Introduction to R for Mass Spectrometrists

Jeff Jones, Heath Patterson, Ryan Benz

2023-03-31

Preface

Who

The Authors Jeff Jones (PhD)

Heath Patterson (PhD)

Ryan Benz (PhD)

The Audience The topics covered in this book are detailed towards an absolute beginner in R and possibly to programming in general. They are intended to have little to no surprises, and taken from begging to end, one should be able to extend their towards developing analytically processes for their own research, providing means for others to accomplish analyses and begin to extend their skills with more advanced literature.

This book is meant for individuals with no previous knowledge of R, although some experience with data and statistical analysis is recommended. Reading and understanding the chapters and exercises should impart the skills for basic data analysis and prepared you for more advanced concepts and skills.

What

Covered Goals for the Short Course - Learn basic fundamentals of the R programming language - Learn how to use to the RStudio integrated development environment (IDE) - Learn about tidy data, what it is, and why it's important for data analysis - Learn basic fundamentals of the tidyverse ecosystem of R packages and how they can be used to streamline the data analysis process - Learn how to make data visualizations using the ggplot2 R package

At the end of the course, are you able to. - start-up RStudio and make an RStudio project? - read a formatted text file (e.g. csv file) into R? - understand basic properties about the data (e.g. # rows, cols)? - tell someone what tidy data is and why it's important? - perform some basic data manipulations and operations on the data? - make a simple plot from the data?

Not Covered Not covered are the topics of statistical analysis, probability, regression, machine learning or any other advanced analytical topic. Additionally, this book does not cover constructing R packages, documentation, markdown or any other advanced R programming topic.

Why

This book serves as the basis for the Introduction to R course offered at the annual conference for the American Society for Mass Spectrometry (ASMS), and was developed by the authors out of the need to create a more permanent, expandable and revisable reference document.

When

Started in 2017 as a series of workshops at the annual ASMS conference drawing between 200 and 300 attendees for three consecutive years. The presenters were offered a formal short course starting in 2020 (a remote COVID year). In 2023 it was decided to convert all the teaching materials into a formal book.

Where

Online You can find the online version here [link].

In Print You can find a print version at XYZ-Press [link].

In-Person This book and it's contents are covered at the annual American Society for Mass Spectrometry (ASMS), typically the first week in June the weekend prior to the scientific meeting.

How

Persistence, sweat, tears, and bookdown, of'course.

Acknowledgements

A special thanks to the American Society of Mass Spectrometry for allowing us to provide an annual basic R introductory course.

Introduction to R

Before we get started, this book contains some basic cues to help facilitate your understanding of the current topic.

1. We small notes and tips as a block quote, see below.
2. At the beginning of each chapter there is a section on what you should be able to accomplish at the end.
3. At the end of each chapter there are some exercises that extend what has been covered.

   At the end of this chapter you should be able to:

   - understand why R is a good choice for data analysis
   - understand that you have just started the learning curve and all your efforts hence forth are worth it

Why choose it?

R is a great language for data analysis! - Many programming languages are general purpose (can be used in any domain), e.g. C/C++, Java, Python - R is not a general purpose programming language, it's a language specifically designed for working with data (that's what scientists do!) - Because R is geared toward data, its design, structure and continued development is focused on making it easier to work with data - R has become one of the top languages for data science, and it's popularity and usage continues to grow - For scientists, R is a great tool to learn

Fast, nimble, forgiving

Lots of specialty tools (CRAN, Github, Bioconductor)

What you can do with it

Read and plot data

construct analysis pipelines

prototype new algorithms

write packages to share

The R Learning Curve

The learning curve for R 10+ years ago was difficult as there where fewer R resources, it was less mature with not a lot of interest. Additionally, there were fewer people in the community and data science wasn't "a thing" yet.

The R programming language is still challenging but worth it. With the introduction of packages encompassed in the tidyverse there are more high-quality resources, mature utilization with well documented explanations and examples. Currently there is lots of current interest in R with a large community of users and developers. Additionally, the data science "revolution has pushed R to develop and evolve, become more user-centric.

Thoughts about learning R and how to code

The first step to learning a programming language is to learn its syntax (the set of rules and symbols that make up structurally correct code). Computers are really picky, and even the smallest violations of the syntax rules will result in code that doesn't run (typos, incorrect names, missing spaces or too many spaces, wrong brackets, . . .). Syntax errors are frustrating, and unfortunately, the most common mistake for beginners. Hang in there, start simple, try to understand very simple cases first, then build and expand on them

Alternatives

Scripting: Python, Julia, Matlab

- Python is a great language, and is another top language for data science
- If you want to become a data scientist or work heavily with data, learning both R and Python is a great idea – both have pros and cons
- But, if you choose Python over R, that's not a bad choice either
- R might work better for some, Python for others
- If R just isn't working for you, give Python a shot

Complied: C, C++, Java, Rust, etc.

