Learning R for Mass Spectrometry

Jeff Jones, Heath Patterson, Ryan Benz

2023-05-16

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.

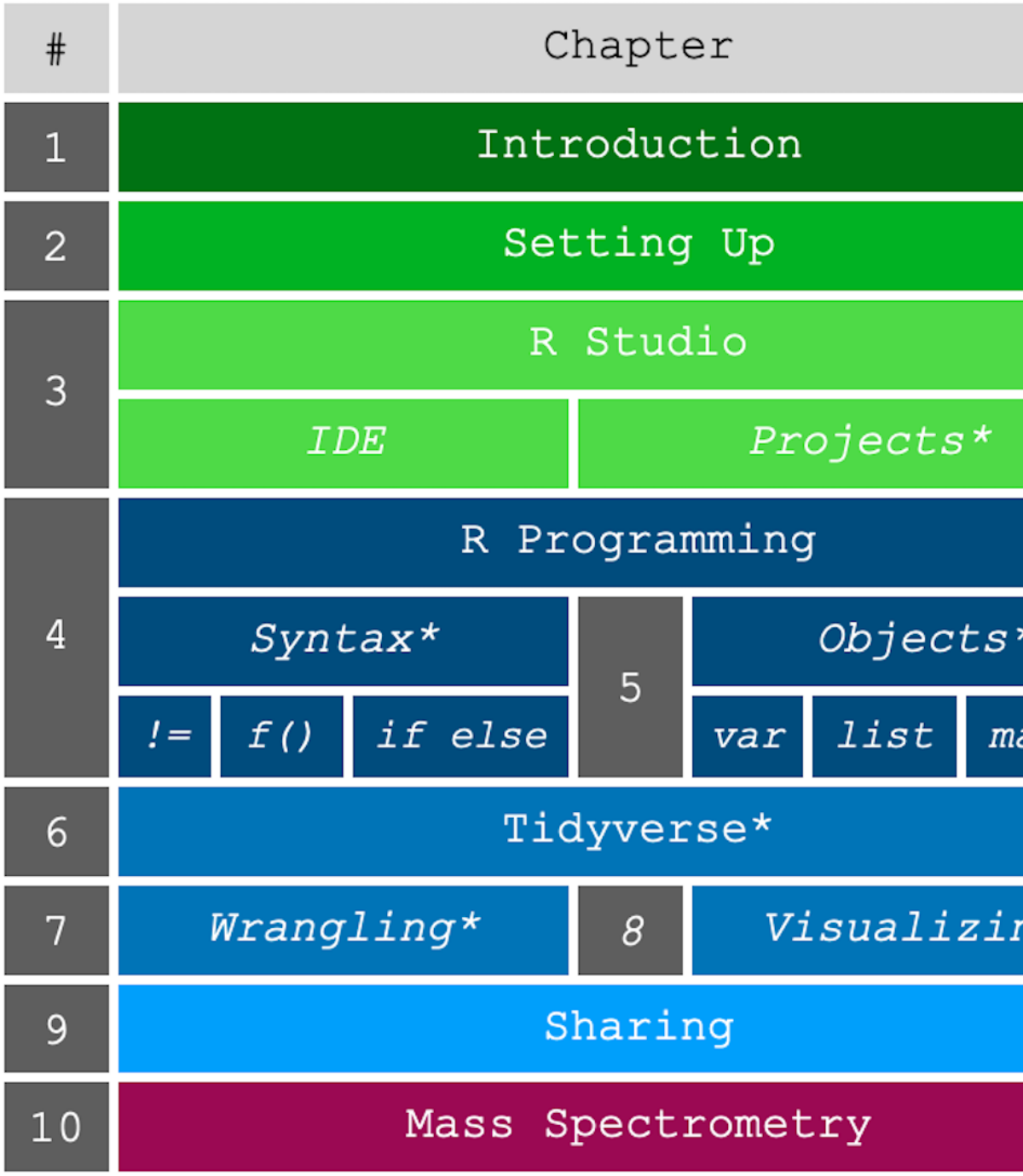
Where can the book be accessed? The book is available online here [link]. Additionally, this book and it's contents are covered at the annual American Society for Mass Spectrometry (ASMS), typically the first week in June the weekend prior to the scientific meeting.

Acknowledgements

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

Contents

This book is organized in a manner such that each chapter builds upon the previous. There are three main sections to this book, where the first section covers the introduction to R and the Integrated Development Environment (IDE) called R Studio. The middle section is the most extensive, basically covering the R programming language its nuances and the main data science packages that will cover the majority of your day-to-day analyses. These chapters all have example exercises to work on that should help sharpen your skills. The last section covers mass spectrometry specific data and some packages you can use to work with that data. This section on mass spectrometry is not meant to be comprehensive, but rather a smaple showcase for what is possible.


#
Chapter
1
Introduction
2
Setting Up
3
R Studio
IDE
Projects*
4
R Programming
Syntax*
!=
f()
if else
5
Objects*
var
list
6
Tidyverse*
7
Wrangling*
8
Visualizing
9
Sharing
10
Mass Spectrometry
* example exercises

Chapter 1

Introduction to R

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

At the end of this chapter you should be able to

- Understand why R is a good choice for data analysis.
- Realize that you have just started the learning curve and all your efforts hence forth are worth it.
- Know where to find additional educational resources.

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.

The Stack Overflow blog post The Impressive Growth of R by David Robinson, discusses the growth and popularity of the programming language R. The post highlights the increase in R's usage on Stack Overflow, as well as the growing interest in R from various industries.

> We found in a previous post that Python has a solid claim to being the fastest-growing programming language in terms of Stack Overflow visits. The same analysis showed that the R programming language has shown remarkable growth in the last five years as well. In fact, R is growing at a similar rate to Python...

The post provides an overview of R's history, its advantages and disadvantages, and its current position in the programming world. The author notes that R's popularity is due to its ability to handle large datasets, its flexibility for data analysis and in increase in popularity of data science and the growing number of companies using R for data analysis. Overall, the post concludes that R's growth and popularity are likely to continue in the future, as more industries recognize the value of data analysis and turn to R as a solution.

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.

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 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

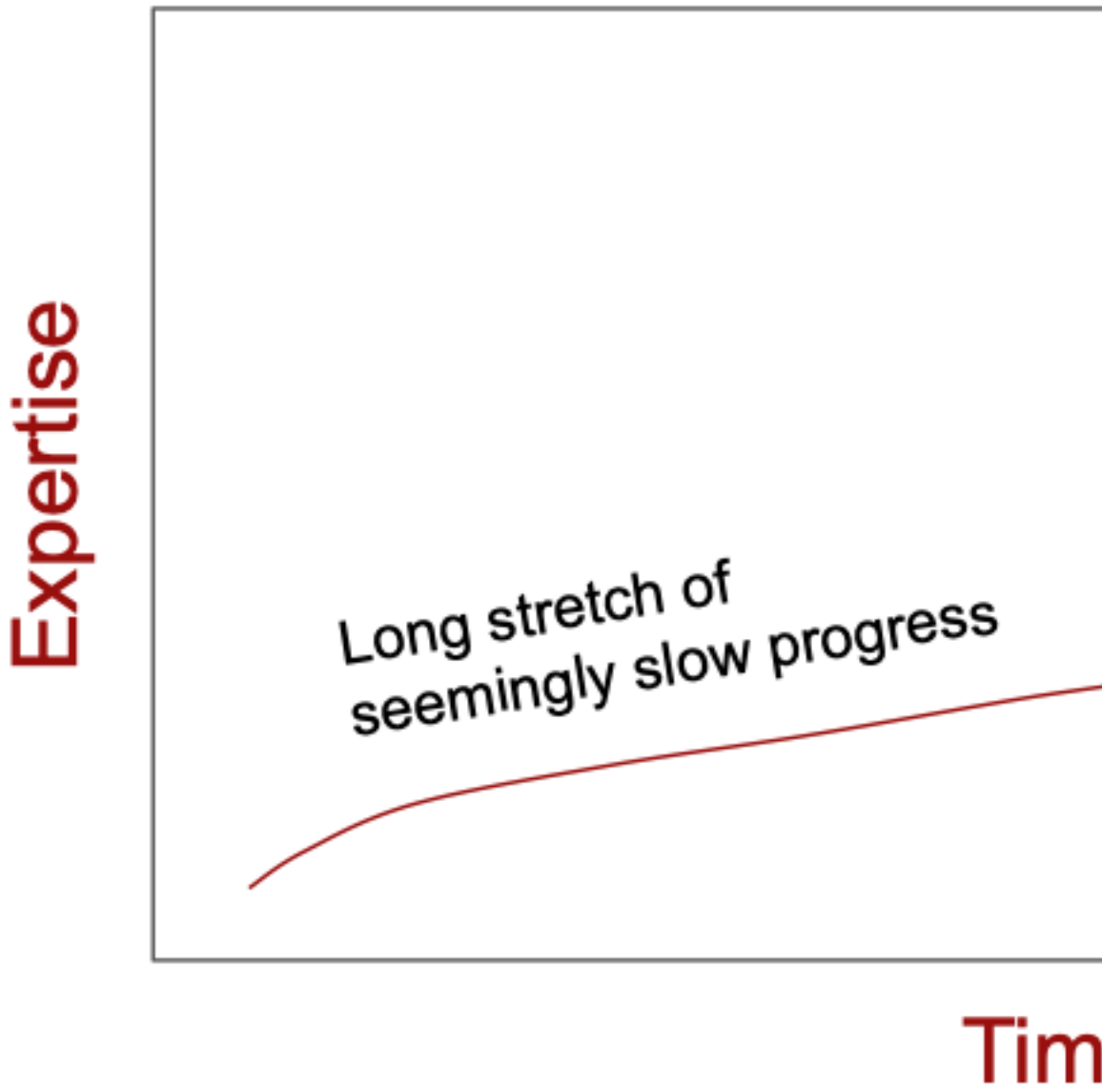


Figure 1.1: R learning curve past

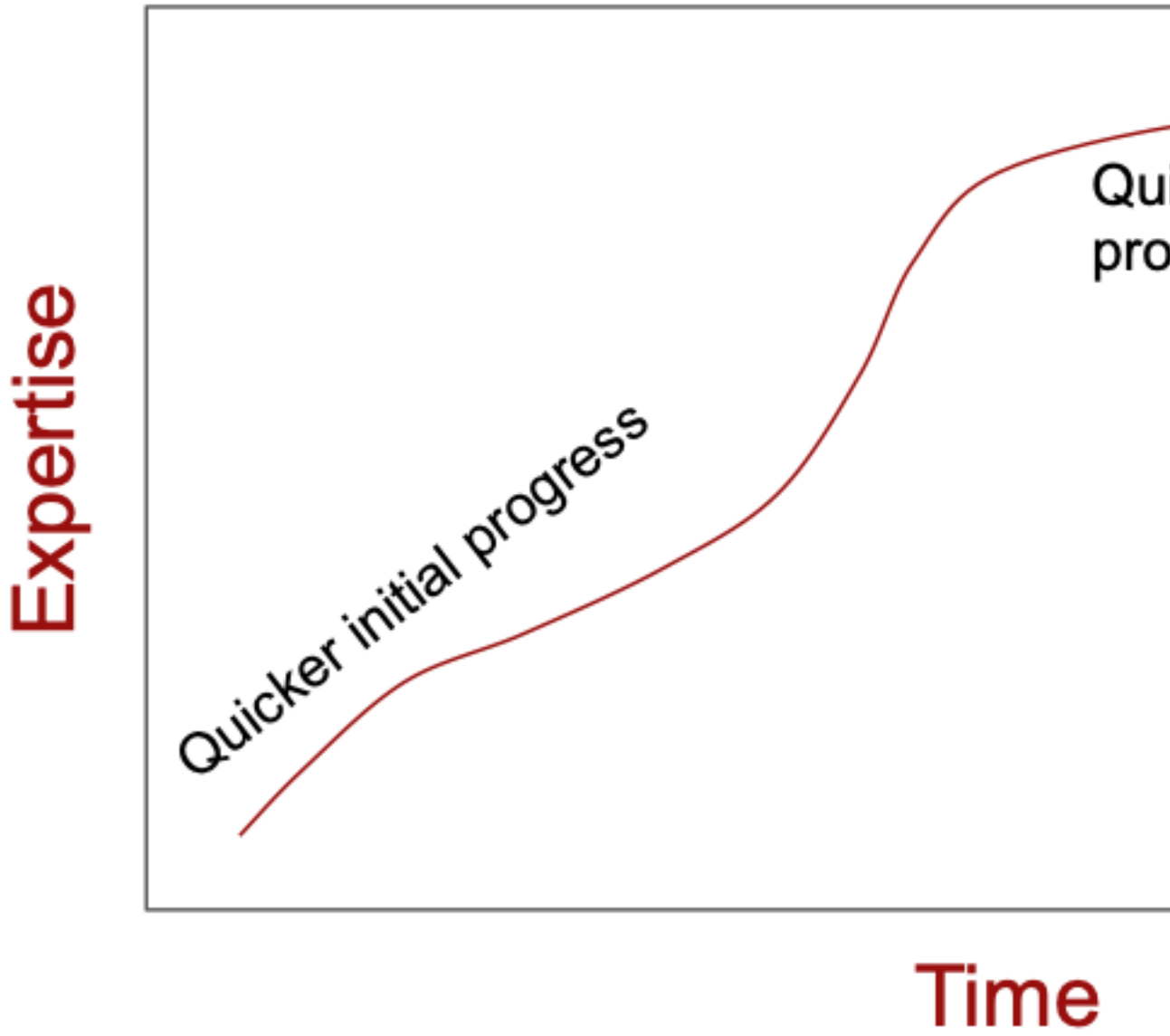


Figure 1.2: R learning curve present

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, that 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.

