Introductory R for Novices 2026

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

Table of Contents

Welcome

Getting Started with R

Practice Exercises

Additional Resources

Introductory R for Novices

Course Description

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

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

This course includes 3-parts:

Part 1: Getting Started with R

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

Part 2: Introduction to Data Wrangling

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

Part 3: Introduction to Data Visualization

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

Course Materials

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

R Installation Instructions

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

This is not required if you have a Biowulf account.

Lesson Recordings

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

Getting Started with R

Getting Started with R

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

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

Lessons

1. March 10, 2026 - Introduction to R and RStudio
2. March 12, 2026 - Basics of R Programming: R Objects and Data Types
3. March 17, 2026 - Basics of R Programming: Vectors
4. March 19, 2026 - Introduction to R Data Structures: Data Import
5. March 24, 2026 - R Data Structures: Data Frames

Required Course Materials

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

Lesson 1: Introduction to R and RStudio

Learning Objectives

By the end of this lesson, you will be able to:

1. Explain the difference between R (the programming language) and RStudio (the integrated development environment).
2. Navigate and work effectively within RStudio, including how to:
3. Create and use an R Project
4. Create and run an R script
5. Identify and manage your working directory
6. Run basic R functions
7. Access built-in help documentation
8. Describe how R Projects and scripting practices support reproducible data analysis.

By the end of this section, you should feel comfortable navigating the RStudio interface and executing basic R commands within a project-based workflow.

> **Important**
>
> In this lesson, we are not learning statistics yet. Instead, we are learning how to use the tools that will allow us to do reproducible data analysis. Think of this as learning how to use the lab bench before running the experiment.

What is R?

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

Why R?

R is a particularly great resource for statistical analyses, plotting, and report generating. The fact that it is widely used means that users do not need to reinvent the wheel. There is a package available for most types of analyses, and if users need help, it is only a Google search away. CRAN currently hosts 20,000+ packages, and the number continues to grow. There are also many field-specific ecosystems, including those useful in the -omics (genomics, transcriptomics, metabolomics, etc.).

Bioconductor, a curated repository focused on bioinformatics and computational biology, is tightly coupled to specific R versions and is updated twice per year. Recent releases include 2,000+ software packages along with experiment data packages, annotation packages, workflows, and online books.

Where do we get R packages?

To take full advantage of R, you need to install R packages. R packages are loadable extensions that contain code, data, documentation, and tests in a standardized, easy to share format that can easily be installed by R users. The primary repository for R packages is the Comprehensive R Archive Network (CRAN). CRAN *(https://cran.r-project.org/#:~:text=CRAN%20is%20a%20network%20of,you%20to%20minimize%20network%20load.)* is a global network of servers that store identical versions of R code, packages, documentation, etc (cran.r-project.org). To install a CRAN package, use `install.packages("packageName")`. Github is another common source used to store R packages; though, these packages do not necessarily meet CRAN standards so approach with caution. To install a Github packages use `devtools::install_github()` or `remotes::install_github("username/package")`.

Many genomics and other packages useful to biologists / molecular biologists can be found on Bioconductor *(https://www.bioconductor.org/)*. Bioconductor and Bioconductor packages use BiocManager for installation; see here *(https://www.bioconductor.org/install/)*.

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

r-universe *(https://r-universe.dev/search)* is a modern R package ecosystem and discovery platform built by the rOpenSci *(https://ropensci.org/)* team. Publish, explore, and evaluate R packages (CRAN and other sources).

Ways to run R

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

line), or interactively through an environment. This course will demonstrate the utility of the RStudio integrated development environment (IDE).

What is RStudio?

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

Installing R and RStudio

Mac or Windows (local installation): Follow instructions here: https://posit.co/download/rstudio-desktop/ *(https://posit.co/download/rstudio-desktop/)*.

NIH-managed machines: Follow your institute's IT installation policies or submit a service request if administrative privileges are required.

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

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

What is Positron?

Positron *(https://posit.co/products/ide/positron/)* is a new open-source IDE from Posit. It is built on the same core technology as Visual Studio Code (VS Code), essentially providing a VS Code–based environment tailored and optimized for data science with R and Python support.

While RStudio remains widely used (especially in HPC environments), Positron reflects Posit's next-generation direction for development tools.

R Vs. RStudio

R	RStudio
Programming language	Integrated development environment (IDE)
Performs calculations	Helps you write and manage code
Can run without RStudio	Requires R to function

A Conceptual Model for How You Will Work

When working in RStudio:

- You will write instructions in a script and run them.
- R executes them in memory.
- Objects will exist temporarily in your R session unless saved.
- Your project folder will stores any permanent files.

We will see this model play out as we work.

Getting Started with R and R Studio

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

Connect to RStudio on NIH HPC Open OnDemand

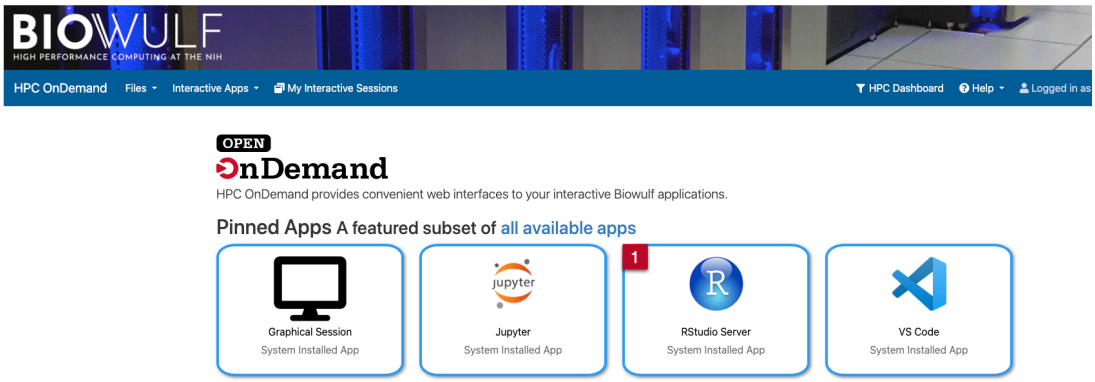
NIH HPC Open OnDemand *(https://hpc.nih.gov/ondemand/index.html)* provides an online dashboard for users to easily access command line interactive sessions, graphical linux desktop environments, and interactive applications including RStudio, MATLAB, IGV, iDEP, VS Code, and Juptyer Notebook. To use NIH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

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

This will take you to the HPC Open OnDemand dashboard.

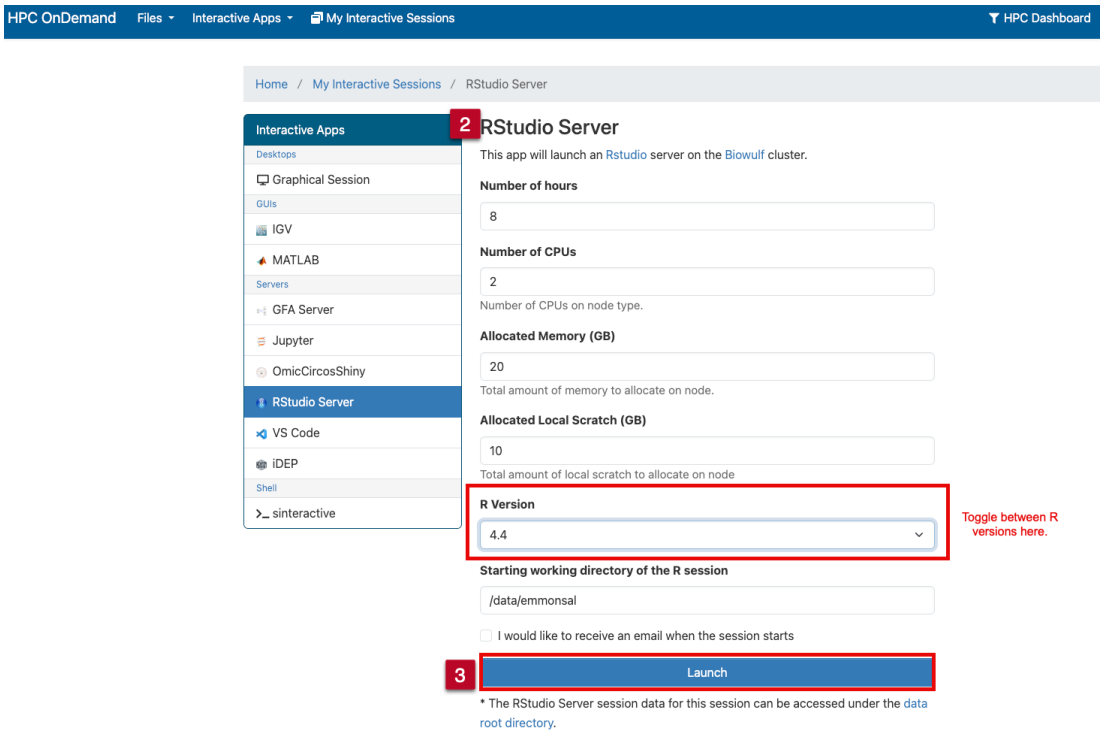
From there you will need to:

