Introductory R for Novices

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

Table of Contents

Welcome

Getting Started with R

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`.

This course will take place on T, Th, 2:00 - 3:00 PM.

Course Materials

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

R Installation Instructions

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

This is not required if you have a Biowulf account.

Lesson Recordings

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

Getting Started with R

Getting Started with R

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

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

Lessons

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

Required Course Materials

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

Lesson 1: Introduction to R and RStudio IDE

Learning Objectives

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

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

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

What is R?

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

Why R?

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

Where do we get R packages?

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

format that can easily be installed by R users. The primary repository for R packages is the Comprehensive R Archive Network (CRAN). CRAN *(https://cran.r-project.org/#:~:text=CRAN%20is%20a%20network%20of,you%20to%20minimize%20network%20load.)* is a global network of servers that store identical versions of R code, packages, documentation, etc (cran.r-project.org). To install a CRAN package, use `install.packages("packageName")`. Github is another common source used to store R packages; though, these packages do not necessarily meet CRAN standards so approach with caution. To install a Github packages use `library(devtools)` followed by `install_github()`. Many genomics and other packages useful to biologists / molecular biologists can be found on Bioconductor *(https://www.bioconductor.org/)*. Bioconductor and Bioconductor packages use BiocManager for installation; see here *(https://www.bioconductor.org/install/)*.

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

Ways to run R

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

What is RStudio?

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

What is Posit?

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

Installing R and RStudio

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

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

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

Getting Started with R and R Studio

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

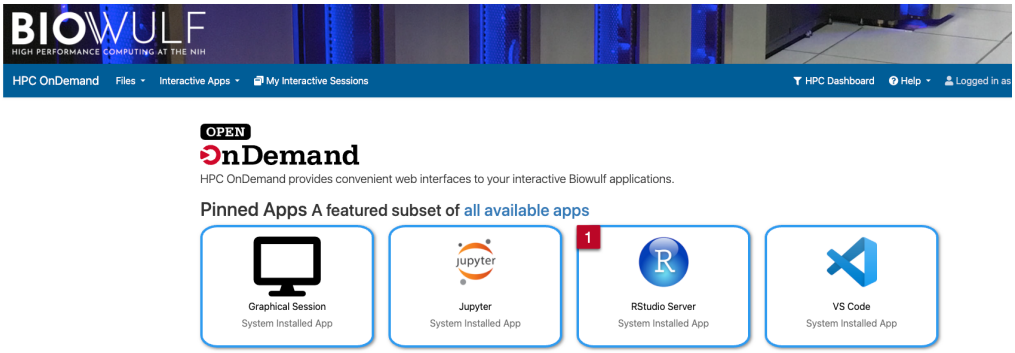
Connect to RStudio on NIH HPC Open OnDemand

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

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

From there you will need to:

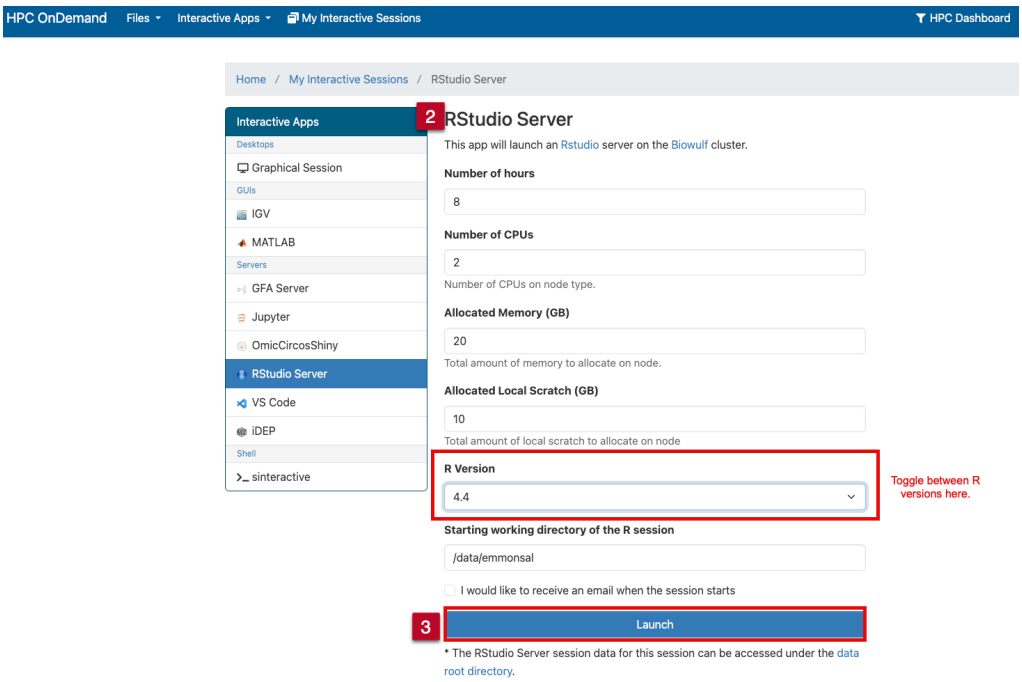
1. Select RStudio Server.