Resources

When working with R, it is important to understand the basics and terms so that you can ask the right questions when seeking help. In the next two sections, we will provide an overview of these concepts to ensure that you have a solid foundation. It is worth noting that while googling your issue can be a great starting point, it is also important to seek out additional resources to help you solve your problem. For instance, you might consider joining an R community or forum where you can ask questions and receive feedback from other users. Additionally, many universities and organizations offer R workshops or training programs that can help you build your skills and knowledge. By taking advantage of these resources, you can develop a deeper understanding of R and become more confident in your ability to use it for data analysis and visualization.

Online

In addition to Googling to find how to do something in R, there are several online resources available for individuals learning R programming and needing assistance with concepts or coding issues. These resources include CRAN, Bioconductor, RStudio Community, R-bloggers, and Stack Overflow. Each of these resources offers different benefits, such as packages, forums, blogs, and Q&A communities, to help R users.

Locating available packages (pre-built algorithms)

- The Comprehensive R Archive Network: cran.r-project.org/
- Bioconductor: bioconductor.org/

Ways to ask for help, or find answers to a similar question

- The RStudio Community: community.rstudio.com/
- R Community Explorer: r-community.org/rstudio/
- R bloggers: r-bloggers.com/

- Stack Overflow: stackoverflow.com/

More ways to search and find what you are looking for

- R-Seek: rseek.org/
- R-universe: r-universe.dev/search/

Cheat-sheets

- Tidyverse: www.tidyverse.org/
  - reading data: readr.tidyverse.org/
  - manipulating data :dplyr.tidyverse.org/
  - arranging table data: tidyr.tidyverse.org/
  - visualizing data ggplot2.tidyverse.org/
  - working with strings: stringr.tidyverse.org/
  - working with date and time: lubridate.tidyverse.org/

In Print

R books in print are becoming increasingly popular due to the growing demand for R programming and data analysis. These books offer several benefits that make them an excellent resource for anyone who wants to learn R programming or improve their data analysis skills. One of the key advantages of R books in print is that they are easy to read and navigate. The authors of these books take into consideration that not everyone who reads their books is an expert in programming. They use simple language and examples to explain concepts from the basics, making it easy for readers to understand. They can be used as a quick reference guide when working on a project or when facing a programming challenge. R books in print are cost-effective. While online resources are free, they are not always reliable, and it can be time-consuming to find the information you need. R books in print, on the other hand, are written and edited by experts who have years of experience in the field making them a reliable source of information.

R for Data Science: Import, Tidy, Transform, Visualize, and Model Data 1st Edition *by Garrett Grolemund, Hadley Wickham*

> Learn how to use R to turn raw data into insight, knowledge, and understanding. This book introduces you to R, RStudio, and the tidyverse, a collection of R packages designed to work together to make data science fast, fluent, and fun. Suitable for readers with no

previous programming experience, R for Data Science is designed to get you doing data science as quickly as possible.

Use R!, a collection of 67 print books.

The Use R! collection of print books is a series of books aimed at helping people learn and use the R programming language. The books in this series cover a wide range of topics related to R, including data analysis, statistical modeling, and data visualization.

Each book in the collection is written by a different author or group of authors, and provides a unique perspective on how to use R for different tasks. The books are targeted at a range of audiences, from beginners who are just starting to learn R, to more advanced users who are looking to expand their skills and knowledge.

Some of the popular books in the Use R! collection include:

- **An Introduction to R** by Venables and Smith: This book provides a comprehensive introduction to the R programming language, covering topics such as data types, control structures, and functions.
- **Data Manipulation with R** by Spector: This book covers how to use R to manipulate data, including topics such as data cleaning, merging, and reshaping.
- **ggplot2: Elegant Graphics for Data Analysis** by Wickham: This book provides an in-depth introduction to the ggplot2 package in R, which is used for creating high-quality data visualizations.
- **Applied Regression Analysis** by Fox: This book covers how to use R to perform regression analysis, including topics such as linear regression, logistic regression, and mixed-effects models.

In summary, the Use R! collection of print books is a valuable resource for anyone looking to learn or improve their skills in the R programming language. With a wide range of topics and authors, there is something for everyone in this collection.

Organizations

Many R organizations offer individuals an opportunity for training and support, networking, access to resources, and help building a reputation. These benefits make R organizations a valuable resource to consider for both individuals and organizations using R for statistical computing and graphics.

R User Group (RUG)
RUGs are a relaxed and friendly way to broaden your contacts, scope and understanding of R. |

R User Groups

R User Groups are communities of people who are interested in using R, a programming language and software environment for statistical computing and graphics. These groups are formed to provide a platform for individuals to learn, share knowledge, and collaborate on projects related to R programming.

R User Groups usually meet on a regular basis, either virtually or in person, and organize events such as talks, workshops, and hackathons. These events are designed to provide members with opportunities to improve their skills, network with like-minded individuals, and work on projects that are of mutual interest.

R User Groups are open to anyone who is interested in using R, regardless of their level of expertise. Members can range from beginners who are just starting to learn R, to experienced professionals who use R on a daily basis. This diversity of membership allows for a rich exchange of ideas and perspectives on the use of R in various fields, such as data science, finance, and healthcare.

Joining an R User Group can be a great way to stay up-to-date with the latest developments in R programming, as well as to learn from and collaborate with other members. Many R User Groups also have online forums or discussion boards where members can ask questions, share resources, and seek feedback on their work.

Where to find a RUG

- R Community
- Another list of RUGs

R Specific Confrences

The R Confrence

The *R Conference* currently takes place in New York, Washington D.C., and soon Dublin, Ireland. They were created to foster the local R communities and

serve as gathering places for people to learn from their peers. The *R Conference* hosts one of the most elite gatherings of data scientists and data professionals who come together to explore, share, inspire and to promote the growth of open source ideals.

D4 Confrence

Innovation and Entrepreneurship in Data, Design, Development and Discovery

The D4 conference exists to bring creative communities together and to bolster the exchange of ideas. Data professionals, software developers, and other creatives can meet and collaborate.

R Education at Confrences

ASMS

American Society for Mass Spectrometry

The American Society for Mass Spectrometry (ASMS) was created in 1969 to promote and share knowledge of mass spectrometry. Membership includes over 8,500 scientists from academia, industry, and government labs. Members focus on technique and instrument advancements, as well as research in various sciences. ASMS offers several short-courses (1 or 2-day) covering a myriad of topics, including using R for data analysis.

MSACL

Mass Spectrometry & Advances in the Clinical Lab

MSACL aims to advance mass spectrometry and other advanced technologies in clinical laboratory medicine through education and training of practitioners, physicians, and other healthcare professionals. They also support the development of new technologies for diagnosis, treatment, and prognosis of clinical disorders. MSACL offers resources through their Learning Center on several topics, including using R in clinical data analysis.

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.

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

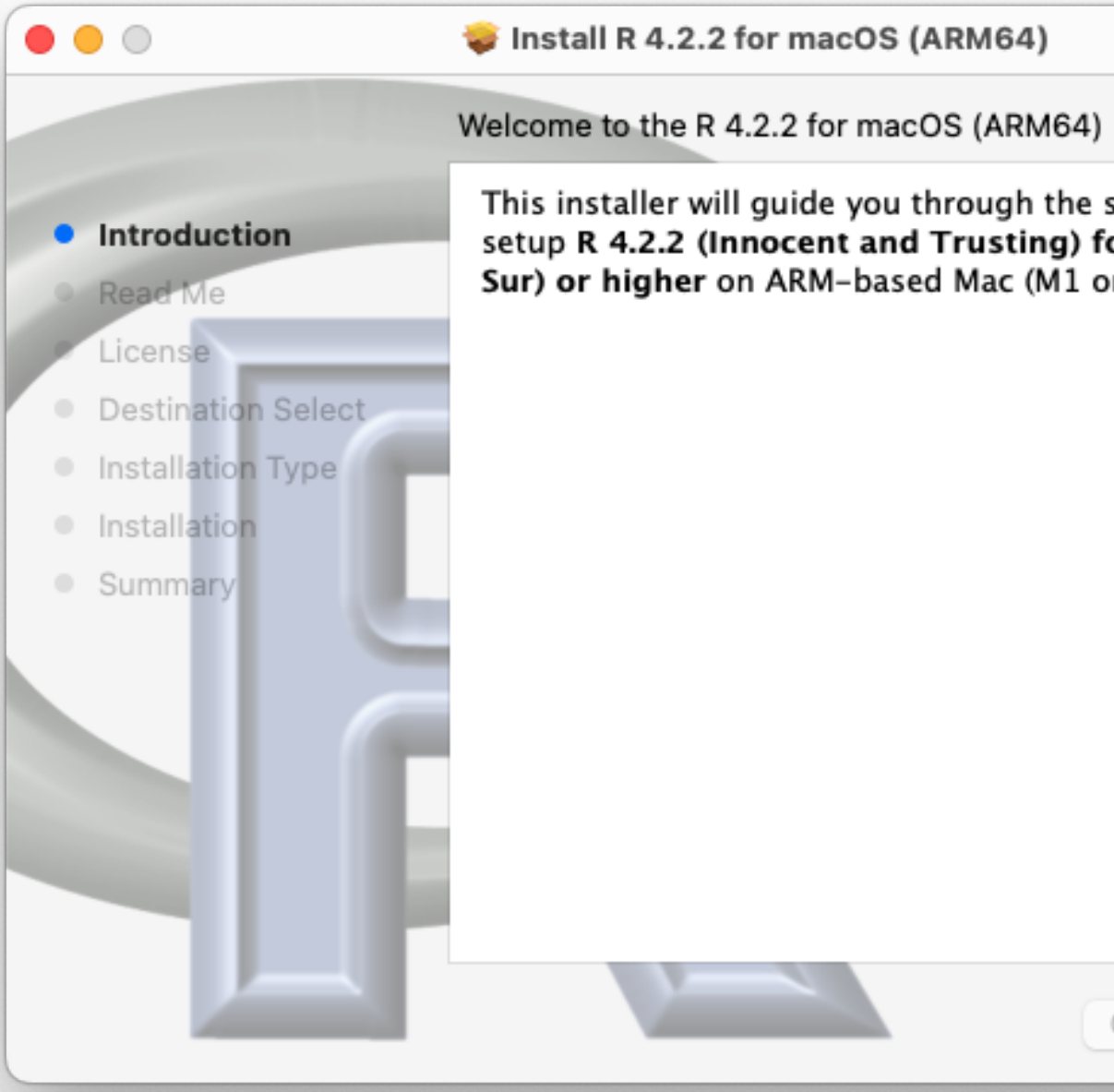


Figure 2.1: Mac Installer

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
>
> "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