1. Select RStudio Server.



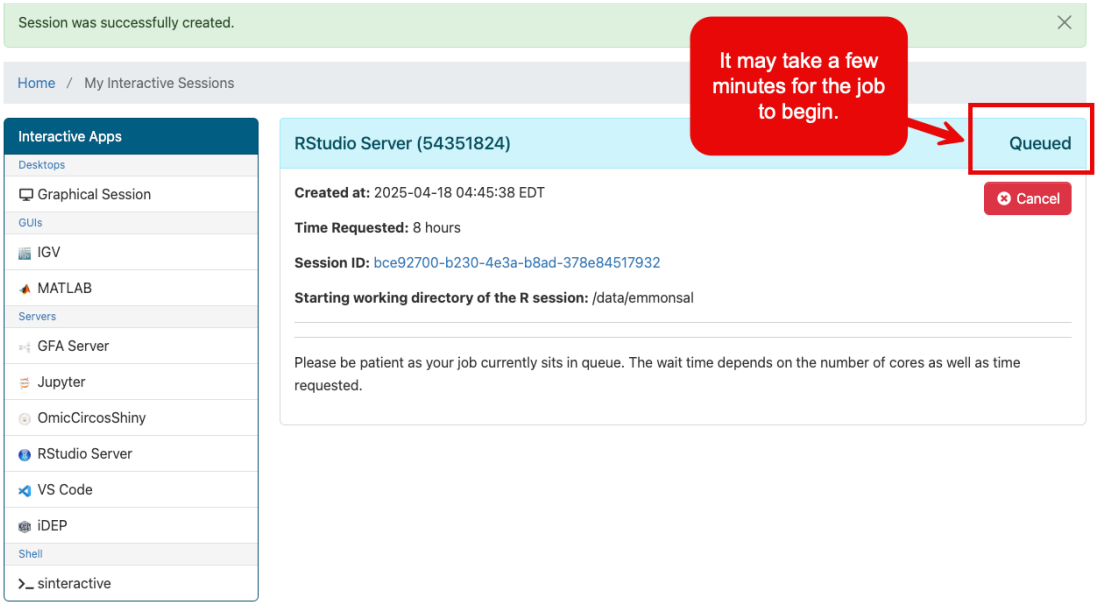
Step 1: Select RStudio Server from the selection of pinned applications.

1. Select parameters for your RStudio session including the version of R you want to use.
2. Click "Launch" to start the session.



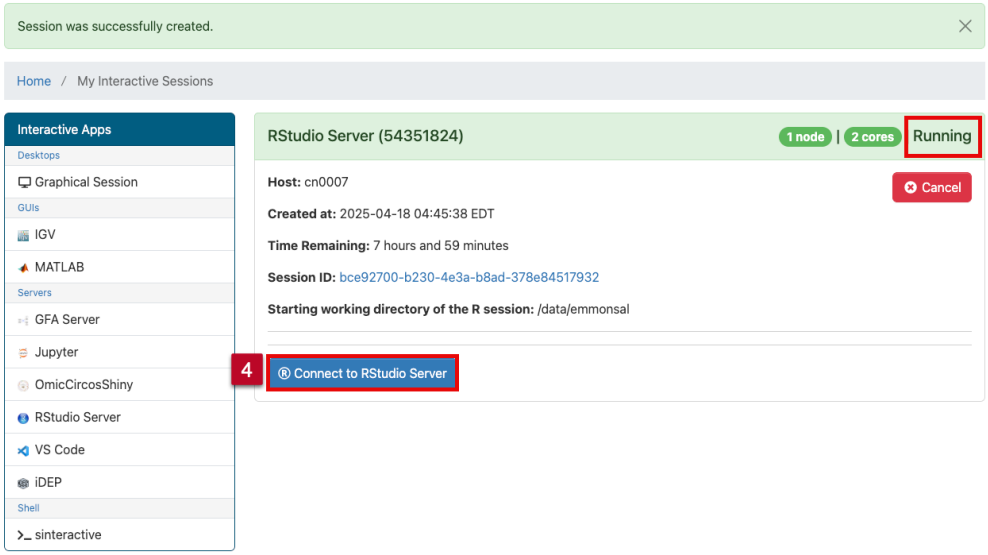
Step 2: Alter any job parameters as you see fit and launch the session.

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



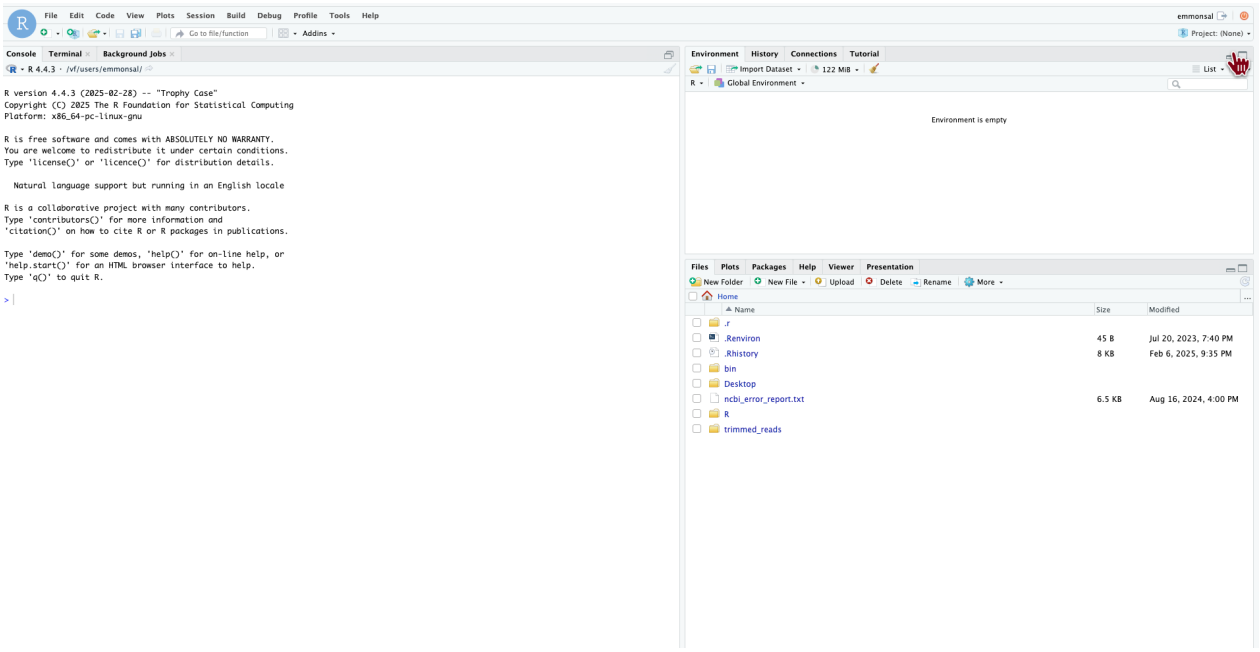
Session is queued.

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



Step 4: Connect to RStudio Server.

Congratulations! You are now connected.



RStudio Server on Biowulf

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

Creating an R project

If you intend to use R for upcoming analysis projects, you will want to create R projects. R projects automatically set your working directory to the directory specified for a given project. R projects are beneficial because they "keep all the files associated with a given project (input data, R scripts, analytical results, and figures) together in one directory" *(https://r4ds.hadley.nz/workflow-scripts.html#rstudio-projects)*.

> **Info**
>
> Think of an R Project as a self-contained laboratory notebook. If you send someone your project folder, they should be able to reproduce your analysis without editing file paths.

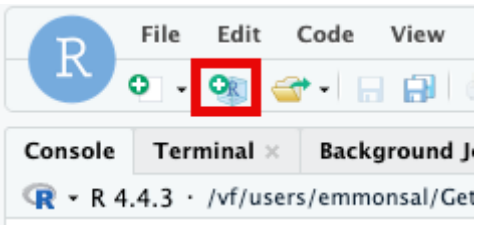
Creating an R project *(https://docs.posit.co/ide/user/ide/guide/code/projects.html)* for each project you are working on facilitates organization and scientific reproducibility.

An RStudio project allows you to more easily:

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

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

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



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

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

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

Why renv?

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

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

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

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

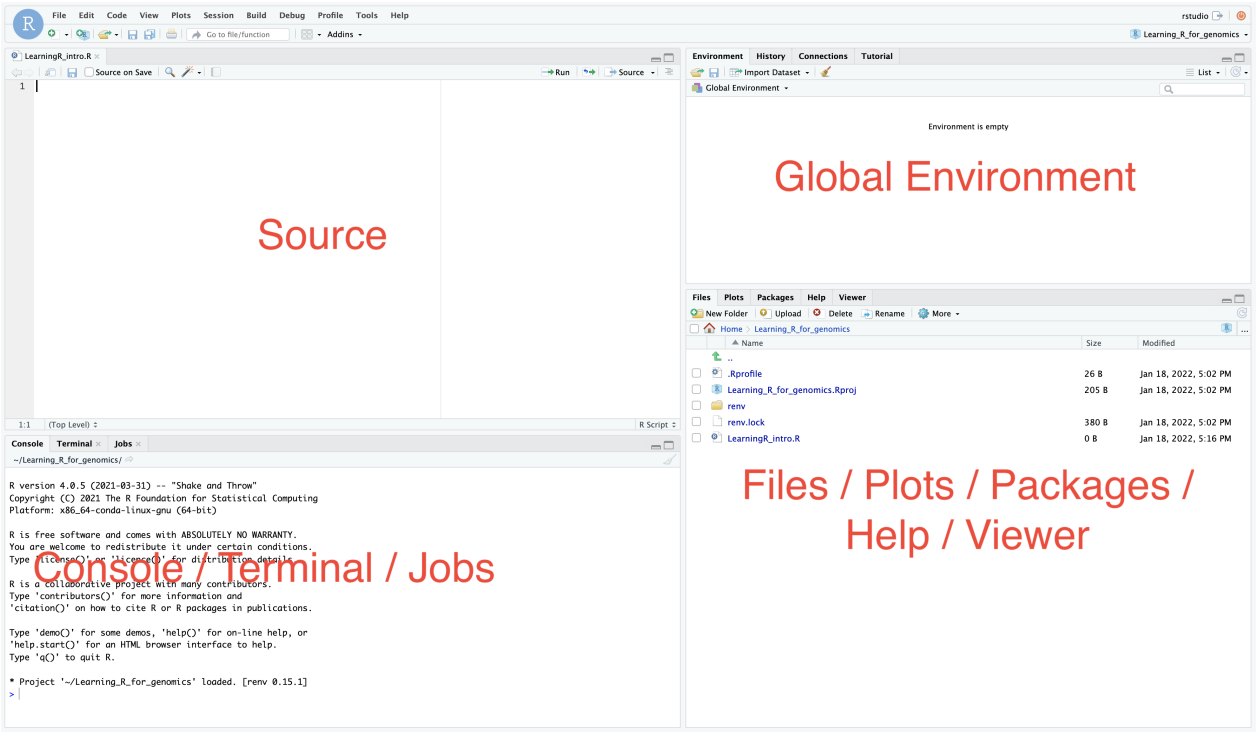
> **Note**
>
> There is even more that can be done to make projects reproducible beyond R Projects and `renv`. For example, you can use version control (git), R packages, and containerization (e.g., Singularity, Docker).

