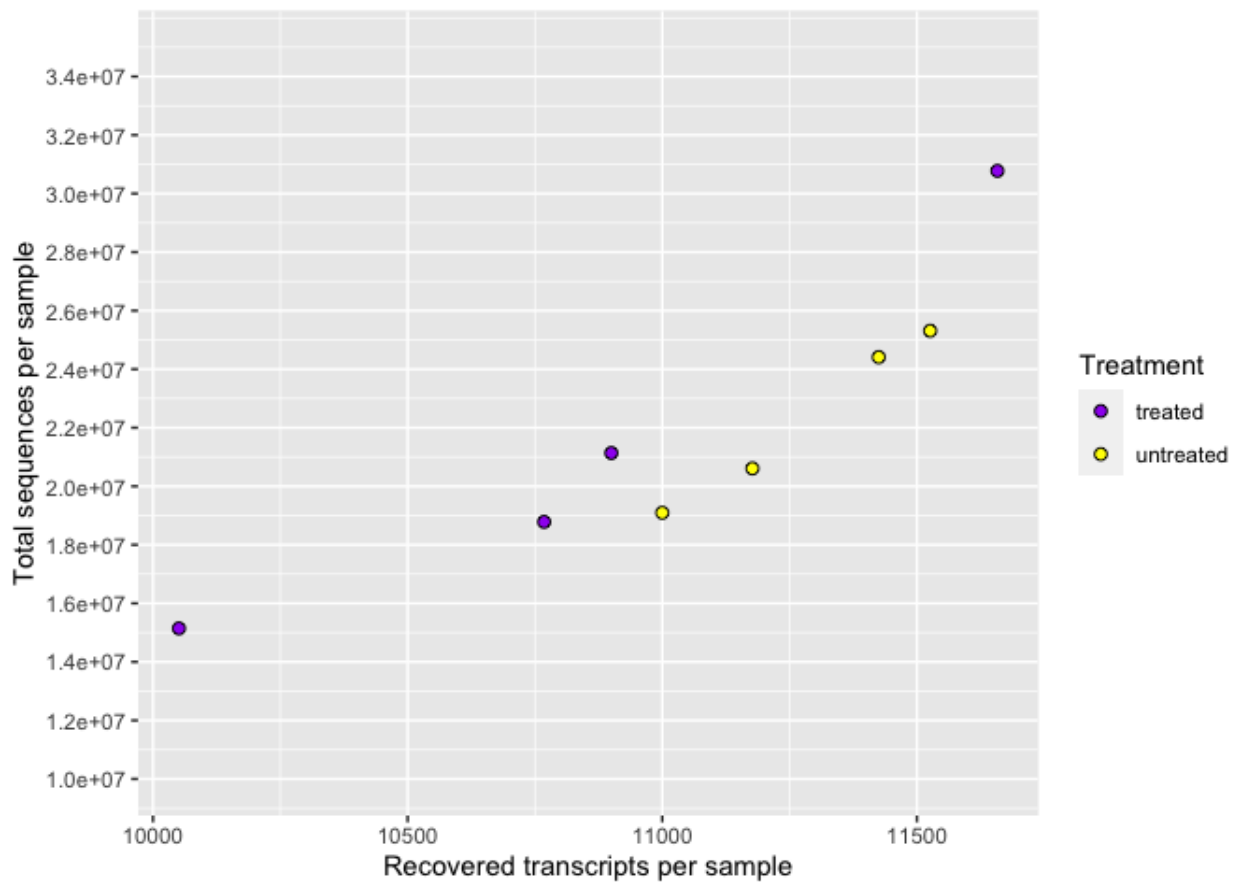
Let's change the legend title.

```
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'))+
  #can change labels of fill levels along with colors
  xlab("Recovered transcripts per sample") + #add x label
  ylab("Total sequences per sample") + #add y label
  guides(fill = guide_legend(title="Treatment"))
```



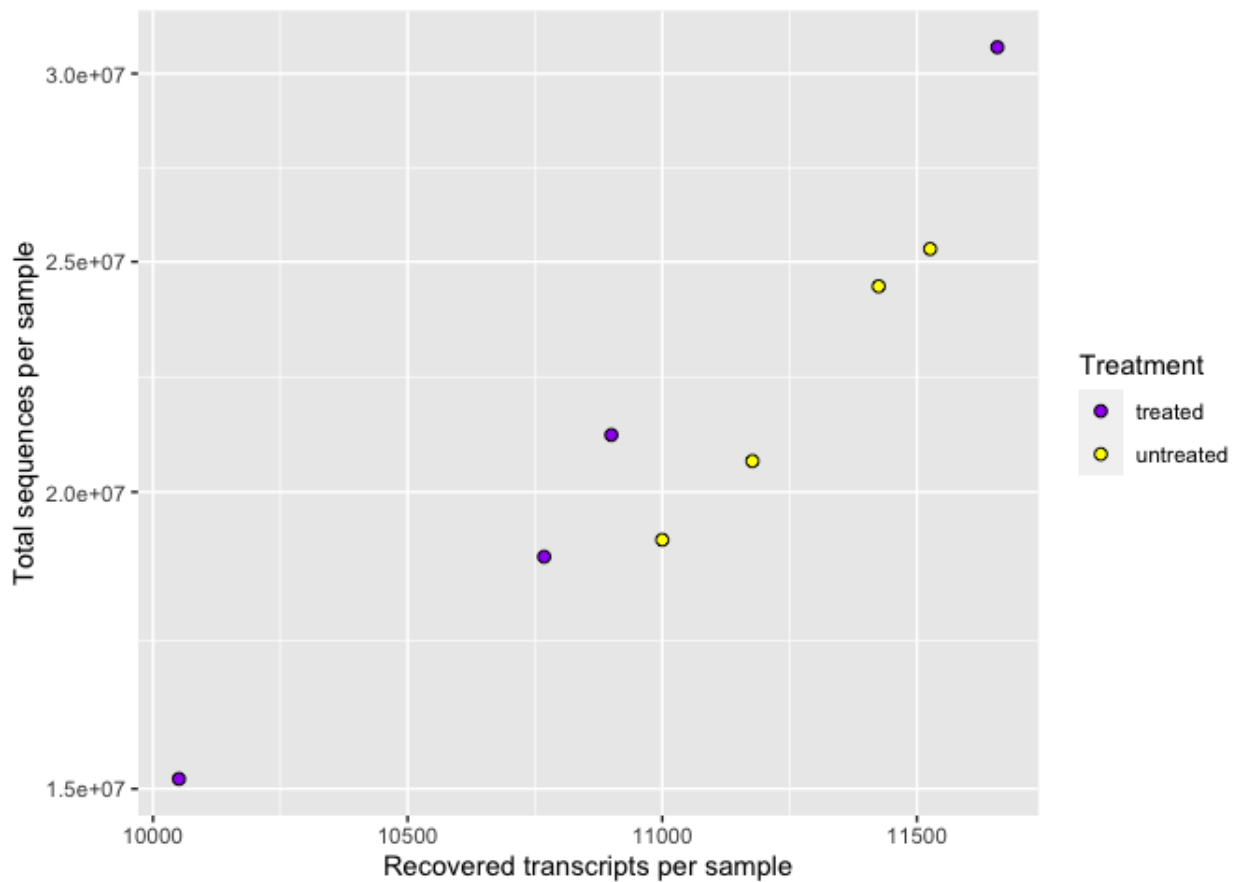
We can modify the axes scales of continuous variables using `scale_x_continuous()` and `scale_y_continuous()`. Discrete (categorical variable) axes can be modified using `scale_x_discrete()` and `scale_y_discrete()`.

```
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'))+
  #can change labels of fill levels along with colors
  xlab("Recovered transcripts per sample") + #add x label
  ylab("Total sequences per sample") + #add y label
  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
```