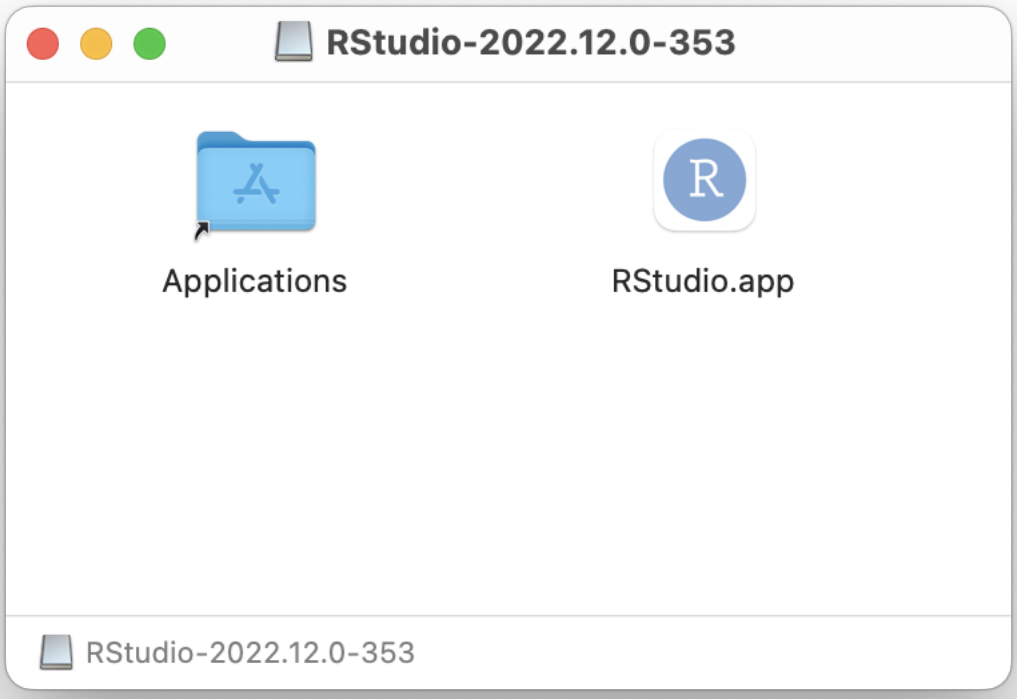


Figure 2.2: RStudio IDE Install

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`

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

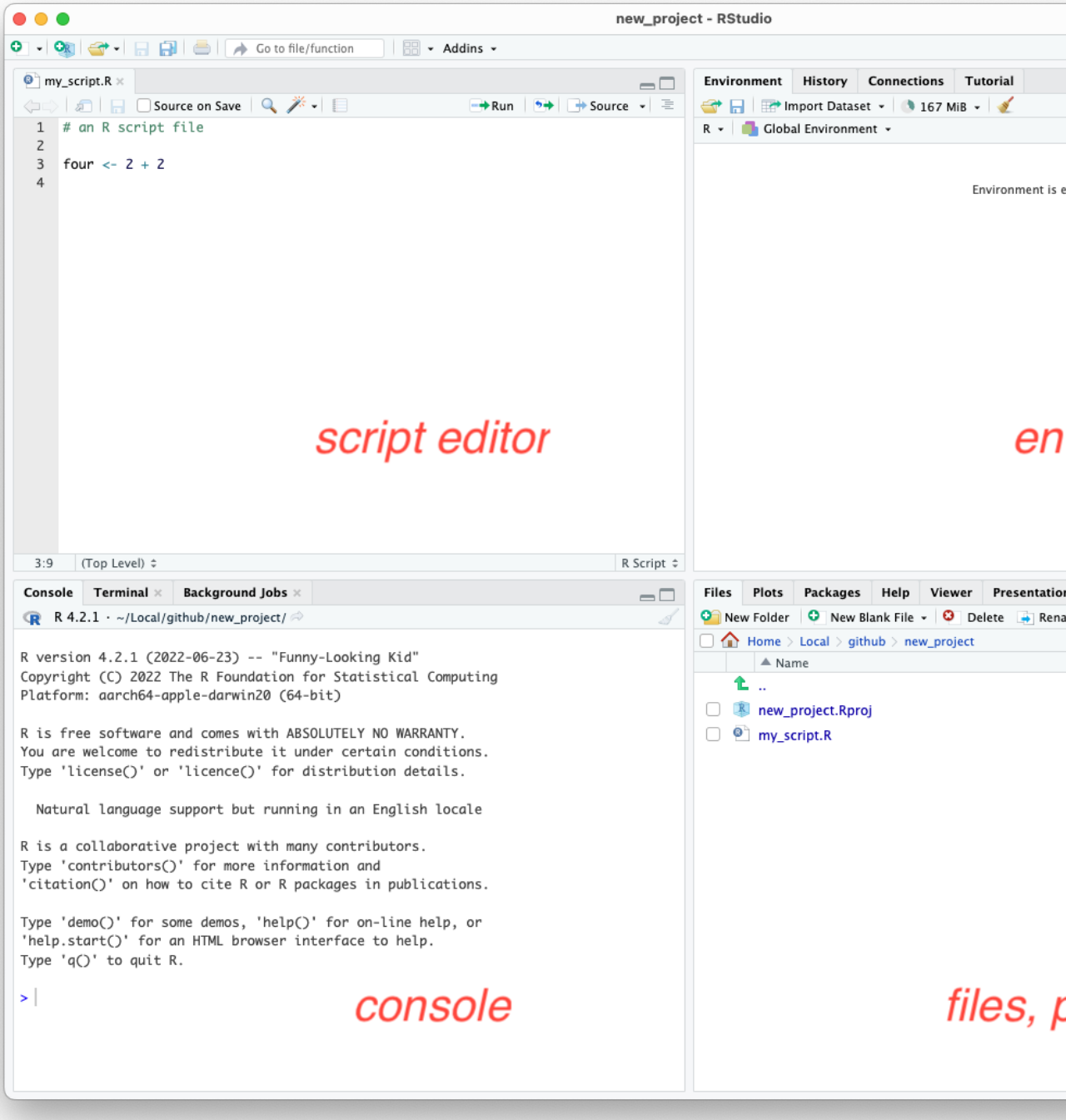


Figure 2.3: RStudio IDE in the default layout

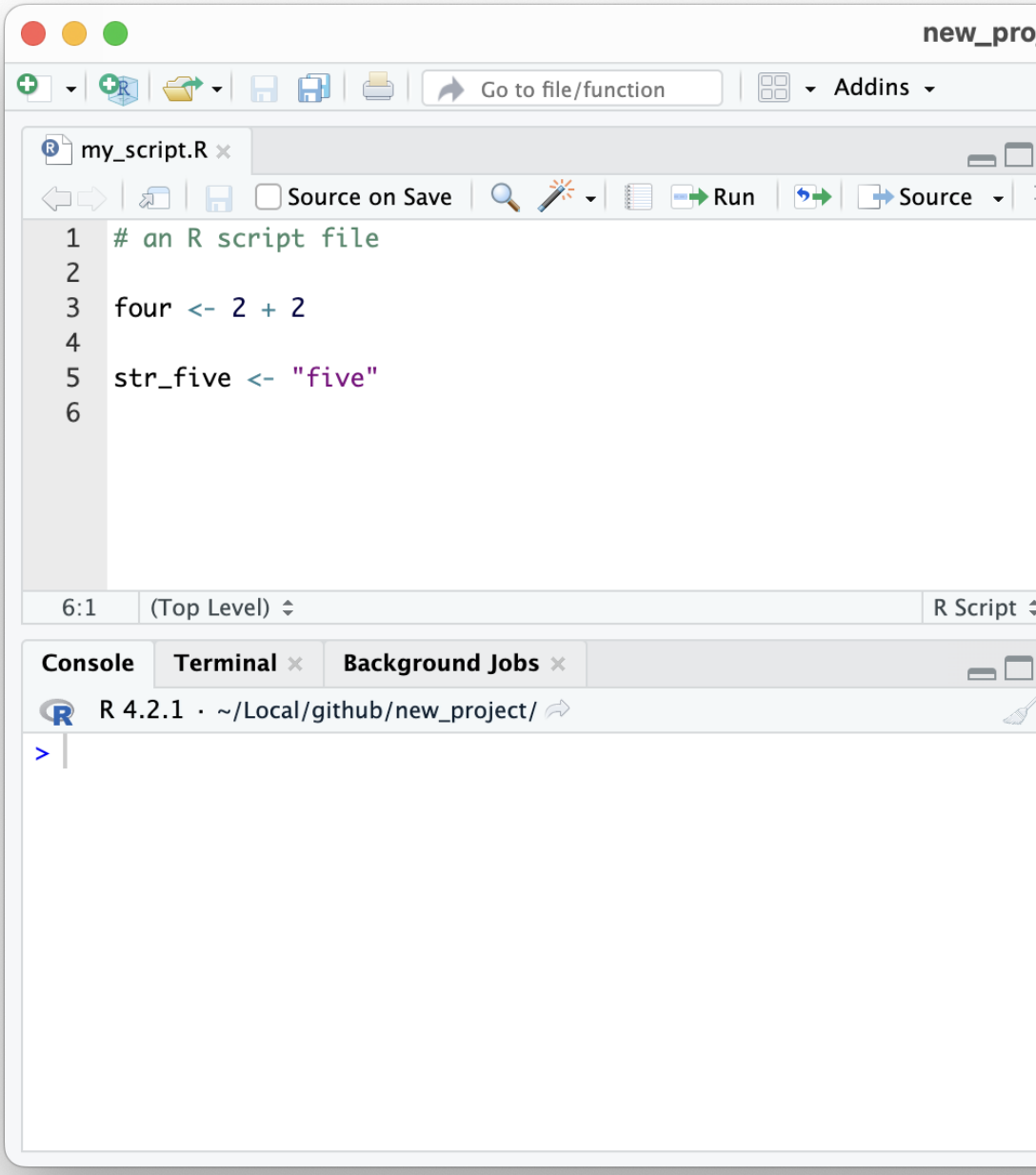


Figure 2.4: RStudio IDE syntax highlighting

anything, and the editor will try to guess what you want to type next. This is a helpful tool that can save you time and effort when writing R code.

Files and Plots

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

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

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

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

The Console

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

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

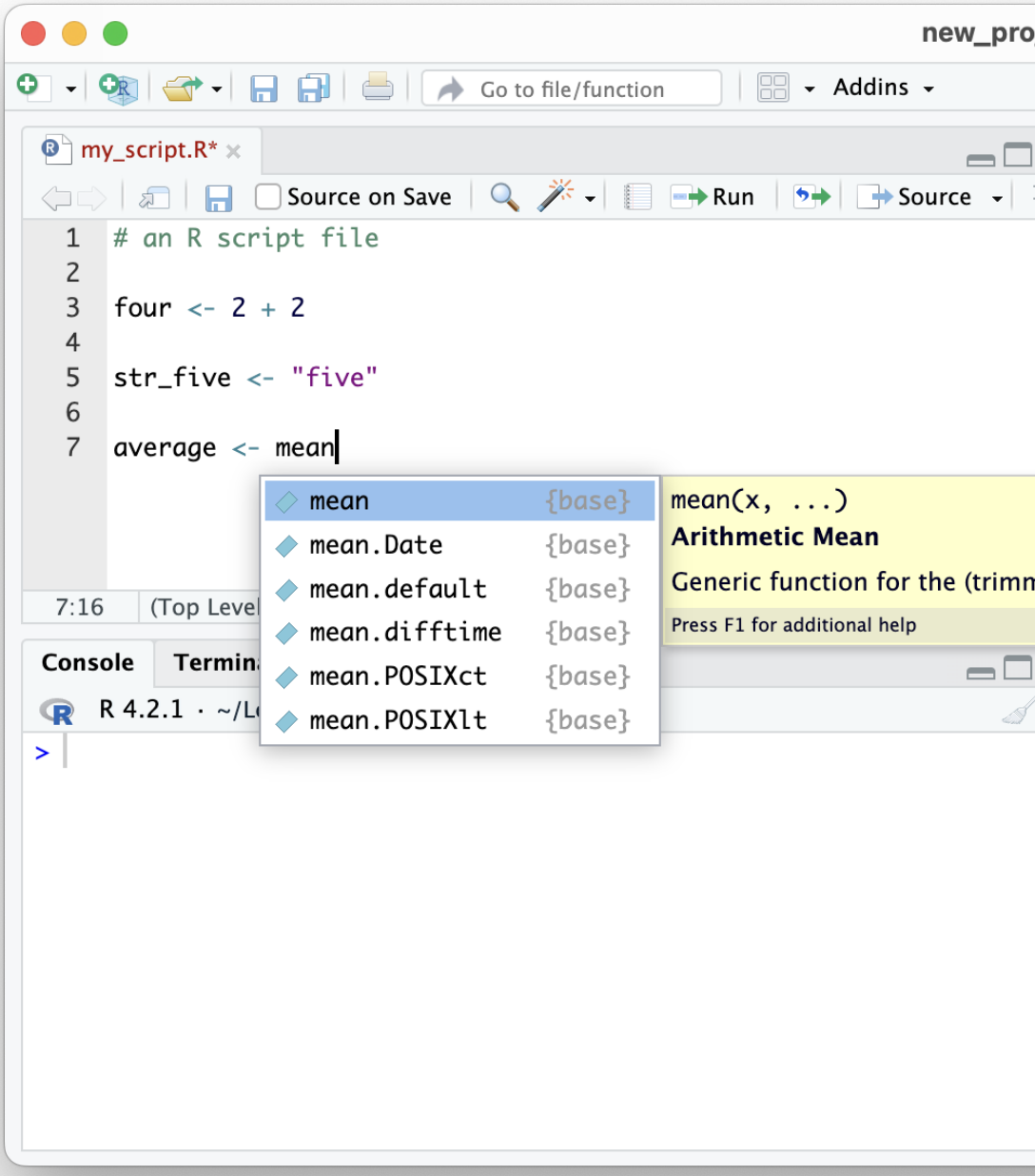


Figure 2.5: RStudio IDE auto complete

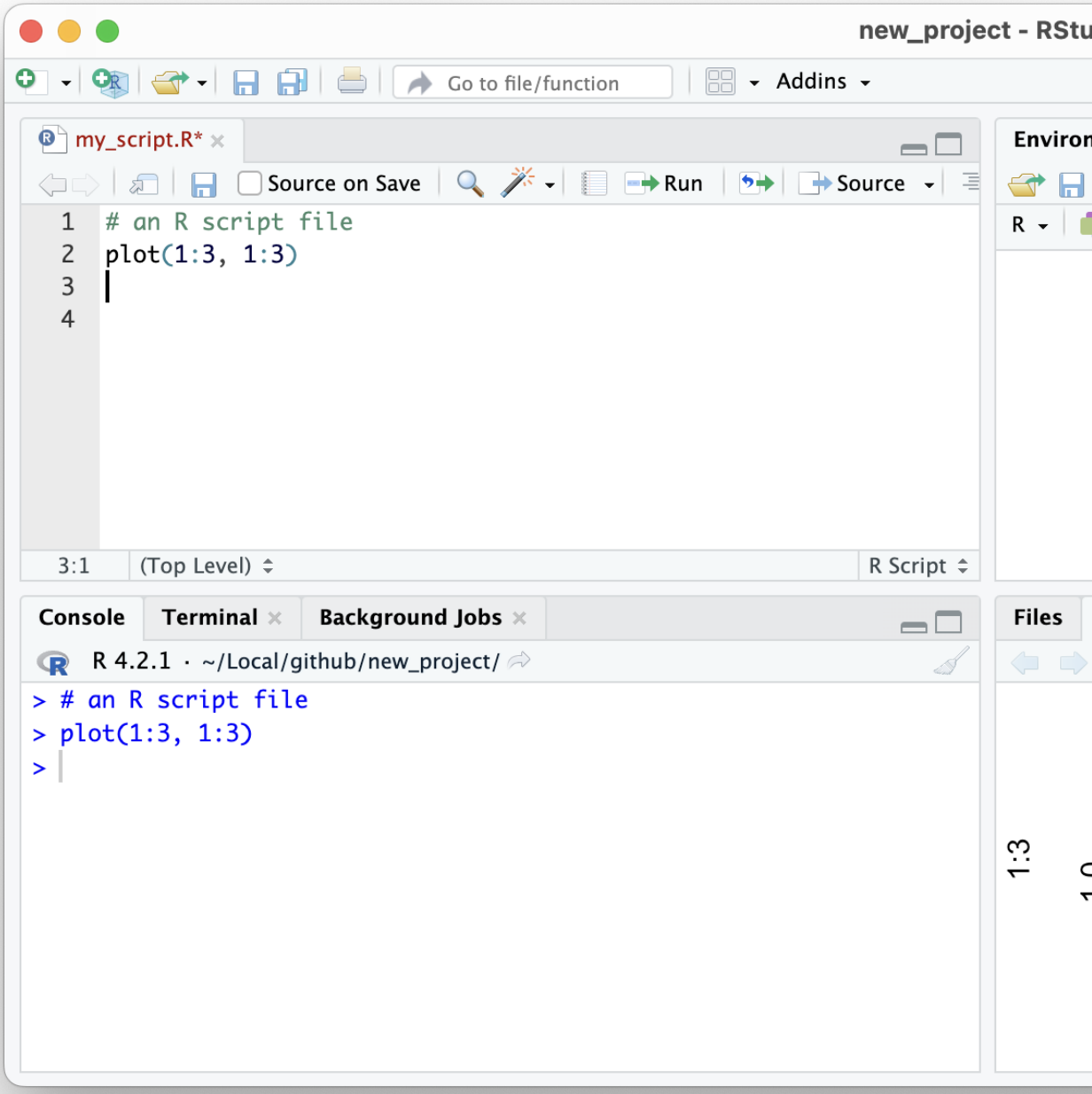


Figure 2.6: RStudio IDE plot window

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 '>'
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.

