R for Environmental Chemistry

David Hall, Steven Kutarna, Kristen Yeh, Hui Peng, Chaerin Song, and Jessica D'eon

Last built on: 2024-02-11

Contents

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

Preface

Howdy,

This website is more-or-less the living result of a collaborative project between us. We're not trying to be an exhaustive resource for all environmental chemists. Rather, we're focusing on developing broadly applicable data science course content (tutorials and recipes) based in `R` chemistry courses and research.

This book will is broken up into four sections:

- **Section 1: Getting Started in R** is a general guide for the complete novice that will help you install, setup, and run R code.
- **Section 2: How to Code in R** introduces the basics of R programming as well as a usual R workflow, and how to use R markdown to communciate your code with others.
- **Section 3: Data Wrangling** introduces data analysis workflows and showcases *how* you can use R and the `tidyverse` to import and clean up your data into a consistent format to tackle the vast majority of the data science/analysis problems you'll encounter in undergraduate environmental chemistry courses.
- **Section 4: Data Analysis Toolbox** provides code and theory behind the most common data analysis practices in environmental chemistry. These include linear regression analysis, a myriad of visualizations, etc.
- **Section 5: Notes for Env. Chem. Labs** consist of chapters specific to individual laboratory experiments. They rely upon knowledge from the previous three sections to introduce concepts unique to individual labs.

We recommend that you read through Sections 1 and 2 in sequential order. These provide the foundation for the consistent data analysys workflow used throughout Sections 3 and 4.

Providing Feedback

If you notice an error, mistake or if you have suggestions for adding features or improving the book, please reach out to us or flag an issue on GitHub.

- Jessica D'eon at jessica.deon@utoronto.ca

Acknowlegements

Additionally, we would like to thank Jeremy Gauthier, Andrew Folkerson, Mark Panas, and Stephanie Schneider for all of their comments, suggestions, and hard work integrating the concepts of this book into the *CHM410* Laboratory curriculum.

Part 1: Getting Started in R

Chapter 1

Intro to R and RStudio

You may have heard about the coding or the R programming language, but figuring out how to get started can be a hurdle; at least it was for us. In this guide, we will walk you through the process of setting up R and RStudio, both locally on your computer and remotely using the University of Toronto's JupyterHub R Studio server.

1.1 R Language

R is the programming language we'll code in. R is hosted on the Comprehensive R Archive Network (CRAN) and is one of the most popular programming languages for statisticians and scientist alike due to its vast array of tools and packages.

A quick aside, but don't be intimidated by the term "coding". Coding is simply writing instructions for the computer to execute. The only catch is has to be in a language that both we, humans, and the computer can understand. For our needs we're using R, and like any language, R has it's own syntax, rules, and quirks which we'll cover in later chapters.

1.2 RStudio

RStudio is a popular *integrated development environment (IDE)* specifically designed for working with R, providing a user-friendly interface and various productivity features. It's where you'll actually be typing your code and interacting with R. Again, R is a language, and you need somewhere to write it down to make use of it. Writing in English can be done with a pencil and notepad or a word processor filed with useful tools to help you write.

R and RStudio work in tandem to provide an efficient and seamless experience

for data analysis, visualization, and model building. RStudio enhances the R workflow with features like code editing, interactive visualization, version control, and package management.

1.3 Setting Up Your Environment

Students learning R have **two options**: working *locally* or *remotely*

Working locally involves installing R and RStudio on their computer, providing direct control over data and code without an internet connection. On the other hand, *working remotely* enables access to RStudio through a web browser, avoiding local installations and allowing collaboration. **We recommend working remotely**, leveraging platforms like the University of Toronto's JupyterHub for its convenience and stable R Studio environment, making learning R easier and more efficient. We will go into more details in the below paragraphs.

1.3.1 Working Remotely (Recommended)

Working remotely means accessing R and RStudio from a remote server or cloud-based platform.

UofT JupyterHub RStudio server

To facilitate remote access to RStudio, the University of Toronto provides a JupyterHub R Studio server. This allows you to access RStudio from any web browser, eliminating the need for local installations. With this, you can perform data analysis, collaborate with others, and work on your R projects remotely with ease.

To get started, visit UofT JupyterHub. You will need to log in with your UofT credentials to access the RStudio environment.

- While working remotely, you may need to upload data to the RStudio server or download analysis results. The RStudio interface allows you to upload files directly from your computer to the server and vice versa.
- When working remotely, ensure that you save your R scripts and analysis files on the server. This will allow you to continue your work from any device with internet access.
- Most R packages are pre-installed on the University of Toronto's R Studio server. However, if you require additional packages, we will soon learn how to install packages.

Remember that while working remotely, a stable internet connection is essential to ensure a smooth and uninterrupted experience. Additionally, always remember to save your work and log out properly after each session to maintain the security of your data. Happy coding!

1.3.2 Working Locally

When you work locally, you need to install both R and RStudio on your personal computer or a machine that you physically have access to.

1.3.2.1 Downloading R and RStudio

You can download the latest build of **R** for your operating system here. Choose the appropriate version for your operating system (Windows, macOS, or Linux) and follow the installation instructions.

You can download the latest version of **RStudio** here.

Once you have both R and RStudio downloaded, go ahead and open up RStudio.

1.4 Using RStudio

When you open your RStudio (either locally or remotely), you'll be greeted with an interface divided into numerous panes. We've highlighted the major ones in the image below:

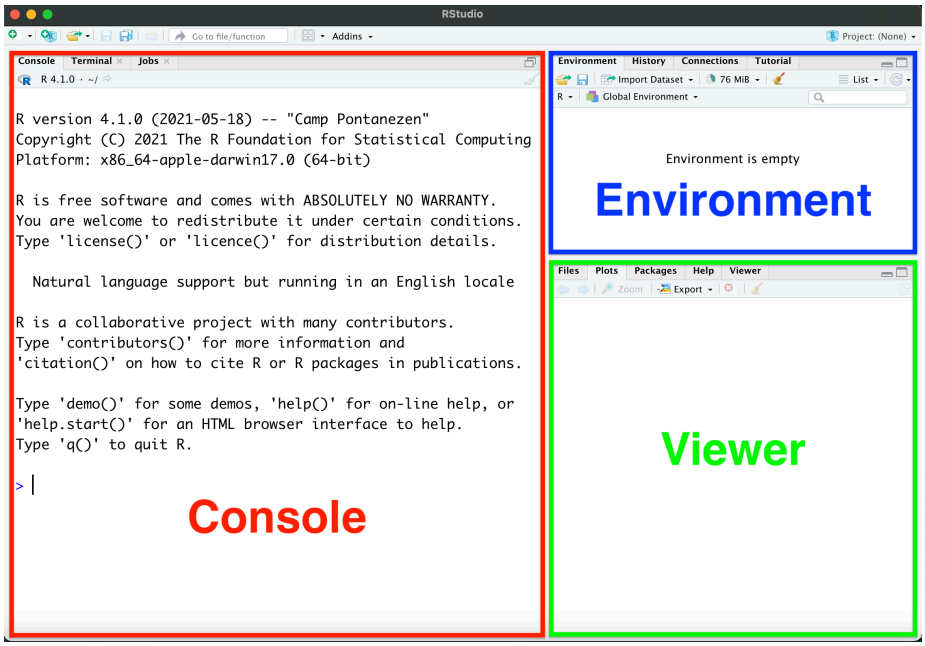


Figure 1.1: The RStudio interface with annottated regions

Each pane serves a specific role:

- **The console** allows you to directly type and run your code. It also provides messages, warnings, and errors from any code you run.
- **The environment** window lists all variables, data, and functions you've created since the start of your coding session.

- **The viewer** shows your outputs, help documents, etc. which each has their own tab.

1.5 Running R Code

As we've already seen, you can run bits of R code directly from the console. Throughout the book, code you can copy and run will look like this:

```
2 + 2
```

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

Noticed that both the code (the first part) and what the code outputs (the second part) are shown. Throughout this book code outputs will be proceeded by `##`. You can run code directly from the console. It's handy for short and sweet snippets of code, something that can be typed in a single line. Examples of this is the `install.packages()` function, or to use R as a calculator:

```
2 * 3
```

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

```
pi * (10/2)
```

```
## [1] 15.70796
```

However, working like this isn't very useful Imagine printing a book one sentence at a time, you couldn't really go back and edit earlier work because it's already printed. That's why we write out code in *scripts*. *Scripts* are similar to recipes, in that they're a series of instructions that R evaluates from the top of the script to the bottom. More importantly, writing your code out in a script makes it *more readable* to humans (presumably this includes you). Don't undervalue the usefulness of legible code. Your code will evaluate in seconds or minutes whereas it may take you hours to understand what it does.

Let's open up a new script in RStudio by going to *File->New File->R Script*, or by clicking on the highlighted button in the image below.

This should open up a new window in the RStudio interface, as shown in the following image.

You can copy and paste the code above into the script, save it, edit it, etc. and ultimately run specific lines of code by highlighting them and pressing Ctrl+Enter (Cmd+Enter on Mac), or by clicking the "Run" button in the top right corner of the Scripts window. Whenever you copy code blocks from this

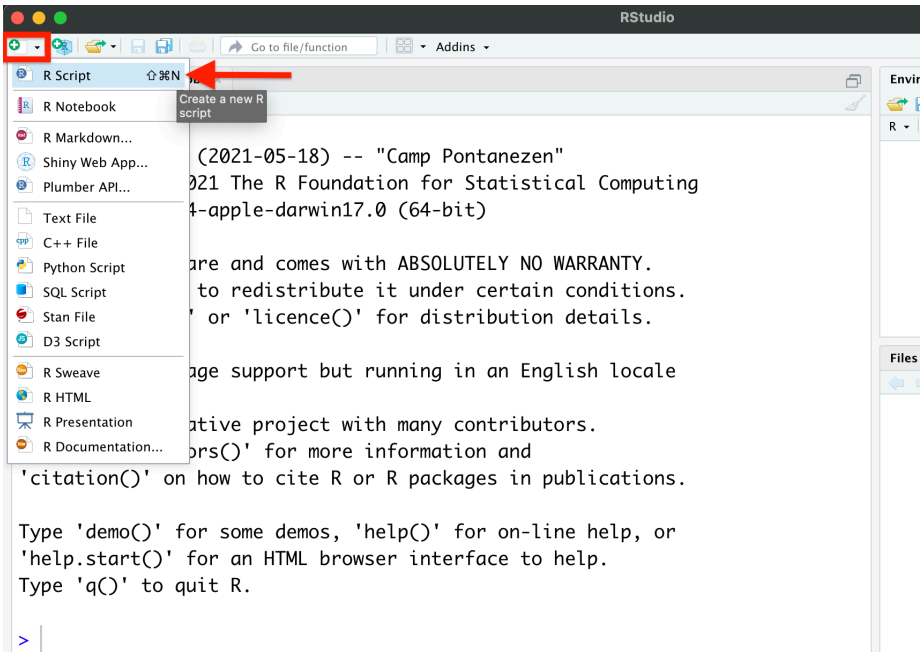


Figure 1.2: Figure 2.5: Opening a new script in RStudio.

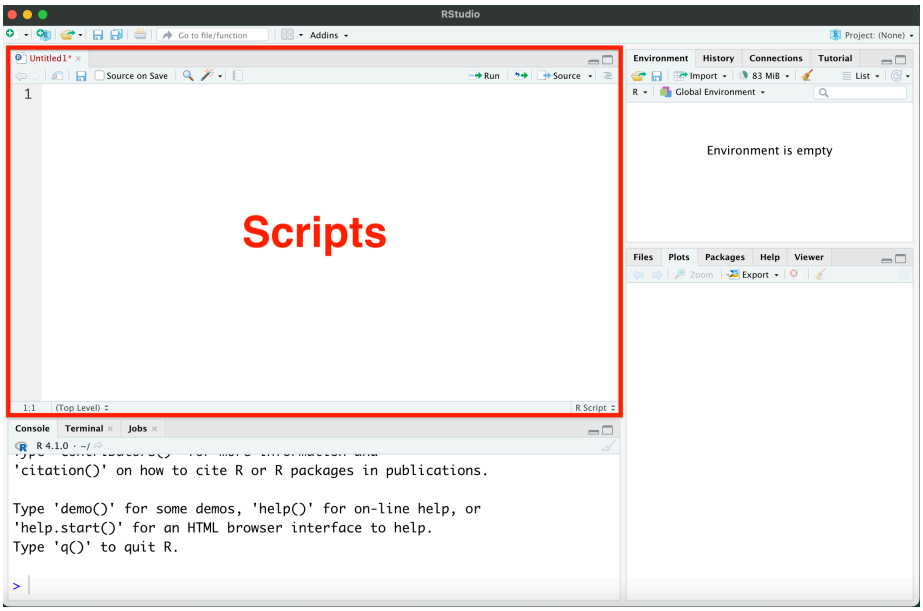


Figure 1.3: Figure 2.6: Scripts window in RStudio.

website (or other online sources). If you're reading this book online, you can easily copy an entire block of code using the `copy` button in the top right corner of the code block.

We'll dive into the basics of coding in R in the next chapter.

1.6 Customizing RStudio

As many of us spend an absurd amount of time staring at bright screens, some of you may be interested in setting your RStudio to Dark Mode.

You can customize the appearance of your RStudio interface by clicking *Tools->Global Options*, or *RStudio->Preferences* on Mac, then clicking "Appearance" on the left. Select your preferred Editor Theme from the list.

1.7 Where to get help

While it's often tempting to contact your TA or Professor at the first sign of trouble, it's better to try and resolve your issues on your own. Given the popularity of R, if you've run into an issue, someone else has too and they've complained about it and someone else has almost certainly solved it! An often unappreciated aspect of coding/data science is knowing *how* to get help, *how* to search for it, and *how* to translate someone's solutions to your unique situation.

Places to get help include:

- Google, Stack Overflow, etc. When in doubt Google it.
- Using built-in documentation (`?help`)
- reference books such as the invaluable *R for Data Science*, which inspired this entire project.
- And yes, when all else fails, holler at your TA/profs.

1.8 Summary

In this chapter we've covered:

- How to use RStudio to do R programming, both remotely and locally
- Installing the `tidyverse()` package, the basis of the subsequent code in this book
- How to customize the appearance of RStudio so you don't burn out your eyes at night

In the next chapter we'll break down how to setup your work in R for legibility, simplicity, and reproducibility. After all, the person cursing any of your sloppy work will invariably be you, so be kind to yourself, and do it right the first time.

