Introduction to R for Mass Spectrometrists

Jeff Jones, Heath Patterson, Ryan Benz

2023-04-26

Contents

Preface

Who are the authors:

- Jeff Jones (PhD), Senior Scientist Proteomics, Division of Physics, Mathematics and Astronomy, Caltech, California USA
- Heath Patterson (PhD), Director of Spatial Biology Bioinformatics, Aspect Analytics, Genk, Belgium
- Ryan Benz (PhD), Director Data Science, Seer Bio, Redwood City, California USA

Who is the audience? This book is aimed at absolute beginners in R and programming in general. The topics covered are designed to be straightforward and easy to follow, and by the end of the book, readers should be able to develop analytical processes for their own research, provide means for others to accomplish analyses, and extend their skills with more advanced literature.

Although no previous knowledge of R is required, some experience with data and statistical analysis is recommended. Reading and understanding the chapters and exercises should provide the skills necessary for basic data analysis and prepare readers for more advanced concepts and skills.

What Is Covered? The purpose of this book is to provide you with a comprehensive guide to the R programming language, as well as to teach you how to use RStudio, tidy data, the tidyverse ecosystem, and ggplot2. By the end of this book, you will have a solid foundation in R and the skills necessary to conduct data analysis and visualization.

Our goals for this book are as follows:

- Learn the basic fundamentals of the R programming language, including variables, data types, functions, and control structures. You will also learn how to write and execute basic R scripts.
- Learn how to use the RStudio integrated development environment (IDE), including how to navigate the interface, create projects, and install packages.

- Learn about tidy data: what it is, why it's important for data analysis, and how to use the tidyr package to transform data into a tidy format.
- Learn the basic fundamentals of the tidyverse ecosystem of R packages, including dplyr, tidyr, and ggplot2, and how they can be used to streamline the data analysis process.
- Learn how to create data visualizations using the ggplot2 R package, including how to customize plots, add themes and colors, and create complex visualizations.

By the end of this course, you should be able to:

- Start up RStudio and create an RStudio project.
- Read a formatted text file (e.g., CSV file) into R.
- Understand basic properties of the data, such as the number of rows and columns.
- Explain what tidy data is and why it's important.
- Perform basic data manipulations and operations on the data.
- Create a simple plot based on the data.

What Is Not Covered? This book is designed to provide you with a comprehensive introduction to R programming. While we cover a variety of essential topics, we don't cover everything. For instance, we do not go into depth on statistical analysis, probability, regression, machine learning, or any other advanced analytical topics. We also do not cover constructing R packages, documentation, markdown, or any other advanced R programming topic.

Why have another book on R? This book is the foundation of the Introduction to R course offered at the annual conference for the American Society for Mass Spectrometry (ASMS). It was created by the authors as a more permanent, expandable, and revisable reference document. The book evolved from a 200+ slide presentation used as instructional material in the "Getting Started with R" two-day short course. It became clear that the amount of material covered in such a compressed time was both limiting in the topics discussed and in students' retention of verbal instructions. Therefore, this tome was created to provide a greater depth of coverage on various topics and to record the instructors' nuanced approach to R. This is intended to be a living document, with improvements made to the explanations and topics covered based on comments and suggestions from each instructional iteration, as well as any external feedback, which is greatly appreciated.

When was this book developed? The initiative that resulted in the creation of the R Book started in 2017, when a group of experts in the field came together to offer a series of workshops at the annual ASMS conference. These workshops were aimed at teaching attendees how to effectively use R, a programming language used in statistical computing and graphics. Year after year, the workshops drew a significant number of attendees, with between 200 and 300 people participating for three consecutive years.

Building on the success of the workshops, the presenters decided to offer a more formal short course in 2020, which was a remote year due to the COVID-19 pandemic. The course proved to be very popular and was well-received by participants.

Following the positive response to the short course, in 2023, it was decided to convert all of the teaching materials and resources used in the workshops and course into a formal book. This book, the R Book, was created with the aim of providing a comprehensive and accessible guide to using R, and has since become a valuable resource for students, researchers, and professionals alike.

Where can the book be accessed? The book is available online here [link]. Additionally, 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.

Acknowledgements

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

Chapter 1

Introduction to R

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

skill

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?

In recent years, R has gained a lot of popularity among data scientists and analysts. The reason for this is simple: R is a language that is specifically designed for working with data. While other programming languages like C/C++, Java, and Python are general purpose languages that can be used in any domain, R is geared towards data analysis and manipulation.

Because R is designed for working with data, it has several features that make it easier to work with large datasets. For instance, R has several built-in data structures that allow users to organize and manipulate data in a variety of ways. Additionally, R has a wide range of libraries and packages that can be used to perform specific tasks like data visualization, statistical analysis, and machine learning.

Another reason why R is so popular among data scientists is that it is an open-source language. This means that anyone can contribute to its development, and there is a vast community of users and developers working together to improve the language and its capabilities.

Despite its many advantages, R does have a few limitations. For example, it is not as fast as some other programming languages, and it can be difficult for beginners to learn. However, there are many resources available online to help users learn R, and once they get the hang of it, they will find that it is a powerful tool for data analysis and visualization.

Overall, R is an excellent language for anyone who wants to work with data. Its specialized features and wide range of capabilities make it a top choice for data scientists and analysts everywhere.

What you can do with it?

The potential of what you can achieve with R is vast and ultimately depends on the level of dedication you have towards learning and expanding your skill set. By utilizing R, you can analyze data through various methods such as reading and plotting data, constructing analysis pipelines, prototyping new algorithms, and even writing your analysis code into shareable packages. With these abilities, you can not only perform data analysis, but also create a more efficient and reproducible workflow. The more you learn and experiment with R, the more you can discover and unlock its full potential.

note ***NOTES*** *Some helpful explanatory notes and tips appear as a block quote.*

R can be a fast, nimble, forgiving scripting language with lots of ready-made tools and resources (CRAN, Github, Bioconductor).

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

When it comes to learning a programming language, it can be daunting to know where to start. However, the first step to learning any programming language is to understand its syntax. Syntax refers to the set of rules and symbols that make up structurally correct code. Without proper syntax, even the smallest of errors