Creating an R script

As we learn more about R and start learning our first commands, we will keep a record of our commands using an R script. Remember, good annotation is key to reproducible data analysis. An R script can also be generated to run on its own without user interaction, from R console using `source()` and from linux command line using `Rscript`.

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

Introduction to the RStudio layout



Let's look a bit into our RStudio layout. (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.

Files/Plots/Packages/Help/Viewer: This multi-purpose pane will show you the contents of directories on your computer. You can also use the "Files" tab to navigate and set the working directory. The "Plots" tab will show the output of any plots generated. In "Packages" you will see what packages are actively loaded, or you can attach installed packages. "Help" will display help files for R functions and packages. "Viewer" will allow you to view local web content (e.g. HTML outputs).

---datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/00-introduction.html)*

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 (.Rmd) or Quarto (.qmd) files. Quarto *(https://quarto.org/)* is now the recommended next-generation publishing system from Posit and supports R, Python, Julia, and Observable JavaScript in a single framework.

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

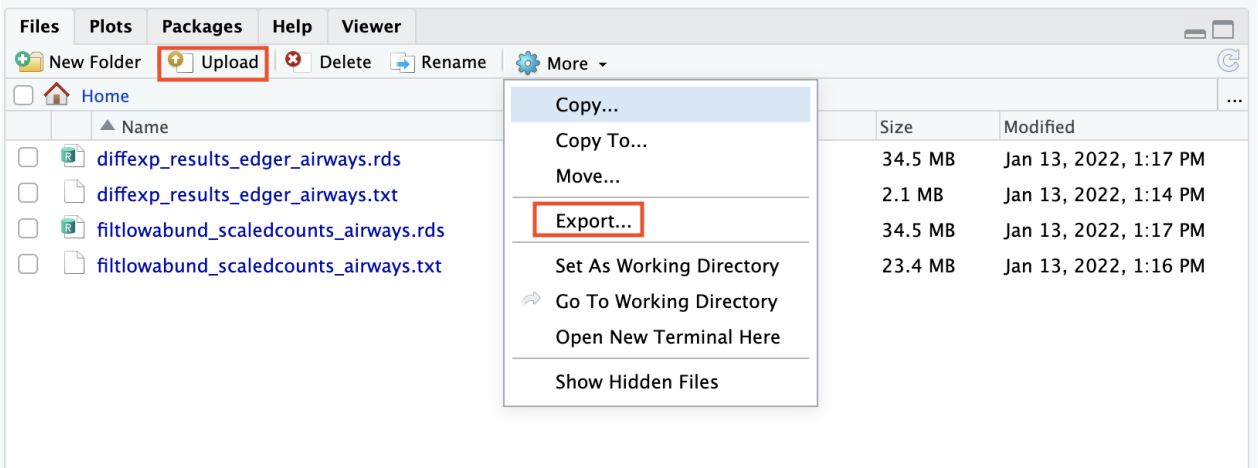
For today, we will primarily use the Source pane, the Console, and occasionally the Files and Help tabs. The other features will become more useful as our analyses become more complex.

When to use `Source` vs `Console`?

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

Uploading and exporting files from RStudio Server

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



Data Management

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

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

Saving your R environment (.Rdata)

When exiting RStudio, you will be prompted to save your R workspace or .RData. The .RData file saves the objects generated in your R environment. You can also save the .RData at any time using the floppy disk icon just below the Environment tab. You may also save your R workspace from the console using `save.image()`. RData files are often not visible in a directory. You can see them using `ls -a` from the terminal. RData files within a working directory associated with a given project will launch automatically under the default option **Restore .RData into workspace at startup**. You may also load .Rdata by using `load()`.

> **Note**
>
> 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". In modern reproducible workflows, it is strongly recommended to disable automatic saving and restoring of .RData files *(https://r4ds.hadley.nz/workflow-scripts.html#what-is-the-source-of-truth)* and instead rely on scripts as the single source of truth.

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

What is a function?

Now we are ready to work with some of our first R commands. In R, commands are generally called functions.

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

An R function has three key properties:

- Functions have a name (e.g. dir, getwd); note that functions are case sensitive!
- Following the name, functions have a pair of ()
- Inside the parentheses, a function may take 0 or more arguments --- datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/00-introduction.html#using-functions-in-r-without-needing-to-master-them)*.

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

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

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

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

To run this function, we have a number of options. First, you can use the `Run` button above. This will run highlighted or selected code. You may also use the source button to run your entire script. My preferred method is to use keyboard shortcuts. Move your cursor to the code of interest and use `command + return` for macs or `control + enter` for PCs. If a command is taking a long time to run and you need to cancel it, use `control + c` from the command line or `escape` in RStudio. Once you run the command, you will see the command print to the console in blue followed by the output.

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

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

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

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

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

```
setwd
```

> In modern workflows, if you are using R Projects correctly, you should rarely need to use `setwd()`.

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.

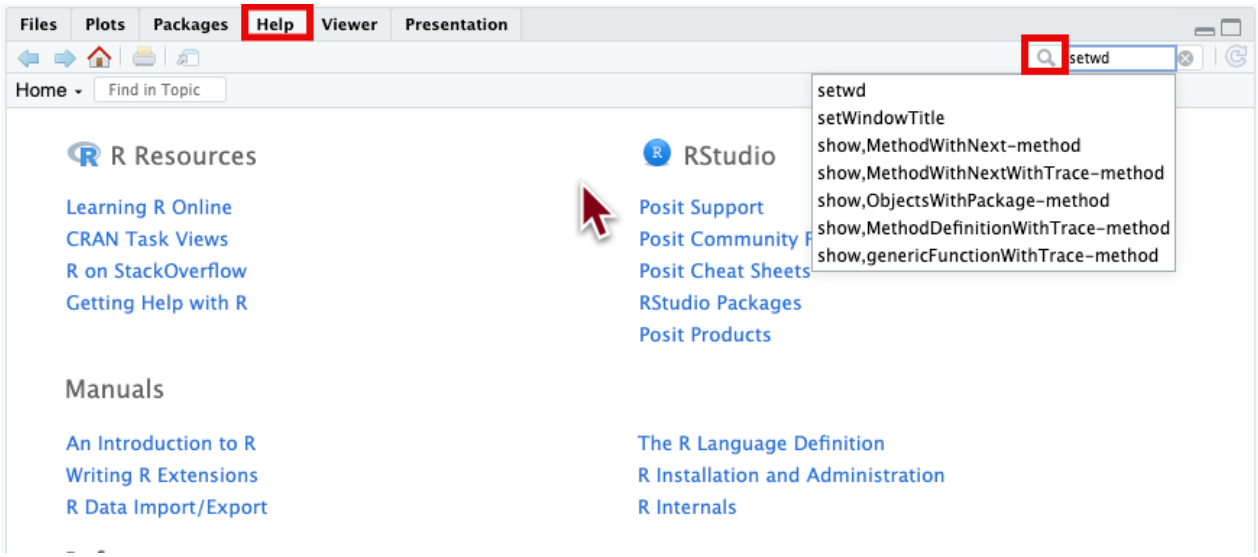
Common Beginner Mistakes

- Forgetting parentheses: writing getwd instead of getwd()
- Forgetting quotes around file paths
- Confusing setwd() with getwd()
- Running code in Console but forgetting to save it in a script

Getting help

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

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



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

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

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

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

```
args(setwd)
```

```
function (dir)
NULL
```

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

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

Addtional Sources for Help

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

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

To provide details about your R session, use

```
sessionInfo()
```

```
R version 4.5.1 (2025-06-13)
Platform: aarch64-apple-darwin20
Running under: macOS Tahoe 26.3

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

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

time zone: Europe/Berlin
tzcode source: internal

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

loaded via a namespace (and not attached):
 [1] htmlwidgets_1.6.4 compiler_4.5.1    fastmap_1.2.0     cli_3.6.5
 [5] tools_4.5.1       htmltools_0.5.9   otel_0.2.0        rstudioap
 [9] yaml_2.3.12       rmarkdown_2.30    knitr_1.51        jsonlite_
[13] xfun_0.56         digest_0.6.39     rlang_1.1.7       evaluate_
```

Test your learning

1. Which of the following functions is used to print your working directory in R?
   a. pwd
   b. Setwd()
   c. getwd()
   d. wkdir()

Answer

C

1. Which of the following can be used to learn more regarding an R function?
   a. ?function()
   b. ??function()
   c. args(function)
   d. All of the above

Answer

D

Acknowledgments

Material from this lesson was either taken directly or adapted from the Intro to R and RStudio for Genomics lesson provided by datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html)*. Material was also inspired by content from Introduction to data analysis with R and Bioconductor *(https://carpentries-incubator.github.io/bioc-intro/)*, which is part of the Carpentries Incubator *(https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator)*.