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

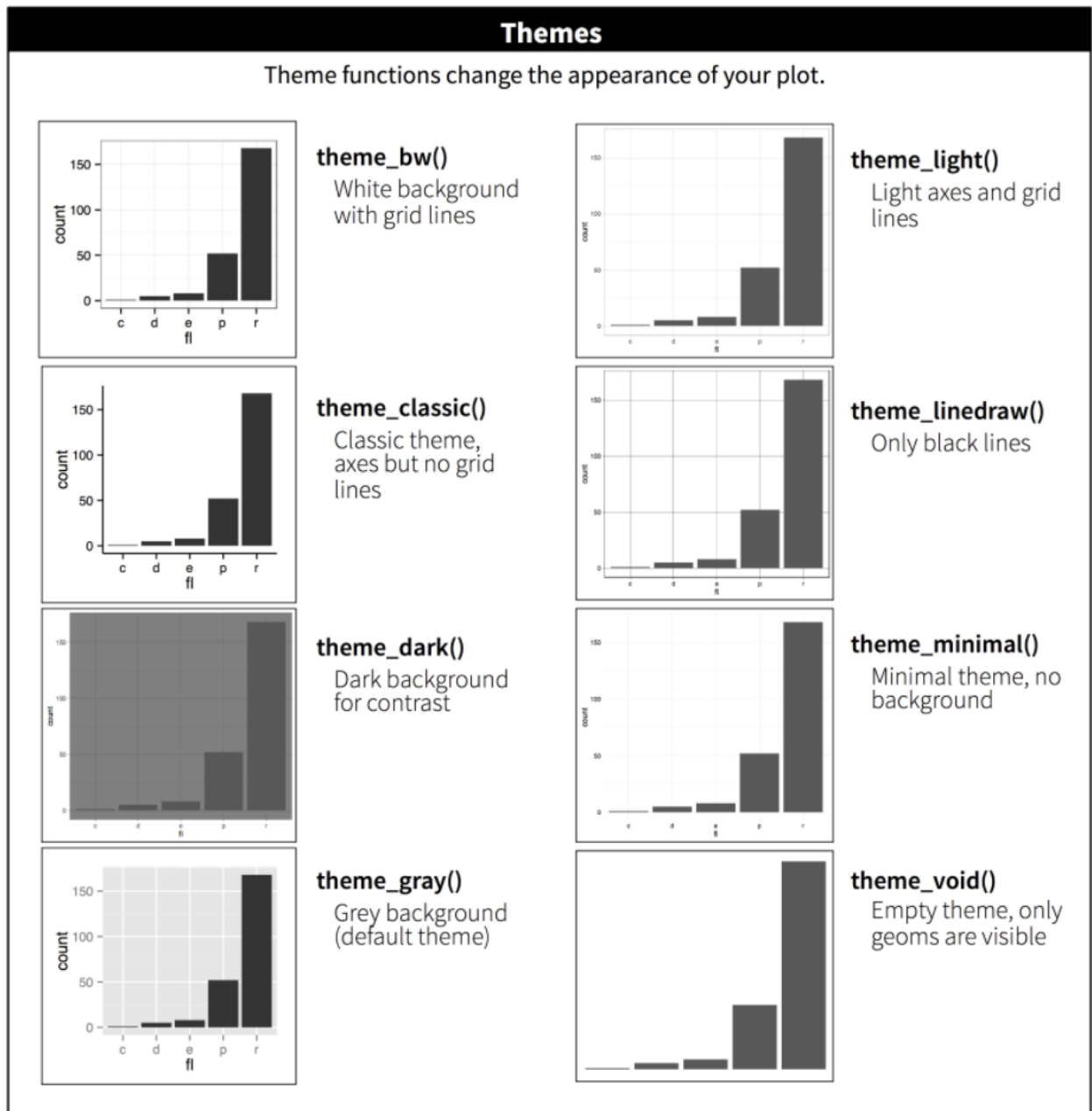
```
ggplot(data=sc) +
  geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
             shape=21,size=2) +
  scale_fill_manual(values=c("purple", "yellow"),
                   labels=c('treated','untreated'))+
  #can change labels of fill levels along with colors
  xlab("Recovered transcripts per sample") + #add x label
  ylab("Total sequences per sample") + #add y label
  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.

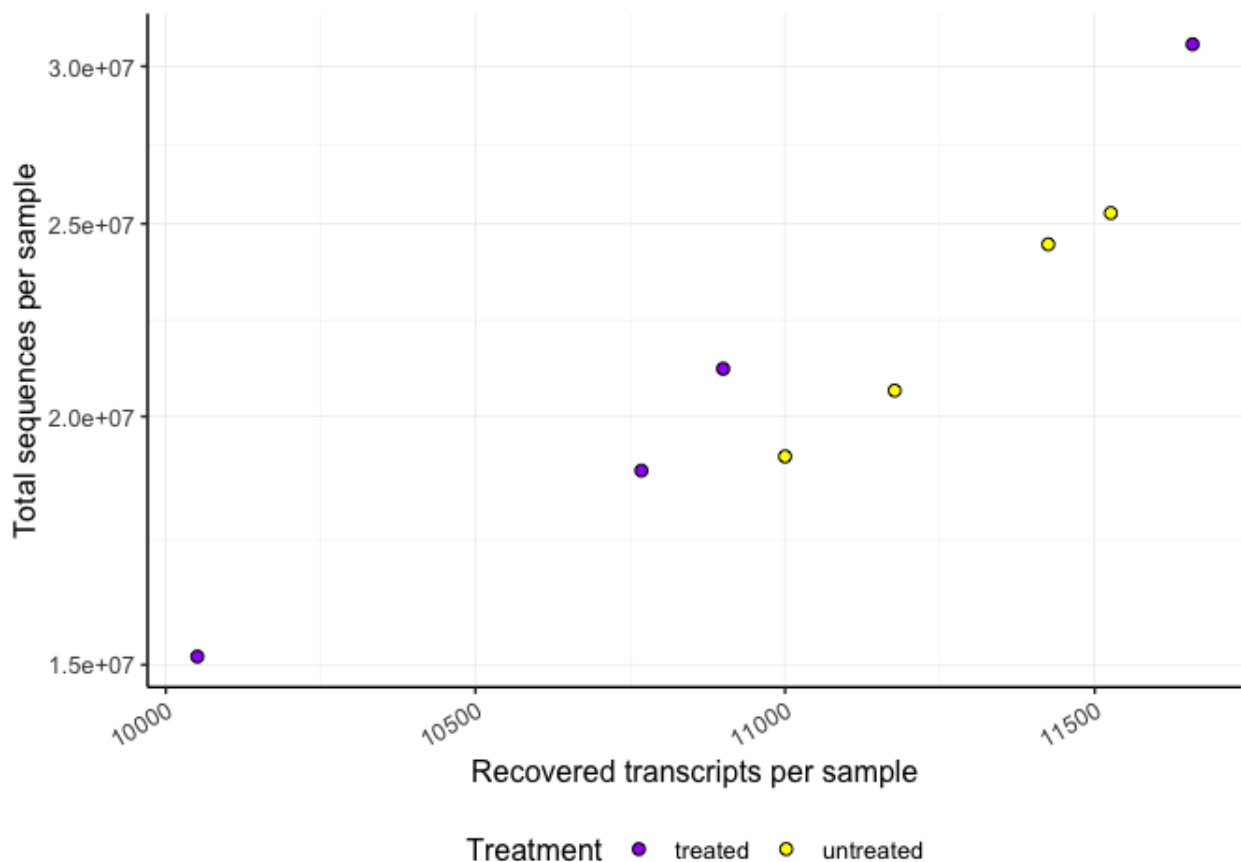
Create a custom theme to use with multiple figures.

```
#Setting a theme
my_theme <-
  theme_bw() +
  theme(
    panel.border = element_blank(),
    axis.line = element_line(),
    panel.grid.major = element_line(size = 0.2),
    panel.grid.minor = element_line(size = 0.1),
    text = element_text(size = 12),
    legend.position = "bottom",
```

```
axis.text.x = element_text(angle = 30, hjust = 1, vjust = 1)
)
```

```
## Warning: The `size` argument of `element_line()` is deprecated as
## i Please use the `linewidth` argument instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warn
## generated.
```

```
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'))+
  #can change labels of fill levels along with colors
  xlab("Recovered transcripts per sample") + #add x label
  ylab("Total sequences per sample") + #add y label
  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)
```

Nice plot example

These steps can be used to create a publish worthy figure. For example, let's create a volcano plot of our differential expression results.

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 \(https://training.galaxyproject.org/training-material/topics/transcriptomics/tutorials/rna-seq-viz-with-volcanoplot/tutorial.html\)](https://training.galaxyproject.org/training-material/topics/transcriptomics/tutorials/rna-seq-viz-with-volcanoplot/tutorial.html)

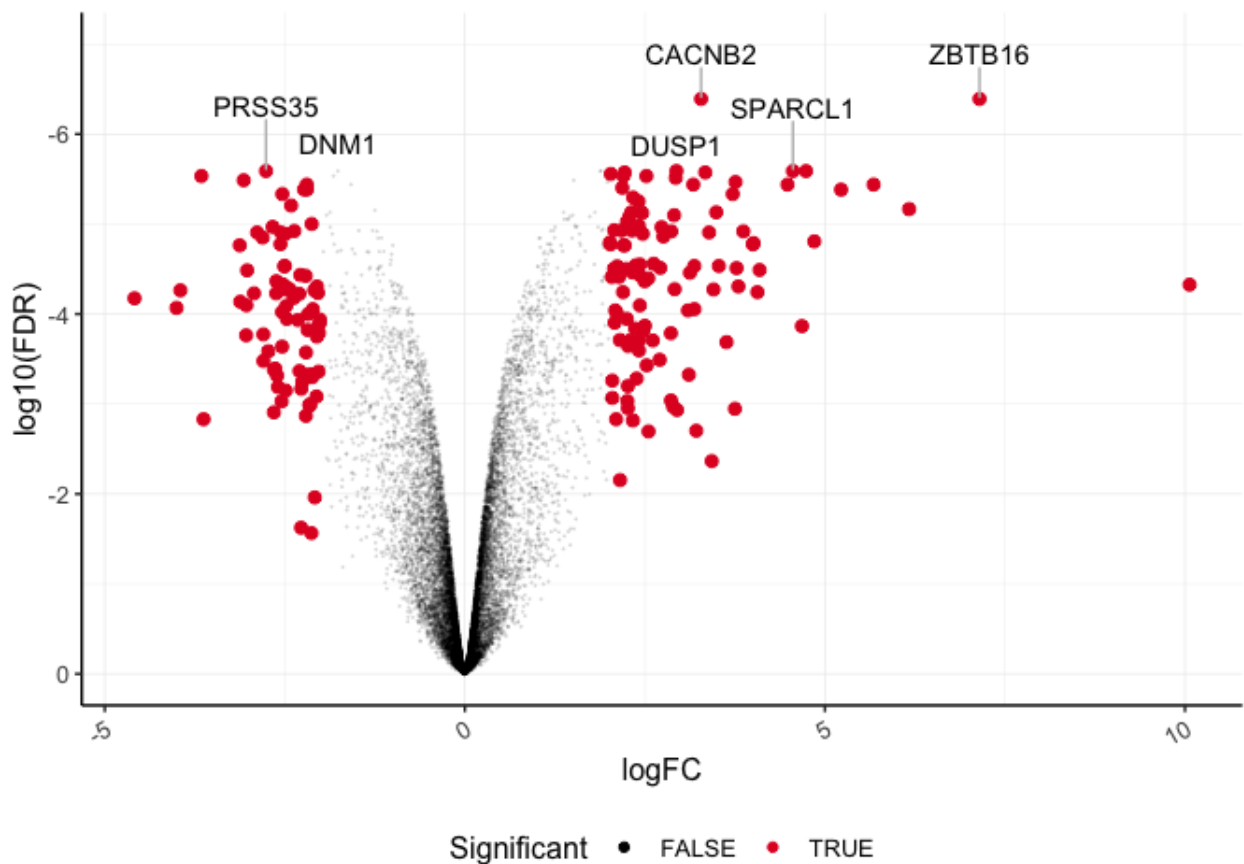
```
#get the data
dexp_sigtrnsc<-dexp %>%
  mutate(Significant = FDR < 0.05 & abs(logFC) >= 2) %>% arrange(FDR)
topgenes<-dexp_sigtrnsc$transcript[1:6]
```

Plot