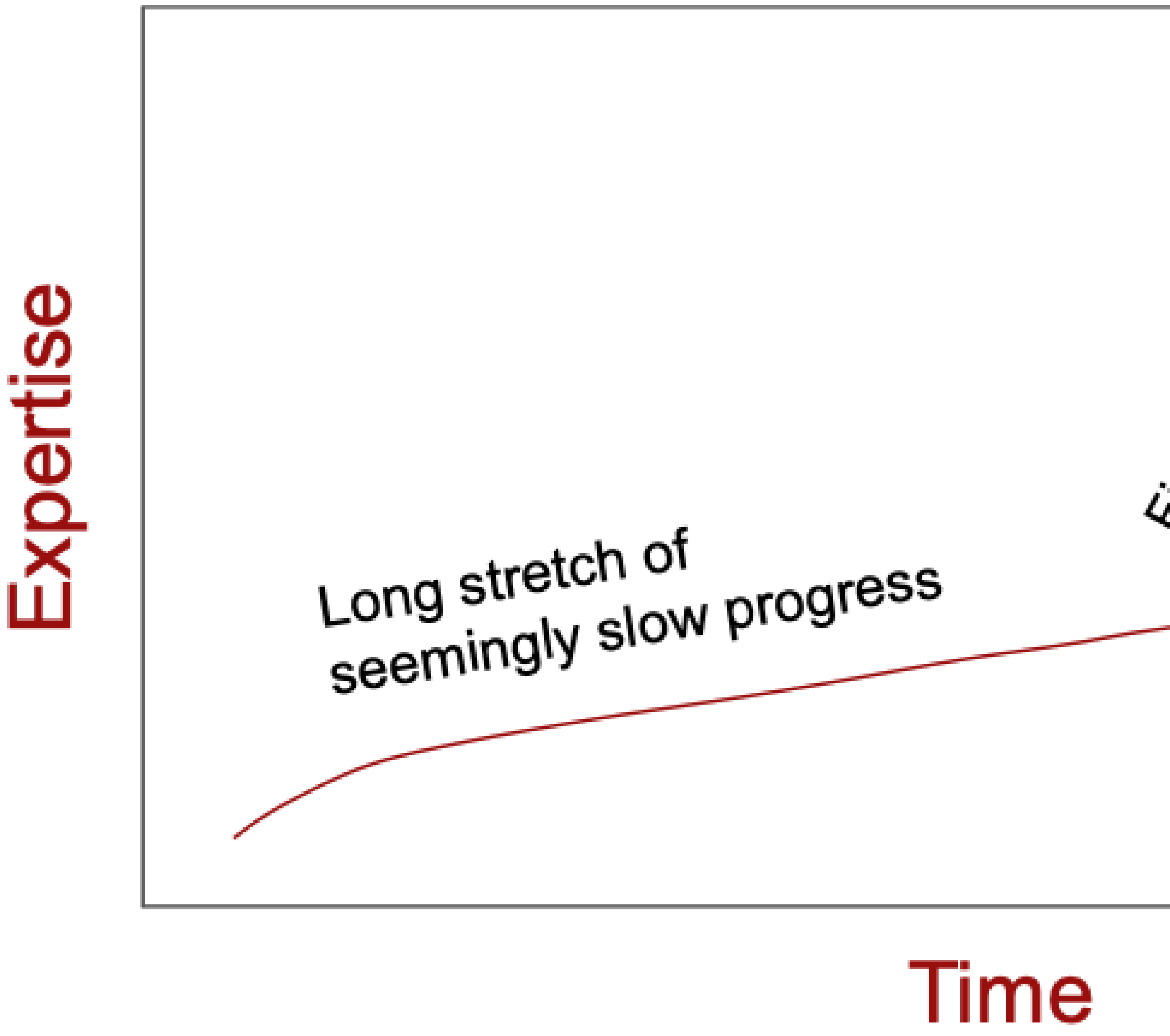


Figure 1.1: R learning curve past

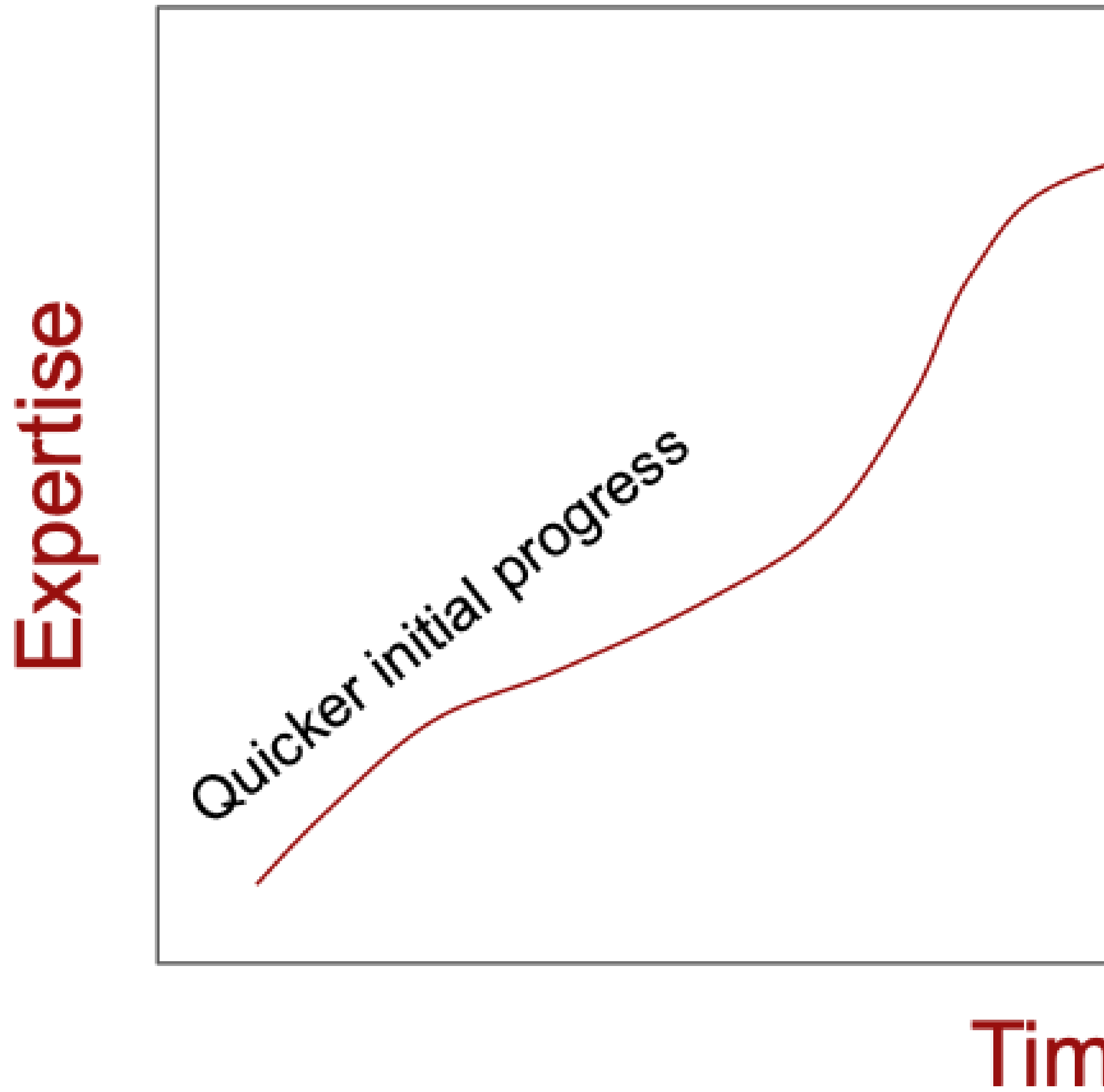


Figure 1.2: R learning curve present

can result in code that doesn't run. These errors could be as simple as a typo, an incorrect name, missing spaces or too many spaces, or even wrong brackets. Syntax errors can be frustrating, especially for beginners, but it's important to hang in there and start simple.

It's best to begin by trying to understand very simple cases first, before building and expanding on them. This approach will help you to get a better grip on the basics of the language and will help you to avoid becoming overwhelmed. If you're learning R, there are many resources available to help you get started. You could start by reading through the R Book, which provides a comprehensive guide to the R programming language. Alternatively, there are many online tutorials available, which can help to break down complex concepts into more manageable pieces.

In short, when learning R, it's important to remember that syntax is key. By taking the time to understand the syntax rules, you can avoid frustrating syntax errors and build a solid foundation for your future coding endeavors

Alternatives

When it comes to data science, R is a popular programming language among statisticians and data analysts. However, there are several data science alternatives to R that are also gaining popularity.

One of the most popular alternatives to R is Python. Python is a general-purpose programming language that has a wide range of libraries and frameworks for data science. It is known for its simplicity, readability, and versatility. Python's libraries such as NumPy, Pandas, and Scikit-Learn are widely used in data science for tasks such as data cleaning, data analysis, and machine learning.

Another alternative to R is Julia, a new programming language that is designed specifically for scientific computing and numerical analysis. Julia is known for its speed and efficiency, making it a great choice for data analysis and modeling. Julia also has a growing package ecosystem with libraries such as DataFrames.jl and Flux.jl that are specifically designed for data science.

Matlab is another alternative to R that is widely used in the scientific community. Matlab is known for its extensive numerical computing capabilities and its strong visualization features. It is commonly used in fields such as engineering, physics, and finance for data analysis and modeling.

In conclusion, while R is a popular language for data science, it is not the only option available. Python, Julia, and Matlab are all viable alternatives with their own strengths and weaknesses. It is important to consider the specific needs of your project and choose the programming language that best suits your requirements.

Did you know

While R on its own is a powerful scripting language, some analytical tasks might require the use of other programming languages such as Python, C++ or Rust. Luckily, R provides different packages that allow us to use these languages within R code. These packages provide a seamless integration between these languages and R, allowing you to leverage the strengths of each language to perform complex tasks.

The `reticulate` package enables the integration of Python code in R. This package allows you to import Python modules and functions directly into R and also allows you to call Python functions from R code. This is especially useful when you need to use Python's machine learning libraries such as TensorFlow or PyTorch, which are not yet available in R.

Similarly, the `Rcpp` package provides a smooth integration between R and C++. With this package, you can easily write C++ functions and use them directly in your R code. This is useful when you need to perform computationally-intensive tasks, such as simulations or optimization, that require the speed of C++.

Finally, the `extendr` package provides an interface between R and Rust, allowing you to use Rust functions in R code and vice versa. Rust is a relatively new programming language that provides a balance between performance and safety. It is especially useful when you need to develop high-performance and low-level code, such as in systems programming or hardware development.

Chapter 2

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.

skill At the end of this chapter you should be able to:
1. Install R, RStudio and a few R packages
2. Understand the major components for working with R.

2.1 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

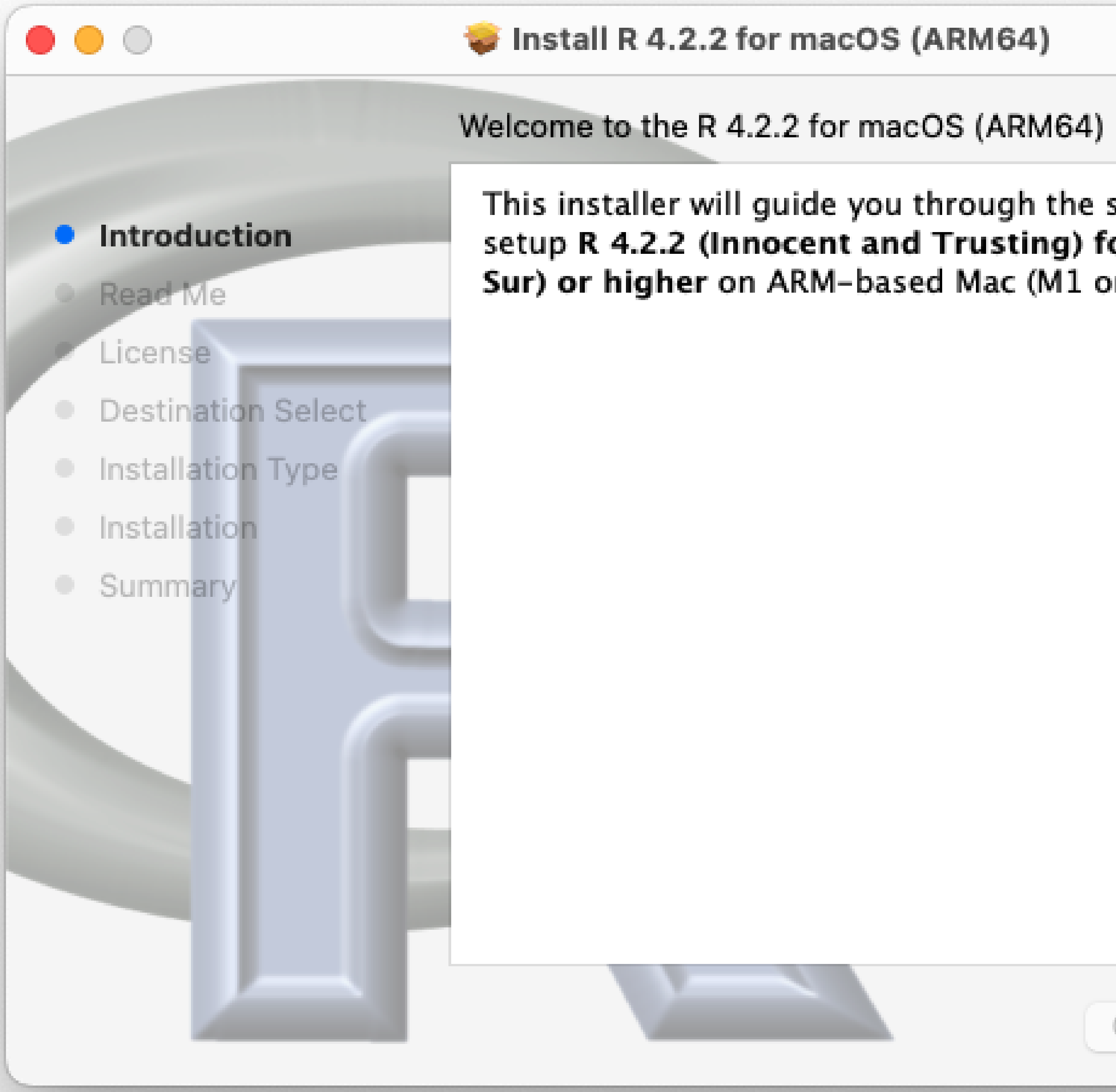
Linux

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.

2.2 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



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 or

Figure 2.1: Mac Installer

> "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)

