

R Introductory Series

2023



Table of Contents

Course Overview

● Course Overview	10
● Welcome to the R Introductory Series 2023	10
● A series of introductory lessons in R for scientists.	10
● Course Expectations / Learning Objectives	10
● Content Organization	10
● Learning R - The Basics	10
● Data frames and Data Wrangling	10
● Data Visualization with GGplot2	11
● Bioconductor and Rmarkdown	11
● Required Course Materials	11

Learning R

Introduction to R and RStudio	13
● Introduction to R and RStudio IDE	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 a R project	15

● Creating a 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	18
● What is a path?	19
● Using functions	19
● Getting help	21
● Additional Sources for help	22
● Test your learning	23
● Acknowledgments	23

R Basics **25**

● R basics	25
● Reminder: Uploading and exporting files from RStudio Server	25
● R objects	25
● Creating and deleting objects	26
● Naming conventions and reproducibility	26
● Reassigning and deleting objects	27
● Object data types	28
● Mathematical operations	30
● Vectors	31
● Test your learning	32
● Creating, subsetting, modifying, exporting	33
● Logical subsetting	35
● Other ways to handle missing data	37
● Using objects to store thresholds	38

● Other helpful tricks	38
● Test your learning	39
● A word about lists	40
● Saving and loading objects	40
● Exporting your R project	40
● Acknowledgments	41
● Additional Resources	41

Data Frames and Data Wrangling

Introduction to data frames 43

● Data frames	43
● Reminder: Uploading files from RStudio Server	43
● Working with tabular data in R	43
● Introducing the airway data	44
● Importing / exporting data	45
● Examining and summarizing data frames	47
● Summarizing data frames	47
● What is the length of our data.frame? What are the dimensions?	49
● Other useful functions for inspecting data frames	49
● Data frame coercion and accessors	49
● Test your learning	52
● Subsetting data frames	52
● Introducing Factors	54
● Bonus: Other factor reordering.	58
● Test your learning	62
● Bonus: Find and Replace using bracket subsetting	63
● Save our data frame to a file	64

● Introduction to data matrices	64
● Exporting files from RStudio	68
● Acknowledgements	68
● Resources	68

Introduction to wrangling with Tidyverse **69**

● Introduction to Data Wrangling with the Tidyverse	69
● Reminder: Uploading files from RStudio Server	69
● Best Practices for organizing genomic data**	69
● Introducing tidy data	69
● What is tidy data?	69
● What is messy data?	70
● Tools for working with tidy data	74
● Load the tidyverse library	75
● Data wrangling with tidyverse	75
● Subsetting with dplyr	78
● Selecting columns	78
● Test your learning	79
● Filtering by row	80
● Test your learning	81
● Introducing the pipe	82
● Test your learning	83
● Mutate and transmute	84
● Test your learning	85
● Arrange, group_by, summarize	85
● Test your learning	90
● Data Reshaping	90
● Exporting files from RStudio	95
● Acknowledgements	95

● Resources	96
-------------	----

Data Visualization

Introduction to data visualization with R	98
---	----

● Data visualization with ggplot2	98
● Reminder: Uploading files from RStudio Server	98
● Why use R for Data Visualization?	98
● Introducing ggplot2	98
● The ggplot2 template	99
● Geom functions	103
● Mapping and aesthetics (aes())	104
● R objects can also store figures	107
● Colors	108
● Facets	114
● Using multiple geoms per plot	118
● Other data visualization options in R	122
● R base graphics	122
● Lattice	124
● Exporting files from RStudio	124
● Resource list	124
● Acknowledgements	125

ggplot2 Continued	126
-------------------	-----

● Objectives	126
● Reminder: Uploading files from RStudio Server	126
● Our grammar of graphics template	126

● Loading the libraries	127
● Importing the data	127
● Statistical transformations	128
● Coordinate systems	133
● Labels, legends, scales, and themes	134
● Saving plots (ggsave())	141
● Nice plot example	142
● Recommendations for creating publishable figures	143
● Complementary packages	144
● Exporting files from RStudio	144
● Acknowledgements	144

Bioconductor and Rmarkdown

● Objectives	145
● Reminder: Uploading files from RStudio Server	145
● Introducing Bioconductor	145
● Package types	145
● Bioconductor versions and install	146
● Bioconductor release schedule	146
● How to install a Bioconductor package?	147
● Bioconductor education and communication	148
● Resources for learning	148
● Communication	149
● Introducing R Markdown	149
● Creating an Rmarkdown file	150
● What is Quarto?	152
● Exporting files from RStudio	153

● Acknowledgements	153
● Resources	153

Additional Exercises

Base R: Objects, vectors, and data types 155

- Lesson 2 Exercise Questions: Base R syntax, objects, and data types 155

Base R and data frames 158

- Lesson 3 Exercise Questions: BaseR dataframe manipulation and factors 158

Practicing the Tidyverse 161

- Lesson 4 Exercise Questions: Tidyverse 161

ggplot2: Changing plot types 164

- Lesson 5 Exercise Questions: ggplot2 164

ggplot2: Making Pretty Plots 167

- Lesson 6 Exercise Questions: ggplot2 167
 - Start by plotting `Petal.Length` on the x-axis and `Petal.Width` on the y-axis. 167
 - 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. 168
 - 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. 169
 - 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. 171
 - Now, save your plot using `ggsave`. 172

Getting the Data

-
- [Data Access](#) 174

Installing R & RStudio

-
- [Installing R & RStudio](#) 175

DNAexus

[Navigating DNAexus](#) 177

-
- [Instruction for using DNAexus for the Intro to R class](#) 177

[DNAexus Outside of Class](#) 181

-
- [Setting up the R-Studio environment outside of Class hours](#) 181
 - [Log into DNAexus](#) 181
 - [To Start the Application](#) 182
 - [Open the R-Studio Web interface](#) 184
 - [Proceed with running content from the R Class](#) 186

Getting help

[Need help?](#) 188

References

For Further Reading	190
● Books and / or Book Chapters of Interest	190
● R Cheat Sheets	190
● Other Resources	190

Course Overview

Welcome to the R Introductory Series 2023

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 / Learning Objectives

The course will include a series of eight lessons taught in one to one hour and 15 minute blocks over four weeks. Lessons will be on Mondays and Wednesdays at 1 pm. Each lesson will be followed by an optional 45 minute 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/) (<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

Learning R - The Basics

Here, we will introduce R and RStudio. Learners will explore the RStudio interactive development environment (IDE) and begin to use functions and assign objects. By the end of this section, learners should understand how to work within the RStudio environment to create R projects and R scripts, navigate between directories, use functions, obtain help, and work with basic objects such as vectors.

Data frames and Data Wrangling

Here, we will learn how to store and work with tabular data in R. Learners will become acquainted with the basics of data frame manipulation including importing, cleaning, transforming, and exporting data. For data wrangling, the focus will be on the R tidyverse collection of packages.

Data Visualization with GGplot2

In this section, we will learn how to create publishable figures using the R (ggplot2) package. This includes an introduction to mapping and aesthetics, building plots iteratively, and improving plot readability. Additional packages for colors, themes, and statistics integration will be demonstrated.

Bioconductor and Rmarkdown

Here, we will review major concepts taught in the first three sections. We will explore R Markdown functionality, to help learners generate shareable, professional, and reproducible data analysis reports. We will also introduce Bioconductor, including how to install and search for packages and how to get help.

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).

Learning R

Introduction to R and RStudio

Introduction to R and RStudio IDE

Objectives

To understand:

1. the difference between R and RStudio IDE.
2. how to work within the RStudio environment including:
 - creating an Rproject and Rscript
 - navigating between directories
 - using functions
 - obtaining help
3. 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 18,944 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 3.16) includes 2,183 software packages, 416 experiment data packages, 909 annotation packages, 28 workflows, and 8 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). 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****

Detailed Instructions for installing R and RStudio can be found [here]

Getting Started with R and R Studio

This tutorial closely follows the Intro to R and RStudio for Genomics lesson provided by [datacarpentry.org \(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html\)](https://datacarpentry.org/genomics-r-intro/01-introduction/index.html).

Creating a 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.

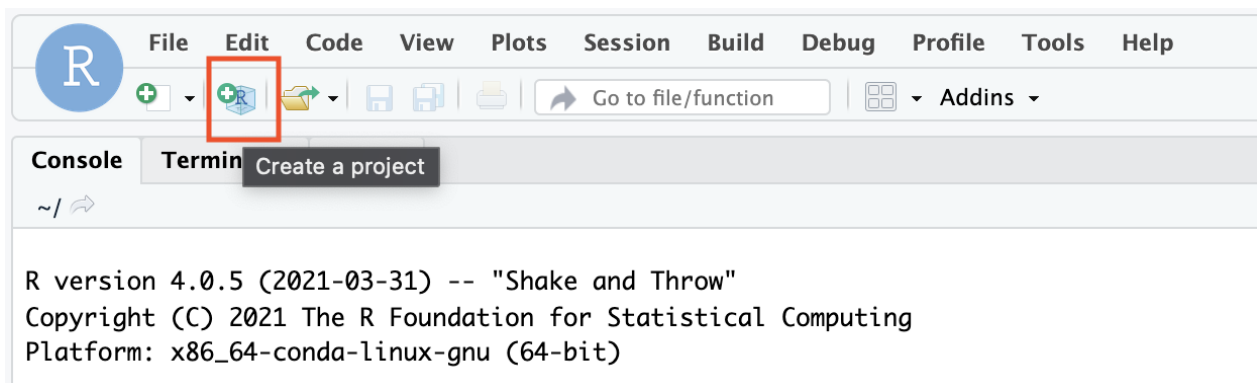
Creating an R project for each project you are working on facilitates organization and scientific reproducibility.

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

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

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



A New project wizard will appear. Click `New Directory` and `New Project`. Choose a new directory name....perhaps "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/01-introduction/index.html>)

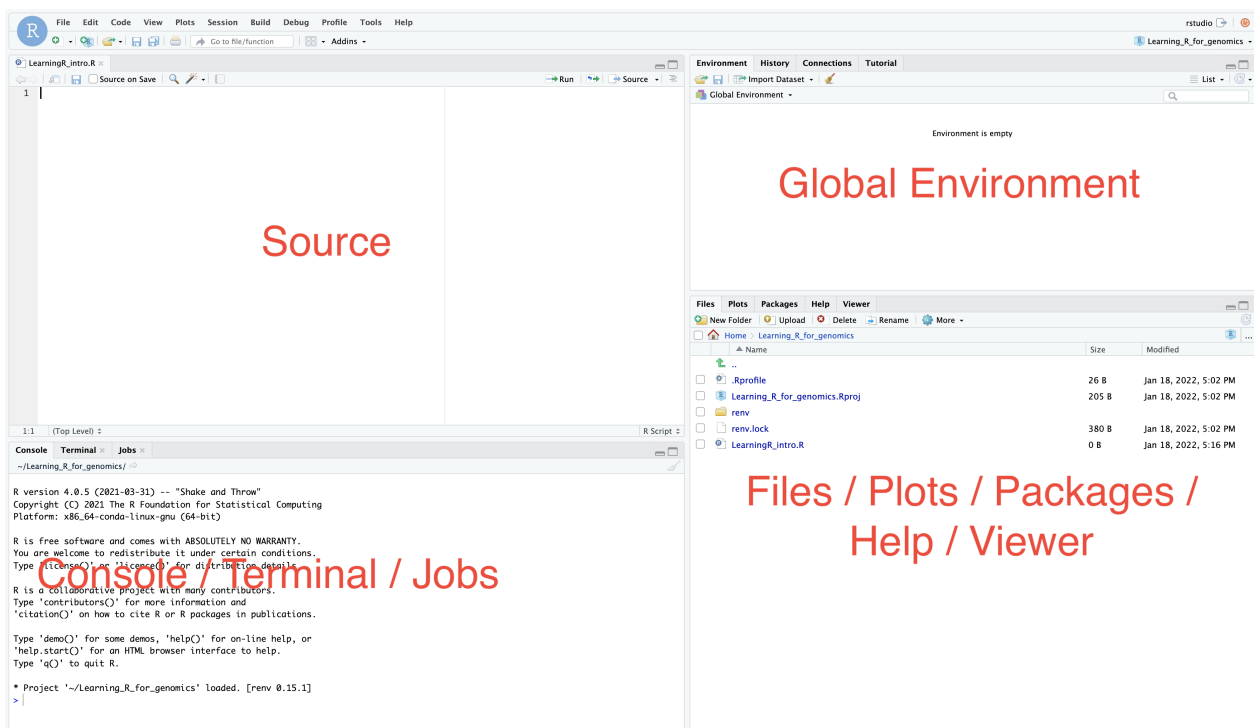
Read more about `renv` [here](https://rstudio.github.io/renv/articles/renv.html) (<https://rstudio.github.io/renv/articles/renv.html>).

Creating a 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. (demonstrate minimize / maximize utility)

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/01-introduction/index.html>)

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

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

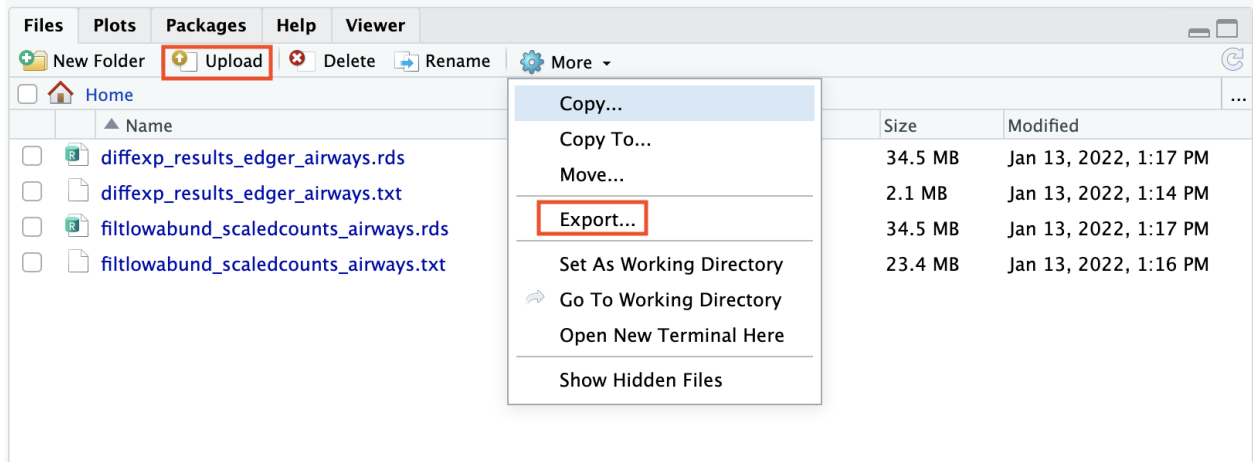
Also, you can change your RStudio layout. See this [blog \(https://www.r-bloggers.com/2018/05/a-perfect-rstudio-layout/\)](https://www.r-bloggers.com/2018/05/a-perfect-rstudio-layout/) if you are interested. For simplicity, please do NOT change the layout during this course.

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.



Data Management

Data organization is extremely important to reproducible science. Consider organizing your project directory in a way that facilitates reproducibility. For example, you may want directories for data, drafts_documents, outputs, and scripts. See additional details in [this lesson \(https://carpentries-incubator.github.io/bioc-intro/20-r-rstudio/index.html\)](https://carpentries-incubator.github.io/bioc-intro/20-r-rstudio/index.html) from Data Carpentries. 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.

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()`.

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/Rlearning")
```

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 \(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html\)](https://datacarpentry.org/genomics-r-intro/01-introduction/index.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 round(x, digits = 0)
```

```
`` {r}
```

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

```
````
```

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

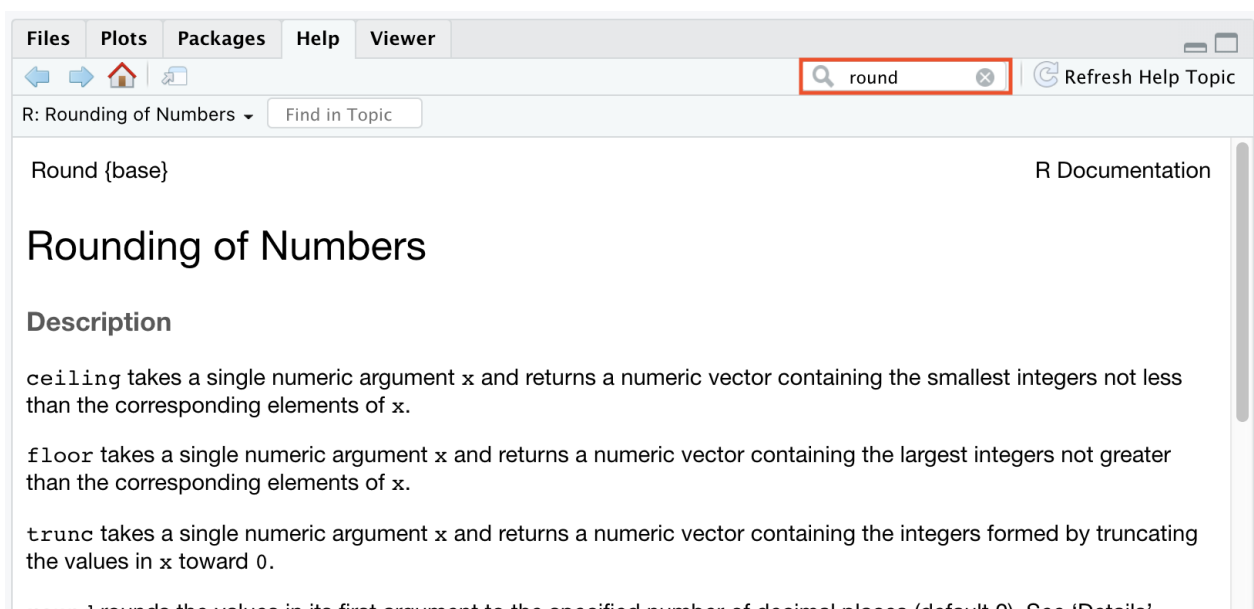
```
[1] 5.7
```

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()`.

## 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).



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` 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/>) package particularly useful.

To provide details about your R session, use

```
sessionInfo()
```

```
R version 4.1.2 (2021-11-01)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Big Sur 10.16
##
Matrix products: default
BLAS: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib
LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib
```



```
##
locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
attached base packages:
[1] stats graphics grDevices utils datasets methods ba
##
loaded via a namespace (and not attached):
[1] digest_0.6.29 R6_2.5.1 jsonlite_1.8.0 magrittr_2.0
[5] evaluate_0.15 stringi_1.7.6 rlang_1.0.2 cli_3.3.0
[9] rstudioapi_0.13 jquerylib_0.1.4 bslib_0.3.1 rmarkdown_2.1
[13] tools_4.1.2 stringr_1.4.0 xfun_0.36 yaml_2.3.5
[17] fastmap_1.1.0 compiler_4.1.2 htmltools_0.5.2 knitr_1.41
[21] sass_0.4.1
```

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

# R Basics

## R basics

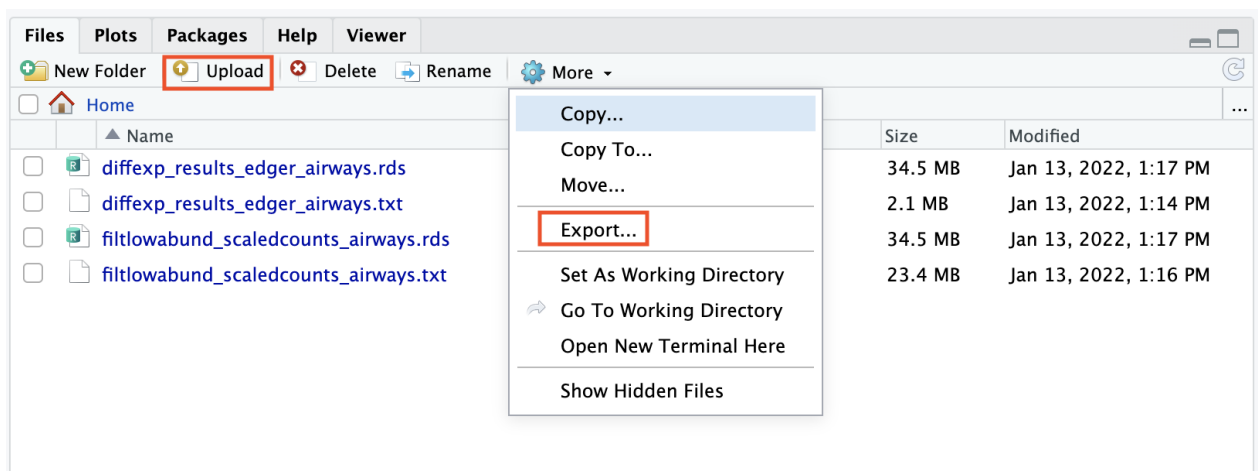
*Objectives:* To understand some of the most basic features of the R language including:

- 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.

## Reminder: Uploading and exporting files from RStudio Server

Any files created by you today will be erased at the end of the session. You can upload any files you downloaded from the last session using the Upload option in the Files pane.



## R objects

Everything assigned a value in R is technically an object in which a method (or function) can act on. Therefore, objects are data structures with specific attributes and methods that can be applied to them. They 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. 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: `cmd+Enter`), or in type directly in 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.

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

## Reassigning and deleting objects

To reassign an object, simply overwrite the object.

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

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

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

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

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

To delete an object from memory:

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

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

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

## Object data types

R objects have certain attributes, and these attributes will be important for how they can interact with certain methods / functions. Understanding the mode (storage type) or the 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                                                                                                                                                                                                              |

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/02-r-basics/index.html) (<https://datacarpentry.org/genomics-r-intro/02-r-basics/index.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 object 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%20obje](https://stats.stackexchange.com/questions/3212/mode-class-and-type-of-r-objects#:~:text=class%20is%20an%20attribute%20of,physical%20characteristic%20of%20an%20obje)

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 \(https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html\)](https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html) shows many of the mathematical operators used in R.



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

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 \(https://datacarpentry.org/genomics-r-intro/02-r-basics/index.html\)](https://datacarpentry.org/genomics-r-intro/02-r-basics/index.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 objects such as matrices and data frames, which will be discussed later.

```
str(transcript_names) #this will return propoerties of the object's t
```

```
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 \(https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r/index.html\)](https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r/index.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" "A
```

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                |

### Comparison Operator Description

|       |         |
|-------|---------|
| ==    | 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
```

## Other helpful tricks

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. 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 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]`

- c. `Total_subjects[Total_subjects < 678]`
- d. `Total_subjects[Total_subjects < 678]`

```
{{Sdet}}
```

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

```
D
```

```
{{Edet}}
```

## A word about lists

Data can also be stored in lists, which include multiple types / modes of data. You may receive output at some point in the form of a list. For a brief introduction to lists, see this nice tutorial on [towards data science \(https://towardsdatascience.com/introduction-to-lists-in-r-ff6469e6ca79\)](https://towardsdatascience.com/introduction-to-lists-in-r-ff6469e6ca79).

## 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.

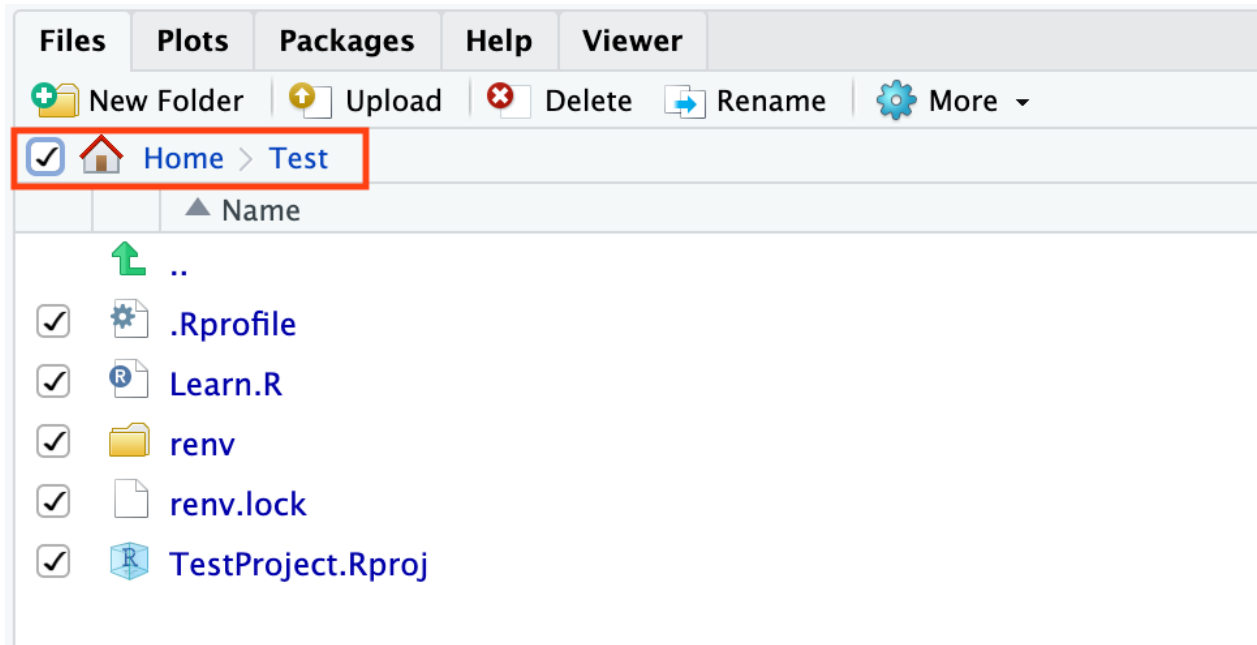
```
saveRDS(transcript_counts, "transcript_counts.rds")
```

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

Remember, because we are using RStudio server through DNAnexus, any files created by you today will be erased at the end of the session.

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/) (<https://rstudio-education.github.io/hopr/>)

# **Data Frames and Data Wrangling**

# Data frames

## Objectives

To be able to load, explore, and access data in a tabular format. To this end, students should understand the following:

1. how to import and export data
2. how to create, summarize, and subset data frames
3. what is a factor and why do we care?
4. how does a data frame differ from a matrix?

## Reminder: Uploading files from RStudio Server

Any files created by you today will be erased at the end of the session. You can upload any files you downloaded from the last session using the `Upload` option in the Files pane.

## Working with tabular data in R

In genomics, we work with a lot of tabular data (and non-tabular data). An old school method of working with this data may be to open in excel and manually work with the data. 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 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 analytical techniques 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 \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html\)](https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html)

For large genomic data sets, I recommend having 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 \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html\)](https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.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 \(https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.html\)](https://datacarpentry.org/genomics-r-intro/03-basics-factors-dataframes/index.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](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:

[filllowabund\\_scaledcounts\\_airways.txt](#) Includes scaled transcript count data

[diffexp\\_results\\_edger\\_airways.txt](#) Includes results from differential expression analysis using EdgeR.

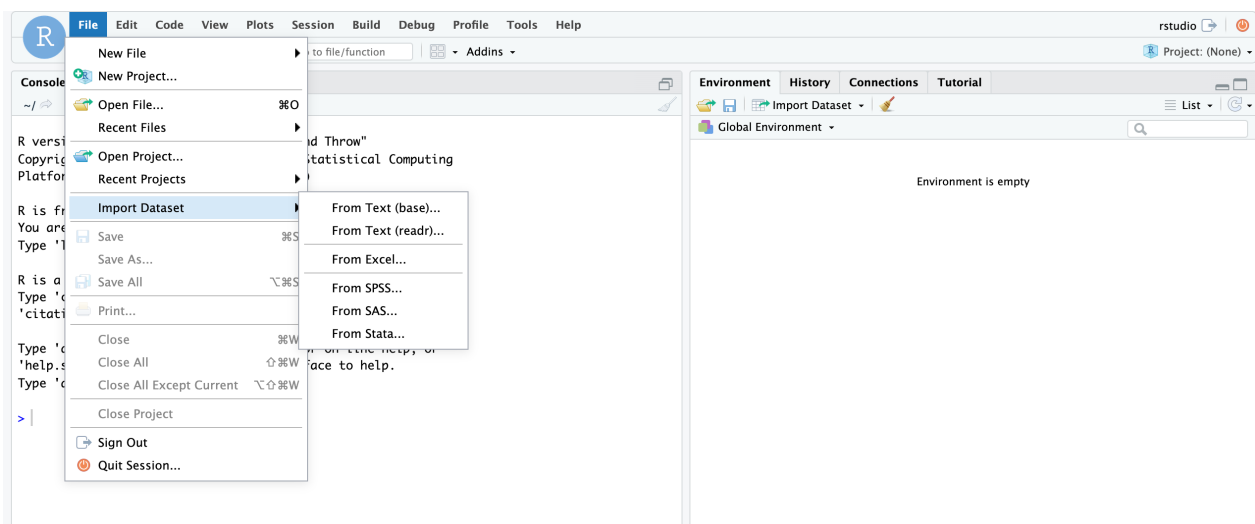
Object (.rds) files have also been included.

Note: Bioconductor will be discussed further in Lesson 4.

## 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 interface. At the top, there are tabs for 'Environment', 'History', 'Connections', and 'Tutorial'. Below the tabs, there are icons for file operations and a memory usage indicator showing '676 MiB'. The 'Environment' pane is active, displaying a search bar and a list of objects. One object, 'scaled\_counts', is selected and highlighted. Below the object name, it says '127408 obs. of 18 variables'. The main workspace area is currently empty.

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    | BioSample |       |
|---------|------------------|--------|------------|------------|--------|-------|-------|------------|------------|-----------|-----------|-------|
| 1       | ENSG00000000003  | 508    | 679        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 2       | ENSG000000000419 | 508    | 467        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 3       | ENSG000000000457 | 508    | 260        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 4       | ENSG000000000460 | 508    | 60         | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 5       | ENSG000000000971 | 508    | 3251       | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 6       | ENSG00000001036  | 508    | 1433       | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 7       | ENSG00000001084  | 508    | 519        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 8       | ENSG00000001167  | 508    | 394        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 9       | ENSG00000001460  | 508    | 172        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 10      | ENSG00000001461  | 508    | 2112       | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 11      | ENSG00000001497  | 508    | 524        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 12      | ENSG00000001561  | 508    | 71         | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 13      | ENSG00000001617  | 508    | 555        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 14      | ENSG00000001629  | 508    | 1660       | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 15      | ENSG00000001630  | 508    | 59         | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 16      | ENSG00000001631  | 508    | 729        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 17      | ENSG00000002016  | 508    | 201        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 18      | ENSG00000002330  | 508    | 206        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 19      | ENSG00000002549  | 508    | 1459       | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 20      | ENSG00000002586  | 508    | 7507       | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 21      | ENSG00000002746  | 508    | 151        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |
| 22      | ENSG00000002822  | 508    | 411        | GSM1275862 | N61311 | untrt | untrt | SRR1039508 | 126        | SRX384345 | SRS508568 | SAMN0 |

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 (`scaled_counts`) that we imported is a data frame. Data frames hold tabular data, like much of the data stored in spreadsheets. They 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.

### Summarizing data frames

Let's learn a bit more about our data frame. First, we can learn more about the structure of our data using `str()`.

```
str(scaled_counts)
```

```
'data.frame': 127408 obs. of 18 variables:
$ feature : chr "ENSG000000000003" "ENSG000000000419" "ENSG000000000004" ...
$ 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. :0.95
Class :character Class :character TRUE:127408 1st Qu.:0.95
Mode :character Mode :character Median :1.00
Mean :1.00
3rd Qu.:1.00
Max. :1.00
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.

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

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

```
[1] 18
```

```
#dimensions
dim(scaled_counts) #dim() returns the rows and columns
```

```
[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/index.html)", *Introduction to data analysis with R and Bioconductor* (<https://carpentries-incubator.github.io/bioc-intro/25-starting-with-data/index.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 vector.

We can access a column of our data frame using `[]`, `[[[]]`, or using the `$` (<http://adv-r.had.co.nz/Subsetting.html>).

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()`). More about factors later.

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
```

```
[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 the column "Sample" to "Accession".

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

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

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

```
#let's recheck our column names
colnames(scaled_counts)
```

```
[1] "feature" "sample" "counts" "SampleName"
[5] "cell" "dex" "albut" "Run"
[9] "avgLength" "Experiment" "Accession" "BioSample"
[13] "transcript" "ref_genome" ".abundant" "TMM"
[17] "multiplier" "counts_scaled"
```

## Test your learning

Which of the following will NOT print the "Run" column from scaled\_counts?

- scaled\_counts\$Run
- scaled\_counts["Run"]
- scaled\_counts[8,]
- scaled\_counts[8]

{{Sdet}}

Solution{{Esum}}

C

{{Edet}}

What is the column index for "avgLength" from the scaled\_counts df?

- 3
- 8
- 12
- 9

{{Sdet}}

Solution{{Esum}}

D {{Edet}}

## Subsetting data frames

**Subsetting** a data frame is similar to subsetting a vector; we will 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. We will discuss matrices in a bit.