```
#install.packages(ggrepel)
library(ggrepel)
ggplot(data=dexp_sigtrnsc,aes(x = logFC, y = log10(FDR))) +
  geom_point(aes( color = Significant, size = Significant,
                 alpha = Significant)) +
  geom_text_repel(data=dexp_sigtrnsc %>%
                 filter(transcript %in% topgenes),
                 aes(label=transcript),
                 nudge_y=0.5,hjust=0.5,direction="y",
                 segment.color="gray") +
  scale_y_reverse(limits=c(0,-7))+
  scale_color_manual(values = c("black", "#e11f28")) +
  scale_size_discrete(range = c(0, 2)) +
  guides(size = "none", alpha= "none")+
  my_theme
```

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

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



Enhanced Volcano

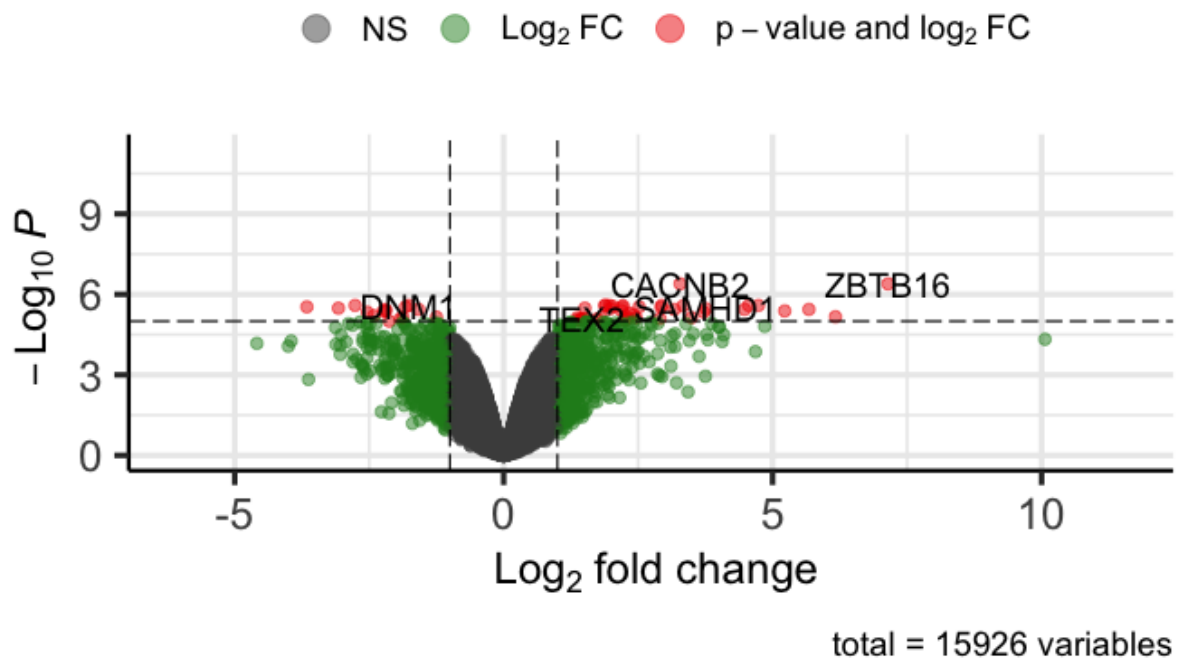


There is a dedicated package for creating volcano plots available in Bioconductor, [EnhancedVolcano](https://bioconductor.org/packages/release/bioc/html/EnhancedVolcano.html) (<https://bioconductor.org/packages/release/bioc/html/EnhancedVolcano.html>). Plots created using this package can be customized 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')
```

Enhanced Volcano with Airways

EnhancedVolcano



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 packages

There are many complementary R packages related to creating publishable figures using `ggplot2`. Check out the packages `cowplot` (<https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html>) and `ggpubr` (<https://github.com/kassambara/ggpubr>). `Cowplot` is particularly great for providing functions that facilitate arranging multiple plots in a grid panel. Usually publications restrict the number of figures allowed, and so it is helpful to be able to group multiple figures into a single figure panel. `GGpubr` is particularly great for beginners, providing easy code to make publish worthy figures. It is particularly great for stats integration and easily incorporating brackets and p-values for group comparisons.

Acknowledgements

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

Introduction to Bioconductor and report generation with R

Objectives

1. To explore Bioconductor, a repository for R packages related to biological data analysis.
2. To learn about options for report generation with R: RMarkdown and Quarto.

Introducing Bioconductor

Bioconductor (<https://bioconductor.org/>) is both an open source project and repository for R packages related to the analysis of biological data, primarily bioinformatics and computational biology, and as such it is a great place to search for -omics packages and pipelines. Read more about the goals of the Bioconductor project [here](https://bioconductor.org/about/). (<https://bioconductor.org/about/>)

Since its inception in 2001, the Bioconductor project has kept pace with emerging technologies from microarrays to spatial transcriptomics.

The current release of Bioconductor (v 3.18) contains:

- 2,266 software packages
- 429 experiment data packages
- 920 annotation packages
- 30 workflows
- 4 books

What types of packages are available in Bioconductor?

Bioconductor packages are divided into four types:

1. software
2. annotation data
3. experiment data
4. workflows.

Software packages themselves can be subdivided into packages that provide infrastructure (i.e., classes) to store and access data, and packages that provide methodological tools to process data stored in those data structures. This separation of structure and analysis is at the core of the Bioconductor project,

encouraging developers of new methodological software packages to thoughtfully re-use existing data containers where possible, and reducing the cognitive burden imposed on users who can more easily experiment with alternative workflows without the need to learn and convert between different data structures.

Annotation data packages provide self-contained databases of diverse genomic annotations (e.g., gene identifiers, biological pathways). Different collections of annotation packages can be found in the Bioconductor project. They are identifiable by their respective naming pattern, and the information that they contain. For instance, the so-called OrgDb packages (e.g., the `org.Hs.eg.db` package) provide information mapping different types of gene identifiers and pathway databases; the so-called EnsDb (e.g., `EnsDb.Hsapiens.v86`) packages encapsulate individual versions of the Ensembl annotations in Bioconductor packages; and the so-called TxDb packages (e.g., `TxDb.Hsapiens.UCSC.hg38.knownGene`) encapsulate individual versions UCSC gene annotation tables.

Experiment data packages provide self-contained datasets that are often used by software package developers to demonstrate the use of their package on well-known standard datasets in their package vignettes.

Finally, **workflow packages** exclusively provide collections of vignettes that demonstrate the combined usage of several other packages as a coherent workflow, but do not provide any new source code or functionality themselves.

--- [Introduction to Bioconductor from The Bioconductor Project \(https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor.html\)](https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor.html), a lesson in the Carpentries Incubator

For a comprehensive list of packages ranked by number of downloads, click [here \(https://bioconductor.org/packages/release/BiocViews.html#___Software\)](https://bioconductor.org/packages/release/BiocViews.html#___Software).

Bioconductor versions and install

Bioconductor release schedule

New versions of Bioconductor are released every 6 months and work with a specific version of R.

Because of this release schedule and associated automated testing, "each Bioconductor release provides a suite of packages that are mutually compatible, traceable, and guaranteed to function for the associated version of R." --- [Introduction to Bioconductor from The Bioconductor Project \(https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor.html\)](https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor.html).