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

Objectives

By the end of this lesson, learners will be able to:

- Create, assign, modify, and remove R objects using appropriate naming conventions and assignment operators.
- Distinguish between common R data types (e.g., double, integer, character, logical) and identify an object's type and class using functions such as typeof(), class(), and str().
- Explain the difference between an object's underlying storage type and its class, and describe how these influence object behavior.
- Perform basic mathematical operations in R using numeric objects and standard operators.
- Recognize that functions are objects in R and understand their role in performing operations on data.

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

HPC Open OnDemand

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here *(http://127.0.0.1:8000/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

R objects

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

Creating and deleting objects

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

> **Note**
>
> In RStudio, you can use the keyboard shortcut to insert the assignment operator `<-`:
>
> - **Windows/Linux:** `Alt + -`
> - **Mac:** `Option + -`

> **Using `<-` vs `=` for Assignment**
>
> The conventional assignment operator in R is <-, and it is strongly recommended for clarity and consistency. While = can also assign values, it is primarily used to specify **function arguments** (e.g., `round(x, digits = 2)`). Using `<-` for assignment helps avoid confusion and improves code readability.

Let's create a simple object and run our code. There are a few methods to run code (the run button, key shortcuts (Windows: `ctrl+Enter`, Mac: `Command+Return`), or type directly into the console).

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

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

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

```
[1] 1
```

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

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

```
a + 5
```

```
[1] 6
```

```
5 - a
```

```
[1] 4
```

```
a^2
```

```
[1] 1
```

```
a + a
```

```
[1] 2
```

Naming conventions and reproducibility

There are rules regarding the naming of objects.

1) Avoid spaces or special characters EXCEPT _ and .
2) No numbers or underscores at the beginning of an object name. Objects may begin with a letter or a period (.), but if they begin with a period, the second character cannot be a number.

For example:

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

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

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

In contrast:

```

```

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

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

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

Strings

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

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

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

How do I know what objects have been created?

To view a list of the objects you have created, use `ls()' or look at your Global Environment pane in RStudio.

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

Reassigning objects

To reassign an object, simply overwrite the object.

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

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

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

```
gene_name
```

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

Warning

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

Deleting objects

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

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

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

Other Considerations

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

  For example:

  ```
  clifford<-"a big red dog"
  ```

  Now, type "clif" into the console and hit tab.
- Quotes are used anytime you are entering character string values. Either single or double quotes can be used. Otherwise, R will think you are calling an object.

Object data types

Understanding **data types and classes** is essential in R because they determine how an object behaves and what functions can operate on it.

> Data types are familiar in many programming languages, but also in natural language where we refer to them as parts of speech (nouns, verbs, adjectives, etc.). Once you know whether a word is a noun, you can usually count it or make it plural. If it is an adjective, you may be able to modify it into an adverb. Similarly, in R, once you know an object's type or class, you can better predict how it will behave. — adapted from datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/01-r-basics.html#understanding-object-data-types-classes-and-modes)*

Base (atomic) Data Types

Every R object has an underlying **storage type**. Common base R data types include:

- **double** (numeric values with decimals — default numeric type)
- **integer**
- **character**
- **logical** (TRUE/FALSE)
- **raw**
- **complex**

> **Note**
>
> In modern R (R ≥ 4.0), numeric values like `1` or `0.47` are stored internally as type `"double"` by default unless explicitly declared as integers (e.g., `1L`).

You can inspect an object's underlying storage type using:

```
typeof(object_name)
```

Class (how an object behaves)

In addition to type, many R objects have a **class** attribute.

The class determines how **generic functions** (like `print()`, `summary()`, or `plot()`) behave when applied to that object.

For example:

- A data frame has class `"data.frame"`
- A categorical variable may have class `"factor"`
- Dates may have class `"Date"`

If an object has no special class assigned, its class is often similar to its underlying type (for example, a simple numeric vector).

You can inspect class using:

```
class(object_name)
```

Type vs. Class

- `typeof()` - shows how the object is stored in memory
- `class()` - shows how the object behaves

These are related but not identical concepts.

In practice, `typeof()`, `class()`, and `str()` are the most useful tools for understanding an object:

```
str(object_name)
```

When R changes an object from one type to another automatically, this is called **coercion**. Sometimes R will display a coercion warning if information could be lost.

Examples

Let's create some objects and inspect their type and class:

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

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

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

chr_position <- "1001701bp"
typeof(chr_position)
```

```
## [1] "character"
class(chr_position)
## [1] "character"

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

Notice:

- Character values are stored as `"character"`
- Numeric values like `0.47` are stored as `"double"`
- `iris` is a `"data.frame"` (a class built on top of other types)
- Logical values are `"logical"`

Checking and changing types

There are helper functions to test types directly:

- `is.numeric()`
- `is.character()`
- `is.logical()`

And functions to explicitly convert between types:

- `as.double()`
- `as.integer()`
- `as.factor()`
- `as.character()`

If an object has a class attribute, there is usually a related **constructor function** used to create objects of that class (e.g., `data.frame()`, `factor()`).

We will discuss data frames and factors in more detail in a later lesson.

Special null-able values

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

Mathematical operations

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

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

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

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

Let's see this in practice.

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

A function is an object.

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

Looking up Compiled Code

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

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

Consider

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

```
[1] 6
```

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

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

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

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

```
[1] 5.7
```

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

The pipe (|>, %>%)

Functions can be chained together using a pipe.

- `|>` is the native base R pipe, introduced in R 4.1.0 (2021).
- `%>%` is the pipe from the magrittr package (commonly used in the tidyverse).

For new learners, it is recommended to use the native pipe `|>` unless working within tidyverse-heavy workflows. The pipe improves readability by minimizing nested function calls.

For example,

```
ex<- -5.679

abs(round(ex)) # nested
```

```
[1] 6
```

```
ex |> round() |> abs() # Using the pipe
```

```
[1] 6
```

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

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

Pre-defined objects

Base R comes with a number of built-in functions, vectors, data frames, and other objects. You can view built-in primitive functions and core base R functions with `builtins()`. You can view built-in datasets using `data()`. To explore datasets included with base R use `help(package = "datasets")`.

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

Answer

B

Which of the following are valid names for R objects? Select all that apply.

```
.3f <- 7

.fff <- 7

$%^ <- 7

This? <- 7

this.one <- 7

this_one <- 7
```

Answer

```
.fff <- 7

this.one <- 7
```

```
this_one <- 7
```

Acknowledgments

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

Lesson 3: Basics of R Programming: Vectors

Objectives

By the end of this lesson, learners will be able to:

- Define a vector and describe its role as a fundamental data structure in R.
- Create vectors using the `c()` function and other common vector-generating functions (e.g., `seq()`, `rep()`).
- Inspect vector properties using functions such as `typeof()`, `length()`, `str()`, and `names()`.
- Perform arithmetic and logical operations on numeric vectors.
- Subset vectors using numeric indexing, logical indexing, and the `%in%` operator.
- Modify vector contents by adding, removing, or replacing elements.
- Handle missing values (`NA`) within vectors using appropriate functions.
- Save and reload R objects using `saveRDS()` and `readRDS()`.

HPC Open OnDemand

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/ *(https://hpcondemand.nih.gov/)*. Then follow the instructions outlined here *(http://127.0.0.1:8000/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

Vectors

> Vectors are probably the most commonly used object type in R. A vector is a collection of values that are all of the same type (numbers, characters, etc.). The columns that make up a data frame are vectors. One of the most common ways to create a vector is to use the `c()` function - the “concatenate” or “combine” function. Inside the function you may enter one or more values; for multiple values, separate each value with a comma. --- datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/01-r-basics.html)*.

Creating vectors

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

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

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

```
typeof(transcript_names)
```

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

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

```
length(transcript_names)
```

```
[1] 4
```

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

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

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

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

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

```
[1] TRUE
```

Vectors can also have a names attribute.

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

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

```
names(counts)
```

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

Vectors contain elements of the same type!

If you mix elements of different types when creating a vector, the vector will be of the most complex type. See the following example from *R for Data Science (https://r4ds.had.co.nz/vectors.html#coercion)*:

```
typeof(c(TRUE, 1L))
```

```
[1] "integer"
```

```
typeof(c(1L, 1.5))
```

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

```
typeof(c(1.5, "a"))
```

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

Creating, modifying, sub-setting exporting

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

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

Creating vectors with functions

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

Vector operations

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

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

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

```
transcript_counts^2 +100
```

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

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

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

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

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

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

Some things to note here:

1. With vectors of the same length, we can add, subtract, multiply, etc., but operations are performed on elements in the same position of each vector.
2. With vectors of different lengths, the shorter vector will be recycled *(https://www.geeksforgeeks.org/vector-recycling-in-r/)* until the operation is complete. If the larger vector is not a multiple of the shorter vector, R will generate a warning. This could create bus in your code, so take caution.

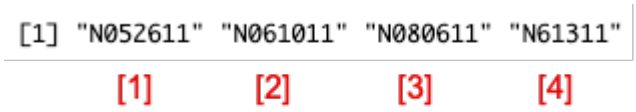
Vector sub-setting

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

```
cell_line
```

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

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



Index positions in `cell_line`.

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

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

```
cell_line[2]
```

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

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

For example:

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

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

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

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

Note

There are other ways to get the same result, for example, `tail(sample_id, 1)`.