Did you know

R can use functions from Python, C and Rust

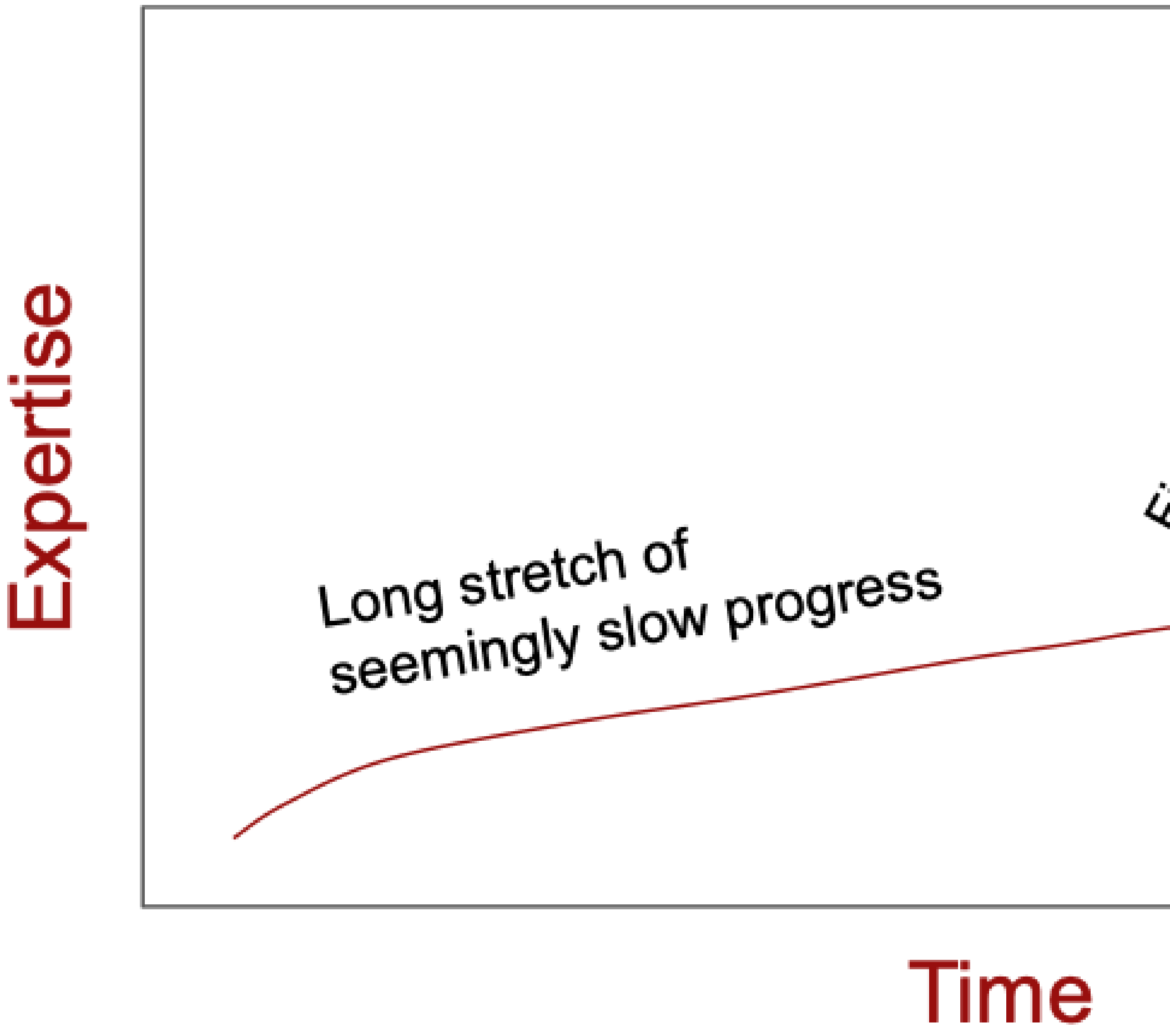


Figure 1: R learning curve past

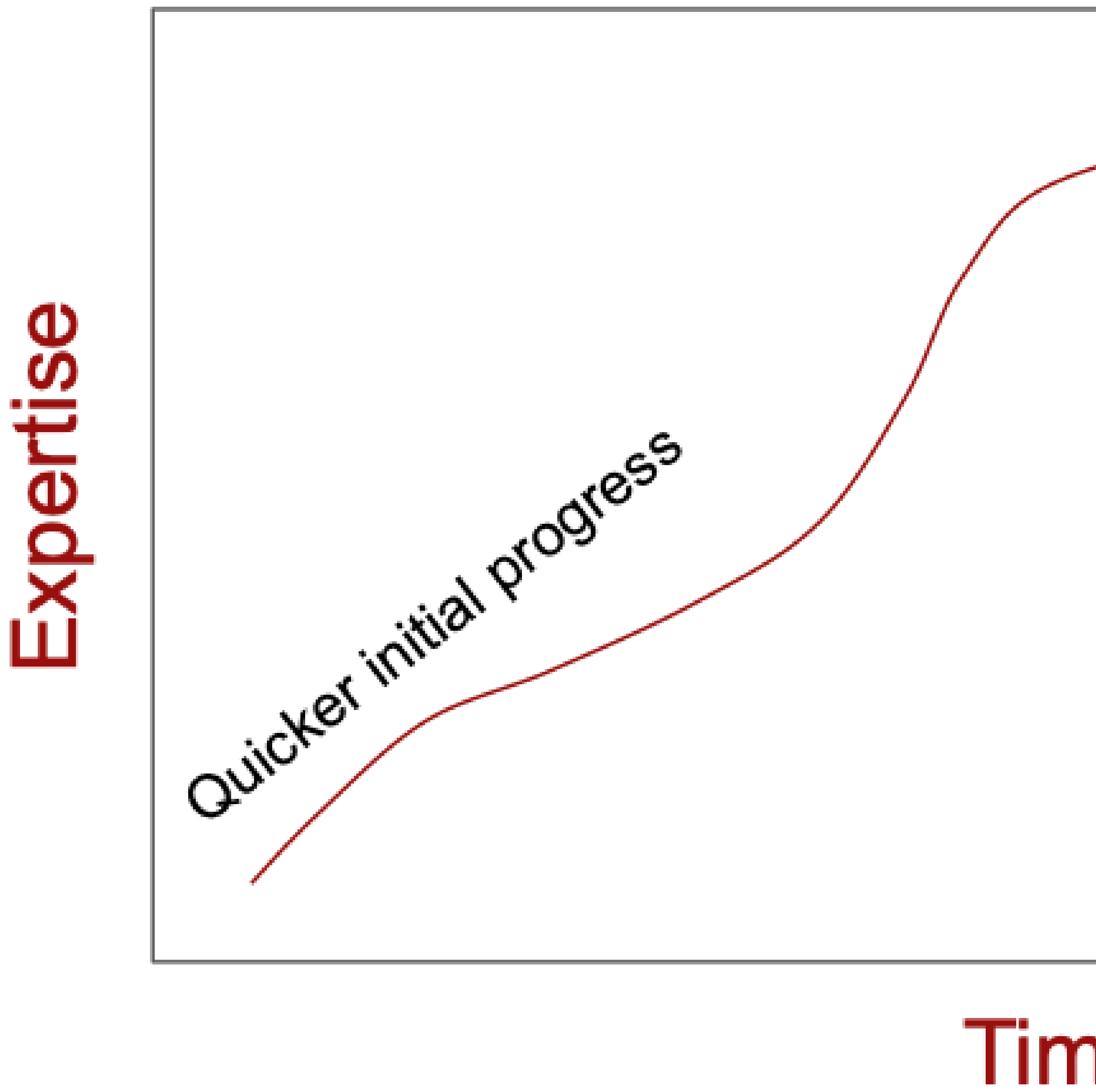


Figure 2: R learning curve present

Schedule

Segment	Chapters	?
Day 1 Morning	Intro and Basics	
Day 1 Afternoon	tidyverse (w/ data set)	
Day 2 Morning	ggplots / mass spec-ish	
Day 2 Afternoon	(tidyverse + ggplot)	
	mass spec data	

Installation

In the scope of this book, there are three main components that need to be installed, and periodically updated:

- The R interpreter - the software that understands math and plotting
- RStudio IDE - the software that makes it easy write code and visualize data
- R Packages - bits of R code that perform specalized operations

In this book we will be utilizing the RStudio integrated development environment (IDE) to interact with R. Two separate components are required for this - the R interpreter and the RStudio IDE. Both are required as the RStudio IDE only provides an interface for the R interpreter, which reads the code and does all the mathematical operations. The R interpreter can be used alone, interacting through the command line (eg. Windows CMD, MacOS and Linux Terminal), a plain text editor or another IDE such as Xcode, VSCode, Eclipse, Notepad++, etc. Rstudio provides a comprehensive, R specific environment, with auto-complete, code syntax highlighting, in-editor function definitions along with package management and plot visualizations.

R interpreter

The underlying "engine" for R programming language can be downloaded from The R Project for Statistical Computing. R is an open-source implementation of the S statistical computing language originally developed at Bell Laboratories. Both langauges contain a variety of statistical and graphical techniques, however, R has been continually extended by professional, academic and amateur contributors and remains the most active today. With the advent of open-source sharing platforms such as GitHub, R has become increasingly popular among data scientists because of its ease of use and flexibility in handling complex analyses on large datasets. Additionally, one of R's strengths is the ease with which well-designed publication-quality plots can be produced.

Steps

1. Navigate to The R Project
2. Click on CRAN under Download, left-hand side
3. Click on https://cloud.r-project.org/ under 0-Cloud
   This will take you to the globally nearest up-to-date repository
4. Click on `Download for ...` and choose the OS compatible with your device

Windows OS

Click on `base`

MacOS

For an Intel CPU: click `R-4.x.x.pgk` to download
For an M1 CPU: click `R-4.x.x-arm64.pkg` to download

After downloading, double-click the installer and follow the instructions

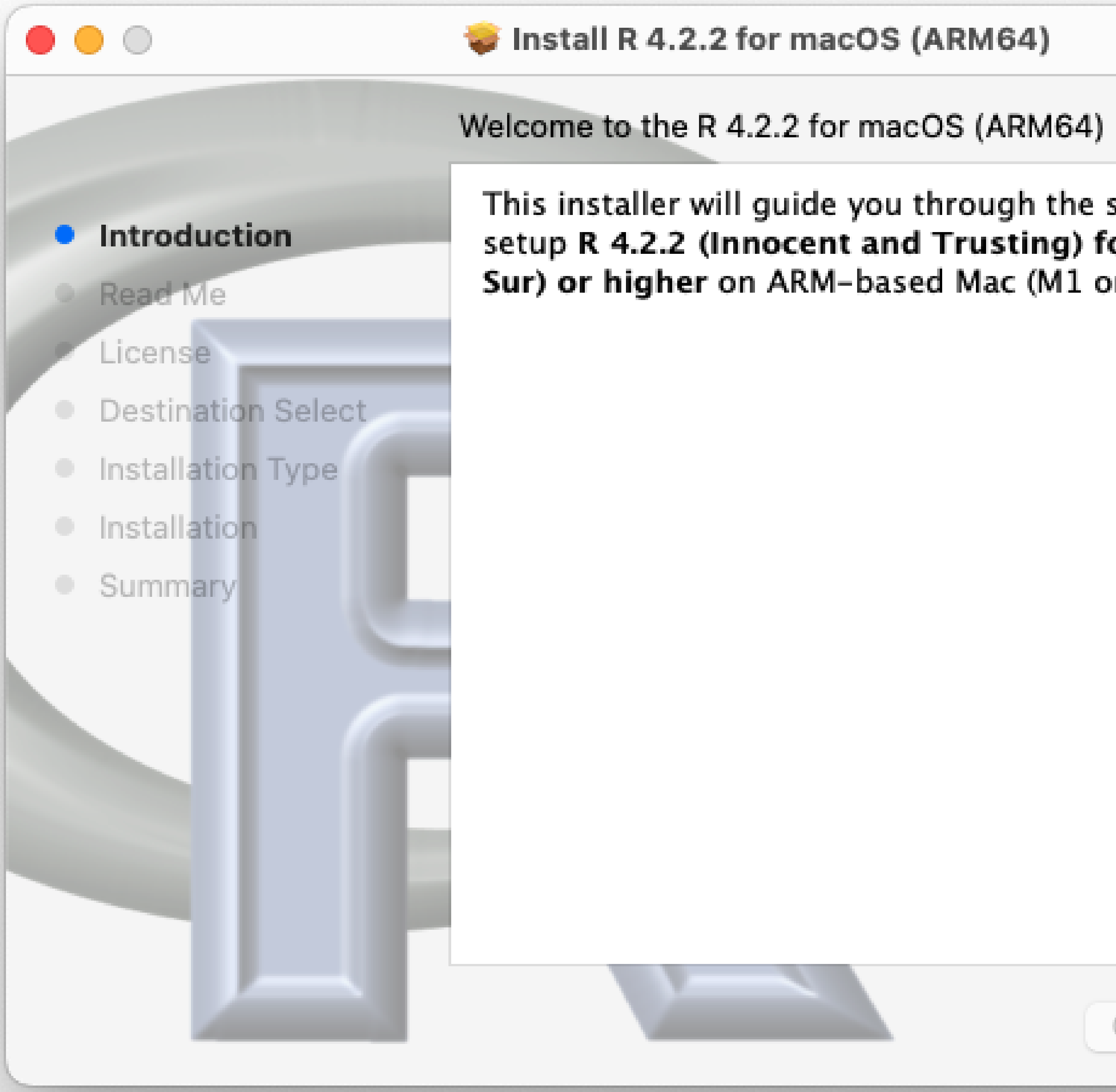
Linux

Click on your distribution and follow the instructions provided. Most of these instructions require knowledge of the Terminal and command line interface for *unix systems.

Rstudio

RStudio, prior to 2023, was an independent software provider for the ever-popular RStudio products, which included both the desktop and server based IDEs, along with the RShiny applications and servers that facilitate easy-to-build interactive web applications straight from R, and deployed on the web. The last chapter in this book will explore the `tidyproteomics` package which also has a Shiny web application. RStudio announced at the beginning of 2023 a soft pivot to Posit, which essentially is a rebranding of the RStudio company to encompass a larger data science audience, one that also provides integration with the Python programming language inside the RStudio IDE.

> The most trusted IDE for open source data science
>
> "RStudio is an integrated development environment (IDE) for R and Python. It includes a console, syntax-highlighting editor that supports direct code execution, and tools for plotting, history, debugging, and workspace management. RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux)."
>
> — www.posit.co (Jan 2023)



Welcome to the R 4.2.2 for macOS (ARM64)

This installer will guide you through the s
setup **R 4.2.2 (Innocent and Trusting) fo**
Sur) or higher on ARM-based Mac (M1 o

Figure 3: Mac Installer

Steps

1. Navigate to posit.co, alternatively rstudio.com redirects to the Posit website.
2. **Click** `Download RStudio` in the menu top right
3. Select `RStudio Desktop`
4. **Click** `Download RStudio`
   skip 1: Install R
5. **Click** `Download RStudio Desktop for ..`

Windows OS

MacOS Opening the .dmg file shows the archive that can be copied into the Applications folder simply by click-dragging the application onto the Applications folder shortcut.

Linux

IDE Layout

The RStudio Integrated Development Environment (IDE) is a powerful tool that can make your data analysis and coding tasks more manageable. One of the key features of the RStudio IDE is that it consists of four individual panes, each containing parts of the total environment. This makes it easier for you to navigate your coding and analysis tasks.

For example, while creating and viewing a plot, you can have the text editor and console open and organized. This way, you can easily see how the code you are writing is impacting the plot you are creating. Having everything in one place can also help reduce the clutter on your desktop, as you don't need to have multiple applications open at the same time.

Overall, the RStudio IDE is an excellent option for anyone looking to streamline their coding and data analysis workflows. By taking advantage of its various features, you can make your work more efficient and enjoyable.

The Editor

> **Tabs**: `All Open Files`

The *Editor* is a tool that allows you to write R code with ease. It is essentially a text editor, but with the added benefit of having knowledge of R. This means that it can automatically color different parts of your code based on their function. This can be a huge time saver, as it makes it easier to read and understand your code.