The latest version of Bioconductor (Bioconductor 3.18) works with R version 4.3. You may need to update your R installation.

How to install a Bioconductor package?

To install a Bioconductor package, you will first need to install `BiocManager`, a CRAN package. You can then use `BiocManager` to install the Bioconductor core packages and specific packages.

To install the Bioconductor core packages, use the following:

```
#install core packages
if (!require("BiocManager", quietly = TRUE))
  install.packages("BiocManager")
BiocManager::install(version = "3.18")
```

To install a specific package:

```
BiocManager::install("tidybulk") #replace tidybulk with the name of
#the package that interests you.
```

To update installed Bioconductor packages, use:

```
BiocManager::install()
```

How to find Bioconductor packages of interest?

The easiest way to search Bioconductor for a topic specific package is to use the [BiocViews search](https://bioconductor.org/packages/release/BiocViews.html#___Software) (https://bioconductor.org/packages/release/BiocViews.html#___Software). `BiocViews` includes a controlled vocabulary to categorize Bioconductor packages. Because packages are tagged using this vocabulary, they can be grouped and searched by topic.

Home > [BiocViews](#)**Bioconductor version 3.18 (Release)**

Find bioViews:

Software (2266)

- AssayDomain (895)
- BiologicalQuestion (959)
- Infrastructure (553)
- ResearchField (1076)
- ShinyApps (23)
- StatisticalMethod (830)
- Technology (1466)
- WorkflowManagement (1)
- WorkflowStep (1218)
- AnnotationData (920)
- ExperimentData (429)
- Workflow (30)

Packages found under Software:

Rank based on number of downloads: lower numbers are more frequently downloaded.

Show All entriesSearch table:

Package	Maintainer	Title	Rank [▲]
BiocVersion	Bioconductor Package Maintainer	Set the appropriate version of Bioconductor packages	1
S4Vectors	Hervé Pagès	Foundation of vector-like and list-like containers in Bioconductor	2
BiocGenerics	Hervé Pagès	S4 generic functions used in Bioconductor	3
GenomeInfoDb	Hervé Pagès	Utilities for manipulating chromosome names, including modifying them to follow a particular naming style	4
IRanges	Hervé Pagès	Foundation of integer range manipulation in Bioconductor	5
Biobase	Bioconductor Package Maintainer	Biobase: Base functions for Bioconductor	6
zlibbioc	Bioconductor Package Maintainer	An R packaged zlib-1.2.5	7
XVector	Hervé Pagès	Foundation of external vector representation and manipulation in Bioconductor	8

Packages are ranked. The more popular the package, the lower the rank.

Bioconductor education and communication

Resources for learning

There are a number of Bioconductor events/conferences throughout the year including the annual BioC conference in North America and similar regional conferences throughout the world (e.g., BioC Asia, BioC Europe). Upcoming events (e.g., conferences, workshops, courses, summer schools, etc.) can be found at the bottom of the home page or in the [Events Calendar](#) (<http://www.bioconductor.org/help/events/>).

 **Events**
[See all events >](#)

EuroBioC2024: Where Software and Biology Connect

04 - 06 September 2024
Hybrid In-person Oxford (UK) and Virtual Conference

BioC2024: Where Software and Biology Connect

24 - 26 July 2024
Hybrid In-person Grand Rapids, MI and Virtual Conference

EuroBioC2023: European Bioconductor Conference

20 - 22 September 2023
Ghent, Belgium

Network Analysis in Systems Biology with R/Bioconductor

11 - 14 September 2023
Online

BioC2023: Where Software and Biology Connect

02 - 04 August 2023
Hybrid In-person Boston and Virtual Conference

Upcoming Events on the Bioconductor homepage

See the "[Learn](#)" (<http://www.bioconductor.org/>) tab or card on the Bioconductor website to find additional resources such as course materials, presentations, and vignettes.

You could also use `browseVignettes()` to search for vignettes directly from R.

Communication

For package support and questions on related topics, there is an active [Bioconductor support site](https://support.bioconductor.org/) (<https://support.bioconductor.org/>) that operates similarly to other forums (e.g., [Biostars](https://www.biostars.org/) (<https://www.biostars.org/>)).

There is also a Slack workspace for general community interaction with a range of channels. For example, important announcements are posted to the `#general` channel in Slack.

Introduction to report generation with R.

Reproducibility in science means being able to generate the same experimental / analytical results with a high degree of reliability. This is necessary for research validation, scientific and public trust, innovation, and collaboration.

Reproducibility is not possible without complete transparency and exceptional documentation of all research steps (i.e., from the lab bench to the computer).

On the other hand, reusability refers to the reuse of data, methods, or workflows either for validation or new purposes. Reusability is important for applying methods to new problems, standardizing methodologies, and advancing discovery.

Reusability is also not possible without exceptional documentation.

We can make our research more reproducible and our data and methods more reusable by documenting, documenting, and documenting more...along with other steps (e.g., version control, containerization, etc.).

There are two report generating systems built into RStudio:

1. [R Markdown](https://rmarkdown.rstudio.com/) (<https://rmarkdown.rstudio.com/>)
2. [Quarto](https://quarto.org/) (<https://quarto.org/>)

Both R Markdown and Quarto support dozens of static and interactive output formats and allow the user to execute code within a larger narrative. Because Quarto is the next generation of R Markdown, that will be the focus here.

What is Quarto?

Quarto® is an open-source scientific and technical publishing system built on Pandoc ---<https://quarto.org/> (<https://quarto.org/>)

What does this mean? Quarto allows you to combine code, commentary, and other features to tell a story about your data or data analysis using articles, presentations, dashboards, websites, blogs, or books. Click [here](https://quarto.org/docs/output-formats/all-formats.html) (<https://quarto.org/docs/output-formats/all-formats.html>) for a list of supported Pandoc output formats.

Unlike R Markdown, Quarto is NOT an R package but instead is a command line tool.

Why use Quarto

Quarto helps you tell others exactly what you did and how you derived your conclusions - code, results, and conclusions wrapped up in a single document.

Advantages of Quarto compared with other publishing systems:

- Can use with the IDE or editor of your choice: Visual Studio Code, RStudio, JupyterLab/ Jupyter notebook, other.
- Does not require R / RStudio.
- Can use directly from the command line.
- Language agnostic; can use the language of your choice (R, python, Julia, Bash, Observable) and can mix languages in a single document (R, Python, Bash, Observable).
- Easy to share with collaborators who prefer a different language or for mixed language projects.
- Better defaults; consistent syntax and approach across languages.
- Similar to RMarkdown but with fewer dependencies, greater consistency, and more flexibility.

Note

Quarto can render most RMarkdown (.Rmd) and Jupyter notebook files (.ipynb) out of the box. No edits necessary. This makes it an excellent tool for collaboration.

If you are already invested in R Markdown, you may want to stick with it. For now, there is no plan to discontinue R Markdown, but there will be no further development. BUT, if you are just getting started with documenting your data analyses and / or you are working on a highly collaborative project, Quarto is a good choice.

Gallery of examples

Let's check out some examples.

The screenshot shows the Quarto website's gallery page. At the top, there is a navigation bar with the 'quarto' logo and links for Overview, Get Started, Guide, Extensions, Reference, Gallery, Blog, and Help. The main content area is titled 'Gallery' and contains the text: 'Quarto can produce a wide variety of output formats. Here are some examples:'. Below this text is a bulleted list of output formats: Articles & Reports, Presentations, Interactive Docs, Websites, and Books. In the center of the page, there is a preview of a Quarto report. The report shows a satellite image of Earth with the text 'Engage readers with interactivity' below it. The report also includes a code block with R and Quarto commands.

[\(https://quarto.org/docs/gallery/\)](https://quarto.org/docs/gallery/)

Examples of Quarto report types from quarto.org

The [Quarto gallery](https://quarto.org/docs/gallery/) (<https://quarto.org/docs/gallery/>) includes many examples of various documentation types. Click on the link to explore more!

Getting Started

Quarto is installed with the latest versions of RStudio. When rendering a Quarto document (.qmd file), the code blocks are processed using either `knitr` or `jupyter`, which is converted to markdown. That markdown is then converted to the [final format](https://quarto.org/docs/output-formats/all-formats.html) (<https://quarto.org/docs/output-formats/all-formats.html>) using `pandoc`.