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 annotation
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]
```

Some 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)

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.

## Introducing Factors

At this point, you have seen the term "factor" pop up a few times.

Factors can be thought of as vectors which are specialized for categorical data. Given R's specialization for statistics, this makes sense since categorical and continuous variables are usually treated differently. Sometimes you may want to have data treated as a factor, but in other cases, this may be undesirable. --- [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)

Let's go ahead and coerce some of our character vectors to factors. "sample", "dex", "cell", and "transcript" are categorical variables.

First, let's create subset our data to only include the columns `sample`, `counts`, `scaled_counts`, `transcript`, and `dex`.

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

str(sscaled)
```

```
'data.frame': 127408 obs. of 6 variables:
$ sample : chr "508" "508" "508" "508" ...
$ cell : chr "N61311" "N61311" "N61311" "N61311" ...
$ dex : chr "untrt" "untrt" "untrt" "untrt" ...
$ transcript : chr "TSPAN6" "DPM1" "SCYL3" "C1orf112" ...
$ counts : int 679 467 260 60 3251 1433 519 394 172 2112
$ counts_scaled: num 960.9 660.9 367.9 84.9 4600.7 ...
```

```
sscaled$sample<-as.factor(sscaled$sample)
sscaled$dex<-as.factor(sscaled$dex)
sscaled$transcript<-as.factor(sscaled$transcript)
sscaled$cell<-as.factor(sscaled$cell)
#note there is an easier solution using tidyverse functionality

#Now let's look at the structure of our data frame
str(sscaled)
```

```
'data.frame': 127408 obs. of 6 variables:
$ sample : Factor w/ 8 levels "508","509","512",...: 1 1 1 :
$ cell : Factor w/ 4 levels "N052611","N061011",...: 4 4 4
```



```
$ dex : Factor w/ 2 levels "trt","untrt": 2 2 2 2 2 2 2
$ transcript : Factor w/ 14575 levels "A1BG-AS1","A2M",...: 1306
$ counts : int 679 467 260 60 3251 1433 519 394 172 2112
$ counts_scaled: num 960.9 660.9 367.9 84.9 4600.7 ...
```

```
#get a summary
#notice that counts of the factor levels are returned
summary(sscaled)
```

```
sample cell dex transcript
508 :15926 N052611:31852 trt :63704 ABHD17AP1: 16
509 :15926 N061011:31852 untrt:63704 ACBD6 : 16
512 :15926 N080611:31852 AGAP9 : 16
513 :15926 N61311 :31852 CBWD6 : 16
516 :15926 FAM106A : 16
517 :15926 (Other) :116648
(Other):31852 NA's : 10680
counts counts_scaled
Min. : 0 Min. : 0.0
1st Qu.: 66 1st Qu.: 95.4
Median : 310 Median : 445.8
Mean : 1376 Mean : 1933.7
3rd Qu.: 960 3rd Qu.: 1369.6
Max. :513766 Max. :632885.3
##
```

We see that "sample" is a factor with 8 levels; cell is a factor with 4 levels, dex a factor with 2 levels, and transcript a factor with 14,575 levels. The levels are the different groups or categories in those variables. R will organize the levels alphabetically by default.

For the sake of efficiency, R stores the content of a factor as a vector of integers, with an integer assigned to each of the possible levels. ---[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)

This explains the numbers following the level names in the `str()` output.

This also results in some interesting behavior during variable coercion from `is.factor()` to `is.numeric()`. To coerce from a factor to a numeric, you have to first convert to a character vector. Otherwise, the numbers that you want to be numeric (the factor level names) will be returned as integers.

For example

```
#let's convert "sample" back to a numeric
head(as.numeric(sscaled$sample)) #notice the ones
```

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

```
#instead we need to do the following
head(as.numeric(as.character(sscaled$sample)))
```

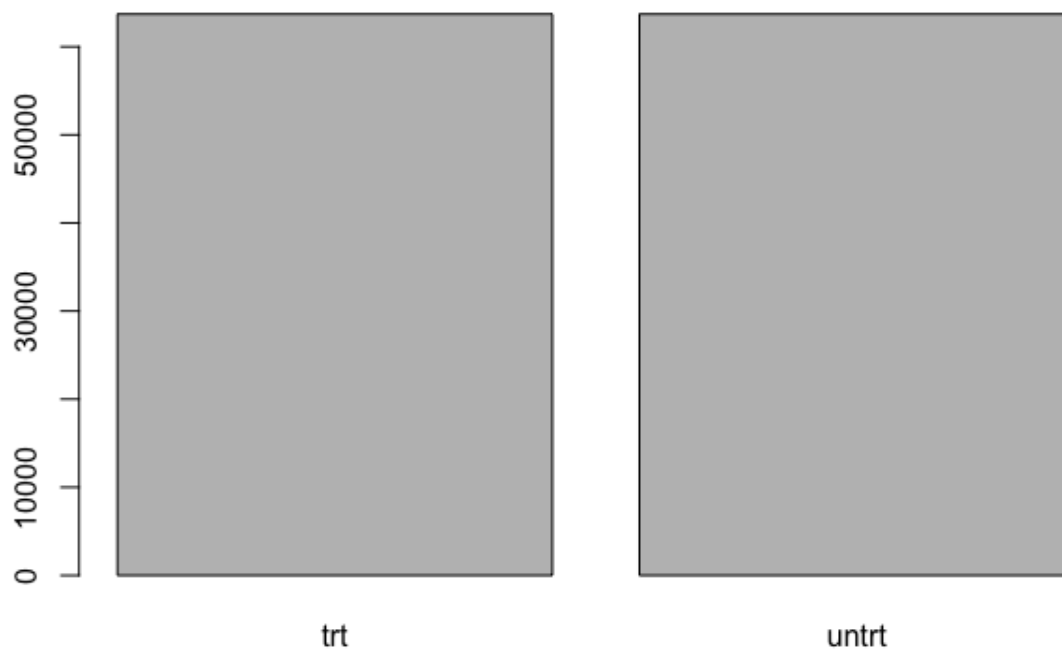
```
[1] 508 508 508 508 508 508
```

Factors are necessary for plotting, and you can rename factors simply by manipulating the levels. For example

```
#Let's rename the levels in the dex column
#first let's see the levels
levels(sscaled$dex)
```

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

```
#notice the column order is alphabetical
plot(sscaled$dex)
```

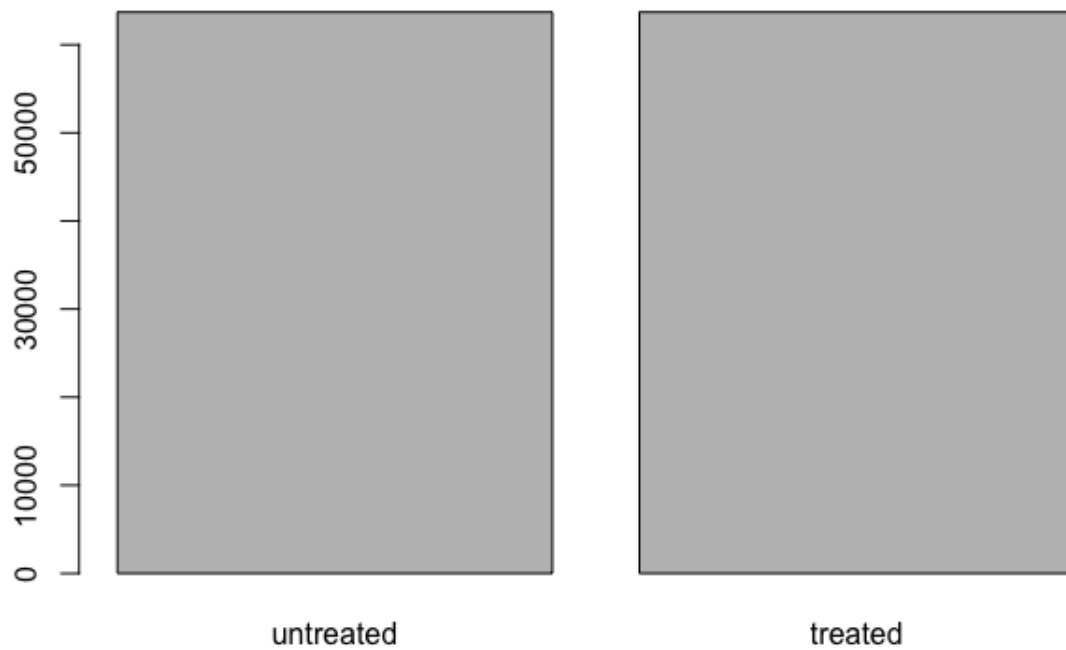


```
#Now, let's rename
#It is critical that the order is maintained
levels(sscaled$dex)<-c("treated","untreated")
levels(sscaled$dex) #We can see that the levels were modified
```

```
[1] "treated" "untreated"
```

To reorder the factor levels, we need to use `factor()` or `fct_reorder()` from `tidyverse`.

```
#We can explicitly state the order
sscaled$dex<-factor(sscaled$dex, levels=c("untreated","treated"))
plot(sscaled$dex) #now the order has changed
```



### Bonus: Other factor reordering.

{{Sdet}}

Bonus content{{Esum}}

Let's say we filtered out all scaled\_counts less than 10,000 and now we want to order our transcripts by transcripts found with at least 10k counts in the greatest number of samples (present at greater than 10k counts in more samples).

```
#let's get this filtered data set
#hopefully this notation seems familiar
transcript_f<-sscaled[sscaled$counts_scaled>10000,]

#Let's look at this using head
head(table(transcript_f$transcript),n=10)
```

```
##
A1BG-AS1 A2M A2M-AS1 A4GALT AAAS AACS AADAT
0 7 0 0 0 0
AAK1 AAMDC
0 0
```

```
#We shouldn't really see zeros at this point.
#A transcript should be found in at least one sample.
#This is because the factor levels are maintained.

#Let's drop the factor levels
transcript_f<-droplevels(transcript_f)
```

```
#head again
head(table(transcript_f$transcript),n=10)
```

```
##
A2M ABCA1 ABI3BP ACTB ACTG1 ACTN1 ACTN4 ADAM33
7 6 4 8 8 8 8 3
```

```
#let's look at our current factor order
head(levels(transcript_f$transcript),n=10)
```

```
[1] "A2M" "ABCA1" "ABI3BP" "ACTB" "ACTG1" "ACTN1" '
[8] "ADAM33" "ADAM9" "ADAMTS1"
```

```
#let's reorder
ordered_factor_transcripts <-
 factor(transcript_f$transcript,
 levels=names(sort(table(transcript_f$transcript),
 decreasing=TRUE)))

#Let's compare the output of sort to our new factor levels
head(sort(table(transcript_f$transcript),decreasing=TRUE), n=10)
```

```
##
ACTB ACTG1 ACTN1 ACTN4 ADAM9 ADH1B ADM AHNAK AKAP12 AKI
8 8 8 8 8 8 8 8 8 8
```

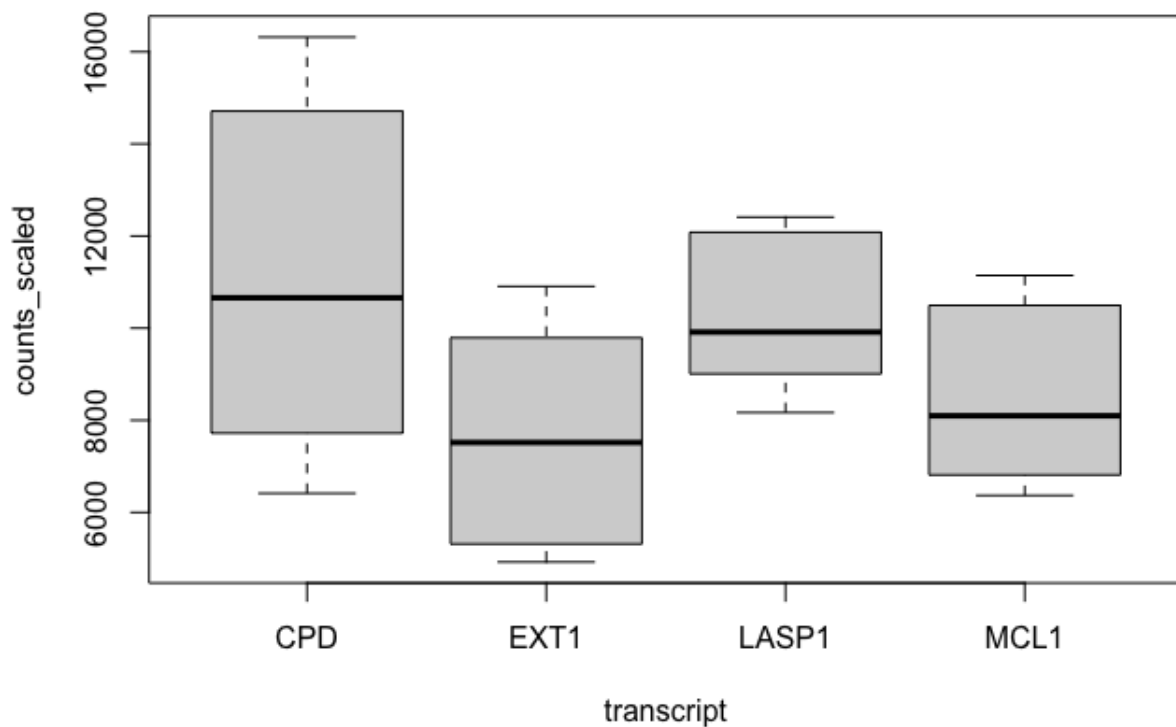
```
head(levels(ordered_factor_transcripts), n=10)
```

```
[1] "ACTB" "ACTG1" "ACTN1" "ACTN4" "ADAM9" "ADH1B" "ADM"
[9] "AKAP12" "AKR1C1"
```

Using tidyverse functionality (`library(forcats)`) we can easily reorder our factor levels by sorting along a different column using `fct_reorder()`. You should also look into `fct_relevel()`.

```
#Let's filter our data to only include 4 transcripts of interest
keep_t<-c("CPD","EXT1","MCL1","LASP1")
interesting_trnsc<-sscaled[sscaled$transcript %in% keep_t,]
interesting_trnsc<-droplevels(interesting_trnsc)

#Look at a basic boxplot of the scaled_counts of these transcripts
boxplot(counts_scaled ~ transcript, data=interesting_trnsc)
```

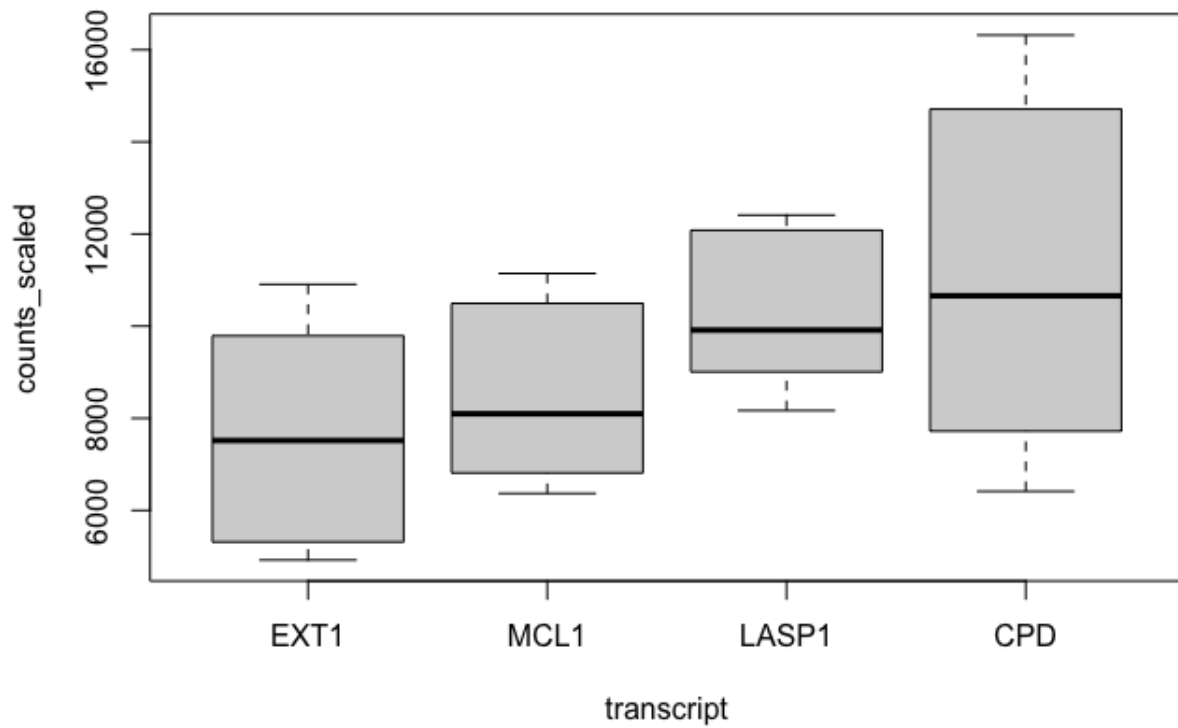


```
#print levels
levels(interesting_trnsc$transcript)
```

```
[1] "CPD" "EXT1" "LASP1" "MCL1"
```

```
#Reorder the transcript factor levels by the maximum of counts_scaled
interesting_trnsc$transcript<-
 forcats::fct_reorder(interesting_trnsc$transcript,
 interesting_trnsc$counts,max)

#plot
boxplot(counts_scaled ~ transcript, data=interesting_trnsc)
```



{{Edet}}

## Test your learning

How many categories or levels are there in `sscaled$cell`?

- a. 4
- b. 2
- c. 7
- d. 1

{{Sdet}}

Solution{{Esum}}

A {{Edet}}

What are the dimensions of `scaled_counts[scaled_counts$counts_scaled <= 500, ]`?

- a. 127408, 6
- b. 66909, 18
- c. 3, 3
- d. 3595, 6

{{Sdet}}



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

```
B {{Edet}}
```

## Bonus: Find and Replace using bracket subsetting

```
{{Sdet}}
```

```
Bonus content{{Esum}}
```

There are infinite uses for find and replace functionality, and like most topics in R, there are multiple ways to search for and replace values in a data frame.

You could use bracket sub-setting. Let's say we noticed a typo in a gene annotation in our `interesting_trnsc` data frame. We want to search for the "MCL1" gene and replace it with "MCL2". We could do the following:

```
#The column interesting_trnsc$transcript is a vector of gene names
#in the interesting_trnsc data frame.
#We can subset like we do with vectors
interesting_trnsc$transcript[interesting_trnsc$transcript=="MCL1"]
```

```
[1] MCL1 MCL1 MCL1 MCL1 MCL1 MCL1 MCL1 MCL1
Levels: EXT1 MCL1 LASP1 CPD
```

```
#we found the gene of interest; now, let's replace with MCL2
interesting_trnsc$transcript[
 interesting_trnsc$transcript=="MCL1"]<-"MCL2"
```

```
Warning in `[<-.factor`(`*tmp*`, interesting_trnsc$transcript == '
invalid factor level, NA generated
```

```
#Ah, we received a warning and the values became NAs.
#This is because transcript is a factor with set levels.
#let's replace those NAs with our original factor level MCL1
interesting_trnsc$transcript[
 is.na(interesting_trnsc$transcript)]<-"MCL1"

#To change the MCL1 value, we can simply change the factor level
levels(interesting_trnsc$transcript)[2] <-"MCL2"

#Alternatively, we could change this factor to a character vector.
interesting_trnsc$transcript<-
```

```

as.character(interesting_trnsc$transcript)
#Let's change MCL2 back to MCL1
interesting_trnsc$transcript[interesting_trnsc$transcript=="MCL2"]<-
 "MCL1"

#if this typo was present in multiple columns we could use
#just creating the same column elsewhere to test
interesting_trnsc$new_transcripts<-interesting_trnsc$transcript
interesting_trnsc[interesting_trnsc=="MCL1"]<- "MCL2"

#Be careful if these columns are factors

```

There are also a number of functions that have similar functionality. Check out `sub()` and `gsub()`. According to R documentation, "sub and gsub perform replacement of the first and all matches respectively". `gsub()` can only work on a single vector for replacement in a data frame. For global replacement across an entire data frame, you will have to get more creative if using `gsub()`. We will not be covering the `apply` functions, but they are useful and can often be used in the place of a complicated `for` loop. [Here \(https://ademos.people.uic.edu/Chapter4.html\)](https://ademos.people.uic.edu/Chapter4.html) is a nice tutorial for those that are interested. The advantage of using functions related to pattern matching and replacement (e.g., `gsub()` or `sub()`) is that you can use [regular expressions \(https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html\)](https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html) to match complicated patterns. {{Edet}}

## Save our data frame to a file

Perhaps we want to use our `sscaled` df in another program. Let's save this to a file.

```

write.table(sscaled,
 file = "sscaled.txt",
 quote=FALSE, row.names=FALSE, sep="\t")
#if you don't know what these arguments mean,
#use ?write.table to get help.

```

## Introduction to 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 (ie. rows ( $m$ ) and columns ( $n$ ), commonly denoted  $m \times n$ ). Note that 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).

However, the primary difference between a df and a matrix is that a matrix only contains values of a single type (i.e., numeric, character, logical, etc.).

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 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        |

## Exporting files from RStudio

Remember, because we are using RStudio server through DNAnexus, any files created by you today will be erased at the end of the session.

To use the materials you generated on the RServer on DNAnexus on your local computer, you will need to export your files using `More` and `Export` in the `Files` pane.

## 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 from a 2021 workshop entitled *Introduction to Tidy Transcriptomics* ([https://stemangiola.github.io/bioc2021\\_tidytranscriptomics/articles/tidytranscriptomics.html](https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html)) by Maria Doyle and Stefano Mangiola.

## Resources

1. [BaseR cheatsheet](#)

# Introduction to Data Wrangling with the Tidyverse

## Objectives

Wrangle data using tidyverse functionality (i.e., `dplyr`). To this end, you should understand:

1. how to use common `dplyr` functions (e.g., `select()`, `group_by()`, `arrange()`, `mutate()`, `summarize()`, and `filter()`)
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 data using `tidyr`.

## Reminder: Uploading files from RStudio Server

Any files created by you today will be erased at the end of the session. You can upload any files you downloaded from the last session using the Upload option in the Files pane.

## 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)
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)

But, what is tidy data???

## 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

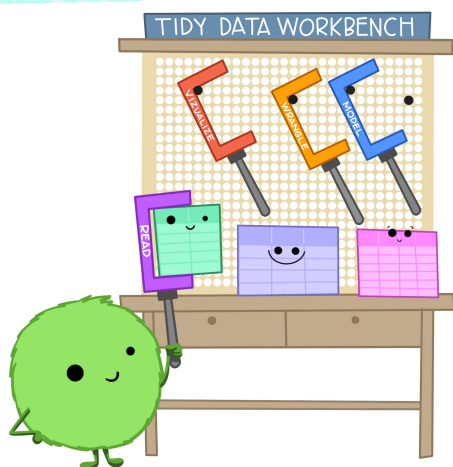
| id | name   | color  |
|----|--------|--------|
| 1  | floof  | gray   |
| 2  | max    | black  |
| 3  | cat    | orange |
| 4  | donut  | gray   |
| 5  | merlin | black  |
| 6  | panda  | calico |

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.

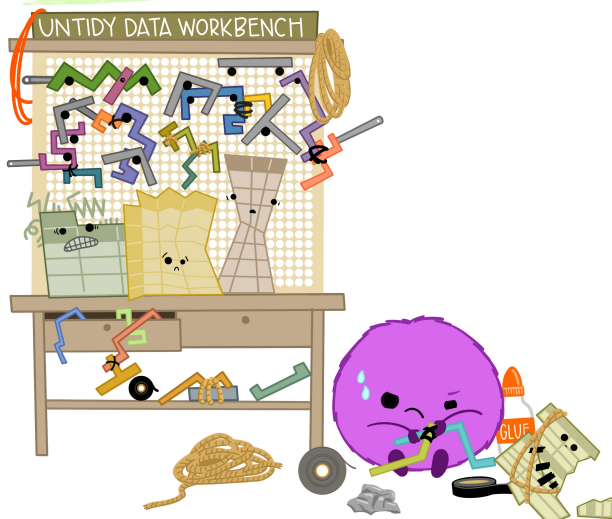


Image from Lowndes and Horst 2020: Tidy Data for Efficiency, Reproducibility, and Collaboration (<https://www.openscapes.org/blog/2020/10/12/tidy-data/>)

## 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** that contains 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 SRR:
2 ENSG000000000419 508 467 GSM1275862 N61311 untrt untrt SRR:
3 ENSG000000000457 508 260 GSM1275862 N61311 untrt untrt SRR:
4 ENSG000000000460 508 60 GSM1275862 N61311 untrt untrt SRR:
5 ENSG000000000971 508 3251 GSM1275862 N61311 untrt untrt SRR:
6 ENSG00000001036 508 1433 GSM1275862 N61311 untrt untrt SRR:
avgLength Experiment Sample BioSample transcript ref_genom
1 126 SRX384345 SRS508568 SAMN02422669 TSPAN6 hg:
2 126 SRX384345 SRS508568 SAMN02422669 DPM1 hg:
3 126 SRX384345 SRS508568 SAMN02422669 SCYL3 hg:
4 126 SRX384345 SRS508568 SAMN02422669 C1orf112 hg:
5 126 SRX384345 SRS508568 SAMN02422669 CFH hg:
6 126 SRX384345 SRS508568 SAMN02422669 FUCA2 hg:
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
```

```
— Attaching packages ————— tidy
```

```
✓ ggplot2 3.3.6 ✓ purrr 0.3.4
✓ tibble 3.1.7 ✓ dplyr 1.0.9
✓ tidyr 1.2.0 ✓ stringr 1.4.0
✓ readr 2.1.2 ✓ forcats 0.5.1
```

```
— Conflicts ————— tidyverse_
✘ dplyr::filter() masks stats::filter()
✘ dplyr::lag() masks stats::lag()
```

```
Adding missing grouping variables: `sample`
```

An untidy version (subset) of `scaled_counts`.

```
c # view a snapshot of an untidy data frame
```

```
##
cell 508
dex N61311
SampleName untrt
Run / Experiment / Accession SRR1039508;SRX384345;SRS508568
TSPAN6 960.886417275434
DPM1 660.874752382368
SCYL3 367.938834302817
C1orf112 84.9089617621886
CFH 4600.65057814792
##
cell 509
dex N61311
SampleName trt
Run / Experiment / Accession SRR1039509;SRX384346;SRS508567
TSPAN6 716.779730254346
DPM1 823.976698841491
SCYL3 337.590453311757
C1orf112 87.9975115267612
CFH 5886.23354376281
##
cell 512
dex N052611
SampleName untrt
Run / Experiment / Accession SRR1039508;SRX384345;SRS508568
```

```

Run / Experiment / Accession SRR1039512;SRX384349;SRS508571
TSPAN6 1075.40953718585
DPM1 764.982041915709
SCYL3 323.977901809712
Clorf112 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
Clorf112 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
Clorf112 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
Clorf112 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
Clorf112 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 tidyverse library

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

```
library(tidyverse)
```

## 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 \(https://datacarpentry.com/genomics-r-intro/05-dplyr/index.html\)](https://datacarpentry.com/genomics-r-intro/05-dplyr/index.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") ## install
library("dplyr")
```

Let's read in some more 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 there is a tidyverse equivalent, glimpse()
str(dexp, give.attr=FALSE)
```

```
tibble [15,926 × 10] (S3: tbl_df/tbl/data.frame)
$ feature : chr [1:15926] "ENSG000000000003" "ENSG000000000419" ...
$ 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 ...
$ FDR : num [1:15926] 0.00283 0.07701 0.84425 0.68233 0.00...
```

```
glimpse(dexp)
```

```
Rows: 15,926
Columns: 10
$ feature <chr> "ENSG000000000003", "ENSG000000000419", "ENSG0000...
$ albut <fct> untrt, untrt, untrt, untrt, untrt, untrt, untrt, untrt, untrt, untrt
$ transcript <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH", "CFH", "CFH", "CFH", "CFH", "CFH"
$ ref_genome <chr> "hg38", "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, -0.124400000, 0.417300000, 0.417300000, 0.417300000, 0.417300000, 0.417300000, 0.417300000
$ logCPM <dbl> 5.059704, 4.611483, 3.482462, 1.473375, 8.089140, 8.089140, 8.089140, 8.089140, 8.089140, 8.089140
$ F <dbl> 3.284948e+01, 6.903534e+00, 9.685073e-02, 3.772000e-01, 29.339000e+00, 29.339000e+00, 29.339000e+00, 29.339000e+00, 29.339000e+00, 29.339000e+00
$ PValue <dbl> 0.0003117656, 0.0280616149, 0.7629129276, 0.5546960000, 0.0003117656, 0.0003117656, 0.0003117656, 0.0003117656, 0.0003117656, 0.0003117656
$ FDR <dbl> 0.002831504, 0.077013489, 0.844247837, 0.6823300000, 0.002831504, 0.002831504, 0.002831504, 0.002831504, 0.002831504, 0.002831504
```

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

Note: We will also be returning to our `sscaled` data, so let's load that here, as well as a data frame called `interesting_trnsc`, which contains only 4 transcripts we deemed interesting.

We can use import functions from the tidyverse, 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
i Specify the column types or set `show_col_types = FALSE` to quie
```

```
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 C1orf112 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.
... with 127,398 more rows
```

```
#Create interesting_trnsc
keep_t<-c("CPD","EXT1","MCL1","LASP1")
interesting_trnsc<-sscaled[sscaled$transcript %in% keep_t,]
interesting_trnsc
```

```
A tibble: 32 × 6
sample cell dex transcript counts counts_scaled
<dbl> <chr> <chr> <chr> <dbl> <dbl>
```

```
1 508 N61311 untreated LASP1 6314 8935.
2 508 N61311 untreated CPD 4533 6415.
3 508 N61311 untreated MCL1 5216 7381.
4 508 N61311 untreated EXT1 6279 8886.
5 509 N61311 treated LASP1 6364 10182.
6 509 N61311 treated CPD 4578 7325.
7 509 N61311 treated MCL1 6239 9982.
8 509 N61311 treated EXT1 3550 5680.
9 512 N052611 untreated LASP1 7831 9647.
10 512 N052611 untreated CPD 8270 10187.
... with 22 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).

## Subsetting with `dplyr`

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

### 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

```
#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 Clorf112 -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
... with 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, transcr
```

{{Edet}}

1. 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}}

## Introducing the pipe

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

For example,

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

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

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

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

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

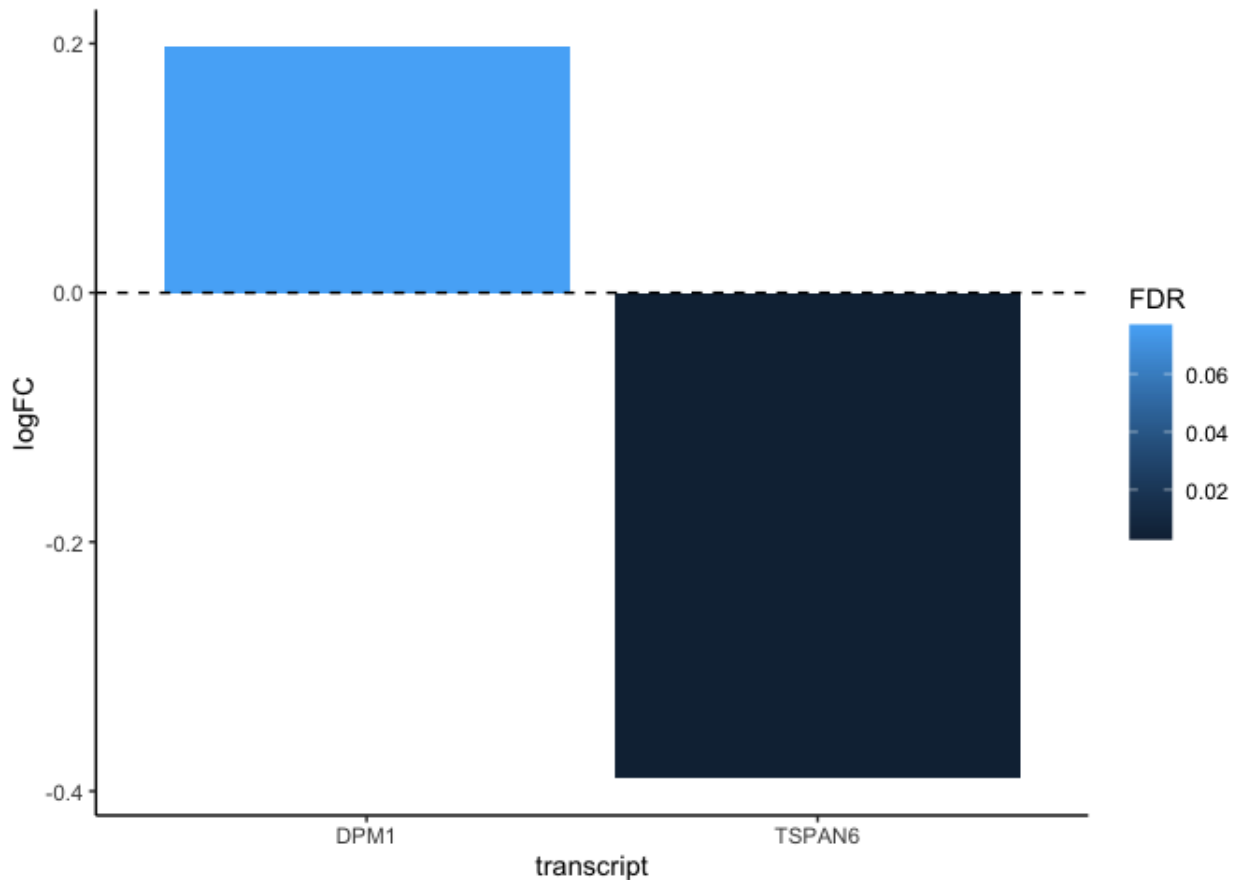
Let's see how this works

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

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

Piping from the beginning:

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

## Test your learning

Using what you have learned about `select()` and `filter()`, create a subsetting 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 and transmute

Other useful data manipulation functions from `dplyr` include `mutate()` and `transmute()`. These functions allow 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()` adds new variables and preserves existing ones; `transmute()` adds new variables and drops existing ones. New variables overwrite existing variables of the same name. --- [dplyr.tidyverse.org \(https://dplyr.tidyverse.org/reference/mutate.html\)](https://dplyr.tidyverse.org/reference/mutate.html)

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

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

The conditional evaluates to a logical vector (contains TRUE or FALSE).