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

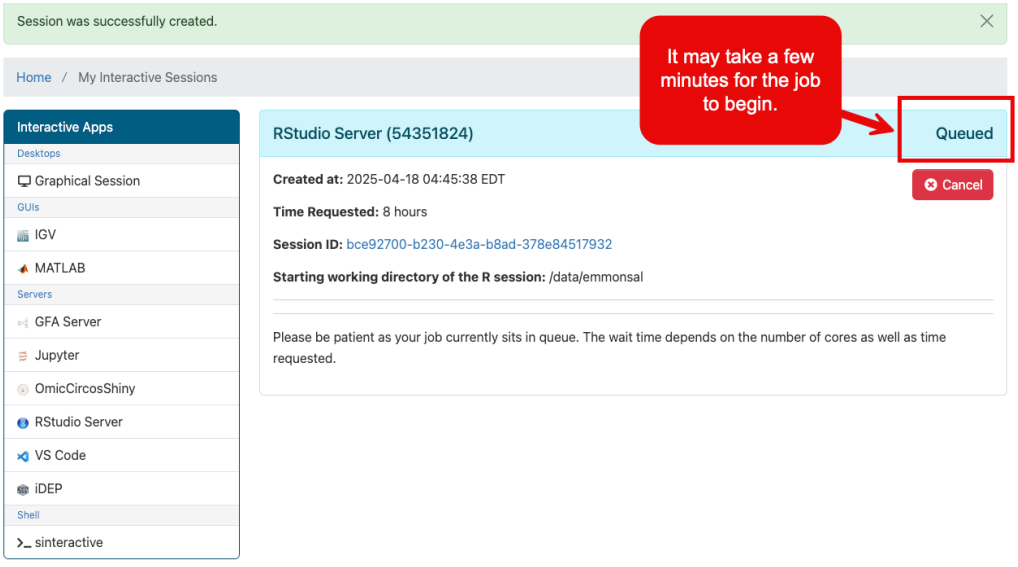
2. Select parameters for your RStudio session including the version of R you want to use.

3. Click "Launch" to start the session.



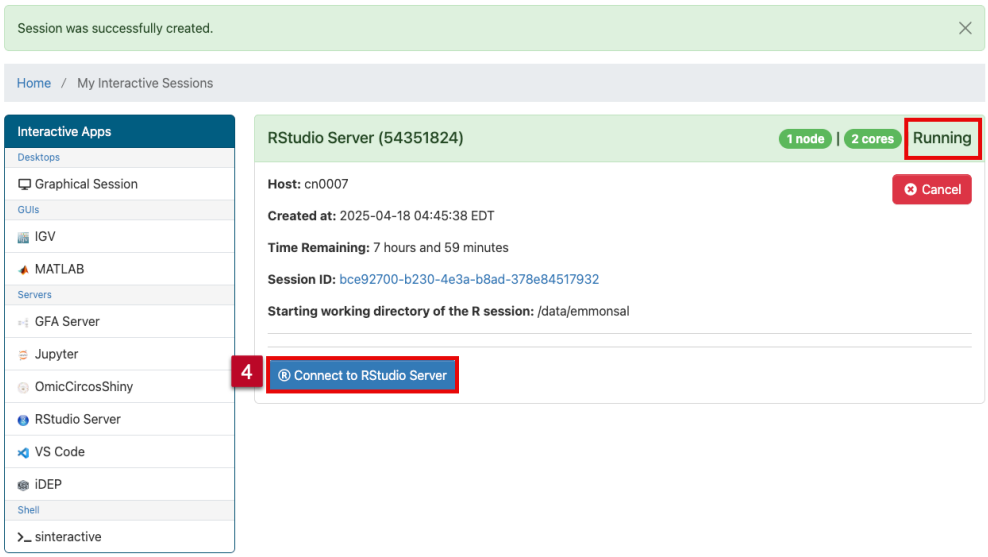
Step 2, 3: Alter any job parameters as you see fit and launch the session.