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

2.2.1 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`

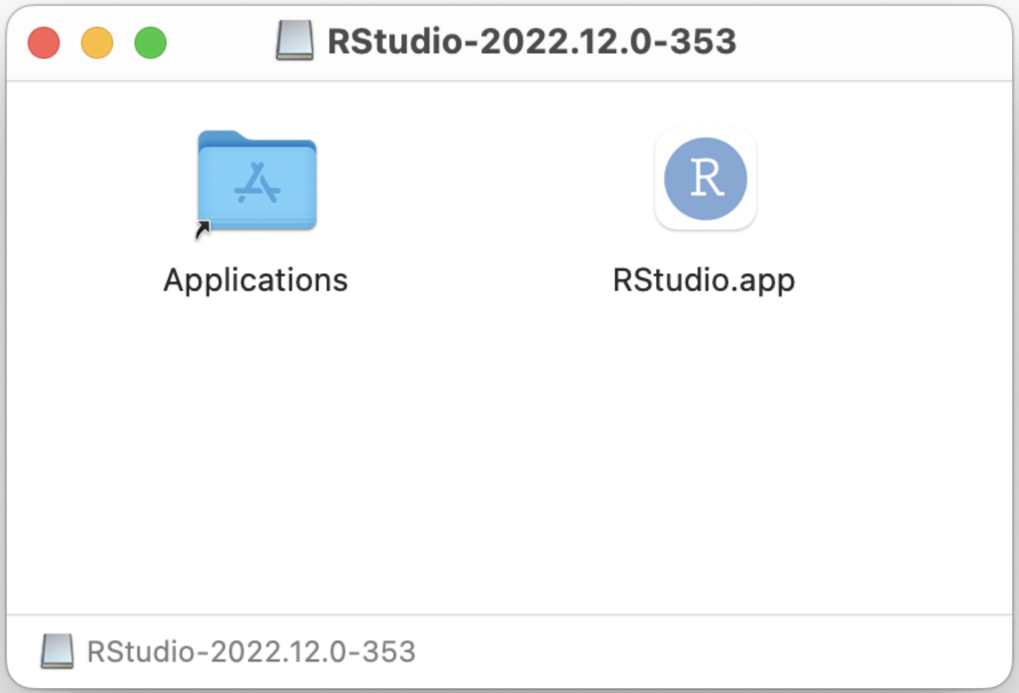
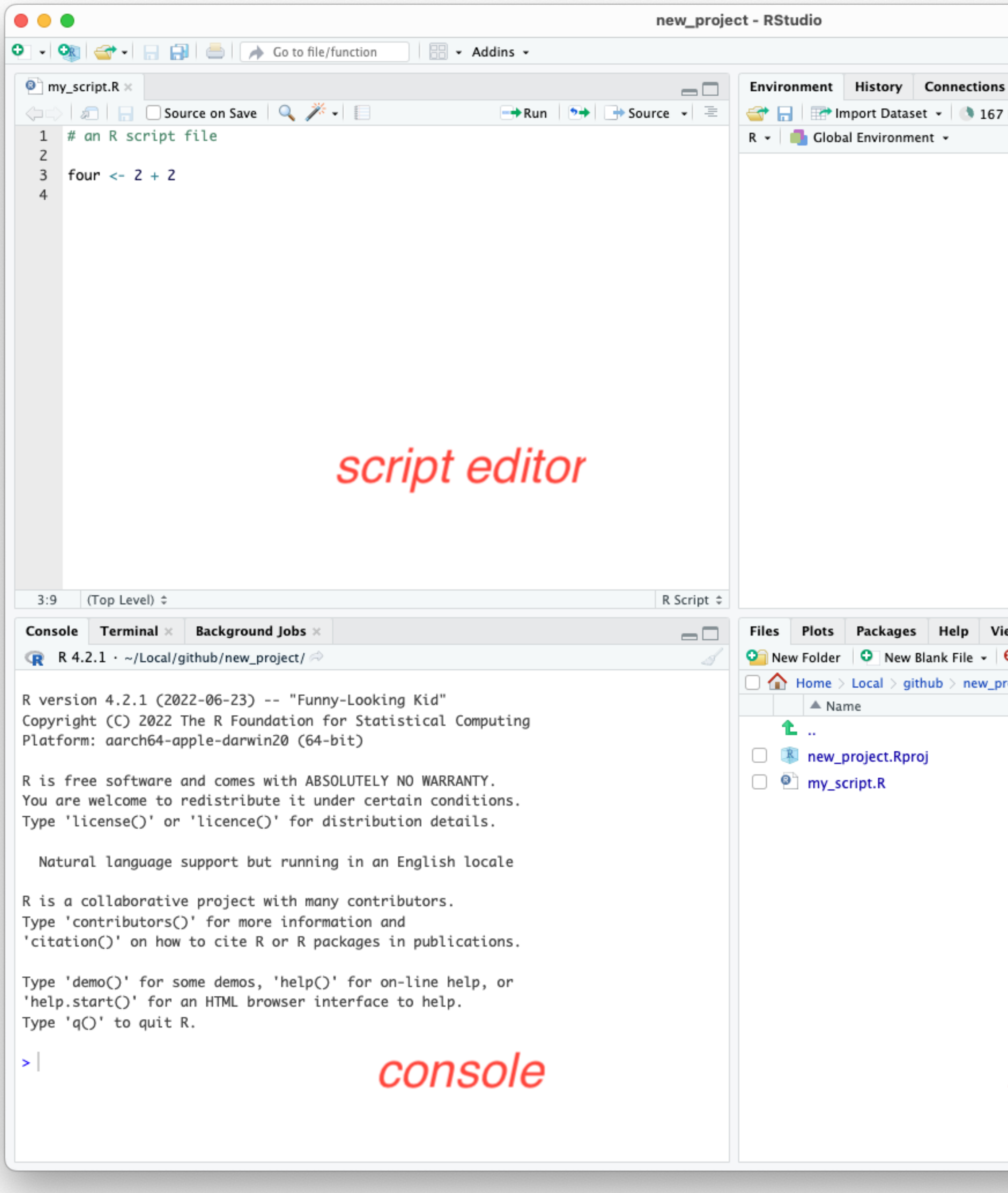


Figure 2.2: RStudio IDE Install



```
# 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 2.3: RStudio IDE in the default layout

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, 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.

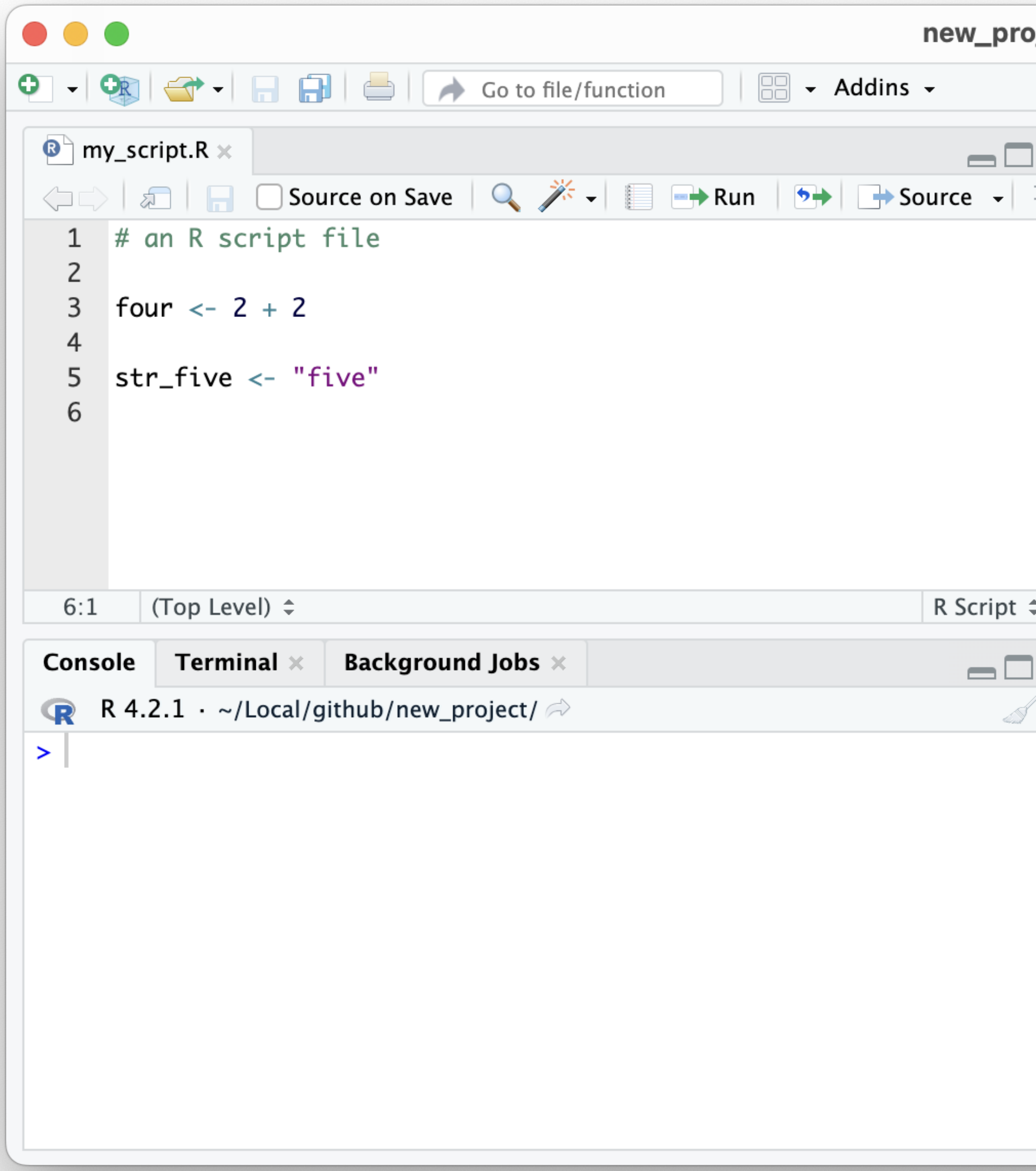
Files and Plots

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.

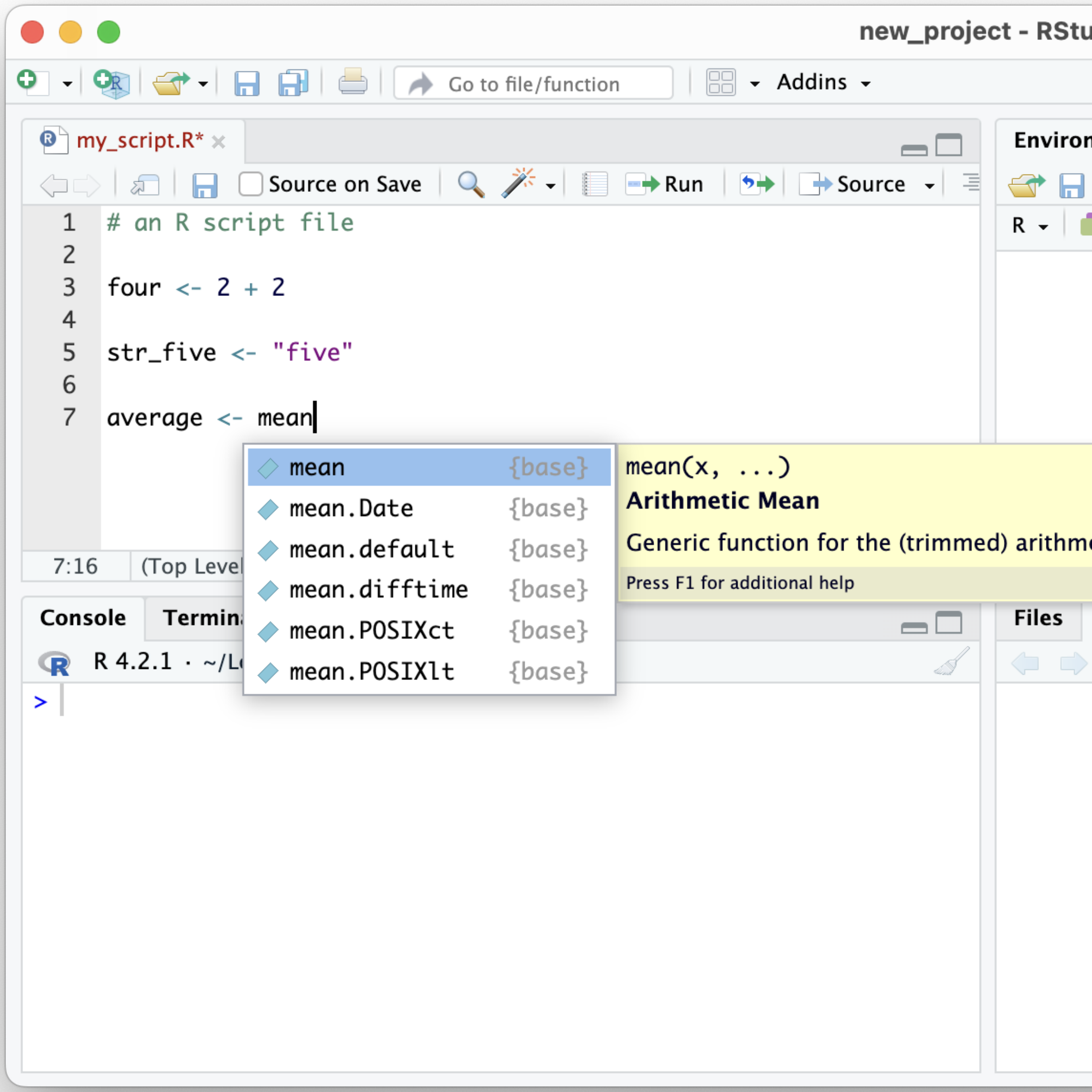
Another useful tab located in the same pane is the *Plots* tab. This tab provides a quick way to view any active plots instantly. You don't need to export your plots or save them separately. Instead, you can view them right within RStudio.