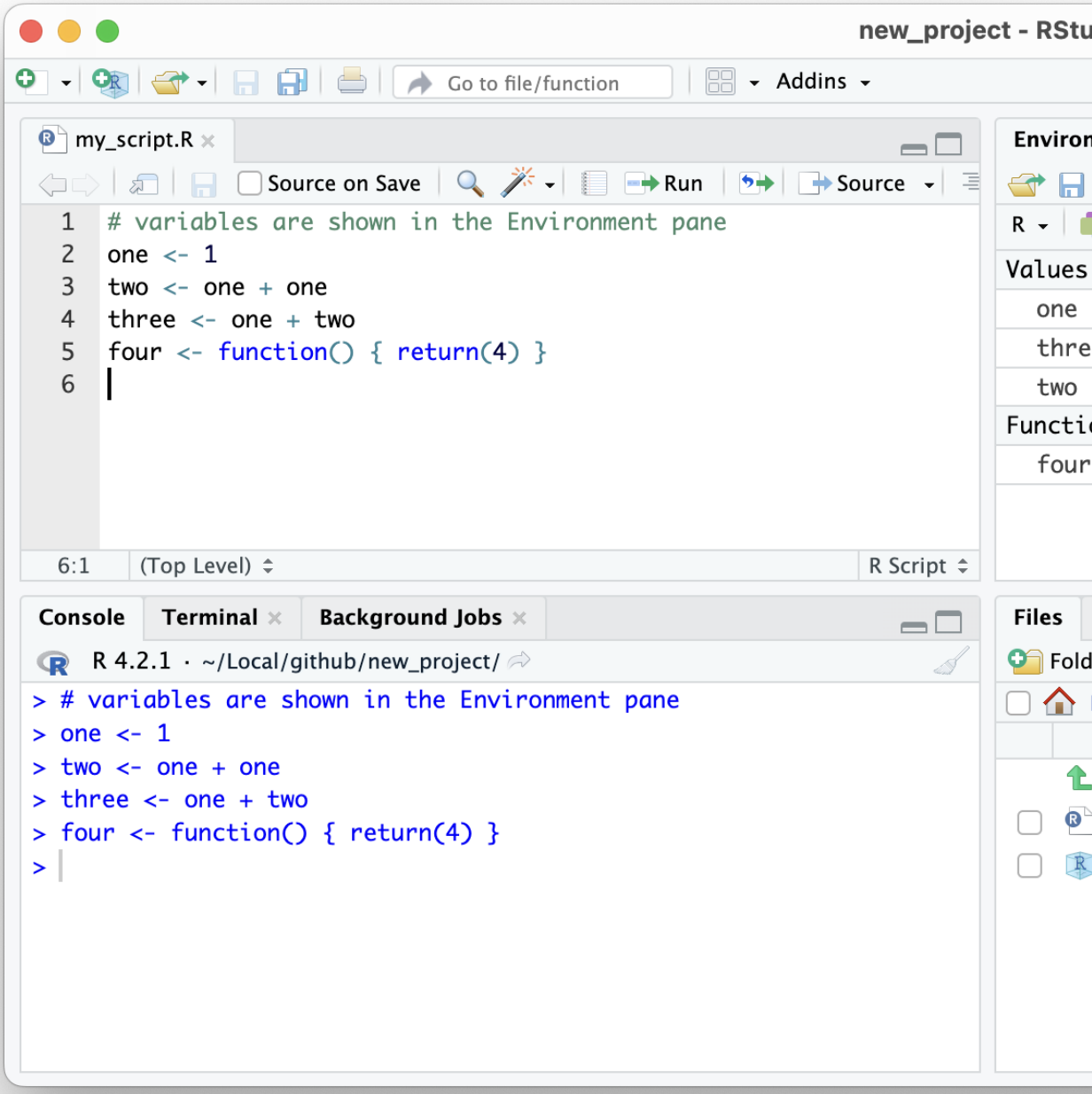


Figure 2.7: RStudio IDE environment window

- 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.

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

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

Installing from Bioconductor

```
# 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

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

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

tidyverse

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

```
library(tidyverse)
```

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

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

```
## Rows: 14 Columns: 7
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (4): ion, seq, pair, type
## dbl (3): mz, z, pos
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

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

other Tidyverse packages, such as **dplyr** and **tidyr**, making it easy to switch between packages and maintain a consistent syntax.

```
print(tbl)
```

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

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

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

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

```
## # A tibble: 2 x 3
```

```
##   type  num_ions avg_mass
##   <chr>    <int>    <dbl>
## 1 b            6     378.
## 2 y            6     424.
```

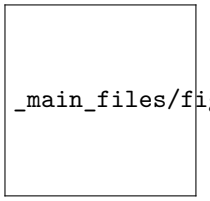
The **tidyr** package contains a set of data table transformations, including pivotting rows to columns, splitting a single column into multiple ones, and tidying or cleaning up data tables for a more usable structure. These transformations are essential for dealing with real-world data tables, which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

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

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

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

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



_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.

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

R Studio 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.

3.1 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`.

3.2 Editing

3.3 Organizing

3.3.1 Data

3.3.2 Scripts

3.3.3 Results

3.3.3.1 Tables

3.3.3.2 Plots

Exercises

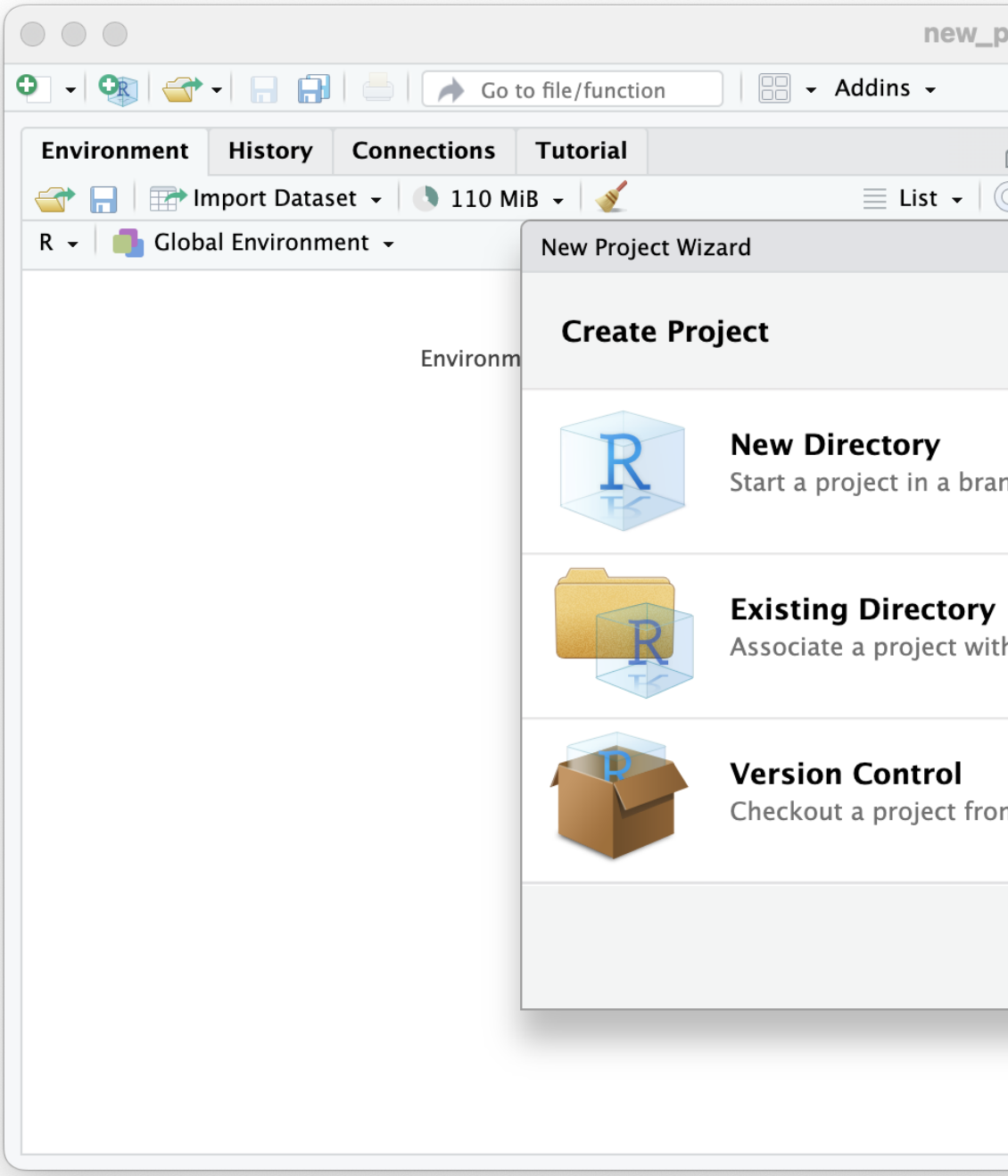


Figure 3.1: RStudio Create New Project

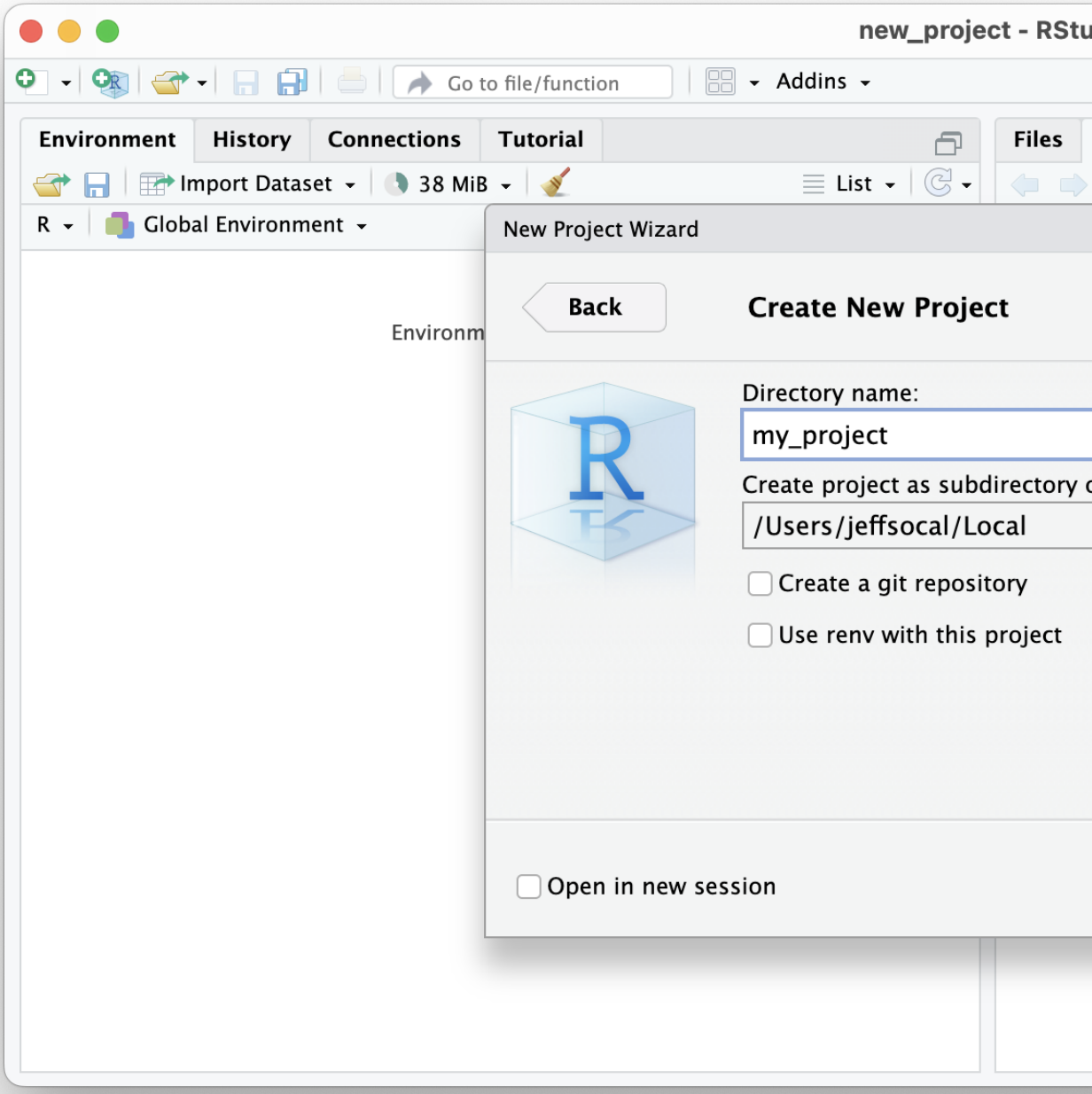


Figure 3.2: RStudio Create New Project

- Create a new R Studio Project and name it **001_new_project**.
- Create a new R script, add your name and date at the top as comments.
- Create an R script file, write something (anything), and save it as **01_test**.
- Create a new R script, add your name and date at the top as comments.
- Locate your new project on your PC.

Chapter 4

R Syntax

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

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

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

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

So, let's get started!

At the end of this chapter you should be able to

- Understand R's syntax, variables, operators and functions.
- Create and edit a project in RStudio.