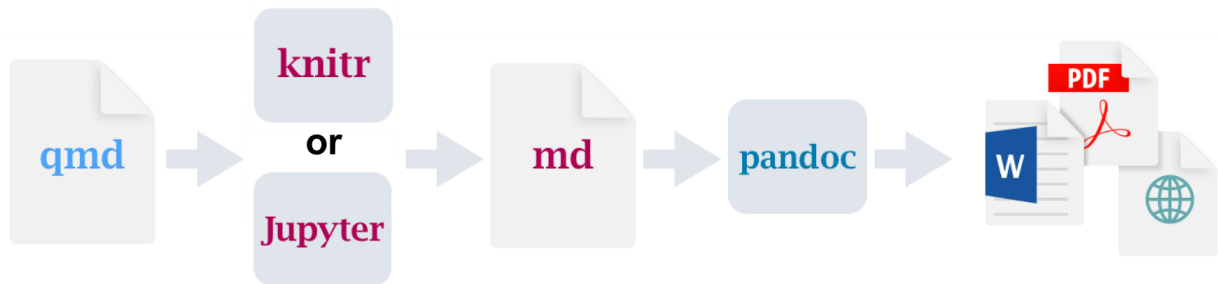


Image modified from quarto.org

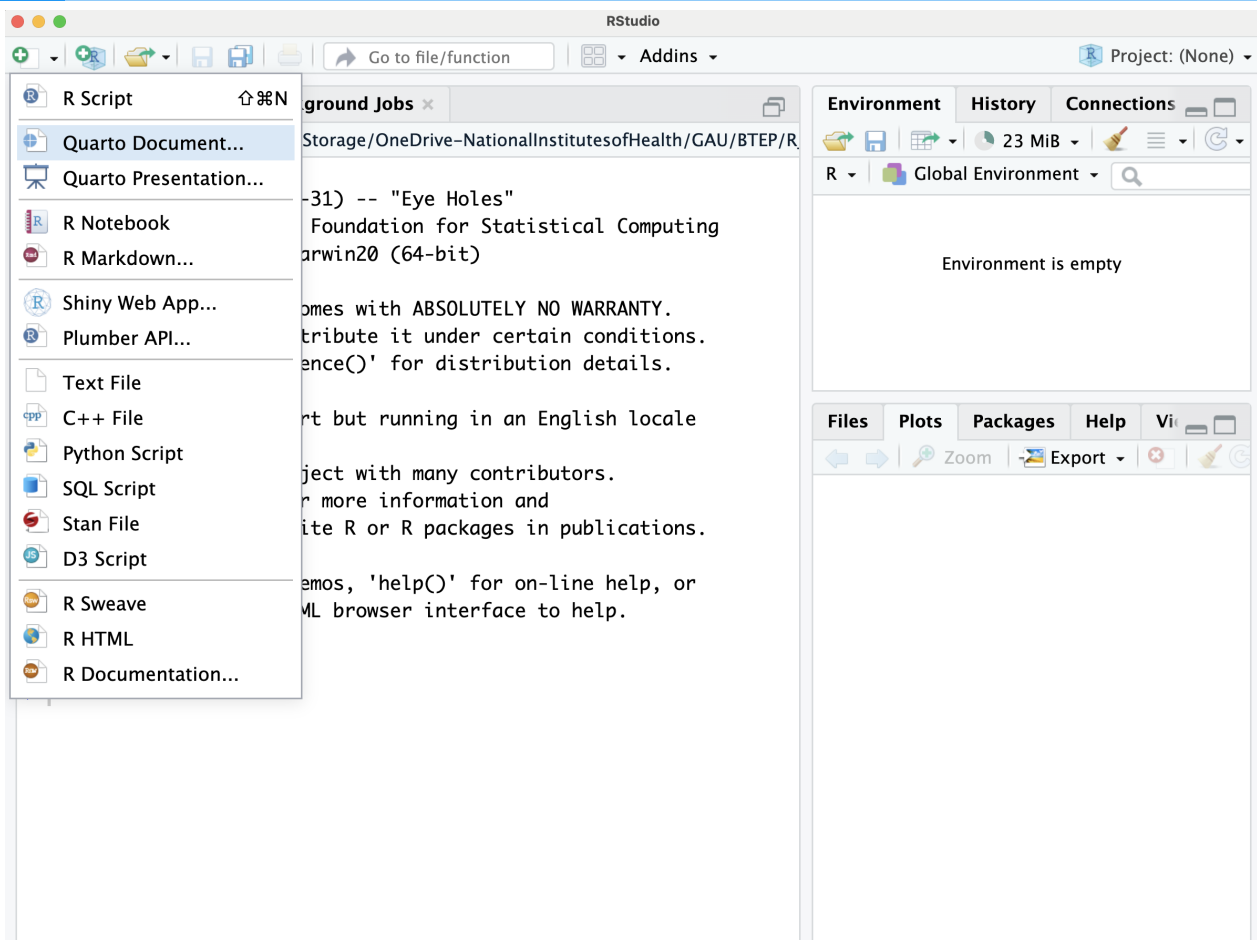
Markdown

Quarto uses [markdown](https://quarto.org/docs/authoring/markdown-basics.html) (<https://quarto.org/docs/authoring/markdown-basics.html>) for formatting text, images, links, code, and other components in plain text documents. It is helpful to know some amount of markdown to get started, but Quarto can also be used similar to word processor (using a visual editor).

Open a new .qmd file

To get started with Quarto in RStudio, navigate to:

File > New File > Quarto Document



This will open a window to easily modify initial options. Here, we can select Quarto outputs such as a document, presentation, or interactive, and the output format (e.g., for a document, html, pdf, word). We can adjust the engine (knitr or jupyter), and our choice of editor (source vs visual editor).

New Quarto Document

Document
Presentation
Interactive

Title: Volcano

Author: Alex Emmons, Ph.D.

HTML
Recommended format for authoring (you can switch to PDF or Word output anytime)

PDF
PDF output requires a LaTeX installation (e.g. <https://yihui.org/tinytex/>)

Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux)

Engine: Knitr

Editor: Use visual markdown editor ?

? [Learn more about Quarto](#)

Create Empty Document Create Cancel

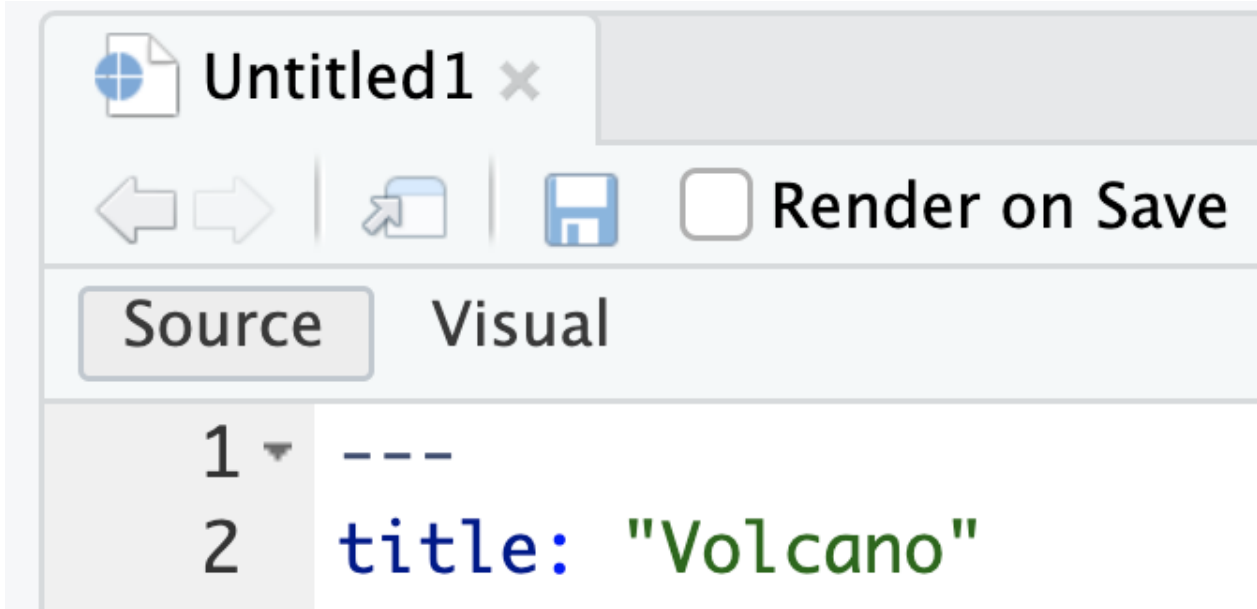
Don't know markdown? No problem. Use the Visual editor.

One of the great things about Quarto is that you do not really need to know markdown to use it. You can use a "What you see is what you mean (WYSIWYM)" editing interface. This provides an editor toolbar along with other shortcuts to enhance the editing process.

Note

The visual editor can be used along with markdown syntax. They do not need to be mutually exclusive.

You can switch between the visual editor and the source editor at the top of the document.



A new Quarto document in RStudio, will include example text to help get you started.

Anatomy of Quarto document

Once you have initiated your document, you can get started documenting your analysis.

There are three basic components to a quarto document:

1. yaml header (bracketed by ---)

The yaml header or file allows us to control document level or project level options. Here, we can specify formats, themes, executable options, and others.

2. markdown text (images, tables, text, etc.)

Your narrative including images, tables, text, and other elements can be added using markdown syntax or using the visual editor.

3. code chunks (bracketed by ```)

Code blocks can be added using ```{r}``` or ```{python}``` or ```{bash}```. Python requires the `reticulate` package when using `knitr`.

How code blocks and associated output behave can be modified using code chunk options denoted by `#|`. Many options can also be applied to all code chunks in the yaml header.

Happy documenting!

Additional Resources

If interested in Quarto, check out our recent [Coding Club session \(https://cbiit.webex.com/cbiit/ldr.php?RCID=2ea9ffe6996ea4d195c65ee141567573\)](https://cbiit.webex.com/cbiit/ldr.php?RCID=2ea9ffe6996ea4d195c65ee141567573) or navigate to [quarto.org \(https://quarto.org/\)](https://quarto.org) and check out the documentation.