We can also use `mutate` to coerce variables. Remember those one liners we used in the factor section to coerce our character vectors to factors?

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

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

```
Rows: 127,408
Columns: 6
$ sample <dbl> 508, 508, 508, 508, 508, 508, 508, 508, 508
$ cell <chr> "N61311", "N61311", "N61311", "N61311", "N6:
$ dex <chr> "untreated", "untreated", "untreated", "unti
```

```
$ transcript <chr> "TSPAN6", "DPM1", "SCYL3", "C1orf112", "CFH"
$ counts <dbl> 679, 467, 260, 60, 3251, 1433, 519, 394, 17
$ counts_scaled <dbl> 960.88642, 660.87475, 367.93883, 84.90896, 4
```

```
#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
$ cell <fct> N61311, N61311, N61311, N61311, N61311, N61311, N61311, N61311, N61311
$ dex <fct> untreated, untreated, untreated, untreated, untreated, untreated, untreated, untreated, untreated
$ transcript <fct> TSPAN6, DPM1, SCYL3, C1orf112, CFH, FUCA2, FUCA2, FUCA2, FUCA2
$ counts <dbl> 679, 467, 260, 60, 3251, 1433, 519, 394, 17
$ counts_scaled <dbl> 960.88642, 660.87475, 367.93883, 84.90896, 4
```

## 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
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.
... with 29,142 more rows
```

The output is a grouped data frame.

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.

```
a<-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
argument.
```

```
#can skip arrange and use slice_max
b<-scaled_counts %>%
 group_by(dex,transcript) %>%
 summarize(median_counts=median(counts_scaled)) %>%
 slice_max(n=5, order_by=median_counts) #notice use of slice_max
```

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

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
argument.
```

```
A tibble: 8 × 3
Groups: dex [2]
dex sample n
<chr> <int> <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)
```

```
dex sample n
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 \(https://datacarpentry.org/genomics-r-intro/05-dplyr/index.html\)](https://datacarpentry.org/genomics-r-intro/05-dplyr/index.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

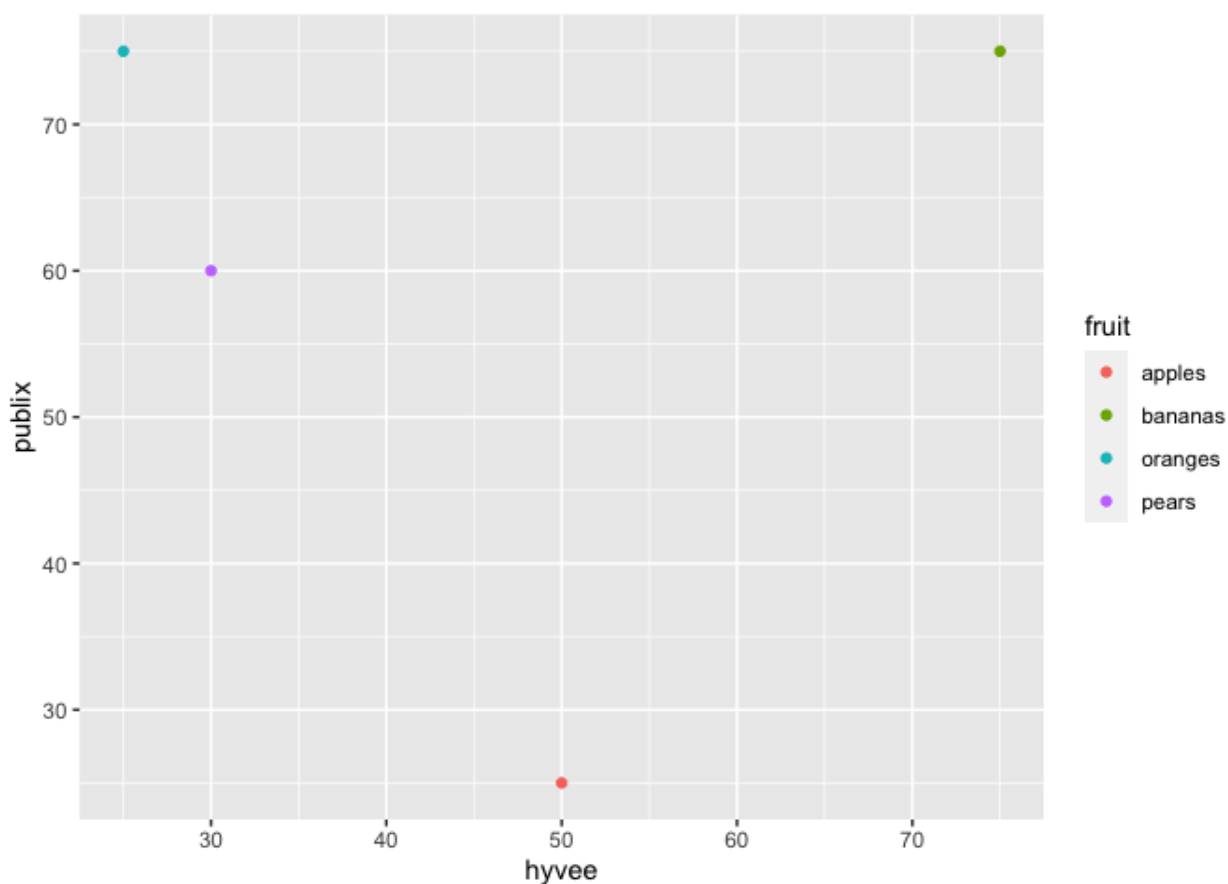
Tidy data implies that we have one observation per row and one variable per column. This generally means data is in a long format. However, whether data is tidy or not will depend on what we consider a variable versus an observation, so wide data sets may also be tidy. Often if data is in a wide format, 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.

Let's look at some grocery data.

```
grocery <- data.frame(fruit=c("apples", "oranges", "bananas", "pears",
 "strawberries", "blueberries", "raspberries", "kiwis",
 "mangoes", "pineapples", "lemons", "limes", "avocados", "cantaloupes", "peaches",
 "pears", "apples", "oranges", "bananas", "pears", "strawberries", "blueberries",
 "raspberries", "kiwis", "mangoes", "pineapples", "lemons", "limes", "avocados",
 "cantaloupes", "peaches"),
 hyvee=c(50,25,75,30),publix=c(25,75,75,60),
 kroger=c(35,80,25,15),safeway=c(45,45,35,55))
```

Formatted here, this data is in a wide format. We can see an initial column with fruit (i.e., apples, oranges, bananas, pears) followed by four columns containing integer data related to grocery stores. These columns (hyvee, publix, kroger, and safeway) all hold data of the same type, collected in the same way. Based on our rules, this isn't really tidy data, but if we want to plot fruit counts in one grocery store against another, this is the best data format. Note: I'm not sure what information you gain from this type of plot. For example,

```
ggplot(grocery)+
 geom_point(aes(x=hyvee,y=publix,color=fruit))
```



This data can easily be converted to long / tidy format using `pivot_longer()`. `pivot_longer()` transforms data to long format, while `pivot_wider()` converts data to wide format.

`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`

```
(g2<-pivot_longer(grocery,cols=2:5,names_to="store",
 values_to="quantity"))
```

```
A tibble: 16 × 3
fruit store quantity
<chr> <chr> <dbl>
1 apples hyvee 50
2 apples publix 25
3 apples kroger 35
4 apples safeway 45
5 oranges hyvee 25
```

```
6 oranges publix 75
7 oranges kroger 80
8 oranges safeway 45
9 bananas hyvee 75
10 bananas publix 75
11 bananas kroger 25
12 bananas safeway 35
13 pears hyvee 30
14 pears publix 60
15 pears kroger 15
16 pears safeway 55
```

If we want this back in wide format, we use `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`

```
pivot_wider(g2, names_from=store, values_from=quantity)
```

```
A tibble: 4 × 5
fruit hyvee publix kroger safeway
<chr> <dbl> <dbl> <dbl> <dbl>
1 apples 50 25 35 45
2 oranges 25 75 80 45
3 bananas 75 75 25 35
4 pears 30 60 15 55
```

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.

At times we may need to work with a matrix while at others we may need a dataframe. These functions can be used to go from a df to a matrix and back again.

```
#take a look at fun_df again;
#we see genes in one column and expression values in another.
fun_df
```

```
genes counts
1 A NA
2 A 214
3 A NA
```

```
4 B 352
5 B 256
6 B NA
7 C 400
8 C 381
9 C 250
10 D 278
11 D NA
12 D 169
```

```
#let's add a sample column and overwrite fun_df
#this data is in long format
fun_df<-data.frame(fun_df, sampleid=rep(c("A1","B1","C1"),4))

#let's convert to wide format using pivot_wider
fun_df_w<-fun_df %>%
 pivot_wider(sampleid,names_from=genes,values_from=counts)
fun_df_w
```

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

```
#convert to matrix
fun_mat<-fun_df_w %>% column_to_rownames("sampleid") %>%
 as.matrix(rownames.force=TRUE)
fun_mat
```

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

```
#Now, we can easily apply functions that require data matrices.
```

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 variable.

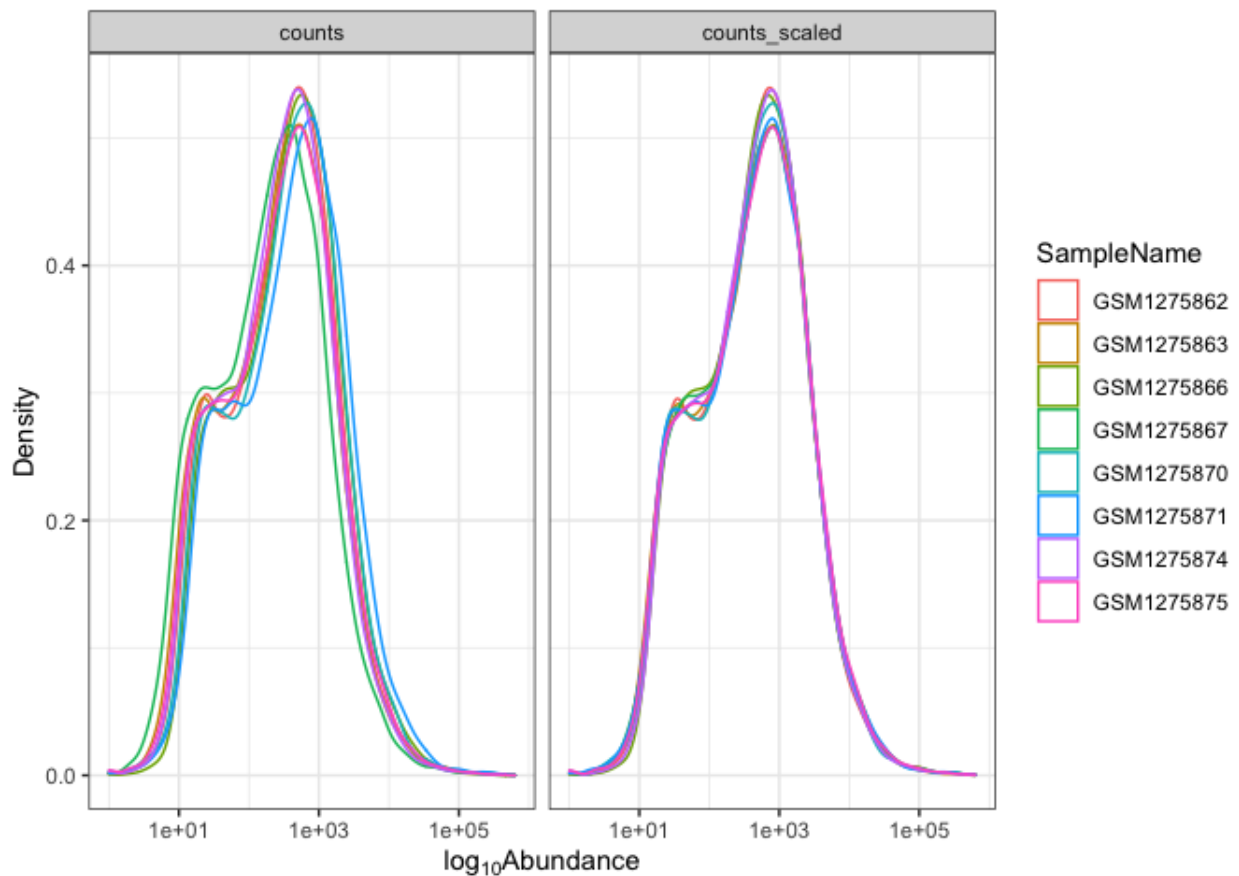
Let's see how this might work with our `scaled_counts` data. We want to plot both "counts" and "counts\_scaled" together in a density plot to understand the distribution of the data. Did scaling the counts improve the distribution?

```
#put counts and counts_scaled into a column named source
#with their values in a column named abundance
scounts_long<- scaled_counts %>% #getting the data
 pivot_longer(cols = c("counts", "counts_scaled"),
 names_to = "source", values_to = "abundance") #pivot
head(scounts_long)
```

```
A tibble: 6 × 18
feature sample SampleName cell dex albut Run avgLength Exp
<chr> <int> <chr> <chr> <chr> <chr> <chr> <int> <chr>
1 ENSG000... 508 GSM1275862 N613... untrt untrt SRR1... 126 SR)
2 ENSG000... 508 GSM1275862 N613... untrt untrt SRR1... 126 SR)
3 ENSG000... 508 GSM1275862 N613... untrt untrt SRR1... 126 SR)
4 ENSG000... 508 GSM1275862 N613... untrt untrt SRR1... 126 SR)
5 ENSG000... 508 GSM1275862 N613... untrt untrt SRR1... 126 SR)
6 ENSG000... 508 GSM1275862 N613... untrt untrt SRR1... 126 SR)
... with 8 more variables: BioSample <chr>, transcript <chr>, ref_
.abundant <lgl>, TMM <dbl>, multiplier <dbl>, source <chr>, al
```

```
#Plot using ggplot2; It's not important to understand this code here
ggplot(data=scounts_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: There are other ways to reformat data in R. Check out the package [reshape2](https://cran.r-project.org/web/packages/data.table/vignettes/datatable-reshape.html) (<https://cran.r-project.org/web/packages/data.table/vignettes/datatable-reshape.html>).

## Exporting files from RStudio

Remember, because we are using RStudio server through DNAnexus, any files created by you today will be erased at the end of the session.

To use the materials you generated on the RServer on DNAnexus on your local computer, you will need to export your files using **More** and **Export** in the **Files** pane.

## 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 from a 2021 workshop entitled [Introduction to Tidy Transcriptomics](https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html) ([https://stemangiola.github.io/bioc2021\\_tidytranscriptomics/articles/tidytranscriptomics.html](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/index.html) (<https://carpentries-incubator.github.io/bioc-intro/30-dplyr/index.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

# 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.

## Reminder: Uploading files from RStudio Server

Any files created by you today will be erased at the end of the session. You can upload any files you downloaded from the last session using the Upload option in the Files pane.

## 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 a 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 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 packages ————— tidy
```

```
✓ ggplot2 3.3.6 ✓ purrr 0.3.4
✓ tibble 3.1.7 ✓ dplyr 1.0.9
✓ tidyr 1.2.0 ✓ stringr 1.4.0
✓ readr 2.1.2 ✓ forcats 0.5.1
```

```
— Conflicts ————— tidyverse_
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag() masks stats::lag()
```

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)
```

## 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")

#If you are curious how this was made; here is the code
#scaled_counts %>% group_by(dex, SampleName) %>%
summarize(Num_transcripts=sum(counts>100),TotalCounts=sum(counts))

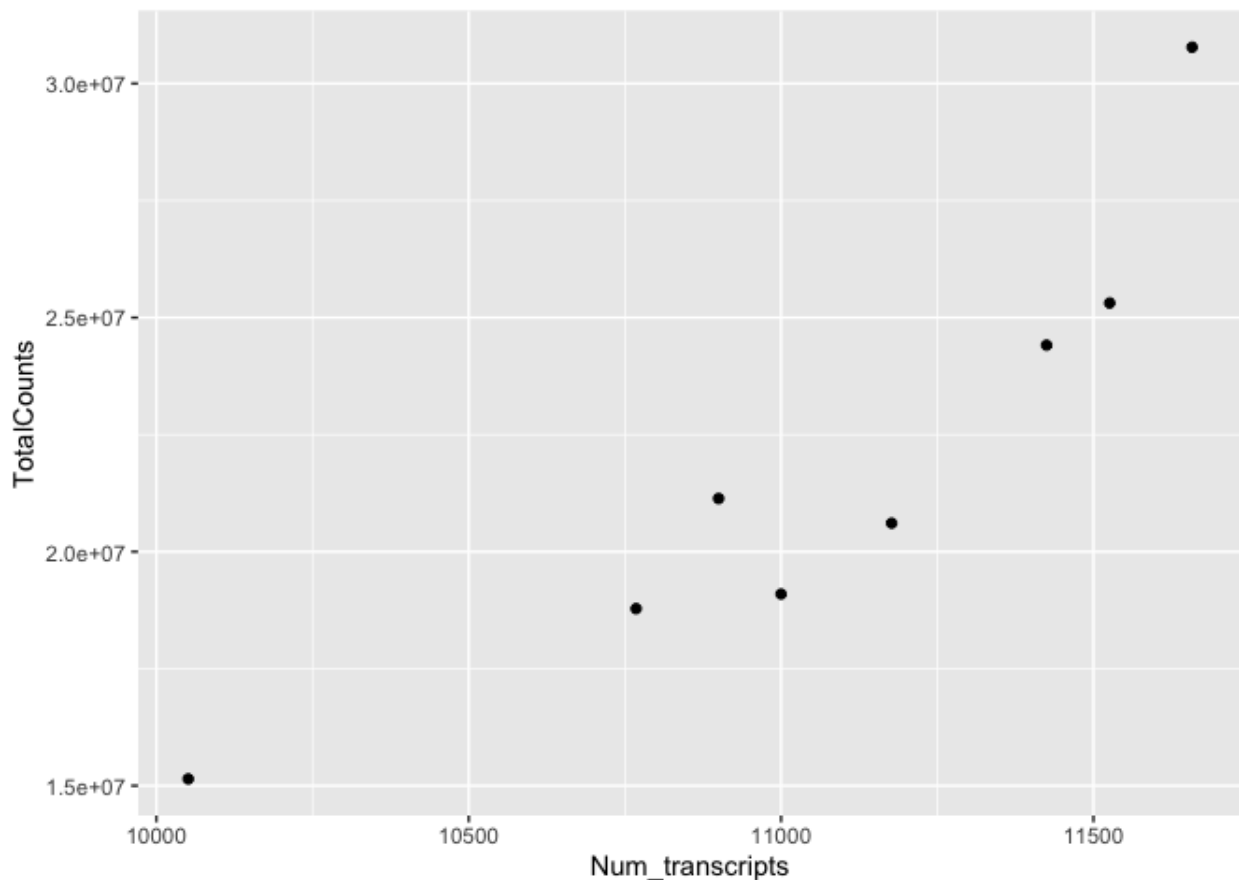
#let's view the data
sc
```

```
dex SampleName Num_transcripts TotalCounts
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.

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.

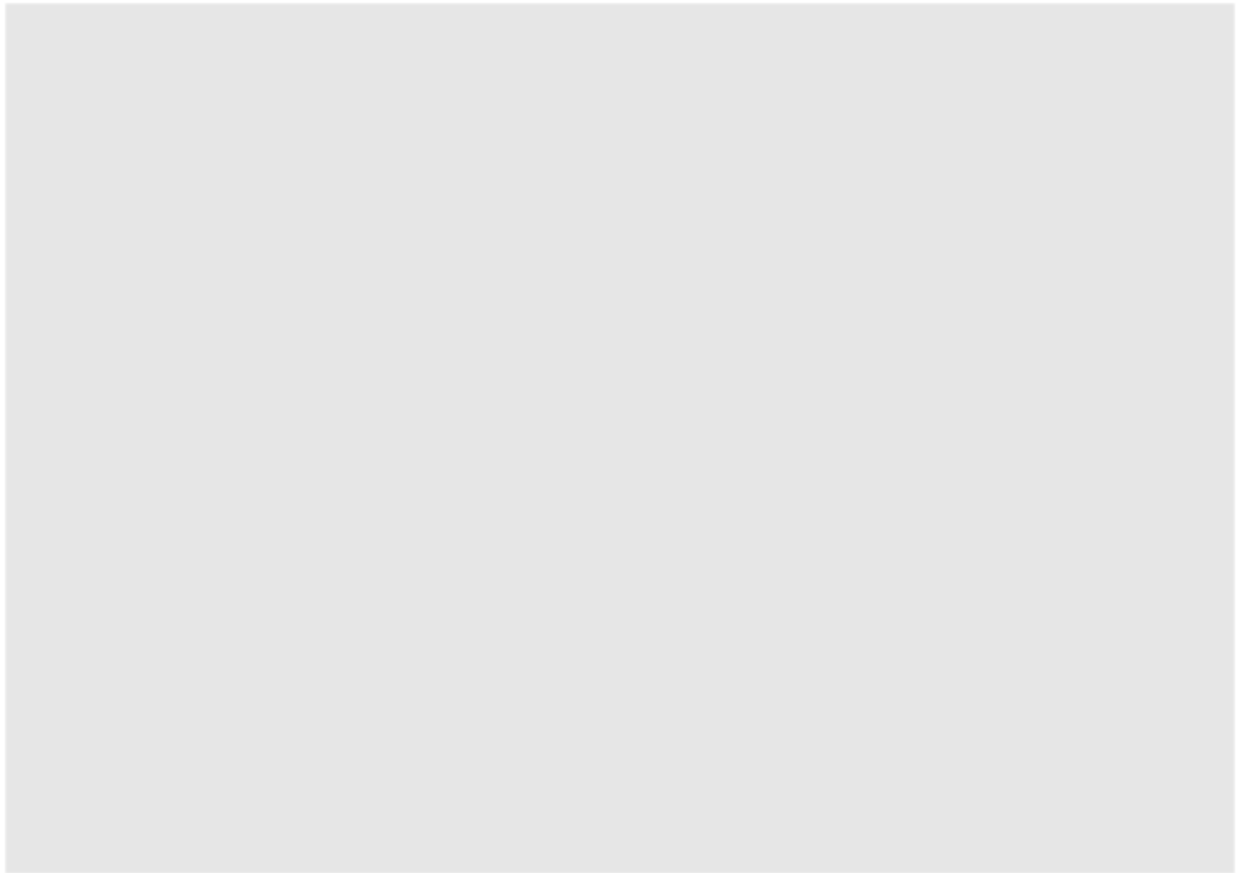
#### What did we do in the above code?

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)
```

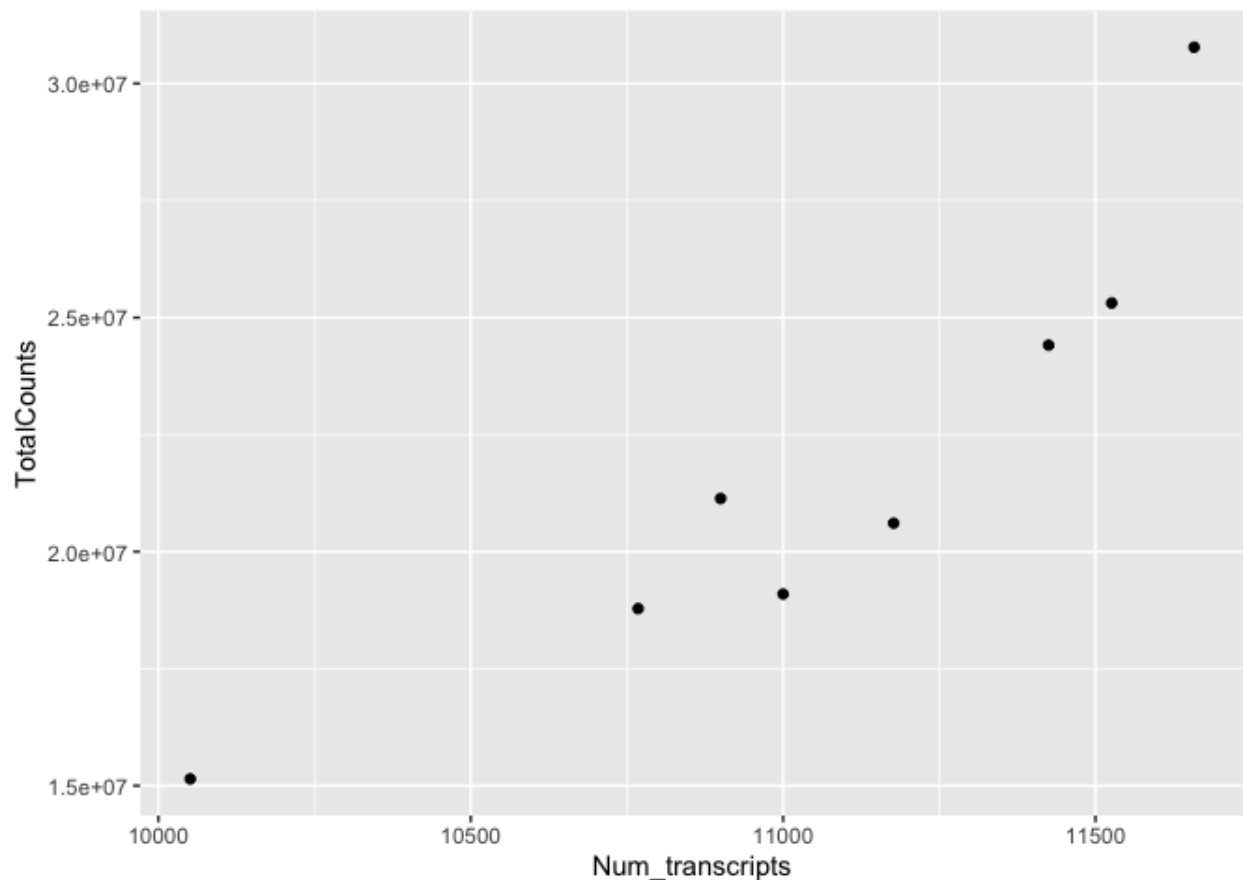


```
#What about just running a geom function?
geom_point(data=sc,aes(x=Num_transcripts, y = TotalCounts))
```

```
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)

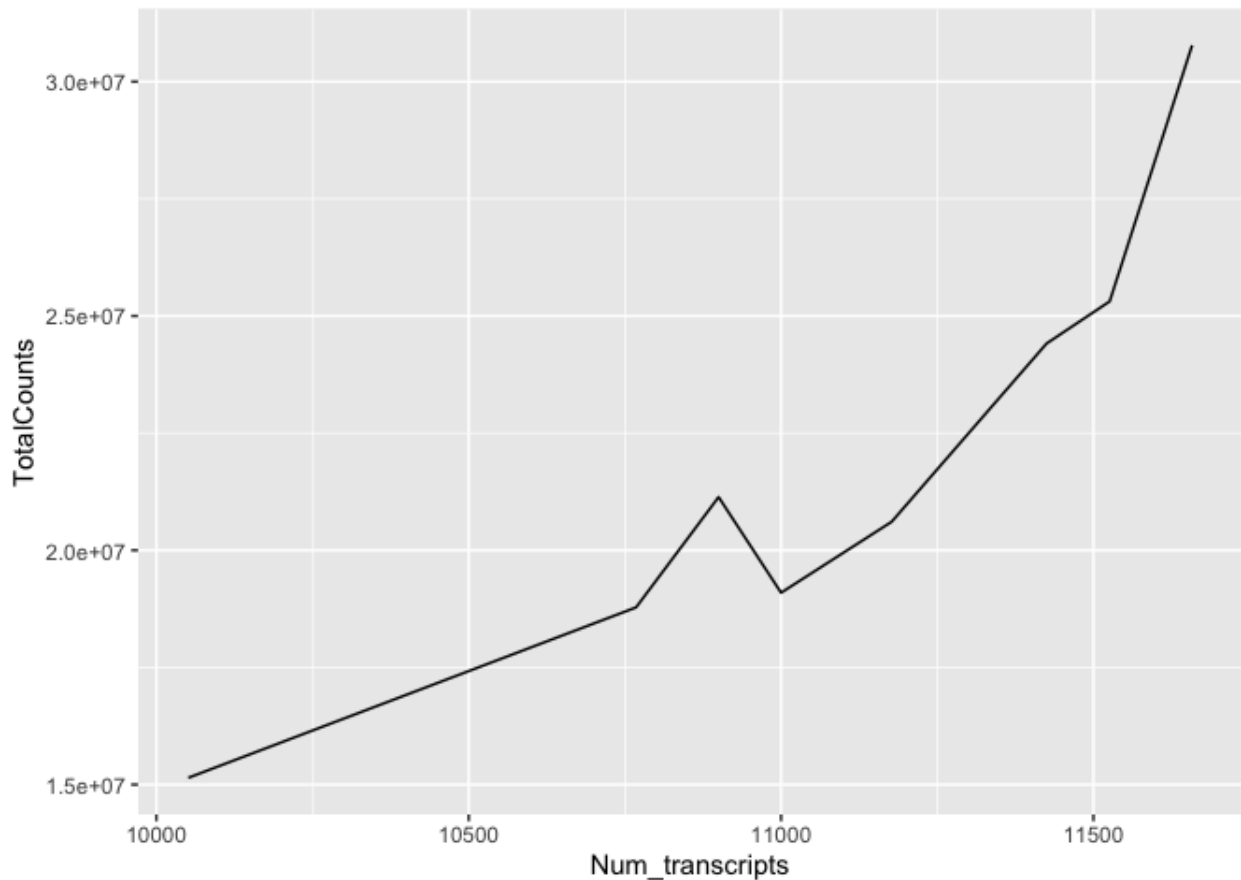
There are multiple geom functions that change the basic plot type or the plot representation. We can create scatter plots (`geom_point()`), line plots (`geom_line()`, `geom_path()`), bar plots (`geom_bar()`, `geom_col()`), line modeled to fitted data (`geom_smooth()`), heat maps (`geom_tile()`), geographic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

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

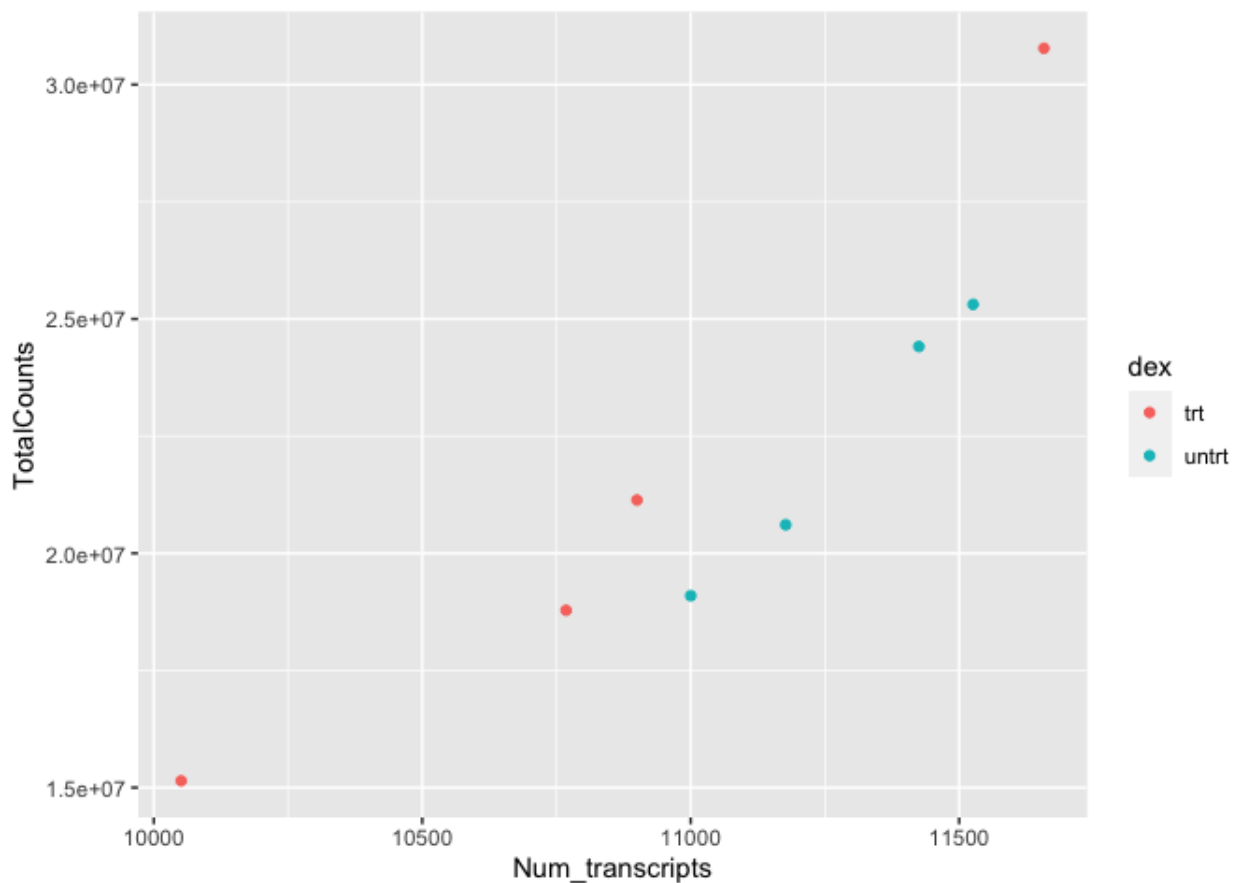
Mapping aesthetics include some of the following:

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))
```



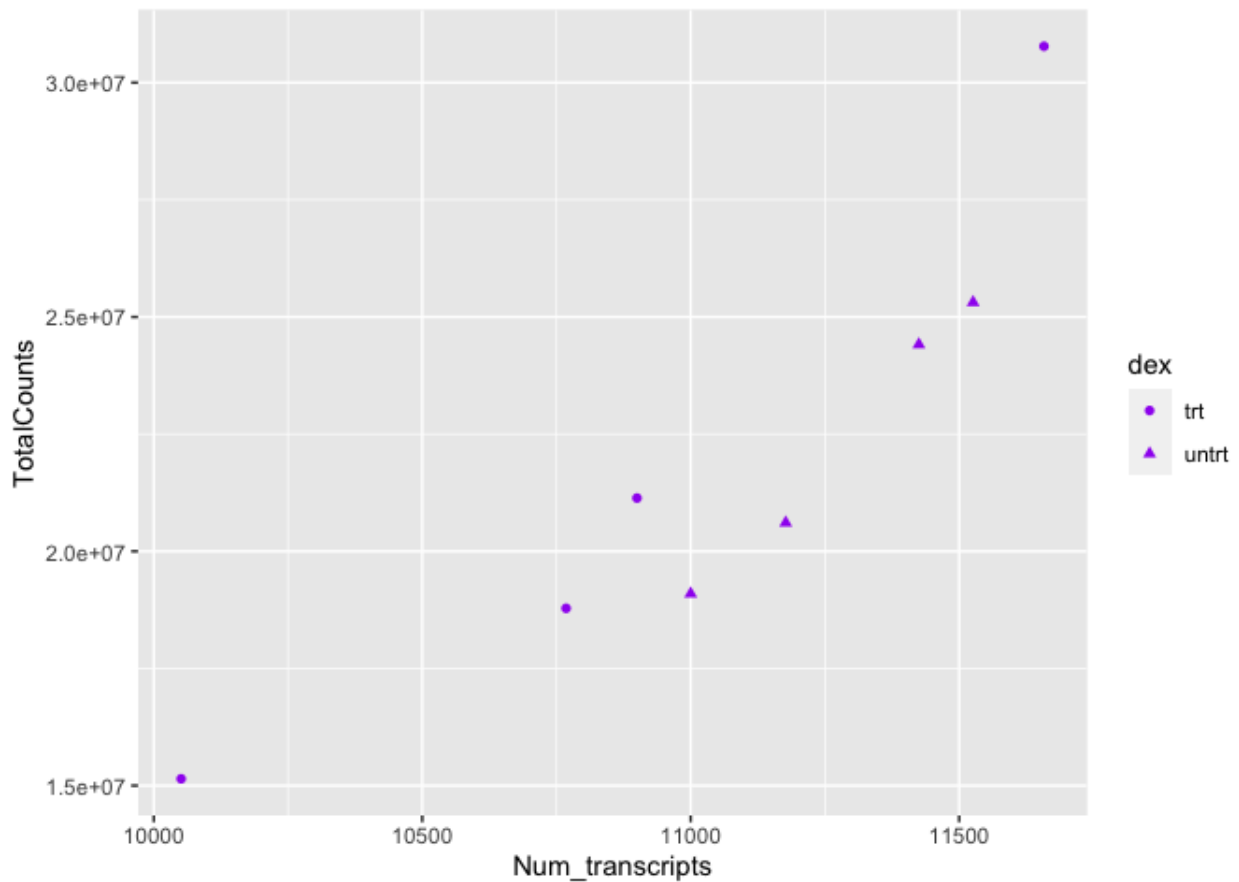
There is potentially a relationship. ASM cells treated with dexamethasone in general have lower total numbers of transcripts and lower total counts.

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")
```

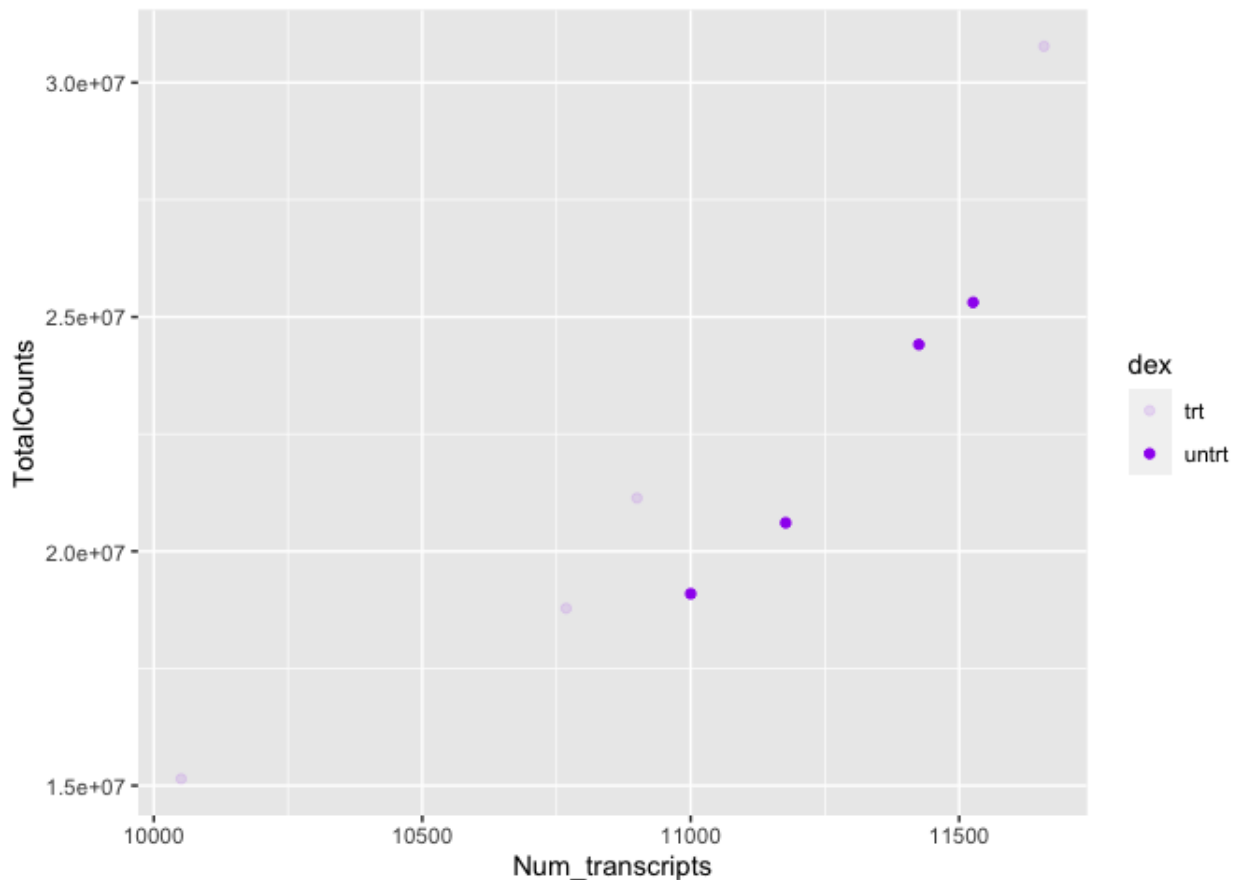


We can also see from this that 'dex' could be mapped to other aesthetics. In the above example, we see it mapped to shape rather than color. By default, ggplot2 will only map six shapes at a time, and if your number of categories goes beyond 6, the remaining groups will go unmapped. This is by design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

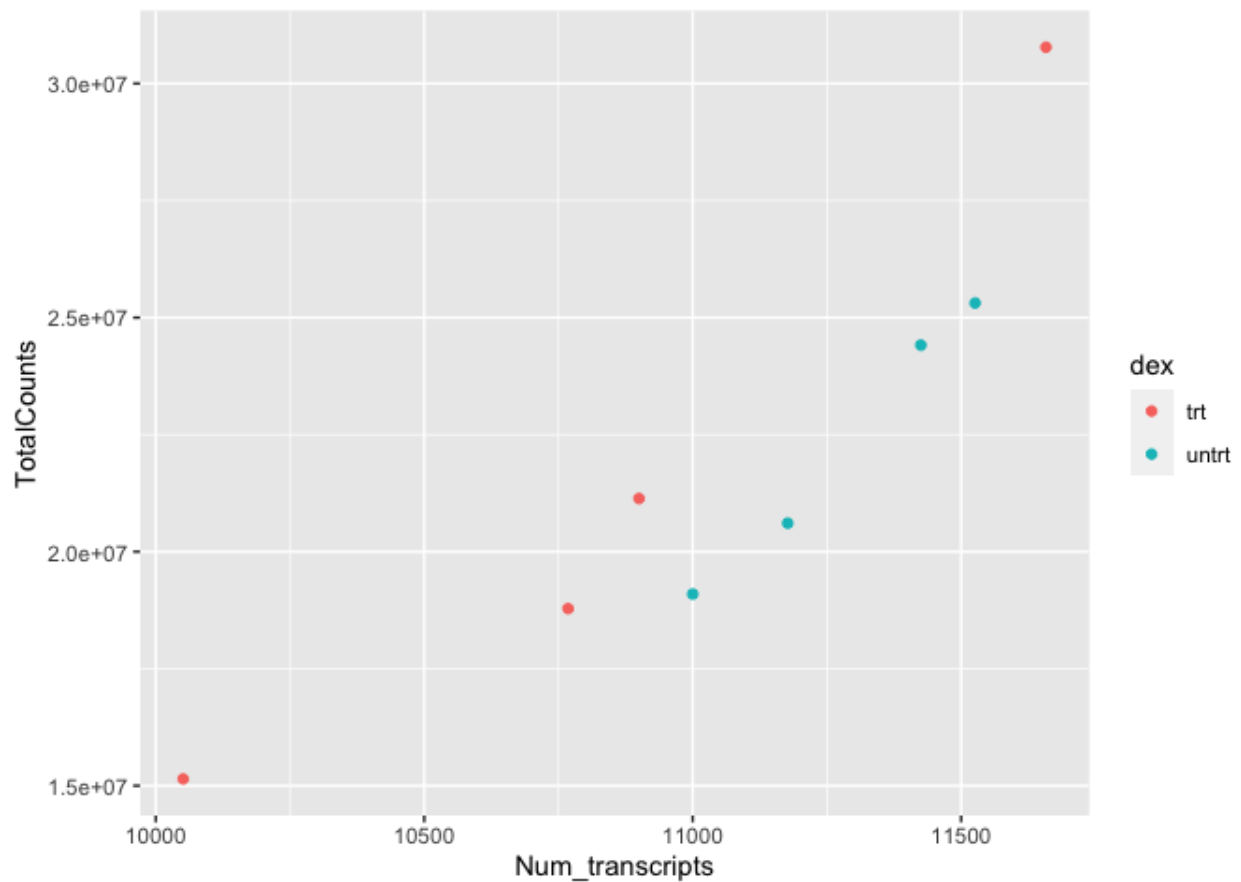
The assignment of color, shape, or alpha to our variable was automatic, with a unique aesthetic level representing each category (i.e., 'trt', 'untrt') within our variable. You will also notice that ggplot2 automatically created a legend to explain the levels of the aesthetic mapped. We can change aesthetic 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
```

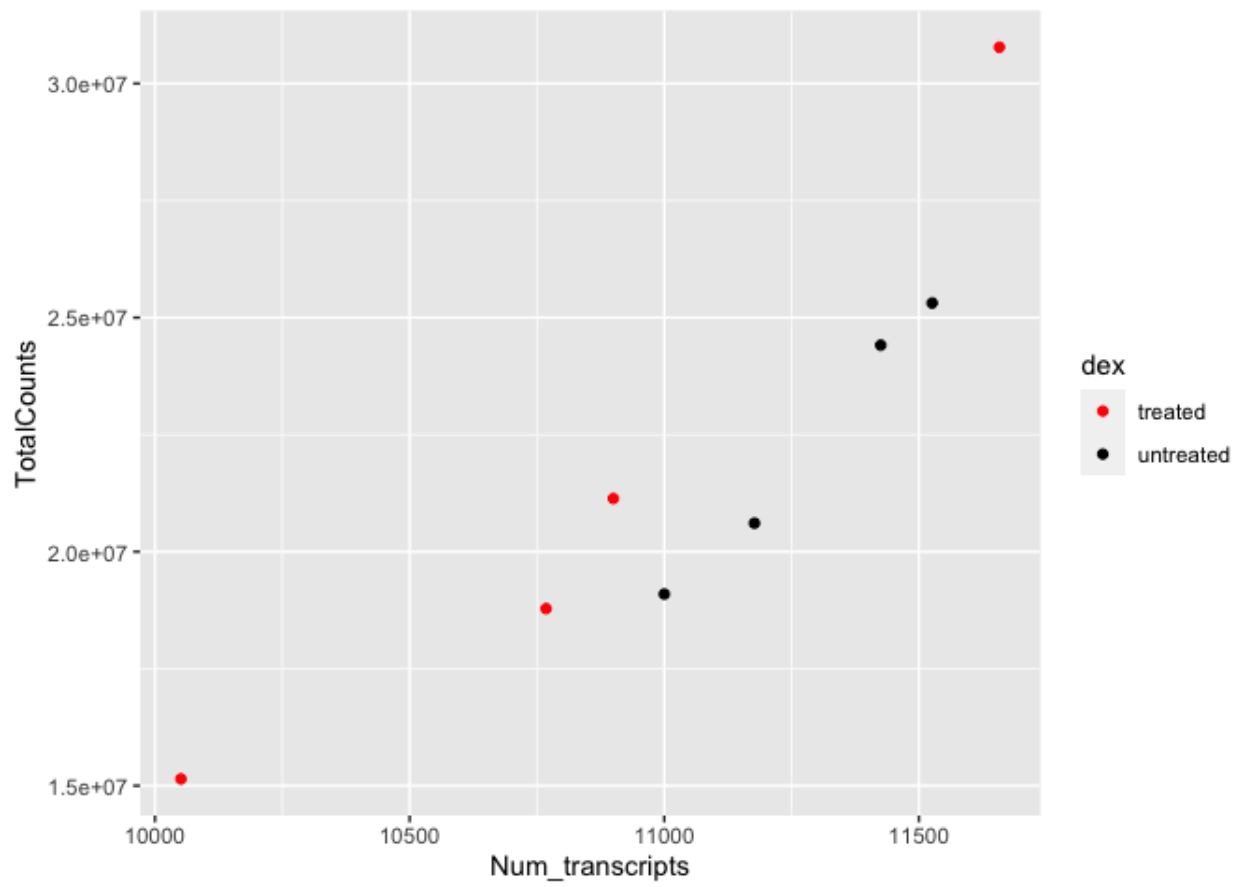


We can add additional layers directly to our object. We will see how this works by defining some colors for our 'dex' variable.

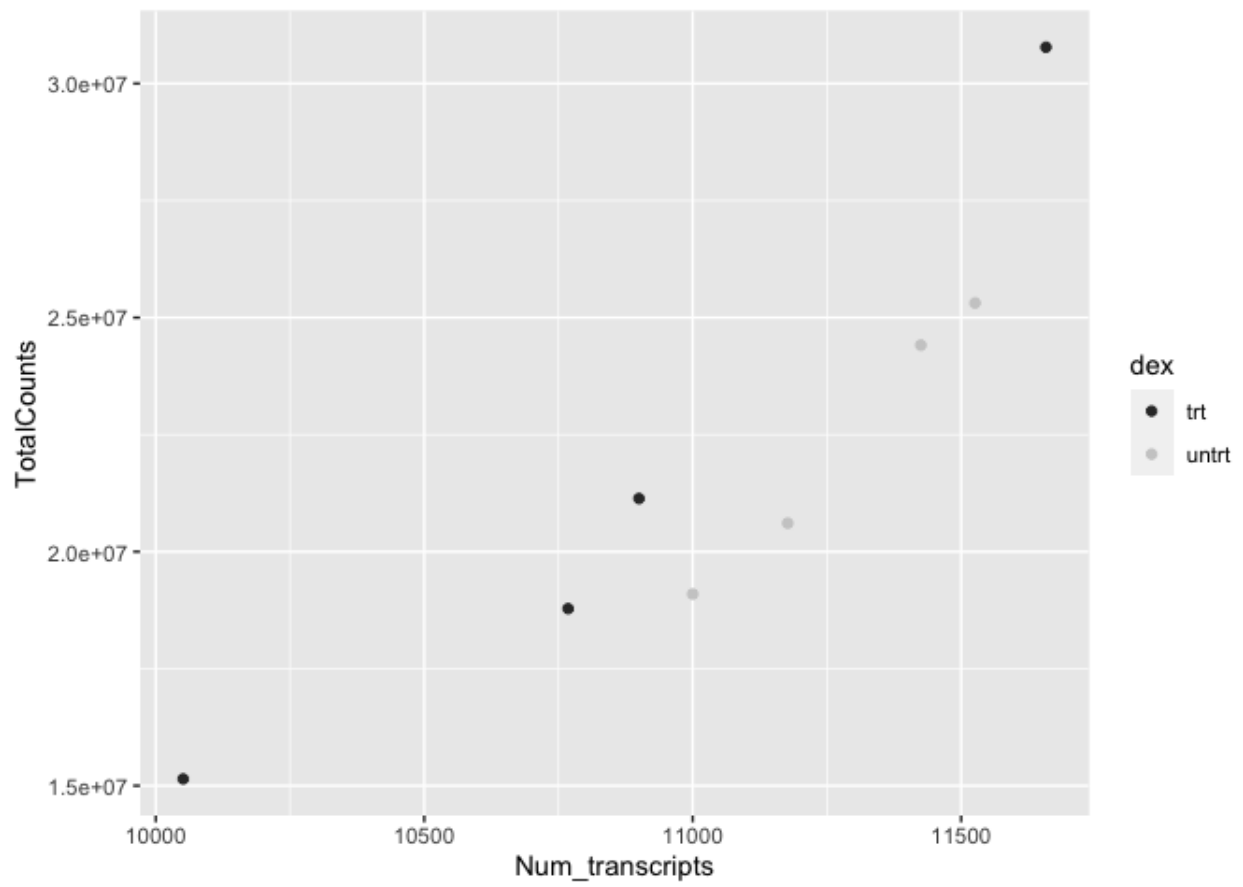
## Colors

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'))
```

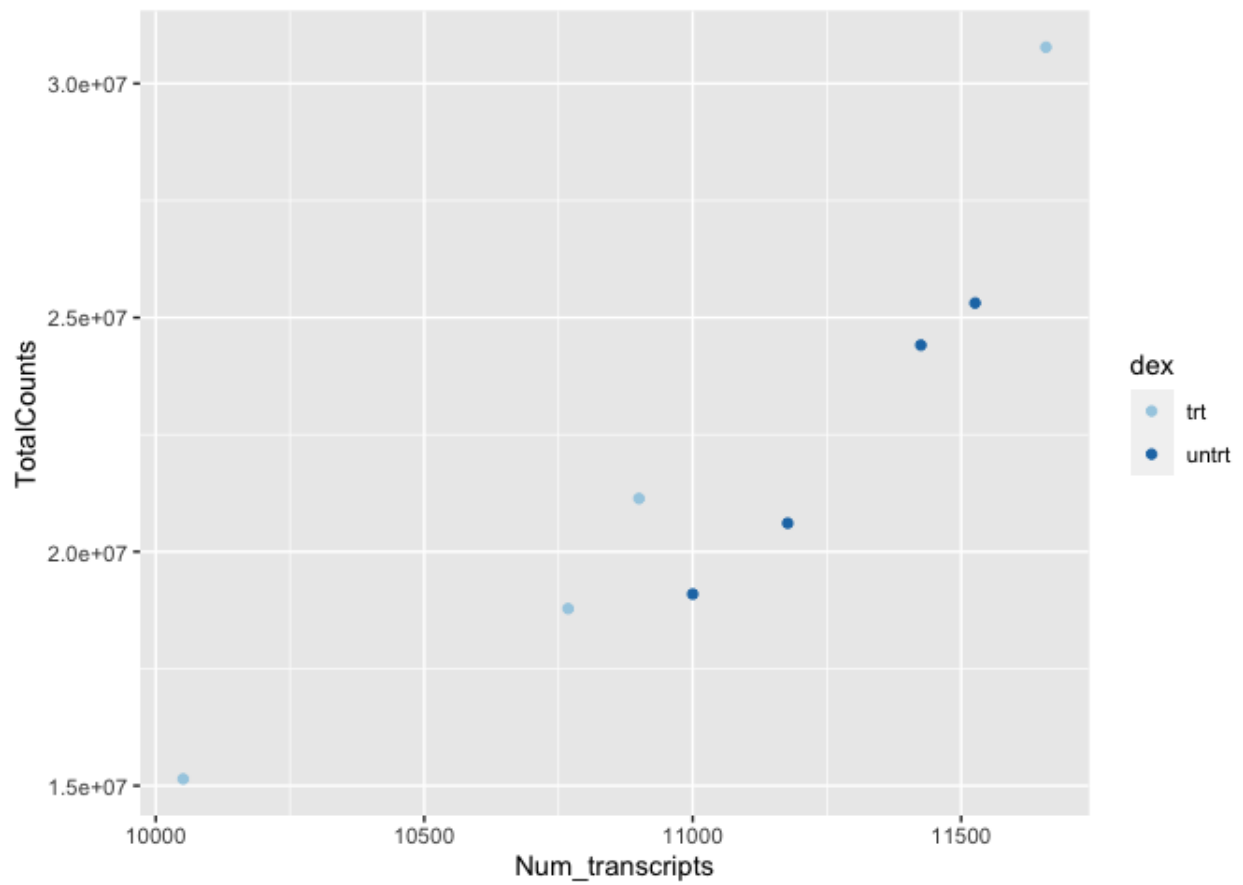


```
dot_plot +
 scale_color_grey()
```

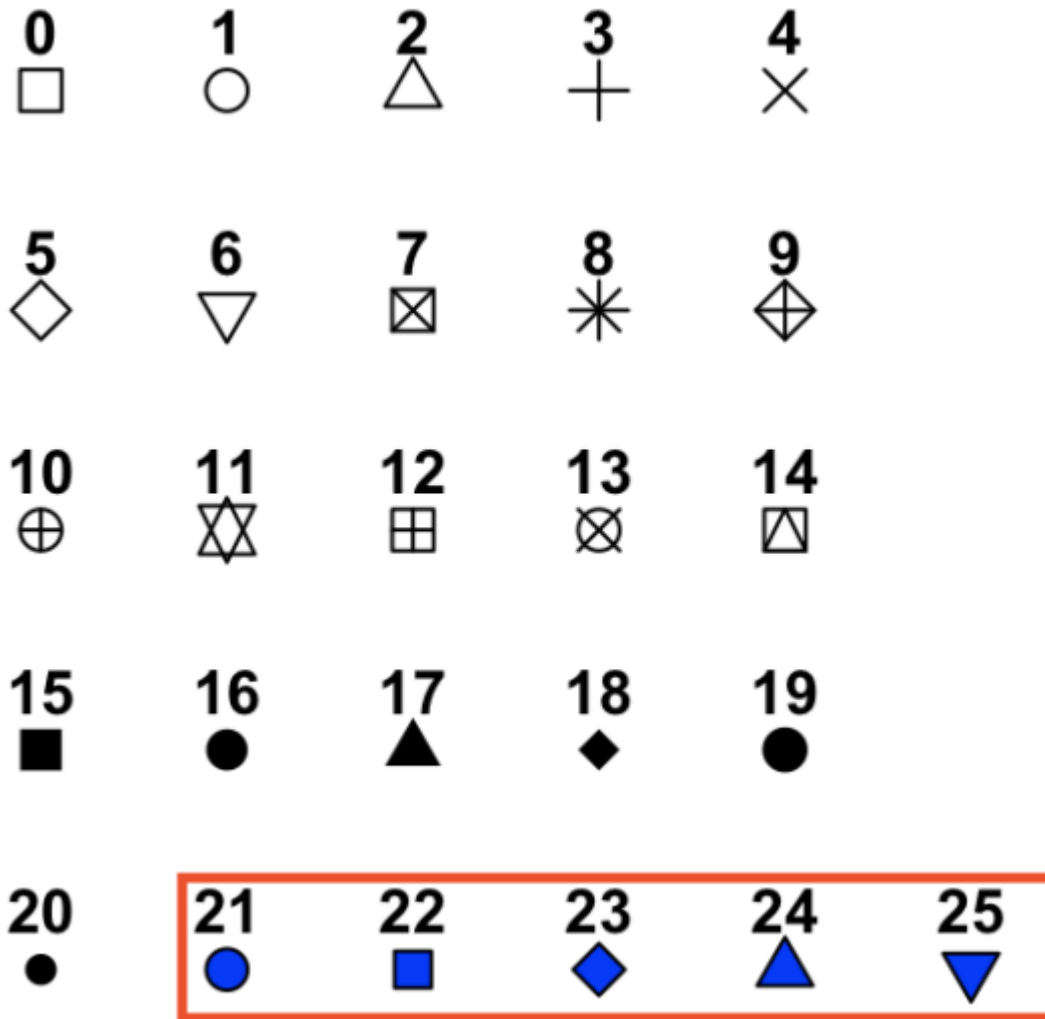


```
dot_plot +
 scale_color_brewer(palette = "Paired")
```

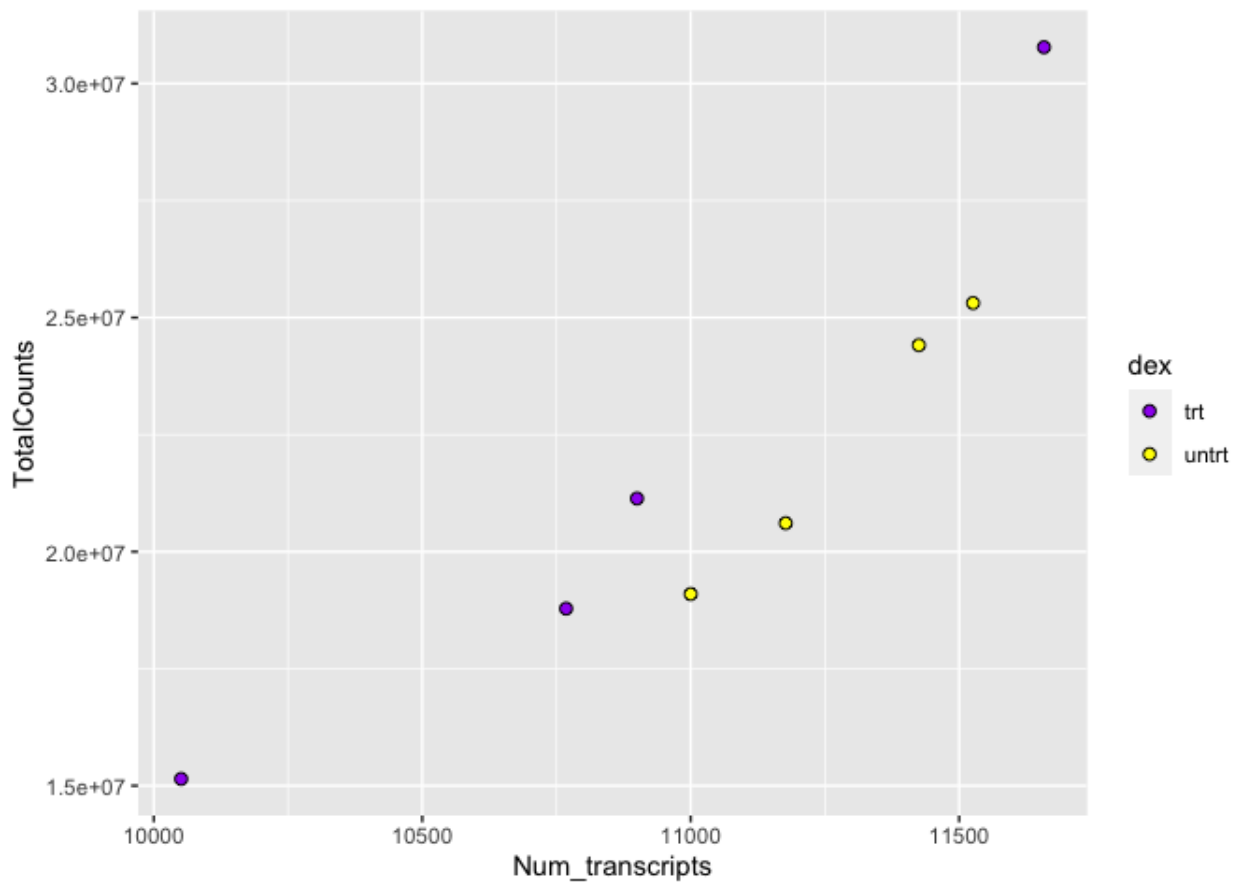




Similarly, if we want to change the fill, we would use the `scale_fill` options. To apply `scale_fill` to shape, we will have to alter the shapes, as only some shapes take a fill argument.



```
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"))
```



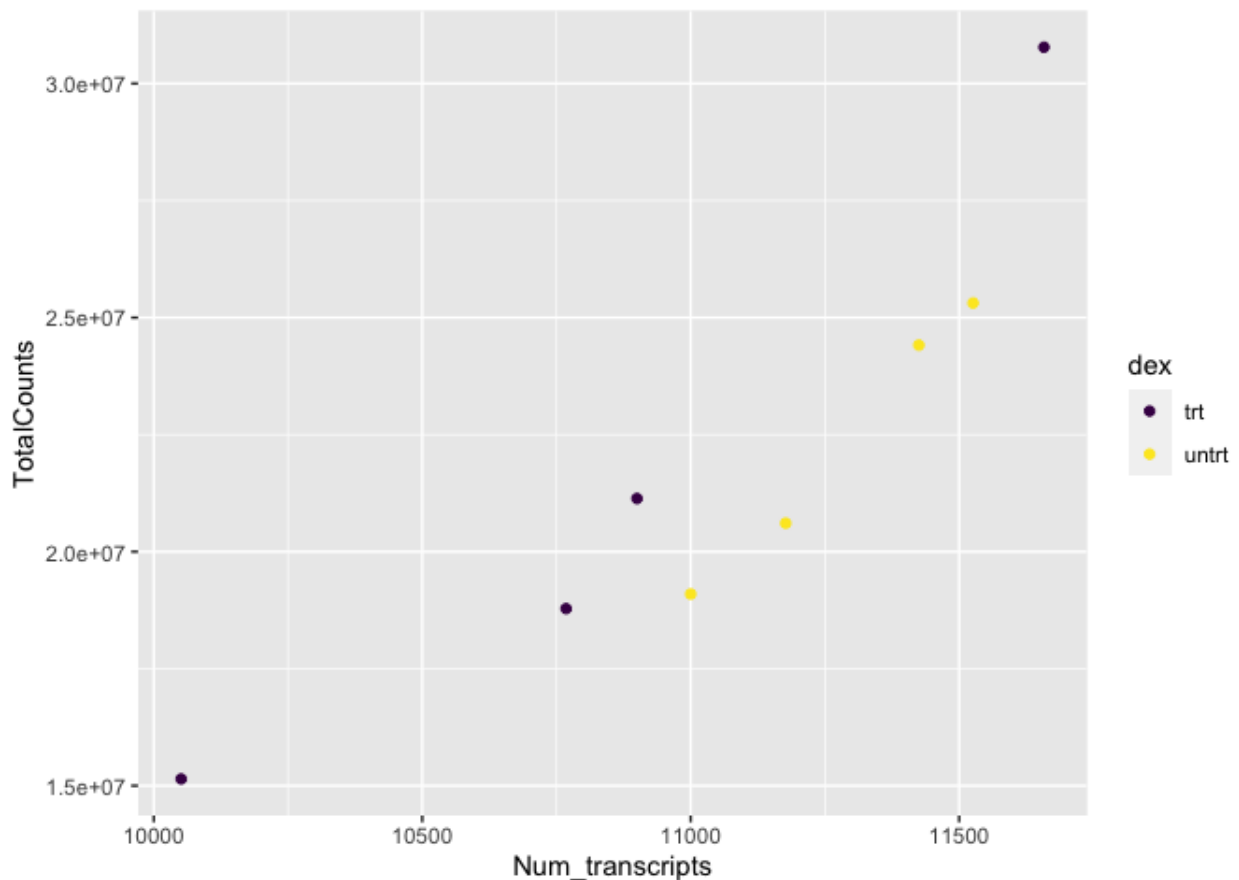
There are a number of ways to specify the color argument including by name, number, and hex code. [Here](https://www.r-graph-gallery.com/ggplot2-color.html) (<https://www.r-graph-gallery.com/ggplot2-color.html>) is a great resource from the [R Graph Gallery](https://www.r-graph-gallery.com/index.html)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

You should remember this plot from our reshaping example. The gene counts in the `scaled_counts` data were scaled to account for technical and composition differences using the trimmed mean of M values (TMM) from EdgeR (Robinson and Oshlack 2010). We can compare scaled vs unscaled counts by sample easily using faceting.