This is the same as:

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

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

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

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

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

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

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

The combine function `c()` can also be used to add 1 or more elements to a vector. To modify a vector in memory, you must reassign it to the original object name.

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

Subtraction can be used to remove a value.

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

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

We can rename a value by

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

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

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

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

```
SCYL3
  467
```

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

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

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

Logical subsetting

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

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

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

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

```
[1] 679 467
```

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

```
transcript_counts > 260
```

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

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

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

```
[1] 679 467
```

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

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

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

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

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

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

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

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

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

```
[1] 2 6
```

Other ways to handle missing data

Other functions you may find useful when working with NAs 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 subs-etting based on the absence of NAs.

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

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

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

Using objects to store thresholds

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

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

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

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

```
[1] 679 467
```

Using the `%in%` operator.

There may be a time you want to know whether there are specific values in your vector. To do this, we can use the `%in%` operator (`?match()`). "x %in% y returns a logical vector the same length as x that is TRUE whenever a value in x is anywhere in y." *(https://r4ds.hadley.nz/logicals.html#in)* This can be used for sub-setting.

For example, let's take a look at transcript_names:

```
transcript_names
```

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

Now, let's test to see if "NNAME" and "ANAPC10P1" are in this vector.

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

```
[1] TRUE TRUE
```

How can we use this for subsetting?

When using `%in%` for subsetting, the logical vector it produces must be the same length as the object being subset. If you provide a logical vector that is not the same length as the object being subset, R will recycle it.

Correct way to subset:

```
find_transcripts<-c("NNAME","ANAPC10P1")
transcript_names[transcript_names %in% find_transcripts]
```

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

Incorrect way to subset:

```
transcript_names[find_transcripts %in% transcript_names]
```

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

Saving and loading objects

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

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

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

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

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

Further Reading and Examples

Check out this in-depth chapter on vectors from *R for Data Science (https://r4ds.had.co.nz/vectors.html#vectors)*.

Acknowledgments

Some material from this lesson was taken directly or adapted from the Intro to R and RStudio for Genomics lesson provided by datacarpentry.org *(https://datacarpentry.org/genomics-r-intro/01-introduction/index.html)*. Material was also inspired by content from Introduction to data analysis with R and Bioconductor *(https://carpentries-incubator.github.io/bioc-intro/)*, which is part of the Carpentries Incubator *(https://github.com/carpentries-incubator/proposals/#the-carpentries-incubator)*.

Lesson 4: Introduction to R Data Structures - Data Import

Learning Objectives

By the end of this lesson, learners will be able to:

1. Describe the structure and purpose of common R data structures, including vectors, factors, lists, matrices, and data frames (tibbles).
2. Import tabular datasets into R as data frames or tibbles using appropriate functions.
3. Export data from the R environment to common file formats (e.g., .csv, .txt).

HPC Open OnDemand

To get started with this lesson, you will first need to connect to RStudio on Biowulf. To connect to NIH HPC Open OnDemand, you must be on the NIH network. Use the following website to connect: https://hpcondemand.nih.gov/. Then follow the instructions outlined here *(http://127.0.0.1:8000/Getting_Started_with_R/Lesson1/#connect-to-rstudio-on-nih-hpc-open-ondemand)*.

Installing and Loading Packages

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

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

Where do we get R packages?

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

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

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

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

Key functions:

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

Load the libraries:

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

```
Warning: package 'readr' was built under R version 4.5.2
```

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

Data Structures

Data structures are objects that store data.

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

Other common data structures in R include:

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

- **matrices**

What are factors?

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

Reference level

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

Important functions

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

For example:

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

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

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

Note

When you create a factor from a character vector, R will order the levels alphabetically by default unless you explicitly specify the order. This behavior will affect modeling and plotting.

Warning

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

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

Lists

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

Important functions

- `list()` - create a list
- `names()` - create named elements (Also useful for vectors)
- `lapply()`, `sapply()` - for looping over elements of the list
- `purrr::map()` - tidyverse alternative for looping

Interested in for loops?

R for Data Science is a great resource for introductory R. Check out this chapter on iteration *(https://r4ds.had.co.nz/iteration.html)* to learn more about for loops and related concepts.

Example

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

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

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

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

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

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

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

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

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

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

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

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

$counts
[1] 200 300 400
```

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

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

Data Matrices

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

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

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

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

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

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

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

```
typeof(a1)
```

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

```
class(a1)
```

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

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

Data Frames: Working with Tabular Data

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

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

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

Best Practices for organizing genomic data

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

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

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

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

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

   R makes data analysis more reproducible and can eliminate some mistakes from human error. However, you should approach data analysis with a plan, and make sure you

understand what a function is doing before applying it to your data. Often using small subsets of data can be used as a form of data debugging to make sure the expected result materialized.

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

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

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

```
   Samples Counts Treatment
1        1   3023   control
2        2   2984   control
3        3   1453   control
4        4   4180   control
5        5   3761   control
6        6   3894   treated
7        7    700   treated
8        8   4037   treated
9        9   4053   treated
10      10   4989   treated
```

Example Data

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

Obtaining the data

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

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

Note

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

Importing Data

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

First, the RStudio IDE has a drop-down menu for data import. Simply go to `File > Import Dataset`, select one of the options, and follow the prompts. RStudio will generate the corresponding R code, which you can copy into your script if you want to reproduce the import later.

Second, data can be imported directly using R functions. This approach is preferred for reproducible workflows because the code used to load the data is recorded in your script.

Pay close attention to the import functions and their arguments. Using the import arguments correctly can save you from a headache later down the road. You will notice two types of import functions under `Import Dataset` → "From Text": **base R functions** and `readr` **functions**. We will use both approaches in this course.

Tip

While the Import Dataset menu is convenient, importing data using R code is generally preferred for reproducible analyses, because the steps used to load the data are recorded in your script.

Row names

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

What is a tibble?

When loading tabular data with `readr`, the default object created will be a `tibble`. Tibbles are like data frames with some small but apparent modifications. For example, they can have numbers for column names, and the column types are immediately apparent when viewing.

Additionally, when you call a tibble by running the object name, the entire data frame does not print to the screen, rather the first ten rows along with the columns that fit the screen are shown.

Reasons to use `readr` functions

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

Excel files (.xls, .xlsx)

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

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

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

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

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

```
[1] "/Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib
```

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

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

```
irisdata
```

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

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

The arguments likely to be most pertinent to you are:

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

Let's check out another example:

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

```
New names:
* `` -> `...2`
* `` -> `...3`
* `` -> `...4`
```

```
sum_air
```

```
# A tibble: 11 x 4
   `Uses Airway Data`                ...2          ...3
   <chr>                             <chr>         <chr>
 1 Some RNA-Seq summary information <NA>          <NA>
 2 <NA>                              <NA>          <NA>
 3 Sample Name                       Treatment     Number of Transcri
 4 GSM1275863                        Dexamethasone 10768
 5 GSM1275867                        Dexamethasone 10051
 6 GSM1275871                        Dexamethasone 11658
 7 GSM1275875                        Dexamethasone 10900
 8 GSM1275862                        None          11177
 9 GSM1275866                        None          11526
10 GSM1275870                        None          11425
11 GSM1275874                        None          11000
```

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

```
colnames(sum_air)
```

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

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

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

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

```
sum_air
```

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

Tab-delimited files (.tsv, .txt)

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

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

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

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

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

Using `read.delim()`:

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

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

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

`row.names` - used to specify row names.
`col.names` - use to specify column names if `header = FALSE`.
`skip` - Similar to `read_excel()`, used to skip a number of lines preceding the data we are interested in importing.
`check.names` - makes names syntactically valid and unique.

Using `read_delim()`:

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

```
Rows: 8 Columns: 9
-- Column specification --------------------------------------------
Delimiter: "\t"
chr (8): SampleName, cell, dex, albut, Run, Experiment, Sample, BioSa
dbl (1): avgLength

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

```
smeta2
```

```
# A tibble: 8 x 9
  SampleName cell    dex   albut Run       avgLength Experiment Samp
  <chr>      <chr>   <chr> <chr> <chr>         <dbl> <chr>      <chr
1 GSM1275862 N61311  untrt untrt SRR10395~       126 SRX384345  SRS5(
```

```
2 GSM1275863 N61311  trt   untrt SRR10395~       126 SRX384346  SRS5(
3 GSM1275866 N052611 untrt untrt SRR10395~       126 SRX384349  SRS5(
4 GSM1275867 N052611 trt   untrt SRR10395~        87 SRX384350  SRS5(
5 GSM1275870 N080611 untrt untrt SRR10395~       120 SRX384353  SRS5(
6 GSM1275871 N080611 trt   untrt SRR10395~       126 SRX384354  SRS5(
7 GSM1275874 N061011 untrt untrt SRR10395~       101 SRX384357  SRS5(
8 GSM1275875 N061011 trt   untrt SRR10395~        98 SRX384358  SRS5(
```

What if we want to retain row names?

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

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

```
                     X Accession.SRR1039508 Accession.SRR1039509
1   ENSG00000000003.TSPAN6                679                  448
2     ENSG00000000005.TNMD                  0                    0
3     ENSG00000000419.DPM1                467                  515
4    ENSG00000000457.SCYL3                260                  211
5 ENSG00000000460.C1orf112                 60                   55
6      ENSG00000000938.FGR                  0                    0
  Accession.SRR1039512 Accession.SRR1039513 Accession.SRR1039516
1                  873                  408                 1138
2                    0                    0                    0
3                  621                  365                  587
4                  263                  164                  245
5                   40                   35                   78
6                    2                    0                    1
  Accession.SRR1039517 Accession.SRR1039520 Accession.SRR1039521
1                 1047                  770                  572
2                    0                    0                    0
3                  799                  417                  508
4                  331                  233                  229
5                   63                   76                   60
6                    0                    0                    0
```

Because this is a count matrix, we want to save column 'X', which was automatically named, as row names rather than a column. Remember that `tidyverse` tools generally discourage the use of row names and instead treat identifiers as explicit columns. Because of this design choice, `readr` does not provide a built-in argument for assigning row names during import. When row names are required (which is common for genomic count matrices), base R functions such as `read.delim()` are often used.