For example, comments in R code start with a hash (#) symbol. In the *Editor*, these comments are colored light green, making them easy to spot. Similarly,

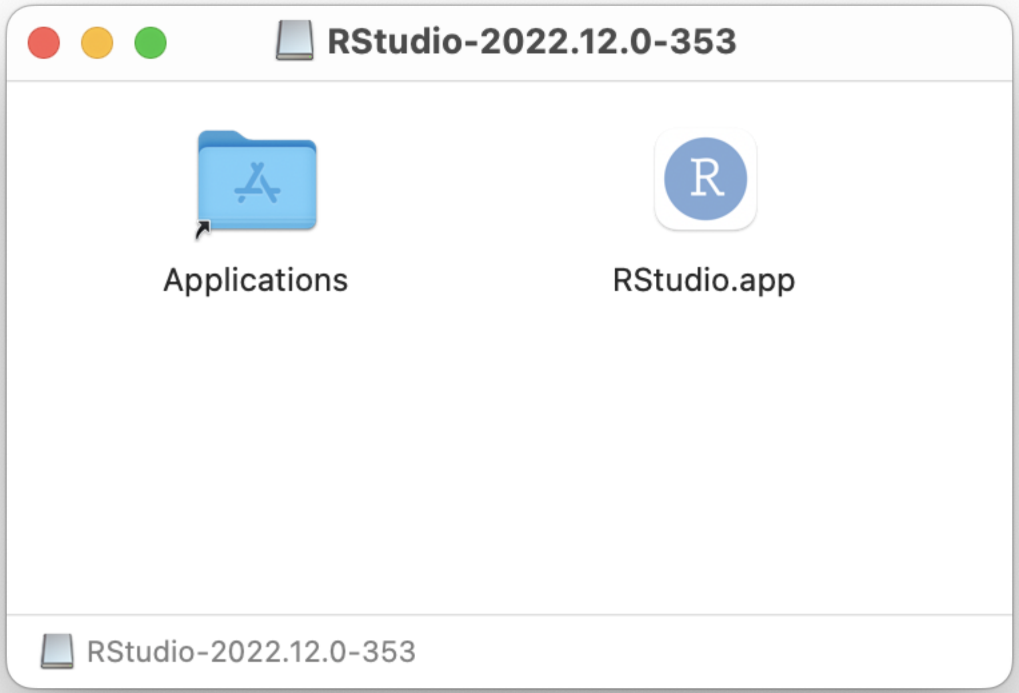
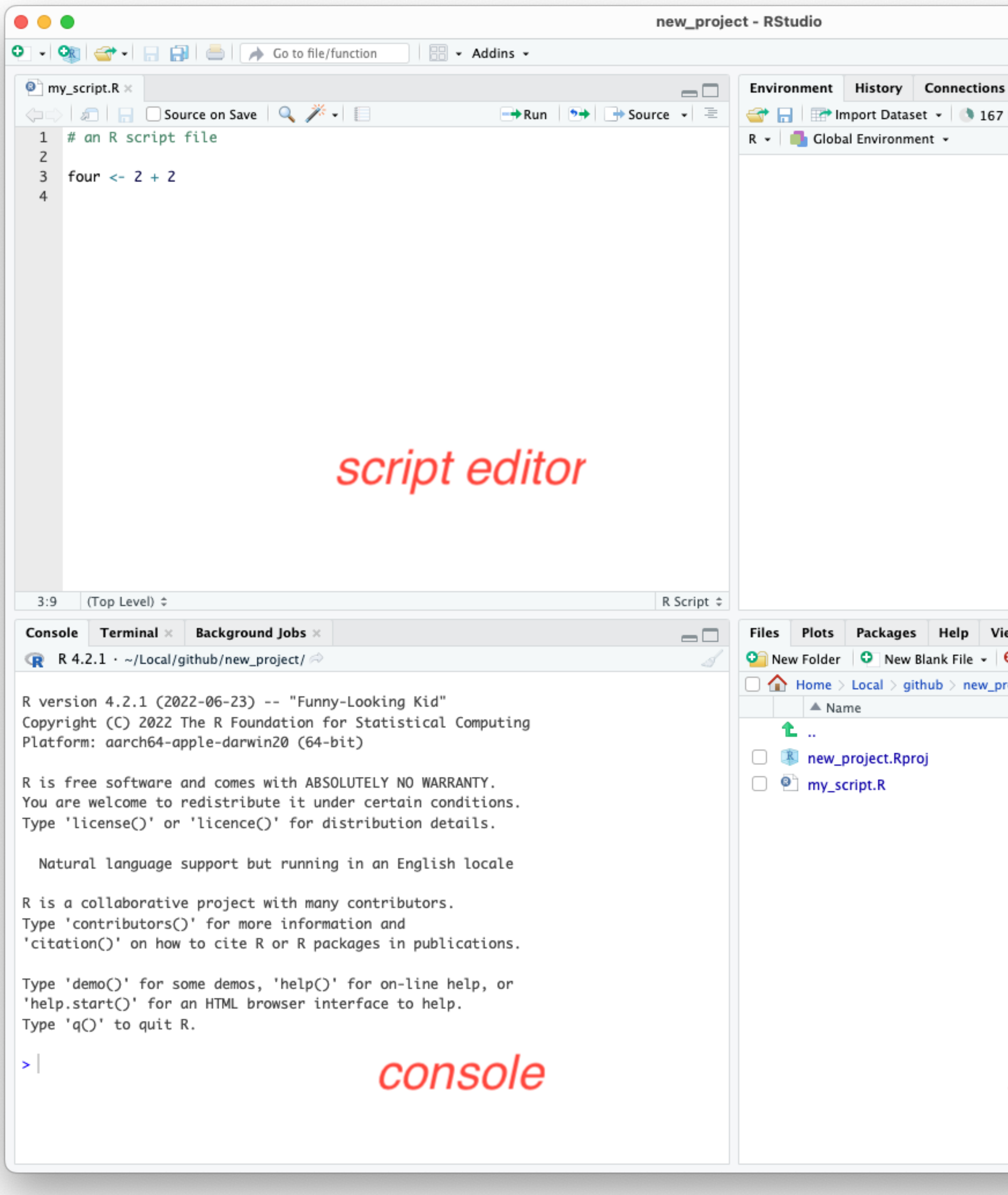


Figure 4: RStudio IDE Install



```r
# an R script file

four <- 2 + 2

```

```
R version 4.2.1 (2022-06-23) -- "Funny-Looking Kid"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 5: RStudio IDE in the default layout

operators like the plus sign (+) and the assignment operator (<-) are colored light blue. This makes it easy to identify where these operators are being used in your code.

Variables are an important part of any programming language, and R is no exception. In the *Editor*, variables are colored black. This makes it easy to distinguish variable names from other parts of your code. Finally, quoted text (also known as strings) are colored purple. This makes it easy to identify where strings are being used in your code.

In summary, the *Editor* is a powerful tool that can help you write R code more efficiently. By automatically coloring different parts of your code, it makes it easier to read and understand. Whether you are a beginner or an experienced R programmer, the *Editor* can help you write better code in less time.

The Editor also has the ability to suggest available variables and functions. In the image provided, the editor suggests using the mean() function to calculate the average of a collection of values. A pop-up with a description accompanies the suggestion. This feature occurs after typing in the first three letters of anything, and the editor will try to guess what you want to type next. This is a helpful tool that can save you time and effort when writing R code.

Files and Plots

Tabs: `Files`, `Plots`, `Packages`, `Viewer`, and `Presentation`

When you're working in RStudio, your workflow is made simple with the various tabs and features available. For instance, the script that you're currently working on is saved to the current project and can be accessed via the *Files* tab located on the top right-hand side of the pane. This tab provides an overview of all the files in the working directory, and you can easily navigate between them.

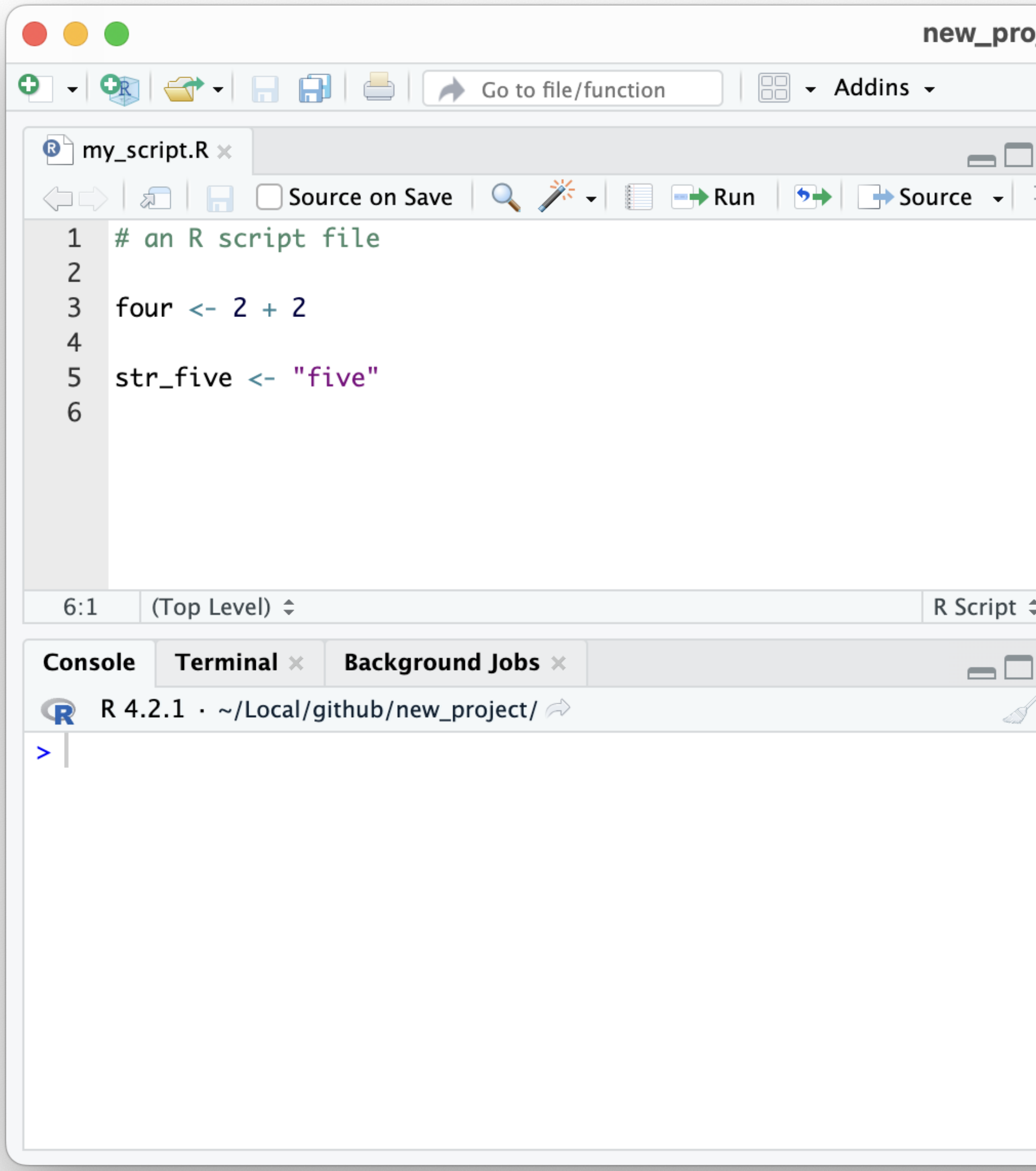
If you need to open another file, you can do so by clicking on the *File* menu or by using the shortcut key. When you open a new file, it will create a new tab in the *Editor* pane, which allows you to switch between open files. This feature is super helpful when you're working on multiple files simultaneously.

Another useful tab located in the same pane is the *Plots* tab. This tab provides a quick way to view any active plots instantly. You don't need to export your plots or save them separately. Instead, you can view them right within RStudio. This is where RStudio truly shines, as it brings together editing and visualization in one application.

The Console

Tabs: `Console`, `Terminal`, and `Background Jobs`

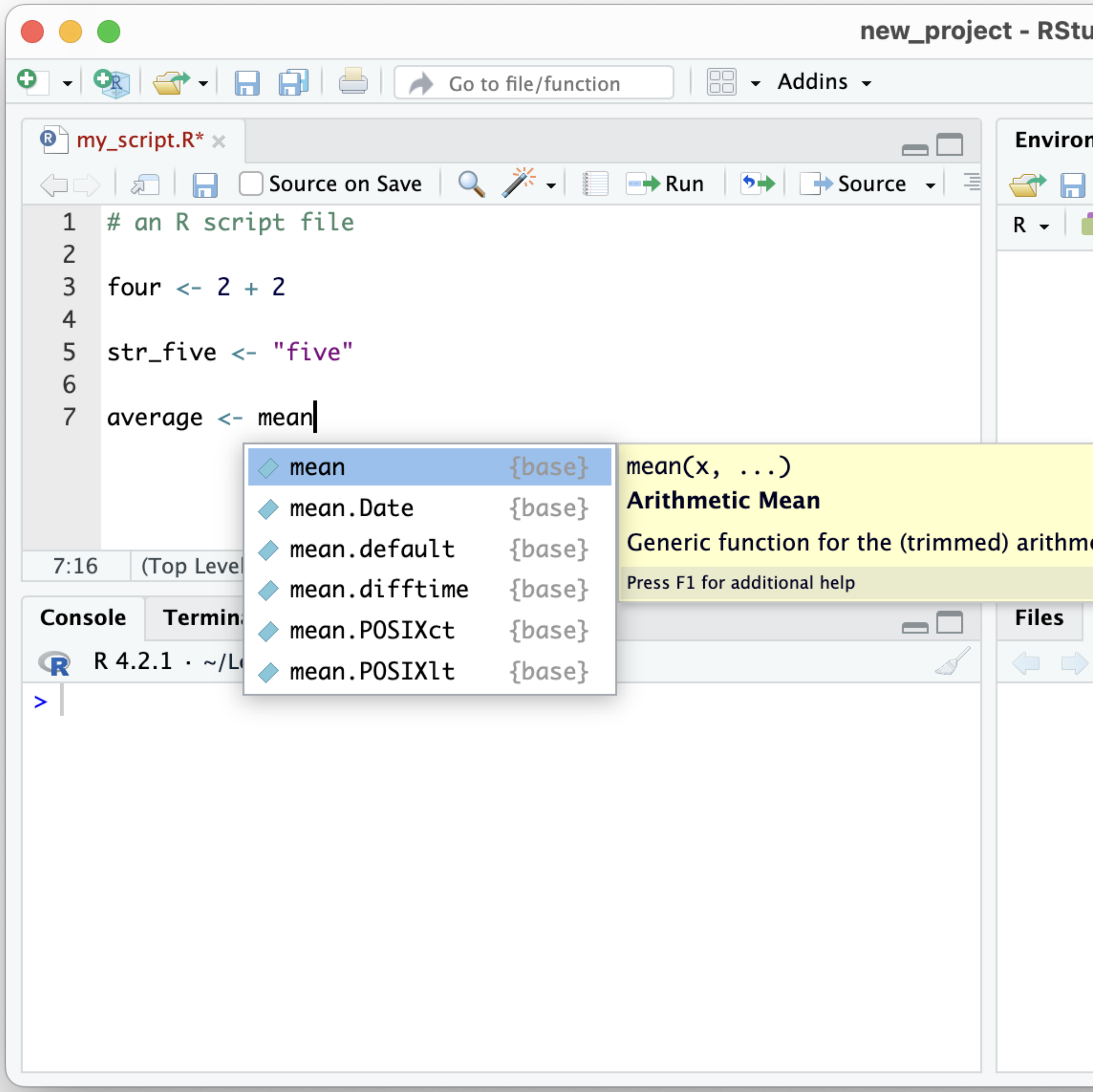
In the RStudio IDE, the *Console* pane is where lines of code are executed from the editor. It is a vital component of the RStudio interface that allows users to interact with R in real-time. The *Console* pane is not only where