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

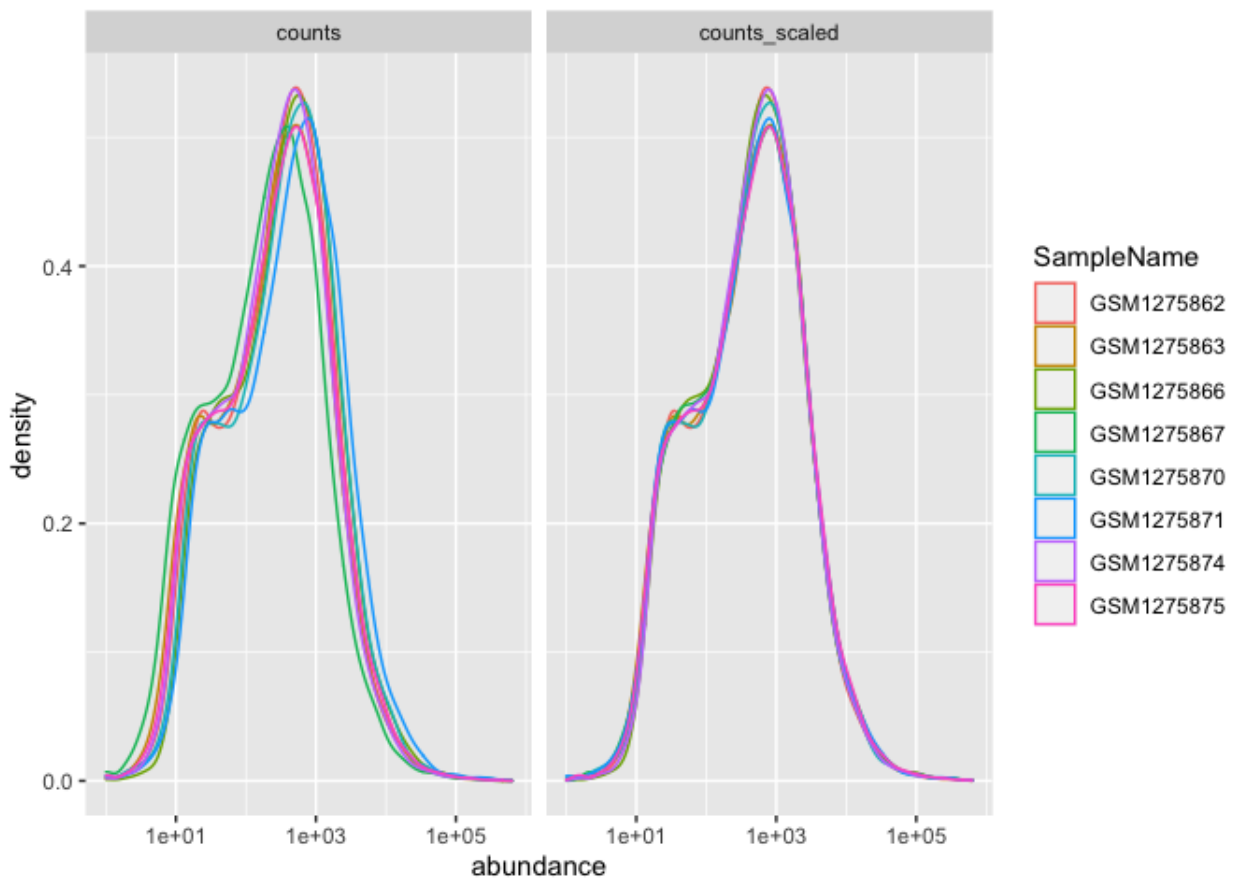
```
head(density_data)
```

```
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 transformatio
```

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

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



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

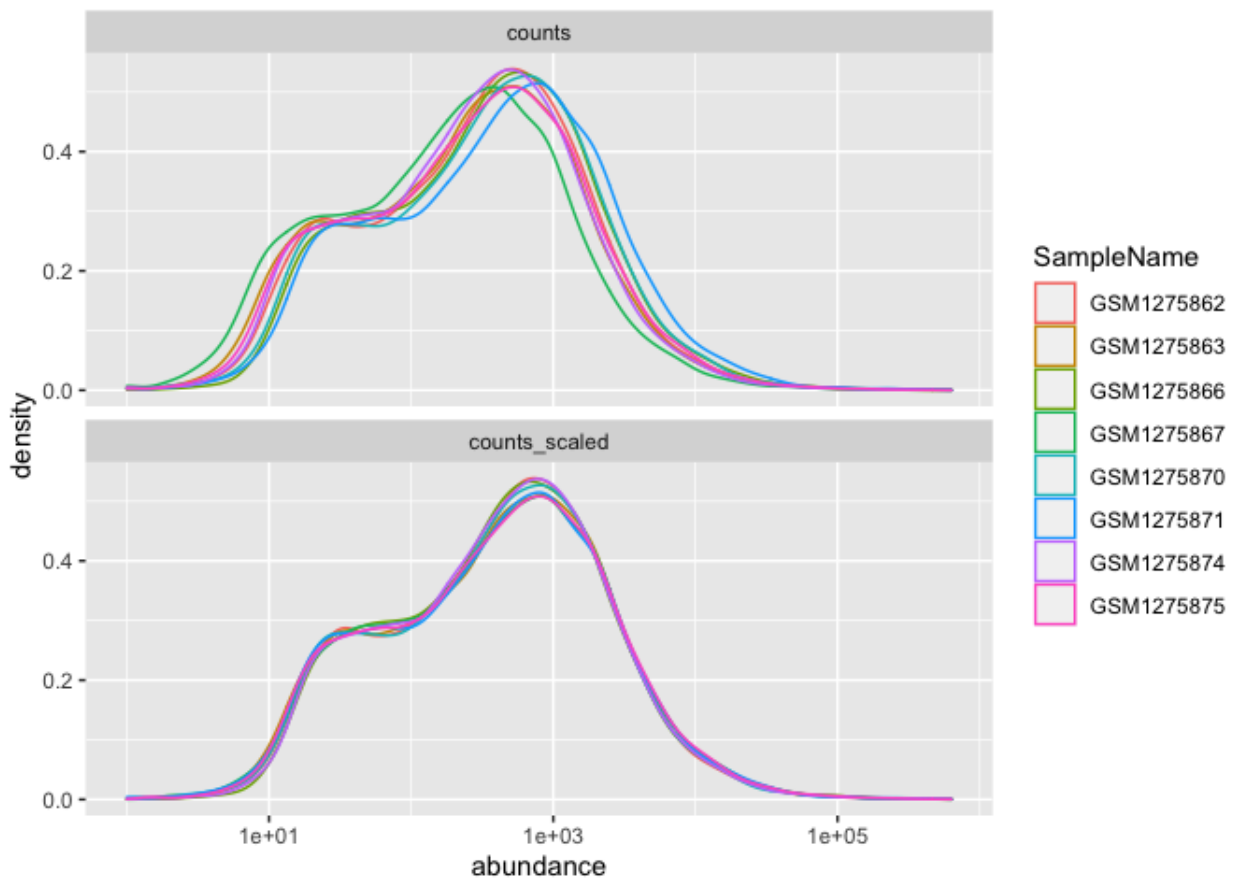
Here, faceting allowed us to visualize multiple features of our data. We were able to see count distributions by sample as well as normalized vs non-normalized counts.

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

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

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

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

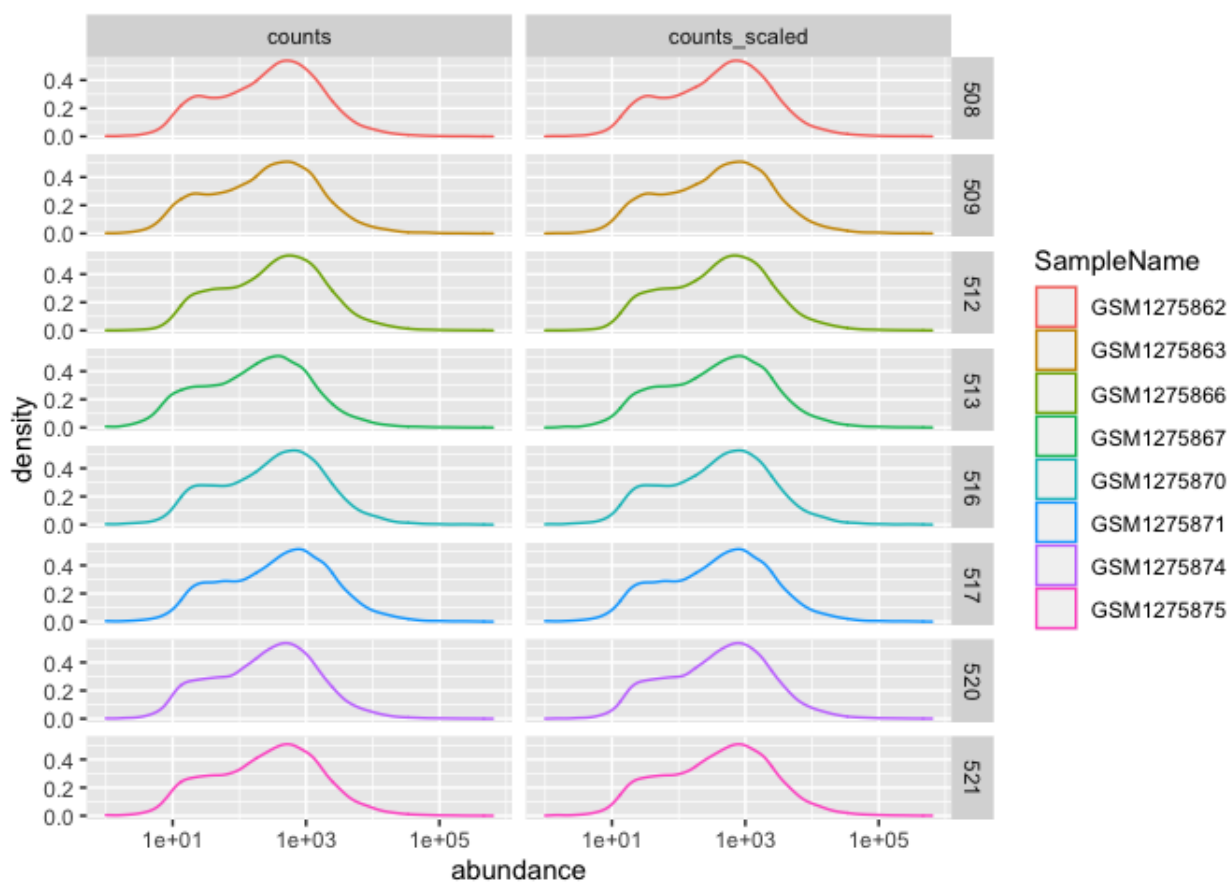


We could plot each sample individually using `facet_grid()`

```
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 x variable
```

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

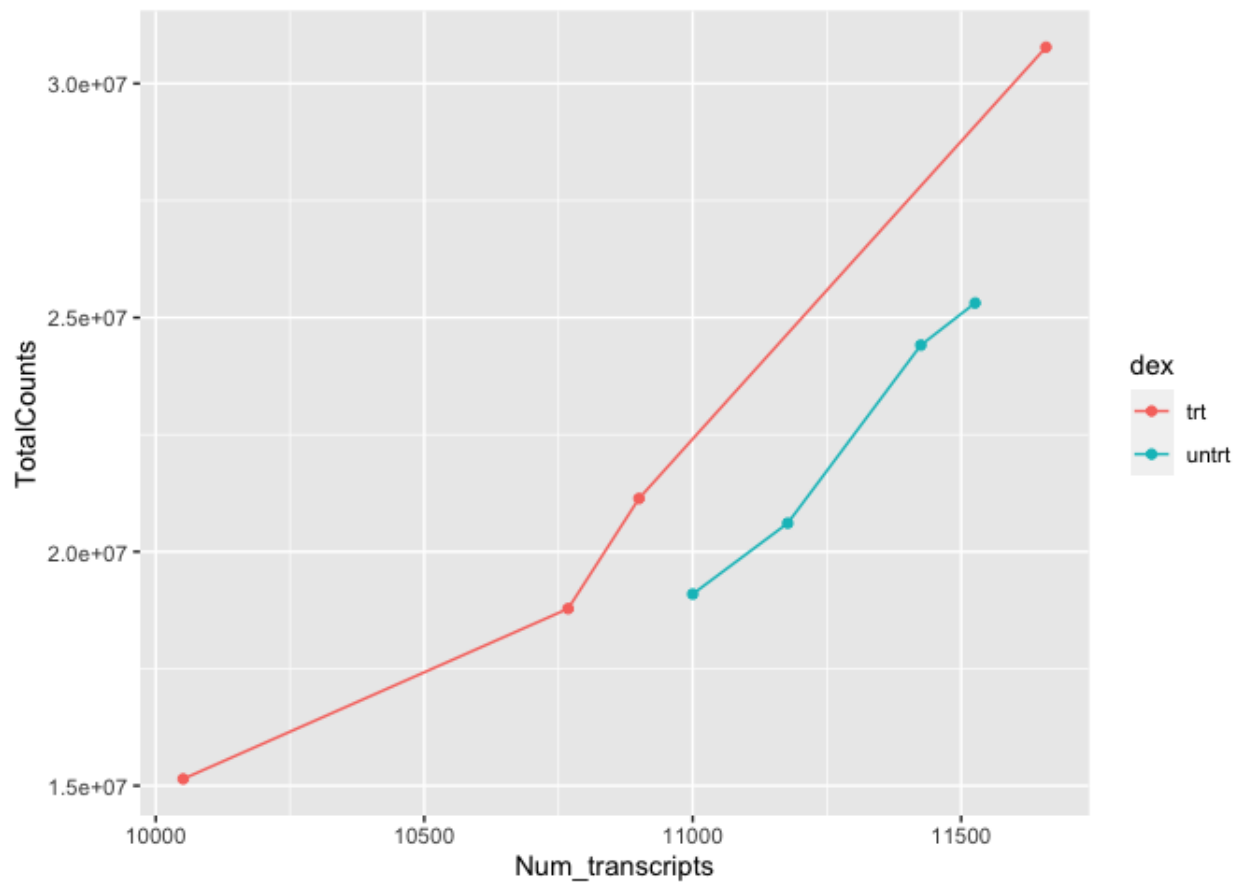


## Using multiple geoms per plot

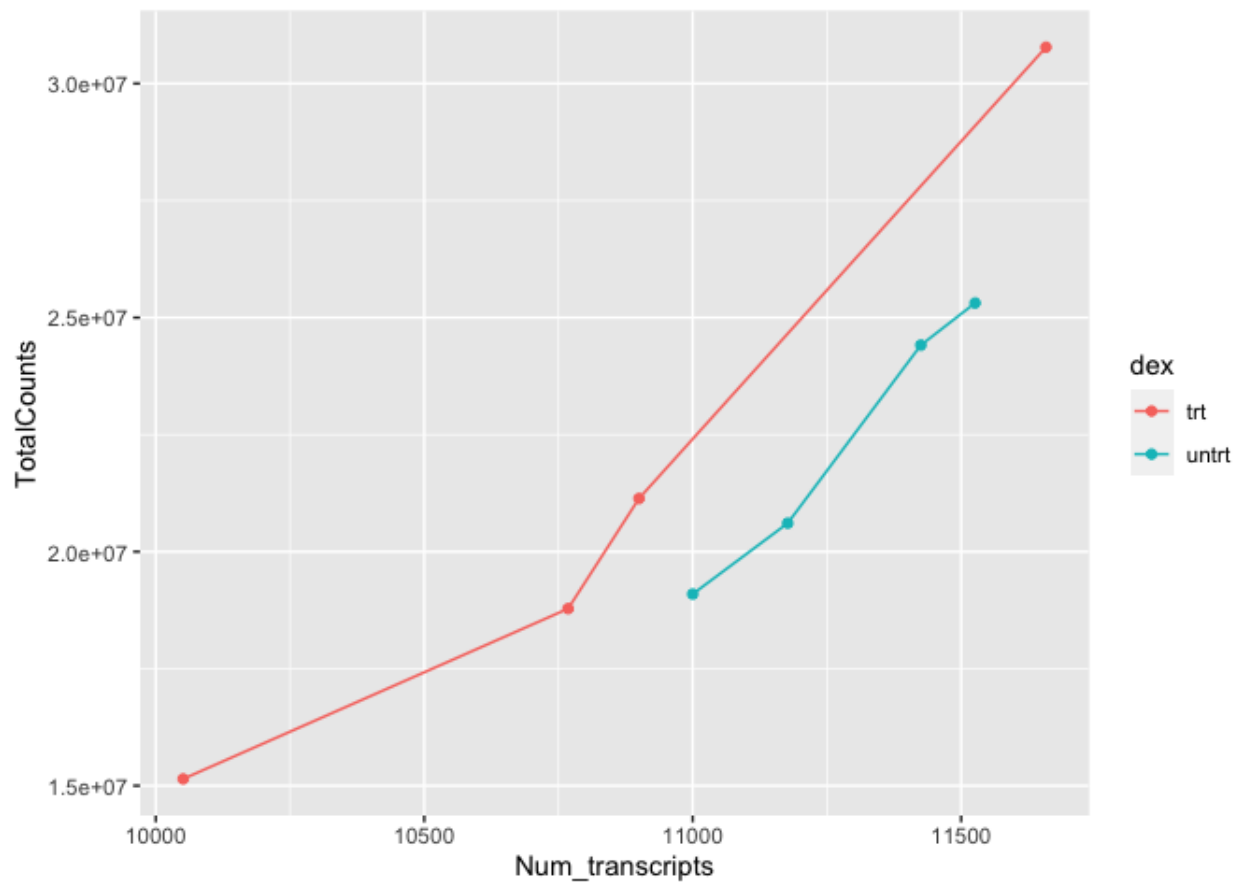
Because we build plots using layers in ggplot2. We can add multiple geoms to a plot to represent the data in unique ways.

```
#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))
```

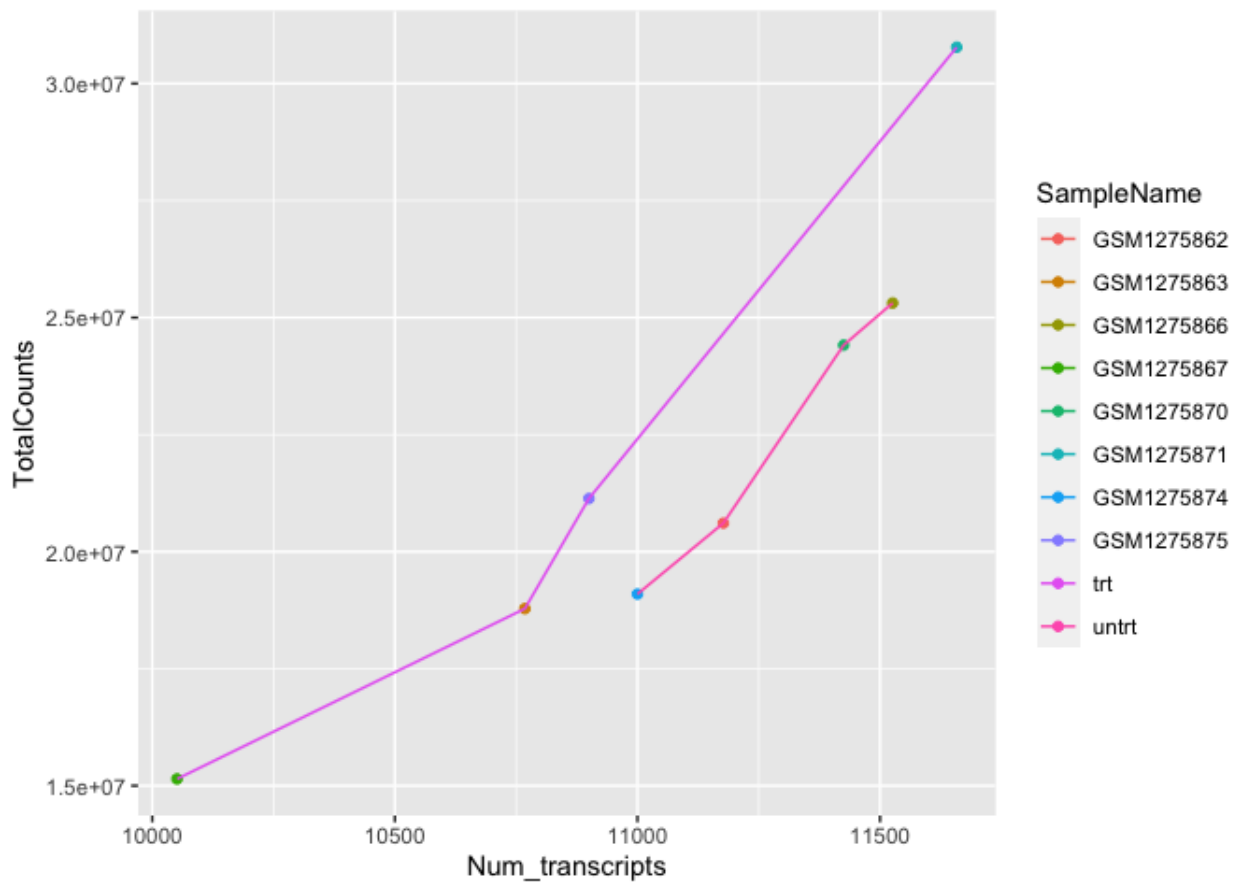




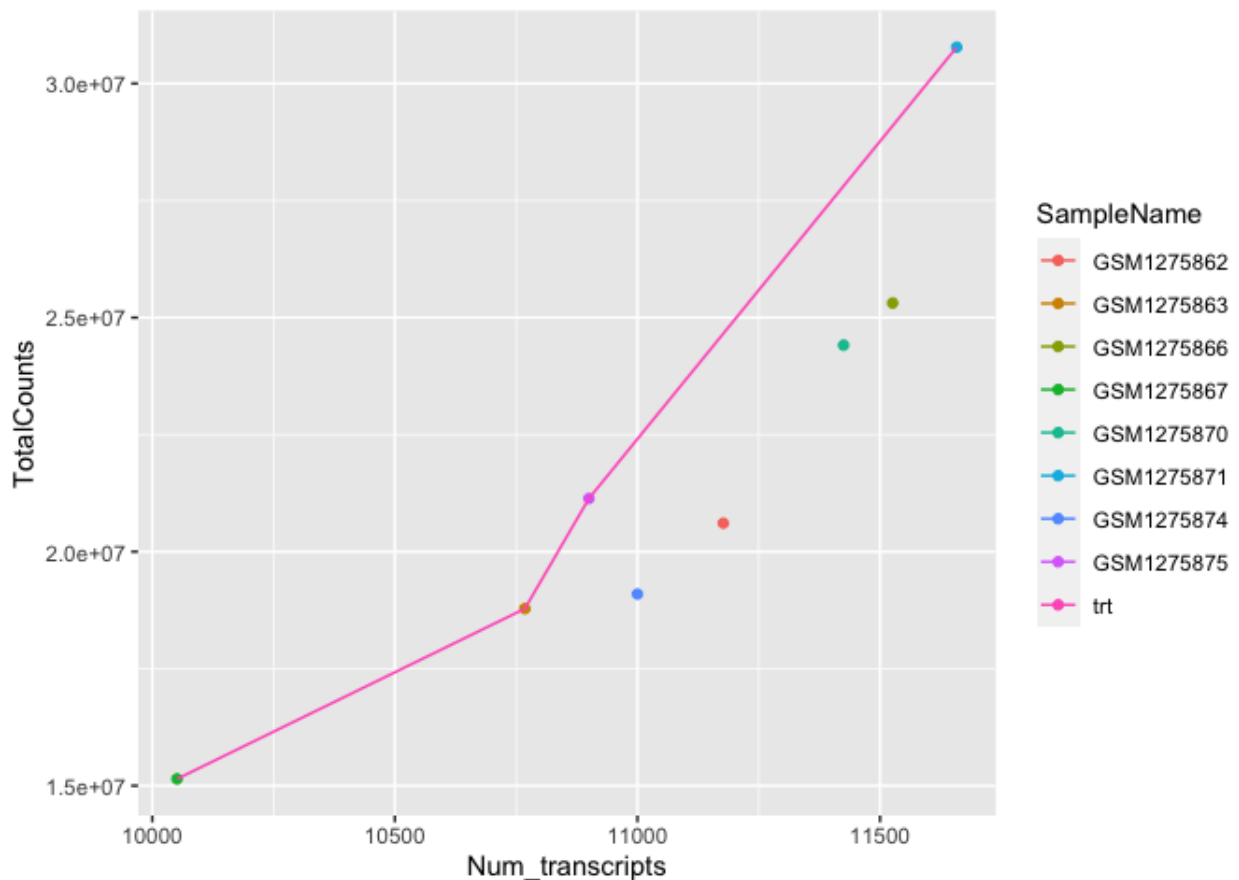
```
#to make our code more effective, we can put shared aesthetics in the
#ggplot function
ggplot(data=sc, aes(x=Num_transcripts, y = TotalCounts,color=dex)) +
 geom_point() +
 geom_line()
```



```
#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

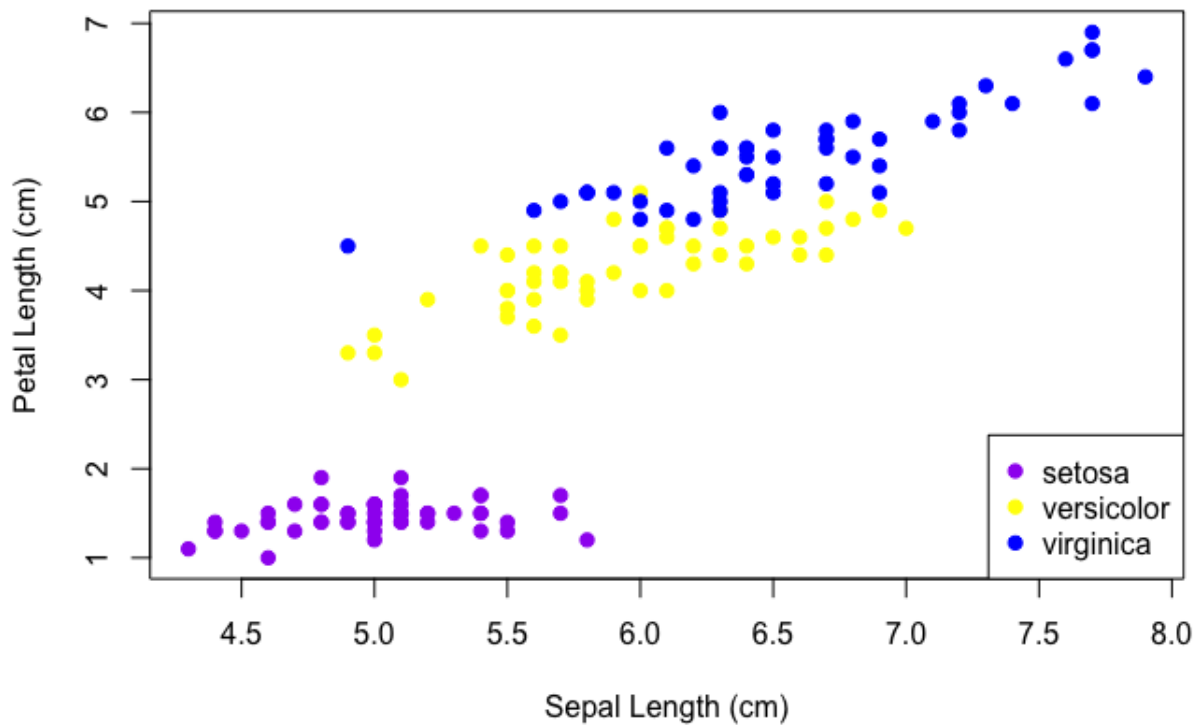
You do not need to load a package to visually explore data. Rather, you can use base R graphics for plotting (from the `graphics` package). This plotting is fairly different from `ggplot2`, which is based on the `grid` package. Unlike `ggplot2` the data does not need to be organized in a data frame to 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")`.

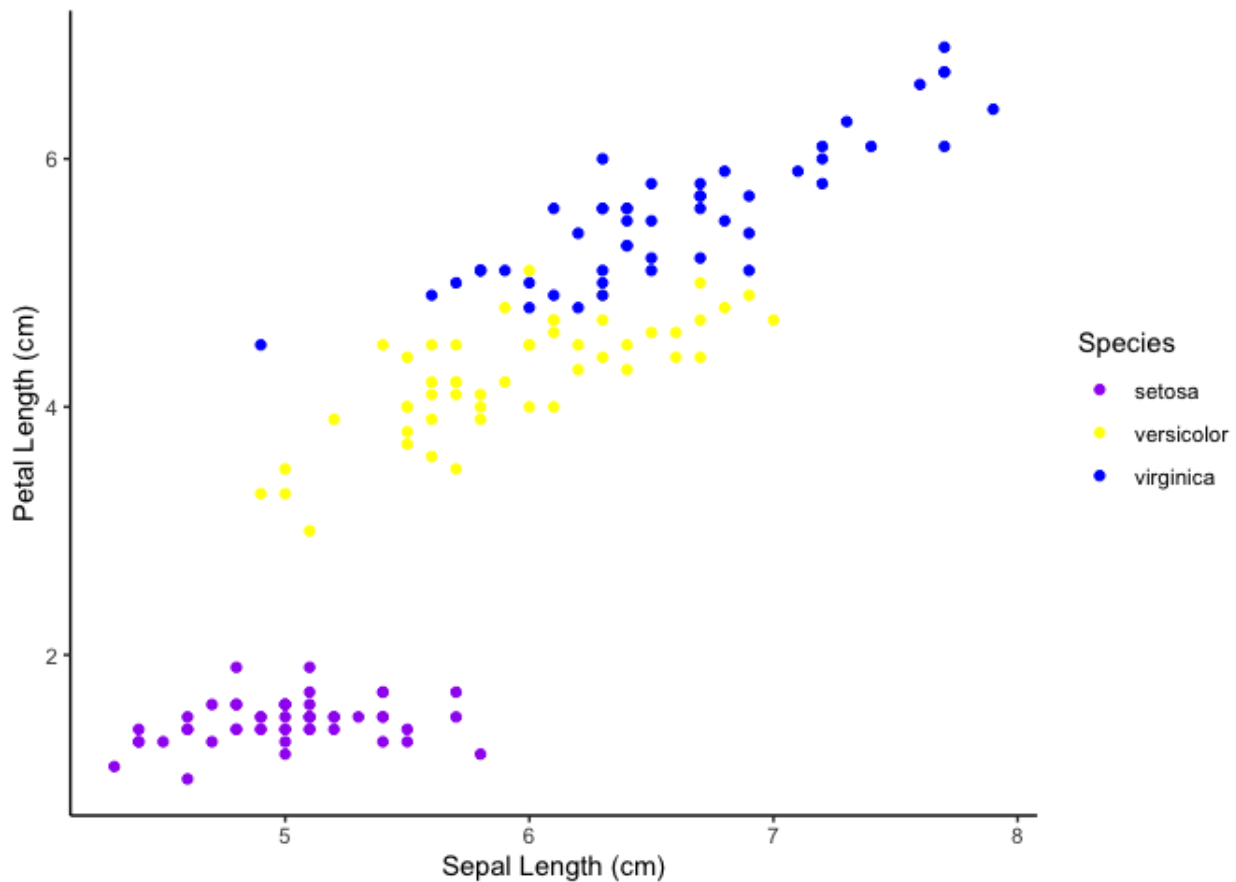
Base R graph

```
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*.

## Exporting files from RStudio

Remember, because we are using RStudio server through DNAnexus, any files created by you today will be erased at the end of the session.

To use the materials you generated on the RServer on DNAnexus on your local computer, you will need to export your files using `More` and `Export` in the `Files` pane.

## 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/>)
5. ggplot2 extensions (<https://exts.ggplot2.tidyverse.org/gallery/>)
6. ggplot2: Elegant Graphics for Data Analysis (<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.

# ggplot2 Continued

## 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.

## Reminder: Uploading files from RStudio Server

Any files created by you today will be erased at the end of the session. You can upload any files you downloaded from the last session using the Upload option in the Files pane.