```
# an R script file

four <- 2 + 2

str_five <- "five"

```

```
>
```

Figure 2.4: 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 2.5: RStudio IDE auto complete

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 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.

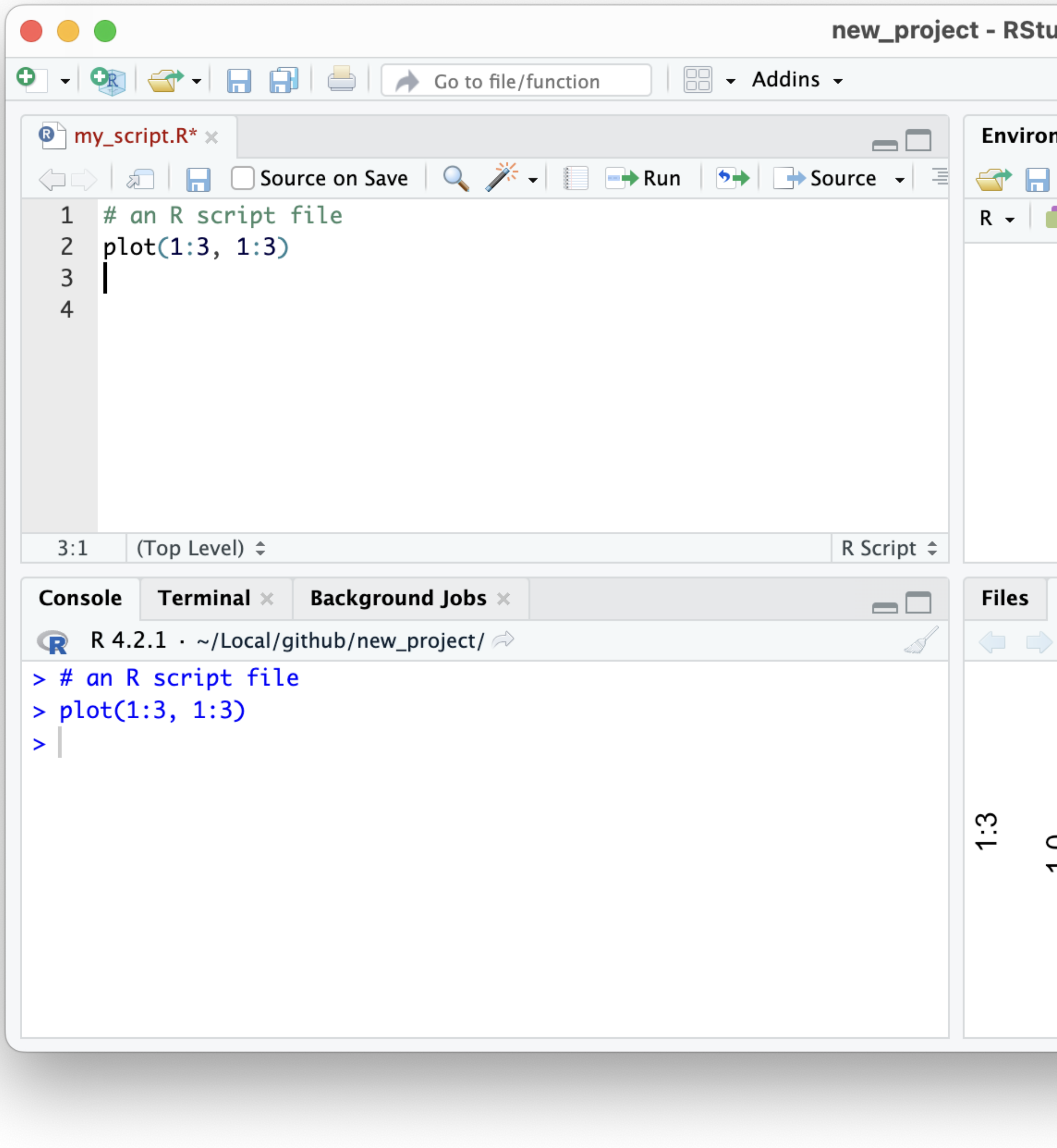
This summary can be useful for new-comers to R because it allows you to quickly see what objects you are currently working with, without having to remember each or manually check. By having a clear overview of your current session, you can avoid mistakes or errors that might arise from using the wrong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

2.2.2 Usage .. Running lines of code in RStudio

2.2.2.1 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 '>'

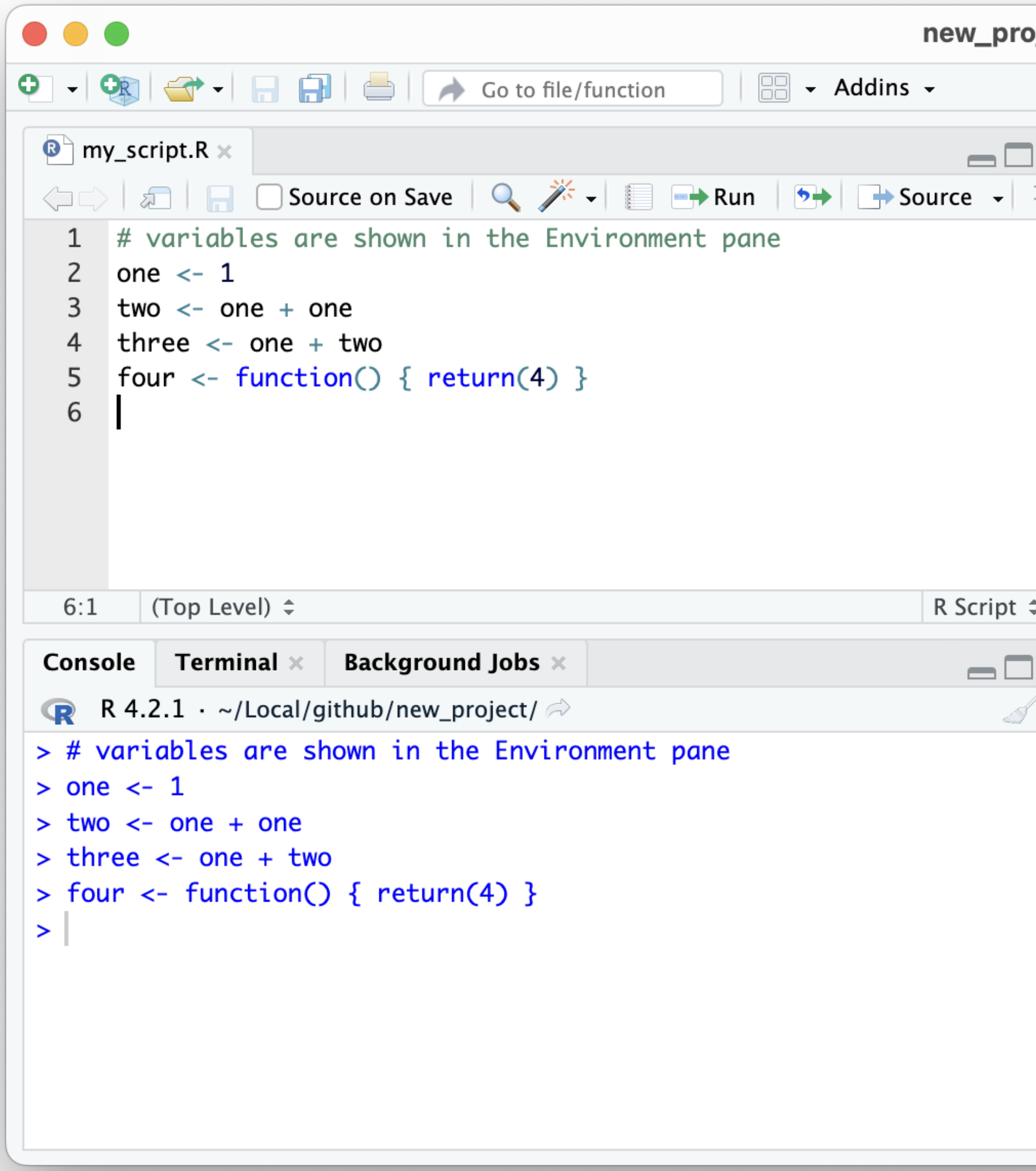


```
# an R script file
plot(1:3, 1:3)
```

R 4.2.1 · ~/Local/github/new_project/

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

Figure 2.6: RStudio IDE plot window



```
# 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 2.7: RStudio IDE environment window

6. Press Enter.
7. Multiple lines, not advised, but copy and paste multiple lines into console then press Enter.

2.3 Packages

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.3.4 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.

2.4 Packages Utilized in This Book

2.4.1 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 pivoting 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")
```

`_main_files/figure-latex/unnamed-chunk-9-1.pdf`

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.

2.4.2 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`.

Chapter 3

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!

skill At the end of this chapter you should be able to:
1. Understand R's syntax, variables, operators and functions
2. Create and edit a project in RStudio

3.1 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.

```
# 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.

3.2 Syntax

Welcome to the R Book! Whether you're just starting out or a seasoned pro, understanding the different components of R code is essential for writing high-quality, efficient R programs. In this section, we'll take a deep dive into the various components of R code that you should be familiar with.

R input is composed of typed characters that represent different parts of a process or mathematical operation. These characters come together to form what we call R code. It's important to note that R code is not just a random collection of characters - each character serves a specific purpose and contributes to the larger structure of the code. As such, understanding the different components of R code is key to writing effective and efficient R programs.

So, what are these different components of R code? Below, we've provided some examples to help you get started:

?? comments	`# this is an important note`
?? strings	`"letters"` or `"numbers"` in quotes
?? numbers	`1` integers or `1.000002` floats
?? operators	`+`, `-`, `/`, `*`, ...
?? variables	`var <- 2` containers for information

?? statements	`==` exactly the same, `!=` not the same
?? functions	`add(x, y)` complex code in a convenient wrapper

By understanding these different components of R code, you'll be well on your way to writing effective and efficient R programs. So let's dive in and get started!

```r
# 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 R does not have an line ending character such as `;` in java, PHP or C++_

3.2.1 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
```

```r
# This is a comment

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

3.2.2 Strings

Strings are essentially a sequence of characters, consisting of letters or numbers. They are commonly used in programming languages and are used to represent text-based data. A string can be as simple as a single character, such as "A", or it can be a longer sequence of characters such as "Hello, World!". Strings are often used to store data that requires text manipulation, such as usernames, passwords, and email addresses. In contrast to words, which are made up of a specific combination of letters to represent a linguistic term, strings do not follow any specific rules of composition and can be a random or semi-random sequence of characters.

```r
# 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"
```

When working with R programming language, it is essential to note that strings play a crucial role in the syntax used. Strings, which define text characters, are used to represent data in R, and they must be enclosed in quotes. Failure to do so will result in the interpreter assuming that you are referring to a variable that is not enclosed in quotes.

For instance, in the example above, the `peptide` variable contains the string of letters representing the peptide amino acid sequence `"QWERTK"`. However, it is essential to note that there are no strict rules for how strings and variables are composed, except that variables **cannot** start with a number.

```r
# 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:

```r
# 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:

- Hadley Wickham's Style Guide
- Google's style Guide
- The tidyverse style guide

3.2.3 Numbers

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.

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

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. In most programming languages, including 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 essential to understand the difference between these two types of numbers for accurate numerical analysis. While integers can only represent whole numbers, floats can represent fractions and decimals. Thus, if you need to represent a number that is not a whole number, you should use a float.

Moreover, it is important to note that integers occupy less space in memory than floats. This can be a consideration when working with large datasets, especially when the whole number is enough to represent the data. Therefore, 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.

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

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

3.2.4 Operators

Operators are fundamental components of programming that enable us to manipulate and process various data types. They are symbols that perform a specific action on one or more operands, which could be numeric values, variables, or even strings. Most commonly these symbols allow us to perform basic arithmetic operations such as addition, subtraction, multiplication, and division on numeric values, as well as more complex mathematical operations like exponentiation and modulus.

In addition to numeric values, operators can also manipulate string variables. For instance, we can use concatenation operators to join two or more strings together, which is particularly useful when working with text data. By utilizing operators, we can perform powerful operations that allow us to build complex programs and applications that can handle large amounts of data. Operators play a crucial role in programming, as they allow us to manipulate data in a way that would be difficult or impossible to achieve otherwise.

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	Name	Description	Example
`<-`	assignmnet	assigns numerics and functions to variables	`x <- 1` x now has the value of 1
`+`	addition	adds two numbers	`1 + 2` = `3`
`-`	subtraction	subtracts two numbers	`1 - 2` = `-1`
`*`	multplication	multiplies two numbers	`1 * 2` = `2`
`/`	division	divides two numbers	`1 / 2` = `0.5`
`^`	power or exponent	raises one number to the power of the other	`1 ^ 2` = `1`
`=`	equals	also an assignment operator	`x = 1` x now has the value of 1

Operator	Name	Description	Example
`==`	double equals	performs a comparison (exactly equal)	`1 == 1` = `TRUE`
`!=`	not equals	performs a negative comparison (not equal)	`1 != 2` = `TRUE`
`%%`	modulus	provides the remainder after division	`5 %% 2` = `1`

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

3.2.5 Variables

In programming, variables are essential elements used to store information that can in essence **vary**. They come in handy when we need to manipulate or retrieve the information stored in them.

Variables can be thought of as containers that can store any kind of information, such as letters, words, numbers, or text strings. They are flexible enough to hold different types of data, and we can use them to store all sorts of information.

One of the most significant advantages of using variables is that we can refer to them repeatedly to retrieve the information stored in them. We can also manipulate the information stored in them with an operation or replace it with an assignment. Variables are a powerful tool in programming that allows us to store and retrieve information, manipulate it, and perform various operations on it.