```
# an R script file

four <- 2 + 2

str_five <- "five"

```

```
>
```

Figure 6: RStudio IDE syntax highlighting



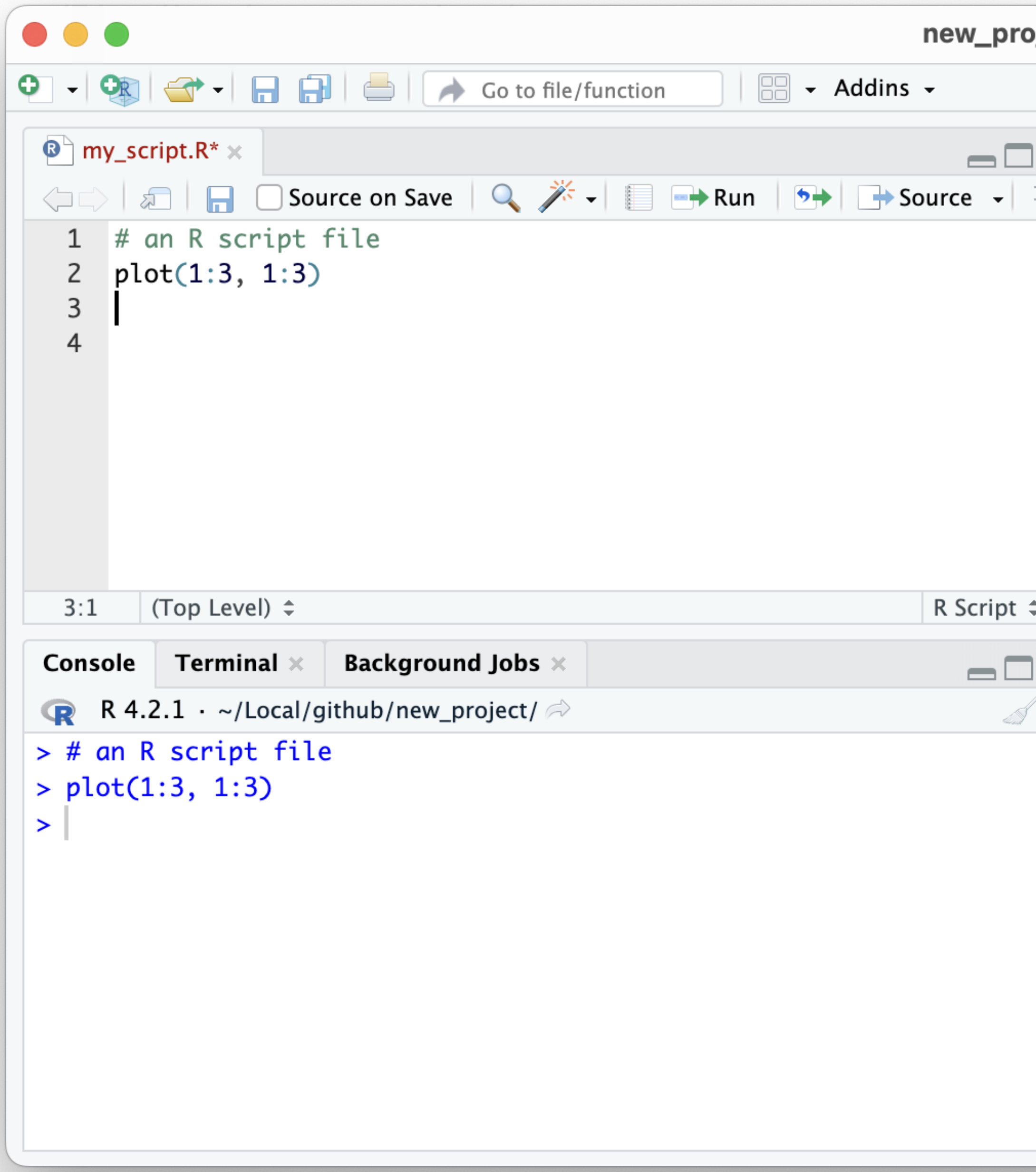
```
# an R script file

four <- 2 + 2

str_five <- "five"

average <- mean
```

```
mean           {base}
mean.Date      {base}
mean.default   {base}
mean.difftime  {base}
mean.POSIXct   {base}
mean.POSIXlt   {base}
```

mean(x, ...)
Arithmetic Mean
Generic function for the (trimmed) arithm
Press F1 for additional help