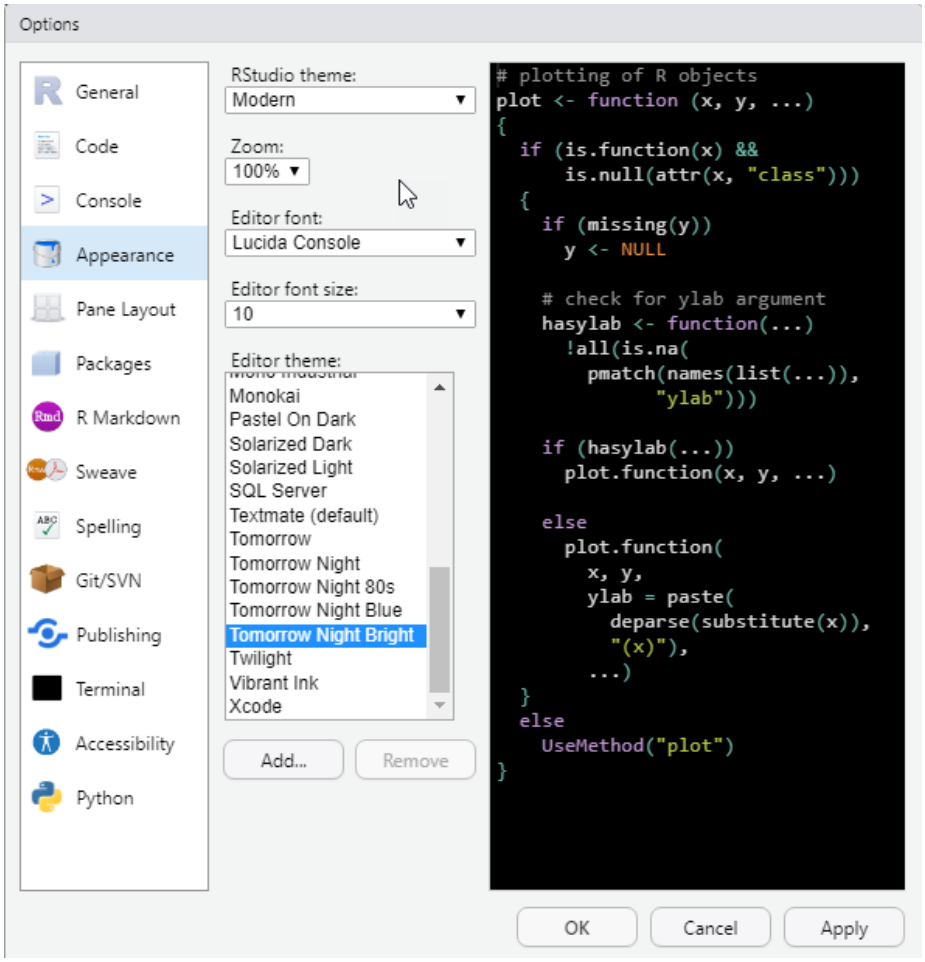
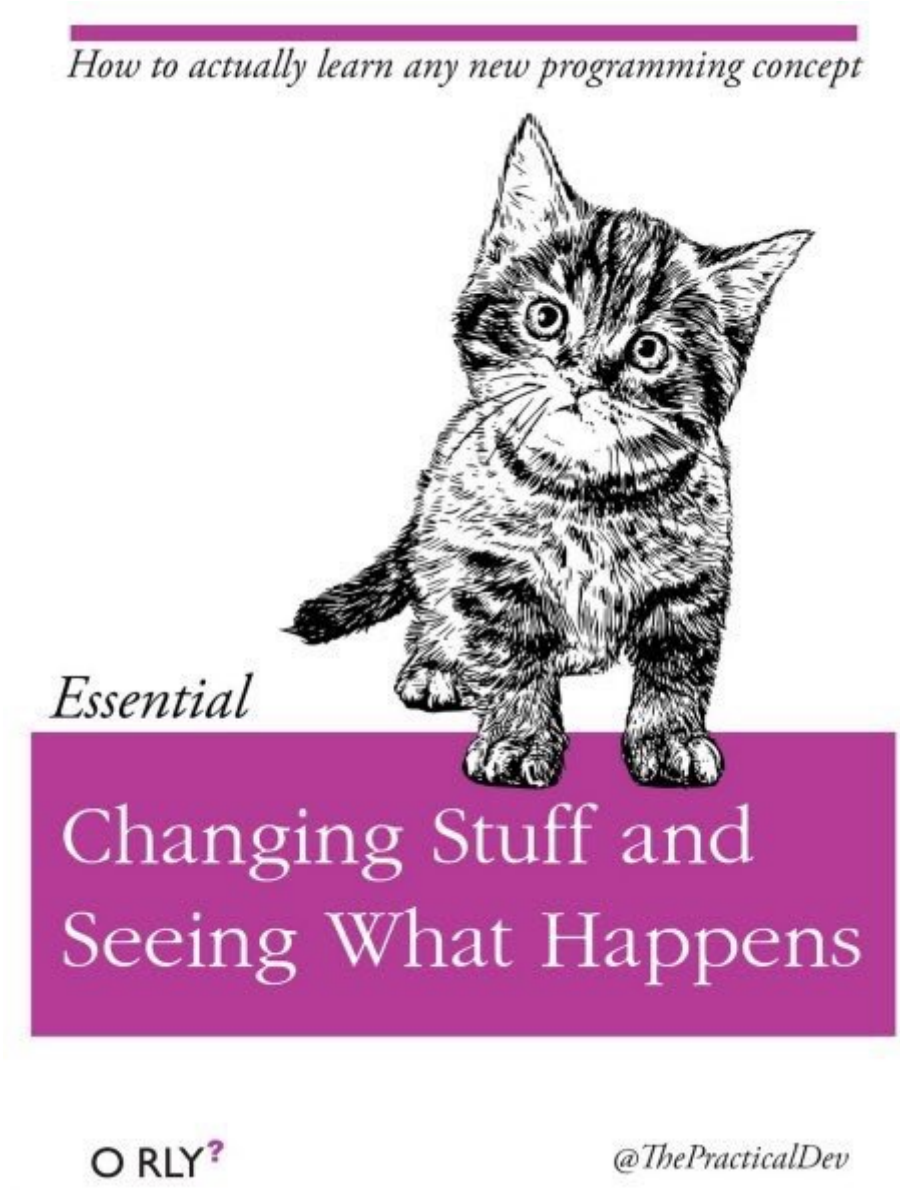


Figure 1.4: Figure 2.4: RStudio Appearance customization window.



1.9 Exercise

Now that you've learned the basics of setting up and customizing R and RStudio, let's put some of that knowledge into practice.

1.9.1 Setup

- Access UofT JupyterHub RStudio server here.
- (Optional) Change your RStudio appearance as you like.

1.9.2 Basic R Commands

- In the `Console` tab, write an expression to calculate 10 plus 5 and press `enter`.
- Open a new R script and type in the following commands:

```
x <- 10
y <- 5
z <- x + y
print(z)
```

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

(In the future, we will work with an R markdown instead of an R script, which we will explain more in the following chapters.)

- Run the script. What is the output?

1.9.3 USing the Help Function

- Let's say you've come across a function in R that you don't know how to use, for example, `sqrt()`. Use the `?` command to access the documentation for this function from your console tab.
- What does the `sqrt()` function do?

1.9.4 Reflection

- What are your first impressions of RStudio as an IDE? Do you have any prior experience with other programming languages or IDEs? If so, how does RStudio compare?

Chapter 2

RStudio Projects

You're probably eager to start coding, but an equally important aspect is understanding the structure of your work. Knowing how to organize the files needed for your analysis and how to access them quickly is critical. Learning this early on will save you plenty of time and heartache down the line. So let's hold off on coding and consider *where* we're working on your computer.

Because we believe in it so much, we'll say it up top: **Always work inside an RStudio Project, and use a unique project for each lab/experiment.**

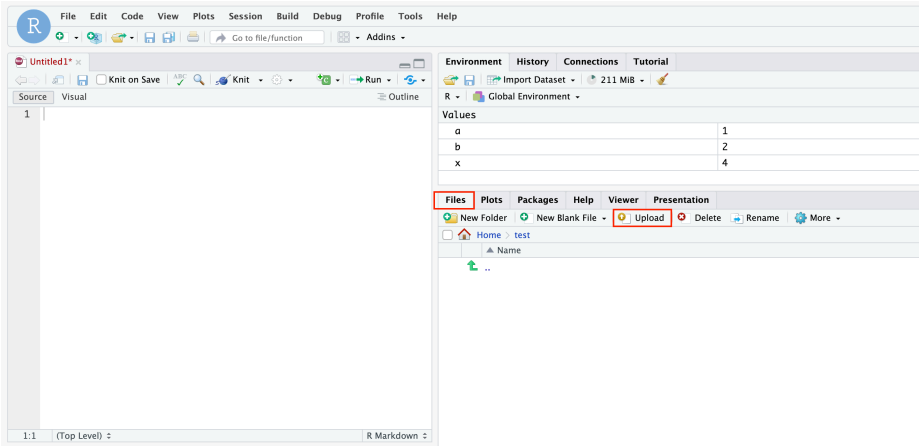
2.1 Uploading Files to RStudio server on JupyterHub

When using the RStudio Server instance provided through JupyterHub, you may want to upload local data files, R scripts, or other relevant resources to work with them directly in RStudio. Here's a straightforward guide on how to accomplish this.

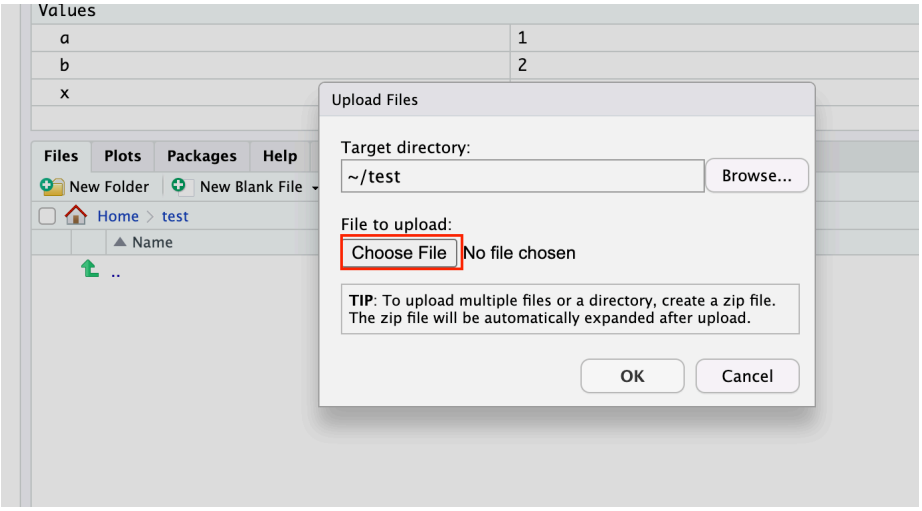
Step-by-step guide

- Once inside the RStudio server, you'll notice several panes. One of these is the `Files` pane, typically found in the bottom right corner. This pane displays the current directory's contents and allows you to manage files and folders.

- In the `Files` pane, locate and click on the `Upload` button.



- Then, click `Choose File` button to navigate to the location of the files on your local computer that you wish to upload to the RStudio Server.



- This will prompt a file dialog to appear. Select the desired file(s) and click on Open or Choose (depending on your browser).

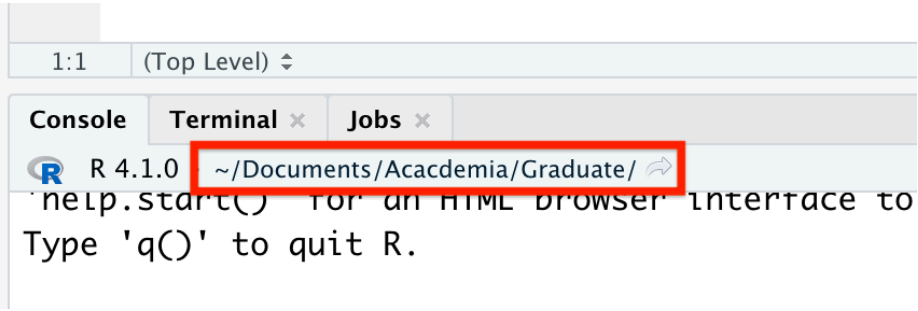
- Once the file names appear in the RStudio interface, there might be a confirmation step to complete the upload. Click on `OK` or `Upload` to finalize the process.

- After uploading, the uploaded files will appear in the `Files` pane.

2.2 Paths and directories

Before you get started with running your code, it is good to know where your analysis is actually occurring, or where your **working directory** is. The working directory is the folder where R looks for files that you have asked it to import, and the folder where R stores files that you have asked it to save.

RStudio displays the current working directory at the top of the console, as shown below, but can also be printed to the console using the command `getwd()`.



By default, R usually sets the working directory to the home directory on your computer. The `~` symbol denotes the home directory, and can be used as a shortcut when writing a file path that references the home directory.

You can change the working directory using `setwd()` and an absolute file path. Absolute paths are references to files which point to the same file, regardless of what your working directory is set to. In Windows, absolute paths begin with `"C:"`, while they begin with with a slash in Mac and Linux (i.e., `"/Users/Vinny/Documents"`). It is important to note that absolute paths and `setwd()` should **never** be used in your scripts because they hinder sharing of code – no one else will have the same file configuration as you do. If you share your script with your TA or Prof, they will not be able to access the files you are referencing in an absolute path. Thus, they will not be able to run the code as-is in your script.

In order to overcome the use of absolute paths and `setwd()`, we strongly recommend that you conduct all work in RStudio within an **R project**. When you create an R project, R sets the working directory to a file folder of your choice. Any files that your code needs to run (i.e., data sets, images, etc.) are placed within this folder. You can then use relative paths to refer to data files in the project folder, which is much more conducive to sharing code with colleagues, TAs, and Profs.

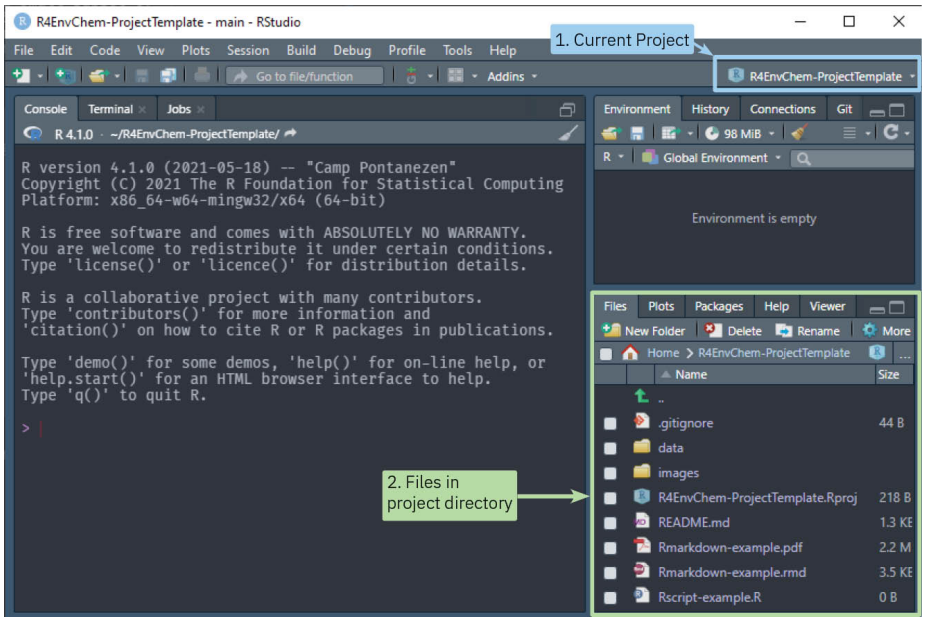
2.3 Importing a project

While you can create a project from scratch (discussed below), we've created a draft project template. Download it, and you'll have a working RStudio project

that you can use as you follow along with the code in the rest of this chapter and the tutorial exercise.

1. Downloading the template project (zip file) from the GitHub repository here; there are instructions on downloading at the bottom of the repositories webpage.
2. Upload the project zip file to JupyterHub, and unzip the folder.
3. From RStudio click `File -> Open Project...` and open the `R4EnvChem-ProjectTemplate.Rproj` file from the unzipped folder.

If you've followed the steps above you should have successfully downloaded and opened an RStudio project, and it should look like this:



Note how the project name is displayed on the top right. You can quickly switch between projects here which is useful if you'll be using R for many different labs/courses. As well, take note that the working directory has changed to the one where the RStudio project is located. Since you've downloaded the entire project, the working directory for the project includes the example scripts and data files you'll need to continue along with the remainder of this book. If you open the project folder (or access it from the `Files` tab) it should look like this:

```
R4EnvChem-ProjectTemplate
    R4EnvChem-ProjectTemplate.Rproj
    Rscript-example.R
|   README.md
|   Rmarkdown-example.rmd

  data
```

```
        2018-01-01_60430_Toronto_ON.csv
        2018-07-01_60430_Toronto_ON.csv
|       ...

  images
        DHall_TorontoPano.jpg
```

With the `R4EnvChem-ProjectTemplate.Rproj` file located in the main folder, this is important as we'll be able to readily look for files we stored in project subfolders such as `data` and `images`.

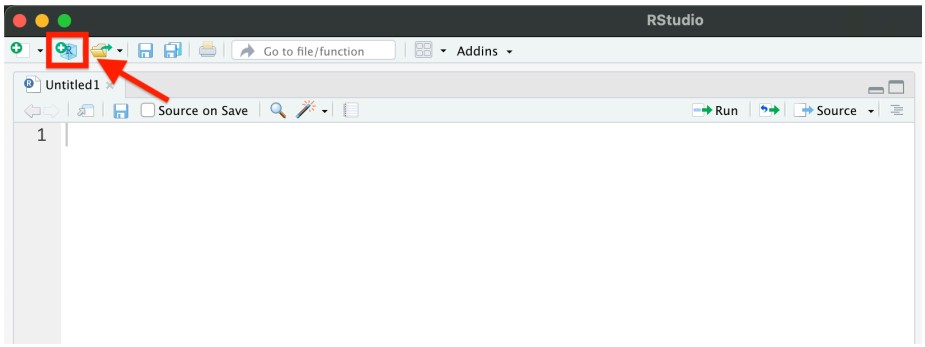
As you can see, the `R4EnvChem-ProjectTemplate.Rproj` file is located in the main folder, which RStudio will now treat as the working directory. Essentially it means we'll be able to quickly access files in project subfolders such as data and images without having to find out what the full file path is for your own computer. You'll appreciate this as you progress through this book.