```
# 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
```

3.2.6 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.

3.2.7 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.

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

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

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

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

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

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

```r
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 . . .

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

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

. . . or just a normal function.

```r
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()`.

3.3 Flow-Control

3.3.1 If-Else Statements

3.3.2 Loops

3.3.2.1 For

3.3.2.2 While

3.4 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.

3.4.1 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`.

3.4.2 Editing

3.4.3 Organizing

3.4.3.1 Data

3.4.3.2 Scripts

3.4.3.3 Results

3.4.3.3.1 Tables

3.4.3.3.2 Plots

Exercises

1. Calculate the sum of 2 and 3.

```
## [1] 5
```

2. Evaluate if 0.5 is equal to 1 divided by 2.

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

3. Test if 3 is an even number. Hint, use the modulus operator and a comparison operator.

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

4. Create a function to test if a value is even resulting in `TRUE` or `FALSE`.

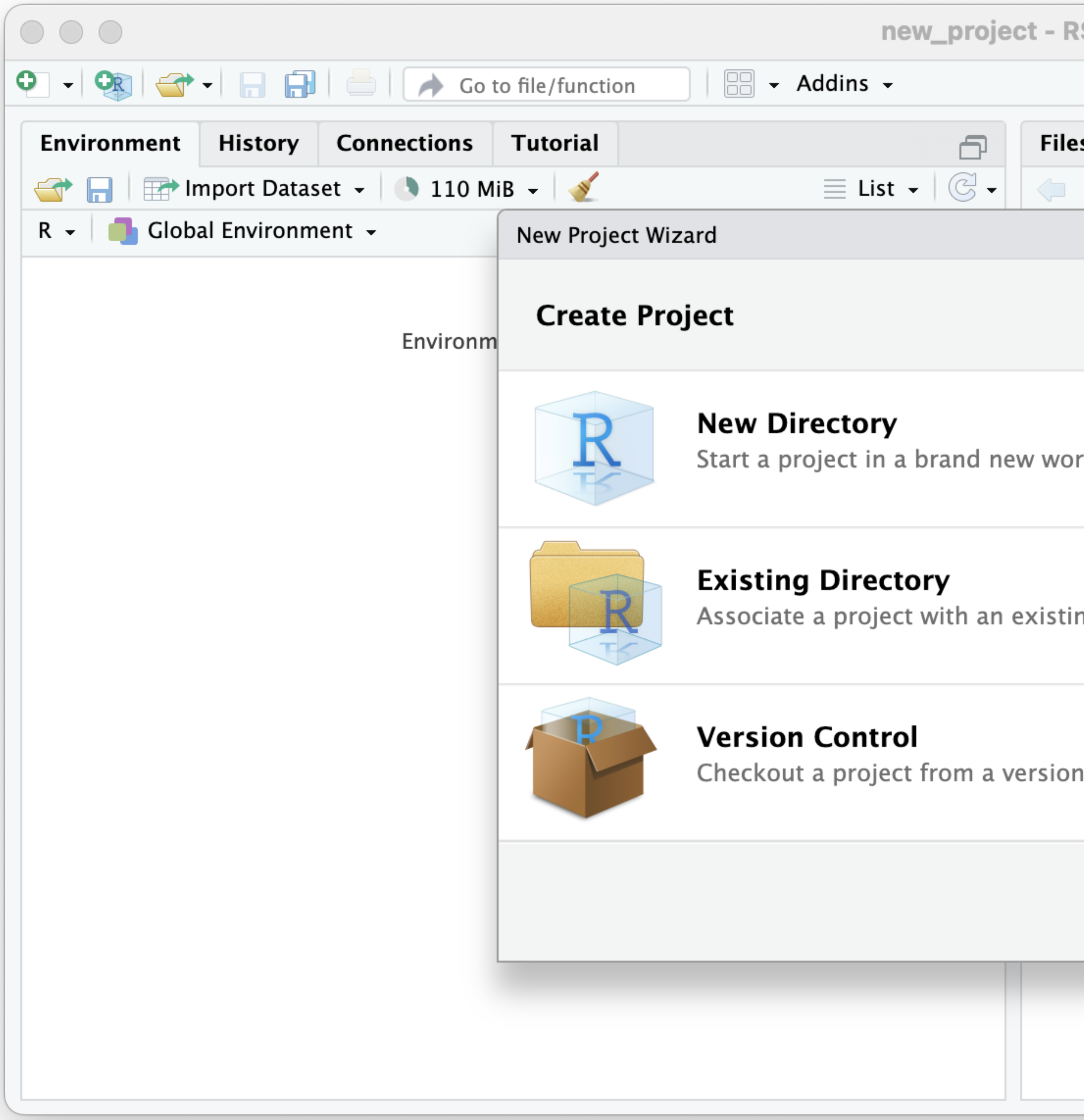
```
even(3)
```

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

4. Create a function to test or *even* or *odd* by returning a string.

```
oddeven(3)
```

```
## [1] "odd"
```



New Project Wizard

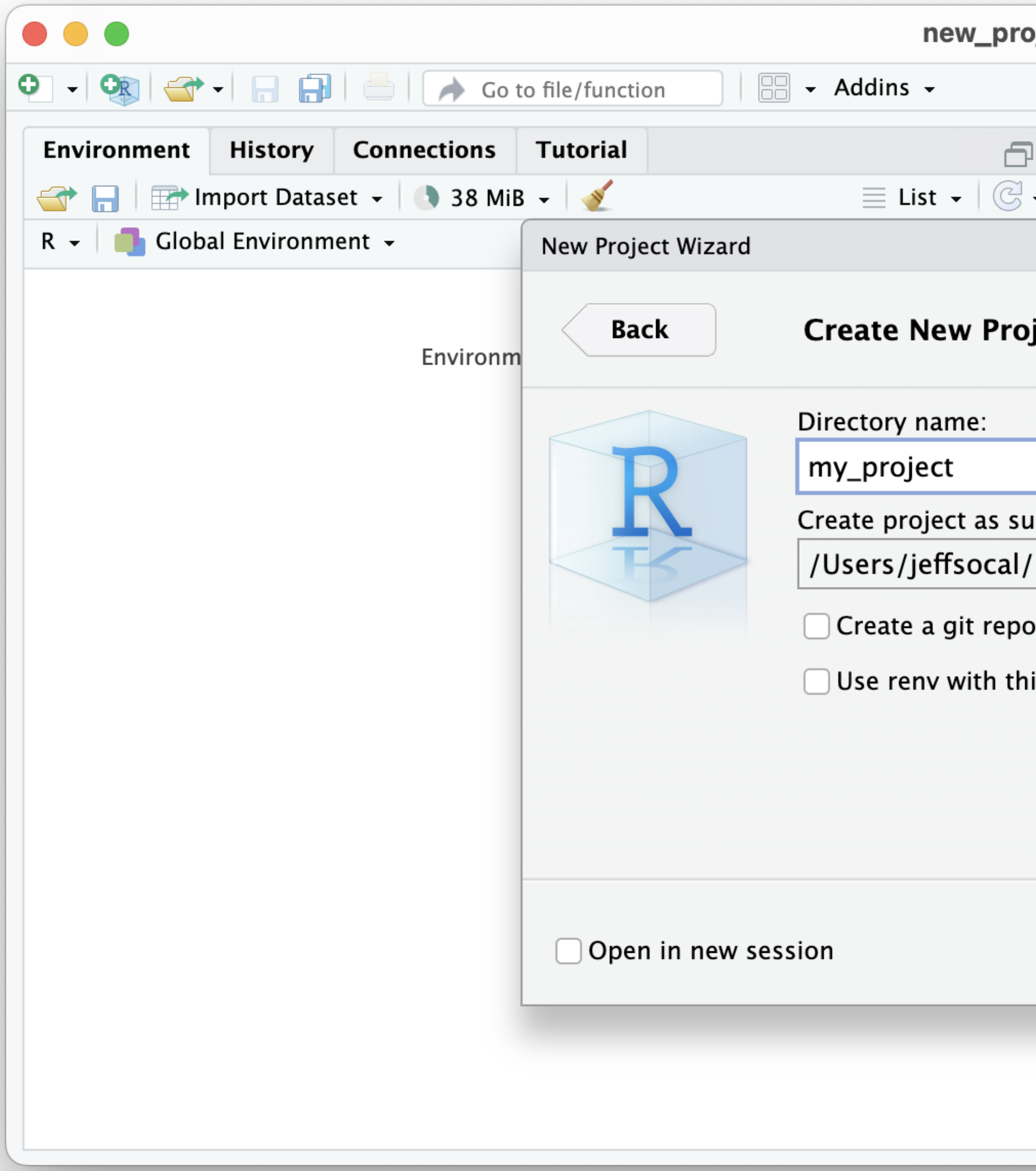
Create Project

New Directory
Start a project in a brand new wor

Existing Directory
Associate a project with an existin

Version Control
Checkout a project from a version

Figure 3.1: RStudio Create New Project



Directory name:
my_project
Create project as su
/Users/jeffsocal/
Create a git repo
Use renv with thi
Open in new session

Figure 3.2: RStudio Create New Project

Chapter 4

Data Structures

The R programming environment includes four basic types of data structures that increase in complexity: `variable`, `vector`, `matrix`, and `list`. Additionally there is the `data.frame` while and independent data structure, it is essentially derived from the `matrix`.

skill At the end of this chapter you should be able to:
1. Understand the 5 most common data structures
2. Understand the data structure lineage
3. Access given subsets of a multi-variable data object

This book introduced variables briefly in **??**. Here, we will expand on that introduction. At its simplest, a variable can be thought of as a container that holds only a single thing, like a single stick of gum. A vector is an ordered, finite collection of variables, like a pack of gum. A matrix consists of columns of equally-sized vectors, similar to a vending machine for several flavors of gum packs. Mentally, you can think of them as a point, a line, and a square, respectively.

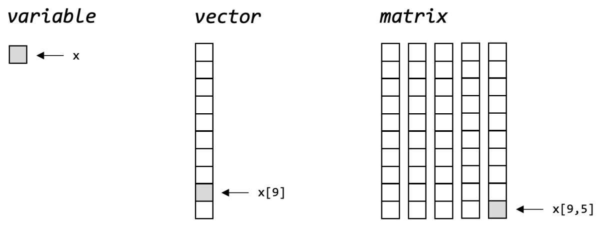


Figure 4.1: R main data structures

4.1 Variable

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

```
a <- 1
```

4.2 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)
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 integers 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" "R" "S" "T
```

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

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t
```

```
# 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 scheme, shown in **??** as `x[9]` and illustrated in the following example.

note Note the use of an integer vector to sub-select another vector based on position. 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.

```r
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 …

```r
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` …

```r
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`.

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

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

4.3 Matrix

Building upon the vector, a matrix is simply composed of columns of either all numeric or string 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.

```r
# 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]`.