## 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 packages ————— tidy
```

```
✓ ggplot2 3.3.6 ✓ purrr 0.3.4
✓ tibble 3.1.7 ✓ dplyr 1.0.9
✓ tidyr 1.2.0 ✓ stringr 1.4.0
✓ readr 2.1.2 ✓ forcats 0.5.1
```

```
— Conflicts ————— tidyverse_
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag() masks stats::lag()
```

## 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)

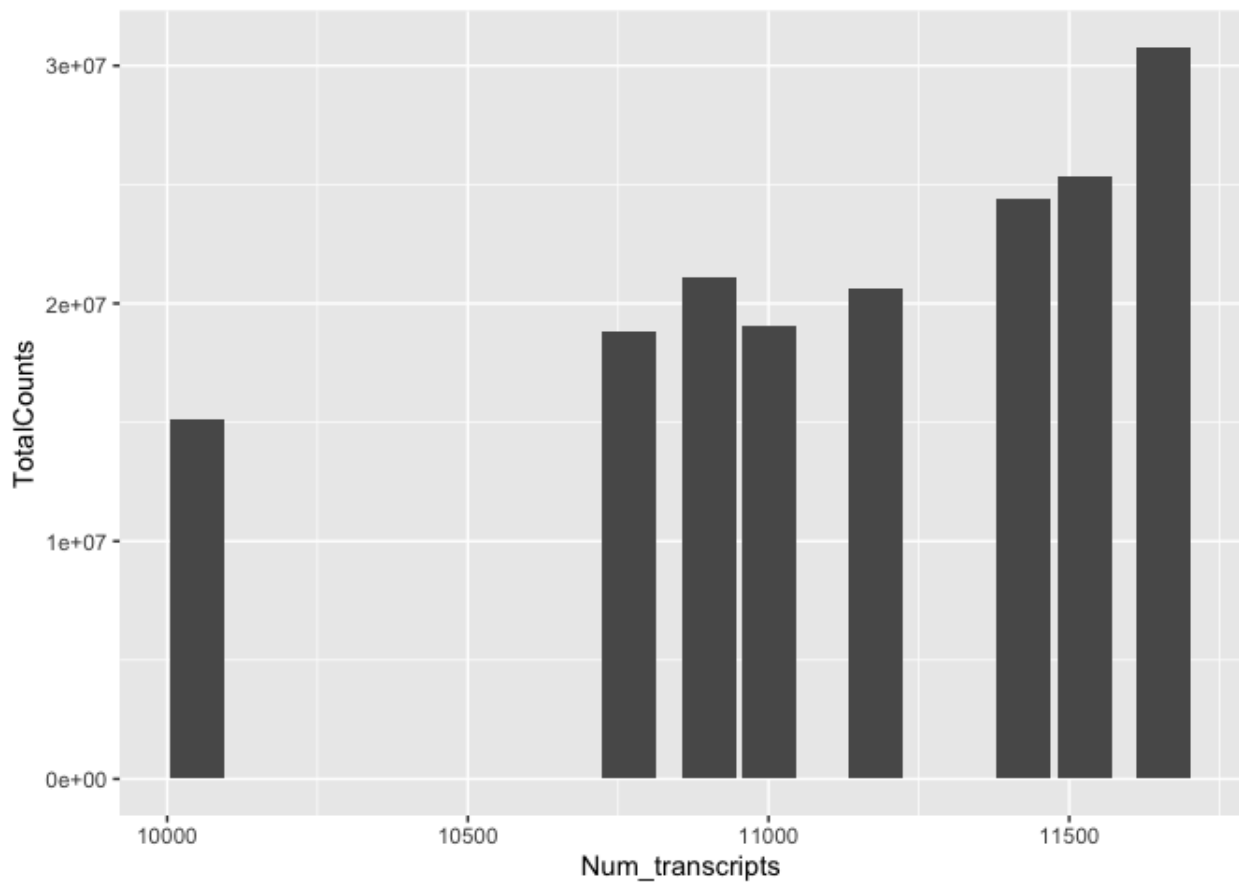
Let's plot a bar graph using the data (sc).

```
#returns an error message. What went wrong?
ggplot(data=sc) +
 geom_bar(aes(x=Num_transcripts, y = TotalCounts))
```

```
Error in `f()`:
! stat_count() can 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.

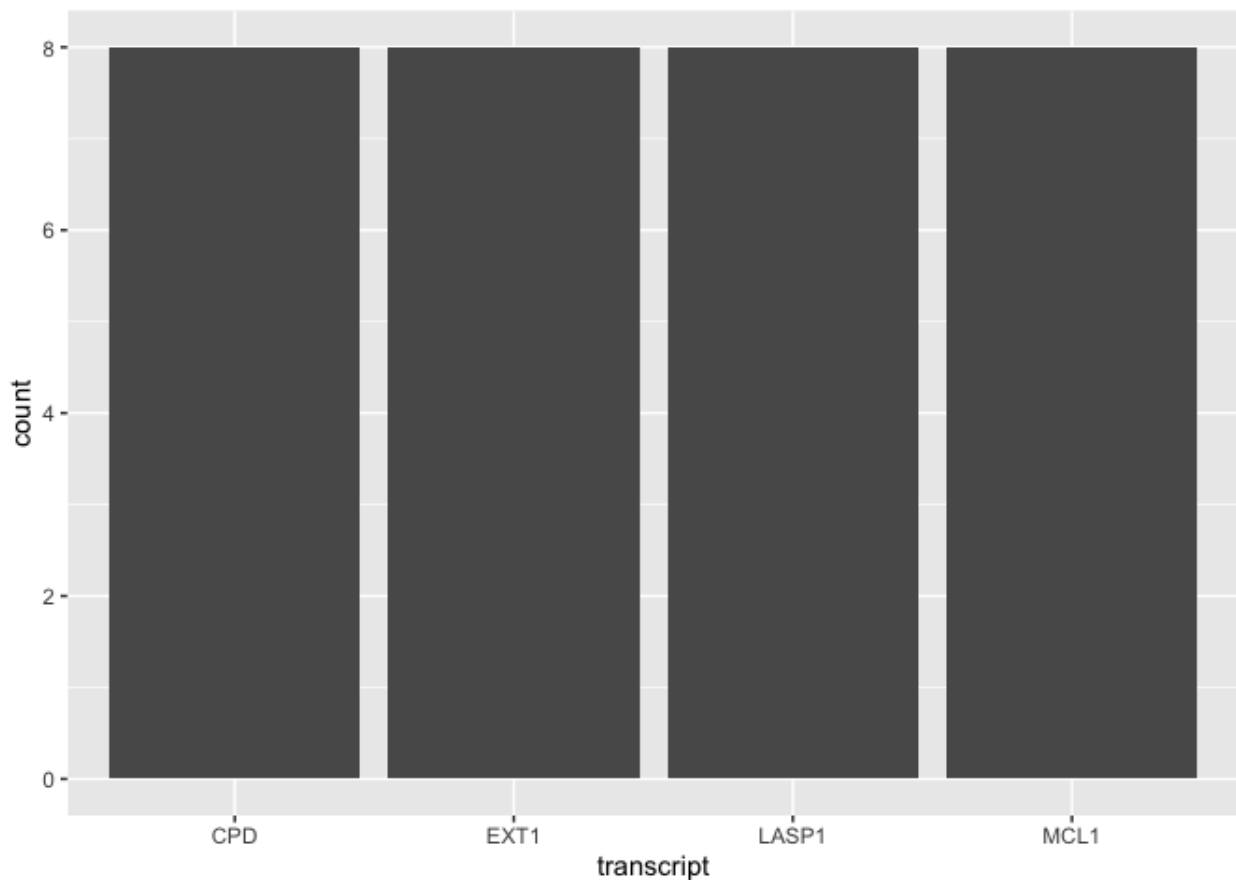
Let's look at another example.

```
#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 `f()`:
! stat_count() can only have an x or y aesthetic.
```

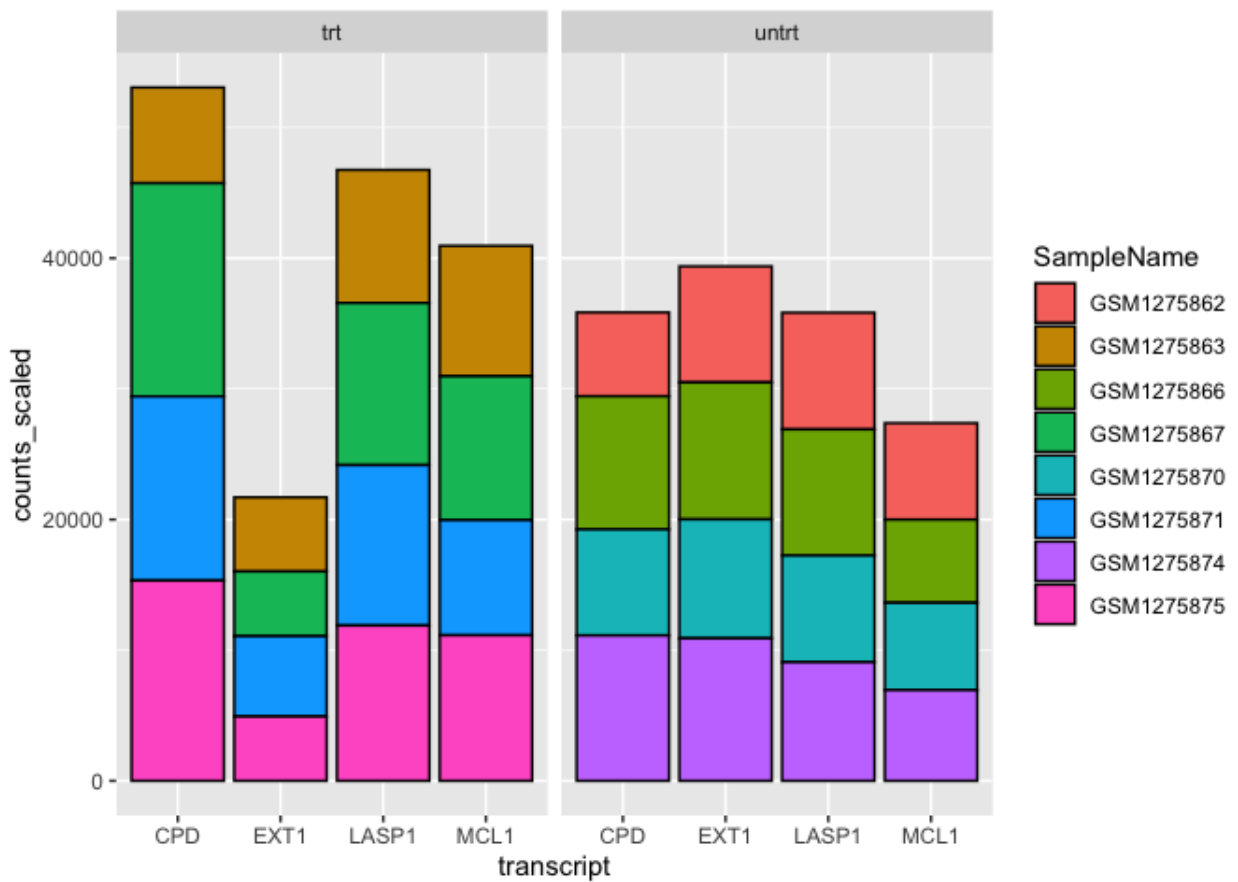
```
#Let's take away the y aesthetic
ggplot(data = interesting_trnsc) +
 geom_bar(mapping = aes(x = transcript))
```



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.

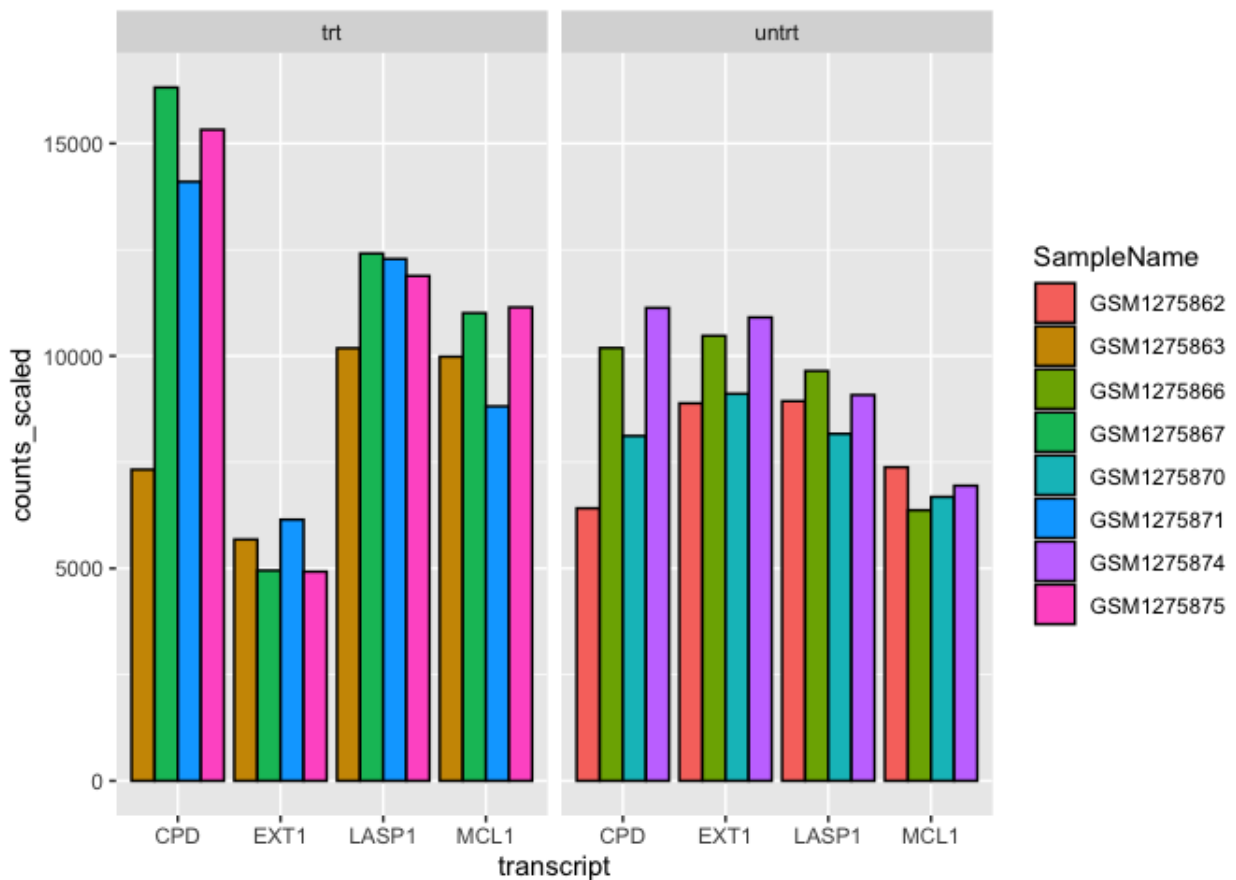
Let's change the `stat` parameter to "identity". This will plot the raw values of the normalized counts rather than how many rows are present for each transcript.

```
#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()`).

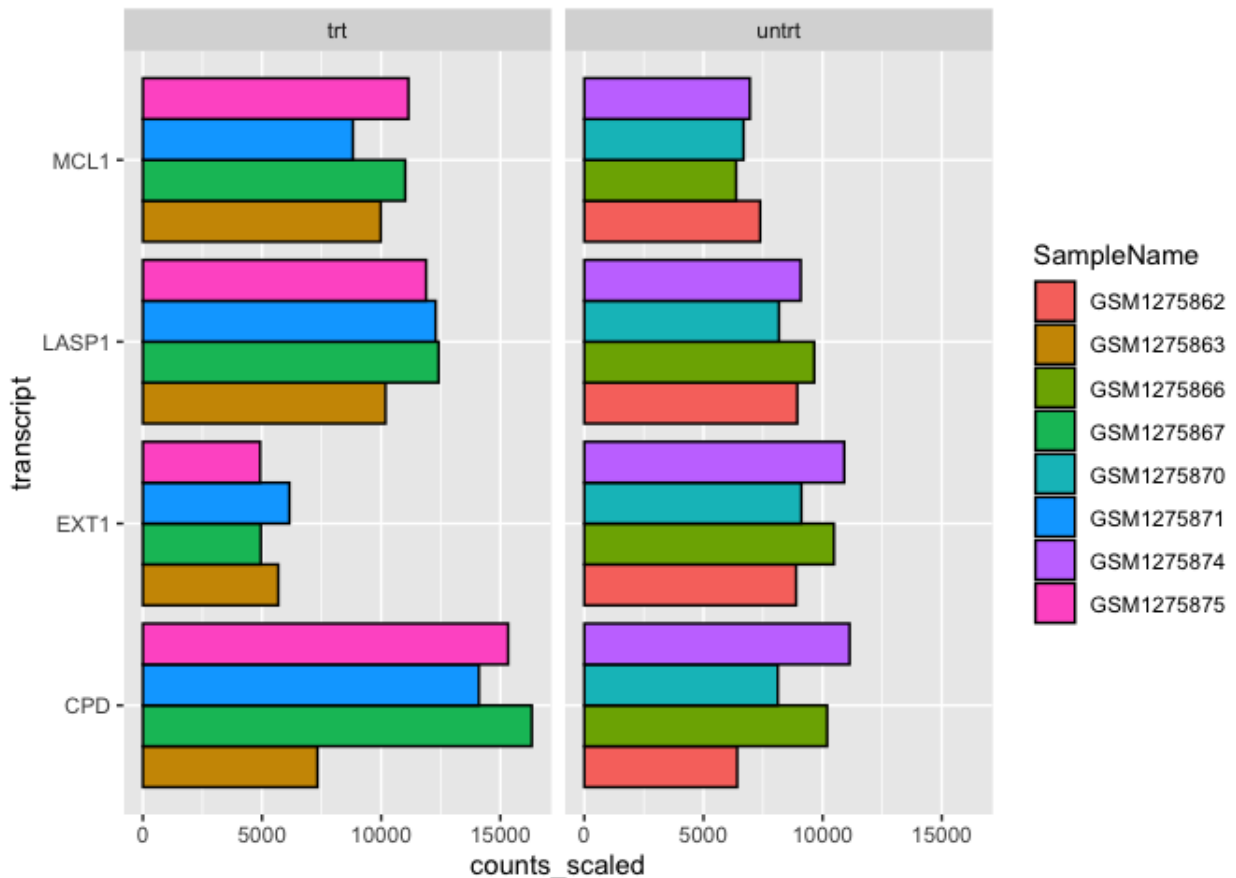
```
#let's return to our bar plot above
#get horizontal bars instead of vertical bars

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>).

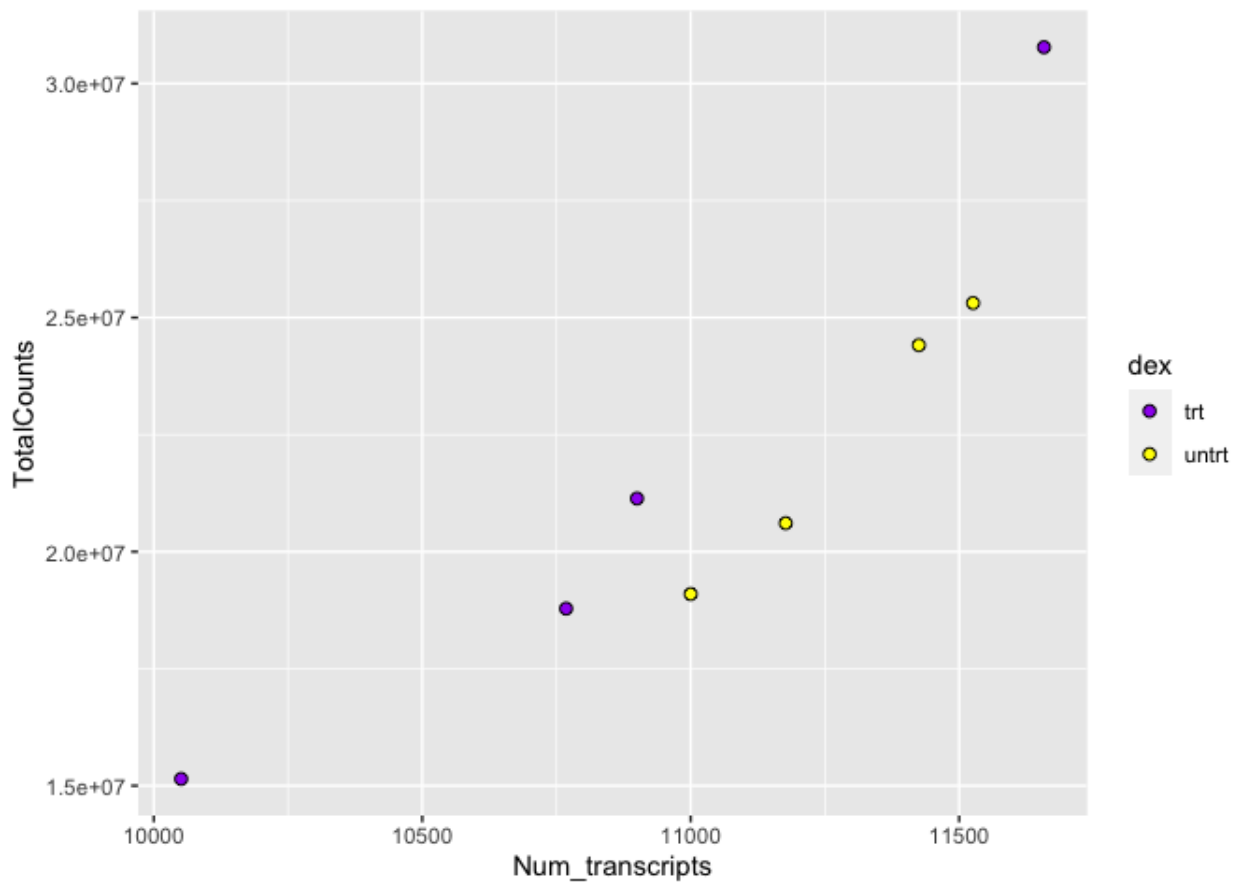
The default axes and legend titles come from the ggplot2 code.

```

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

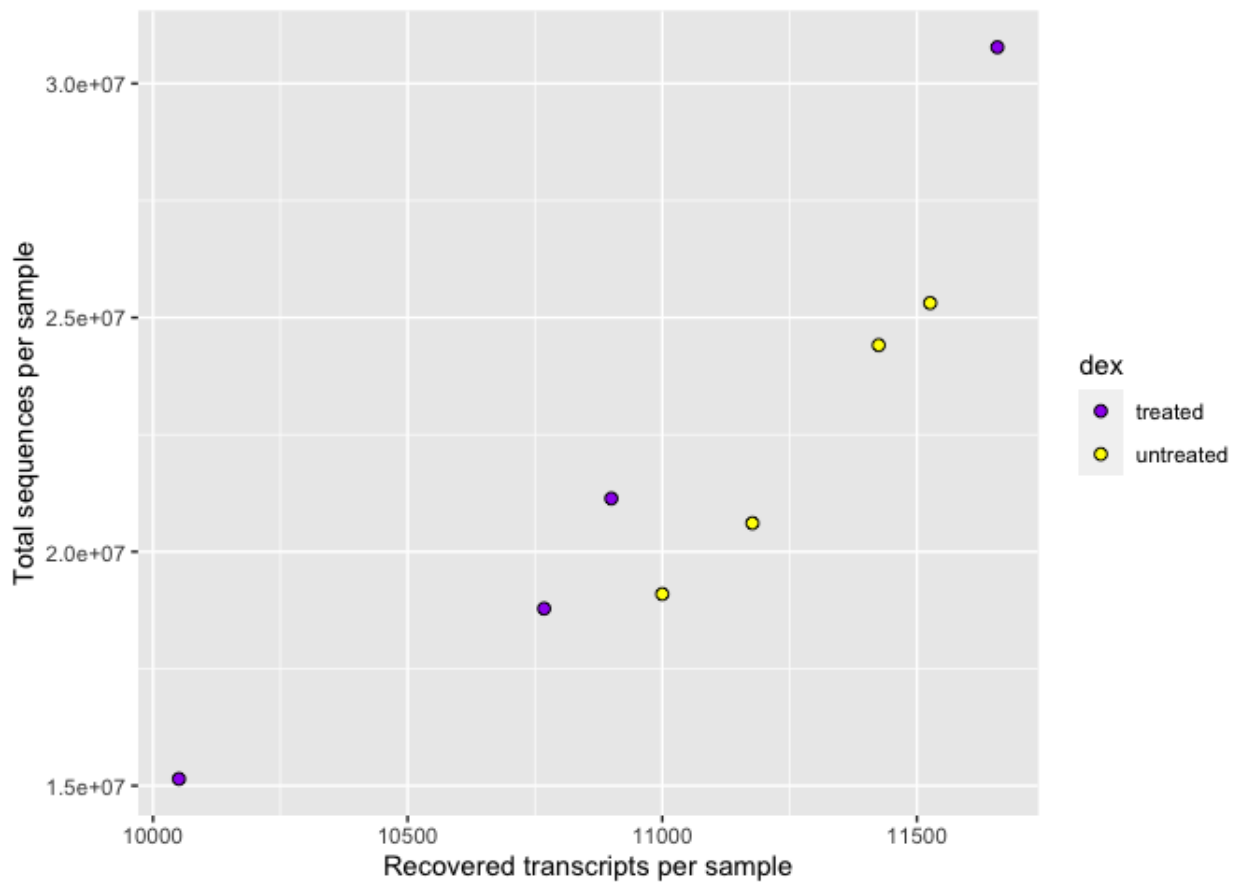
```





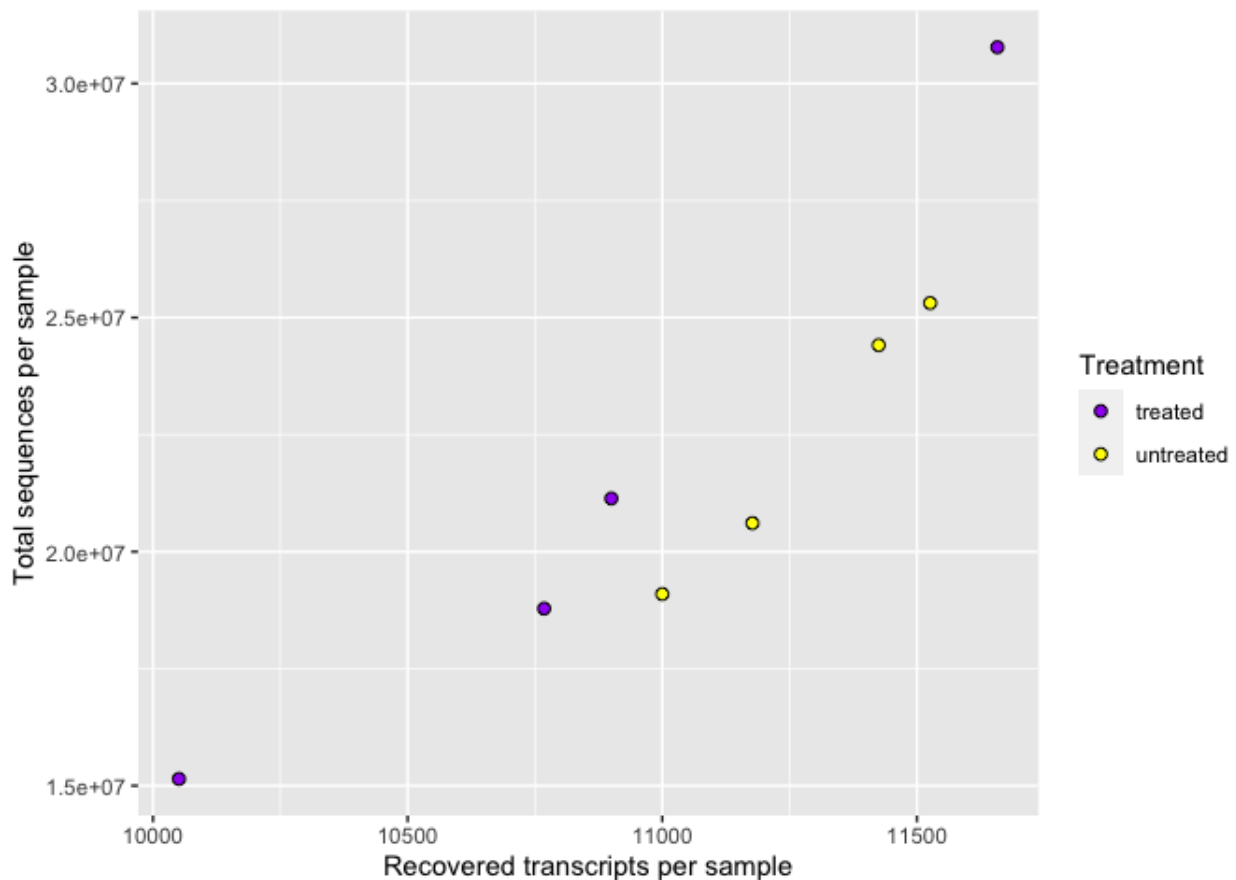
In the above plot, the y-axis label (TotalCounts) is the variable name mapped to the y aesthetic, while the x-axis label (Num\_transcripts) is the variable name named to the x aesthetic. The fill aesthetic was set equal to "dex", and so this became the default title of the fill legend. We can change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



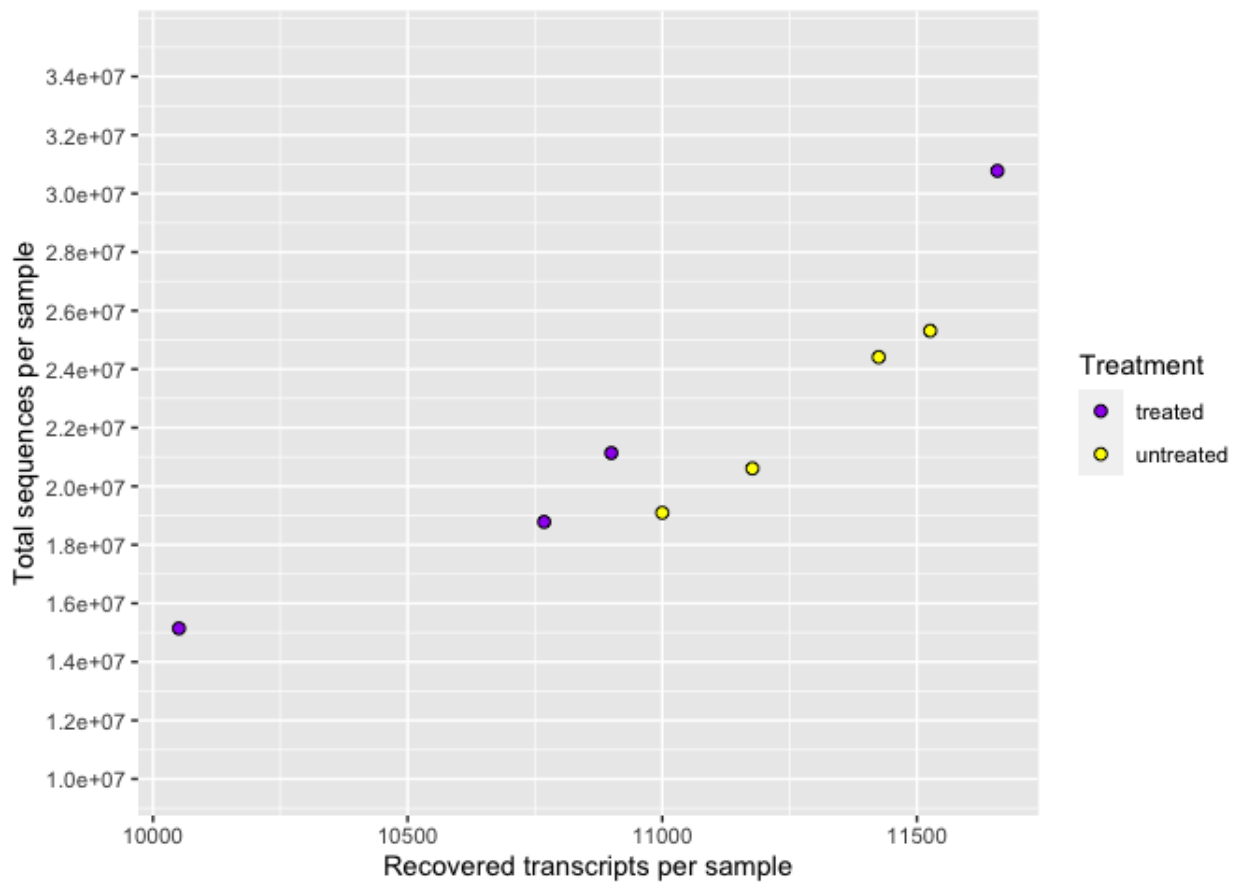
Let's change the legend title.

```
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"))
```

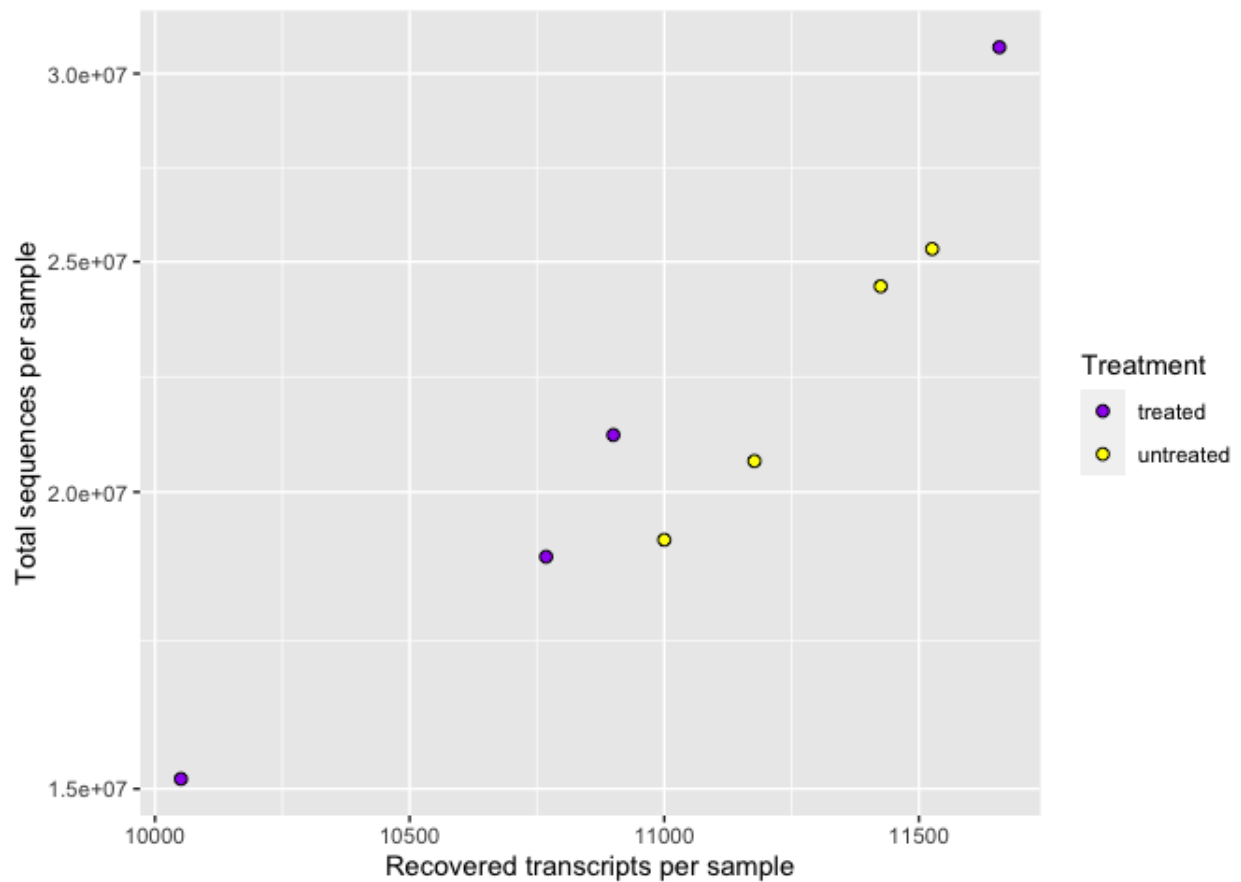


We can modify the axes scales of continuous variables using `scale_x_continuous()` and `scale_y_continuous()`. Discrete (categorical variable) axes can be modified using `scale_x_discrete()` and `scale_y_discrete()`.