In the future you can create your own projects from scratch, but it behooves you to follow the template layout. Having consistently named folders you'll use in every project will help simplify your life down the road.

2.4 Creating an RStudio project

We've provided instructions on creating your own RStudio project from scratch, but you can always copy the template project folder above (or any for that matter) to re-purpose it as you see fit.

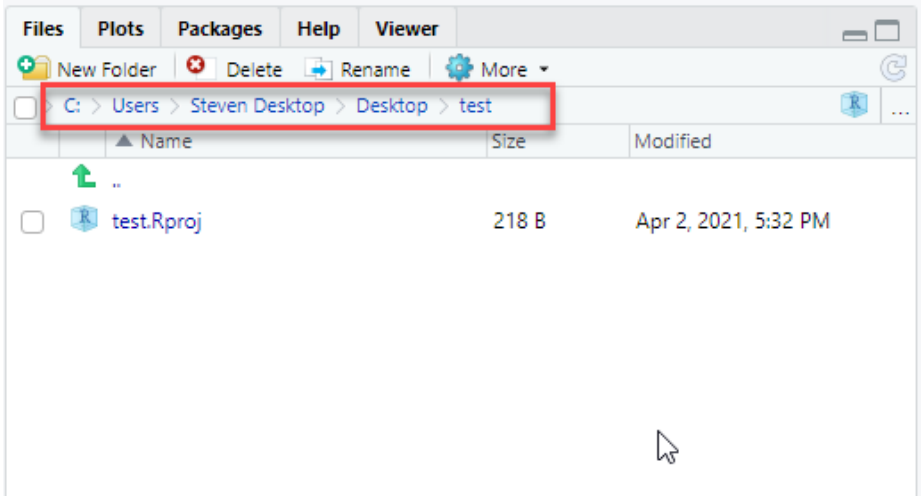
To create a new project: go to *File->New Project*, or click the button highlighted in the image below. Click *New Directory*, then *New Project*.



You may want your project directory to be a sub-folder of an existing directory on your computer which already contains your data sets. If this is the case, click *Existing Directory* instead of *New Directory* at the previous step, and then select the folder of your choice.

Next, you'll be asked to choose a sub-directory name and location. Enter your selected name and choose an appropriate location for the folder on your com-

puter. Click *Create Project*, and you should now see your chosen file path displayed in the *Files* tab of the Viewer pane:



When working on assignments for coursework, it is good practice to create a new R project for each assignment you work on. You should store the data, images, and any other files required for that assignment within the folder for the designated R project. You can create sub-folders for data and images, however, you may want to avoid making too many nested sub-folders, as this will make your paths long and tiresome to type. For a hypothetical course with 5 Labs (*cough* CHM410 *cough*), your coursework would look like this:

```
CHM410
|
   Project 1
|     |
|     |   project1.Rproj
|     |   project1WriteUp.Rmd
|       data
|           ...
|       images
|           ...
|
   Project 2
      |
      |   project2.Rproj
      |   project2WriteUp.Rmd
        data
            ...
        images
            ...
```

...

With a separate folder for each experiment, and within each folder is an RStudio project, data, images, and other files required for *that* specific project. You shouldn't have nested R studio project as their is no benefit to this approach. Keep everything you need in one location, and no more.

2.5 A sneak peek at `.Rmd` files

In this textbook, **you will exclusively work with `.Rmd` (R Markdown) files**, which offer a dynamic and interactive platform for blending code, text, and output.

Within an `.Rmd` file, you will encounter two distinct components: **code and text**.

- *Text fields*, easily accessible by inserting regular text, allow you to compose explanations, context, and interpretations using plain language. These text fields can be created directly within the `.Rmd` document.
- *Code chunks*, on the other hand, house R code that can be executed to generate results and graphics.

We will learn more about working with R markdown in the later chapters.

2.6 Summary

In this chapter we've covered:

- Importing the *R4EnvChem Project Template* so we have access to data for the tutorial (amongst other things)
- The concept of paths and directories and how relative referencing withing a project greatly simplify this

2.7 Exercise

For this chapter, you will create your own R project in UofT JupyterHub RStudio.

2.7.1 Get Started:

- Launch RStudio on the UofT JupyterHub server.

2.7.2 Confirm Your Working Directory

- Use the `getwd()` function in RStudio to display the current working directory in the console.

- Ensure that the working directory in RStudio is the location where you'd like to set up your project.

2.7.3 Creating Your Own Project:

- Launch a new RStudio project. To do this, go to File -> New Project.
- Choose "New Directory".
- Select "New Project".
- Name the project "MyFirstRProject" and choose a convenient location to save it.
- Click on "Create Project".
- Use the `getwd()` function again to check your current working directory and confirm you're in the "MyFirstRProject" directory.
- Inside the "MyFirstRProject" directory, create two new folders: "data" and "notebook". You can do this using RStudio's 'Files' tab or using the `dir.create()` function in the R console.

2.7.4 Create Your Rmd File:

- Within your "MyFirstRProject" directory, create a new `.Rmd` (R Markdown) file. You can do this by going to File -> New File -> R Markdown.
- Name the file "MyFirstRMarkdown" and set HTML as the default output format.
- In the text section of the `.Rmd` file, write one thing you remember about R and RStudio.
- Insert a code chunk below what you wrote. In this code chunk, type `sum(1:10)`, which calculates the sum of numbers from 1 to 10.
- Knit the document to see the results. This will produce an HTML or PDF document that shows both your text and the results of your R code.

2.7.5 Upload a file from your computer:

- Start by visiting the Exploring Air Quality Data website and navigate to the `My Data` tab.
- Once there, choose your preferred options on the left side. Then, on the right side, input your student number to retrieve your data. After your data displays, click on the `Download Your Data!` button to download the data as a CSV file (as shown in Figure 1).
- Next, upload this downloaded data to the "data" folder within your RStudio project. If you've forgotten how, just refer back to the beginning of this chapter for a quick reminder.
- After the upload, you'll be able to spot your data in the `Files` pane of RStudio. Simply click on the data's name and then select `View File` to peek at your raw data.

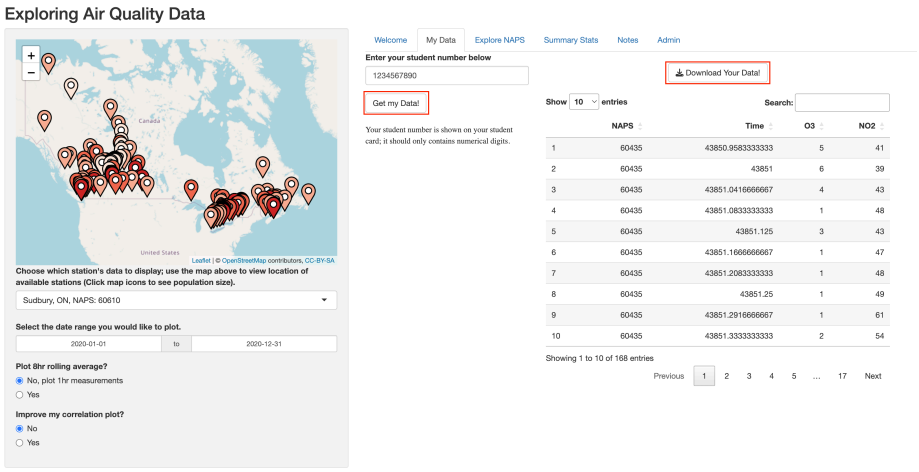


Figure 2.1: Figure 1

2.7.6 Reflection:

- Explain the difference between relative and absolute paths. Why are relative paths preferred when working in RStudio projects?

Chapter 3

How to Use This Textbook

Before we move onto the actual coding part, let's talk about how to navigate and utilize this textbook.

3.1 Keep in mind

1. Reading and Active Engagement

This textbook encourages active learning. Don't merely read through the content—interact with it. Type out the code in your R environment and see the results firsthand. This hands-on approach will solidify your comprehension and enhance your practical skills. Observe how the code behaves, experiment with modifications, and observe how changes impact the outcomes.

2. Curiosity and Inquisitiveness

When you encounter code you don't fully understand or want to know the underlying process, lean into your curiosity. Don't hesitate to ask "Why?" and explore concepts beyond the immediate scope. Seek to understand the "why" and "how" alongside the "what."

3. Resources and Further Explanation

This textbook is a stepping stone to your R journey. Beyond the content provided, explore the references, suggested readings, and online resources mentioned throughout the chapters. Embrace a curious attitude and continue to expand your knowledge by delving into more advanced topics or specific applications that align with your interests.

4. Discussion and Collaboration

If you're using this textbook as part of a class or a group, engage in discussions with your peers. Sharing insights, clarifying doubts, and collaborating on ex-

ercises can enhance your learning experience. Don't hesitate to ask questions, seek help, and contribute to a supportive learning environment.

3.2 Useful features

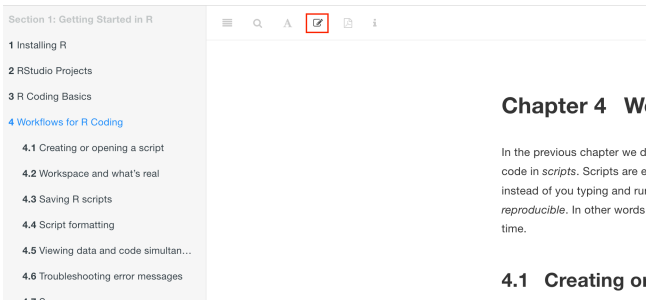
3.2.1 Searching throughout the textbook

By clicking on the magnifying glass icon in the top left corner, you have the ability to search for keywords across the entire textbook without worrying about case sensitivity. For instance, entering "tidyverse" will display all chapters where tidyverse is mentioned. This gives you a glimpse into future chapters, offering a preview of the various ways you'll be engaging with tidyverse later on!

3.2.2 How to view the original R Markdown of textbook chapters

This textbook is assembled from individual Rmd files, each representing a chapter. As you progress through the chapters, you may wish to examine the associated Rmd files to delve deeper into the code and its execution.

Simply click on the edit icon in the top left corner to be directed to the corresponding Rmd file on GitHub, opened in a new tab. You're encouraged to download these files, experiment with the code, and observe our Rmd formatting techniques!



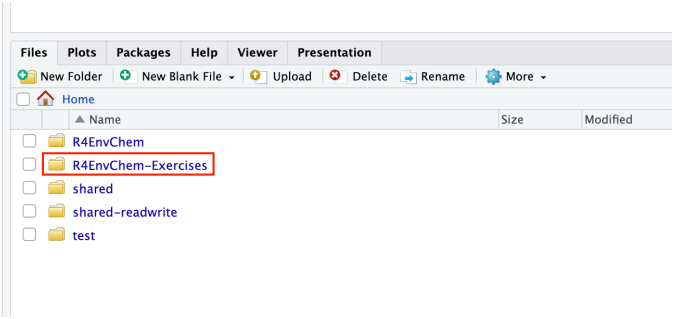
3.3 End-of-chapter Exercises

As you progress through each chapter, you'll find `.Rmd` (R Markdown) files available for practice and reinforcement. These Rmd files are designed to provide you with hands-on exercises that align with the concepts covered in the textbook.

Within each Rmd file, you'll encounter straightforward exercises that give you the opportunity to apply what you've learned in each chapter. After completing an exercise, you can run the provided unittest cell to check your answers and receive instant feedback.

Here is an instruction to how to start on the exercises:

1. Access the Repository:
   - **Recommended for those who have access to UofT Jupyter-Hub**: Click on the following link to automatically clone the "R4EnvChem-Exercises" repository to your UofT JupyterHub: R4EnvChem-Exercises Repository. This is a preferred method for anyone completing exercises as part of a UofT course.
   - Alternatively, you can directly download the files from this repository.
2. Work on the Exercise:
   - Once inside your JupyterHub's RStudio environment, in the "Files" pane you'll see "R4EnvChem-Exercises" folder (figure below). If you click into this folder, you will see a list of chapter folders. Each chapter folder contains the respective exercise Rmd files. Navigate to the desired chapter's folder and click on the exercise Rmd file you wish to work on. This will open the file in the RStudio editor.



   - Once the Rmd file is open, you can edit, run code chunks, and add your solutions directly in the file. Remember to save your progress regularly. If you want to generate an output document (like a PDF or HTML) to view your results, click on the "Knit" button usually located at the top of the script editor.
   - To enhance readability, you optionally click on the `Visual` tab at the top of the file view and work on the exercises.

Optional Extra Questions

For those seeking an additional challenge and a chance to delve into topics beyond the textbook, we offer optional extra questions. Resources and explanations will be provided to support you in tackling these optional questions.

By engaging with these interactive Rmd files, you can actively reinforce your learning, gain practical experience, and explore R concepts in depth. We encourage you to make the most of these resources to enhance your R proficiency. Happy learning!

Part 2: How to Code in R

Chapter 4

R Coding Basics

Now that you know how to navigate RStudio and have a working project, we'll take a look at the basics of R. As we're chemist first, and not computer programmers, we'll try and avoid as much of the nitty-gritty underneath the hood aspects of R. However, a risk of this approach is being unable to understand errors and warnings preventing your code from running. As such, we'll introduce the most important and pertinent aspects of the R language to meet your environmental chemistry needs.

4.1 Variables

We've already talked about how R can be used like a calculator:

```
(1000 * pi) / 2
```

```
## [1] 1570.796
```

```
(2 * 3) + (5 * 4)
```

```
## [1] 26
```

But managing these inputs and outputs is simplified with **variables**. Variables in R, like those you've encountered in math class, can only have one value, and you can reference or pass that value along by referring the variable name. And, unlike the variables in math classes, you can change that value whenever you want. Another way to think about it is that a variable is a box in which you store your value. When you want to move (reference) your value, you move the box (and whatever is inside of it). Then you can simply open the box somewhere else without having to worry about the hassle of what's inside.

You can assign the a value to a variables using `<-`, as shown below.

```
x <- 12
x
```

```
## [1] 12
```

In addition to reading code top to bottom, you often *read it from right to left*. `x <- 12` would be read as "take the value `12` and store it into the variable `x`". The second line of code, `x`, simply returns the value stored inside `x`. Note that when a variable is typed on it own, R will print out it's contents. You can now use this variable in snippets of code:

```
x
```

```
## [1] 12
```

```
x <- x * 6.022e23
x
```

```
## [1] 7.2264e+24
```

Remember, R evaluates from right to left, so the code above is taking the number `6.022e23` and multiplying it by the value of `x`, which is 12 and storing that value back into `x`. That's how we're able to modifying the contents of a variable using it's current value. You can also overwrite the contents of a variable at anytime (i.e. `x <- 25`).

Note that variable names are case sensitive, so if your variable is named `x` and you type `X` into the console, R will not be able to print the contents of `x`. Variable names can consist of letters, numbers, dots (`.`) and/or underlines (`_`). Here are some rules and guidelines for naming variables in R:

- **Variable Name Requirements** as dictated by R
  - names must begin with a letter or with the dot character. `var` and `.var` are acceptable.
  - Variable names *cannot* start with a number or the `.` character cannot be preceded by number. `var1` is acceptable, `1var` and `.1var` are not.
  - Variable names *cannot* contain a space. `var 1` is interpreted as two separate values, `var` and `1`.
  - Certain words are reserved for R, and cannot be used as variable names. These include, but are not limited to, `if`, `else`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NA`, and `NaN`

Good names for variables are short, sweet, and easy to type while also being somewhat descriptive. For example, let's say you have an air pollution data set. A good name to assign the data set to would be `airPol` or `air_pol`, as these names tell us what is contained in the data set and are easy to type. A bad name for the data set would be `airPollution_NOx_O3_June20_1968`. While this name is much more descriptive than the previous names, it will take you a long time to type, and will become a bit of a nuisance when you have to type

it 10+ times to refer to the data set in a single script. Please refer to the *Style Guide* found in *Advanced R* by H. Wickham for more information.

Lastly, R evaluates code from top-to-bottom of your script. So if you reference a variable it must have already been created at an earlier point in your script. For example:

```
y + 1
```

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

```
y <- 12
```

The code above returns the `object 'y' not found` error because we're adding `+ 1` to `y` which hasn't been created yet, it's created on the next line. These errors also pop up when you edit your code without clearing your workplace. All variables created in a session are stored in the working environment so you can call them, even if you change your code. This means you can accidentally reference a variable that isn't reproduced in the latest iteration of your code. Consequently, a good practice is to frequently clear your work-space suing the 'broom' button in the *environment* pane. This will help you to ensure the code you're writing will is organized in the correct order; see [Saving R scripts] for why this is important.