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

Just like a vector, a matrix can be used to compute operations on all elements simultaneously, apply a comparison and extract the variable(s) matching the condition …

```
m_half <- m / 2
w_point5 <- which(m_half > 1)
m[w_point5]
```

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

. . . or more sincintly.

```
m[which(m/2 > 1)]
```

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

4.4 Data Frame

Tables are one of the fundamental data structures encountered in data analysis, and what separates them from matrices is the mixed use of numerics and strings, and the orientation that `data.frames` are columns of vectors, with a row association. A table can be cinstructed with the `data.frame()` function as shown in the example.

```
df <- data.frame(
  let = LETTERS,
  pos = 1:length(LETTERS)
)
```

```
##   let pos
## 1   A   1
## 2   B   2
## 3   C   3
## 4   D   4
## 5   E   5
## ...
```

Lets talk about the structure of what just happened in constructing the `data.frame` table. Note that we defined the column with *let* and *pos* referring to letter and position, respectively. Second, note the use of the single `=` to assign a vector to that column rather than the "out-of-function" assignment operator `<-` – meaning that functions use the `=` assignment operator, while data structures use the `<-` assignment operator.

The printed output of the `data.frame` shows the two column headers and also prints out the row names, in this case the integer value. Now, that this table is organized by column with row assiciations, we can perform an evalutaion on one column and reterive the value(s) in the other.

4.5 List

In R programming, a 'list' is a powerful and flexible collection of objects of different types. It can contain vectors, matrices, data frames, and even other lists, making it an extremely versatile tool in data analysis, modeling, and visualization.

With its ability to store multiple data types, a list can be used to represent complex structures such as a database table, where each column can be a vector or a matrix. Furthermore, a list can be used to store multiple models for model comparison, or to store a set of parameters for a simulation study.

In addition to its flexibility, a list is also efficient, as it allows for fast and easy data retrieval. It can be used to store large datasets, and its hierarchical structure makes it easy to navigate and manipulate.

Here's an example of how to create a list in R:

```r
# create a list
my_list <- list(name = "Janie R Programmer",
                age = 32,
                salary = 100000,
                interests = c("coding", "reading", "traveling"))

print(my_list)
```

```
## $name
## [1] "Janie R Programmer"
##
## $age
## [1] 32
##
## $salary
## [1] 1e+05
##
## $interests
## [1] "coding"    "reading"   "traveling"
```

In the above code, we have created a list 'my_list' with four elements, each having a different data type. The first element 'name' is a character vector, the second element 'age' is a numeric value, the third element 'salary' is also a numeric value, and the fourth element 'interests' is a character vector.

We can access the elements of a list using the dollar sign '$' or double brackets '[[]]'. For example:

```r
# accessing elements of a list
print(my_list$name)
```

```
## [1] "Janie R Programmer"
```

```
print(my_list[["salary"]])
```

```
## [1] 1e+05
```

Lists are also useful for storing and manipulating complex data structures such as data frames and tibbles.

4.6 How to tell what you are dealing with

You can use the `str()` function to peak inside any data object to see how it is structured.

The contents of a data.frame:

```
plant_data <- data.frame(
  age_days = c(10, 20, 30, 40, 50, 60),
  height_inch = c(1.02, 1.10, 5.10, 6.00, 6.50, 6.90)
)

str(plant_data)
```

```
## 'data.frame':    6 obs. of  2 variables:
##  $ age_days   : num  10 20 30 40 50 60
##  $ height_inch: num  1.02 1.1 5.1 6 6.5 6.9
```

The contents of a tible as very similar:

```
plant_data <- tibble(
  age_days = c(10, 20, 30, 40, 50, 60),
  height_inch = c(1.02, 1.10, 5.10, 6.00, 6.50, 6.90)
)

str(plant_data)
```

```
## tibble [6 x 2] (S3: tbl_df/tbl/data.frame)
##  $ age_days   : num [1:6] 10 20 30 40 50 60
##  $ height_inch: num [1:6] 1.02 1.1 5.1 6 6.5 6.9
```

The contents of a linear regression data object are quite different:

```
# linear prediction of salary based on age
linear_model <- lm(data = plant_data, height_inch ~ age_days)

linear_model
```

```
##
## Call:
## lm(formula = height_inch ~ age_days, data = plant_data)
##
```

```
## Coefficients:
## (Intercept)     age_days
##     -0.2133       0.1329
```

```
str(linear_model)
```

```
List of 12
 $ coefficients : Named num [1:2] -0.213 0.133
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "age_days"
 $ residuals    : Named num [1:6] -0.0952 -1.3438 1.3276 0.899 0.0705 ...
  ..- attr(*, "names")= chr [1:6] "1" "2" "3" "4" ...
 $ effects      : Named num [1:6] -10.868 5.558 1.296 0.602 -0.492 ...
  ..- attr(*, "names")= chr [1:6] "(Intercept)" "age_days" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:6] 1.12 2.44 3.77 5.1 6.43 ...
  ..- attr(*, "names")= chr [1:6] "1" "2" "3" "4" ...
```

Exercises

1. Construct the following vector and store as a variable.

```
## [1] "red"   "green" "blue"
```

2. Extract the 2nd element in the variable.

```
## [1] "green"
```

3. Construct a numerical vector of length 5, containing the diameters of circles with integer circumferences 1 to 5. Remember PEMDAS.
4. Extract all circumferences greater than 50.

```
## [1] 50.26549 78.53982
```

Chapter 5

Data Tables

5.1 Tibble, a new Data Structure

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

Chapter 6

Data Wrangling

6.1 Tidy Data

Chapter 7

Data Visualization

Visualizing your data is crucial because it helps you understand the patterns, trends, and relationships within the data. A well-designed visualization can make complex data easy to understand and convey insights that would be hard to discern from raw data.

skill At the end of this chapter you should be able to:
1. Understand the need for visualization
2. Create some simple plots of points, lines and bars

Anscombe's quartet is a classic example that demonstrates the importance of visualizing your data. This quartet comprises four datasets with nearly identical simple descriptive statistics. However, when graphed, they have very different distributions and appear very different from one another. This example shows that relying solely on summary statistics to understand data can be misleading and inadequate.

In data analysis, creating a plot to convey a message or demonstrate a result is a common endpoint. To achieve this, this book utilizes the `GGPlot2` package, which is part of the `Tidyverse`. This package complements the data pipelining

Table 7.1: Summary stats for Anscombe's quartet.

set	mean_x	var_x	mean_y	var_y	intercept	slope	r.squared
A	9	11	7.500909	4.127269	3.000091	0.5000909	0.6665425
B	9	11	7.500909	4.127629	3.000909	0.5000000	0.6662420
C	9	11	7.500000	4.122620	3.002454	0.4997273	0.6663240
D	9	11	7.500000	4.126740	3.000000	0.5000000	0.6663856

demonstrated in the previous chapters, making it a perfect choice for creating a wide range of plots, from simple scatter plots to complex heat maps, making it ideal for data visualization.

7.1 Base: plot

R comes standard with the fairly basic plotting function `plot()`. While this function forms the basis for all plotting interactions in R, it can be greatly extended with additional packages. Two such packages will be explored in this chapter, `GGplot2` and `Plotly`. The former is great for static publication quality plots, of which you will begin to recognize, while the latter provides stunning interactive graphics suitable for web development and Shiny applications.

```
set.seed(1234)
plot(sample(1:20),sample(1:20))
```

_main_files/figure-latex/unnamed-chunk-67-1.pdf

```
set.seed(1234)
df <- data.frame( x1 = sample(1:20), y1 = sample(1:20) )
df %>% plot()
```

_main_files/figure-latex/unnamed-chunk-68-1.pdf

7.2 GGPlot

```
library(tidyverse)
```

7.2.1 Syntax

The motivation behind GGplot is based on the grammar of graphics such that

> *"the idea that you can build every graph from the same few components"*

Ideally this accomplishes dual goals of allowing you to quickly construct plots for initial analyses and checking for oddities (as explained above) and following the logical process of the plot construction.

To graph in GGPlot there are a few core embodiments that need to be considered.

data	a table of numeric and/or categorical values	data.frame or tibble
geom	a geometric object or visual representation, that can be layered	points, lines, bars, boxs, etc.
aesthetics	how variables in the data are mapped to visual properties	eg. $x = col\_a$, $y = col\_b$
coordinate	orientation of the data points	eg. *cartesian (x,y), polar*

```r
# the basic structure
ggplot(data, aes(x,y)) + geom_point() + coord_cartiesian()

# combined with dplyr makes for a readable process
data %>% ggplot(aes(x,y)) + geom_point()
```

In this example the `ggplot()` function contains the two components, the data table `data` and mapping function `aes()`. Since *GGPlot* follows a layered modality, the `ggplot()` function "sets" the canvas and passes the data table `data` and mapping function `aes()` to all the functions that follow with the `+` operator.

Lets create some data . . .

```r
set.seed(5)
tbl_mz <- tibble(
  mz = sample(3500:20000/10, 100),
  int = rlnorm(meanlog = 5, sdlog = 2, 100),
  class = sample(c('A','B','C','D'), 100, replace = TRUE)
)
```

7.2.2 Basic Data Plotting

Points and Lines

Points and lines graphing is a simple way of representing data in a two-dimensional space. In this graph, we use points to represent individual data values, and lines to connect them. The x-axis usually represents the **independent** variable while the y-axis represents the **dependent** variable - or in other words, what y was observed while measuring x.

To plot a point, we use an ordered pair of values (x, y) that correspond to the position of the point on the graph. For example, the point (2, 5) would be plotted 2 units to the right on the x-axis and 5 units up on the y-axis.

We can also connect points with lines to show a trend or pattern in the data. These lines can be straight or curved, depending on the nature of the data. A straight line can be drawn to connect two points or to represent a linear relationship between the variables.

```
p01 <- tbl_mz %>% ggplot(aes(mz, int)) + geom_point()
p02 <- tbl_mz %>% ggplot(aes(mz, int)) + geom_line()
p03 <- tbl_mz %>% ggplot(aes(mz, int)) + geom_point(color='red') + geom_line(color='bla
p04 <- tbl_mz %>% ggplot(aes(mz, int)) + geom_line(color='black') + geom_point(color='r
```

It's important to take note of the difference between plots *p03* and *p04*. While both plots showcase a similar data representation, a closer look reveals a notable difference. Specifically, in the latter plot (*p04*), we can see that the red points appear under the black line. This occurs because the points were layered first, and then the lines were layered over them. This is a crucial distinction to make as it highlights the importance of the order in which layers are applied in the plot.

_main_files/figure-latex/unnamed-chunk-74-1.pdf

Segments

Line segments are an important concept in geometry and are used in various applications. A line segment is a part of a line that is bounded by two distinct end points. It is also a default representation of centroided mass spectra. In this case the segment will start and end on the same x (mz), while the y (int) component will end at 0.

```
p05 <- tbl_mz %>% ggplot(aes(mz, int)) + geom_segment(aes(xend = mz, yend = 0))
```

Bar Chart

When it comes to representing categorical data, bar charts are considered to be the most effective visualization tool. Bar charts are simple, yet powerful, and can be used to display data in a clear and concise way. They are easy to read and understand, and are a popular choice among data analysts, researchers, and business professionals. Whether you're trying to visualize sales data, survey results, or demographic information, bar charts are a great option to consider.

So, if you're looking for a way to represent categorical data, consider using a bar chart for the most accurate and comprehensive representation.

```
p06 <- tbl_mz %>% ggplot(aes(class)) + geom_bar()
```

Pie Chart

You maybe considering a pie chart, which is a circular diagram divided into sectors, with each sector representing a proportion of the whole. It is commonly used to display percentages, where the sum of the sectors equals 100%. There is no specific `geom` to build piechart with ggplot2. The trick is to build a barplot and use `coord_polar()` to make it circular. However, interpreting pie charts can be challenging since humans are not very skilled at reading angles. For instance, it is often difficult to determine which group is the largest and to arrange them by value. As a result, it is advisable to refrain from using pie charts.

```
p07 <- tbl_mz %>% ggplot(aes(class, fill=class)) + geom_bar()
p08 <- tbl_mz %>% ggplot(aes(class, fill=class)) + geom_bar() + coord_flip()
p09 <- tbl_mz %>% ggplot(aes(1, fill=class)) + geom_bar(position = 'fill') + coord_polar(theta =
```

Note how difficult it is in the pie chart to tell (by eye) that A is the smallest.