4.1 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

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. |

4.2 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!

```
# 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
```

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

4.2.1 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
```

```
# This is a comment

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

4.2.2 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.

```
# 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.

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

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

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

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

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

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

4.2.3 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.

```
# 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.

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

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

4.2.4 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
`==`	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`

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

4.2.5 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
```

In R it is easy to overwrite existing variables, either initialized by R or created by you, causing error and confusion.

```
pi <- 9.876543
pi
```

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

4.2.6 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`

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. |

4.2.7 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.

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

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

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

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

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

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

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

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

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

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

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

... or just a normal function.

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

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

The notion of an infix operator you and ignore for the most part. But, we will

see it again when diving into the `tidyverse` - a collection of arguably the most powerful data manipulation packages you will encounter. For now, lets move on with more about `functions()`.

4.3 Flow-Control

4.3.1 If-Else Statements

4.3.2 Loops

4.3.2.1 For

4.3.2.2 While

Exercises

- Create a new R Studio Project and name it **002_basics**.
- Create a new R script, add your name and date at the top as comments.

```
# Your Name
# YYYY-MM-DD
# Institution
#
# Description
```

3. Calculate the sum of 2 and 3.

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

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

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

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

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

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

```
even(3)
```

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

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

```
oddeven(3)
```

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

Chapter 5

R Objects

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`.

At the end of this chapter you should be able to

- Understand the 5 most common data structures.
- Understand the data structure lineage.
- 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.

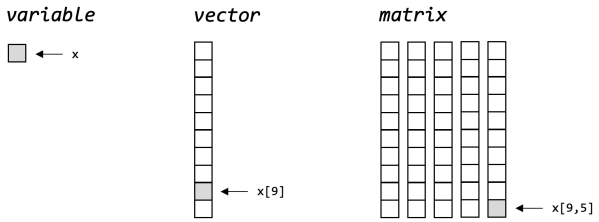


Figure 5.1: R main data structures

5.1 Variable

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

```
a <- 1
```

5.2 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" "U" "V" "
## [24] "X" "Y" "Z"
```

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

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

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

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

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

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

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

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

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

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

While variables don't require a referencing scheme, because they only contain a single value, vectors need to have some kind of referencing scheme, shown in **??** as `x[9]` and illustrated in the following example.

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.

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

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

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

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

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

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

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

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

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

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

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

5.3 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.

```
# 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
```

5.4 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.

5.5 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:

```
# 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:

```
# 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.

5.6 Tibbles

A tibble is a modern data frame in R programming language. Tibble is a part of tidyverse package that provides an efficient and user-friendly way to work with data frames. Tibbles are similar to data frames, but they have better printing capabilities, and they are designed to never alter your data.

Tibbles are created using the `tibble()` function. You can create a tibble by passing vectors, lists, or data frames to the `tibble()` function. Once created, you can manipulate the tibble using the `dplyr` package.

```
library(tidyverse)

df <- tibble(
  let = LETTERS,
  pos = 1:length(LETTERS)
)
```

```
## # A tibble: 26 × 2
##    let     pos
##    <chr> <int>
```

```
##  1 A         1
##  2 B         2
##  3 C         3
##  4 D         4
##  5 E         5
##  6 F         6
##  7 G         7
##  8 H         8
##  9 I         9
## 10 J        10
## # ... with 16 more rows
## #  Use `print(n = ...)` to see more rows
```

Tibbles have several advantages over data frames. They print only the first 10 rows and all the columns that fit on the screen. This makes it easier to view and work with large datasets. Tibbles also have better error messages, which makes debugging easier. Additionally, tibbles are more consistent in handling columns with different types of data.

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 tibble is 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 plant growth (eg. height) 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

- Create a new R Studio Project and name it **003_data_structures**.
- Create a new R script, add your name and date at the top as comments.

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.

```
## [1]  3.141593 12.566372 28.274337 50.265488 78.539825
```

4. Extract all circumferences greater than 50.

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

Chapter 6

Tidyverse

The `tidyverse` is actually a collection of R packages designed for data analysis and visualization. It is an essential tool for data scientists and statisticians who work with large datasets.

At the end of this chapter you should be able to

- Grasp the utility of the tidyverse.
- Understand how to construct a data pipeline.
- Composed a simple workflow.

The `tidyverse` packages are built around a common philosophy of data manipulation. The goal is to provide a consistent and intuitive syntax for data analysis that is easy to learn and use. The packages in the `tidyverse` include:

?? magrittr	provides the pipe, %>% used throughout the tidyverse.
?? tibble	creates the main data object.
?? readr	reading and writing data in various formats.
?? dplyr	data manipulation.

?? tidyr	transforming messy data into a tidy format.
?? purrr	functional programming with vectors and lists.
?? stringr	working with strings.
?? lubridate	working with dates and date strings.
?? ggplot2	graphical plotting and data visualization.

These packages work seamlessly together, allowing users to easily manipulate and visualize their data. The `tidyverse` also includes a set of conventions and best practices for data analysis, making it easy to follow a consistent workflow.

Cheat-sheets 1

Consider the following workflow to read in data, calculate a linear regression and visualize the data using nine (9) of the underlying packages in the tidyverse.

```
library(tidyverse)

# readr, tibble, magrittr: read in a table of comma separate values
tbl_csv <- "data/denver_climate.csv" %>% read_csv()

# define a function to fit a linear regression model
lm_func <- function(data) {
  lm(snowfall ~ min_temp, data = data)
}

# readr, tibble, magrittr: using the data imported from above
tbl_csv_lm <- tbl_csv %>%
  # dplyr
  group_by(year) %>%
  # tidyr
  nest() %>%
  # dplyr, purrr: apply the function to each nested data frame
  mutate(model = map(data, lm_func)) %>%
  # dplyr, broom, purrr: extract the coefficients from each model
  mutate(tidy = map(model, broom::tidy)) %>%
```

```
  # tidyr
  unnest(tidy) %>%
  # dplyr, stringr
  mutate(term = term %>% str_replace_all("\\(|\\)", "")) %>%
  # dplyr: retain only specific columns
  select(year, term, estimate) %>%
  # tidyr: convert from a long table to a wide table
  pivot_wider(names_from = 'term', values_from = 'estimate')


#ggplot2
tbl_csv %>%
  ggplot(aes(min_temp, snowfall)) +
  geom_point() +
  # use the linear model data to plot regression lines
  geom_abline(data = tbl_csv_lm,
              aes(slope = min_temp, intercept = Intercept)) +
  # plot each year separately
  facet_wrap(~year)
```

To get started with the `tidyverse`, you can install the package using the following command:

```
install.packages("tidyverse")
```

Once installed, users can load the package and begin using the individual packages within the tidyverse:

```
library(tidyverse)
```

Overall, the tidyverse is an essential tool for data analysis and visualization in R. Its user-friendly syntax and consistent conventions make it easy for data scientists and statisticians to work with large datasets.

6.1 Core Packages

Two important packages in the tidyverse are tibble and magrittr. These core packages enable other data manipulation operations to work seamlessly, improving efficiency and ease of use when working with data in R. Despite their importance, they are often taken for granted.

6.1.1 magrittr

The tidyverse package `magrittr` is a popular R package that provides a set of operators for chaining operations in a sequence. The package was developed by

Stefan Milton Bache and Hadley Wickham. The main goal of `magrittr` is to make code more readable and easier to maintain.

The pipe operator, `%>%`, is the most famous operator provided by `magrittr.` It allows you to chain multiple operations without the need to use intermediate variables. The pipe operator takes the output of the previous function and passes it as the first argument to the next function. This chaining of operations allows for more concise and readable code.

Here is an example of how to use the pipe operator with `magrittr`:

```
# create a vector of numbers
numbers <- c(1, 2, 3, 4, 5)

# use the pipe operator to chain operations
numbers %>%
  sum() %>%
  sqrt()
```

```
## [1] 3.872983
```

In this example, we create a vector of numbers and use the pipe operator to chain the `sum()` and `sqrt()` functions. The output of the `sum()` function is passed as the first argument to the `sqrt()` function. This allows us to calculate the sum and square root of the vector in a single line of code.

Magrittr also provides other useful operators, such as the assignment pipe `%<>%`, which allows you to update a variable in place, and the tee operator `%T>%`, which allows you to inspect the output of an operation without interrupting the chain.

6.1.2 tibble

R `tibble` is a class of data frame in the R programming language. It is an improved alternative to the traditional data frame and is part of the tidyverse package. Tibbles are data frames with stricter requirements, and they provide a streamlined and more efficient way to work with data.

One of the main advantages of tibbles is that they provide a cleaner and more consistent way to display data. Tibbles only show the first 10 rows and all the columns that fit on the screen, making it easier to work with large datasets. Additionally, tibbles automatically convert character vectors to factors, preventing common errors that can occur when working with data frames.

Another important feature of tibbles is the way they handle column names. Tibbles will not allow spaces in column names, and they use backticks to reference columns with non-standard names. This makes it easier to work with datasets that have complex column names. Tibbles also provide a more consistent way to handle missing values. In data frames, missing values can be represented in different ways, such as NA, NaN, or NULL. Tibbles, on the other hand, only use NA to represent missing values, making it easier to work with missing data.

6.2 Importing

6.2.1 readr

R `readr` is a package in the R language that is used to read structured data files into R. The package is an efficient and user-friendly toolkit that allows for the reading of different types of flat files such as CSV, TSV, and fixed-width files. It is part of the tidyverse collection of packages, which is popular among data scientists and statisticians.

One of the key features of `readr` is its ability to quickly read data into R, making it an ideal package for data analysis and data cleaning. `readr` is designed to handle various types of data, including numeric, date, and character data. The package also has advanced features such as automatic guessing of column types, encoding detection, and parsing of dates and times.

```
# read comma separated values
tbl_csv <- "data/denver_climate.csv" %>% read_csv()
```

One of the best things about `readr` is its consistency in dealing with file formats, which allows for easy and fast data manipulation. The package provides a high level of control over the import process, allowing you to specify the location of the data file, the delimiter, and the encoding type. Additionally, `readr` can handle large datasets with ease, making it one of the most efficient packages for data handling.

6.3 Wrangling

6.3.1 dplyr

R `dplyr` is perhaps pne of the most powerful libraries in the `tidyverse`, providing a set of tools for data manipulation and transformation. It is designed to work seamlessly with data stored in data frames.

The library comes with a set of functions that can be used to filter, arrange, group, mutate, and summarize data. These functions are optimized for speed and memory efficiency, allowing users to work with large datasets easily.

Some of the most commonly used functions in `dplyr` are:

- **filter**: used to extract specific rows from a data frame based on certain conditions.
- **arrange**: used to sort the rows of a data frame based on one or more columns.
- **select**: used to select specific columns from a data frame.
- **mutate**: used to add new columns to a data frame.

In addition to these functions, dplyr also includes a set of convenient shortcuts and helpers, such as the pipe operator (%>%) which allows users to chain

multiple operations together in a more intuitive and readable way.

6.3.2 tidyr

R `tidyr` is a package in R that helps to reshape data frames. It is an essential tool for data cleaning and analysis. Tidyr is used to convert data from wide to long format and vice versa, and it also helps to separate and unite columns.

- **pivot_longer**: used to reshape data from a column-based wide format to a row-based long format.
- **pivot_wider**: used to reshape data from a row-based long format to a column-based wide format.

The package `tidyr` also has functions to separate and unite columns. The "separate" function is used when you have a column that contains multiple variables. For example, if you have a column that contains both the first and last name of a person, you can separate them into two columns. The "unite" function is the opposite of separate. It is used when you want to combine two or more columns into one column.

- **seperate**: used to separate a column with multiple values into two or more columns.
- **seperate_row**: used to duplicate a row with multiple values from a given column.

6.3.3 purrr

The `purrr` package is a functional programming toolkit for R that enables users to easily and rapidly apply a function to a set of inputs, returning a list or vector of outputs. It is designed to work seamlessly with the tidyverse ecosystem of packages, but can also be used with base R functions.

The most important feature in `purrr` is its ability to replace loops with functions that save time and effort. The package has a collection of functions that allow you to work with functions that take one or more arguments. Some of these functions include `map`, `map2`, `pmap`, and `imap.`

The map function is purrr's flagship function and is used to apply a function to each element of a list or vector, returning a list of outputs. The `map2` function applies a function to two lists or vectors in parallel, returning a list of outputs. The `pmap` function applies a function to an arbitrary number of lists or vectors in parallel, returning a list of outputs. The `imap` function is similar to map, but also provides the index of the current element in the input vector.

Purrr also includes features such as the possibility of mapping over nested lists, using `map` and variants to iterate over grouped data, and using `map` and variants to modify data in place.

```r
numbers <- list(1, 2, 3, 4, 5)

# define a function to square a number
square <- function(x) { x ^ 2 }

# use map to apply the function to each element of the list
squared_numbers <- map(numbers, square)

# print the result
squared_numbers
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
##
## [[5]]
## [1] 25
```

6.3.4 glue

R `glue` is a tidyverse package that provides a simple way to interpolate values into strings. It allows users to combine multiple strings or variables together into a single string with minimum efforts, simpler than using base R fuctions.

The `glue` function can handle various types of inputs, including vectors, lists, and expressions. It also supports user-defined formats and allows users to specify separators between the values.

One of the significant advantages of using `glue` is that it provides a more readable and concise way to create strings in R. It eliminates the need for multiple `paste()` or `paste0()` statements, which can be cumbersome and error-prone.

For example, instead of writing:

```r
paste0("The value of x is: ", x, ", and the value of y is: ", y)
```

we can use R glue:

```r
glue("The value of x is: {x}, and the value of y is: {y}")
```

This code will produce the same output, but it's more readable and easier to modify.

6.4 Data Types

6.4.1 stringr

The tidyverse package `stringr` provides a cohesive set of functions designed to make working with strings more efficient. It is especially useful when dealing with messy or unstructured data that needs to be cleaned and transformed into a more structured format.