Acknowledgements

Material from this lesson was either taken directly or adapted from the Intro to R and RStudio for Genomics lesson provided by [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html\)](https://datacarpentry.org/genomics-r-intro/01-introduction/index.html) and [The Bioconductor Project: Introduction to Bioconductor \(https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor.html\)](https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor.html) from the Carpentries Incubator.

Additional Exercises



Lesson 2 Exercise Questions: Base R syntax, objects, and data types

1. Let's use some functions.

a. Use `sum()` to add the numbers from 1 to 10.

{{Sdet}}

Solution{{Esum}}

```
sum(1:10)
```

{{Edet}}

b. Compute the base 10 logarithm of the elements in the following vector and save to an object called `logvec`: `c(1:10)`.

{{Sdet}}

Solution{{Esum}}

```
logvec<- log10(c(1:10))
```

{{Edet}}

c. What does the function `paste()` do? Use it to combine the following vectors. Use an `_` as a separator.


```
id <- LETTERS[1:5]  
idnum<- c(1,3,6,9,12)
```

{{Sdet}}

Solution{{Esum}}

```
paste(id, idnum, sep="_")
```

{{Edet}}

d. What does the function `identical()` do? Use it to compare the following vectors. 

```
a<-seq(2,10,by=2)
b<-c(2,4,6,8,10)
```

{{Sdet}}

Solution{{Esum}}

```
#tells us whether the two vectors are the same
identical(a,b)
```

{{Edet}}

2. What is the value of each object? You should know the value without printing the value of the object.

```
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>)

3. 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. Create an object containing the names of four different bones.

c. Create an object with values 1 to 100.

{{Sdet}}

Solution{{Esum}}

```
bone_num<- 206
bone_names<- c("talus","calcaneus","tibia","fibula")
values<-c(1:100)
```

{{Edet}}

- 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")
```

{{Sdet}}

Solution{{Esum}}

```
#These were coerced into a single data type
typeof(num_char)
num_char
typeof(num_logical)
num_logical
typeof(char_logical)
char_logical
typeof(tricky)
tricky
```

{{Edet}} (Question taken from <https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r/index.html>)

5. 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`.

{{Sdet}}

Solution{{Esum}}

```
combined <- c(num_char[2:3], char_logical[length(char_logical)],
             tricky[1])
```

{{Edet}}



Lesson 3 Exercise Questions: BaseR dataframe manipulation and factors

The `filtdownbund_scaledcounts_airways.txt` 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/) (<https://pubmed.ncbi.nlm.nih.gov/24926665/>).

We are going to use the `filtdownbund_scaledcounts_airways.txt` file for this exercise. Get the data [here](#).

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.

1. Import the `filtdownbund_scaledcounts_airways.txt` into R and save to an R object named `transcript_counts`. Try not to use the dropdown menu for loading the data.

{{Sdet}}

Solution{{Esum}}

```
transcript_counts <- read.delim("../data/filtdownbund_scaledcounts
```

{{Edet}}

2. What are the dimensions of `transcript_counts`?

{{Sdet}}

Solution{{Esum}}

```
dim(transcript_counts)
```

{{Edet}}

3. What are the column names?

{{Sdet}}

Solution{{Esum}}



```
colnames(transcript_counts)
```

{{Edet}}

4. How many categories of transcripts are there? Think about what you know regarding factors.

{{Sdet}}

Solution{{Esum}}

```
nlevels(factor(transcript_counts$transcript, exclude=NULL))
```

{{Edet}}

5. Rename the column "sample" in transcript_counts to "SampleID".

{{Sdet}}

Solution{{Esum}}

```
colnames(transcript_counts)[2] <- "SampleID"
```

{{Edet}}

6. What is the mean and standard deviation of "avgLength" across the entire transcript_counts data frame? Hint: Read the help documentation for mean() and sd().

{{Sdet}}

Solution{{Esum}}

```
mean_avgLength <- mean(transcript_counts$avgLength)
sd_avgLength <- sd(transcript_counts$avgLength)
```

{{Edet}}

7. Make a data frame with the column names "Mean" and "Standard_Dev" that holds the values from question 6. Hint: check out the function data.frame().

{{Sdet}}

Solution{{Esum}}



```
data.frame(Mean=mean_avgLength, Standard_Dev=sd_avgLength)
```

{{Edet}}



Lesson 4 Exercise Questions: Tidyverse

The `filtdownbund_scaledcounts_airways.txt` 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/) (<https://pubmed.ncbi.nlm.nih.gov/24926665/>). You can obtain the data outside of class [here](#).

The `diffexp_results_edger_airways.txt` includes results from differential expression analysis using EdgeR. You can obtain the data outside of class [here](#).

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 try to solve the problem using tidyverse.

The normalized and non-normalized count data should be saved to the object `scaled_counts`. The differential expression results should be saved to the object `dexp`.

1. Using `scaled_counts`, is there a difference in the number of transcripts with greater than 0 normalized counts ("counts_scaled") per sample? What did you use to answer this question.

{{Sdet}}

Solution{{Esum}}

```
table(scaled_counts[scaled_counts$counts_scaled>0,]$sample)
```

{{Edet}}

2. Select the following columns from the `scaled_counts` data frame: `sample`, `cell`, `dex`, `Run`, `transcript`, `avgLength`, and `counts_scaled`. However, rearrange the columns so that the column 'Run' follows 'sample' and 'avgLength' is the last column. Save this to the object `df_counts`.

{{Sdet}}

Solution{{Esum}}

```
df_counts<-scaled_counts %>%
  select(sample, Run, cell, dex, transcript, counts_scaled, avgLen
```

{{Edet}}

3. Using the differential expression results, create a data frame with the top five differentially expressed genes by p-value. Hint: 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. (**Lesson 4 challenge question**)

{{Sdet}}

Solution{{Esum}}

```
topgene<-dexp %>%
  arrange(FDR) %>%
  filter(logFC >= abs(2)) %>%
  head(5)
```

{{Edet}}

4. Filter the data frame `scaled_counts` to include only our top five differentially expressed genes (from question 3) and save to a new object named `top_gene_counts`.

{{Sdet}}

Solution{{Esum}}

```
top_gene_counts<-
  scaled_counts %>%
  filter(transcript %in% topgene$transcript)
```

{{Edet}}

5. Return a filtered data frame of the differential expression results. We want to look at only the transcripts with logCPM greater than 3 with a logFC greater than or equal to an absolute value of 2.5 and an adjusted (FDR) p-value less than 0.001.

{{Sdet}}

Solution{{Esum}}

```
dexp %>%
  filter(logCPM > 3, logFC >= abs(2.5), FDR < 0.001)
```


{{Edet}}



Lesson 5 Exercise Questions: Tidyverse

The `filtdownbund_scaledcounts_airways.txt` 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/) (<https://pubmed.ncbi.nlm.nih.gov/24926665/>). You can obtain the data outside of class [here](#).

The `diffexp_results_edger_airways.txt` includes results from differential expression analysis using EdgeR. You can obtain the data outside of class [here](#).

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 try to solve the problem using tidyverse.

The normalized and non-normalized count data should be saved to the object `scaled_counts`. The differential expression results should be saved to the object `dexp`.

1. Explore the column "avgLength" in `scaled_counts`. Does the data in this column vary within a sample? How could we figure this out if we didn't know what was in this column?

{{Sdet}}

Solution{{Esum}}

```
scaled_counts %>% group_by(sample) %>% summarize(median=median(a
                                                    max=max(avgLength) ,
                                                    min=min(avgLength))
```

{{Edet}}

2. Create a column in `scaled_counts` named "z-counts" that contains a z-score transformation of the "counts" column.

{{Sdet}}

Solution{{Esum}}

```
scaled_counts %>% mutate(z_counts=scale(counts))
```

```
{{Edet}}
```



3. Coerce the columns "sample" and "SampleName" from `scaled_counts` to type factor.

```
{{Sdet}}
```

Solution{{Esum}}

```
scaled_counts %>% mutate(across(c(sample, SampleName), as.factor
```

```
{{Edet}}
```

4. In the lesson 4 exercise, you created a data frame with the top five differentially expressed genes by p-value and logFC.

```
topgene<-dexp %>%
  arrange(FDR) %>%
  filter(logFC >= abs(2)) %>%
  head(5)
```

Create a data frame of the mean, median, and standard deviation of the normalized counts ("counts_scaled") for each of our top transcripts by treatment ("dex"). Is there a large amount of variation within a treatment?

```
{{Sdet}}
```

Solution{{Esum}}

```
scaled_counts %>%
  filter(transcript %in% topgene$transcript) %>%
  group_by(dex, transcript) %>%
  summarize(mean_counts=mean(counts_scaled),
            sd=sd(counts_scaled),
            median=median(counts_scaled))
```

```
{{Edet}}
```



Lesson 5 Exercise Questions: ggplot2

1. 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>).

{{Sdet}}

Solution{{Esum}}

- ```
a. geom_point
b. geom_line
c. geom_histogram
d. geom_bar
e. geom_bar with coord_polar
```

{{Edet}}

2. 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.