4.2 Data types

Data types refer to how data is stored and handled by and in R. This can get complicated quickly, but we'll focus on the most common types here so you can get started on your work. Firstly, here are the data types you'll likely be working with:

- **character**: `"a"`, `"howdy"`, `"1"`, is used to represent string values in R. Basically it's text that you'd read. Strings are wrapped in quotation marks. For example, `"1"`, despite being read as number by us, is stored as a character and treated as such by R.
- **numeric** is any real or decimal number such as `2`, `3.14`, `6.022e23`.
- **integer** such as `2L`, note the 'L' tells R this is an integer.

- **logical** is a Boolean logic value, they can only be `TRUE` or `FALSE`

Sometimes R will misinterpret a value as the wrong data type. This can hamper your work as you can't do arithmetic on a string! So let's look at some helpful functions to test the data type of a value in R, and how to fix it.

```
x <- "6"
x / 2
```

```
## Error in x/2: non-numeric argument to binary operator
```

"non-numeric argument to binary operator" is a commonly encountered error, and it's simply telling you that you're trying to do math on something you can't do math on. You might think if `x` is 6, why can't I divide it by 2? Let's see what type of data `x` is:

```
is.numeric(x)   # test if numeric
```

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

```
is.logical(x)   # test if logical
```

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

```
is.integer(x)   # test if integer
```

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

```
is.character(x) # test if character
```

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

So the value of `x` is a character, in other words R treats it as a word, and we can't do math on that (think, how would you divide a word by a number?). So let's convert the data type of `x` to numeric to proceed.

```
x
```

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

```
x <- as.numeric(x)
is.numeric(x)
```

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

```
x
```

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

```
x / 2
```

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

So we've converted our character string `"6"` to the numerical value `6`. Keep in mind there are other conversion functions which are described elsewhere, but you can't always convert types. In the above example we could convert a character to numeric because it was ultimately a number, but we couldn't do the same if the value of `x` was `"six"`.

```
x <-"six"
x <- as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
x
```

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

"NAs introduced by coercion" means that R didn't know how to convert "six" to a numeric value, so it instead turned it into an *NA*, representing a missing value.

4.3 Data structures

Data structures refers to how R stores data. It's easy to get lost in the weeds here, so we'll start with the focus on the most common and useful data structure for your work: *data frames*.

4.3.1 Data Frames

Data frames consist of data stored in rows and columns. If you've ever worked with a spreadsheet (i.e. *Excel*), it's essentially that with the caveat that *all data stored in a column must be of the same type*. Again, different columns can have different data types, but *within* a column all the data needs to be the same type. R will convert your data otherwise to make it all the same. A common error is a single character in a column of numerical values leading to the entire column to be interpreted as character values; similar to what we discussed above. Errors like this most often stem from mistakes in recording and importing your data so be careful!

4.3.1.1 Creating a new dataframe from scratch

Let's see how we can create a dataframe by explicitly listing out the values.

```
# data
names <- c("Alice", "Bob", "Charlie", "David", "Eve")
ages <- c(20, 21, 22, 23, 19)
food <- c("Bubble Tea", "Pineapple Pizza", "Diet Pepsi", "Korean BBQ", "Sushi AYCE")

# Creating the dataframe
students <- data.frame(Name = names, Age = ages, Food = food)

# Displaying the dataframe
print(students)
```

```
##      Name Age            Food
## 1   Alice  20      Bubble Tea
## 2     Bob  21 Pineapple Pizza
## 3 Charlie  22      Diet Pepsi
## 4   David  23      Korean BBQ
## 5     Eve  19      Sushi AYCE
```

4.3.1.2 Reading data from a file

Obviously when we have many more data, it would be unrealistic to manually list them out in our code. So instead, we can create a dataframe by reading a file.

From the `R4EnvChem-ProjectTemplate`, downloaded in Importing a project, let's import some real data as follows by typing the following into the console:

```
airPol <- read.csv("data/2018-01-01_60430_Toronto_ON.csv")
```

`read.csv()` is a useful R built-in function which, as you might guess from its name, can read a `.csv` file and convert it into a data frame. The data we just imported contains air quality data measured in downtown Toronto around January 2018. The "Column specification" summary printed to the console is a useful feature of `read.csv()`. It tells you what data type was determined for each column when it was imported. Note that *double* is simply another term for the *numeric* data type. Some of the variables are:

- `naps`, `city`, `p`, `latitude`, `longitude` to tell you where the data was measured.
- `data.time` for when the measurements were taking. Note this is a `datetime`, which is a subset of numeric data. The values contained herein correspond to time elements such as year, month, data, and time.
- `pollutant` for the chemical measured
- `concentration` for the measured concentration in parts-per-million (ppm).

We've assigned it to the variable: `airPol`. This is so we can reference it and make use of it later on (see below). If we didn't do this our data would simply be printed to the console which isn't helpful. Let's take a look at the first few rows of the data:

```
head(airPol)
```

```
##    naps    city  p latitude longitude           date.time pollutant
## 1 60430 Toronto ON 43.70944  -79.5435 2018-01-01 00:00:00        O3
## 2 60430 Toronto ON 43.70944  -79.5435 2018-01-01 00:00:00       NO2
## 3 60430 Toronto ON 43.70944  -79.5435 2018-01-01 00:00:00       SO2
## 4 60430 Toronto ON 43.70944  -79.5435 2018-01-01 01:00:00        O3
## 5 60430 Toronto ON 43.70944  -79.5435 2018-01-01 01:00:00       NO2
## 6 60430 Toronto ON 43.70944  -79.5435 2018-01-01 01:00:00       SO2
##   concentration
## 1             3
## 2            39
## 3             1
## 4             1
## 5            47
## 6             3
```

Here we see that the data is stored in a **tidy** format, which is to say each column is a variable and each row is an observation. So reading the first row, we know that the Toronto 60430 station on 2018-07-01 at midnight measured ambient O_3 concentrations of 46 ppm (Note the concentration column isn't printed due to width). The concept of tidy data is important and is integral to working in R. It's discussed further in Tidying Your Data. Lastly, R will only output a small chunk of our data for us to see. If you'd like to see it in full, go the the `Environment` pane and double click on the `airPol` data.

4.3.2 Accessing data in subfolders

Note that `read.csv()` requires us to specify the file name, but in the above example we prefixed our file name with `"data/2018..."`. This is because the `.csv` file we want to open is stored in the `data` sub-folder. By specifying this in the prefix, we tell `read.csv()` to first go to the `data` sub folder in the working directory and *then* search for and open the specified data file.

What we've done above is called *relative referencing* and it's a huge benefit of projects. The actual data file is stored somewhere on your computer in a folder like `"C:/User/Your_name/Documents/School/Undergrad/Second_Year/R4EnvChemTemplate/data/2018-01-01_6`
If we weren't in a project, this is what you'd need to type to open your file, but since we're working in the project, R assumes the long part, and begins searching for files inside the project folder. Hence, why we only need `"data/2018..."`. Not only is this much simpler to type, and but it makes sharing your work with colleagues, TAs, and Profs (and yourself!) much easier. In other words, if you wanted to share your code, you would send the entire project folder (code & data) and the receiver could open it and run it as is.

4.3.3 Other data structures

R has several other data structures. They aren't as frequently used, but it's worth being aware of their existence. Other structures include:

- **Vectors**, which contain multiple elements *of the same type*; either numeric, character (text), logical, or integer. Vectors are created using `c()`, which is short for combine. A data frame is just multiple vectors arranged into columns. Some examples of vectors are shown below.

```
num <- c(1, 2, 3, 4, 5)
num
```

```
## [1] 1 2 3 4 5
```

```
char <- c("blue", "green", "red")
char
```

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

```
log <- c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE)
log
```

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

- **Lists** are similar to vectors in that they are one dimensional data structures which contain multiple elements. However, lists can contain multiple elements of different types, while vectors only contain a single type of data. You can create lists using `list()`. Some examples of lists are shown below. You can use `str()` to reveal the different components of a list, in a more detailed format than if you were to simply type the assigned name of the list.

```
hi <- list("Greetings" = "Hello", "someNumbers" = c(5,10,15,20), "someBooleans" = c(TRU
str(hi)
```

```
## List of 3
##  $ Greetings   : chr "Hello"
##  $ someNumbers : num [1:4] 5 10 15 20
##  $ someBooleans: logi [1:3] TRUE TRUE FALSE
```

```
hi
```

```
## $Greetings
## [1] "Hello"
##
## $someNumbers
## [1]  5 10 15 20
##
## $someBooleans
## [1]  TRUE  TRUE FALSE
```

```
hi$Greetings
```

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

There are many freely available resources online which dive more in depth into different data structures in R. If you are interested in learning more about different structures, you can check out the *Data structure* chapter of *Advanced R* by Hadley Wickham.

4.4 Conditional Statements

In programming, it's often necessary to make decisions and execute certain portions of code based on specific conditions. That's where conditional statements come into play.

In R, the primary mechanism to make decisions is the `if-else` construct. With it, you can evaluate a condition and, based on whether it's true or false, choose

which code block to execute.

4.4.1 Understanding R Syntax

Before diving into conditional statements, let's take a moment to understand the syntax used in R.

R, like many programming languages, uses a combination of parentheses `()`, curly braces `{}`, and other symbols to organize and structure the code.

1. Parentheses `()`: These are primarily used to enclose arguments of functions and conditions in control statements, like if. For example, in `if (x > 5)`, the condition `x > 5` is enclosed in parentheses.
2. Curly Braces `{}`: These are used to group multiple lines of code into a block. This is particularly useful in control statements where more than one line of code should be executed based on a condition.

The reason the curly bracket might span multiple lines is for readability. It makes it clear where a block of code begins and ends. While it's possible to write if-else statements without braces if only one statement is being conditioned, it's a good practice to always use them for clarity.

Now, with this understanding, let's move on to how R uses these in conditional statements.

4.4.2 The basic `if` statement

```r
x <- 10

# In this example, R checks if x is greater than 5.
if (x > 5) {
  print("x is greater than 5!")
}
```

```
## [1] "x is greater than 5!"
```

4.4.3 Expanding with `else` and `else if`

For situations where you want to specify actions for both true and false conditions, you can add an `else` section.

```r
x <- 3

if (x > 5) {
  print("x is greater than 5!")
} else { # if x <= 5
  print("x is 5 or less!")
}
```

```
## [1] "x is 5 or less!"
```

Here, because `x` is 3 (which is not greater than 5), R prints "x is 5 or less!".

For situations where multiple conditions need to be evaluated in sequence, you can use the else if construct. This allows you to add more conditions after the initial if.

```
x <- 6

if (x > 10) {
  print("x is greater than 10!")
} else if (x > 5) {
  print("x is greater than 5 but less than or equal to 10!")
} else {
  print("x is 5 or less!")
}
```

```
## [1] "x is greater than 5 but less than or equal to 10!"
```

In terms of syntax, it's important to remember:

- Always enclose the condition you're testing within parentheses `()`.
- Use curly braces `{}` to group the lines of code that should be executed for a particular condition.
- Make sure each else if or else follows an if or another else if. They cannot stand alone.

4.5 R built-in functions

Built-in functions are the essential tools that allow you to perform a wide range of tasks without having to write the underlying code from scratch. These functions are part of the R language itself and are readily available for your use.

In this chapter, you've already come across a few built-in functions that are incredibly useful. For instance, you've used the `read.csv()` function to import data from CSV files into your R environment. Additionally, the `as.numeric()` function has been employed to convert data to numeric format, and the `list()` function has aided in creating lists to organize and store data elements.

4.5.1 Exploring More Built-In Functions

Let's delve into a few more built-in functions that are integral to your R experience:

print(): The `print()` function displays output on the console. When you want to see the result of an expression or the contents of a variable, `print()` makes

it effortless.

```
print("Hello, R!")
```

```
## [1] "Hello, R!"
```

mean(): The `mean()` function calculates the average of a numeric vector.

```
mean(c(5, 10, 15, 20))
```

```
## [1] 12.5
```

max()/min(): With the `max()` function, you can effortlessly determine the maximum value within a numeric vector. Similarly, `min()` function returns the minimum value within a vector.

```
max(c(5, 10, 15, 20))
```

```
## [1] 20
```

```
min(c(5, 10, 15, 20))
```

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

4.5.2 Function documentation

An often unappreciated aspect of packages is that they not only contain functions we can use, but documentation. Documentation provides a description of the function (what it does), what arguments it takes, details, and working examples. Often the easiest way to learn how to use a function is to take a working example and change it bit by bit to see how it works etc. To see documentation check the "help" tab in the "outputs" window or type a question mark in front of a functions name:

```
# Takes you to the help document for the read.csv function
?read.csv
```

You can also write you're own functions. Please see [Programming with R] for additional details.

4.6 Summary

In this chapter we've covered:

- The basics of coding in R including variables, data types, and data structures (notably `data.frames`).
- Importing data from your project folder into R
- Using if-else structure to build a conditional logic
- Using R built-in functions and opening function documentations

Now that you're familiar with navigating RStudio and some basic R coding, you may have realized that working the console can get real messy, real quick. Read on to Workflows for R Coding where we'll discuss R workflows to make everyone's lives easier.

4.7 Exercise

There is a set of exercises available for this chapter!

Not sure how to access and work on the exercise Rmd files?

- Refer to Chapter 3.3 for step-by-step instructions on accessing the exercises and working within the UofT JupyterHub's RStudio environment.
- Alternatively, if you'd like to simply access the individual files, you can download them directly from this repository.

Always remember to save your progress regularly and consult the textbook's guidelines for submitting your completed exercises.

Chapter 5

Workflows for R Coding

In the previous chapter, we conducted our coding in the console, which quickly became unwieldy. To address this, we transition to using R Markdown `.Rmd` files, which we briefly talked about before. Instead of running code interactively in the console, we write code blocks within the .rmd files, creating a comprehensive document that others can follow.

5.1 Creating or opening an R Markdown document

To start an R Markdown document:

1. Go to *File -> New File -> R Markdown*
2. Then save your document by going to *File -> Save As...*. a. Make sure to save your file with the `.Rmd` suffix. b. **Save your script in your project folder**, otherwise you'll run into issues.

We've also provided an example script in the R4EnvChem project template. Assuming you're currently in the template project you can open the script as follows:

1. Go to `File` -> `Open File` -> open the `Rscript-example.Rmd` file. This action will open a new pane above the console, dedicated to writing your R Markdown content.

5.2 Workspace and what's real

We've already mentioned the *environment* pane that displays objects present in your R session. While they are useful to work with, they're not considered *real*. That is to say, if you close your R session, those objects will be lost. And while

RStudio allows you to save a working environment (and it's associated objects), it's crucial to understand that *only your saved scripts/markdown documents are real*. You can't readily share your working environment, and even so it's bad practice as you may reference a previous iteration of an object giving you erroneous results. Think back to the chemistry labs: you may jot notes down on loose leaf, but only what's written in your lab book is considered real... we'll that's how it's supposed to work anyways.

The idea is everything you need can be generated from the original data and the instructions in your R script/markdown document. Anyone should be able to take your data and your code and get the same results you got. This is paramount for the reproducibility of your work and your results.

5.3 Saving R Markdown

To save an R Markdown document:

- Navigate to *File -> Save* or use the 'Save' icon in the top left corner of your document.

Content saved to an .Rmd file is considered real and self-contained. Variables, plots, or datasets that appear in your workspace or the `Environment` window aren't self-contained. Whenever you close RStudio, any objects in R that are not considered *real* will be lost in that R session. Furthermore when you need to share your code (for school or publication) you'll need to share your data and your script, but never your work-space. This is to increase predictability and helps people (and you) to make sure your work is reproducible, an under appreciate hallmark of science.

5.3.1 What should I save?

At this point in the chapter, two things should be clear:

1. R Markdown documents saved to `.Rmd` files are the real record of your work.
2. Objects in your work-space/environment are not real, and will not be available to you after you close and re-open RStudio unless you re-run the code used to generate the work-space.

So what is important to save in R, and how often should you save these files?

- Save the R Markdown scripts you write, and do so regularly. Even minor changes are worth saving before closing RStudio, as it's easy to forget those small differences upon return.
- Ensuring that even if you lose an object in your workspace, your R Markdown script contains the code needed to recreate that object.

- Generate the object before referencing it in subsequent commands. This ensures that variables are generated in the workspace before being referenced by later commands when running scripts from top to bottom.