7.2.3 Data Distributions

In statistics, a distribution refers to the way in which a set of data is spread out or dispersed. It describes the pattern of values that a variable can take and how frequently each value occurs. A distribution can be characterized by its shape, center, and spread, and can be represented graphically using tools such as histograms, box plots, and density plots.

Histograms

A histogram is a graphical representation of the distribution of a dataset. It is a type of bar chart that displays the frequency of data values falling into specified intervals or ranges of values, known as bins. The x-axis of the histogram represents the bins or intervals, and the y-axis represents the frequency or count of values falling into each bin.

Histograms are widely used to summarize large datasets and identify patterns or trends and to visualize the shape of a distribution, whether it is symmetric or skewed, and whether it has any outliers or gaps in the data. They can also be used to compare the distributions of two or more datasets, by plotting them on the same graph with different colors or patterns.

```r
p10 <- tbl_mz %>% ggplot(aes(log(int))) + geom_histogram(binwidth = 1)
```

Density

A density plot is a graphical representation of the distribution of a dataset. It is formed by smoothing the data values and representing them as a continuous probability density function. The density plot is a variation of the histogram that provides a smoother representation of the data, eliminating the need for binning. It is particularly useful when the data is continuous and the sample size is large. The density plot can be used to identify the shape of the distribution, the presence of multiple modes, and the presence of outliers. Again, it can also be used to compare the distributions of two or more datasets by overlaying them on the same plot.

```r
p11 <- tbl_mz %>% ggplot(aes(log(int))) + geom_density()
```

7.2.4 Extended Syntax

7.2.4.1 Colors

7.2.4.2 Scales

7.2.4.3 Faceting

7.2.4.4 Labels

7.2.4.5 Annotations

7.2.4.6 Package ggrepel

7.3 Plotly

Chapter 8

Mass Spectrometry Data

8.1 Commercial

8.1.1 RAW (Thermo)

8.1.2 WIFF (Sciex)

8.1.3 D (Agilent)

8.1.4 D (Bruker)

8.2 Open Access Raw Data

Mass spectrometry (MS) is a powerful analytical method that can be used to determine the mass-to-charge ratio (m/z) of ions in a sample. Mass spectrometry data is generated from MS experiments and can be stored in various formats, including mzXML, mzML, and mzH5.

Conversion Tools

ProteoWizard is an open-source software tool that provides a collection of open-source, cross-platform software libraries and tools for extracting raw mass spectrometry data from from various instrument vendor formats and converting it to the formats listed below.

> *Kessner, D., Chambers, M., Burke, R., Agus, D., & Mallick, P. (2008). ProteoWizard: open source software for rapid proteomics tools development. Bioinformatics, 24(21), 2534–2536. https://doi.org/10.1093/bioinformatics/btn323*
>
> https://proteowizard.sourceforge.io/

mzXML

mzXML is an open XML-based format for encoding MS data. It was developed by the Seattle Proteome Center and is widely used in the mass spectrometry community. mzXML files contain raw MS data, as well as metadata describing the instrument parameters used to acquire the data. mzXML files can be processed using a variety of software tools, such as the Trans-Proteomic Pipeline and ProteoWizard.

The mzXML format has been shown to be effective in handling data from a wide range of instruments. It has a simple structure that makes it easy to parse and process, making it an attractive choice for many researchers. The format is also relatively lightweight, which makes it easy to transfer and store large amounts of data.

> *Pedrioli, P.G., Eng, J.K., Hubley, R., Vogelzang, M., Deutsch, E.W., Raught, B., Pratt, B., Nilsson, E., Angeletti, R.H., Apweiler, R. and Cheung, K., 2004. A common open representation of mass spectrometry data and its application to proteomics research. Nature biotechnology, 22(11), pp.1459-1466.*

mzML

The mzML format is another open XML-based format for MS data, developed by the Proteomics Standards Initiative. It is designed to be more flexible than mzXML and includes more detailed metadata. mzML files can be processed using software tools such as OpenMS and mzR.

The mzML format allows for more detailed and comprehensive data storage than mzXML. This is because mzML has a more complex structure, which enables the storage of a wider range of experimental metadata. The format is also more flexible, which means that it can be easily adapted to different types of experiments and instruments.

> *Martens, L., Chambers, M., Sturm, M., Kessner, D., Levander, F., Shofstahl, J., Tang, W.H., Römpp, A., Neumann, S., Pizarro, A.D. and Montecchi-Palazzi, L., 2011. mzML—a community standard for mass spectrometry data. Molecular & cellular proteomics, 10(1), p.R110. 000133.*

mzMLb

Recently proposed as a new file format based on HDF5 and NetCDF4 standards, mzMLb is faster and more flexible than existing approaches while preserving the XML encoding of metadata. Additionally, it is optimized for both read/write speed and storage efficiency. The format has a reference implementation provided within the ProteoWizard toolkit.

Bhamber, Ranjeet S., et al. “mzMLb: A future-proof raw mass spectrometry data format based on standards-compliant mzML and optimized for speed and storage requirements.” Journal of proteome research 20.1 (2020): 172-183._

Chapter 9

Mass Spectrometry R Packages (work in progress)

9.1 xcms

9.2 deqms

9.3 msempire

9.4 msqrob

9.5 MSstatsTMT

9.6 MSstats

The MSstats package is an R package designed for the analysis of label-free mass spectrometry data. It provides a wide range of statistical tools for the analysis of protein abundance data, including normalization, missing value imputation, quality control, and differential expression analysis. MSstats provides a powerful and flexible way to analyze mass spectrometry data, making it an essential tool for researchers in the field.

Documentation	web, PDF
Code	Bioconductor
Literature	*Bioinformatics* 30.17 (2014): 2524-2526

9.6.1 Installation

To install MSstats, you can use the following code:

```r
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")

BiocManager::install("MSstats")
```

9.6.2 Loading Data

To load your data into MSstats, you can use the following code:

```r
library(MSstats)
data("QCMAPData")
data <- QCMAPData$data
design <- QCMAPData$design
head(data)
```

9.6.3 Quality Control

To perform quality control on your data, you can use the following code:

```r
QCmetrics <- MSstats:::calculateQCmetrics(data, featureLevel = "protein")
MSstats:::plotQCmetrics(QCmetrics, plotType = "boxplot",
                        ylim = c(0, 200000), ylab = "Total Ion Current")
```

9.6.4 Differential Expression Analysis

To perform differential expression analysis on your data, you can use the following code:

```r
MSstatsResult <- MSstats::MSstats(data, design,
                                  normalizationMethod = "quantile",
                                  normalization = "median",
                                  msLevel = 1, verbose = TRUE)
summary(MSstatsResult)
```

9.6.5 Conclusion

Overall, MSstats is a powerful and flexible R package for the analysis of label-free mass spectrometry data. It provides a wide range of statistical tools for the analysis of protein abundance data, making it an essential tool for researchers in the field. With its easy-to-use interface and comprehensive documentation, MSstats is a must-have tool for anyone working with mass spectrometry data.

9.7 msgrob2

`msgrob2` is an R package that provides functions to perform robust estimation in linear models with missing data. With the help of the Expectation-Maximization (EM) algorithm, the package estimates the parameters of the linear model and imputes the missing data. Additionally, the package offers robust methods for estimating the covariance matrix, including the Minimum Covariance Determinant (MCD) estimator and the S-estimator.

The `msgrob2` package is particularly useful in situations where data is missing from a linear model. The EM algorithm implemented in the package is a powerful tool for imputing missing data, and the robust covariance estimators allow for a better understanding of the data. The package is designed to provide efficient and accurate results when working with incomplete data, making it an essential tool for researchers and data analysts.

Documentation	PDF
Code	Bioconductor
Literature	*Molecular & Cellular Proteomics*, **15**(2), 657-668., *Molecular & Cellular Proteomics*, **19**(7), 1209-1219., *Analytical Chemistry*, **92**(9), 6278–6287.

9.7.1 Code Examples

9.7.1.1 Installing Package

To install the `msgrob2` package, you can use the following code:

```
install.packages("msgrob2")
```

9.7.1.2 Loading Package

After installation, you can load the `msgrob2` package using the following code:

```
library(msgrob2)
```

9.7.1.3 Estimating Parameters

To estimate the parameters of a linear model with missing data using `msgrob2`, you can use the `msgrob()` function. The following code demonstrates how to use `msgrob()`:

```
# Loading Example Data
data("exMiss")

# Estimating Parameters
model <- lm(y ~ x1 + x2 + x3, data = exMiss)
msgrob(model)
```

9.7.1.4 Estimating Covariance Matrix

To estimate the covariance matrix using `msgrob2`, you can use the `covrob()` function. The following code demonstrates how to use `covrob()`:

```r
# Loading Example Data
data("exMiss")

# Estimating Covariance Matrix
covrob(exMiss[,2:4], method = "MCD")
```

In summary, `msgrob2` is a comprehensive R package that provides an array of functions for robust estimation in linear models with missing data. The package's implementation of the EM algorithm and robust covariance estimators make it an essential tool for researchers and data analysts working with incomplete data. The package is easy to install and use, with code examples readily available for reference.

9.8 StatsPro R Package

Documentation	github, PDF
Code	github
Literature	*Journal of Proteomics* 250 (2022): 104386.

9.9 Tidyproteomics

The tidyproteomics R package is a tool that provides a set of functions to preprocess and analyze proteomics data using the tidy data framework. This package is built on top of the tidyverse and Bioconductor packages, which are widely used in the R community for data manipulation and analysis.

Some of the main features of the tidyproteomics package include:

- Data preprocessing functions for common tasks such as filtering, normalization, and imputation.
- Functions for quality assessment and visualization of proteomics data.
- Integration with other Bioconductor packages for downstream analysis such as differential expression analysis and pathway analysis.