```
>
```

Figure 7: RStudio IDE auto complete



```r
# an R script file
plot(1:3, 1:3)
```

```
> # an R script file
> plot(1:3, 1:3)
>
```

Figure 8: RStudio IDE plot window

code is run, but it is also where users can view output and error messages. Additionally, the *Console* pane provides users access to the computer's terminal. This feature allows users to execute commands outside of the R environment, such as navigating files and directories or installing packages. Overall, the *Console* pane is an essential tool for any RStudio user and should be utilized to its full potential.

Environment

Tabs: `Environment`, `History`, `Connections`, and `Tutorial`

When you're working on a project in R, it's essential to keep track of the variables and functions that you're using in your current session. The *Environment* tab, located at the top left of the RStudio interface, provides a concise summary of in-memory variables and functions that were created locally, as opposed to functions that were loaded from a package.

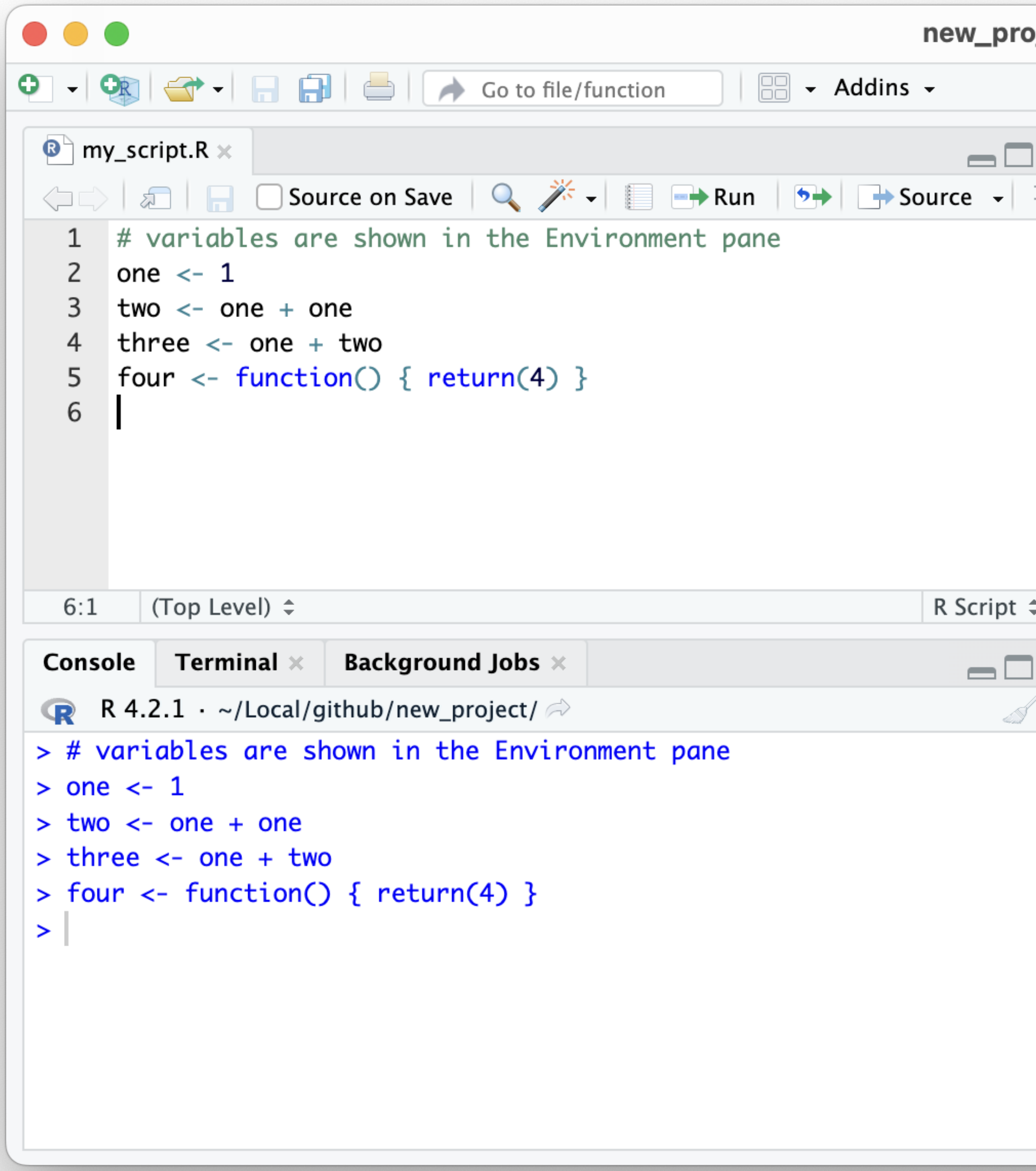
This summary can be useful for new-comers to R because it allows you to quickly see what objects you are currently working with, without having to remember each or manually check. By having a clear overview of your current session, you can avoid mistakes or errors that might arise from using the wrong object or function.

Overall, the *Environment* tab is a helpful feature of RStudio that can save you time and frustration. If you're new to R or just starting to use RStudio, make sure to keep an eye on the *Environment* tab and make use of its features as often as possible. As you become more versed in RStudio this tab may become less relevant.

Usage .. Running lines of code in RStudio

Run from the editor (recommended)

1. Type in the code in the Editor (top-left pane)
2. Put editor cursor anywhere on that line
3. Press Ctrl/CMD+Enter.
4. Multiple lines: highlight multiple lines then press Ctrl/CMD+Enter #### Run from the onsole (occasionally)
5. Type code into Console (bottom-left) after the '>'
6. Press Enter.
7. Multiple lines, not advised, but copy and paste multiple lines into console then press Enter.



```
# variables are shown in the Environment pane
one <- 1
two <- one + one
three <- one + two
four <- function() { return(4) }
```

```
> # variables are shown in the Environment pane
> one <- 1
> two <- one + one
> three <- one + two
> four <- function() { return(4) }
>
```

Figure 9: RStudio IDE environment window

Packages

What are R Packages?

R packages are a powerful tool in the R programming language that allow you to easily use code written by others in your own projects. They can save a lot of time and effort in the development of your own code, as they often provide new functions to deal with specific problems. For example, the popular ggplot2 package provides a variety of functions to help you create beautiful visualizations, while the mzR package allows you to read mass spectrometry data files with ease. Additionally, the twitteR package is a great tool for accessing Twitter data and conducting analysis.

Where to get R Packages

It's worth noting that packages can be written by anyone, which means that their quality can vary widely. While there are many high-quality packages available, it's important to be wary of randomly coming across packages on the internet. To ensure that you're working with trustworthy code, it's a good idea to stick with well-established and frequently updated packages from reputable sources such as the CRAN (The Comprehensive R Archive Network) and Bioconductor repositories. By doing so, you can ensure that your code is reliable, efficient, and secure.

- CRAN cran.r-project.org
- Bioconductor bioconductor.org
- GitHub github.com

In addition to using established packages, it's also possible to create your own packages in R. This is a great way to share your own code with others and make it accessible to a wider audience. When creating a package, it's important to follow a set of best practices to ensure that your code is well-documented, easy to use, and compatible with other packages. This includes providing clear and concise documentation, including examples and tutorials, and following established coding conventions.

Another important consideration when working with R packages is version control. It's essential to keep track of the versions of the packages you're using, as updates can sometimes break existing code. By using a tool like Git or GitHub, you can easily manage different versions of your code and keep track of changes over time. This can be especially useful when collaborating with others on a project.

Overall, R packages are an essential tool for anyone working with R. By using established packages and following best practices when creating your own, you can ensure that your code is efficient, reliable, and easy to use. And by using version control, you can keep track of changes over time and collaborate effectively with others.

Installing R Packages

When working with R, it is important to understand how to install packages. R packages are collections of functions, data, and documentation that extend the capabilities of R. Most R packages have binary versions available for direct installation with no additional steps required. Binary packages are pre-compiled and ready-to-use packages that are platform-specific. They can be installed with the `install.packages()` function in R.

> **NOTE**: Follow the examples below to install all the required packages used in this book. Jump to the following section if you run into any issues. Use the copy-paste button in the top-right of each code block.

Installing from CRAN

```r
# this installs all of the packages in the tidyverse collection
install.packages('tidyverse')
```

Installing from Bioconductor

```r
# do this once to install the Bioconductor Package Manager
install.packages("BiocManager")
# this installs the mzR package
BiocManager::install(c("mzR", "xcms", "MSstats", "MSnbase"))
```

Installing from GitHub

```r
# do this once to install the devtools package
install.packages("devtools")
# this installs the tidyproteomics package
install_github("jeffsocal/tidyproteomics")
```

> **NOTE:** There maybe several additional packages to install including additional operating system level installs. Go to the tidyproteomics webpage for additional installation help.

Potential Gotchas

However, there are cases where a binary version of a package may not be available. This could be because the package is new or has just been updated. In such cases, the package may need to be compiled before it can be installed. Compiling a package involves converting the source code into machine-readable code that can be executed.

To compile R packages, you'll need to have the necessary programs and libraries installed on your computer. For Windows, you'll need to install RTools, which provides the necessary tools for package compilation. For Mac, you'll need to install Command Line Tools. Once these tools are installed, you can use them to compile packages that are not available as binaries.

However, it's worth noting that package compilation can sometimes fail for various reasons. This can be frustrating, especially if you're new to R. Therefore, it is generally recommended to stick with using binary packages whenever possible. Binary packages are more stable and easier to install, making them the preferred option for most users.

In summary, when working with R, it's important to understand how to install packages. Most packages have binary versions available for direct installation, but there may be cases where you need to compile a package yourself. While package compilation can be useful in some cases, it can also be frustrating and time-consuming. Therefore, it's generally recommended to stick with using binary packages whenever possible.

Packages Utilized in This Book

tidyverse

The Tidyverse R package is a collection of data manipulation and visualization packages for the R programming language. It includes popular packages such as dplyr, ggplot2, and tidyr, among others. The Tidyverse R package is a powerful and versatile tool for data analysis in R. It includes a collection of data manipulation and visualization packages designed to work seamlessly together, making it easy to analyze and visualize data in R.

```
library(tidyverse)
```

The **readr** package provides a versatile means of reading data from various formats, such as comma-separated (CSV) and tab-separated (TSV) delimitated flat files. In addition to its versatility, the **readr** package is also known for its speed and efficiency. It is designed to be faster than the base R functions for reading in data, making it an ideal choice for working with large datasets.

```
tbl <- "./data/table_peptide_fragmnets.csv" %>% read_csv()
```

```
## Rows: 14 Columns: 7
## -- Column specification ------------------------------------------------
## Delimiter: ","
## chr (4): ion, seq, pair, type
## dbl (3): mz, z, pos
##
## 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.
```

The **tibble** package embodies a modern, flexible take on the data table, making it a powerful tool for data analysis in R. This package includes a suite of functions that allow you to easily manipulate and reshape data. It also has a printing method that makes it easy to view and explore data, even when dealing with large datasets. Additionally, tibble objects are designed to work seamlessly with other Tidyverse packages, such as **dplyr** and **tidyr**, making it easy to switch between packages and maintain a consistent syntax.

```
print(tbl)
```

```
## # A tibble: 14 x 7
##    ion      mz     z seq     pair    pos type
##    <chr> <dbl> <dbl> <chr>   <chr> <dbl> <chr>
##  1 b1+    98.1     1 P       p01       1 b
##  2 y1+   148.      1 E       p06       1 y
##  3 b2+   227.      1 PE      p02       2 b
##  4 y2+   263.      1 DE      p05       2 y
##  5 b3+   324.      1 PEP     p03       3 b
##  6 y3+   376.      1 IDE     p04       3 y
##  7 MH++  401.      2 PEPTIDE p00      NA precursor
##  8 b4+   425.      1 PEPT    p04       4 b
##  9 y4+   477.      1 TIDE    p03       4 y
## 10 b5+   538.      1 PEPTI   p05       5 b
## 11 y5+   574.      1 PTIDE   p02       5 y
## 12 b6+   653.      1 PEPTID  p06       6 b
## 13 y6+   703.      1 EPTIDE  p01       6 y
## 14 MH+   800.      1 PEPTIDE p00      NA precursor
```

The **readxl** package is a complement to **readr** providing a means to read Excel files, both legacy .xls and the current xml-based .xlsx. It is capable of reading many different types of data, including dates, times, and various numeric formats. The package also provides options for specifying sheet names, selecting specific columns and rows, and handling missing values.

The **dplyr** package is widely known and used among data scientists and analysts for its interface that allows for easy and efficient data manipulation in *tibbles*. Providing a set of "verbs" that are designed to solve common tasks in data transformations and summaries, such as filtering, arranging, and summarizing data, all designed to work seamlessly with other Tidyverse packages making it easy to switch between packages and maintain a consistent syntax. One of the key benefits of the **dplyr** package is its ease of use, making it perfect for beginners and advanced users alike. It is widely used in the R community and is a valuable tool for anyone working with R and data tables.