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



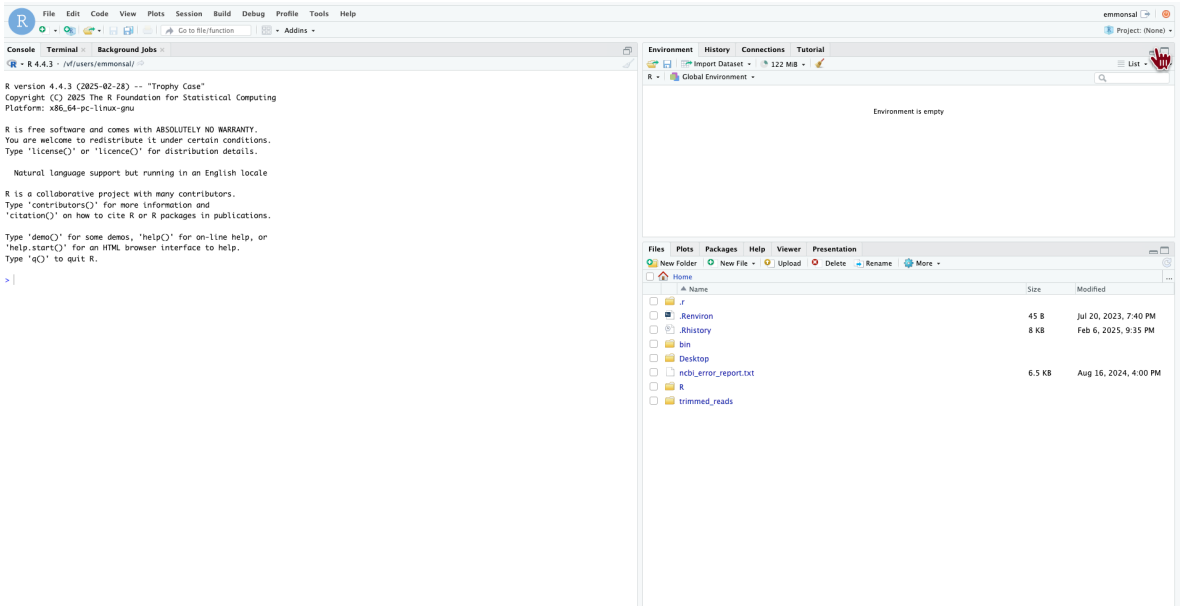
Session is queued.

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



Step 4: Connect to RStudio Server.

Congratulations! You are now connected.



RStudio Server on Biowulf

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

Creating an R project

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

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

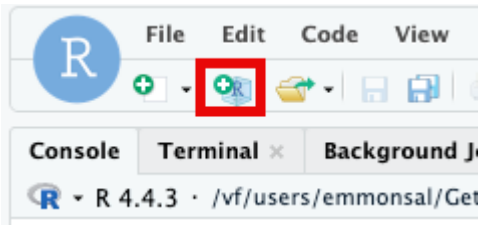
Creating an R project *(https://docs.posit.co/ide/user/ide/guide/code/projects.html)* for each project you are working on facilitates organization and scientific reproducibility.

An RStudio project allows you to more easily:

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

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

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



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

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

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

Why renv?