Let's reload and overwrite the previous object:

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

```
                      Accession.SRR1039508 Accession.SRR1039509
ENSG00000000003.TSPAN6                  679                  448
ENSG00000000005.TNMD                      0                    0
ENSG00000000419.DPM1                    467                  515
ENSG00000000457.SCYL3                   260                  211
ENSG00000000460.C1orf112                 60                   55
ENSG00000000938.FGR                       0                    0
                      Accession.SRR1039512 Accession.SRR1039513
ENSG00000000003.TSPAN6                  873                  408
ENSG00000000005.TNMD                      0                    0
ENSG00000000419.DPM1                    621                  365
ENSG00000000457.SCYL3                   263                  164
ENSG00000000460.C1orf112                 40                   35
ENSG00000000938.FGR                       2                    0
                      Accession.SRR1039516 Accession.SRR1039517
ENSG00000000003.TSPAN6                 1138                 1047
ENSG00000000005.TNMD                      0                    0
ENSG00000000419.DPM1                    587                  799
ENSG00000000457.SCYL3                   245                  331
ENSG00000000460.C1orf112                 78                   63
ENSG00000000938.FGR                       1                    0
                      Accession.SRR1039520 Accession.SRR1039521
ENSG00000000003.TSPAN6                  770                  572
ENSG00000000005.TNMD                      0                    0
ENSG00000000419.DPM1                    417                  508
ENSG00000000457.SCYL3                   233                  229
ENSG00000000460.C1orf112                 76                   60
ENSG00000000938.FGR                       0                    0
```

Comma separated files (.csv)

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

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

Let's see this in action:

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

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

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

Let's check out `read_csv()`:

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

```
Rows: 35549 Columns: 9
-- Column specification --------------------------------------------
Delimiter: ","
chr (2): species_id, sex
dbl (7): record_id, month, day, year, plot_id, hindfoot_length, weigh

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

```
cexamp2
```

```
# A tibble: 35,549 x 9
   record_id month   day  year plot_id species_id sex   hindfoot_leng
       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <db
 1         1     7    16  1977       2 NL         M
 2         2     7    16  1977       3 NL         M
 3         3     7    16  1977       2 DM         F
 4         4     7    16  1977       7 DM         M
 5         5     7    16  1977       3 DM         M
 6         6     7    16  1977       1 PF         M
 7         7     7    16  1977       2 PE         F
 8         8     7    16  1977       1 DM         M
 9         9     7    16  1977       1 DM         F
```

```
10        10     7    16  1977        6 PF          F
# i 35,539 more rows
```

Other file types

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

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

Data Export.

To export data to a file, you can use base R functions such as:

- `write.table()`
- `write.csv()`
- `saveRDS()`

Or tidyverse functions such as:

- `readr::write_csv()`
- `readr::write_tsv()`

`saveRDS()` is particularly useful when saving R objects that you want to reload later without converting to a text format.

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

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

Acknowledgements

Some material from this lesson was either taken directly or adapted from *Intro to R and RStudio for Genomics (https://datacarpentry.github.io/genomics-r-intro/03-basics-factors-dataframes.html)* provided by datacarpentry.org. Other material from this lesson was inspired by R4DS *(https://r4ds.had.co.nz/data-import.html)* and Tidyverse Skills for Data Science *(https://jhudatascience.org/tidyversecourse/)*. The survey data *(https://figshare.com/articles/dataset/*

Portal_Project_Teaching_Database/1314459/10) loaded in this lesson was taken from datacarpentry.org *(https://datacarpentry.org/R-ecology-lesson/index.html)*.

Lesson 5: R Data Structures - Data Frames

Learning Objectives

This is the last lesson in Part 1 of *Introductory R for Novices: Getting Started with R*. This lesson focuses on working with data frames. Attendees will apply foundational R skills to examine, summarize, access, and subset data stored in data frames.

1. Review and apply data import techniques to load tabular data into R as data frames.
2. Inspect and summarize the structure and contents of a data frame using base R and `tidyverse` functions.
3. Access columns and elements of a data frame using appropriate accessors (e.g., `$`, `[ ]`, `[[]]`).
4. Subset data frames using index-based and logical operations in base R.

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

Load the libraries

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

```
library(tidyverse)
```

```
Warning: package 'ggplot2' was built under R version 4.5.2
```

```
Warning: package 'tibble' was built under R version 4.5.2
```

```
Warning: package 'tidyr' was built under R version 4.5.2
```

```
Warning: package 'readr' was built under R version 4.5.2
```

```
Warning: package 'purrr' was built under R version 4.5.2
```

```
Warning: package 'dplyr' was built under R version 4.5.2
```

```
-- Attaching core tidyverse packages ------------------------ tidyver
v dplyr     1.2.0     v readr     2.1.6
v forcats   1.0.1     v stringr   1.6.0
v ggplot2   4.0.2     v tibble    3.3.1
v lubridate 1.9.4     v tidyr     1.3.2
v purrr     1.2.1
-- Conflicts ------------------------------------------ tidyverse_con
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to forc
```

Examining and summarizing data frames

All of the objects we imported in the previous lesson were data frames. In modern R workflows, especially when using the tidyverse, data are often stored as `tibbles`, which are a modern re-imagining of data frames with improved printing and stricter behavior.

The object we will create here using `read.delim()` is a base R data frame, but most of the concepts we learn apply equally to both data frames and tibbles.

Let's use the R object smeta as an example.

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

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

```
5 SAMN02422682
6 SAMN02422673
```

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

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

```
str(smeta)
```

```
'data.frame':   8 obs. of  9 variables:
 $ SampleName: chr  "GSM1275862" "GSM1275863" "GSM1275866" "GSM127586
 $ cell      : chr  "N61311" "N61311" "N052611" "N052611" ...
 $ dex       : chr  "untrt" "trt" "untrt" "trt" ...
 $ albut     : chr  "untrt" "untrt" "untrt" "untrt" ...
 $ Run       : chr  "SRR1039508" "SRR1039509" "SRR1039512" "SRR10395
 $ avgLength : int  126 126 126 87 120 126 101 98
 $ Experiment: chr  "SRX384345" "SRX384346" "SRX384349" "SRX384350"
 $ Sample    : chr  "SRS508568" "SRS508567" "SRS508571" "SRS508572"
 $ BioSample : chr  "SAMN02422669" "SAMN02422675" "SAMN02422678" "SAM
```

```
#A tidyverse-friendly alternative
glimpse(smeta)
```

```
Rows: 8
Columns: 9
$ SampleName <chr> "GSM1275862", "GSM1275863", "GSM1275866", "GSM1275
$ cell       <chr> "N61311", "N61311", "N052611", "N052611", "N080611
$ dex        <chr> "untrt", "trt", "untrt", "trt", "untrt", "trt", "u
$ albut      <chr> "untrt", "untrt", "untrt", "untrt", "untrt", "untr
$ Run        <chr> "SRR1039508", "SRR1039509", "SRR1039512", "SRR1039
$ avgLength  <int> 126, 126, 126, 87, 120, 126, 101, 98
$ Experiment <chr> "SRX384345", "SRX384346", "SRX384349", "SRX384350"
$ Sample     <chr> "SRS508568", "SRS508567", "SRS508571", "SRS508572"
$ BioSample  <chr> "SAMN02422669", "SAMN02422675", "SAMN02422678", "S
```

`str()` shows us that we are looking at a data frame object with 8 rows by 9 columns. The column names are to the far left preceded by a `$`. The `$` symbol is a data frame accessor that allows us to extract a specific column by name (for example `smeta$Run`). We will explore this in more detail later in the lesson. We can also see the data types (e.g., character, integer,

logical, double) after the column name. This will help us understand how we can transform and visualize the data in these columns.

`glimpse()` displays the structure of the data frame in a more readable horizontal format and is commonly used when working with tidyverse tools.

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

```
summary(smeta)
```

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



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

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

Tip

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

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

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

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

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

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

```
#length returns the number of columns for data frames
#(because a data frame is internally a list of columns)
length(smeta)
```

```
[1] 9
```

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

```
[1] 8 9
```

Other useful functions for inspecting data frames

Size:
`nrow()` - number of rows
`ncol()` - number of columns

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

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

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

Data frame coercion and accessors

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

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

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

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

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

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

```
#Using $
```

```
smeta$SampleID
```

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

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

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

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

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

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

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

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

```
[1] TRUE
```

```
#let's convert it to a character vector
smeta$SampleID<-as.character(smeta$SampleID)
#confirm conversion
is.character(smeta$SampleID)
```

```
[1] TRUE
```

```
#should now be FALSE
is.numeric(smeta$SampleID)
```

```
[1] FALSE
```

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

Be careful with data coercion. What happens if we change a character (is.character(smeta$SampleID)) vector into a numeric?

```
head(as.numeric(smeta$Run))
```

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

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

A warning is thrown and any values that cannot be converted to numbers become NA.

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

Using `colnames()` to rename columns

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

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

```
```

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

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

```
[1] 10
```

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

Subsetting data frames with base R

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
     BioSample SampleID
2 SAMN02422675        2
```

Subsetting Tibbles

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

What happens when we provide a single argument?