By adhering to these practices, you ensure your R Markdown documents remain accurate, your code is complete, and your work remains reproducible.

5.3.2 Saving objects

In some cases, your code may be used to generate large datasets which require quite a bit of time to create. It can be quite tedious to re-run the code used to generate these large data sets every time you open RStudio, and you might find yourself wanting to save the data to a *real* file that you can simply import the next time you open the application. Also, you may be finished with your analysis and want to save the final data. You can save your the data contained in your data frame as a `.csv` file using `write.csv()`.

```
# dummy data frame to save
df <- data.frame(x = c(1,2,3),
                 y = c("yes", "no", "maybe"))

write.csv(x = df,
          file = "testData.csv")
```

Breaking it down:

- we created a dummy data frame `df`; in reality you'll already use a data frame from your analysis.
- we called `write.csv()` and
  - `x = df` specifies we want to save the data.frame `df`
  - `file = "data/testData.csv"` specifies *where* we want the file to save (in the *data* sub-directory, more later), and *what* our file will be called (*testData.csv*). It's important to specify the file extension so R knows how to save it.

5.4 Script formatting

You should now be familiar with how to open the Scripts window, as well as some of the advantages of typing your code into this window rather than into the console directly. Before you write your first script, let's review some basic script formatting.

Before you enter any code into your script, it is good practice to fill the first few lines with text comments which indicate the script's title, author, and creation or last edit date. You can create a comment in a script by typing `#` before any text. An example is given below.

```
#Title: Ozone time series script
#Author: Georgia Green
#Date: January 8, 2072
```

Below your script header, you should include any packages that need to be loaded for the script to run. Including the necessary packages at the top of the script allows you, and anyone you share your code with, to easily see what packages they need to install. This also means that if you decide to run an entire script at once, the necessary packages will always be loaded before any subsequent code that requires those packages to work.

The first few lines of your scripts should look something like the following.

```
# Title: Example R Script for Visualizing Air Quality Data
# Author: John Guy Rubberboots
# Date: 24 June 2021

# 1. Packages ----

# Install tidyverse if you haven't already
#install.packages("tidyverse")

library(tidyverse)
```

The rest of your script should be dedicated to executable code. It is good practice to include text comments throughout the script, and in between different chunks of code to remind yourself what the different sections of code are for (i.e., `# 1. Packages ----` in the above example). This also makes it easy for anyone you share your code with to understand what you're trying to do with different sections within the script.

You can also use headers and sub-headers in your scripts using `#`, `##`, and `###` before your text and `---` after as shown below:

```
# Section ----
## Subsection ----
### Sub-subsection ----
```

Headings and subheadings are picked up by RStudio and displayed in the Document Outline box. You can open the Document Outline box by clicking the button highlighted in the image below. Use of these headings allows easy navigation of long scripts, as you can navigate between sections using the Document Outline box.

5.5 Viewing data and code simultaneously

Before we get into more about coding and workflows, you may find yourself wanting to be able to to view your scripts and data side-by-side. You can open

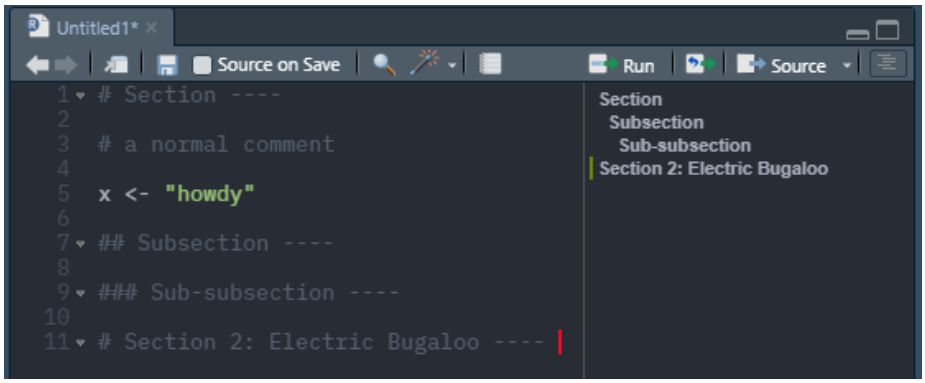


Figure 5.1: Example script headings, document outlines, and comments. Note the "—" which tells RStudio this comment is to be treated as a script heading.

a script, plot, or data set in a new window by clicking and dragging the tab in RStudio or by clicking the button highlighted in the image below.

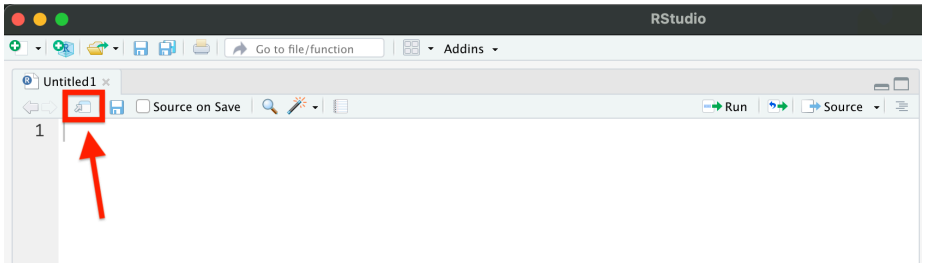


Figure 5.2: How to open an R script/plot/data set in a new window.

5.6 Troubleshooting error messages

In the previous section, you were introduced to your first error message in R, and we briefly discussed how to resolve the issue. As you begin to code, many of your errors will be routine syntax error such as unmatched parenthesis (the dreaded "Incomplete expression:"). Fortunately, RStudio will highlight any syntax errors in your code with a red squiggly line and an 'x' in the side bar, as shown below. You can hover over the 'x' to see what is causing the error.

In the above message, R is telling you that it is not sure what to do with `b`. As mentioned previously, variable assignment is done in the format `name <- assignment`. However, in the above example, the variable assignment statement is written as `name name <- assignment`. Since variable names cannot contain

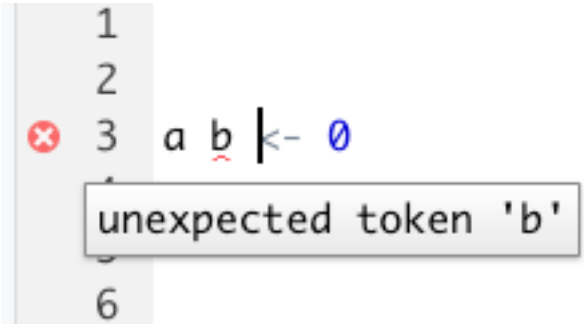


Figure 5.3: Figure 3.8: RStudio highlights syntax errors in the Scripts window.

spaces, R reads `a b` as two separate input variable names, not as a single string. If you wanted to assign a value of 0 to both `a` and `b`, you would need to write the statement once per variable, as shown below.

```
a <- 0
b <- 0
```

Let's look at another example. Some functions require you to write code with nested parentheses. A good example would be the `aes()` argument that is called inside of `ggplot()`, as shown below.

```
#plot ozone concentration vs. time
ggplot(data = airPol,
       aes(x = date.time,
           y = concentration,
           colour = pollutant)) +
  geom_point()
```

(For more detail about importing and using `ggplot2`, please re-visit Chapter 2, section 2.3.4, or see Chapter 11.)

If you were to forget one of the parentheses in the previous line of code, RStudio would highlight it similar to below:

Here R is telling you that you have an unmatched opening bracket. To resolve the error, simply add a closing bracket to match.

The `expected ',' after expression` is a common error that you will see accompanying unmatched opening brackets. Sometimes you might get this error in the console after running code that is missing a bracket somewhere. It is good practice to check your parentheses a few times before running your code to make sure that all the commands are closed, and that R doesn't keep waiting for you to continue inputting code after you've click *Run*. If you notice that the `>` in your R console has turned into a `+`, this is likely because you've just run a command that is missing a closing bracket, and thus, R is not aware that your

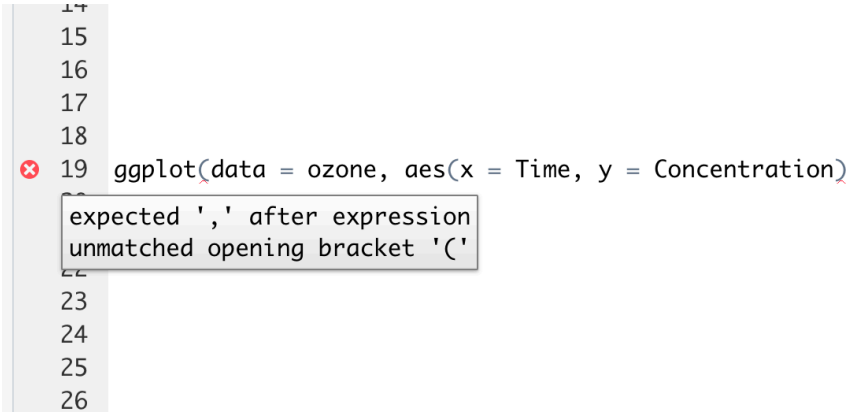


Figure 5.4: Figure 3.9: RStudio highlights unmatched parentheses in the script window.

code is finished. Simply input a closing bracket into the console, and the `>` should return.

While the script window is very useful for pointing out syntax errors in your code, there are many other errors that can arise in RStudio which the script window is not able to capture. These are generally errors that arise from trying to execute your code, rather than from mistakes in your syntax.

The following is a prime example of such an error.

```
q <- 8 + "hi"
```

```
## Error in 8 + "hi": non-numeric argument to binary operator
```

Here we are trying to add a numeric value (8) to a character string ("hi"), then set the sum of the two to variable `q`. R has given us an error in return, because there is no logical way for R to add a numeric value to non-numeric text. The error indicates that we have passed a `non-numeric argument to binary operator`, meaning we have used a non-numeric data type for an expression which is exclusively reserved for numeric data.

It is important to be aware of error codes as many functions require specific data types as their inputs. You can alwways consult the function documentation by via the *Help* tab of the *Viewer* pane or by typing a `?` followed by the name of the function in the console (i.e. `?ggplot`).

5.7 Summary

In this chapter we've covered:

- R workflows in the context of projects and markdown documents
- What's considered *real* when working in RStudio
- How to format your markdown for legibility (Remember you're the one who's going to be stuck rereading it!)
- Troubleshooting some common error messages

Now that you're familiar the above, we'll introduce Using R Markdown, a way to combine your R code, it's outputs, and your writing all in one dynamic document (like your lab reports!).

5.8 Exercise

There is a set of exercises available for this chapter!

Not sure how to access and work on the exercise Rmd files?

- Refer to Chapter 3.3 for step-by-step instructions on accessing the exercises and working within the UofT JupyterHub's RStudio environment.
- Alternatively, if you'd like to simply access the individual files, you can download them directly from this repository.

Always remember to save your progress regularly and consult the textbook's guidelines for submitting your completed exercises.

Chapter 6

Using R Markdown

Before going into more details of R Markdown, let's talk about two common options in the world of R coding: the R script (.R) and the dynamic R Markdown document (.Rmd).

R Scripts: Imagine coding as crafting a detailed recipe of R commands—a script—that guides R through specific tasks. Conventional R scripts (.R files) are dedicated to these commands, handling calculations and operations. However, as scripts grow, they become complex and sharing insights alongside code becomes challenging.

R Markdown: R Markdown (.Rmd) elevates the coding experience by harmonizing code with explanatory text. Within an R Markdown document, code blocks act like individual scripts—smaller, more focused units. These blocks merge code with explanations seamlessly, creating a coherent narrative. Unlike isolated scripts, R Markdown emphasizes both code functionality and its significance within the context. For these reasons, we'll be sticking to working in `.Rmd` files.

In a nutshell, R Markdown allows you to analyse your data with R and write your report in the same place (this entire book was written with R Markdown). This has loads of benefits including a *reproducible workflow*, and streamlined thinking. No more flipping back and forth between coding and writing to figure out what's going on.

Let's run some simple code as an example:

```
x <- 2+2
x
```

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

What we've done here is write a snippet of R code, ran it, and printed the results (as they would appear in the console). While the above code isn't anything

special, we can extend this concept so that our R markdown document contains any data, figures or plots we generate throughout our analysis in R. For example:

```
library(tidyverse)
library(knitr)

airPol <- read_csv("data/2018-01-01_60430_Toronto_ON.csv")

ggplot(data = airPol,
       aes(x = date.time,
           y = concentration,
           colour = pollutant)) +
  geom_line() +
  theme_classic()
```

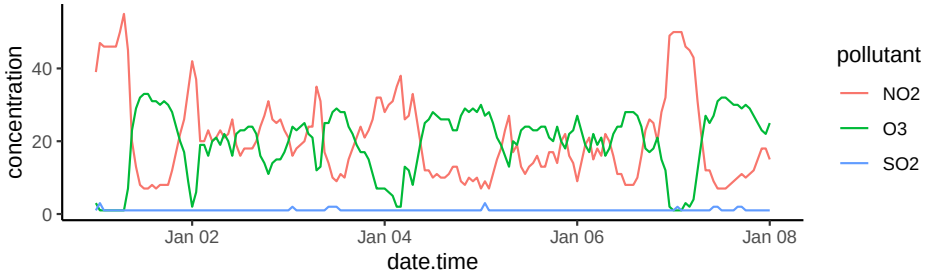


Figure 6.1: Time series of 2018 ambient atmospheric O_3, NO_2, and SO_2 concentrations (ppb) in downtown Toronto

```
sumAirPol <- airPol %>%
  drop_na() %>%
  group_by(city, naps, pollutant) %>%
  summarize(mean = mean(concentration),
            sd = sd(concentration),
            min = min(concentration),
            max = max(concentration))

knitr::kable(sumAirPol, digits = 1)
```

city	naps	pollutant	mean	sd	min	max
Toronto	60430	NO2	20.5	11.5	7	55
Toronto	60430	O3	19.7	8.7	1	33
Toronto	60430	SO2	1.1	0.3	1	3

Pretty neat, eh? You might not think so, but let's imagine a scenario you'll encounter soon enough. You're about to submit your assignment, you've spent hours analyzing your data and beautifying your plots. Everything is good to go

until you notice at the last minute you were supposed to *subtract* value `x` and not value `y` in your analysis. If you did all your work in *Excel* (tsk tsk), you'll need to find the correct worksheet, apply the changes, reformat your plots, and import them into word (assuming everything is going well, which is never does with looming deadlines). Now if you did all your work in R markdown, you go to your one `.rmd` document, briefly apply the changes and re-compile your document.

A lot of scientists work with R Markdown for writing their reports for numerous reasons:

1. **Integrated Workflow**: Combines narrative, data analyses, and visualizations in one document, promoting reproducibility and transparency.
2. **Versatility**: Easily exports to diverse formats like HTML, PDF, and Word, catering to different dissemination needs.
3. **Plot Management**: Offers precise control over visual presentations, allowing for tailored figure sizes, resolutions, and formats.

In sum, R Markdown provides a streamlined platform for scientific communication, merging data analysis with polished publication seamlessly.

6.1 Getting started with R Markdown

As you've already guessed, R markdown documents use R and are most easily written and assembled in RStudio. If you have not done so, revisit Chapter 1:[Installing R]. Once setup with R and R Studio, you'll need to install the `R Markdown` and `tinytex` packages by running the following code in the console:

```
# These are large packages so it'll take a couple of minutes to install

install.packages("R Markdown") # downloaded from CRAN

install.packages("tinytex")
tinytex::install_tinytex()  # install TinyTeX
```

The `R Markdown` package is what we'll use to generate our documents, and the `tinytex` package enables compiling documents as PDFs. There's a lot more going on behind the scenes, but you shouldn't need to worry about it.

Now that everything is setup, you can create your first R Markdown document by opening up R Studio, selecting `FILE -> NEW FILE -> R Markdown`. A dialog box will appear asking for some basic input parameters for your R markdown document. Add your title and select *PDF* as your default output format (you can always change these later if you want). A new file should appear that's already populated with some basic script illustrating the key components of an R markdown document.