Documentation	github
Code	github
Literature	

9.9.1 Loading Data

To load your data into tidyproteomics, you can use the following code:

```
# Load the tidyproteomics package
library(tidyproteomics)

# Import data
data_proteins <- "path_to_data.xlsx" %>%
   import("ProteomeDiscoverer", "proteins")
```

9.9.2 Data Summaries

```
rdata <- rdata %>%
  # plot some simple summary stats
  plot_counts(destination = "png") %>%
  plot_quantrank(destination = "png") %>%
  plot_venn(destination = "png") %>%
  plot_euler(destination = "png")
```

Summary Stats

```
rdata <- rdata %>%
  # save a table of simple summary stats
  summary("sample", destination = "save") %>%
  # save a report on contamination
  summary(contamination = "CRAP", destination = "save") %>%
  # remove contamination
  subset(!description %like% "^CRAP")
```

9.9.3 Normalization and Imputation

```
rdata <- rdata %>%
  # normalize via several methods, best method will be automatically selected
  normalize(c("median","linear","limma","randomforest")) %>%
  # impute with a minimum value (this is a knock-out)
  impute(base::min)
  # plot visualizations comparing normalization methods
  plot_normalization(destination = "png") %>%
  plot_variation_cv(destination = "png") %>%
  plot_variation_pca(destination = "png") %>%
  plot_dynamic_range(destination = "png") %>%
  # plot visualizations of unbiased clustering
  plot_heatmap(destination = "png") %>%
  plot_pca(destination = "png")
```

9.9.4 Expression Analysis

```r
rdata <- rdata %>%
  # calculate the expression between experiment: ko and control: wt
  expression(kndw/ctrl) %>%
  # plot the expression analysis
  plot_volcano(kndw/ctrl, destination = "png", significance_column = "p_value") %>%
  plot_proportion(kndw/ctrl, destination = "png")
```

Enrichment Analysis

```r
rdata <- rdata %>%
  # calculate the enrichment of the GO term(s) using the results
  # from the expression analysis
  enrichment(kndw/ctrl, .term = "biological_process") %>%
  enrichment(kndw/ctrl, .term = "cellular_component") %>%
  enrichment(kndw/ctrl, .term = "molecular_function") %>%
  # plot the enrichment analysis
  plot_enrichment(kndw/ctrl, .term = "biological_process", destination = "png") %>%
  plot_enrichment(kndw/ctrl, .term = "cellular_component", destination = "png") %>%
  plot_enrichment(kndw/ctrl, .term = "molecular_function", destination = "png")
```

9.9.5 Advanced

9.9.5.1 Plot Quantitation-Rank with Log2 Cutoff

```r
# SUPPLEMENTAL FIGURES
# plot an alternate quantitative ranking
rdata %>%
  plot_quantrank(display_filter = "log2_foldchange",
                 display_cutoff = 5)
ggsave("plot_proteins_quantitation_rank_filtered.png",
       width = 6, h = 4)
```

9.9.5.2 Plot Comparison of Two Expressions

```r
# import the data again to now avoid imputation
rdata <- path_to_package_data("p97KD_HCT116") %>%
  # import the data set
  import('ProteomeDiscoverer', 'proteins') %>%
  # change the labels on the samples containing 'ko'
  reassign("sample", "ctl", "ctrl") %>%
  reassign("sample", "p97", "kndw")

# run a an expression analysis using a t.test statistical comparison
tbl_expression_ttest <- rdata %>%
  expression(kndw/ctrl, .method = stats::t.test) %>%
  # export the results table to the assigned object
  export_analysis(kndw/ctrl, .analysis = "expression")

# run a an expression analysis using the limma statistical method
tbl_expression_limma <- rdata %>%
  expression(kndw/ctrl, .method = "limma") %>%
  # export the results table to the assigned object
  export_analysis(kndw/ctrl, .analysis = "expression")

# plot the two expression tables two compare similarities between methods
plot_compexp(tbl_expression_ttest,
             tbl_expression_limma,
             labels_column = "gene_name",
             log2fc_min = 1, significance_column = "p_value") +
  ggplot2::labs(x = "(log2 FC) Wilcoxon Rank Sum",
                y = "(log2 FC) Emperical Bayes (limma)",
                title = "Hela p97 KD ~ Ctrl")

ggsave("plot_enrichment_comparison.png",
       width = 6, h = 4)
```

Overall, the tidyproteomics package provides a useful set of tools for preprocessing and analyzing proteomics data using the tidy data framework in R.

Chapter 10

Sharing

This topic covers a variety of ways to share R code with others. The goal is to make your work accessible and reproducible for others. Sharing can take many forms, including sharing your RStudio project, creating a distilled version of your analysis that others can follow, developing a web-based application for others to use, or finding ways to contain and disseminate reproducible analyses. By sharing your work, you enable others to learn from and build upon your research, making it more impactful and useful for the wider community.

10.1 Packrat

Packrat is an R package that provides a way to manage R package dependencies for projects. It is a powerful tool for reproducible research, as it allows you to create a local library of packages specific to a project that can be shared with collaborators or moved to another machine. With Packrat, packages used in a project are kept at a specific version, ensuring that the same results can be obtained regardless of the version of the package used.

Initiating a Packrat Project

To initiate a Packrat project, you need to run the `packrat::init()` function in your R console. This will create a `packrat` directory in your project folder, which will contain all the necessary files and information for Packrat to manage the package dependencies for your project.

```
library(packrat)
packrat::init()
```

Installing a Package into the Project Library

To install a package into the project-specific library, you can use the `packrat::install.packages()` function. Packrat will automatically detect package dependencies and install them as well.

```
packrat::install.packages("dplyr")
```

Loading a Package from the Project Library

To load a package from the project library, you simply use the `library()` function as usual. Packrat will ensure that the correct versions are used.

```
library(dplyr)
```

Updating a Package in the Project Library

To update a package in the project library, you can use the `packrat::update.packages()` function. Packrat will update the package and all its dependencies.

```
packrat::update.packages("dplyr")
```

Overall, Packrat is a valuable tool for reproducible research, as it allows you to manage package dependencies for your projects and ensure that the same results can be obtained regardless of the version of the package used.

10.2 Notebooks

Notebooks in RStudio IDE are interactive documents that allow developers to create and share code, visualizations, and narrative text in a single document. R Notebooks provide an intuitive interface for data analysis, making it easy to explore data, create models, and communicate results.

Using Notebooks

To use notebooks in RStudio IDE, follow these steps:

1. Open RStudio IDE and create a new R Notebook by navigating to File > New File > R Notebook.
2. Add code chunks by clicking the "Insert a new code chunk" button in the toolbar or by using the keyboard shortcut "Ctrl + Alt + I".
3. Write R code in the code chunks and run them by clicking the "Run" button in the toolbar or by using the keyboard shortcut "Ctrl + Enter".
4. Add markdown text to the notebook by typing in markdown cells.

One of the most significant benefits of R Notebooks is that they allow you to mix code and narrative text. You can add markdown cells to your notebook to provide context for your code, explain your thought process and methodology, and document your findings. This feature makes it easy for others to understand your work and reproduce your analysis.

With R Notebooks, it is also possible to:

- Insert tables and images: You can add tables and images to your notebooks using markdown syntax or by using the "Insert" menu in the toolbar.
- Use LaTeX to display formulas: You can use LaTeX syntax to display mathematical formulas in your notebooks.
- Use HTML to display interactive widgets: You can use HTML code to create interactive widgets that allow users to interact with your code and data.

Sharing A Notebook

Sharing your notebook project in RStudio IDE is a straightforward process. By sharing your projects, you can collaborate with other data scientists and benefit from the insights and expertise of your colleagues. Here are the steps to share your projects:

10.3 GitHub

GitHub is an online platform that provides version control and collaboration features for software development projects. It is widely used by developers to store and manage their code repositories, track changes made to code over time, and collaborate with others on projects. It is a powerful tool that simplifies the process of managing code and makes it easier for developers to work together. A key benefit of RStudio IDE is that it has built-in support for version control systems like GitHub, which makes it easy to manage and share code with others.

How to Use GitHub

To use GitHub within RStudio IDE, you need to first create a GitHub account and set up a repository. Once you have created a repository, you can follow these steps to use it within RStudio IDE:

1. Open RStudio IDE and navigate to the "New Project" tab.
2. Select "Version Control" and then "Git".
3. Enter the URL of your GitHub repository and choose a project directory.
4. Click "Create Project" to create a new RStudio project that is linked to your GitHub repository.

How to Share A GitHub Repository

Sharing code with others using GitHub and RStudio IDE is a straightforward process. Once you have set up your GitHub repository and linked it to your RStudio project, you can follow these steps to share code with others:

1. Make changes to your code in RStudio IDE.
2. Commit your changes to the local Git repository using the “Commit” button in the “Git” tab.
3. Push your changes to your GitHub repository using the “Push” button in the “Git” tab.
4. Share the URL of your GitHub repository with others so they can access your code.

10.4 Docker

Docker is an open-source platform that allows developers to easily create, deploy, and run applications in containers. Containers are lightweight, portable, and self-contained environments that can run isolated applications. Docker helps to simplify the process of software development, testing, and deployment by providing a consistent environment that runs the same way on any machine, independent of the host operating system.

For more information check out the main Docker website in addition to the Rocker R Project.

10.5 R Packages

In R, packages are collections of R functions, data, and compiled code that can be easily shared and reused with others. They are an essential part of the R ecosystem and are used for a variety of purposes, such as data analysis, visualization, and statistical modeling.

Creating a package in R is a straightforward process, and RStudio IDE provides several tools to simplify the package development process. Packages are a way of organizing your code and data into a single, self-contained unit that can be easily shared and distributed with other R users.

Creating a Package in RStudio

To create a package in RStudio, follow these simple steps:

1. Create a new R Project. Go to “File” -> “New Project” -> “New Directory” -> “R Package”
2. Choose a name for the package, such as my_new_rpackage and a directory location where it will be saved.
3. Once the project is created, RStudio will generate a basic package structure with the following files:

- DESCRIPTION: This file contains information about the package, such as its name, version, author, and dependencies.
- NAMESPACE: This file defines the package’s API, i.e., the set of functions and objects that are intended for public use.

- R/: This directory contains the package's R source code files.
- man/: This directory contains the package's documentation files.

1. Now it's time to write some code. You can start by creating a simple function that outputs "Hello ASMS". Here's an example:

```r
#' Hello ASMS Function
#'
#' This function prints "Hello ASMS" to the console.
#'
#' @return A character vector with the message "Hello ASMS".
#' @export
say_hello <- function() {
  return("Hello ASMS")
}
```

1. Save the function in a new R script file called "hello_world.R" and place it in the package's R/ directory.
2. Build the package by running "Build" -> "Build & Reload" from the "Build" tab. This will compile the package code and create a binary package file (.tar.gz) in the "build/" directory.
3. Finally, install the package by running "Install and Restart" from the "Build" tab. This will install the package on your local machine, making it available for use.

Using the Package

Once the package is installed, you can load it into your R session using the `library()` function. Here's an example:

```r
library(my_new_rpackage)
say_hello()
```

This will output "Hello ASMS" to the console.

10.6 R Shiny Applications

R Shiny is an R package that allows users to create interactive web applications using R. With R Shiny, users can create and customize web-based dashboards, data visualization tools, and other interactive applications that can be easily shared with others.

The benefits of using R Shiny include creating powerful data-driven web applications with ease and providing a user-friendly interface for data analysis. R Shiny is widely used in various industries, including finance, healthcare, and e-commerce.

How to Create an R Shiny Application in the RStudio IDE

Creating an R Shiny application is relatively easy, and it can be done in the RStudio IDE. Here are the steps to follow:

1. Open RStudio and create a new R script file.
2. Install the 'shiny' R package by running the following command:

   ```
   install.packages("shiny")
   ```

3. Load the 'shiny' package by running the following command:

   ```
   library(shiny)
   ```

4. Create a new Shiny application by running the following command:

   ```
   shinyApp(ui = ui, server = server)
   ```

   The 'ui' argument should contain the user interface (UI) code for the application, while the 'server' argument should contain the server-side code for the application.

5. Write the UI code and server-side code for your application, and save the file with a '.R' extension.
6. Run the application by clicking on the 'Run App' button in the RStudio IDE, or by running the following command:

   ```
   runApp("path/to/your/app.R")
   ```

Example of an R Shiny Application for Plotting Points

Here's an example of an R Shiny application that allows users to plot points on a graph:

```
library(shiny)

# Define UI for application
ui <- fluidPage(
  titlePanel("Plotting Points"),
  sidebarLayout(
    sidebarPanel(
      numericInput("x", "X Coordinate:", 0),
      numericInput("y", "Y Coordinate:", 0),
      actionButton("plot", "Plot Point")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

```
# Define server logic
server <- function(input, output) {
  coords <- reactiveValues(x = numeric(), y = numeric())

  observeEvent(input$plot, {
    coords$x <- c(coords$x, input$x)
    coords$y <- c(coords$y, input$y)
  })

  output$plot <- renderPlot({
    plot(coords$x, coords$y, xlim = c(0, 10), ylim = c(0, 10), pch = 19, col = "blue")
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

In this example, the UI code defines a sidebar panel with input fields for the X and Y coordinates of a point, as well as a button to plot the point. The main panel contains a plot that displays all of the points that have been plotted by the user.

The server-side code defines a reactive variable called ‘coords’ which stores the X and Y coordinates of each plotted point. When the user clicks the ‘Plot Point’ button, an observer function is triggered that adds the new point to the ‘coords’ variable. The renderPlot function then plots all of the points on the graph.

Check out the R Shiny web page for more information.