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

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

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

smeta[2] #returns column two
```

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

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

Note

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

Using %in%

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

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

Using == is a bit tedious.

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

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

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

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

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

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

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

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

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

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

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

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

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

Tips to remember for subsetting

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

Info

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

Data Wrangling

Part 2 of this course will focus on Data Wrangling. Learn how to filter, modify, summarize, and reshape your data. Check the BTEP calendar for updates on upcoming classes / courses.

Acknowledgements

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

Practice Exercises

Part 1: Exercises

Exercise 1: Lesson2

Q1. What is the value of each object? Run the code and print the values.

```
mass <- 47.5            # mass?
age  <- 122             # age?
mass <- mass * 2.0      # mass?
age  <- age - 20        # age?
mass_index <- mass / age  # mass_index?
```

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

Q1: Solution

```
mass <- 47.5            # mass?
mass
## [1] 47.5
age  <- 122             # age?
age
## [1] 122
mass <- mass * 2.0      # mass?
mass
## [1] 95
age  <- age - 20        # age?
age
## [1] 102
mass_index <- mass / age  # mass_index?
mass_index
## [1] 0.9313725
```

Q2. Create the following objects; give each object an appropriate name.

a. Create an object that has the value of the number of bones in the adult human body.

b. We can create a vector of values using `c()`. For example to create a vector of fruits, we could use the following: `fruit <- c("apples", "bananas", "mango", "kiwi")`. Use this information to create an object containing the names of four different bones. (We will learn more about vectors in Lesson 3.)

Q2: Solution

```
# a.
bone_num<- 206
bone_num
```

```
## [1] 206

# b.
bone_names<- c("talus","calcaneus","tibia","fibula")
bone_names
## [1] "talus"     "calcaneus" "tibia"     "fibula"
```

Q3. What types of data are stored in the objects created in question 2.

Q3: Solution

```
typeof(bone_num)
## [1] "double"
typeof(bone_names)
## [1] "character"
```

Q4. Modify `bone_num` to contain the number of bones in an adult human hand.

Q4: Solution

```
bone_num <- 27
bone_num
## [1] 27
```

Q5. Here is an object storing multiple values:

```
num_vec <- c(1:100)
```

What is the mean of this vector? How about the median? What functions can you use to find this information?

Q5: Solution

```
mean(num_vec)
## [1] 50.5
median(num_vec)
## [1] 50.5
```

Q6. What does the function `paste()` do? How can you find out? Can you use it to collapse `bone_names` into a string of length 1? **Hint: Read the help documentation closely.**

Q6: Solution

```
# To find help, use the ?
?paste

# To collapse the vector to length 1, check the collapse argument
paste(bone_names, collapse=", ")
## [1] "talus, calcaneus, tibia, fibula"
length(bone_names)
## [1] 4
```

Exercise 2: Lesson 3

Q1. Let's use some functions.

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

Q1a: Solution

```
sum(1:10)
## [1] 55
```

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

Q1b: Solution

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

c. Combine the following vectors and compute the mean.

```
a <- c(45, 67, 34, 82)
b <- c(90, 45, 62, 56, 54)
```

Q1c: Solution

```
mean(c(a,b))
## [1] 59.44444
```

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

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

Q1d: Solution

```
#tells us whether the two vectors are the same
identical(c, d)
## [1] TRUE
```

Q2. Vectors include data of a single type, so what happens if we mix different types? Use `typeof()` to check the data type of the following objects.

```
num_char <- c(1, 2, 3, "a")
num_logical <- c(1, 2, 3, TRUE, FALSE)
char_logical <- c("a", "b", "c", TRUE)
tricky <- c(1, 2, 3, "4")
```

Q2: Solution

```
#These were coerced into a single data type
typeof(num_char)
## [1] "character"
num_char
## [1] "1" "2" "3" "a"
typeof(num_logical)
## [1] "double"
num_logical
## [1] 1 2 3 1 0
typeof(char_logical)
## [1] "character"
char_logical
## [1] "a"    "b"    "c"    "TRUE"
typeof(tricky)
## [1] "character"
tricky
## [1] "1" "2" "3" "4"
```

(Question taken from https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r.html *(https://carpentries-incubator.github.io/bioc-intro/23-starting-with-r.html)*)

Q3. `fruit` is a vector containing the common names of different types of fruit. Can you replace "kiwi" with "mango".

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

Q3: Solution

```
fruit[5] <- "mango"
fruit
## [1] "apples"  "bananas" "oranges" "grapes"  "mango"   "kumquat"
```

Q4. Given the following R code, return all values less than 678 in the vector "Total_subjects".

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

Q4: Solution

```
Total_subjects[Total_subjects < 678]
## [1]  23   4  12 654
```

Q5. This question uses the vectors created in Q2. Using indexing, create a new vector named `combined` that contains:

The 2nd and 3rd value of `num_char`.
The last value of `char_logical`.
The 1st value of `tricky`.

`combined` contains data of what type?

Q5: Solution

```
combined <- c(num_char[2:3], char_logical[length(char_logical)],
              tricky[1])
typeof(combined)
## [1] "character"
combined
## [1] "2"    "3"    "TRUE" "1"
```

Exercise 3: Lesson 4

Loading data

The data used in this practice exercise can be found here.

Q1. Import data from the sheet "iris_data_long" from the excel workbook (file_path = "./data/iris_data.xlsx"). Make sure the column names are unique and do not contain spaces. Save the imported data to an object called `iris_long`.

Q1: Solution

```
iris_long<-readxl::read_excel("../data/iris_data.xlsx",sheet="iris_data_long",.name
## New names:
## • `Iris ID` -> `Iris.ID`
## • `Measurement location` -> `Measurement.location`
iris_long
## # A tibble: 600 × 4
##    Iris.ID Species Measurement.location Measurement
##      <dbl> <chr>   <chr>                       <dbl>
##  1       1 setosa  Sepal.Length                  5.1
##  2       1 setosa  Sepal.Width                   3.5
##  3       1 setosa  Petal.Length                  1.4
##  4       1 setosa  Petal.Width                   0.2
##  5       2 setosa  Sepal.Length                  4.9
##  6       2 setosa  Sepal.Width                   3
##  7       2 setosa  Petal.Length                  1.4
##  8       2 setosa  Petal.Width                   0.2
##  9       3 setosa  Sepal.Length                  4.7
## 10       3 setosa  Sepal.Width                   3.2
## # ℹ 590 more rows
```

Q2. Import a tab delimited file (file_path= "./data/species_datacarpentry.txt"). Save the file to an object named `species`. `genus`,`species`, and `taxa` should be converted to factors upon import.

Q2: Solution

```
species<-readr::read_delim("../data/species_datacarpentry.txt",col_types="cfff")
species
## # A tibble: 54 × 4
##    species_id genus           species         taxa
##    <chr>      <fct>           <fct>           <fct>
##  1 AB         Amphispiza      bilineata       Bird
##  2 AH         Ammospermophilus harrisi        Rodent
##  3 AS         Ammodramus      savannarum      Bird
##  4 BA         Baiomys         taylori         Rodent
```

```
##  5 CB         Campylorhynchus  brunneicapillus Bird
##  6 CM         Calamospiza      melanocorys     Bird
##  7 CQ         Callipepla       squamata        Bird
##  8 CS         Crotalus         scutalatus      Reptile
##  9 CT         Cnemidophorus    tigris          Reptile
## 10 CU         Cnemidophorus    uniparens       Reptile
## # ℹ 44 more rows
```

Q3. Load in a comma separated file with row names present (file_path= "./data/countB.csv") and save to an object named `countB`.

Q3: Solution

```
countB<-read.csv("../data/countB.csv",row.names=1)
head(countB)
##        SampleA_1 SampleA_2 SampleA_3 SampleB_1 SampleB_2 SampleB_3
## Tspan6       703       567       867        71       970       242
## TNMD         490       482        18       342       935       469
## DPM1         921       797       622       661         8       500
## SCYL3        335       216       222       774       979       793
## FGR          574       574       515       584       941       344
## CFH          577       792       672       104       192       936
```

Challenge data load

Q4. Load in a **tab delimited file** (file_path= "./data/WebexSession_report.txt") using `read_delim()`. You will need to troubleshoot the error message and modify the function arguments as needed.

Q4: Solution

```
library(tidyverse)
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.4     ✔ readr     2.1.5
## ✔ forcats   1.0.0     ✔ stringr   1.5.1
## ✔ ggplot2   3.5.2     ✔ tibble    3.2.1
## ✔ lubridate 1.9.4     ✔ tidyr     1.3.1
## ✔ purrr     1.0.4
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all confl
read_delim("../data/WebexSession_report.txt",delim="\t",locale = locale(encoding =
## Rows: 10 Columns: 21
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: "\t"
## chr   (7): Name, Date, Invited, Registered, Duration, Network joined from:, ...
## dbl   (1): Participant
## lgl  (11): Audio Type, Email, Company, Title, Phone Number, Address 1, Addre...
## time  (2): Start time, End time
```