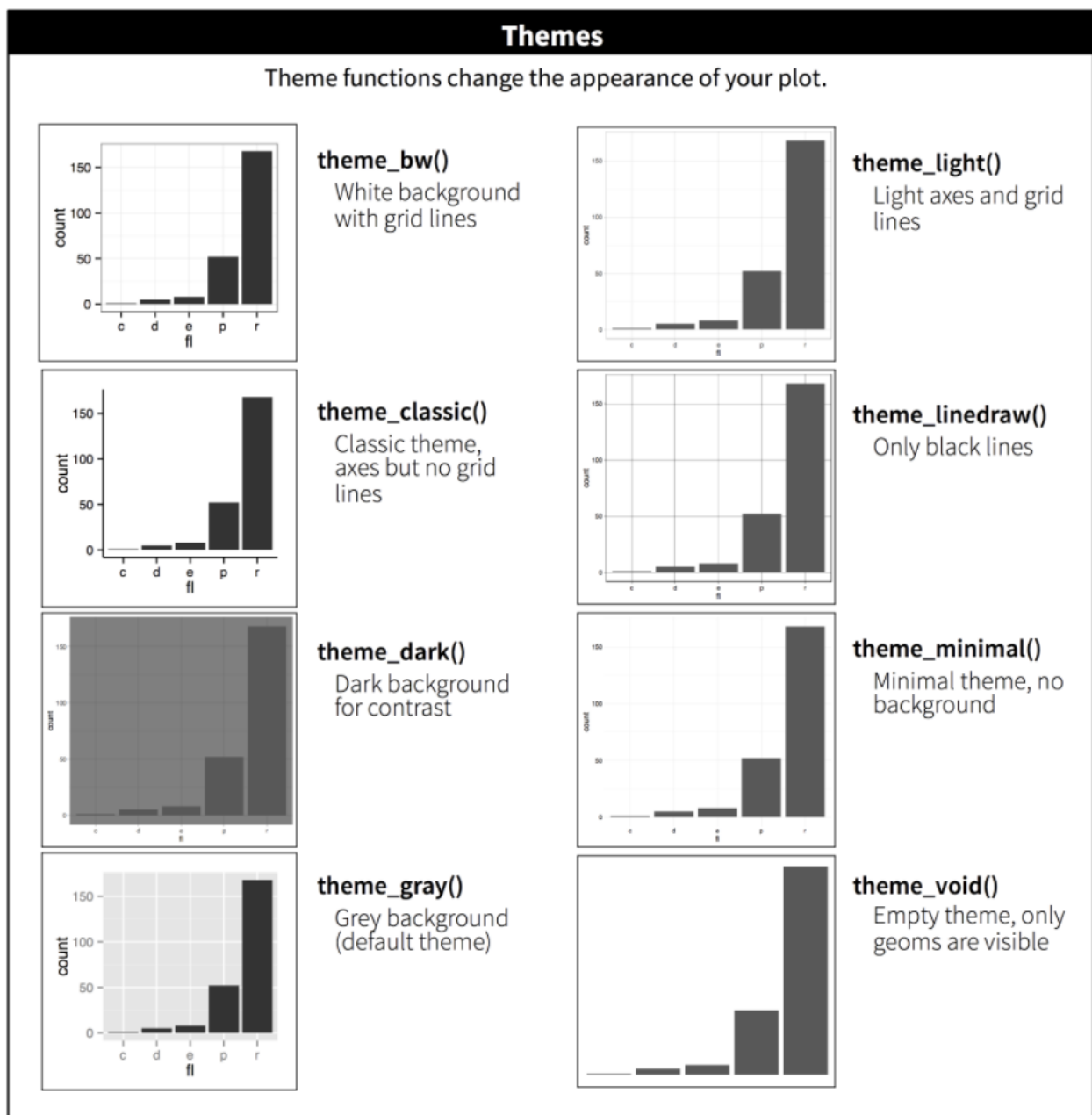
```
ggplot(data=sc) +
 geom_point(aes(x=Num_transcripts, y = TotalCounts,fill=dex),
 shape=21,size=2) +
 scale_fill_manual(values=c("purple", "yellow"),
 labels=c('treated','untreated'))+
 #can change labels of fill levels along with colors
 xlab("Recovered transcripts per sample") + #add x label
 ylab("Total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



```
#maybe we want this 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
```



Finally, we can change the overall look of non-data elements of our plot (titles, labels, fonts, background, gridlines, 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.

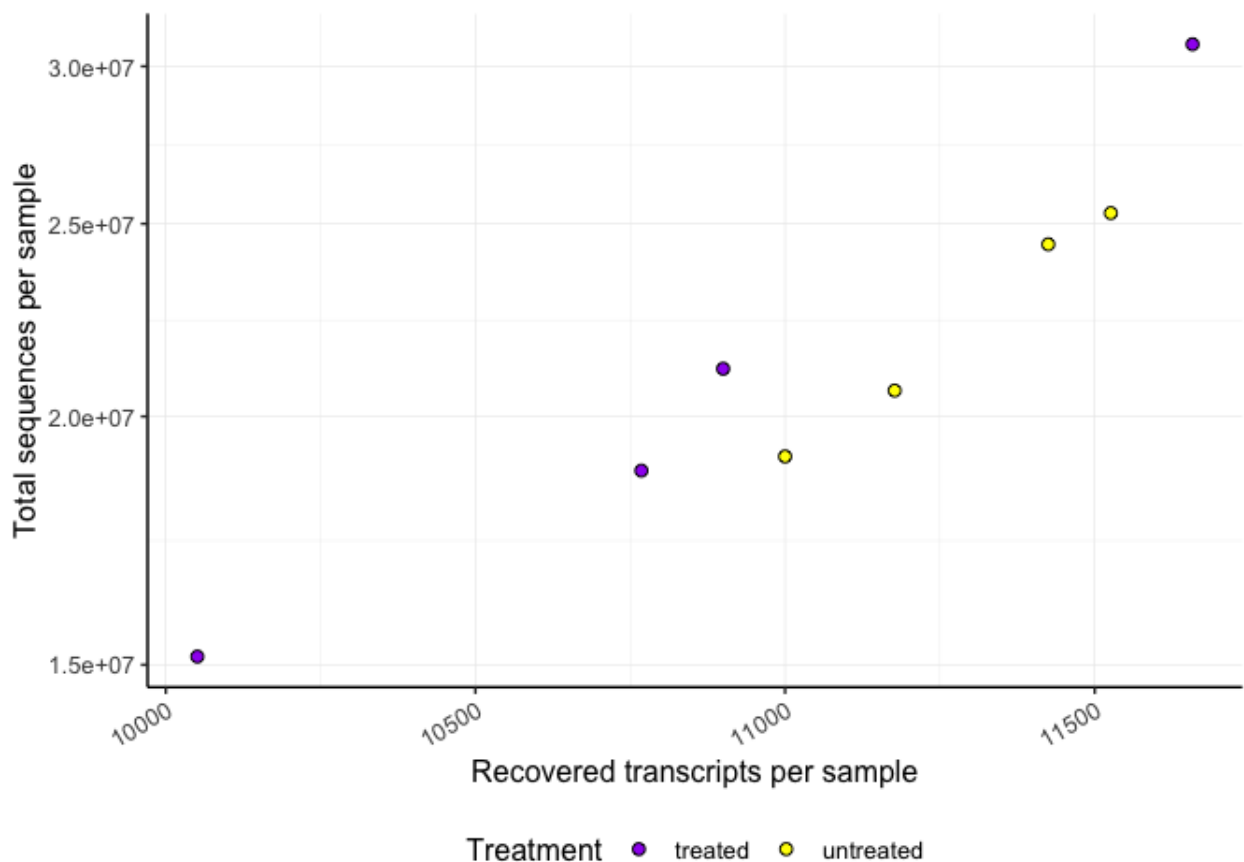
```
#Setting a theme
my_theme <-
 theme_bw() +
 theme(
 panel.border = element_blank(),
 axis.line = element_line(),
 panel.grid.major = element_line(size = 0.2),
 panel.grid.minor = element_line(size = 0.1),
 text = element_text(size = 12),
 legend.position = "bottom",
 axis.text.x = element_text(angle = 30, hjust = 1, vjust = 1)
```

```

)

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[c(1:6)]
```

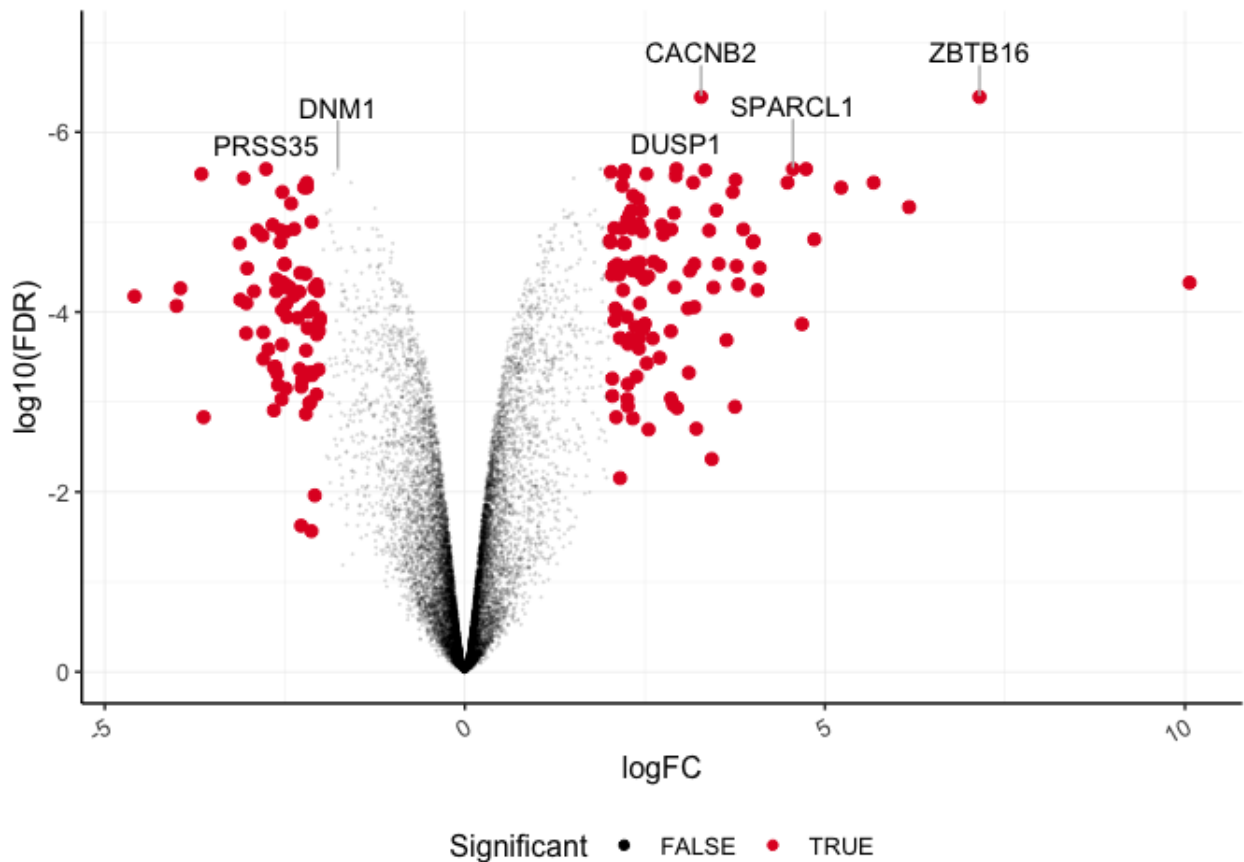
Plot

```
#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.
```





## 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
  - 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

Do not try to convey too much information in the same plot

6.

- 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.

## Exporting files from RStudio

Remember, because we are using RStudio server through DNAnexus, any files created by you today will be erased at the end of the session.

To use the materials you generated on the RServer on DNAnexus on your local computer, you will need to export your files using `More` and `Export` in the `Files` pane.

## 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](https://stemangiola.github.io/bioc2021_tidytranscriptomics/articles/tidytranscriptomics.html)) by Maria Doyle and Stefano Mangiola.

# Bioconductor and Rmarkdown

## Objectives

1. To explore Bioconductor, a repository for R packages related to biological data analysis.
2. To generate high quality data reports using R Markdown to make data analysis more reproducible.

## Reminder: Uploading files from RStudio Server

Any files created by you today will be erased at the end of the session. You can upload any files you downloaded from the last session using the Upload option in the Files pane.

## Introducing Bioconductor

Bioconductor (<https://bioconductor.org/>) is a repository for R packages related to biological data analysis, primarily bioinformatics and computational biology, and as such it is a great place to search for -omics packages and pipelines.

### Package types

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 \(https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor/index.html\)](https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor/index.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 \(https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor/index.html\)](https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor/index.html).

The latest version of Bioconductor (Bioconductor 3.16) works with R version 4.2 for complete implementation. 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 or any specific package.

To install the Bioconductor core packages, use the following:

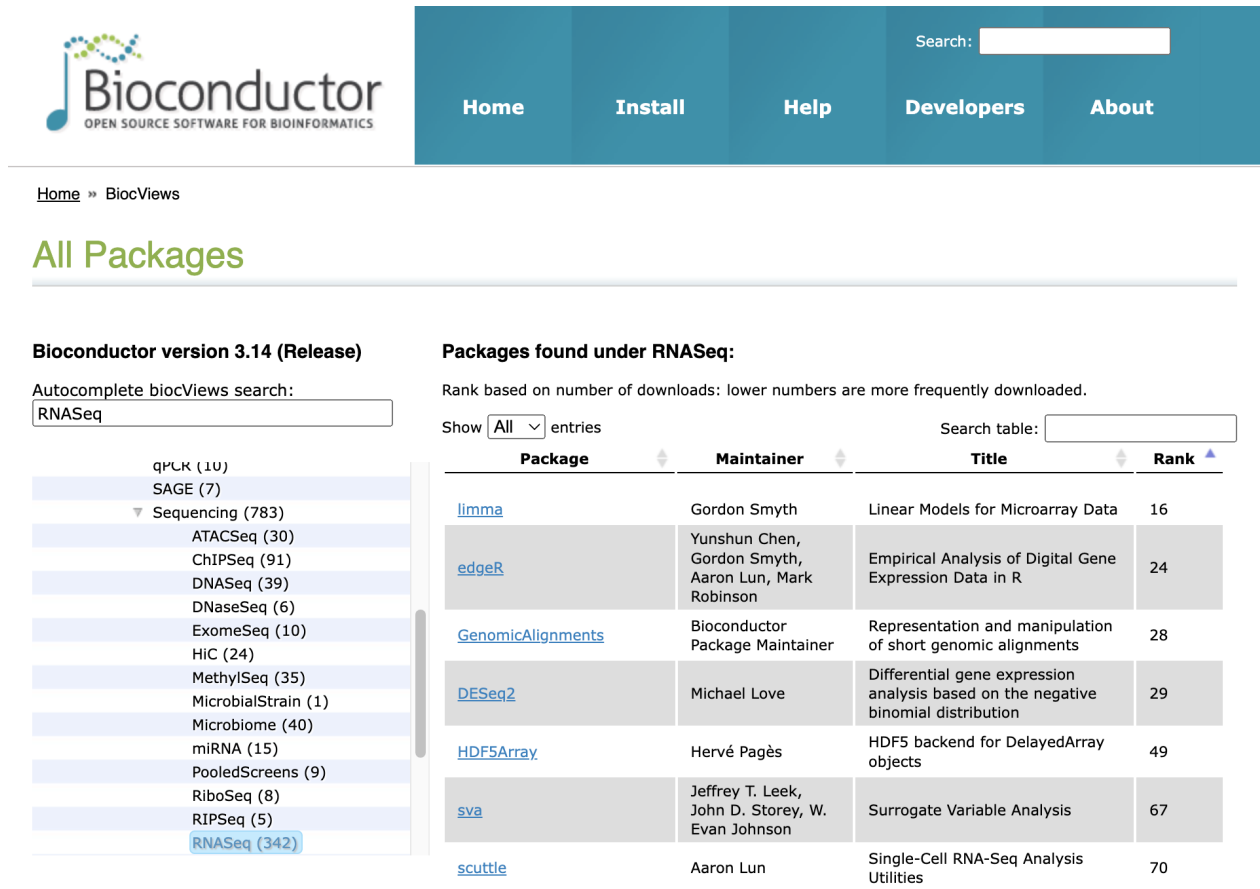
```
#install core packages
if(!require("BiocManager", quietly = TRUE))
 install.packages("BiocManager")
BiocManager::install()

###if you just want to install BiocManager use:
install.packages("BiocManager")
```

To install a specific package:

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

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](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. Here is an example searching for an RNAseq related package:



The screenshot shows the Bioconductor website interface. At the top left is the Bioconductor logo with the tagline "OPEN SOURCE SOFTWARE FOR BIOINFORMATICS". To the right is a navigation bar with links for Home, Install, Help, Developers, and About. A search bar is located in the top right corner. Below the navigation bar, the breadcrumb "Home » BiocViews" is visible. The main heading is "All Packages". Underneath, there are two sections: "Bioconductor version 3.14 (Release)" and "Packages found under RNASeq:". The "Bioconductor version 3.14 (Release)" section includes an "Autocomplete biocViews search:" field with "RNASeq" entered, and a list of categories with their respective package counts: qPCR (10), SAGE (7), Sequencing (783), ATACSeq (30), ChIPSeq (91), DNaseSeq (39), DNaseSeq (6), ExomeSeq (10), HIC (24), MethylSeq (35), MicrobialStrain (1), Microbiome (40), miRNA (15), PooledScreens (9), RiboSeq (8), RIPSeq (5), and RNASeq (342). The "Packages found under RNASeq:" section includes a note "Rank based on number of downloads: lower numbers are more frequently downloaded." and a "Show All entries" dropdown. Below this is a table of search results.

| Package                           | Maintainer                                           | Title                                                                             | Rank |
|-----------------------------------|------------------------------------------------------|-----------------------------------------------------------------------------------|------|
| <a href="#">lirmma</a>            | Gordon Smyth                                         | Linear Models for Microarray Data                                                 | 16   |
| <a href="#">edgeR</a>             | Yunshun Chen, Gordon Smyth, Aaron Lun, Mark Robinson | Empirical Analysis of Digital Gene Expression Data in R                           | 24   |
| <a href="#">GenomicAlignments</a> | Bioconductor Package Maintainer                      | Representation and manipulation of short genomic alignments                       | 28   |
| <a href="#">DESeq2</a>            | Michael Love                                         | Differential gene expression analysis based on the negative binomial distribution | 29   |
| <a href="#">HDF5Array</a>         | Hervé Pagès                                          | HDF5 backend for DelayedArray objects                                             | 49   |
| <a href="#">sva</a>               | Jeffrey T. Leek, John D. Storey, W. Evan Johnson     | Surrogate Variable Analysis                                                       | 67   |
| <a href="#">scuttle</a>           | Aaron Lun                                            | Single-Cell RNA-Seq Analysis Utilities                                            | 70   |

As you can see, the most popular packages are listed first.

## 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 in the [Events Calendar](http://www.bioconductor.org/help/events/) (<http://www.bioconductor.org/help/events/>).

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

## Learn »

### Master *Bioconductor* tools

- [Courses](#)
- [Education and Training](#)
- [Support site](#)
- [Package vignettes](#)
- [Literature citations](#)
- [Common work flows](#)
- [FAQ](#)
- [Community resources](#)
- [Videos](#)
- *Bioconductor* [Community Blog](#)

## 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.

## Introducing R Markdown

For the purposes of reproducibility or collaboration, it is good practice to generate a report summarizing what has been done along with output results. This saves collaborators or your future self from trying to figure out how results were generated or from which script they were generated. Fortunately, there is `rmarkdown` for easy reporting of R code, results, and interpretation. R Markdown is integrated within RStudio, and the `rmarkdown` package can be installed using the following:

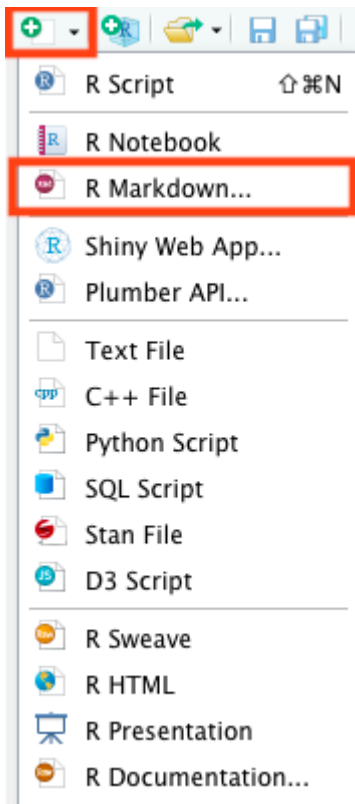
```
install.packages("rmarkdown")
```

In addition, R Markdown reports are dynamic, and as code is modified a new report can easily be generated using the `knitr` package, which is also integrated into RStudio. The key to `knitr` is a mixture of explanatory text with code chunks that are executed with each "knit" of the document.

```
install.packages("knitr")
```

## Creating an Rmarkdown file

To create an Rmarkdown file, select the new file icon and then R Markdown.





A box will appear prompting for an author, title, and output format. Give your document an initial title and select the output that you want. Note: this information can be modified at any time.

**New R Markdown**

Document

Presentation

Shiny

From Template

**Title:**

**Author:**

**Default Output Format:**

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

**PDF**  
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

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

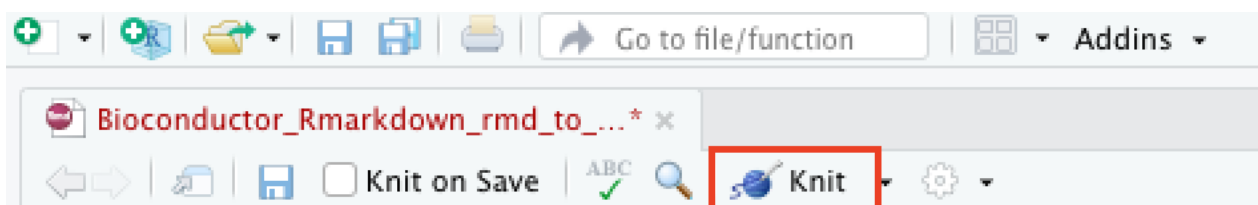
Select OK. A new R Markdown document should have been created.

```

1 ---
2 title: "Untitled"
3 author: "Alexandra Emmons"
4 date: "3/1/2022"
5 output: html_document
6 ---
7
8 ```{r setup, include=FALSE}
9 knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.
15
16 When you click the Knit button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:
17
18 ```{r cars}
19 summary(cars)
20 ```
21
22 ## Including Plots
23
24 You can also embed plots, for example:
25
26 ```{r pressure, echo=FALSE}
27 plot(pressure)
28 ```
29
30 Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.
31

```

Now you can begin generating a report. Thankfully, the document you just created includes some information to get you started, including some initial code chunks. I am NOT going to provide more detail on report generation here. There is extensive documentation only a google search away. See the resources section of this document for help. When you are ready to generate the output, whether an html, doc, or pdf, simply select the "Knit" button at the top of the page.



## What is Quarto?

You may have noticed the option for Quarto documents and presentations in new versions of RStudio. "Quarto® is an open-source scientific and technical publishing system built on Pandoc" ([quarto.org](https://quarto.org/) (<https://quarto.org/>)). It is a bit more dynamic than RMarkdown, especially regarding integration of different coding languages and report outputs.

[Quarto](https://quarto.org/) (<https://quarto.org/>) is worth learning and will likely replace RMarkdown in the future.

## Exporting files from RStudio

Remember, because we are using RStudio server through DNAnexus, any files created by you today will be erased at the end of the session.

To use the materials you generated on the RServer on DNAnexus on your local computer, you will need to export your files using `More` and `Export` in the `Files` pane.

## 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/index.html) (<https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor/index.html>) from the Carpentries Incubator.

## Resources

1. [R markdown documentation from RStudio](https://rmarkdown.rstudio.com/lesson-1.html) (<https://rmarkdown.rstudio.com/lesson-1.html>)
2. Other helpful resources, including comprehensive guides and cheatsheets can be accessed from [here](https://cran.r-project.org/web/packages/rmarkdown/vignettes/rmarkdown.html) (<https://cran.r-project.org/web/packages/rmarkdown/vignettes/rmarkdown.html>).

# **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
4. `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_scaledcount
```

{{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. Is there a difference in the number of transcripts with greater than 0 normalized counts (`counts_scaled`) per sample? What commands did you use to answer this question.

{{Sdet}}

Solution{{Esum}}

```
table(transcript_counts[transcript_counts$counts_scaled>0,]$samp
```

{{Edet}}

5. How many categories of transcripts are there? Think about what you know regarding factors. Why is this number much smaller than the results of question 4?

{{Sdet}}

Solution{{Esum}}

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

{{Edet}}

6. Subset `transcript_counts` to only include the following columns: `sample`, `cell`, `dex`, `transcript`, `avgLength`, `counts_scaled`. Save this new dataframe to a new object called `transc_df`.

{{Sdet}}

Solution{{Esum}}

```
transc_df <- transcript_counts[c("sample","cell","dex",
 "transcript","avgLength",
 "counts_scaled")]
```

{{Edet}}

7. Using your new data frame from question six (`transc_df`), rename the column "sample" to "Sample".

```
{{Sdet}}
```

▼

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

```
colnames(transc_df)[1]<-"Sample"
```

```
{{Edet}}
```

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

```
{{Sdet}}
```

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

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

```
{{Edet}}
```

9. Make a data frame with the column names "Mean" and "Standard\_Dev" that holds the values from question 8. Hint: check out the function `data.frame()`.

```
{{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. 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}}

2. Explore the column 'avgLength' in `df_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}}

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

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

{{Sdet}}

Solution{{Esum}}

```
top_gene_counts %>%
 group_by(dex, transcript) %>%
```

```
summarize(mean_counts=mean(counts_scaled),
 sd=sd(counts_scaled),
 median=median(counts_scaled))
```

{{Edet}}

6. 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: 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.

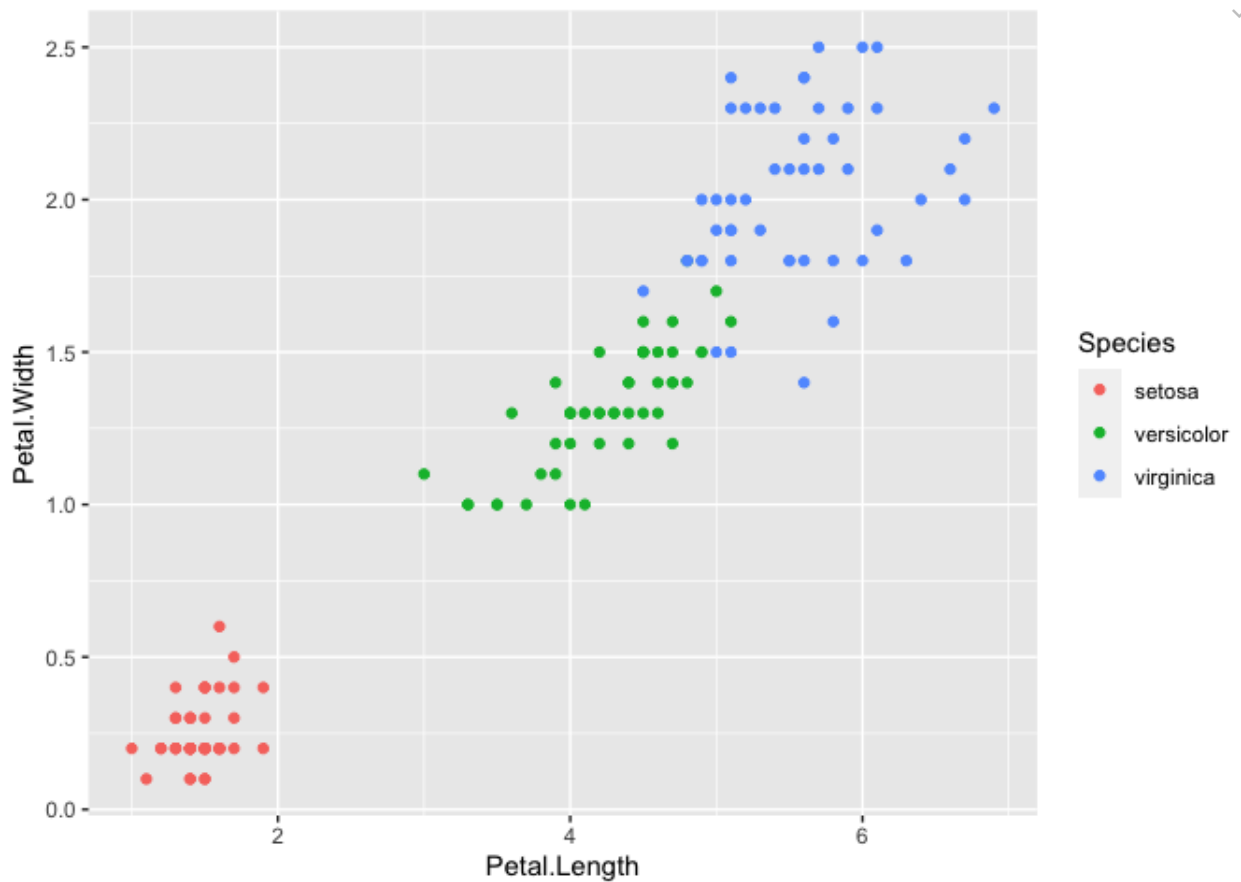
Start by plotting `Petal.Length` on the x-axis and `Petal.Width` on the y-axis.

{{Sdet}}

Solution{{Esum}}

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

{{Edet}}



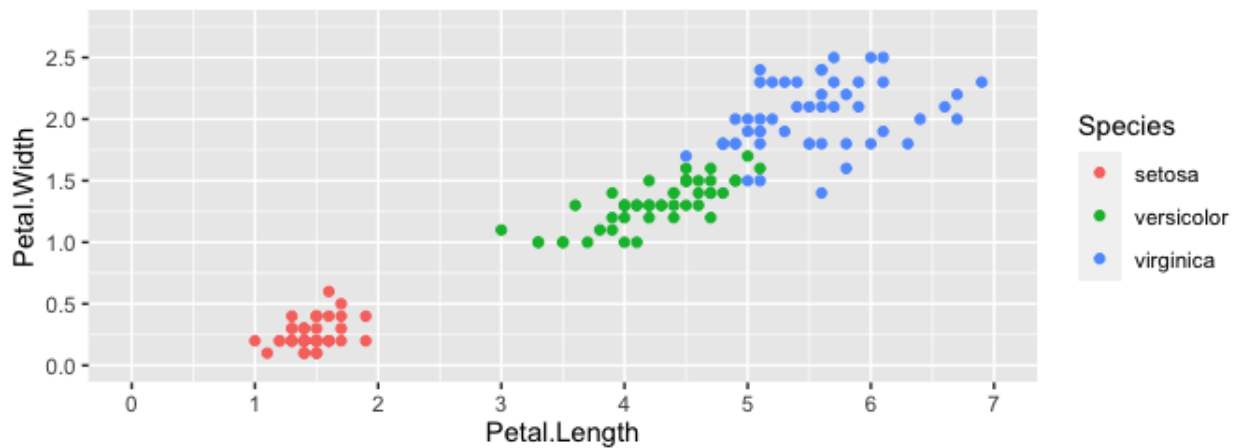
Fix the axes so that the dimensions on the x-axis and the y-axis are equal. Both axes should start at 0. Label the axis breaks every 0.5 units on the y-axis and every 1.0 units on the x-axis.

{{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}}

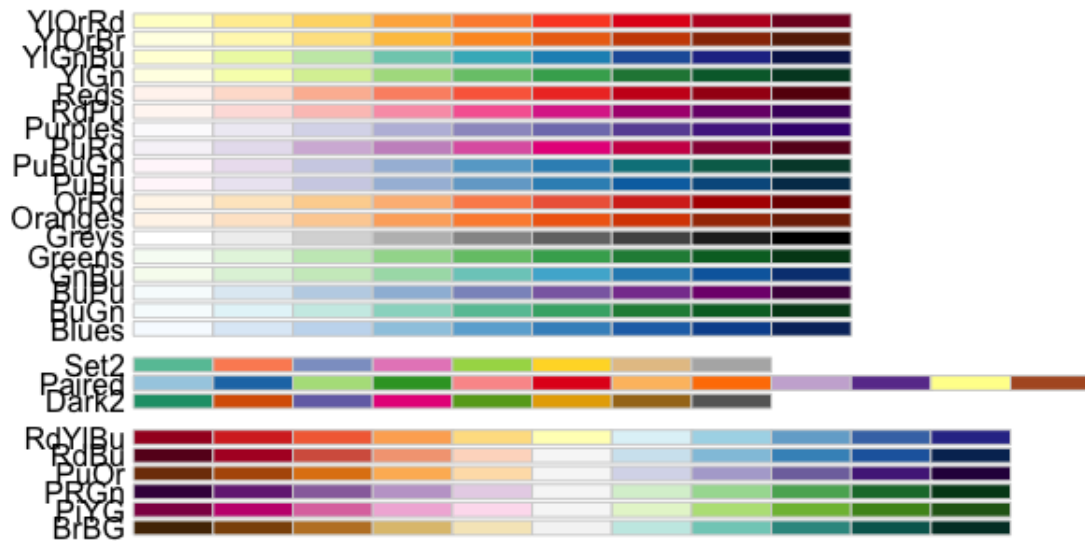


Change to color of the points by species to be color blind friendly, and change the legend title to "Iris Species". Label the x and y axis to eliminate the variable names and add unit information.