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

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

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

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

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

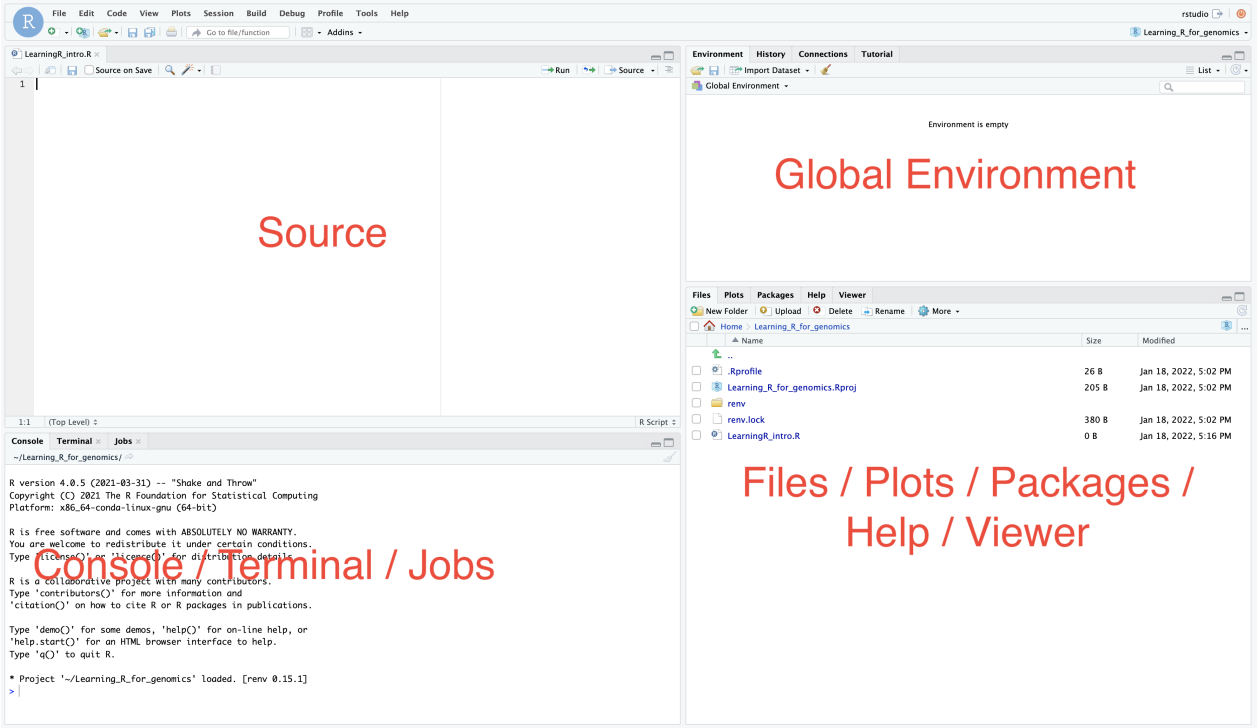
> **Reproducibility**
>
> There is even more that can be done to make projects reproducible beyond R Projects and `renv`. For example, you can use version control (git), R packages, and containerization (e.g., Singularity, Docker).

Creating an R script

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

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

Introduction to the RStudio layout



Let's look a bit into our RStudio layout.

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

Look under the files tab

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

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

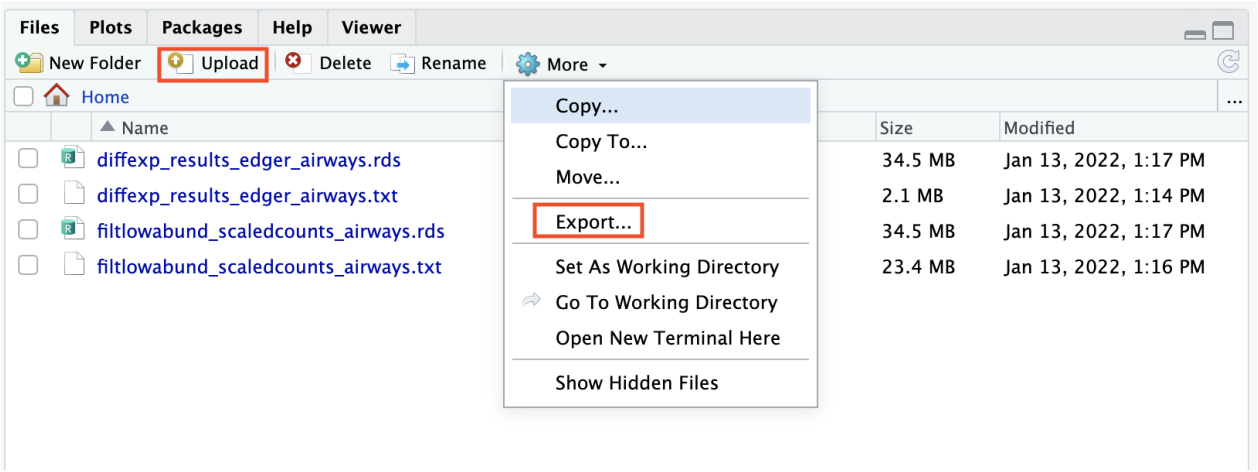
Also, you can change your RStudio layout. See this blog *(https://www.r-bloggers.com/2018/05/a-perfect-rstudio-layout/)* if you are interested. **For simplicity, please do NOT change the layout during this course.**

When to use `Source` vs `Console`?