6.1.1 Understanding R Markdown

Your first reaction when you opened your newly created R markdown document is probably that it doesn't look anything at all like something you'd show your prof. You're right, what you're seeing is the plain text code which needs to be compiled (called *knit* in R Studio) to create the final document. When you create a R markdown document like this in R Studio a bunch of example code is already written. You can compile this document (see below) to see what it looks like, but let's break down the primary components. At the top of the document you'll see something that looks like this:

```
---
title: "Temporal Analysis of Foot Impacts While Birling Down the White Water"
author: "Jean Guy Rubberboots"
date: "24/06/2021"
output: pdf_document
---
```

This section is known as the *preamble* and it's where you specify most of the document parameters. In the example we can see that the document title is "Temporal Analysis of Foot Impacts While Birling Down the White Water", it's written by Jean Guy Rubberboots, on the 24th of June, and the default output is a PDF document. You can modify the preamble to suit your needs. For example, if you wanted to change the title you would write `title: "Your Title Here"` in the preamble. Note that none of this is `R` code, rather it's `YAML`, the syntax for the document's metadata. Apart from what's shown you shouldn't need to worry about this much, just remember that indentation in YAML matters.

6.1.1.1 Output Options in R Markdown

You can compile your entire document using the *Knit document* button. This is a great way to tinker with your code before you compile your document. Knitting will sequentially run all of your code chunks, generate all the text, knit the two together and output a PDF. You'll basically save this for the end.

R Markdown offers flexibility in terms of output formats, allowing users to knit their documents into various outputs tailored to their needs.

Three Common Output Options:

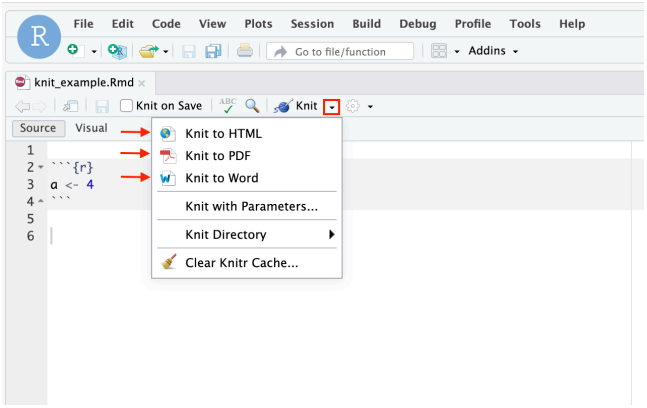
- **HTML** (html_document): Produces an HTML file, suitable for hosting on websites or for sharing via email. This format allows for interactive content, making it ideal for interactive graphs or web applications.
- **PDF** (pdf_document): Creates a PDF file. This format is best for documents intended for print or formal submissions, as it maintains consistent formatting across different devices and platforms.
- **Word** (word_document): Generates a Microsoft Word document, which

can be useful when sharing drafts or collaborating with colleagues who use Word for edits.

Controlling the Output:

- **Modifying the YAML Header**: You can change the output format directly in the YAML header of your Rmd file. In the last example, replacing `output: pdf_document` with `output: html_document` or `output: word_document` would knit the document into HTML or Word, respectively.

- **Using RStudio's Knit Button**: In the RStudio IDE, at the top of the script editor pane, there's a Knit button. Clicking the small dropdown arrow next to this button allows you to choose the output format you desire. Selecting one of the options will knit the document into that format and update the YAML header accordingly.

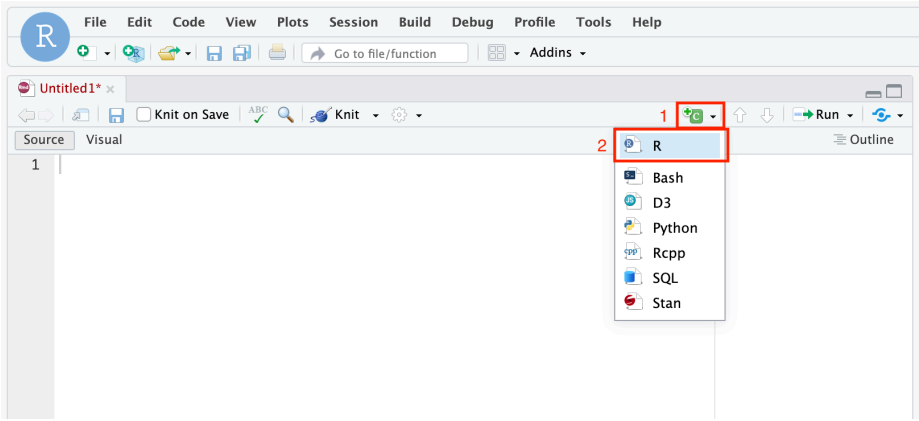


6.1.2 Running code in R Markdown

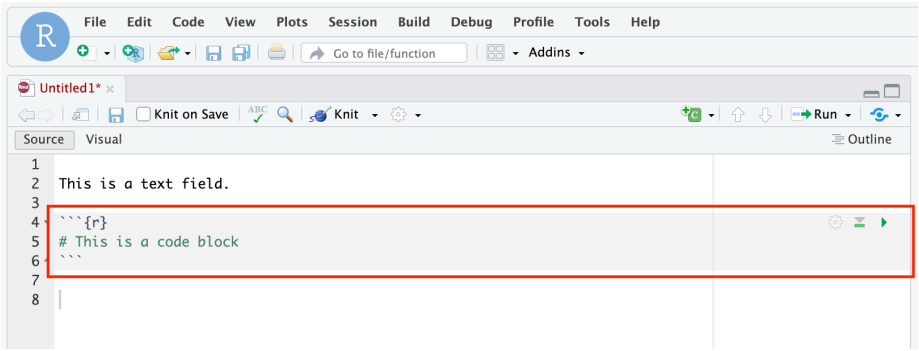
6.1.2.1 How to create code chunks

To create a code chunk within RStudio, you have several options:

1. Use the green "c" button located at the top right corner of your file view and select "R". Make sure your cursor is positioned at the desired location within your .rmd file when you do this.



2. Type ```` ```{r} ```` – three back-ticks followed by `{r}` – to initiate a new code chunk, and type ```` ``` ```` – three backticks (“‘) – to end the code chunk. You can specify code chunks options in the curly braces. i.e. ```` ```{r, fig.height = 2} ```` sets figure height to 2 inches. See the *Code Chunk Options* section below for more details.



3. Inline code expression, which starts with `` `r `` and ends with `` ` `` in the body text. Earlier we calculated `x <- 2 + 2`, we can use inline expressions to recall that value.

6.1.2.2 How to run code chunks

To run code within an R Markdown document, you again have various options to choose from.

1. You can run a specific code chunk by clicking the green triangle button located within each chunk. This action will execute the entire chunk, including all the code it contains.

2. For more control, you can run selected lines or chunks. To do this, use the “Run” button at the top of the file view. This button provides a range of

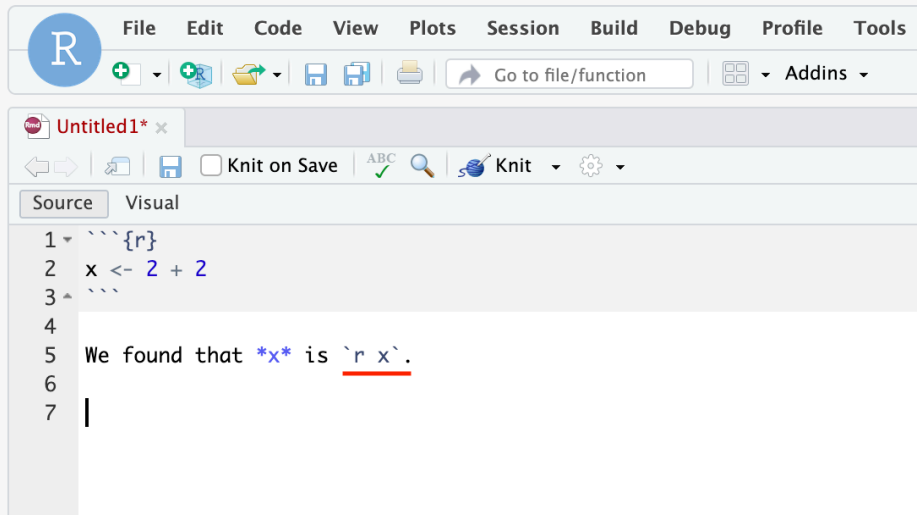


Figure 6.2: When we knit the markdown file shown in the figure, the knitted document will say "We found that x is 4."

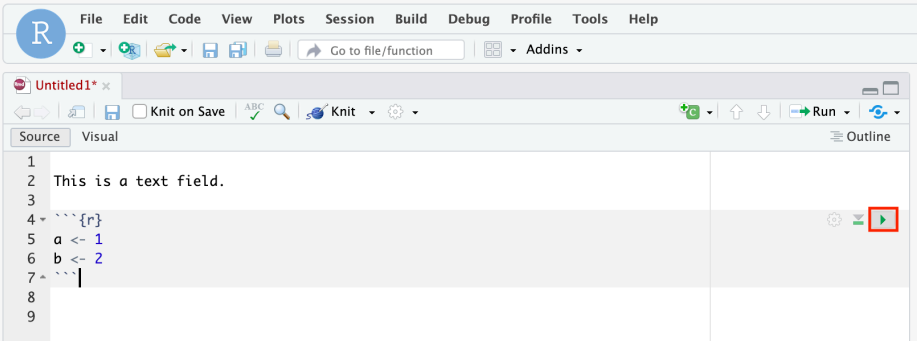


Figure 6.3: Run a code chunk using its own run button.

execution options that allow you to run code in a manner that suits your needs.

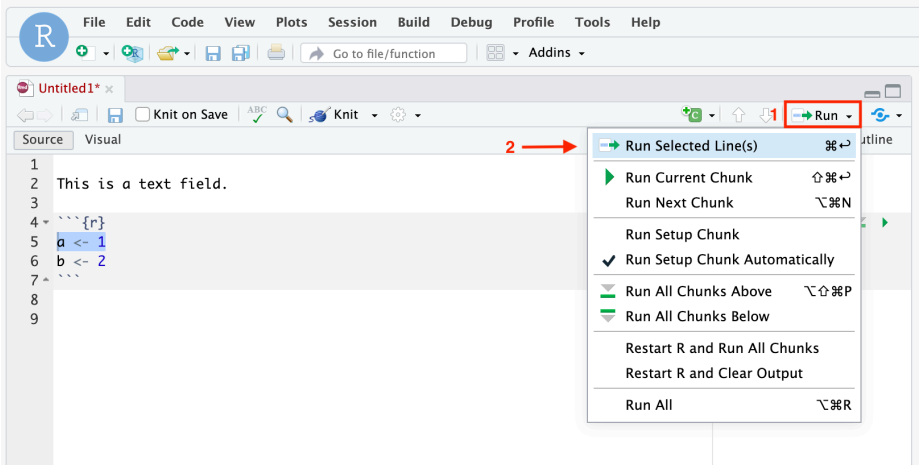


Figure 6.4: Example: Run selected line(s)

Note all the code chunks in a single markdown document work together like a normal R script. That is if you assign a value to a variable in the first chunk, you can call this variable in the second chunk; the same applies for libraries. **Also note that every time you compile a markdown document, it's done in a "fresh" R session.** If you're calling a variable that exist in your working environment, but isn't explicitly created in the markdown document you'll get an error.

6.1.3 Headings and Subheadings

In R Markdown, structuring your document with clear headings and subheadings is simple using the pound (`#`) sign. This not only helps in organizing content but also aids in creating a table of contents if required.

- **Main Headings**: Use a single pound sign (i.e. `# Main Heading`)
- **Subheadings**: Increase the number of pound signs based on the level of the subheading.
  - `## Subheading Level 1`
  - `### Subheading Level 2`
  - `#### Subheading Level 3`

R Markdown will automatically format these appropriately when the document is knit. For example, a main heading will typically appear larger and bolder than its subheadings, like this:

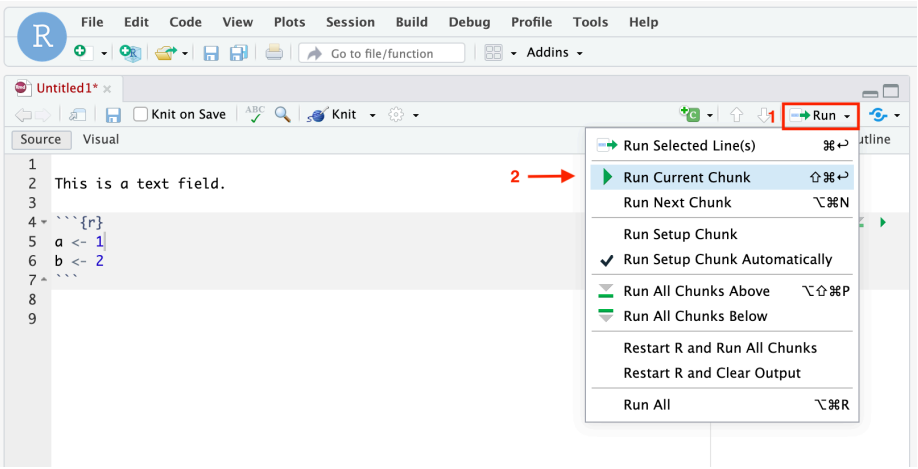
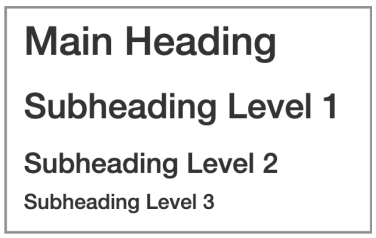


Figure 6.5: Example: Run Current Chunk



By effectively utilizing headings and subheadings, you can provide clear structure and flow to your document, making it more readable and navigable for your audience.

6.1.4 LaTeX Basics

LaTeX (pronounced "Lay-tech") is a typesetting system that's popular in academia due to its high-quality output format and the ability to handle complex formatting tasks. It's especially favored for documents that contain mathematical symbols, equations, and other specialized notation.

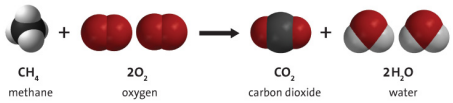
In R Markdown, LaTeX code can be integrated directly into your document to allow for advanced formatting, especially formathematical expressions and equations. When you knit your R Markdown document, the LaTeX code is rendered into beautifully formatted text.

There are two common ways to turn your expressions in a math mode.

1. **Display mathematical expressions**: centers the mathematical expression on its own line

2. **Inline mathematical expressions**: appears within the flow of a sentence or text

For chemistry students, one common use of LaTeX is to typeset chemical equations. We will provide examples on the combustion of methane:



6.1.4.1 Display math mode

You can have an entire line in a math mode using either `\[...\]` or `$$...$$`.

When you write it in rmd, it would look like :

- `\[ \text{CH}_4 + 2\text{O}_2 \rightarrow \text{CO}_2 + 2\text{H}_2\text{O} \]` *or* `$$ \text{CH}_4 + 2\text{O}_2 \rightarrow \text{CO}_2 + 2\text{H}_2\text{O} $$`

When you knit it, it will be displayed as:

$$\text{CH}_4 + 2\text{O}_2 \rightarrow \text{CO}_2 + 2\text{H}_2\text{O}$$

6.1.4.2 Inline math mode

On the other hand, if you want to insert your expression within your sentence, you can use `$...$` syntax.

With our methane combustion example, we can write something like this:

- `Methane ($\text{CH}_4$) reacts with oxygen ($\text{O}_2$) to produce carbon dioxide ($\text{CO}_2$) and water ($\text{H}_2\text{O}$).`

When you knit it, this will be displayed as:

- Methane (CH_4) reacts with oxygen (O_2) to produce carbon dioxide (CO_2) and water (H_2O).

6.1.4.3 Useful LaTeX Syntax

Now that you've seen how you can write your scientific expression in two different ways, let's look at some useful LaTeX Syntax for our purpose.

1. **Symbols**
   - Greek letters: Use a backslash followed by the name of the letter, e.g., `\alpha` for α.
   - Special symbols
     - `\times` for $\times$

   - `\approx` for $\approx$
   - `\geq` for $\geq$
   - `\rightarrow` for $\rightarrow$
2. **Superscripts and Subscripts**
   - Superscripts: `x^2` renders as x^2.
   - Subscripts: `H_2O` renders as H_2O
3. **Formatting**
   - Boldface: `\textbf{Text}` for **Text**
   - Italics: `\textit{Text}` for *Text*

In R Markdown, you can place your cursor over a LaTeX expression to preview its rendered output. This allows you to verify that the LaTeX formatting is correct before knitting the document.

6.1.4.4 More LaTeX Resources

There are numerous online resources dedicated to LaTeX symbols and their usage.