Several functions in `stringr` provides methods working with strings, for example:

- **str_replace**: replaces a pattern with another pattern in a string.

```
str_replace("Hello World", "W.+", "Everyone")
```

```
## [1] "Hello Everyone"
```

- **str_extract**: extracts the first occurrence of a pattern from a string.

```
str_extract("Hello World", "W.+")
```

```
## [1] "World"
```

- **str_split**: splits a string into pieces based on a specified pattern.

```
str_split("Hello World", "\\s")
```

```
## [[1]]
## [1] "Hello" "World"
```

6.4.2 lubridate

The tidyverse package `lubridate` helps with the handling of dates and times. The package has several functions that make it easier to work with dates and times, especially when dealing with data that has different formats.

Some of the functions in `lubridate` package include:

- **ymd** - this is used to convert dates in the format of year, month, and day to the date format in R. For example, `ymd("20220101")` will return the date in R format.
- **dmy** - this is used to convert dates in the format of day, month, and year to the date format in R. For example, `dmy("01-01-2022")` will return the date in R format.
- **hms** - this is used to convert time in the format of hours, minutes, and seconds to the time format in R. For example, `hms("12:30:15")` will return the time in R format.

- **ymd_hms** - this is used to convert dates and times in the format of year, month, day, hours, minutes, and seconds to the date and time format in R. For example, `ymd_hms("2022-01-01 12:30:15")` will return the date and time in R format.

There are also functions for extracting information from dates and times such as `year()`, `month()`, `day()`, `hour()`, `minute()`, and `second()`.

6.4.3 forcats

R `forcats` is a tidyverse package that provides a set of tools for working with categorical data. It is designed to make it easier to work with factors in R, which are used to represent categorical data.

The `forcats` package provides several functions that can be used to manipulate factors, including reordering levels, combining levels, and handling missing values. It also provides functions for working with ordered factors, which are used to represent data that has a natural ordering, such as age groups or ratings.

One of the key benefits of using `forcats` is that it allows you to easily visualize and analyze categorical data. The package provides functions for creating categorical plots, such as bar charts and pie charts, as well as for calculating summary statistics for categorical data.

In addition to its core functionality, `forcats` is also highly customizable. It provides a wide range of options for controlling the appearance of plots and for customizing the behavior of factor manipulation functions.

6.5 Summarizing

6.5.1 dplyr

In the R tidyverse package, summarizing data is a common task performed on data frames. The dplyr package provides a set of functions that makes it easy to summarize data based on one or more variables.

- **group_by**: used to group rows of a data frame by one or more columns.
- **summarize**: used to summarize the data based on one or more aggregate functions.

The summarise() function is used to perform simple summary statistics on data frames. It takes the name of the new variable as well as the summary function that should be used to calculate its value. For example, to calculate the mean and standard deviation of a variable named 'x' in a data frame named 'df', we can use the following code:

```
tbl_csv %>%
  summarise(mean = mean(precipitation),
            sd = sd(precipitation))
```

```
## # A tibble: 1 x 2
##    mean    sd
##   <dbl> <dbl>
## 1  1.29  1.32
```

The group_by() function is used to group data frames by one or more variables. This is useful when we want to summarize data by different categories. For example, to calculate the mean and standard deviation of 'x' by 'group', we can use the following code:

```
tbl_csv %>%
  group_by(year) %>%
  summarise(mean = mean(precipitation),
            sd = sd(precipitation))
```

```
## # A tibble: 4 x 3
##    year  mean    sd
##   <dbl> <dbl> <dbl>
## 1  2016  1.32 1.29
## 2  2017  1.43 1.85
## 3  2018  1.03 1.16
## 4  2019  1.37 0.945
```

The summarize_at() and summarize_all() functions are used to perform summary statistics on multiple variables at once. The summarize_at() function takes a list of variables to summarize, while the summarize_all() function summarizes all variables in the data frame. For example, to calculate the mean and standard deviation of all numeric variables in a data frame named 'df', we can use the following code:

```
tbl_csv %>%
  summarise_all(list(mean = mean, sd = sd))
```

```
## # A tibble: 1 x 12
##   year_mean month~1 max_t~2 min_t~3 preci~4 snowf~5 year_sd month~6 max_t~7 min_t~8
##       <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1     2018.     6.5    64.6    36.6    1.29    4.09    1.13    3.49    15.2    14.2
## # ... with abbreviated variable names 1: month_mean, 2: max_temp_mean, 3: min_temp_m
## #   4: precipitation_mean, 5: snowfall_mean, 6: month_sd, 7: max_temp_sd, 8: min_tem
## #   9: precipitation_sd, *: snowfall_sd
```

Summarizing data is an essential task that can be performed using several functions. These functions make it easy to calculate summary statistics based on one or more variables, group data frames by different categories, and summarize multiple variables at once.

6.5.2 ggplot2

The tidyverse package `ggplot2`, demonstrated at the onset of this chapter, is a data visualization package in R programming language that provides a flexible and powerful framework for creating graphs and charts. It is built on the grammar of graphics, which is a systematic way of mapping data to visual elements like points, lines, and bars.

With `ggplot2`, you can create a wide range of graphs including scatterplots, bar charts, line charts, and more. The package offers a variety of customization options, such as color schemes, themes, and annotations, allowing you to create professional-looking visualizations with ease.

One of the key benefits of `ggplot2` is that it allows you to quickly explore and analyze your data visually. You can easily create multiple graphs with different variables and subsets of your data, and compare them side by side to identify patterns and trends.

Exercises

- Create a new R Studio Project and name it **004_tidyverse**.
- Create a new R script, add your name and date at the top as comments.

1. Calculate the mean of following vector.

```
##  [1]  7.48 14.15  6.23 10.21 15.13  8.19  8.58  8.09  9.14 10.41
```

```
## [1] 9.761
```

2. Pipeline (eg. `%>%`) a data operation that provides the mean of following vector.

```
## [1] 9.761
```

2. Employing a pipeline (eg. `%>%`), construct a tibble with columns named `circ` and `diam` which contains the diameters of circles with integer circumferences 1 to 5. Remember PEMDAS.

```
## # A tibble: 5 x 2
##    circ  diam
##   <int> <dbl>
## 1     1  3.14
## 2     2 12.6
```

```
## 3     3 28.3
## 4     4 50.3
## 5     5 78.5
```

3. Extract all circumferences greater than 50.

```
## # A tibble: 2 x 2
##    circ  diam
##   <int> <dbl>
## 1     4  50.3
## 2     5  78.5
```

4. Add a column named `circ_type` where you assign the string *odd* or *even* depending on the column `circ`. Attempt to use the `purrr::map` function, along with the `oddeven()` function from the previous chapter, then compute the mean, standard deviation, and coefficient of variation of the diameters for each `circ_type`.

```
## # A tibble: 2 x 4
##   circ_type diam_mean diam_sd diam_cv
##   <chr>         <dbl>   <dbl>   <dbl>
## 1 even           31.4    26.7   0.849
## 2 odd            36.7    38.4   1.05
```

Chapter 7

Data Wrangling

7.1 Tidy Data

Exercises

- Create a new R Studio Project and name it **005_data_wrangling**.
- Create a new R script, add your name and date at the top as comments.
- Locate and/or download a Tidyverse cheat-sheet and refer to it as needed.

1. Download *Denver Climate Data* to use as an example data file.

```
url <- "https://raw.githubusercontent.com/ZenBrayn/asms-2022-r-shortcourse/main/exercises/intro_t
download.file(url, destfile = "./data/denver_climate.csv")
```

2. Read in the dataset .csv using the `tidyverse` set of packages.
3. How many rows does the table contain? Number of columns? What are the column names?

```
## [1] 48
```

```
## [1] 6
```

```
## [1] "year"          "month"         "max_temp"      "min_temp"      "precipitation" "snowfall"
```

4. Find the month and year with the highest snowfall.

```
## # A tibble: 1 x 2
##   month  year
##   <dbl> <dbl>
## 1     4  2016
```

5. Find months where there was no snowfall.

```
## # A tibble: 5 x 1
##   month
##   <dbl>
## 1     6
## 2     7
## 3     8
## 4     9
## 5    10
```

6. Find average precipitation per year.

```
## # A tibble: 4 x 2
##    year precipitation_avg
##   <dbl>             <dbl>
## 1  2016              1.32
## 2  2017              1.43
## 3  2018              1.03
## 4  2019              1.37
```

7. Convert temperatures to Celsius and add new columns with these to table. Reminder `F to C conversion -> (F - 32) * (5/9)`.

```
## # A tibble: 48 x 8
##     year month max_temp min_temp precipitation snowfall min_temp_C max_temp_C
##    <dbl> <dbl>    <dbl>    <dbl>         <dbl>    <dbl>      <dbl>      <dbl>
##  1  2016     1     46.8     19.3          0.39      7.4     -7.06        8.22
##  2  2016     2     49.8     22.5          0.47     14.7     -5.28        9.89
##  3  2016     3     53.2     25.7          1.47     19       -3.5        11.8
##  4  2016     4     55.5     33.2          2.13     19.2      0.667      13.1
##  5  2016     5     64.6     39.6          1.95      0.7      4.22       18.1
##  6  2016     6     82.4     54.7          4.64      0       12.6        28
##  7  2016     7     87       57.4          2.08      0       14.1        30.6
##  8  2016     8     83.8     53.9          1.18      0       12.2        28.8
##  9  2016     9     79.8     48.5          0.12      0        9.17       26.6
## 10  2016    10     74.1     40.6          0.16      0        4.78       23.4
## # ... with 38 more rows
```

8. Find the month with the greatest difference in C between max and min temperature.

```
## # A tibble: 1 x 2
##   month temp_diff_C_avg
##   <dbl>           <dbl>
```

```
## 1     9            16.9
```

9. Write to file an .csv of the data.frame which contains the C data.

Chapter 8

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.

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 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.

Table 8.1: 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

At the end of this chapter you should be able to

- Understand the need for visualizations.
- Create some simple plots of points, lines and bars.
- Manipulate how a plot looks.

8.1 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. Three such packages widely used are `lattice`, `GGplot2`, `Plotly`. This chapter will dive into `GGplot2` which is integrated with the tidyverse, and is great for static publication quality plots. The other two will briefly be covered as suitable alternatives.

```
plot(sample(1:20),sample(1:20))
```

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

8.2 GGPlot2

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.

8.2.1 Syntax

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*

```
# 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.

```
library(tidyverse)
```

Lets create some data ...

```
set.seed(5)
n_peaks <- 300
tbl_mz <- tibble(
  mz = sample(3500:20000/10, n_peaks),
  int = rlnorm(meanlog = 5, sdlog = 2, n_peaks),
  class = sample(c('A','B','C','D'), n_peaks, replace = TRUE)
)
```

8.2.2 Basic Data Plotting

Cheat-sheets 1

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='
```

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.

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 a pie-chart 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.

8.2.3 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.

```
p10 <- tbl_mz %>%
  ggplot(aes(log10(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.

```
p11 <- tbl_mz %>%
  ggplot(aes(log10(int))) +
  geom_density()
```

Box-Plot

One of the most commonly used types of plots in GGplot2 is the box plot. A box plot is used to display the distribution of a continuous variable. It shows the median, interquartile range, and any outliers present in the data.

Box plots are useful in scientific analysis because they allow us to quickly see the distribution of a variable and identify any potential outliers. They are particularly useful when comparing the distribution of a variable across different groups or categories. For example, we may use a box plot to compare the distribution of values across different class levels.

To create a box plot in GGplot2, we use the `geom_boxplot()` function. We specify the variable we want to plot on the y-axis and any grouping variables on the x-axis.

```
p12 <- tbl_mz %>%
  ggplot(aes(class, log10(int))) +
  geom_boxplot()
```

8.2.4 Extended Syntax

One way to enhance the functionality of ggplots is by using additional modifiers. These modifiers can help you to create more intricate and detailed visualizations that better represent your data. By tweaking the parameters of your ggplots, you can create visualizations that are more informative, aesthetically pleasing, and tailored to your specific needs. Whether you want to adjust the color scheme, add annotations, or modify the axis labels, additional modifiers can help you to achieve your desired outcome.

Colors

We saw a bit how to adjust colors in the previous plots. The two color arguments to consider are `color`, which modifies the point, line and edge color, and `fill`, which modifies the internal color of a shape for plots such as `geom_bar` and `geom_histogram`.

```
p13 <- tbl_mz %>%
  ggplot(aes(log10(int))) +
  geom_histogram(color = 'blue', fill = 'purple', binwidth = 1)

p14 <- tbl_mz %>%
  ggplot(aes(log10(int))) +
  geom_density(color = 'red', fill = 'orange')
```

Colors can also take on a transparency called `alpha`, which allows one layer to show through when two or more are plotted together.

```
p15 <- tbl_mz %>%
  ggplot(aes(log10(int))) +
  geom_density(aes(fill = class))

p16 <- tbl_mz %>%
  ggplot(aes(log10(int))) +
  geom_density(aes(fill = class), alpha = .25)
```

Scales

GGplot2 is a popular data visualization package in R that allows users to create stunning and insightful visualizations. One of the key features of GGplot2 is its ability to handle scales, which are critical for displaying data accurately and effectively. In this document, we will explore how to use scales in GGplot2, specifically for log10 and manual scales.

Log10 Scale Logarithmic scales are useful when the data spans several orders of magnitude. GGplot2 makes it easy to create log10 scales using the `scale_y_log10()` and `scale_x_log10()` functions.

Using the same examples from above, yet instead of applying the `log10()` function directly to the variable, we can apply it to the scale instead.