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

Uploading and exporting files from RStudio Server

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



Data Management

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

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

Saving your R environment (.Rdata)

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

Restoring your R environment

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

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

What is a function?

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

> A function in R (or any computing language) is a short program that takes some input and returns some output.
>
> An R function has three key properties:
>
> - Functions have a name (e.g. dir, getwd); note that functions are case sensitive!
> - Following the name, functions have a pair of ()
> - Inside the parentheses, a function may take 0 or more arguments --- datacarpentry.org *(https://datacarpentry.github.io/genomics-r-intro/00-introduction.html#using-functions-in-r-without-needing-to-master-them)*.

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

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

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

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

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

in blue followed by the output.

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

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

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

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

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

What is a path?

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

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

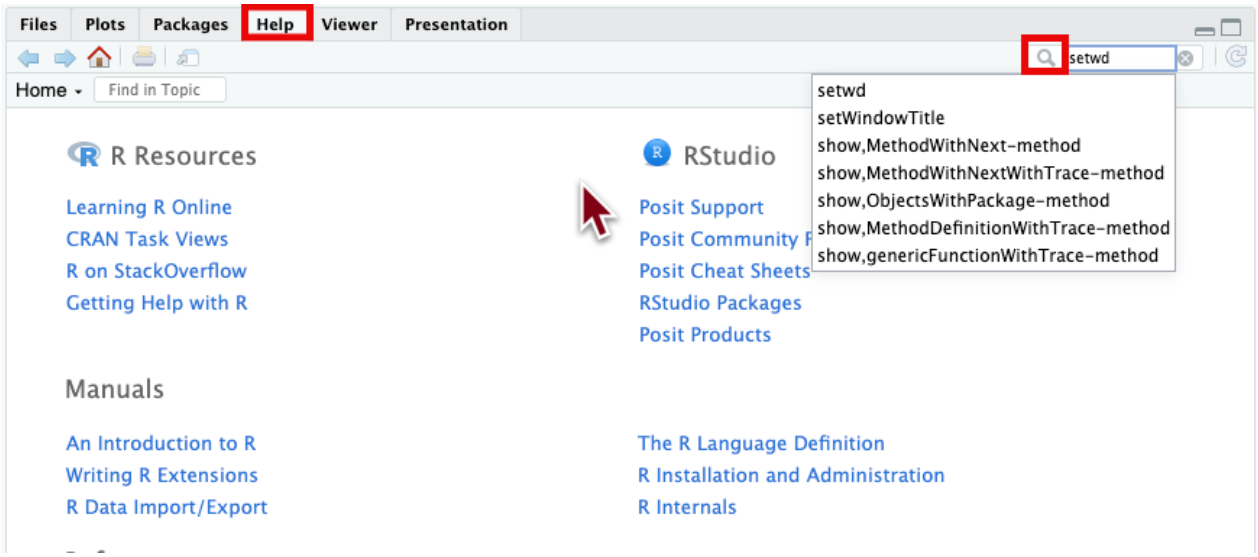
The working directory is the location in your file system that you are currently working in. In other words, it is the default location that R will look for input files and write output files.

> **Note**
>
> R uses Unix formatting for directories, so regardless of whether you have a Windows computer or a mac, the way you enter the directory information will be the same. You can use tab completion to help you fill in directory information.

Getting help

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

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



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

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

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

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

```
args(setwd)
```

```
function (dir)
NULL
```

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

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

Additional Sources for help

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

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

To provide details about your R session, use

```
sessionInfo()
```

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

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

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

time zone: Europe/Berlin
tzcode source: internal

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

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

Acknowledgments

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

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

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

Objectives

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

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

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

R objects

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

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

Creating and deleting objects

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

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

> **Using = for assignment?**

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

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

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

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

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

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

```
[1] 1
```

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

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

```
a + 5
```

```
[1] 6
```

```
5 - a
```

```
[1] 4
```

```
a^2
```

```
[1] 1
```

```
a + a
```

```
[1] 2
```

Naming conventions and reproducibility

There are rules regarding the naming of objects.

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

   For example:

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

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

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

In contrast:

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

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

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

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

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

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

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

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

Reassigning objects

To reassign an object, simply overwrite the object.

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

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

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

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

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

Deleting objects

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

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

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

Object data types

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

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

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

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

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

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

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

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

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

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

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


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

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

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

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

Special null-able values

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

Mathematical operations

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

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

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

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

Let's see this in practice.