```
{{Sdet}}
```

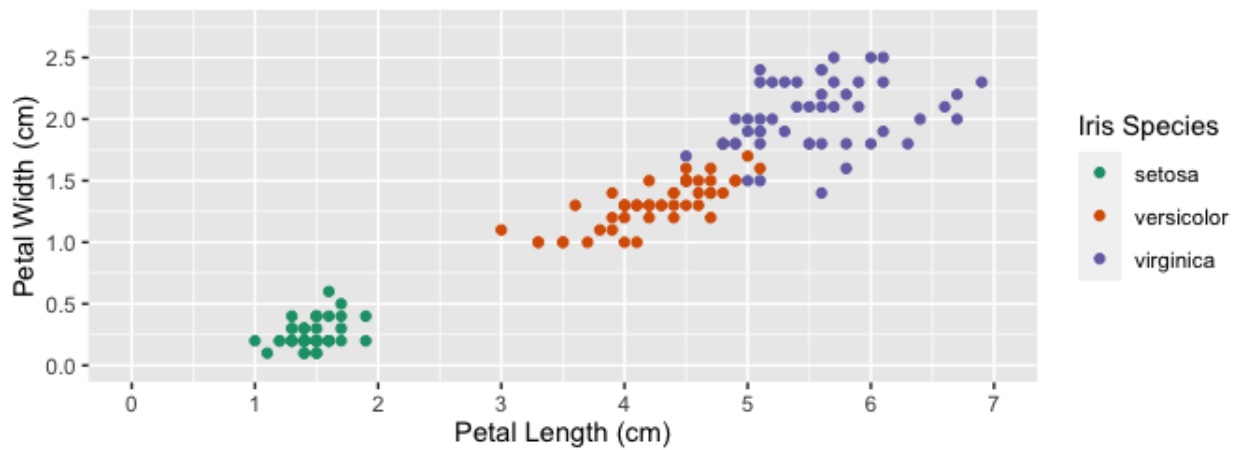
```
Solution{{Esum}}
```

```
#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)")
```

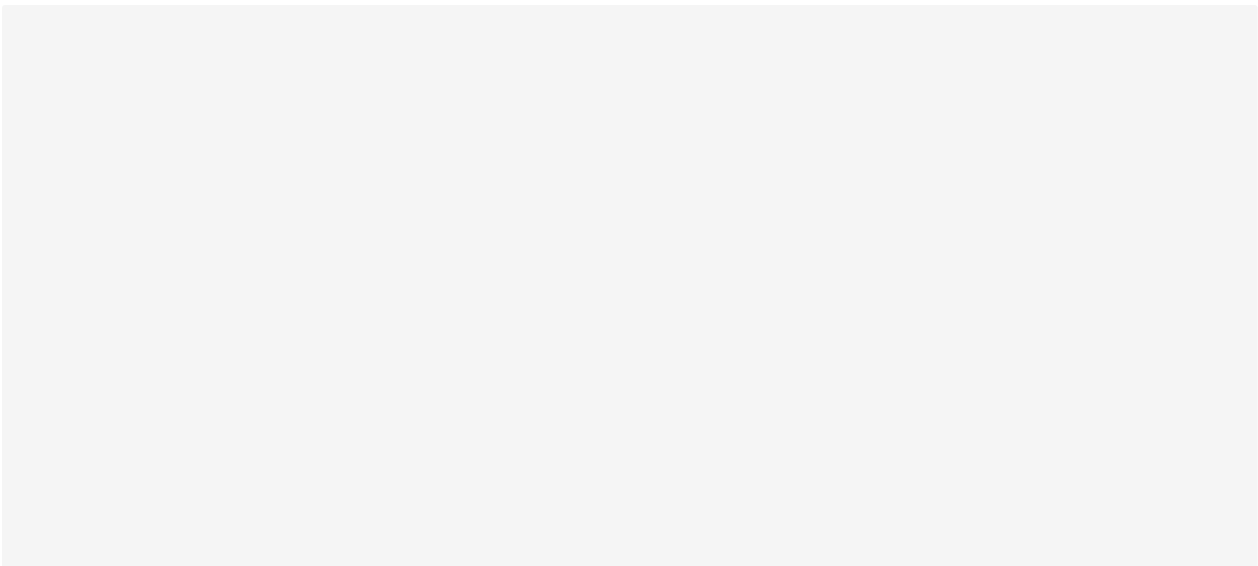
{{Edet}}



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.

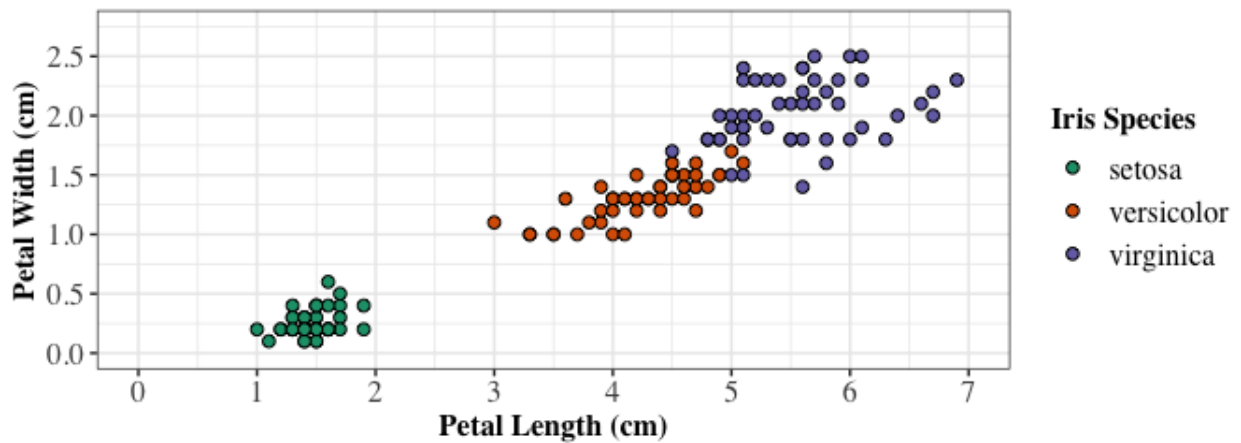
```
{{Sdet}}
```

```
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](#).



## Installing R & RStudio

Detailed Instructions for installing R and RStudio can be found [here \(https://btep.ccr.cancer.gov/docs/rtools/\)](https://btep.ccr.cancer.gov/docs/rtools/).

**DNAnexus**

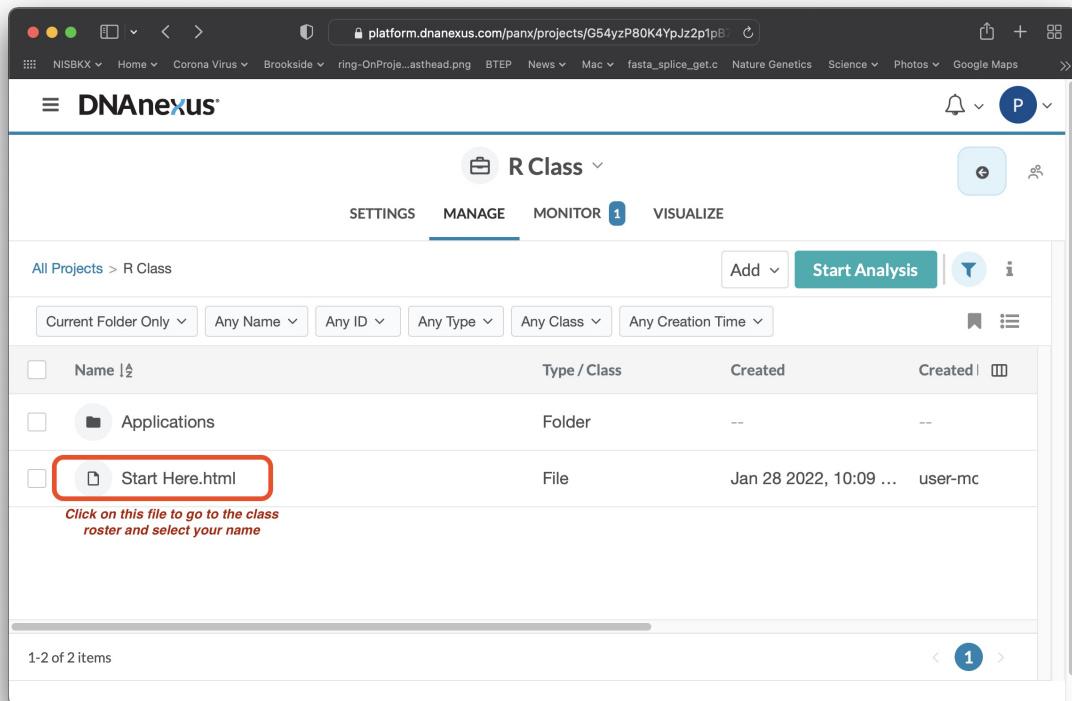
## Navigating DNAnexus

DNAnexus is a Cloud-based platform for NextGen Sequence analysis for which CCR has a "site-license". For this class we are using the platform to provide a uniform, stable, preinstalled interface for R training. This interface makes use of the Web version of R-studio. In addition to the R-studio interface this process also integrates the course-notes for the class in one window.

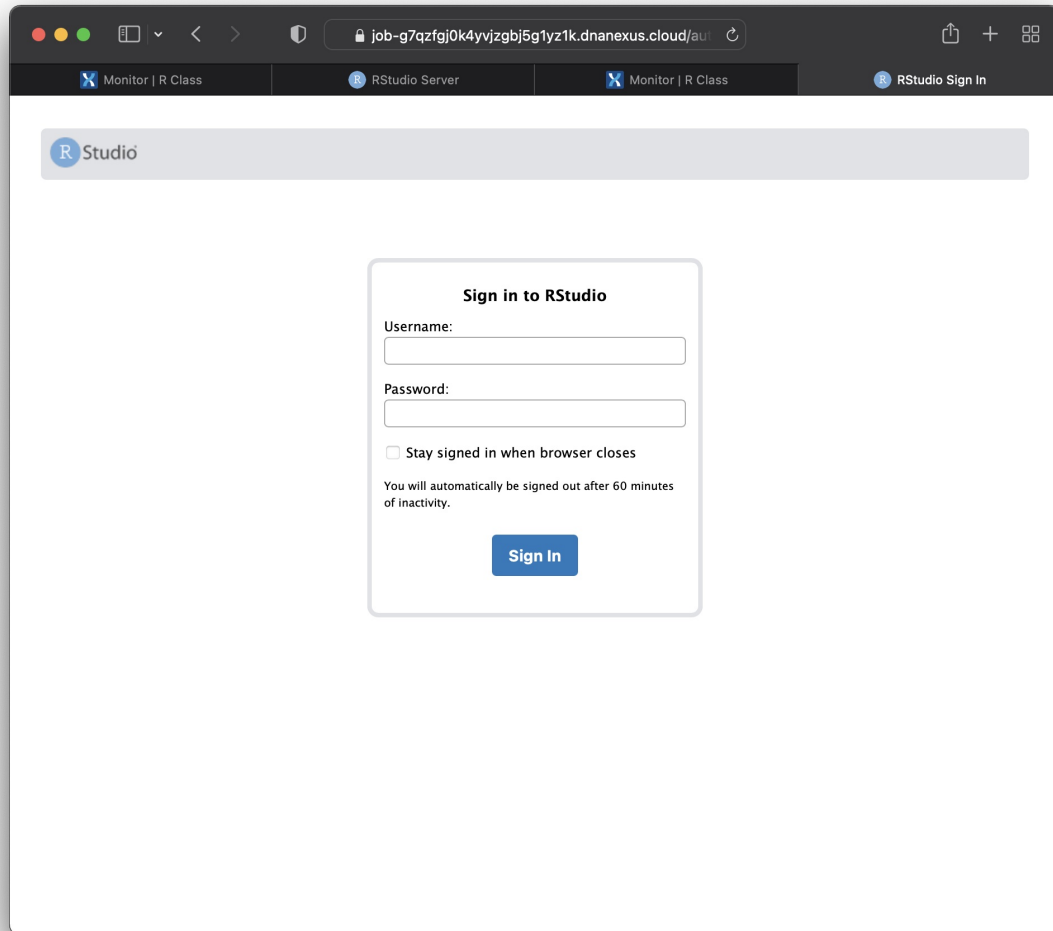
The following instructions should be followed when using this resource during formal class time. For using this resource outside class times see the document entitled "[DNAnexus Basics](#)".

## Instruction for using DNAnexus for the Intro to R class

1. **Getting a DNAnexus account** - every student should go to the main DNAnexus web page (<https://dnanexus.com/>) and apply for a "free account". The BTEP staff will associate each account with the NCI/CCR paid account prior to the first class.
2. **Logging into DNAnexus account** - Prior to the class each student should log into their account, and navigate to the R Class project (R\_Class\_Jan\_2023).
3. **Starting R** - Starting 30 mins before each class there will be a file labelled "Start\_Here.html" at the top level of the project. Select this file by clicking on it, and then find your name on the list (arranged alphabetically by first name) and click on it. **Note: if there is more than one Start\_Here.html file, you will need to select the file that has a range of letters in which your first name would be included.** For example, if there were two files, Start\_Here\_A\_J.html and Start\_Here\_K\_Z.html, and my name was "Alex", I would select Start\_Here\_A\_J.html. The names of these files will vary based on the total number of students in the class.



Once you select your name from the correct file, a window with the RStudio login page will open.



Log in using the username "rstudio" and the password "rstudio". At this point you will be presented with the RStudio main interface (shown below).

4. **Splitting the window** - If you wish to integrate the class notes into the same window as the R-Studio interface, click on the file "**Hsplit.html**" or "**Vsplit.html**" (found in the lower right hand segment) and select the "**View in Web Browser**" option from the pop-up menu. This will add the class notes to the top portion of the browser window. There is a horizontal or vertical bar separating the class notes window from the RStudio interface, and this bar can be dragged up and down or right to left, depending on which file you selected (Hsplit.html vs Vsplit.html), to change the size of the window dedicated to each function.

The screenshot shows a web browser window displaying the 'Bioinformatics Training and Education Program' page. The page has a dark header with the program name and contact information, and a red navigation bar for the 'R Introductory Series 2022'. Below the navigation bar is a 'Course Overview' section. An RStudio window is overlaid on the bottom half of the browser. The RStudio window shows the console with R version information and help text, and a file explorer on the right showing a list of files and folders. A red double-headed arrow points to the horizontal drag bar between the browser and RStudio windows, with the text 'Drag bar up or down to resize the windows' above it.

Drag bar up or down to resize the windows

Course Overview

Table of contents

File Edit Code View Plots Session Build Debug Profile Tools Help

Environment History Connections Tutorial

Files Plots Packages Help Viewer

| Name                              | Size    | Modified         |
|-----------------------------------|---------|------------------|
| diffexp_results_edger_airways.rds | 34.5 MB | Jan 28, 2022, 9: |
| diffexp_results_edger_airways.txt | 2.1 MB  | Jan 28, 2022, 9: |
| fitlowabund_scaledcounts_airw...  | 34.5 MB | Jan 28, 2022, 9: |
| fitlowabund_scaledcounts_airw...  | 23.4 MB | Jan 28, 2022, 9: |
| HISTORY.fitzgepe                  | 905 B   | Jan 28, 2022, 9: |
| Rlearning                         |         |                  |
| site                              |         |                  |
| Split_Window.html                 | 260 B   | Jan 21, 2022, 6: |

# DNAnexus Outside of Class

## Setting up the R-Studio environment outside of Class hours

These instructions should be followed if you are setting up the R-Studio Web environment outside the normal class hours. For instructions about using the resource during class hours follow the instructions found in the document "[Navigating DNAnexus](#)".

## Log into DNAnexus

Each student should log into their account, and navigate to the R Class project (R\_Class\_Jan\_2023). For these instructions to make sense, you should use the "New Version" of DNAnexus. If you are instead using the "Classic Version", your screen will look like this:

The screenshot shows the DNAnexus interface for the project 'R\_Class\_Jan\_2023'. At the top, there is a navigation bar with 'PROJECTS', 'TOOLS', and 'HELP'. Below this is a notification banner: 'We are introducing a new look for this page! This current design will be phased out.' with a 'Start Using New Version' button. The main header shows the project name 'R\_Class\_Jan\_2023' and tabs for 'SETTINGS', 'MANAGE', 'MONITOR', and 'VISUALIZE'. Below the header are several action buttons: '+ Add Data', 'New Folder', 'New Workflow', and 'Start Analysis'. There are also search filters for 'SEARCH SCOPE', 'NAME', 'ID', and 'MODIFIED'. The main content area shows a file browser on the left with a tree view: 'R\_Class\_Jan\_2023' (expanded) contains 'Applications' and 'classroom\_docs'. To the right is a table listing files and folders:

| Name                                   | Type   | Size      | Created |
|----------------------------------------|--------|-----------|---------|
| Applications                           | Folder |           |         |
| classroom_docs                         | Folder |           |         |
| 1. NIH Warning -Government Managed ... | File   | 10.11 KiB | AL      |
| 2.NIH Warning.html                     | File   | 9.91 KiB  | AL      |
| Practicing R - Outside of Class App    | Applet | 1.14 GiB  | Ja      |

You need to select "Start Using New Version".

Your screen should now look like the following:

DNAnexus PROJECTS TOOLS HELP

R\_Class\_Jan\_2023 SETTINGS MANAGE MONITOR VISUALIZE

All Projects > R\_Class\_Jan\_2023

Current Folder Only Any Name Any ID Any Type Any Class

| <input type="checkbox"/> | Name   ↕                                                             | Type / Class | Created              | Created By     | ⌵ |
|--------------------------|----------------------------------------------------------------------|--------------|----------------------|----------------|---|
| <input type="checkbox"/> | Applications                                                         | Folder       | --                   | --             |   |
| <input type="checkbox"/> | classroom_docs                                                       | Folder       | --                   | --             |   |
| <input type="checkbox"/> | 1. NIH Warning -Government...                                        | File         | Aug 09 2022, 8:41 AM | user-pcf       |   |
| <input type="checkbox"/> | 2.NIH Warning.html                                                   | File         | Aug 09 2022, 8:47 AM | user-pcf       |   |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> Practicing R - Outside of Cla... | Applet       | Jan 25 2023, 3:14 PM | user-mcintoshc |   |

Back to Classic Version

Add Stop

If you are in the right version, you will see this!

Select the "Practicing R - Outside of Class" applet.

No Item Select an item

## To Start the Application

To start the application, select "Practicing R - Outside of Class App".

DNAnexus PROJECTS TOOLS HELP

R\_Class\_Jan\_2023 SETTINGS MANAGE MONITOR VISUALIZE

All Projects > R\_Class\_Jan\_2023

Current Folder Only Any Name Any ID Any Type Any Class

| <input type="checkbox"/> | Name   ↕                                                             | Type / Class | Created              | Created By     | ⌵ |
|--------------------------|----------------------------------------------------------------------|--------------|----------------------|----------------|---|
| <input type="checkbox"/> | Applications                                                         | Folder       | --                   | --             |   |
| <input type="checkbox"/> | classroom_docs                                                       | Folder       | --                   | --             |   |
| <input type="checkbox"/> | 1. NIH Warning -Government...                                        | File         | Aug 09 2022, 8:41 AM | user-pcf       |   |
| <input type="checkbox"/> | 2.NIH Warning.html                                                   | File         | Aug 09 2022, 8:47 AM | user-pcf       |   |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> Practicing R - Outside of Cla... | Applet       | Jan 25 2023, 3:14 PM | user-mcintoshc |   |

Back to Classic Version

Add Stop

If you are in the right version, you will see this!

Select the "Practicing R - Outside of Class" applet.

No Item Select an item

Then follow these steps:

- Step 1: Under Analysis Settings, enter your name in the Execution Name field.
- Step 2: Change Priority from "Normal" to "High".



Step 1: Enter your name in "Execution Name"

Step 2: Change "Priority" from "Normal" to "High"

home\_rstudio\_upload \*.tar.gz

RStudio Session App v3.0.1

ANALYSIS SETTINGS ANALYSIS INPUTS APPLET SETTINGS

Execution Name  
Alex Emmons

Execute in Project  
R\_Class\_Jan\_2023 (project-GP4g32j0yftzV1JgG1q0YBqp0)

Execution Output Folder  
/

Priority <sup>?</sup>  
High

Spending Limit <sup>?</sup>  
No Limit

ADVANCED

Allow SSH Access  OFF  
Allow SSH access to enable developer debugging features. To use these options, you need to first set up SSH access via dx toolkit. [More about SSH Access.](#)

Start Analysis

- Step 3: Select the green button in the upper right labeled "Start Analysis".

Step 3: Select "Start Analysis".

home\_rstudio\_upload \*.tar.gz

RStudio Session App v3.0.1

ANALYSIS SETTINGS ANALYSIS INPUTS APPLET SETTINGS

Execution Name  
Alex Emmons

Execute in Project  
R\_Class\_Jan\_2023 (project-GP4g32j0yftzV1JgG1q0YBqp0)

Execution Output Folder  
/

Priority <sup>?</sup>  
High

Spending Limit <sup>?</sup>  
No Limit

ADVANCED

Allow SSH Access  OFF  
Allow SSH access to enable developer debugging features. To use these options, you need to first set up SSH access via dx toolkit. [More about SSH Access.](#)

Start Analysis

- Step 4: Navigate to the "Monitor" tab to check the status of the job.

DNAnexus PROJECTS TOOLS HELP

R\_Class\_Jan\_2023 SETTINGS MANAGE **MONITOR** VISUALIZE

Alex Emmons WAITING

Launch as new Job View Log View all Inputs/Outputs View Info Terminate

EXECUTION ID job-GP8kq7Q0yFzZ1P93XqQzVp1v PRIORITY high LAUNCHED ON 01/25/2023 4:22 pm RUNNING FOR -- EXECUTABLE RStudio Session ...

|            |            |            |            |
|------------|------------|------------|------------|
| 04:22:38PM | 04:23:02PM | 04:23:18PM | 04:23:34PM |
|------------|------------|------------|------------|

Alex Emmons

Show Inputs/Outputs

## Open the R-Studio Web interface

Step 5: Wait approximately 5 minutes and then select the link under "Worker URL". The link will look similar to this: "<https://job-g7v6z280k4yp6qg4pbkz1gvb.dnanexus.cloud>".

DNAnexus PROJECTS TOOLS HELP

R\_Class\_Jan\_2023 SETTINGS MANAGE **MONITOR** VISUALIZE

Terminate All Analyses

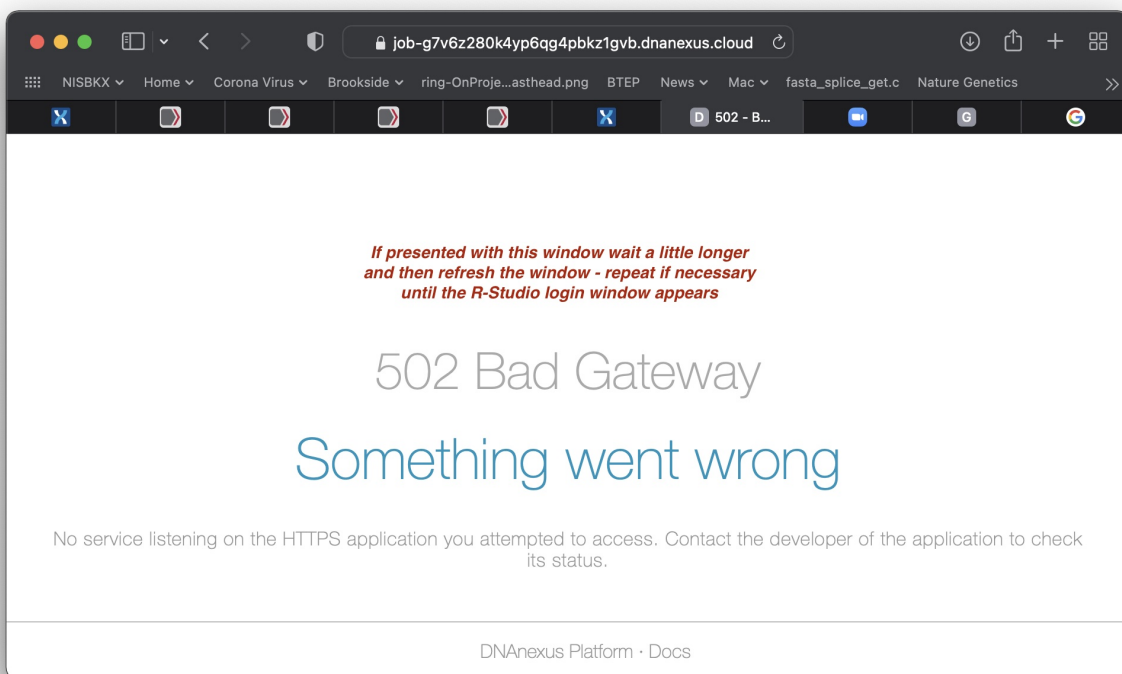
Status will switch from "Waiting" to "Running"

Once the duration has reached 5 mins, select the "Worker URL"

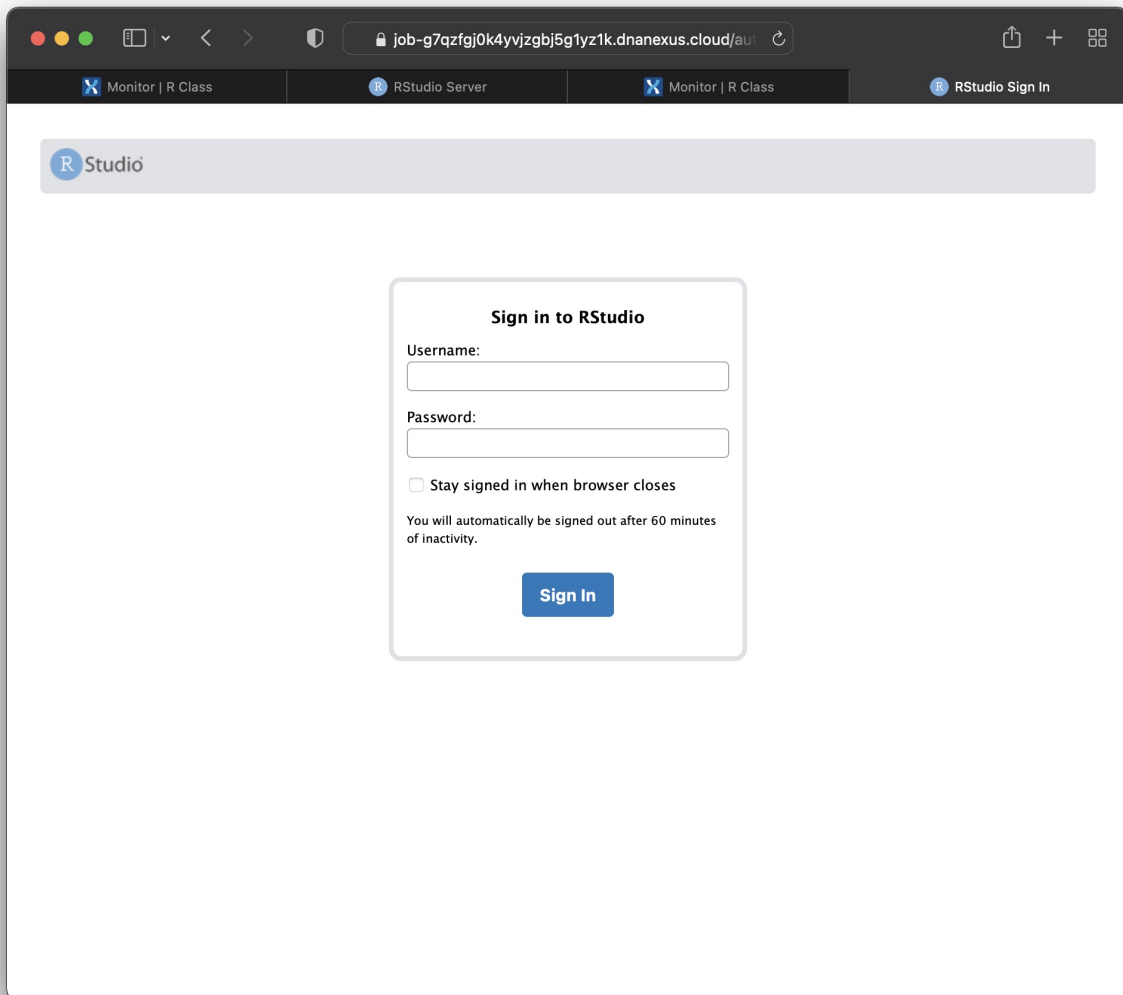
SEARCH SCOPE Root executions only STATE Any 307 recent jobs NAME Any ID Any CREATED Any LAUNCHED BY Any

| Status  | Name        | Executable                 | Launched by | Started running    | Duration | Price  | Priority | Worker URL                                                                                                            |
|---------|-------------|----------------------------|-------------|--------------------|----------|--------|----------|-----------------------------------------------------------------------------------------------------------------------|
| Running | Alex Emmons | RStudio Session App v3.0.1 | Alex Emmons | 01/25/2023 4:24 pm | 3m       | - cost | high     | <a href="https://job-GP8kq7Q0yFzZ1P93XqQzVp1v.dnanexus.cloud">https://job-GP8kq7Q0yFzZ1P93XqQzVp1v.dnanexus.cloud</a> |

The reason for waiting ~5 mins is to give the system time to get everything in place. If you click too soon you will see an error message.



Don't PANIC, just wait a little longer and refresh the screen, until you are finally presented with the RStudio login screen.



## Proceed with running content from the R Class

From the login screen, login with the username/passwd "rstudio/rstudio", and proceed from there.

If you encounter any problems send email to BTEP at [ncibtep@nih.gov](mailto:ncibtep@nih.gov) (<mailto:ncibtep@nih.gov>)

**Getting help**

## Need help?

Optional help sessions are hosted on Mondays and Wednesdays immediately following each lesson (2:15 - 3:00 pm).

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