```
tbl %>%
  filter(type != 'precursor') %>%
  group_by(type) %>%
  summarise(
    num_ions = n(),
    avg_mass = mean(mz)
  )
```

```
## # A tibble: 2 x 3
##   type  num_ions avg_mass
##   <chr>    <int>    <dbl>
## 1 b            6     378.
## 2 y            6     424.
```

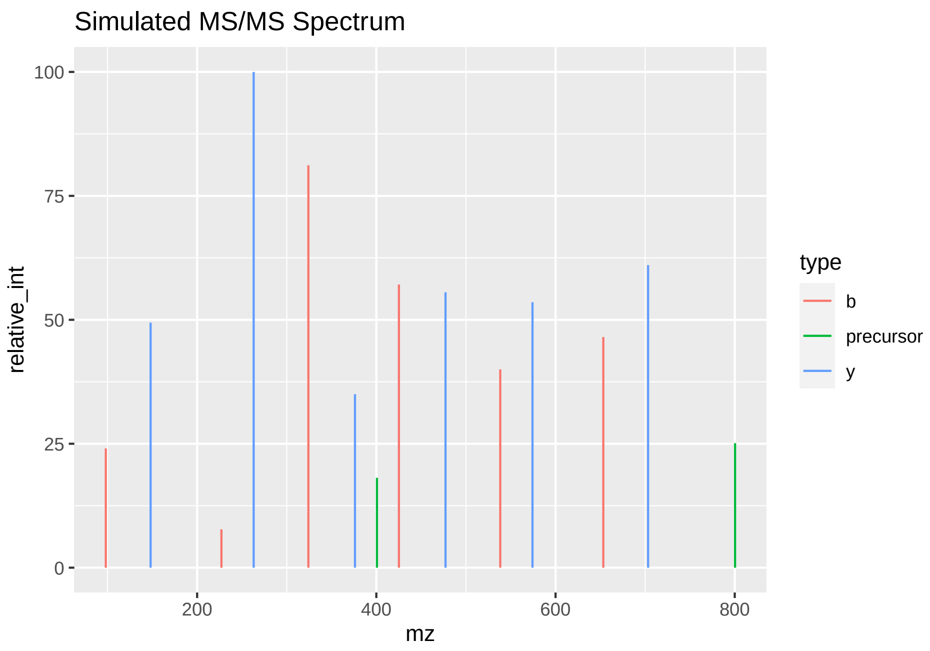
The **tidyr** package contains a set of data table transformations, including pivotting rows to columns, splitting a single column into multiple ones, and tidying or cleaning up data tables for a more usable structure. These transformations are essential for dealing with real-world data tables, which are often messy and irregular. By using **tidyr**, you can quickly and easily manipulate data tables to extract the information you need and prepare them for further analysis.

```
tbl %>%
  filter(type == 'precursor') %>%
  pivot_wider(z, names_from = 'type', values_from = 'mz')
```

```
## # A tibble: 2 x 2
##       z precursor
##   <dbl>     <dbl>
## 1     2      401.
## 2     1      800.
```

The **ggplot2** package stands out as the most advanced and comprehensive package for transforming tabulated data into meaningful and informative graphics. With its wide range of visualization tools, this package allows you to create expressive and compelling graphics that not only look great but also convey detailed information in a clear and concise manner. Apart from other visualization tools, **ggplot2** takes a layered approach to creating graphics, allowing for the additive layering of additional data, labels, legends, and annotations, which helps to provide a more comprehensive view of your analysis.

```
tbl %>%
  mutate(int = rnorm(n(), mean = 1e5, sd=5e4),
         relative_int = int/max(int) * 100) %>%
  ggplot(aes(mz, relative_int, color=type)) +
  geom_segment(aes(xend = mz, yend = 0)) +
  labs(title = "Simulated MS/MS Spectrum")
```



One of the key benefits of using the Tidyverse is the standardization of syntax and functions across each package. This means that once you learn the basics of one package, you can easily switch to another package and be confident in your ability to use it. This makes it easier to create reproducible code and improves the efficiency of your data analysis.

The Tidyverse is widely used in the R community and is a valuable tool for any data scientist or analyst working with R. It is especially useful for those who need to manipulate and visualize data quickly and efficiently, without sacrificing accuracy. Whether you are new to R or an experienced user, the Tidyverse is a must-have tool in your data analysis toolkit.

Mass Spectrometry Specific Packages

This book, while providing a beginners level guide to R programming, also introduces several mass spectrometry-specific packages in many of the code examples. While these examples may only touch on some of their functions, the last chapter is dedicated to a more formal, albeit not comprehensive introduction to many of these packages. For example the `mzR` package, which enables users to read and process mass spectrometry data, as well as the `xcms` package, which is used for preprocessing and feature detection. Additionally, the book introduces the `MSnbase` package, which provides a framework for quantitative and qualitative analysis of mass spectrometry data, and the `MSstats` package, which is used for statistical analysis of quantitative proteomics experiments. Lastly, the book covers the `tidyproteomics` package, which provides a collection of tools for analyzing post-analysis quantitative proteomics data using a framework similar to the `tidyverse`.

The Basics

Welcome to the R Book! In this chapter, we will explore the basics of R, a powerful programming language used for statistical computing and graphics.

At its most fundamental level, R is a calculator capable of performing simple, and complex, mathematical operations. It can read and write data to and from files, manipulate the data, calculate summaries and plot visual representations of the data. Essentially, it is a programmatic version of a spreadsheet program.

However, R is much more than just a calculator. It is also a platform for conducting complex analyses, statistical evaluations, predictive inferencing, and machine learning. With R, you can explore and visualize data in a variety of ways, perform advanced statistical analyses, and build predictive models.

In this chapter, we will start by examining the simplest operations of R. We will cover basic arithmetic, working with variables, and creating basic plots. By the end of this chapter, you will have a solid understanding of the fundamentals of R and be ready to tackle more complex topics.

So, let's get started!

> At the end of this chapter you should be able to:
>
> - understand R's syntax, variables, operators and functions
> - create and edit a project in RStudio

Reserved Words

As we begin our journey, it's important to keep in mind that there are certain reserved words that carry a special meaning and cannot be used as identifiers. These words have been set aside by the R programming language, and using them as variable names or function names could lead to errors in your code.

Therefore, before we dive too deeply into our R programming endeavors, let's take a moment to familiarize ourselves with these reserved words. This will help us avoid potential issues down the road and ensure that our code runs smoothly.

```r
# to read more about them type
?reserved
```

Word	Use
`if`, `else`	flow control, part of the if-then-else statement
`for`, `repeat`, `while`, `break`, `next`	flow control, part of the for-loop statement
`function`	basis for defining new algorithms
`TRUE`, `FALSE`	Boolean logic values
`NULL`	an undefined value
`Inf` , `-Inf`	an infinite value (eg. `1/0`)
`NaN`	'not a number'
`NA`	a missing value indicator

> **NOTE:** a `Null` results when a value is missing and could be a *string* or a *numeric*, where as a NA results when a known value, such as in a column of numbers, is missing.

Syntax

Welcome to the R Book! In this section, we will discuss the different components of R code that you should be familiar with.

R input is composed of typing characters to represent various parts of a process or mathematical operation. These characters come together to form what we call R code. Understanding the different components of R code is essential to writing effective and efficient R programs. Below are some examples of the different components of R code:

- Comments
- Variables
- Operators
- Numerics
- Strings
- Functions

Comments are used to add notes or explanations to your code. Variables are used to store data values that can be used later on in your program. Operators are used to perform mathematical operations, such as addition or subtraction. Numerics are values that represent numerical data, such as integers or decimals. Strings are values that represent text data. Functions are pre-defined sets of instructions that can be called upon to perform specific tasks in your program.

By understanding these different components of R code, you will be able to write more effective and efficient R programs.

> **NOTE:** the following is a code block, you can copy the code here and paste into RStudio

```
# adding two numbers here and storing it as a variable
four <- 2 + 2

# using the function 'cat' to print out my variable along with some text
cat("my number is ", four)
```

```
## my number is  4
```

> **NOTE:** the ## at the beginning of the output is only present in the book, you will not see it in your interactions

Comments

Comments are essential parts of the code you will write. They help explain why you are taking a certain approach to the problem, either for you to remember at a later time or for a colleague. Comments in other coding languages, including R package development, can become quite expressive, representing parts and structures to a larger documentation effort. Here, however, comments are just simple text that gets ignored by the R interpreter. You can put anything you want in comments.

```
oops, not a comment
```

```
# This is a comment

# and here a comment tag is used to ignore legitimate R code
# four <- 2 + 2
four <- 2 * 2
```

Strings

Words composed of alphabet letters, simply put are called strings - probably because they are independent characters (eg, A, B, C, etc.) strung together. While words are pattenred characters representing a linguistic term (eg. noun, verb, adjective), strings do not follow any rules of composition and can simply be a random or semi-random "string" of characters (letters and numbers).

```
# a string can be a word, this is a string variable
three <- 1 + 2
# or an abbreviation, this is a variable (thr) representing the string "three"
thr <- "three"
# a mass spec reference
peptide <- "QWERTK"
# or an abbreviated variable
pep <- "QWERTK"
```

In R strings must be quoted, otherwise the R intrepreter will assume you are referring to a variable, which is not quoted. In the example above, the variable `peptide` contains the string if letters representing th peptide amino acid sequence `"QWERTK"`. In either case, there are no hard and fast rules for how strings and variables are composed, except that variables **cannot** start with a number.

```
# permitted
b4 <- 1 + 3
# not permitted
4b <- 1 + 3. ## Error: unexpected symbol in "4b"
```

There are however, conventions that you can follow when constructing variable names that aid in the readability of the code and convey information about the contents. This is especially useful in long code blocks, or, when the code becomes more complex and divested across several files. For example:

```
# a string containing a peptide sequence
str_pep <- "QWERTK"

# a data table of m/z values and their identifications
tbl_mz_ids <- read_csv("somefile.csv")
```

To learn more about and follow specific conventions, explore the following resources.

Numbers (integers and floats)

Numbers are the foundation upon which all data analysis is built. Without numbers, we would not be able to perform calculations, identify patterns, or draw conclusions from our data. In the programming language R, there are two main types of numbers: `integers` and `floats`. An integer is a whole number with no decimal places, while a float is a number with decimal places. Understanding the difference between these two types of numbers is essential for accurate numerical analysis.

In R, integers are represented as whole numbers, such as 1, 2, 3, and so on, while floats are represented with a decimal point, such as 1.5, 2.75, and so on. It is important to note that integers occupy less space in memory than floats, which can be a consideration when working with large datasets. This means that when possible, it is generally better to use integers over floats in R, as they are more efficient and can improve the overall performance of your code.