- A popular starting point is the Comprehensive LaTeX Symbol List, available on CTAN (Comprehensive TeX Archive Network). This extensive compilation offers a wide range of symbols used in various disciplines.
- Additionally, platforms like Detexify allow users to sketch a symbol, and the tool then identifies the corresponding LaTeX command.
- Engaging with online communities, such as the TeX - LaTeX Stack Exchange, can also be invaluable for finding specific symbols or seeking advice on LaTeX-related challenges.

6.2 Compiling your final report

To create a PDF to hand in, you'll need to compile, or knit, your entire markdown document as mentioned above. To knit your R markdown script, simply click the *knit* button in R Studio (yellow box, Figure 2). You can specify what output you would like and R Studio will (hopefully) compile your script. Remember, you can test or run individual code chunks as outlined in Running code in R Markdown.

6.3 Authoring with R Markdown

Below is a brief summary of the major elements required to author an R Markdown document. They shoudl address the majority of your needs, but please see the R Markdown resources for more information.

6.3.1 R markdown syntax

Unlike *Microsoft Word*, R Markdown utilizes a specific syntax for text formatting. Once you get used to it, it makes typing documents much easier than *Word*'s approach. The table below is how some of the most common text formatting is typed in your R Markdown document (syntax & example column) and how it'll appear in the final output.

Text formatting	syntax	Example	Example output
italics	*text*	this is *italics*	this is *italics*
bold	**text**	this is **bold**	this is **bold**
subscript	~text~	this is ~subscript~	this is $_{\text{subscript}}$
superscript	^text^	this is ^superscript^	this is $^{\text{superscript}}$
monospace	‘text‘	this is ‘monospaced‘	this is `monospace`

For a collection of other R Markdown syntax, please see the useful (and brief) list compiled online here. This includes ordered and unordered lists, headers, code blocks, hyperlinks, and figure captions.

6.3.2 R code chunk options

Your R code is run in chunks and the results will be embedded in the final output file. To each chunk you can specify options that'll effect how you're code chunk is run and displayed in the final output document. You include options in the chunk delimiters ```` ```{r} ```` and ```` ``` ````. For example the following code tells markdown you're running code written in `R`, that when you compile your document this code chunk should be evaluated, and that the resulting figure should have the caption "Some Caption".

````
```{r, eval = FALSE, fig.cap = "Some caption"}

# some code to generate a plot worth captioning.

```
````

The most common and useful chunk options are shown below. Note that they all have a default value. For example, `eval` tells R markdown whether the code within the block should be run. It's default option is `TRUE`, so by default any code in a chunk will be run when you knit your document. If you don't want that code to be run, but still displayed, you would set `eval = FALSE`. Another example would be setting `echo = FALSE` which allows the code to run, but the code *won't* be displayed on the output document (the outputs will still be displayed though); useful for creating clean documents with plots only (i.e. lab reports…).

option	default	effect
eval	TRUE	whether to evaluate the code and include the results
echo	TRUE	whether to display the code along with its results
warning	TRUE	whether to display warnings
error	FALSE	whether to display errors
message	TRUE	whether to display messages
tidy	FALSE	whether to reformat code in a tidy way when displaying it
fig.width	7	width in inches for plots created in chunk
fig.height	7	height in inches for plots created in chunk
fig.cap	NA	include figure caption, must be in quotation makrs (“ “)

6.3.3 Inserting images

Images not produced by R code can easily be inserted into your document. The markdown code isn't R code, so between paragraphs of bodytext insert the following code.

```
![Caption for the picture.](path/to/image.png){width=50%, height=50%}
```

Note that in the above the use of image atributes, the `{width=50%, height=50%}` at the end. This is how you'll adjust the size of your image. Other dimensions you can use include `px`, `cm`, `mm`, `in`, `inch`, and `%`.

A final note on images: when compiling to PDF, the LaTeX call will place your image in the “optimal” location (as determined by LaTeX), so you might find your image isn't exactly where you though it would be. A more in-depth guide to image placement can be found here

6.3.4 Generating tables

There are multiple methods to create tables in R markdown. Assuming you want to display results calculated through R code, you can use the `kable()` function.

Or you can consult the Summarizing Data chapter for making publication ready tables.

Alternatively, if you want to create simple tables manually use the following code in the main body, outside of an `R` code chunk. You can increase the number of rows/columns and the location of the horizontal lines. To generate more complex tables, see the `kable()` function and the `kableExtra` package.

```
Header 1 | Header 2| Header 3
---------|---------|---------|
Row 1    | Data    | Some other Data
Row 2    | Data    | Some other Data
---------|---------|---------|
```

	Header 1	Header 2	Header 3
Row 1	Data		Some other Data
Row 2	Data		Some other Data

We know this can be a tedious process. Luckily, there is a website that generates the markdown syntax when you input the values, and this can save your time trying to correctly format tables. Check it out here.

6.3.5 Spellcheck in R Markdown

While writing an R markdown document in R studio, go to the `Edit` tab at the top of the window and select `Check Spelling`. You can also use the `F7` key as a shortcut. The spell checker will literally go through every word it thinks you've misspelled in your document. You can add words to it so your spell checker's utility grows as you use it. **Note** that the spell check may also check your R code; be wary of changing words in your code chunks because you may get an error down the line.

6.3.6 Exporting R Markdown documents

You'll most likely be exporting your R Markdown documents as PDFs, but the beauty of R Markdown is it dosen't stop there. Your R Markdown documents can be knitted as a HTML document, a book (or both like this book!). You can even make slideshow presentations and yes, if need be, export as a word document that you can open in *Microsoft Word*.

You specify the output format in the document header. To specify you want your document to be outputted as a PDF your YAML header would look like this:

```
---
title: "Your title here"
```

```
output: pdf_document
---
```

Here are some links to different output formation available in R Markdown and how to use them:

- `pdf_document` creates a PDF document via Latex; probably your defacto output.
- `word_document` creates a Word document. Note that the formatting options are pretty basic, so while evereything will be where you want it to be, you'll need to pretty it up in Word to comply with your instructors specifications.
- `tufte_handout` for a PDF handout in the style of Edward Tufte. Check it out.
- `ioslides_presentation`, `revealjs::revealjs_presentation`, and `powerpoint_presentation` are all options to create sldieshow presentations. I personally use `revealjs` for my own work. It has the steepest learning curve of the bunch, but once setup, you can make incredibly slick slides with ease.
  - Note: like `word_document`, `powerpoint_presentation`'s outputs are stylistically simple. You'll definitly need to pretty them up manually in Powerpoint.

6.3.7 RStudio tips and tricks

To further the usefulness of R Markdown, the latest release of RStudio has a *Visual R Markdown* editor which introduces many useful features for authoring documents in R Markdown. Some of the most pertinent are:

- Visual editor so you can see how your document looks (top left of script pane)
- Combining Zotero and RStudio for easy citatiosn of your document (read more here

6.4 R Markdown resources

There's a plethora of helpful online resources to help hone your R markdown skills. We'll list a couple below (the titles are links to the corresponding document):

- Chapter 2 of the *R Markdown: The Definitive Guide* by Xie, Allair & Grolemund (2020). This is the simplest, most comprehensive, guide to learning R markdown and it's available freely online.
- *The R markdown cheat sheet*, a great resource with the most common R markdown operations; keep on hand for quick referencing.
- *Bookdown: Authoring Books and Technical Documents with R Markdown* (2020) by Yihui Xie. Explains the `bookdown` package which greatly ex-

pands the capabilities of R markdown. For example, the table of contents of this document is created with `bookdown`.

Chapter 7

R Packages

Before we do any real data work with R, now is the good time to introduce you to R packages.

R, as a powerful programming language for statistics and data analysis, boasts a rich ecosystem of packages. In this chapter, we'll demystify what these packages are, their importance, and how to utilize them efficiently.

7.1 What are R packages?

In a programming context, a package is akin to a toolbox. It contains sets of functions and data sets crafted to perform specific tasks, similar to how a toolbox contains various tools for different jobs. Instead of building every tool from scratch each time you need it, you can simply open your toolbox and grab the necessary instrument. In R, these tools come in the form of functions and data sets bundled inside packages.

Packages are previously written snippets of code that extend the capabilities of base R. Typically packages are created to address specific issues or workflows in different types of analysis.

7.1.0.1 Benefits of using packages:

- **Efficiency**: Why reinvent the wheel? Packages save time by offering tried and tested functions for specific tasks.
- **Community Support**: R packages often have a strong community of developers and users. This means frequent updates, thorough documentation, and a network of users to answer questions and offer support.
- **Versatility**: The vast library of R packages means that you have tools at your disposal for almost every conceivable task or analysis.

7.2 How to use R packages

7.2.1 How to install packages

Before using a package, you must first install it. This is a one-time process unless you need to update the package to a newer version. To install a package, use the `install.packages()` function.

This book will make frequent use of a family of packages called the `tidyverse` (a popular collection of packages for data manipulation). These packages all share a common thought process and integrate naturally with one another. If you want to install the package named "tidyverse", you would use:

```
# You can run this code in your R console as well.
install.packages("tidyverse")
```

7.2.2 How to load packages

You'll see a flurry of lines printed to the console indicating the status of the installation. Once installed you won't be able to use these functions until you load it with `library()`. For example, to load the `tidyverse` package:

```
# You can run this code in your R console as well.
library(tidyverse)
```

The output show's us which packages are included in the `tidyverse()` and their current version numbers, as well as conflicts (where functions from different packages share the same name). Don't worry about these for now.

After this, all the functions and data sets contained in the "tidyverse" package are available for you to use in your session. If you're ever uncertain about how to use a particular function, the R community and the package's documentation are excellent resources. For example, you can take a look at the official tidyverse documentation here.

7.2.3 Calling specific functions

We've called functions like `ggplot()` and `read_csv()` from the `ggplot2` and `readr` packages, respectively. When we did so, they were implicitly imported when we called `library(tidyverse)`. What library does is import *all* of the functions within a package into the R workspace, so we can simply refer to them by name later on. Sometimes you'll want to be explicit to which function you call, as you can run into conflicts where different functions from different packages have the same name. Or you might not want to import the entire package when you only need to call one function. Either way, to explicitly call a function from a specific package you type the package name, followed by `::`, and the function name. I.e. We can use `read_csv()` without importing the `tidyverse`/`readr` packages by simply typing: `readr::read_csv()`. Note the

package still needs to be installed on your computer for this to work.

7.3 Tidyverse: The Golden Toolbox of R

We've emphasized before that `tidyverse` is an indispensable collection of R packages tailored specifically for data science and in-depth data analysis. As you proceed, you'll find that our chapters heavily, if not exclusively, rely on its functionalities. Let's dive into some of its pivotal functions.

7.3.1 Loading the Tidyverse

Before using the functions from the `tidyverse`, make sure to load the entire collection.

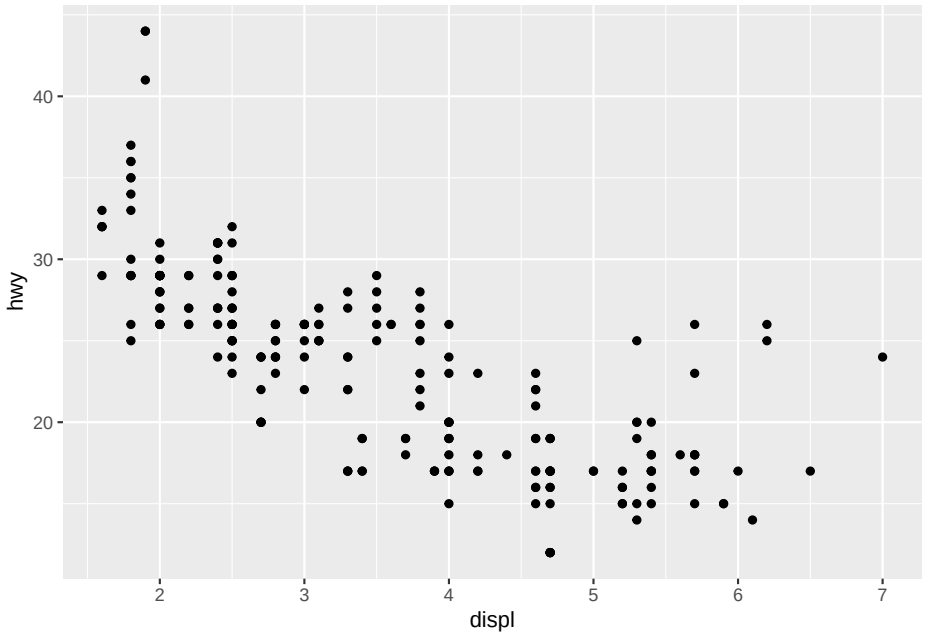
```
library(tidyverse)
```

7.3.2 Data Visualization

One of the strengths of the `tidyverse` is data visualization. The example below shows how you can iteratively build plots by adding layers of details.

```
# ggplot2 built-in dataset on fuel economy
data(mpg)

# draw a scatterplot
ggplot(mpg, aes(x=displ, y=hwy)) + geom_point()
```



7.3.3 Data Manipulation

The tidyverse provides a set of functions to help solve common data manipulation challenges. The syntax is intuitive and readable, which simplifies both writing and understanding the code.

```
mpg %>%
  filter(class == "suv") %>%
  summarize(mean_hwy = mean(hwy))
```

```
## # A tibble: 1 x 1
##   mean_hwy
##      <dbl>
## 1     18.1
```

Code explanation (Optional for now; For your curiosity):

- The `%>%` symbol (pipe operator) is a key part of tidyverse. The operator is used to pass an object into a function, allowing for sequential operations to be performed without the need for intermediate variables. For the example above, it's using the operator for data manipulation (filter and summarizing). The result of the left-hand side of the `%>%` is used as the first argument to the function on the right-hand side.
- The `filter()` function is used to extract a subset of rows from a data frame based on logical conditions. It returns all rows where the condition is `TRUE`.
- The `summarize()` function is used to create summary statistics for different variables. You provide named arguments where the name will be the name of a new column, and the value will be the summary statistic to compute.

7.3.4 Efficient Data Reading

The `tidyverse` allows for efficient reading of rectangular data, such as CSV and TSV files. For instance, instead of using the R built-in function `read.csv()`, you can employ a similar function, `read_csv()`, from `tidyverse` to quickly import a CSV file as a data frame. This function provides more flexibility and is optimized for faster performance.

7.4 Smile: Handling Chemical Structures in R

Let's look at one more package that you might find interesting: `smile`.

7.4.1 Installation

```
# to be added
```

7.4.2 A simple example

```
# to be added
```

7.5 Summary

In this chapter, we delved into the world of R packages, understanding their significance and advantages over built-in R functions. We highlighted the functionalities of `tidyverse`, showcasing practical examples.

Armed with this foundational knowledge of R packages, we're now poised to harness their capabilities in our R programming journey.

7.6 Exercise

There is a set of exercises available for this chapter!

Not sure how to access and work on the exercise Rmd files?

- Refer to Chapter 3.3 for step-by-step instructions on accessing the exercises and working within the UofT JupyterHub's RStudio environment.
- Alternatively, if you'd like to simply access the individual files, you can download them directly from this repository.

Always remember to save your progress regularly and consult the textbook's guidelines for submitting your completed exercises.

Chapter 8

Loops and Functions in R

In this chapter, we will delve into two fundamental concepts in R programming: loops and writing functions. Loops are a powerful tool for iterating over a sequence, making them essential for tasks that require repetitive operations. Writing functions, on the other hand, allows you to create reusable blocks of code, enhancing the efficiency and organization of your programming projects. By the end of this chapter, you will gain a solid understanding of how to effectively use for loops to automate repetitive tasks and how to write your own functions in R, thereby elevating your coding skills to a new level.

8.1 Loops

Repetition is a key component of programming. Loops enable you to execute the same piece of code multiple times, making data processing more efficient. If you have a previous programming experience in any language, these syntax should look familiar.

8.1.0.1 The `for` loop

The `for` loop in R is used to iterate over elements in a vector or list.

```
for (i in 1:5) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

This will print numbers from 1 to 5. The loop iterates over each element in the sequence `1:5`, setting the value to `i` and executing the code inside the loop.

8.1.0.2 The `while` loop

The `while` loop continues executing as long as a specified condition remains true.

```
count <- 1

while (count <= 5) {
  print(count)
  count <- count + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

In this example, the loop will keep printing and incrementing the value of counter until counter is no longer less than or equal to 5.

Remember, be cautious with while loops. If the condition never becomes false, the loop will run indefinitely!

8.2 Writing custom functions in R

Functions form the backbone of most programming languages, and R is no exception. While R provides a rich library of built-in functions, there are times when you'll need to define your own. Custom functions in R allow you to encapsulate a series of commands into a single reusable unit.

Why write custom functions?