```
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message
## # A tibble: 10 × 21
##    Participant `Audio Type` Name     Email Date  Invited Registered `Start time`
##          <dbl> <lgl>        <chr>    <lgl> <chr> <chr>   <chr>      <time>
##  1           1 NA           Partici… NA    6/8/… No      N/A        13:00
##  2           2 NA           Partici… NA    6/9/… <NA>    <NA>       13:00
##  3           3 NA           Partici… NA    6/10… No      N/A        12:57
##  4           4 NA           Partici… NA    6/11… <NA>    <NA>       12:57
##  5           5 NA           Partici… NA    6/12… No      N/A        12:55
##  6           6 NA           Partici… NA    6/13… <NA>    <NA>       12:55
##  7           7 NA           Partici… NA    6/14… No      N/A        12:32
##  8           8 NA           Partici… NA    6/15… <NA>    <NA>       12:32
##  9           9 NA           Partici… NA    6/16… Yes     N/A        12:42
## 10          NA NA           <NA>     NA    <NA>  <NA>    <NA>         NA
## # ℹ 13 more variables: `End time` <time>, Duration <chr>, Company <lgl>,
## #   Title <lgl>, `Phone Number` <lgl>, `Address 1` <lgl>, `Address 2` <lgl>,
## #   City <lgl>, `State/Province` <lgl>, `Zip/Postal Code` <lgl>,
## #   `Country/region` <lgl>, `Network joined from:` <chr>,
## #   `Internal Participant:` <chr>
```

Exercise 4: Lesson 5

For this exercise we will use `filtlowabund_scaledcounts_airways.txt`, which includes normalized and non-normalized transcript count data from an RNAseq experiment. You can read more about the experiment here *(https://pubmed.ncbi.nlm.nih.gov/24926665/)*. To obtain this file, click here.

The following questions synthesize several of the skills you have learned thus far. It may not be immediately apparent how you would go about answering these questions. Remember, the R community is expansive, and there are a number of ways to get help including but not limited to google search. These questions have multiple solutions, but you should try to stick to the tools you have learned to use thus far.

Q1. Import `filtlowabund_scaledcounts_airways.txt` into R and save to an R object named `transcript_counts`. Try not to use the drop-down menu for loading the data.

Q1 Solution

```
transcript_counts <-read.delim("../data/filtlowabund_scaledcounts_airways.txt")
```

Q2. What are the dimensions of `transcript_counts`?

Q2 Solution

```
dim(transcript_counts)
## [1] 127408     18
```

Q3. What are the column names?

Q3 Solution

```
colnames(transcript_counts)
##  [1] "feature"       "sample"        "counts"        "SampleName"
##  [5] "cell"          "dex"           "albut"         "Run"
##  [9] "avgLength"     "Experiment"    "Sample"        "BioSample"
## [13] "transcript"    "ref_genome"    ".abundant"     "TMM"
## [17] "multiplier"    "counts_scaled"
```

Q4. Is there a difference in the number of transcripts with greater than 0 normalized counts (counts_scaled) per sample? What commands did you use to answer this question.

Q4 Solution

```
#using table
table(transcript_counts[transcript_counts$counts_scaled>0,]$sample)
##
##   508   509   512   513   516   517   520   521
## 15921 15919 15923 15918 15913 15920 15914 15910

#alternative solution
summary(factor(transcript_counts[transcript_counts$counts_scaled>0,]$sample))
##   508   509   512   513   516   517   520   521
## 15921 15919 15923 15918 15913 15920 15914 15910

# or using the tidyverse
library(dplyr)
transcript_counts %>% filter(counts_scaled>0) %>% count(sample)
##   sample     n
## 1    508 15921
## 2    509 15919
## 3    512 15923
## 4    513 15918
## 5    516 15913
## 6    517 15920
## 7    520 15914
## 8    521 15910
```

Q5. How many categories of transcripts are there? Think about what you know regarding factors. Why is this number much smaller than the results of question 4?

Q5 Solution

```
nlevels(factor(transcript_counts$transcript, exclude = NULL))
## [1] 14576
```

Q6. Subset transcript_counts to only include the following columns: sample, cell, dex, transcript, avgLength, counts_scaled. Save this new dataframe to a new object called `transc_df`.

Q6 Solution

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

Q7. Using your new data frame from question six (`transc_df`), rename the column "sample" to "Sample".

Q7 Solution

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

Q8. What is the mean and standard deviation of "avgLength" across the entire `transc_df` data frame? Hint: Read the help documentation for `mean()` and `sd()`.

Q8 Solution

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

Q9. Make a data frame with the column names "Mean" and "Standard_Dev" that holds the values from question 8. Hint: check out the function `data.frame()`.

Q9 Solution

```
data.frame(Mean=mean_avgLength, Standard_Dev=sd_avgLength)
##     Mean Standard_Dev
## 1 113.75     14.85561
```

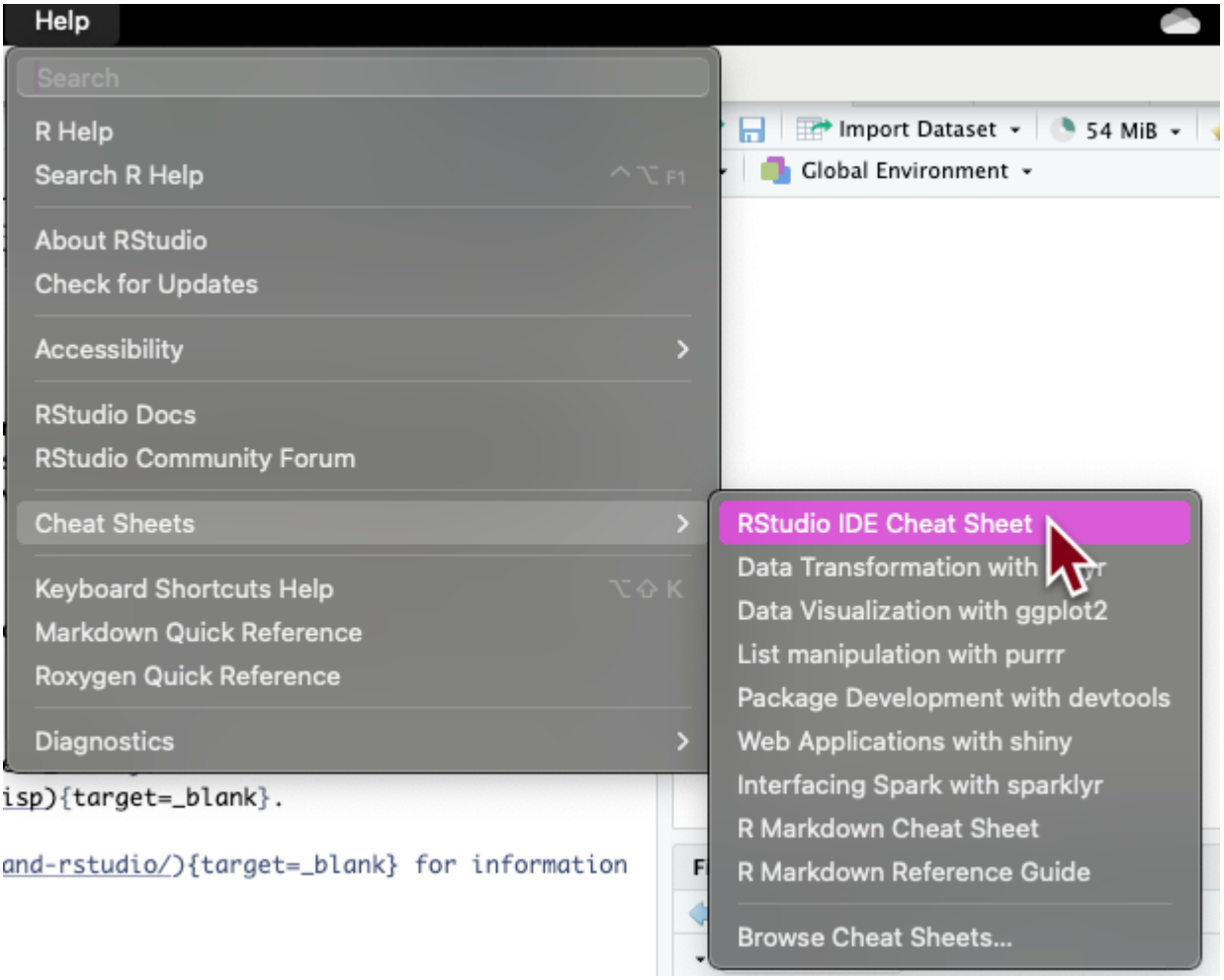
Additional Resources

Books and / or Book Chapters of Interest

1. R for Data Science *(https://r4ds.hadley.nz/)*
2. Hands-on Programming with R *(https://rstudio-education.github.io/hopr/)*
3. Statistical Inference via Data Science: A ModernDive into R and the Tidyverse *(https://moderndive.com/v2/preface.html#about-the-book)*
4. The R Graphics Cookbook *(https://r-graphics.org/index.html)*
5. ggplot2: Elegant Graphics for Data Analysis *(https://ggplot2-book.org/index.html)*
6. Advanced R *(https://adv-r.hadley.nz/)*
7. YaRrr! The Pirate's Guide to R *(https://bookdown.org/ndphillips/YaRrr/)*

R Cheat Sheets

Cheat sheets can be accessed directly using the Help tab within RStudio (Help > Cheat Sheets > Browse Cheat Sheets).



Other Resources

1. The R Graph Gallery *(https://www.r-graph-gallery.com/)*
2. From Data to Viz *(https://www.data-to-viz.com/)*
3. RMarkdown from RStudio *(https://rmarkdown.rstudio.com/lesson-1.html)*
4. Quarto for R *(https://quarto.org/docs/computations/r.html)*
5. Ten simple rules for teaching yourself R, Lawlor et al. 2022, *PLoS Comput Biol (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9436135/)*
6. Learn R *(https://www.learn-r.org/)*
7. Dplyr Learn R tutorial *(https://allisonhorst.shinyapps.io/dplyr-learnr/#section-welcome)*
8. A Beginner's Guide to Troubleshooting R Code *(https://bioinformatics.ccr.cancer.gov/docs/btep-coding-club/CC2023/Troubleshooting_Rprog/)*