```
p17 <- tbl_mz %>%
  ggplot(aes(int)) +
  geom_histogram(aes(fill = class), position = 'identity', alpha = .5, binwidth = 1) +
  scale_x_log10()

p18 <- tbl_mz %>%
  ggplot(aes(int)) +
  geom_density(aes(fill = class), alpha = .25) +
  scale_x_log10()
```

This results in a plot where the y-axis is scaled logarithmically, making it easier to see the differences between the different car classes. Note, that when we specified the `binwidth = 1` in the `geom_histogram()`, GGplot2 applied that to the log10 space specified from the `scale_x_log10()`.

Manual Scales Sometimes, we may want to manually define the scale for our plots. For example, we may want to create a plot where the y-axis only shows

values between 0 and 10. We can do this using the `scale_y_continuous()` function in GGplot2.

Here is an example of how to use the `scale_y_continuous()` function to manually define the y-axis scale:

```
p19 <- tbl_mz %>%
  ggplot(aes(mz, int)) +
  geom_segment(aes(xend = mz, yend = 0)) +
  scale_y_continuous(n.breaks = 13) +
  scale_x_continuous(n.breaks = 5, limits = c(500,1500))
```

In this example, we added a manual scale to both the x- and y-axis using the `scale_x_continuous()` and `scale_y_continuous()` functions, respectively, and specifying the number of breaks `n.breaks =` and the limits `limits =`.

Faceting Faceting is a powerful feature in ggplot2 that allows us to split a single plot into multiple small plots based on a categorical variable. It enables us to visualize complex data patterns and relationships in a more understandable way. There are two types of faceting in ggplot2: facet_wrap and facet_grid.

facet_wrap `facet_wrap` creates a grid of plots by wrapping the facets from left-to-right and top-to-bottom in the plot. Each facet is displayed in a separate panel, and the panels are arranged in rows and columns based on the levels of the specified categorical variable.

```
p20 <- p17 + facet_wrap(. ~ class)

p21 <- p17 + facet_wrap(. ~ class, scales = 'free')
```

In this example, we are reused ggplot object `p17` and created two additional plots. The `facet_wrap` function is used to split the plot into multiple panels based on the categorical variable `class` using the tilde `. ~ class`. In this case, the dot `.` prior to the tilde `~` tells ggplot to consider only a single variable, `class` as we had defined it. You can think of the tilde as a type of function `this 'by' that` or `y ~ x`. This becomes more important in the `facet_grid()` function. Notice in the `p21` plot we set the scales `free`, allowing each facet to dictate `x` and `y` plot scales.

facet_grid `facet_grid` creates a grid of plots by specifying one or more categorical variables that define the rows and columns of the grid. It allows us to create more complex faceted plots than `facet_wrap`. In this example we will randomly add a new varaible called `group` that will allow us to create the `y` direction of the facet.

```
tbl_new <- tbl_mz %>%
  mutate(group = sample(c('positive', 'negative'), n_peaks, replace = TRUE))

p22 <- tbl_new %>%
  ggplot(aes(int)) +
  geom_histogram(binwidth = 1) +
  scale_x_log10() +
  facet_grid(group ~ class)
```

To illistrate the difference between `facet_wrap()` and `facet_grid()` consider what happens when a set of data is missing. Note in `p24` it is not immediately intuitive in `facet_wrap()` that `c-negative` is missing, where as in `facet_grid()`, the layout highlights this realization.

```
tbl_new <- tbl_mz %>%
  mutate(group = sample(c('positive', 'negative'), n_peaks, replace = TRUE))

w <- which(tbl_new$class == 'C' & tbl_new$group == 'negative')

p23 <- tbl_new[-w, ] %>%
  ggplot(aes(int)) +
  geom_histogram(binwidth = 1) +
  scale_x_log10()

p24 <- p23 + facet_wrap(group ~ class)
p25 <- p23 + facet_grid(group ~ class)
```

Labels

Annotations

Style

Creating a plotting style can help you to quickly improve the appearance of your plots and make them more consistent with your brand. When working with data visualization, it's important to keep in mind that the appearance of your plots can significantly impact the way your audience interprets your data. GGplot2 themes and colors offer an easy way to create professional-looking visualizations that will make your data stand out.

Themes To apply a theme to your plot, you simply need to call the `theme()` function and specify the name of the theme you want to use. Some of the most popular themes include:

- `theme_gray()`: A simple, gray background with white gridlines.
- `theme_dark()`: A simple, gray background with white gridlines.
- `theme_classic()`: A classic black and white theme with no gridlines.
- `theme_minimal()`: A minimalistic theme with no background or gridlines.
- `theme_bw()`: A black and white theme with gray gridlines.

You can also create your own custom themes by modifying various theme elements. For example, you can change the background color, font, and size of the plot elements. To do this, you can use the `element_*()` functions. For example, the `element_text()` function allows you to modify the font size, color, and family of your text.

Another great feature of GGplot2 themes is that they allow you to maintain consistency across multiple visualizations. If you're creating a series of plots, applying the same theme to each one will give your work a more polished and professional look.

```
p28 <- p12 + theme_gray() # default
p29 <- p12 + theme_dark()
p30 <- p12 + theme_light()
p31 <- p12 + theme_classic()
p32 <- p12 + theme_minimal()
p33 <- p12 + theme_bw()
```

Colors In addition to applying a theme to your layout, you can should also consider the color scheme. GGplot2 is a powerful data visualization package in R that allows users to create beautiful and informative graphs. The package is highly customizable, and one of its most important features is the ability to customize colors using Brewer and manual color scales.

Brewer Color Scales The Brewer color scales in GGplot2 are color palettes that have been specifically designed to be distinguishable by people with color vision deficiencies. These color scales are useful when creating visualizations where color is used to convey information.

The Brewer palettes are particularly useful because they are carefully curated to ensure that the colors are distinguishable from one another, even for individuals with color vision deficiencies. This makes them a great option for creating informative data visualizations.

To use Brewer color scales in GGplot2, you can simply specify the name of the color scale as an argument to the `scale_color_brewer()` or `scale_fill_brewer()` functions. Other popular Brewer color scales include Blues, Greens, Oranges, and Purples. By using these scales, you can create beautiful visualizations that are both aesthetically pleasing and informative.

Manual Color Scales In addition to the Brewer color scales, GGplot2 also allows users to specify custom color scales using the `scale_color_manual()` or `scale_fill_manual()` functions. These functions take a vector of colors as an argument, which can be specified using names, hex codes, or RGB values.

Manual color scales are particularly useful when you want to use specific colors that are not included in the Brewer palettes. For example, if you are creating a visualization for a company and you want to use the company's brand colors, you can specify the colors using a manual color scale.

```
p34 <- p31
p35 <- p31 + scale_fill_brewer(palette = 'Set1')
p36 <- p31 + scale_fill_brewer(palette = 'Blues')
p37 <- p31 + scale_fill_manual(values = c("#d97828", "#83992a", "#995d81","#44709d"))
```

Package ggrepel

8.3 Alternatives

lattice

The Lattice package is an R package that is used for plotting graphs, and is based on the grid graphics system. The package provides a high-level interface to grid graphics, which makes it easy to create complex visualizations with an emphasis on multivariate data. It is designed to meet most typical graphics needs with minimal tuning, but can also be easily extended to handle most nonstandard requirements.

Trellis Graphics, originally developed for S and S-PLUS at the Bell Labs, is a framework for data visualization developed by *R. A. Becker, W. S. Cleveland, et al, extending ideas presented in Cleveland's 1993 book Visualizing Data.* The Lattice API is based on the original design in S, but extends it in many ways.

Various types of lattice plots available for data visualization. Among the different types of plots, **univariate** plots stand out as they utilize only a single variable for plotting. The different options available for univariate plots include bar plots, box-and-whisker plots, kernel density estimates, dot plots, histograms, quantile plots, and one-dimensional scatter plots. **Bivariate** plots involve plotting two variables against each other. Examples of bivariate plots include scatterplots and quantile plots. These types of plots are useful in analyzing the relationship between two variables and can provide valuable insights into the data. **Trivariate** plots, as the name implies, involve plotting three variables and provide a more complex visualization of the data. Options for trivariate plots include level plots, contour plots, three-dimensional scatter plots, and three-dimensional surface plots. These types of plots can be particularly helpful in analyzing complex data sets and identifying patterns in the data that may not be immediately apparent.

```
library(lattice)

p51 <- xyplot(int ~ mz, data = tbl_mz, main = "Scatter Plot")
p52 <- xyplot(int ~ mz, data = tbl_mz, type='a', main = "Line Plot")
p53 <- histogram(~ log10(int) | class, data = tbl_mz, main = "Histogram")
p54 <- densityplot(~ log10(int) | class, data = tbl_mz, main = "Density Plot")
```

plotly

Plotly is an open-source data visualization library that allows you to create interactive visualizations in R plotly.com/r/. It offers a wide range of graphs and charts, including line plots, scatter plots, area charts, bar charts, error bars, box plots, histograms, heatmaps, subplots, plots with multiple-axes, 3D plots, and more. The package is built on top of **htmlwidgets**, which means that you can easily embed your visualizations in web applications or other HTML documents.

Plotly also nativity supports many data science languages such as R, Python, Julia, Java-script, MATLAB and F#.

```
library(plotly)
```

```
plot_ly(tbl_mz, x = ~mz, y = ~int, color = ~class,
        mode = "markers", type = "scatter") %>%
  layout(title = 'Scatter Plot')
```

Exercises

- Create a new R Studio Project and name it **006__data__visualizing**.
- Create a new R script, add your name and date at the top as comments.
- Locate and/or download a GGplot2 cheat-sheet and refer to it as needed.

1. If not already done, download *Bacterial Metabolite Data* to use as an example data file.

```
url <- "https://raw.githubusercontent.com/jeffsocal/ASMS_R_Basics/main/data/bacterial-metabolites
download.file(url, destfile = "./data/bacterial-metabolites_dose-simicillin_tidy.csv")
```

2. Read in the dataset .csv using the `tidyverse` set of packages.

3. Create a metabolite `abundance` by `time_min` ...

4. ... facet by `culture` and `metabolite`...

4. ... adjust the y-axis to log10, color by `dose_mg`, and add a 50% transparent line ...

5. ... change the theme to something publishable, add a title, modify the x- and y-axis label, modify the legend title, adjust the y-axis ticks to show the actually measured time values, and pick a color scheme that highlights the dose value...

Chapter 9

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.

At the end of this chapter you should be able to

- Understand the options for sharing analyses.
- Start an R Notebook project to share.

9.1 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:

9.2 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.

9.3 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.

9.4 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.

9.5 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:

```
#' 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:

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

This will output “Hello ASMS” to the console.

9.6 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.

Chapter 10

Mass Spectrometry

At the end of this chapter you should be able to

- Understand commercially proprietary raw data.
- How to convert raw data to an open access format.
- Be familiar with some R packages specific to mass spectrometry.

10.1 Data Formats

RAW (Thermo)

WIFF (Sciex)

D (Agilent)

D (Bruker)

10.2 Open Access 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.

ProteoWizard is an open-source software suite 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.
>
> 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.*

10.3 R Calculate Mass

Molecular mass is the sum of the atomic masses of all the atoms in a molecule. It is an important parameter used in various fields of chemistry. The molecular mass of a molecule is usually expressed in atomic mass units (amu) or daltons (Da). Mass spectrometry based measurements require a charge and are expressed as mass-to-charge (m/z) or as Thompsons (Th).

Isotopic probabilities are also important in determining molecular mass. Isotopes are atoms of the same element that have different numbers of neutrons in their nuclei. Isotopes of an element have different atomic masses. The isotopic probability of an element is the probability that a given isotope of that element will occur in nature. For example, carbon (C) has two stable isotopes, carbon-12 (12C) and carbon-13 (13C), with atomic masses of 12.000 amu and 13.003 amu, respectively. The isotopic probability of carbon-12 is 98.9%, while that of

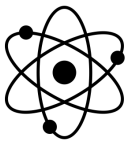
carbon-13 is 1.1%. Therefore, the average atomic mass of carbon in nature is 12.011 amu.

There are several measures of a molecular mass that can be reported. The *average mass* is that of the weighted average of all isotopes and likely to be reported for small molecules. While the *nominal mass* of a molecule is defined as the sum of the integer masses of the most abundant isotopes in a molecule. The *monoisotopic mass* is commonly considered the as the sum of the exact masses of the lightest isotopes, and this value is considered in all peptide-based proteomics applications.

BRAIN

Baffling Recursive Algorithm for Isotope distributioN calculations

R BRAIN is an isotopic abundance calculator implemented in R programming language and is especially useful for chemists and researchers who deal with complex molecules and need to calculate their isotopic composition accurately. In addition, it has a handy function for calculating the mass directly from an amino acid sequence.



Documentation | Bioconductor
Literature | *Analytical chemistry* 2013 85(4), 1991-1994

Installation

```
BiocManager::install("BRAIN")
```

Use

```
library(BRAIN)

# Human insulin amino acid sequence
str_seq <- "MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAEDLQVGQVELGGGPGA
```

```
# get a list-object of atoms
lst_atm <- getAtomsFromSeq(str_seq)
```

Calculate the average mass.

```
calculateAverageMass(lst_atm)
```

```
## [1] 11980.82
```

Calculate the monoisotopic mass.

```
calculateMonoisotopicMass(lst_atm)
```

```
## [1] 11973.02
```

Calculate the isotopic abundances (probabilities) mass.

```
lst_isotopes <- useBRAIN(lst_atm, nrPeaks = 20)

plot(lst_isotopes$masses, lst_isotopes$isoDistr, xlab='Mass', ylab='Probability', type = 'h')
```

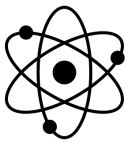
Calculate the isotopic abundances (probabilities) mass for a metabolite [C100H200S2Cl5].

```
lst_atm <- list(C=100, H=200, S=2, Cl=5)

lst_isotopes <- useBRAIN(lst_atm, nrPeaks = 13)

plot(x = lst_isotopes$masses,
     y = lst_isotopes$isoDistr / max(lst_isotopes$isoDistr) * 100,
     xlab='Mass', ylab='Relative abundance', type = 'h')
```

10.3.1 enviPat

Isotope Pattern, Profile and Centroid Calculation for Mass Spectrometry

A new method (year 2022) for calculating theoretical isotope patterns in mass spectrometry. This method uses a treelike structure to derive sets of subisotopologues for each element in a molecule. By doing so, it allows for early pruning of low-probability isotopologues and the detection of the most probable isotopologue. The method was validated in a large-scale benchmark simulation.