{{Sdet}}

Solution{{Esum}}

```
ggplot(mpg)+
 geom_boxplot(aes(class, hwy))
```

{{Edet}}

3. Fill each box with color by `class`.

{{Sdet}}

Solution{{Esum}}

```
ggplot(mpg)+
 geom_boxplot(aes(class,hwy,fill=class))
```



{{Edet}}

4. Reorder the boxes by the median of hwy. Hint: See `fct_reorder()` from `forcats`. Change the x and y labels.

{{Sdet}}

Solution{{Esum}}

```
ggplot(mpg)+
 geom_boxplot(aes(fct_reorder(factor(class),hwy,median),hwy,fill=
 labs(y="Miles per gallon (hwy)", x="Vehicle Class")
```

{{Edet}}

5. Visualize question two as a violin plot instead.

{{Sdet}}

Solution{{Esum}}

```
ggplot(mpg)+
 geom_violin(aes(class,hwy))
```

{{Edet}}

6. Visualize a cars engine size in liters (`displ`) versus fuel efficiency on the hwy (`hwy`).

{{Sdet}}

Solution{{Esum}}

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

{{Edet}}

7. Fit a smooth line (loess) to the data from question 6. Color the points by car class.

{{Sdet}}

Solution{{Esum}}

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

{{Edet}}

8. Visualize a histogram of hwy and facet by year. Explore the binwidth and color the bars red with a black outline.

{{Sdet}}

Solution{{Esum}}

```
ggplot(mpg)+
 geom_histogram(aes(hwy),fill="red",color="black", binwidth=5)
 facet_wrap(~year)
```

{{Edet}}



## Lesson 6 Exercise Questions: ggplot2

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.

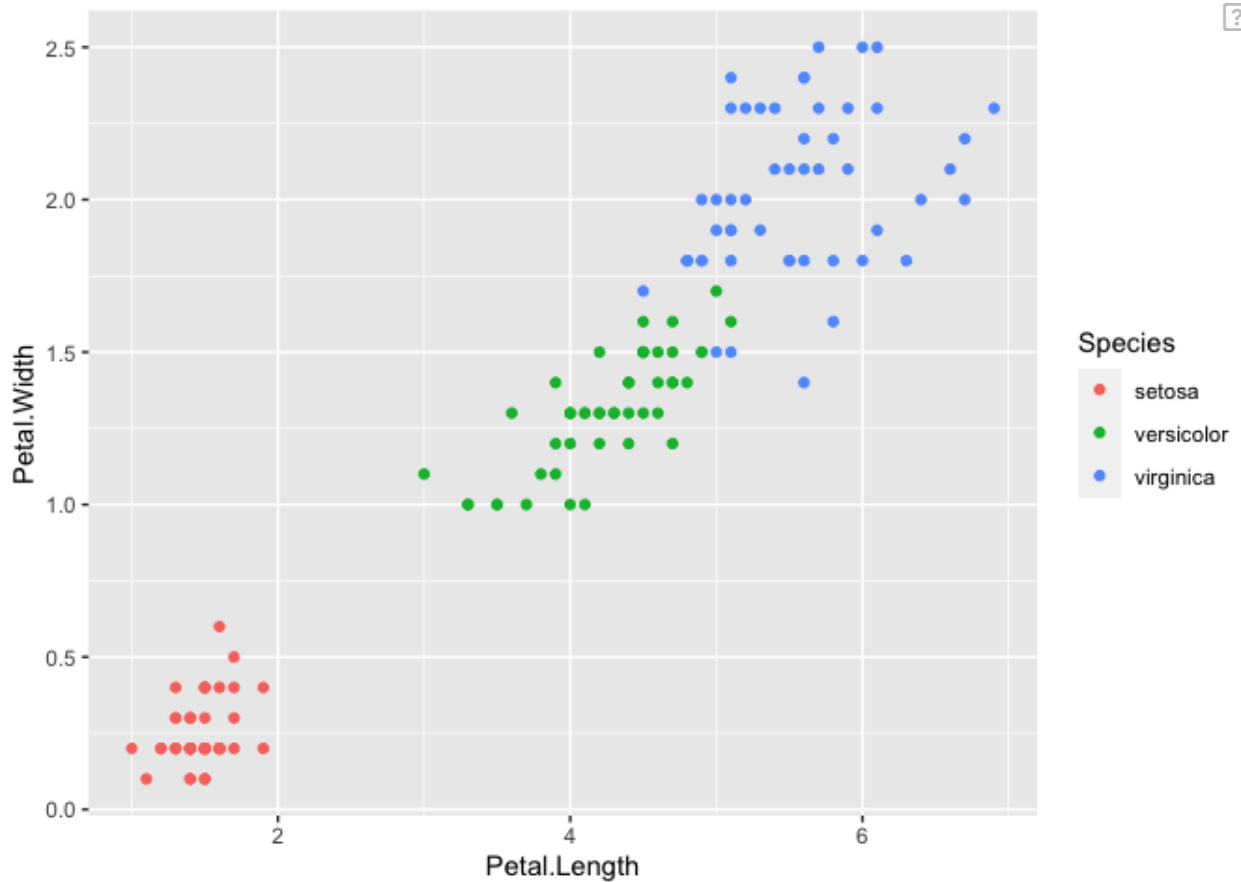
Start by plotting `Petal.Length` on the x-axis and `Petal.Width` on the y-axis.

{{Sdet}}

Solution{{Esum}}

```
library(ggplot2)
ggplot(iris)+
 geom_point(aes(Petal.Length,Petal.Width,color=Species))
```

{{Edet}}



Fix the axes so that the dimensions on the x-axis and the y-axis are equal. 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.

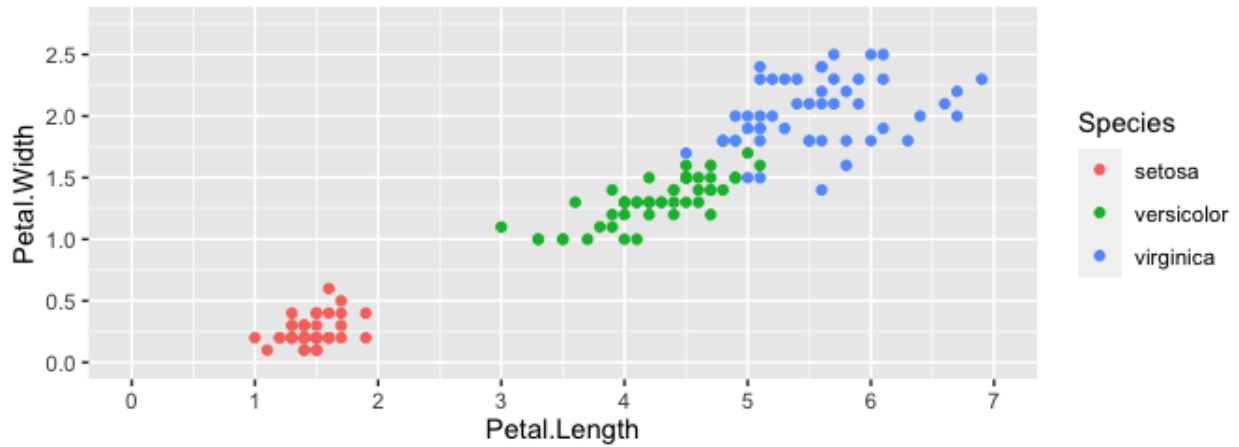
{{Sdet}}

Solution{{Esum}}

```
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=c(0,0.5,1,1.5,2,2.5)) +
 scale_x_continuous(breaks=c(0,1,2,3,4,5,6,7))
```

{{Edet}}



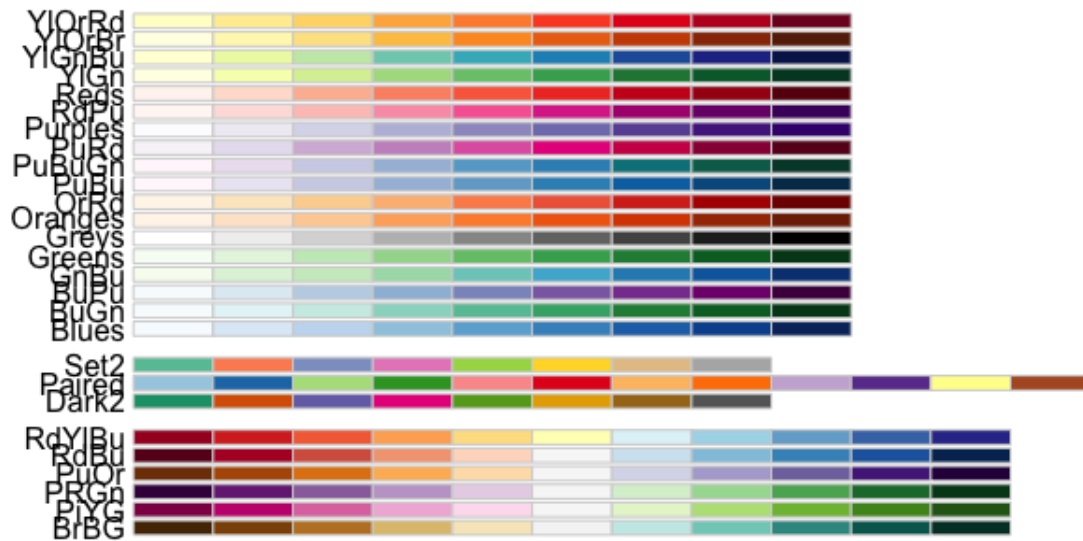


Change to color of the points by species to be color blind friendly, and change the legend title to "Iris Species". Label the x and y axis to eliminate the variable names and add unit information.

```
{{Sdet}}
```