```
# integers
1,  12345, -17, 0
```

A float is a real number with a fractional component, represented as a decimal value. The name "float" comes from the way computers do math. In computing, a floating-point number can be represented using an integer with a fixed precision, called the significand, scaled by an integer exponent of a fixed base, such as 10. For example, 12.345 can be represented as a base-ten floating-point number.

Additionally, it's important to be aware of potential errors when working with floats. Due to the way floats are represented in binary, certain decimal numbers cannot be accurately represented. This can result in rounding errors and unexpected behavior in your code. Therefore, it's important to use caution when comparing floats, which is outside the scope of this book. For more information consider using packages, such as `decimal` or `Rmpfr`, that provide precise floating-point arithmetic.

```
# floats
significand <- 12345
exponent <- -3
base <- 10

# 12.345 = 12345 * 10^-3
significand * base ^ exponent
```

Operators

In programming, operators are essential components that allow us to manipulate various data types. Operators are symbols that perform a specific action on one or more operands. The operands could be numeric values, variables, or even strings. These symbols allow us to add, subtract, multiply, and divide numeric values, as well as perform more complex mathematical operations such as exponentiation and modulus. In addition to numeric values, operators can also manipulate string variables. For example, we can use concatenation operators to join two or more strings together. By using operators, we can perform powerful operations that allow us to build complex programs and applications.

> **NOTE:** Remember order of operations (PEMDAS): Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right).

At their very basic, operators allow you to perform ***calculations*** *..*

```
1 + 2
```

```
## [1] 3
```

```
1 / 2
```

```
## [1] 0.5
```

.. ***assign*** *values to string variables ..*

```
myvar <- 1
```

.. and ***compare*** *values.*

```
1 == myvar
```

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

```
2 != myvar + myvar
```

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

Here is a table summarizing of some common operators in R.

Operator	Description	Example
`<-`	assigns numerics and functions to variables	`x <- 1` x now has the value of 1
`+`	adds two numbers	`1 + 2` = `3`
`-`	subtracts two numbers	`1 - 2` = `-1`
`*`	multiplies two numbers	`1 * 2` = `2`
`/`	divides two numbers	`1 / 2` = `0.5`
`^`	raises one number to the power of the other	`1 ^ 2` = `1`
`=`	also an assignment operator	`x = 1` x now has the value of 1
`==`	performs a comparison (exactly equal)	`1 == 1` = `TRUE`
`!=`	performs a negative comparison (not equal)	`1 != 2` = `TRUE`

Variables

Variables are "things" (letters, words, text strings) that can hold information that can vary. In one instance it can hold a number and in another it can hold text. We can use variables to hold all kinds of information and then refer to that variable again and again to retrieve that information, manipulate it with an operation or replace it with an assignment.

```
# create two viables and assign values to each
var_a <- 1
var_b <- 3.14

var_a + var_b
```

```
## [1] 4.14
```

R even has some intrinsic variables that come in handy, like *pi*.

```
pi
```

```
## [1] 3.141593
```

> **NOTE:** in R it is easy to overwrite existing variables, either initalized by R or created by you, causing error and confusion

```
pi <- 9.876543
pi
```

```
## [1] 9.876543
```

Statements

Using a comparison operator, you can make logical comparisons called statements.

Operator	Description	Example
`|`	an either **or** comparison, `TRUE` if both are true `FALSE` if one is false.	`1 == 1 | 1 != 2` = `TRUE` `1 == 1 | 1 == 2` = `FALSE`
`&`	a comparison where **both** must be `TRUE`	`1 == 1 & 1 != 2` = `TRUE` `1 == 1 & 1 != 2` = `FALSE`

> **NOTE:** there are also the double operators || and &&, these are intended to work as flow control operators and stop at the first condition met. In the most recent versions of R, the double operators will error out if a vector is applied.

Functions

In programming, a function is a type of operator that performs a specific task and can accept additional information or parameters. In fact, all operators are functions in a sense, as they take inputs and produce outputs.

The R programming language has a special class of operators called “binary infix” operators. Infix means “in between,” and these operators are placed in between two inputs. These operators have a unique syntax that may confuse beginners, but they are essential for more complex operations in R.

Now, you may wonder why we are discussing these esoteric aspects of R in a beginner’s book. The reason is that understanding these unique features of the language can give you a better understanding of what the R programming language is doing, how it is structured, and how you can relate to it. It is important to have a solid foundation in the basics of any language, but gaining a deeper understanding of its more complex elements can help you become a more proficient programmer.

So, while binary infix operators may seem like an advanced topic, they are an essential part of the R language and can help you unlock its full potential.

```
1 + 2           # as an infix operator
```

```
## [1] 3
```

```
`+`(1,2)        # as the function
```

```
## [1] 3
```

```
sum(1,2)        # same result just using a named function
```

```
## [1] 3
```

```
sum(1,2,3,4,5) # this function however can take in more than 2 values
```

```
## [1] 15
```

We can even create a user defined infix operator ...

```
`%zyx%` <- function(a,b) { a + b }
1 %zyx% 2
```

```
## [1] 3
```

... or just a normal function.

```
zyx <- function(a,b) { a + b }
zyx(1,2)
```

```
## [1] 3
```

The notion of an infix operator you and ignore for the most part. But, we will see it again when diving into the `tidyverse` - a collection of arguably the most powerful data manipulation packages you will encounter. For now, lets move on with more about `functions()`.

Flow-Control

If-Else Statements

Loops

For

While

Projects

Project are how RStudio organizes your work. Think of project as singular goal oriented collection. There are no rules but some basic organizational tips should help simplify your project.

Creating

Creating a new project is very forgiving, you can create a new directory with a project name, or create a project out of an existing directory.

Either 1. Click on the drop down in the top right 2. OR: Under the menu item select `File > New Project`

In the `New Project Wizard` select `New Directory > New Project`, enter the name of the project and click `Create Project`.

Editing

Organizing

Data

Scripts

Results

Tables

Plots

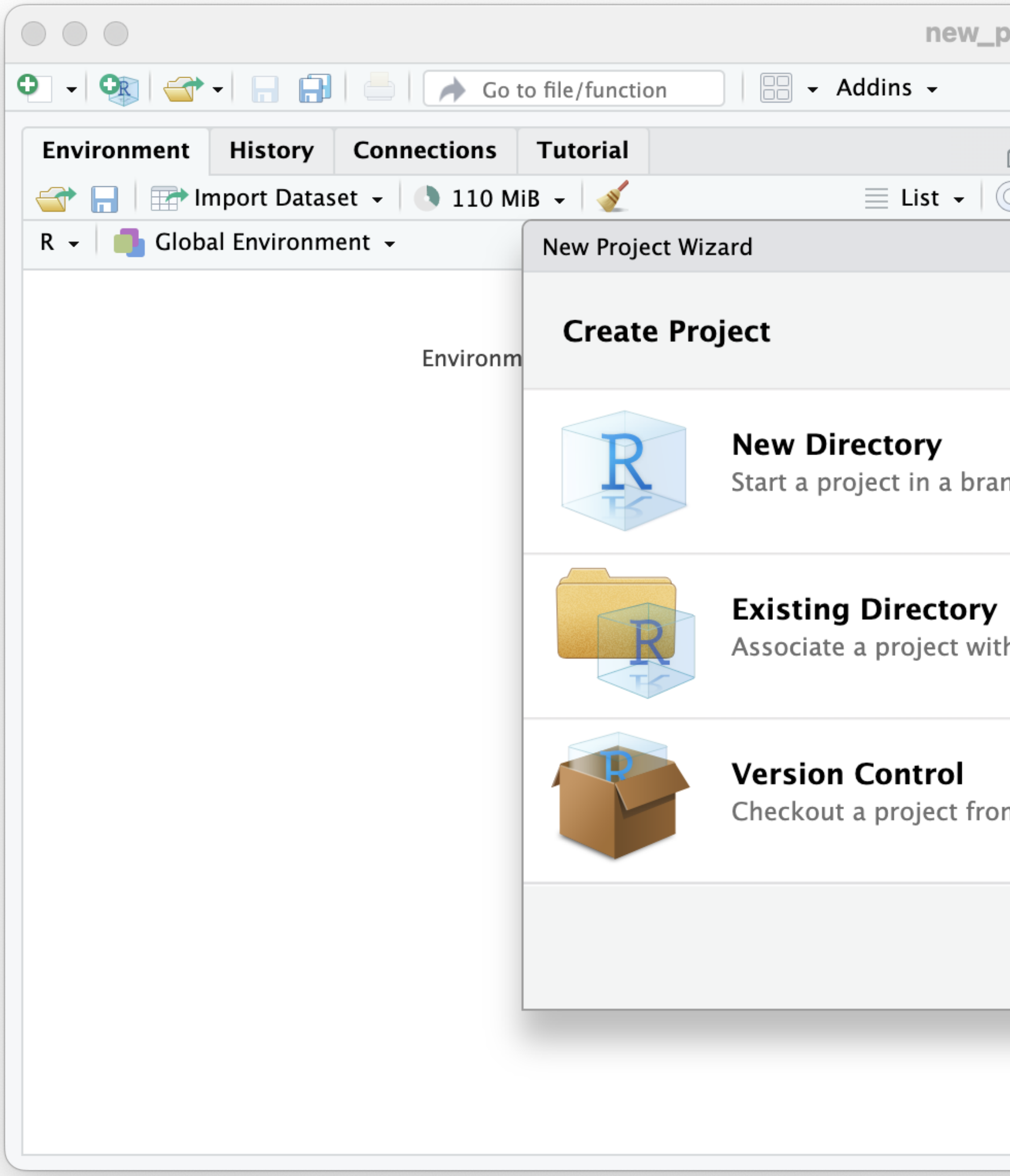
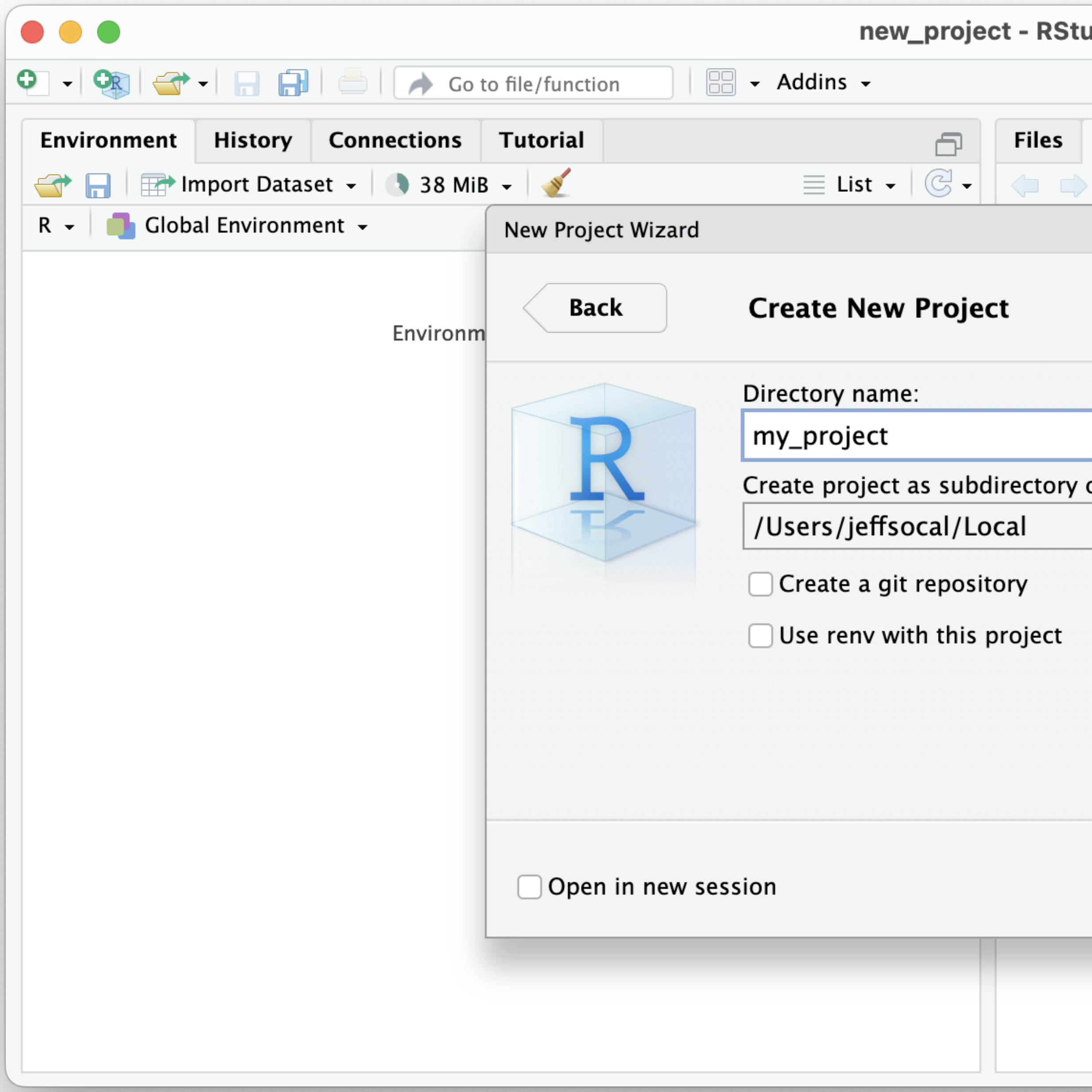


Figure 10: RStudio Create New Project



Directory name:

my_project

Create project as subdirectory c

/Users/jeffsocal/Local

- [ ] Create a git repository
- [ ] Use renv with this project
- [ ] Open in new session

Figure 11: RStudio Create New Project

Data Structures

The R programming environment embodies four basic types of data structures that increase in complexity from `variable`, `vector`, `matrix` and `list`. The `data.frame` structure, while independent is derived from the `matrix`. This book introduced variables only briefly in @ref(syntax), here we will expand on those introductions. In most simplistic sense a variable can be thought of as a container, holding only a single thing like a single stick of gum, the vector is an ordered, finite collection of variables, like a pack of gum, and the matrix are columns of equal sized vectors, like a vending machine for several gum pack flavors. Mentally, consider them a point, a line and a square, respectively.

Variable

Again, a variable is the most basic information container, capable of holding only a single *numeric* or *string* value.