Documentation | CRAN
Literature | *Anal. Chem.* 2015, 87, 11, 5738–5744

Installation

```
install.packages("enviPat")
```

Use

```
library(enviPat)

data("isotopes")

pattern <- isopattern(
  isotopes,
  "C100H200S2Cl5",
  threshold=0.1,
  plotit=TRUE,
  charge=FALSE,
  emass=0.00054858,
  algo=1
)
```

10.4 R MS Data

xcms

mzR

10.5 R MS Analysis

Finally, there are numerous R packages that implement methods, common and advanced, in statistical post analysis of quantitative data. Explored here are only a few of the available packages.

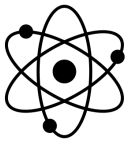
deqms

msempire

msqrob

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

Installation

Install MSstats from Bioconductor:

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

Loading Data

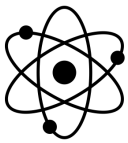
The preferred data structure for use in MSstats is a .csv file in a "long" format with 10 columns representing the following variables: *ProteinName, PeptideSequence, PrecursorCharge, FragmentIon, ProductCharge, IsotopeLabelType, Condition, BioReplicate, Run, and Intensity.* Note that the variable names are fixed and case-insensitive.

```
# example data provided by the MSstats package
head(SRMRawData)
```

```
    ProteinName PeptideSequence PrecursorCharge FragmentIon ProductCharge IsotopeLabelType Condit
243        IDHC   ATDVIVPEEGELR               2          y7            NA                H
244        IDHC   ATDVIVPEEGELR               2          y7            NA                L
```

```
245        IDHC   ATDVIVPEEGELR              2          y8           NA
246        IDHC   ATDVIVPEEGELR              2          y8           NA
247        IDHC   ATDVIVPEEGELR              2          y9           NA
248        IDHC   ATDVIVPEEGELR              2          y9           NA
    BioReplicate Run   Intensity
243        ReplA   1 84361.08350
244        ReplA   1   215.13526
245        ReplA   1 29778.10188
246        ReplA   1    98.02134
247        ReplA   1 17921.29255
248        ReplA   1    60.47029
```

StatsPro R Package

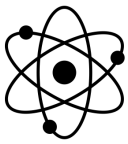


Documentation | github, PDF
Code | github
Literature | *Journal of Proteomics* 250 (2022): 104386.

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.
Literature | *Molecular & Cellular Proteomics*, **19**(7), 1209-1219.
Literature | *Analytical Chemistry*, **92**(9), 6278–6287.

Installation

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

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

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)
```

Estimating Covariance Matrix

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

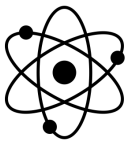
```
# Loading Example Data
data("exMiss")

# 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.

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.



Documentation | github
Code | github
Web App | tidyproteomics
Literature

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.

Installation

To install the `tidyproteomics` package, you will need to install GitHub and Bioconductor repositories:

```
install.packages("devtools")
devtools::install_github("jeffsocal/tidyproteomics")

install.packages("BiocManager")
BiocManager::install(c("limma","qvalue","fgsea","Biostrings"))
```

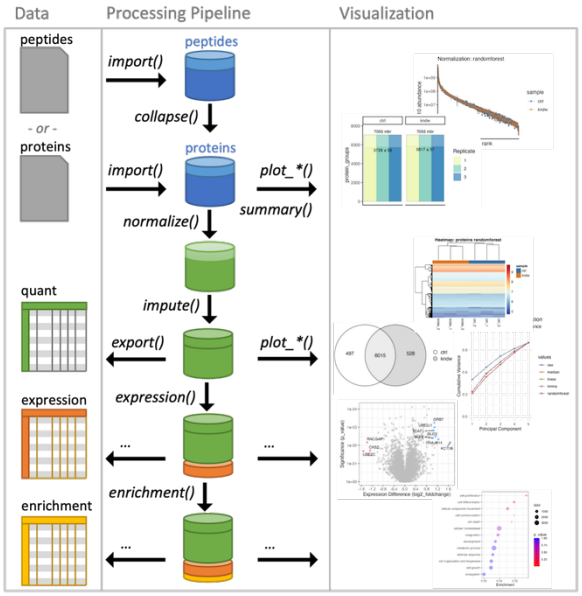


Figure 10.1: tidyproteomics workflow

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")
```

Data Summaries

Currently, tidyproteomics implements two summary quantitative visualizations. The first is a simple grouped bar chart that displays individual and grouped proteins, as well as all and unique peptides. The match-between-runs is shown as a margin above the MS2 evidenced identifications. In recent literature, a summary of protein quantitation has been visualized as a rank-based dot plot. This plot can be extended to highlight statistical differences via an unbiased all-pair-wise comparison, which gives an anticipated view on how to guide downstream analyses.

```
p01 <- data_proteins %>% plot_counts()
p02 <- data_proteins %>% plot_quantrank()
```

Summary Stats {-} Summarizing proteomics data is vital to understanding the bigger picture and conveying summary stats that set the tone for the larger analysis. The results of each summary can be directed to via the destination option to "print" on screen, "save" to a file or "return" as a tibble.

```
data_proteins <- data_proteins %>%
  # 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")
```

Normalization and Imputation

Quantitative proteomics requires accurate normalization, which can be difficult to implement. The `normalize()` function in the tidyproteomics package is a wrapper for various normalization methods, while `select_normalization()` automatically selects the best method based on a weighted score. Both functions allow for downstream analyses such as `expression()` and `enrichment()`. The package attempts to apply each function universally to peptide and protein values using the `identifier` variable to identify the thing being measured.

```
data_proteins <- data_proteins %>%
  # normalize via several methods, best method will be automatically selected
  normalize(.method = c("median","linear","limma","randomforest")) %>%
  # impute with a minimum value (this is a knock-out)
  impute(base::min)

# plot visualizations comparing normalization methods
p03 <- data_proteins %>% plot_normalization()
p04 <- data_proteins %>% plot_variation_cv()
p05 <- data_proteins %>% plot_variation_pca()
p06 <- data_proteins %>% plot_dynamic_range()
```

```
# plot visualizations of unbiased clustering
p07 <- data_proteins %>% plot_heatmap()
p08 <- data_proteins %>% plot_pca()
```

Expression Analysis

```
data_proteins <- data_proteins %>%
  # 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")
```

Overall, the tidyproteomics package provides a useful set of tools for preprocessing and analyzing proteomics data using the tidy data framework in R. There are several more workable examples in the online documentation.

Answers to Exercises

4 The Basics

```r
# John Doe
# 2023-06-02
# Institution Inc.
#
# Some basic R practice

# 3.  Calculate the sum of 2 and 3.
2 + 3

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

0.5 == 1 / 2

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

3 %% 2 == 0

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

even <- function(x) { x %% 2 == 0 }

even(3)

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

oddeven <- function(x) {
  if(x %% 2 == 0) return('even')
  else return('odd')
}
```

```
oddeven(3)
```

5 Data Structures

```
# John Doe
# 2023-06-02
# Institution Inc.
#
# Data Structures Exercises

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

str_gbu <- c('red', 'green', 'blue')
print(str_gbu)


# 2.  Extract the 2nd element in the variable.

print(str_gbu[2])


# 3.  Construct a numerical vector of length 5, containing the diameters of
#     circles with integer circumferences 1 to 5. Remember PEMDAS.

cir <- (1:5) ^ 2 * pi


# 4.  Extract all circumferences greater than 50.

cir[which(cir > 50)]
```

6 Tidyverse

```
# John Doe
# 2023-06-02
# Institution Inc.
#
# Tidyverse Exercises

# 1. Pipeline (eg. `%>%`) a data operation that provides the mean of following vector.

c(7.48, 14.15, 6.23, 10.21, 15.13, 8.19, 8.58, 8.09, 9.14, 10.41) %>% mean()
```

```r
# 2.  Employing a pipeline (eg. `%>%`), construct a tibble with columns named
#     `circ` and `diam` which contains the diameters of circles with integer
#     circumferences 1 to 5. Remember PEMDAS.

library(tidyverse)

tbl_cir <- tibble(
  circ = 1:5
) %>%
  mutate(diam = circ ^ 2 * pi)

tbl_cir


# 3.  Extract all circumferences greater than 50.

tbl_cir %>% filter(diam > 50)


# 4. Add a column named `circ_type` where you assign the string *odd* or *even*
#     depending on the column `circ`. Attempt to use the `purrr::map` function,
#     along with the `oddeven()` function from the previous chapter, then compute
#     the mean, standard deviation, and coefficient of variation of the diameters
#     for each `circ_type`.

oddeven <- function(x) {
  if(x %% 2 == 0) return('even')
  else return('odd')
}

tbl_cir %>%
  mutate(circ_type = map(circ, oddeven)) %>%
  mutate(circ_type = unlist(circ_type)) %>%
  group_by(circ_type) %>%
  summarise(
    diam_mean = mean(diam),
    diam_sd = sd(diam),
    diam_cv = diam_sd / diam_mean
  )
```

7 Data Wrangling

```
# John Doe
# 2023-06-02
# Institution Inc.
#
# Data Wrangling Exercises

# 1. Download *Denver Climate Data* to use as an example data file.

url <- "https://raw.githubusercontent.com/ZenBrayn/asms-2022-r-shortcourse/main/exercis
download.file(url, destfile = "./data/denver_climate.csv")


# 2. Read in the dataset .csv using the `tidyverse` set of packages.

tbl_den <- "./data/denver_climate.csv" %>% read_csv()


# 3. How many rows does the table contain? Number of columns? What are the column name

tbl_den %>% nrow()
tbl_den %>% ncol()
tbl_den %>% colnames()


# 4. Find the month and year with the highest snowfall.

tbl_den %>%
  filter(snowfall == max(snowfall)) %>%
  select(month, year)


# 5. Find months where there was no snowfall.

tbl_den %>%
  filter(snowfall == 0) %>%
  select(month) %>%
  unique()


# 6. Find average precipitation per year.

tbl_den %>%
  group_by(year) %>%
  summarise(precipitation_avg = mean(precipitation))
```

```
# 7. Convert temperatures to Celsius and add new columns with these to table.
#    Reminder `F to C conversion -> (F - 32) * (5/9)`.

tbl_den <- tbl_den %>%
  mutate(min_temp_C = signif((min_temp - 32) * (5/9), 3),
         max_temp_C = signif((max_temp - 32) * (5/9), 3))

tbl_den


# 8. Find the month with the greatest difference in C between max and min temperature.

tbl_den %>%
  mutate(temp_diff_C = max_temp_C - min_temp_C) %>%
  group_by(month) %>%
  summarise(temp_diff_C_avg = mean(temp_diff_C)) %>%
  filter(temp_diff_C_avg == max(temp_diff_C_avg))


# 9. Write to file an .csv of the data.frame which contains the C data.

tbl_den %>% write_csv("data/denver_climate.csv")
```

8 Data Visualization

```
# John Doe
# 2023-06-02
# Institution Inc.
#
# Data Visualization Exercises

# 1. If not already done, download *Denver Climate Data* to use as an example data file.

url <- "https://raw.githubusercontent.com/ZenBrayn/asms-2022-r-shortcourse/main/exercises/intro_t
download.file(url, destfile = "./data/denver_climate.csv")


# 2. Read in the dataset .csv using the `tidyverse` set of packages.

tbl_den <- "./data/denver_climate.csv" %>% read_csv()


# 3. Plot precipitation ~ snowfall ...
```

```
tbl_den %>%
  ggplot(aes(precipitation, snowfall)) +
  geom_point()


# 4. ... color by `min_temp`, creating a color scheme of low = "lightblue" and
#    high = "orange", mid = "yellow"...

tbl_den %>%
  ggplot(aes(precipitation, snowfall)) +
  geom_point(aes(color = min_temp)) +
  scale_color_gradientn(colors = c('lightblue','yellow', 'orange'))


# 5. ... add a smoothed line under the points colored grey, facet by `year`
#    keeping all plots on a single line, and change the theme to have a dark
#    background ...

tbl_den %>%
  ggplot(aes(precipitation, snowfall)) +
  geom_smooth(method = lm, color = 'grey', fill = NA) +
  geom_point(aes(color = min_temp)) +
  scale_color_gradientn(colors = c('lightblue','yellow', 'orange')) +
  facet_wrap(~year, nrow = 1) +
  coord_cartesian(ylim = c(0, 20)) +
  theme_dark()


# 6. Using the same data, construct a plot showing the distribution of min_ and
#    max_temp for each month such that min_ and max_ are plotted separately for
#    each month, add point values on top, gradient colored as above. Substitute
#    the month number for the month 3-letter string (hint, use lubridate), remove
#    the legend, remove the grid lines, remove the x-axis labels, add a title and
#    fix the y-axis title. Use the internet to find answers if you need to.

library(glue)

tbl_den %>%
  mutate(month = lubridate::month(month, label = TRUE, abbr = TRUE)) %>%
  pivot_longer(cols = c('min_temp', 'max_temp'), names_to = 'temp_ends', values_to = '
  ggplot() +
  geom_boxplot(aes(month, temp, group = temp_ends), fill=NA, color='grey40', position =
  geom_point(aes(month, temp, group = temp_ends, color = temp)) +
  facet_wrap(~month, nrow = 1, scales = 'free_x') +
  scale_color_gradientn(colors = c('lightblue','yellow', 'orange')) +
```

```
  labs(title = "Monthly Temperatures for Denver",
       subtitle = glue("{min(tbl_den$year)} - {max(tbl_den$year)}"),
       y = "Temperature (C)") +
  theme_classic() +
  theme(
    legend.position = 'none',
    axis.text.x = element_blank(),
    axis.title.x = element_blank(),
    axis.ticks.x = element_blank()
  )
```