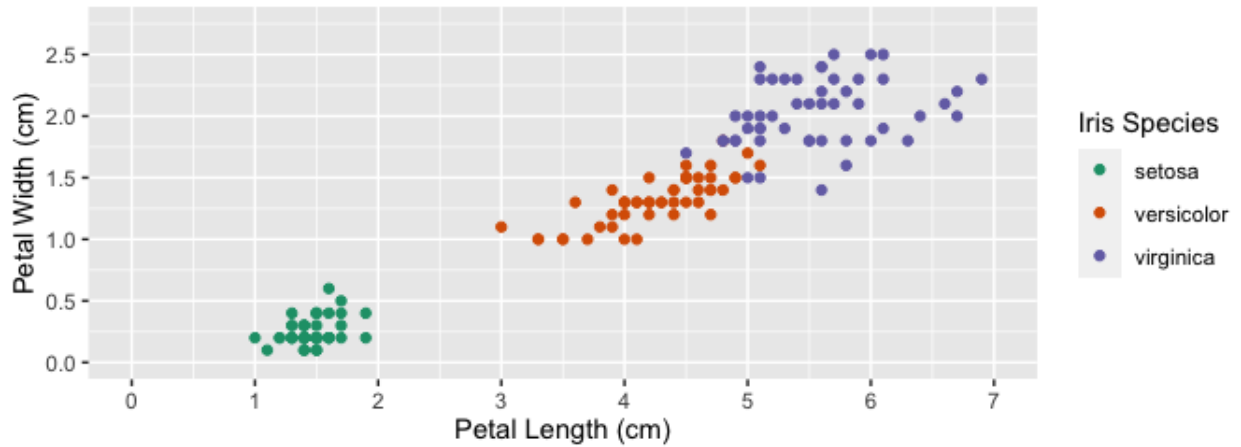
```
Solution{{Esum}}
```

```
#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=c(0,0.5,1,1.5,2,2.5)) +
 scale_x_continuous(breaks=c(0,1,2,3,4,5,6,7)) +
 scale_color_brewer(palette = "Dark2",name="Iris Species") +
 labs(x="Petal Length (cm)", y= "Petal Width (cm)")
```

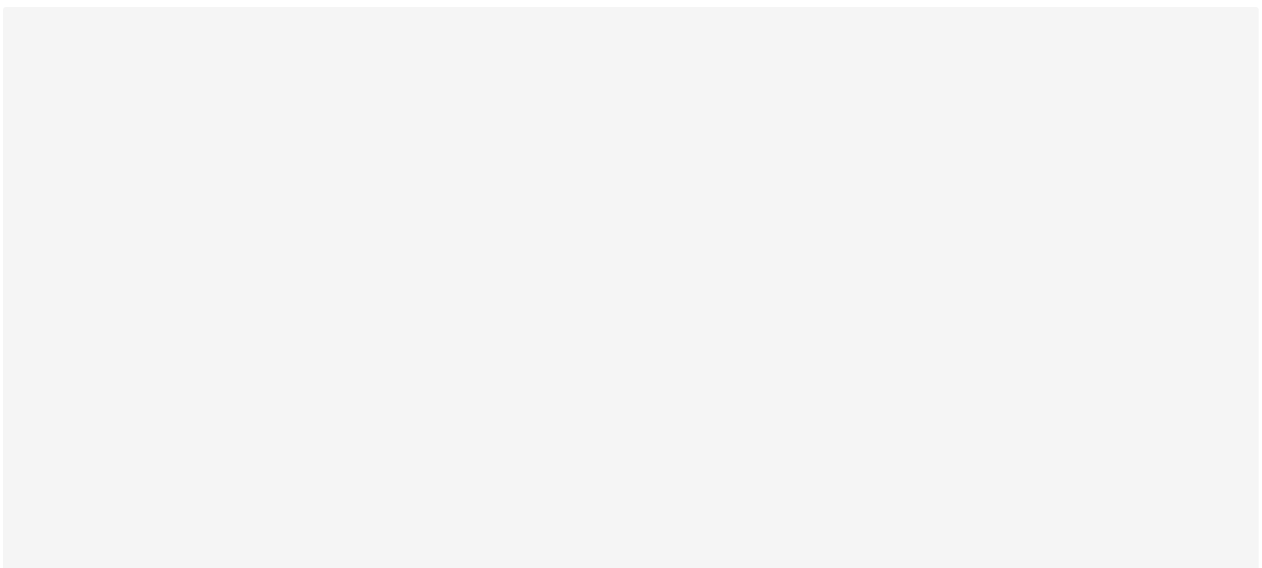
{{Edet}}



Play with the theme to make this a bit nicer. 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.

{{Sdet}}

Solution{{Esum}}

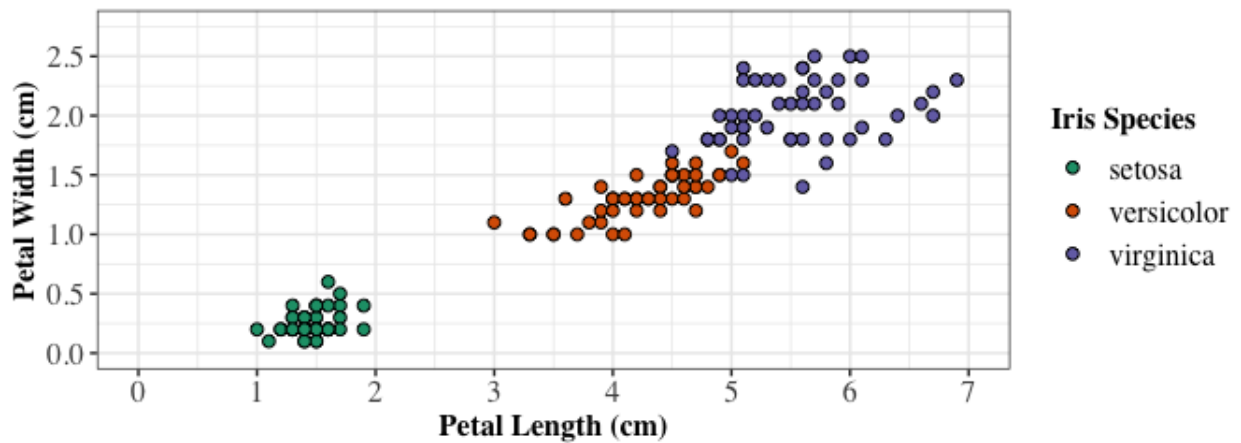


```

ggplot(iris)+
 geom_point(aes(Petal.Length,Petal.Width,fill=Species),size=2,shape=
coord_fixed(ratio=1,ylim=c(0,2.75),xlim=c(0,7)) +
scale_y_continuous(breaks=c(0,0.5,1,1.5,2,2.5)) +
scale_x_continuous(breaks=c(0,1,2,3,4,5,6,7)) +
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=
)

```

{{Edet}}



Now, save your plot using `ggsave`.

{{Sdet}}

Solution{{Esum}}

```
ggsave("iris.tiff", width=5.5, height=3.5, units="in")
```

{{Edet}}

## Data Access

Data used in this course series is available for download here: [rintro\\_data.zip](#).

**Getting help**

## Need help?

Optional help sessions are hosted on Tuesdays and Thursdays immediately following each lesson (2:00 - 3:00 pm) on Webex.

Please email us at [ncibtep@nih.gov](mailto:ncibtep@nih.gov) (<mailto:ncibtep@nih.gov>) if you have questions about the course material or you need help with a specific concept, problem, and/or project. We are happy to set up a time to meet one-on-one.

BTEP now offers [in-person office hours \(https://bioinformatics.ccr.cancer.gov/btep/introducing-btep-office-hours/\)](https://bioinformatics.ccr.cancer.gov/btep/introducing-btep-office-hours/) every 2nd Monday of the month. Join us 1-4 PM at the NIH Bethesda campus, Building 37, Suite 3041.



# References

## For Further Reading

### Books and / or Book Chapters of Interest

1. R for Data Science (<https://r4ds.had.co.nz/index.html>)
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/3-wrangling.html>)
4. The R Graphics Cookbook (<https://r-graphics.org/recipe-quick-bar>)
5. ggplot2: Elegant Graphics for Data Analysis (<https://ggplot2-book.org/index.html>)
6. Advanced R (<https://adv-r.hadley.nz/>)

### R Cheat Sheets

1. BaseR cheatsheet
2. dplyr cheatsheet
3. tidyr cheatsheet
4. ggplot2 cheatsheet
5. Other cheatsheets (<https://www.rstudio.com/resources/cheatsheets/>)

### 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/>)