```
a <- 1
```

Vector

A vector is simply a collection of variables of all the same type. In other programming languages these are called arrays, and can be more permissive allowing for different types of values to be stored together. In R this is not permitted, as vectors can only contain either numbers or strings. If a vector contains a single string value, this "spoils" the numbers in the vector, thus making them all strings.

```
# permitted
a <- c(1, 2, 3)
a
```

```
## [1] 1 2 3
```

```
# the numerical values of 1 and 3 are lost, and now only represented as strings
b <- c(1, 'two', 3)
```

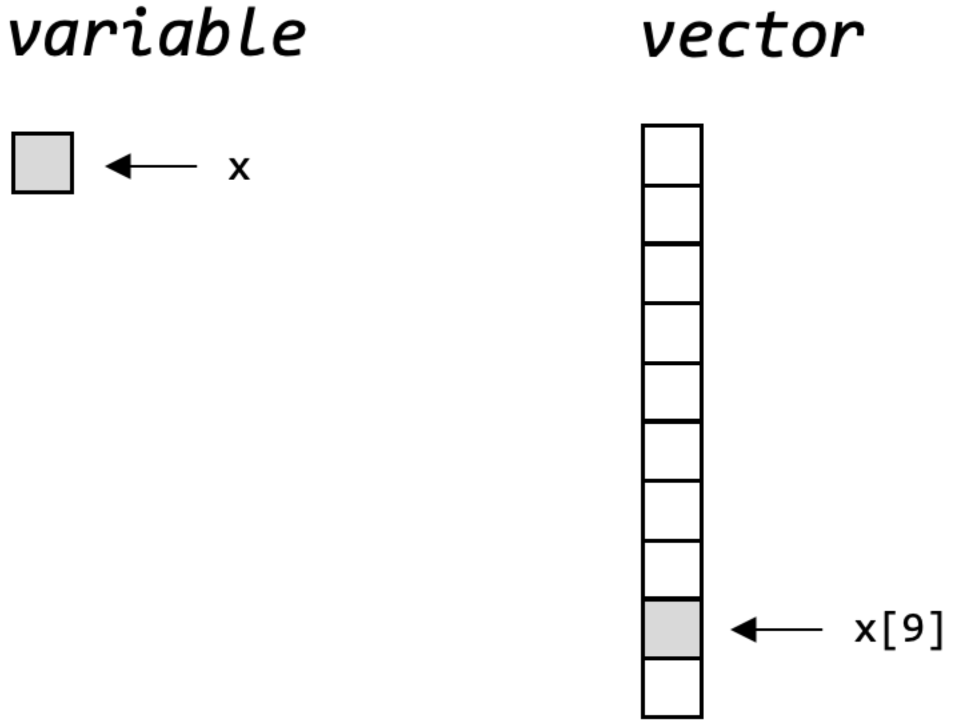


Figure 12: R main data structures

```
b
```

```
## [1] "1"   "two" "3"
```

Vectors can be composed through various methods, either by concatenation with the `c()` function, as seen above, or using the range operator `:`. Note that the concatenation method allows for the non-sequential construction of variables, while the range operator constructs a vector of all sequential intergers between the two values.

```
1:3
```

```
## [1] 1 2 3
```

There are also a handful of pre-populated vectors and functions for constructing patters.

```
# all upper case letters
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
# all lower case letters
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
# a repetitive vector of the letter "a"
rep('a', 5)
```

```
## [1] "a" "a" "a" "a" "a"
```

```
# a repetitive vector of a previous vector
rep(b, 2)
```

```
## [1] "1"   "two" "3"   "1"   "two" "3"
```

```
# a sequence of integers between two values, in this case reverse order
seq(10, 5)
```

```
## [1] 10  9  8  7  6  5
```

```
# same as above
10:5
```

```
## [1] 10  9  8  7  6  5
```

While variables don't require a referencing scheme, because they only contain a single value, vectors need to have some kind of referencing sheme, shown in @ref(4001) as `x[9]` and illistrated in the following example.

> **NOTE:** the use of an integer vector to sub-select another vector based on position. **NOTE:** R abides by the 1:N positional referencing, where as other programming languages refer to the first vector or array position as 0. A good topic for a lively discussion with a computer scientist.

```
x <- LETTERS
# 3rd letter in the alphabet
x[3]
```

```
## [1] "C"
```

```
# the 9th, 10th, 11th and 12th letters in the alphabet
x[9:12]
```

```
## [1] "I" "J" "K" "L"
```

```
# the 1st, 5th, 10th letters in the alphabet
x[c(1,5,10)]
```

```
## [1] "A" "E" "J"
```

Numerical vectors can be operated on simultaneously, using the same conventions as variables, imparting convenient utlity to calculating on collections of values.

```
x <- 1:10
x / 10
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

In addition, there are facile ways to extract information using a coonditional statement ...

```
x <- 1:10 / 10
x < .5
```

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

... the `which()` function returns the integer reference positions for the condition `x < 0.5` ...

```
which(x < .5)
```

```
## [1] 1 2 3 4
```

... and since the output of that function is a vector, we can use it to reference the original vector to extract the elements in the vector that satisfy our condition `x < 0.5`.

```
x[which(x < .5)]
```

```
## [1] 0.1 0.2 0.3 0.4
```

Matrix

Building upon the vector, a matrix is simply composed of columns of numeric vectors. That statement is not completely accurate as matrices can be row based, however, if we mentally orient ourselves to column based organizations, then the following `data.frame` will make sense. Matrices are constructed using a function as shown in the following example.

```
# taking the vector 1:4 and distributing it by 2 rows and 2 columns
m <- matrix(1:4,2,2)
```

Elements within the matrix have a reference schema similar to vectors, with the first integer in the square brackets is the row and the second the column `[row,col]`. ## Data Frame

List

Data Tables

Tibble, a new Data Structure

–> (08 break, assignment) ## Reading Data into R ## Summarization –> (09 break, assignment) ## Data Manipulation ### dplyr ### tidyr –> (10 break, assignment)

Data Wrangling

Tidy Data

Data Visualization

GGPlot

Syntax

Points and Lines

–> (11 break, assignment) ### Bar and histograms –> (12 break, assignment) ## Extended Syntax ### Colors, Scales and Faceting ### Labels and Annotations –> (13 break, assignment) #### Install Package ggrepel –> (14 break, assignment)

Mass Spectrometry Data

Commercial

RAW (Thermo)

WIFF (Sciex)

D (Agilent)

D (Bruker)

Open Access Raw Data

mzXML (HUPO)

mzML (HUPO - most common)

mzH5 (HDF5)

Derived|Processed|Translated Data Outputs

Common Utilities (Proteomics)

PD, OpenMS, Skyline, TransPP, MSFragger, MaxQuant Comet, Mascot, XTandem

mzIdent, mzQuant, mzPeptides

MGF (Mascot, Comet)

PIN (Comet)

CSV (Search Engines)

Mass Spectrometry R Packages (work in progress)

xcms

MSstats

MALDIquant

tidyproteomics