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

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

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

```
[1] 46
```

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

For example,

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

```
[1] 1.618034
```

A function is an object.

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

Looking up Compiled Code

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

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

Consider

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

```
[1] 6
```

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

```

```

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

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

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

```
[1] 5.7
```

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

The pipe (|>, %>%).

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

For example,

```
ex<- -5.679

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

```
[1] 6
```

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

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

Pre-defined objects

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

Acknowledgments

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

Practice 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
```

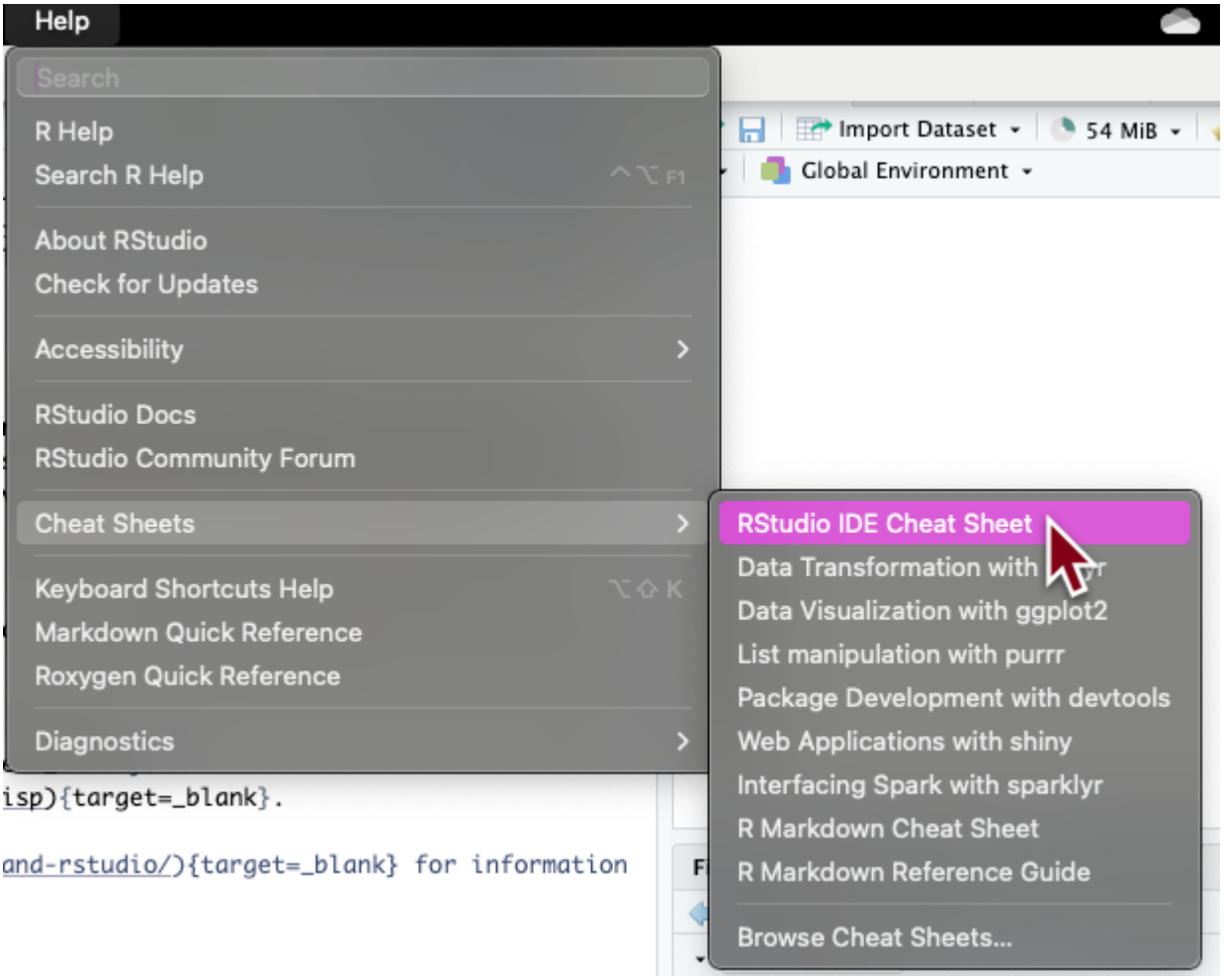
Additional Resources

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/)*