- **Reusability**: Once you've written and tested a function, you can use it repeatedly without having to retype or copy-paste the same lines of code.
- **Maintainability**: Changes can be made in one place (inside the function) rather than at multiple locations where the code might be used.
- **Clarity**: Well-named functions can make your main code more readable, as they abstract away the complexity.

8.2.0.1 Simple example

```
add_two <- function(a, b) {
  # Add a and b and return the value
  added <- a + b
```

```
  return(added)
}
```

- `add_two` is the custom name of the function.
- `a` and `b` are the inputs to the function. You can name them `x` and `y`, or anything else, as long as they are kept consistent throughout the function definition. You can also change the number of inputs for your function.
- The `return` statement specifies the output of your function. If omitted, the function will return the result of the last expression evaluated.

You can run this function with various choices of `a` and `b`:

```
add_two(1, 3)
```

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

```
add_two(20, 35.5)
```

```
## [1] 55.5
```

At this time, we shall keep it simple. You will eventually see more complex usages of custom functions.

8.3 Conditional arguments

Condaitional arguments used to specify a path in a function depending on whether a statement is `TRUE` or `FALSE`. These are explored in greater detail via the links in the Further reading section, but here's a quick example of a function that uses the conditional `if` statement to print out which number is largest:

```
isGreater <- function(x, y){
  if(x > y){
    return(paste(x, "is greater than", y, sep = " "))
  } else if (x < y){
    return(paste(x, "is less than", y, sep = " "))
  }
  return(paste(x, "is equal to", y, sep = " "))
}

isGreater (2, 1)
```

```
## [1] "2 is greater than 1"
```

```
isGreater (1, 2)
```

```
## [1] "1 is less than 2"
```

```
isGreater (1, 1)
```

```
## [1] "1 is equal to 1"
```

Our simple function compares two numbers, `x` and `y` and if `x > y` evaluate to `TRUE` it returns the pasted string `x is greater than y`. If `x < y` evaluates to `FALSE`, as in `y > x`, our function returns the pasted string `x is less than y`, and finally if neither `x > y` and `x < y` evaluate to TRUE, they must be equal! Therefore the final output is `x is equal to y`. This is an example of an `else if` statement. If you're simply evaluating two conditions (`TRUE` or `FALSE`) you only need the `if()` conditional, see Further reading for more details.

8.3.1 Piping conditional statements

You can already see the potential for simple conditional statements in the pipe. However, to keep piping operations legible, `dplyr` offers the `case_when` function, which works similarly to the `else if` statements showcased above. Let's see how it works using a real world example.

In mass spectrometry undetected compounds are recorded by the instrument as having an intensity of 0; but it's a common practice to replace 0 with $\frac{limit\ of\ detection}{2}$ for subsequent analysis However, we don't want to replace every value with $\frac{LOD}{2}$, only 0s. Let's use the `case_when()` function to create a new values with the recorded intensities

```
lod <- 4000 # previously calculated LOD
results <- data.frame("mz" = c(308.97, 380.81, 410.11, 445.34 ), # dummy data
                      "intensities" = c(0, 1000, 5000, 10000))

results %>%
  mutate(reportedIntensities = case_when(intensities < lod ~ lod/2,
                                         TRUE ~ intensities))
```

```
##       mz intensities reportedIntensities
## 1 308.97           0                2000
## 2 380.81        1000                2000
## 3 410.11        5000                5000
## 4 445.34       10000               10000
```

Firstly we're creating a new column called `reportedIntesities` using `mutate()` and using `case_when()` to conditionally fill that column. The inputs we've passed to `case_when()` are two-sided formulas. Essentially if the conditions on the left-hand side of the tilda (`~`) evaluate to `TRUE`, `case_when` will execute the right-hand side. Thee first two-sided formula is `intensities < lod ~ lod/2` and checks if the intensities value is less than the previously calculated limit of detection. If `intensitis < lod` evaluates to `TRUE` we insert half of the LOD value for that row. If `intentisites < lod` evaluates to `FALSE`, we move onto the next two-side formula and reevaluate again. The second two-sided formula

`TRUE ~ intensities` basically means for everything that's remaining (greater than LOD in our instance) just use the value from the `intensities` column.

Some ideas to consider when working with `case-when()`:

- There's no limits to the conditions you can pass to `case_when()`.
- *However* `case_when()` evaluates in order so put the more specific conditions before the more general.
- Remember that the point of `case_when()` and piping is legibility. If you're passing multiple conditions, consider writing a function using `else if` statements to keep the pipe legible.

8.4 Further reading

These chapter has been intentional succinct. We've omitted several other aspects of programming in R such as `for` loops, and other iterative programming. To get a better sense of programming in R and to learn more, please see the following links:

- case_when(): the documentation for the `case_when()` function and several useful examples.
- Chapter 19: Functions, Chapter 20: Vectors, and Chapter 21: Iteration of *R for Data Science* by H. Wickham and G. Grolemund.

- Hands-on Programming in R by G. Grolemund for a more in-depth (but still approachable) take on programming in R.

Part 3: Data Analysis in R

Chapter 9

Intro to Data Analysis

Now, we will embark on a comprehensive journey through the data analysis process, focusing on the essential steps of data wrangling and advanced analytical techniques. The chapter is designed to equip you with the necessary skills to use R effectively for organizing and transforming your data, a crucial foundation for any data analysis project. We will delve into the core workflow that is applicable to every data analysis task, regardless of its complexity or duration. This workflow is not just a one-time learning curve but a set of skills you will repeatedly use across various projects.

The explicit workflow we'll be teaching was originally described by Wickham and Grolemund, and consists of six key steps:

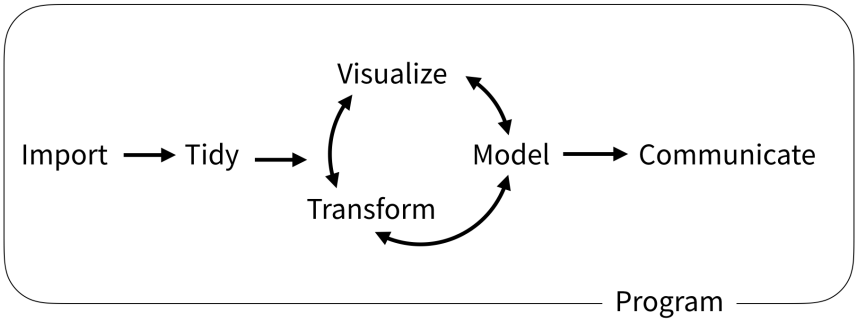


Figure 9.1: Data science workflow describes by Wickham and Grolemund; image from *R for Data Science*, Wickham and Grolemund (2021)

- **Import** is the first step and consist of getting your data into R. Seems

obvious, but doing it correctly will save you time and headaches down the line.
- **Tidy** refers to organizing your data in a *tidy* manner where each variable is a column, and each observation a row. This is often the least intuitive part about working with R, especially if you've only used Excel, but it's critical. If you don't tidy your data, you'll be fighting it every step of the way.
- **Transform** is anything you do to your data including any mathematical operations or narrowing in on a set of observations. It's often the first stage of the cycle as you'll need to transform your data in some manner to obtain a desired plot.
- **Visualize** is any of the plots/graphics you'll generate with R. Take advantage of R and plot often, it's the easiest way to spot an error.
- **Model** is an extension of mathematical operations to help understand your data. The *linear regressions* needed for a calibration curve are an example of a model.
- **Communicate** is the final step and is where you share the *knowledge* you've squeezed out of the information in the original data.

Import, *Tidy*, and *Transformation* go hand-in-hand in a process called *wrangling*. Wrangling is all of the steps needed to get your data ready for analysis. It's often the most tedious and frustrating, hence wrangling (it's a fight…), but once done make the subsequent cycle of understanding your data via *transformation*, *visualizations*, and *modelling* much easier and more predictable.

9.1 Example Data

Throughout this section and the next we'll be making use of a couple of example datasets. These datasets are all available in the `data` subfolder of the *R4EnvChem Project Template*. If you haven't already, read Importing a project for instructions on dowloading the repository and data.

9.2 Sneak Peek at Data Analysis

In this trailer section, we'll explore how data analysis can provide insights into real-world phenomena. We'll use the `storm` dataset from the `tidyverse` suite in R to investigate the patterns and relationships of storms over the years.

9.2.1 Setting Up

Let's begin by loading our dataset and essential packages.

```
library(tidyverse)

# Import the storm dataset
data(storms)
```

Let's see how the data looks:

```
DT::datatable(storms)
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```

9.2.2 Tidying Our Data

Now that you saw how the data looks, before delving into deeper analysis, it's important to have a general sense of our data.

To narrow our focus, let's consider the storm `name`, `year`, `month`, `day`, `lat`, `long`, and `wind` speed.

```
# 1. Selecting relevant variables

selected_storms <- storms %>%
  select(name, year, month, day, lat, long, wind)

DT::datatable(selected_storms)
```

Next, instead of having separate columns for `year`, `month`, and `day`, it might be more useful to have a single date column.

```
# 2. Creating a unified date column

storms_with_date <- selected_storms %>%
  unite("date", year, month, day, sep = "-") %>%
  mutate(date = as.Date(date, format = "%Y-%m-%d"))

DT::datatable(storms_with_date)
```

Column names should be self-explanatory. Let's rename `wind` to `wind_speed_knots`.

```
# 3. Renaming columns for clarity

tidied_storms <- storms_with_date %>%
  rename(wind_speed_knots = wind)
```

Now that our data is tidied up, let's see if we can do more in-depth analysis.

9.2.3 Investigating Storm Patterns

9.2.3.1 Analyzing powerful storms

For safety and preparedness reasons, meteorologists are often interested in particularly powerful storms. Let's identify storms with wind speeds exceeding 100 knots.

```
powerful_storms <- tidied_storms %>%
  filter(wind_speed_knots > 100) %>%
  arrange(desc(wind_speed_knots))
```

9.2.3.2 Yearly trends

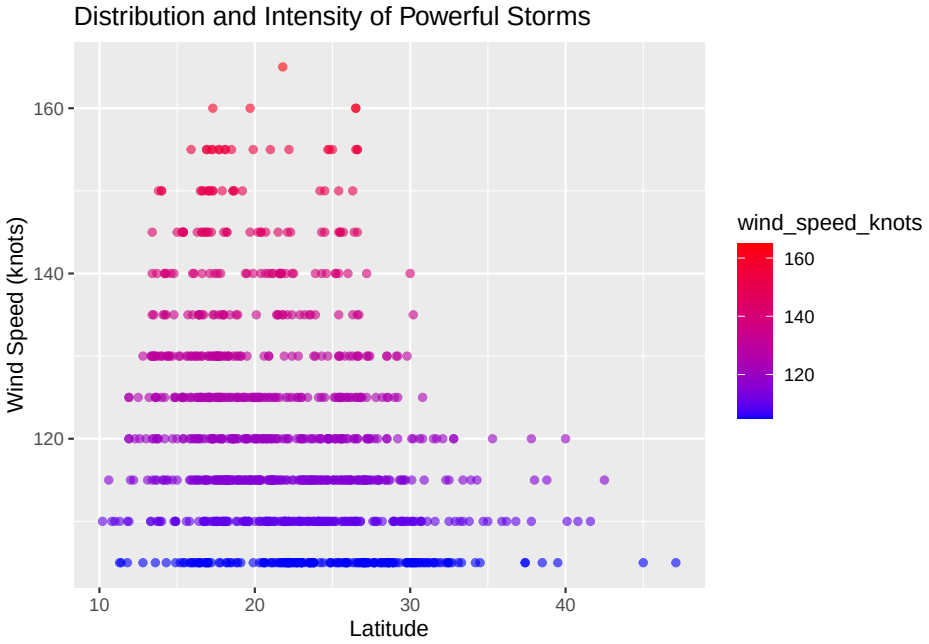
How has the frequency of these powerful storms changed over the years? We can group our data by year to answer this.

```
yearly_storms <- powerful_storms %>%
  mutate(year = year(date)) %>%
  group_by(year) %>%
  summarise(storm_count = n())
```

9.2.4 Visualization: Geographical Distribution

One of the key aspects of understanding storms is analyzing where they occur. Using our powerful storms data, let's plot a scatterplot of `latitude` versus `wind_speed_knots` to visualize their geographical distribution and intensity.

```
ggplot(data = powerful_storms, aes(x = lat, y = wind_speed_knots)) +
  geom_point(aes(color = wind_speed_knots), alpha = 0.6) +
  ggtitle("Distribution and Intensity of Powerful Storms") +
  xlab("Latitude") +
  ylab("Wind Speed (knots)") +
  scale_color_gradient(low = "blue", high = "red")
```



Through this brief exploration, we've seen how data analysis can provide insights into storm patterns. With more advanced techniques, which we'll explore in the subsequent chapters, we can delve even deeper, helping inform decisions, ensuring preparedness, and advancing our understanding of meteorological phenomena.

9.3 Further Reading

In case it hasn't been apparent enough, this entire endeavour was inspired by the *R for Data Science* reference book by Hadley Wickham and Garrett Grolemund. Every step described above is explored in more detail in their book, which can be read freely online at https://r4ds.had.co.nz/. We strongly encourage you to read through the book to supplement your R data analysis skills.

Chapter 10

Importing Your Data Into R

Unlike *Excel*, you can't copy and paste your data into R (or RStudio). Instead you need to *import* your data into R so you can work with it. This chapter will discuss how your data is stored, and how to import it into R (with some accompanying nuances).

10.1 .csv files

While there are a myriad of ways data is stored, raw instrument often record results in a proprietary vendor format, the data you're likely to encounter in an undergraduate lab will be in the form of a `.csv` or *comma-separated values* file. As the name implies, values are separated by commas (go ahead and open any `.csv` file in any text editor to observe this). Essentially you can think of each line as a row and commas as separating values into columns, which is exactly how R and *Excel* handle `.csv` files.

10.2 read_csv

Importing a `.csv` file into R simply requires the `read.csv` or the `read_csv` function from tidyverse. The first variable is the most important as it's the file path. Recall that R, unless specified, uses relative referencing. So in the example below we're importing the `ATR_plastics.csv` from the `data` sub-folder in our project by specifying `"data/ATR_plastics.csv"` and assigning it to the variable `atr_plastics`. Note the inclusion of the file extension.

```
atr_plastics <- read_csv("data/ATR_plastics.csv")
```

```
## Rows: 7157 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (5): wavenumber, EPDM, Polystyrene, Polyethylene, Sample: Shopping bag
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

A benefit of using `read_csv` is that it prints out the column specifications with each column's name (how you'll reference it in code) and the column value type. Columns can have different data types, but a data type must be consistent within any given column. Having the columns specifications is a good way to ensure R is correctly reading your data. The most common data types are:

- **int** for integer values (*-1,1, 2, 10, etc.*)
- **dbl** for doubles (decimals) or real numbers (*-1.20, 0.0, 1.200, 1e7, etc.*)
- **chr** for character vectors or strings (*"A", "chemical", "Howdy ma'am", etc.*)
  - note numbers can be encoded as strings, so while you might read "1" as a number, R treats it as a character, limiting how you can use this value.
- **lgl** for logical values, either `TRUE` or `FALSE`

We can also quickly inspect either through the *Environment* pane in *RStudio* or quickly with the `head()` function. Note the column specifications under the column name.

```
head(atr_plastics)
```

```
## # A tibble: 6 x 5
##   wavenumber  EPDM Polystyrene Polyethylene `Sample: Shopping bag`
##        <dbl> <dbl>       <dbl>        <dbl>                  <dbl>
## 1       550. 0.212      0.0746     0.000873                 0.0236
## 2       551. 0.212      0.0746     0.000834                 0.0238
## 3       551. 0.213      0.0745     0.000819                 0.0239
## 4       552. 0.213      0.0745     0.000825                 0.0239
## 5       552. 0.214      0.0745     0.000868                 0.0240
## 6       553. 0.214      0.0746     0.000949                 0.0240
```

As you can see, the `head()` function, by default, shows the first six rows of the dataframe. If you want to inspect more or fewer rows, you can provide an optional `n` argument like `head(data, n=10)`. Note the column specifications under the column name.

Also note how the first line of the `ATR_plastics.csv` has been interpreted as columns names (or *headers*) by R. This is common practice, and gives you a handle by which you can manipulate your data. If you did not intend for R to interpret the first row as headers you can suppress this with the additional argument `col_names = FALSE`.

```
head(read_csv("data/ATR_plastics.csv", col_names = FALSE))
```

```
## Rows: 7158 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (5): X1, X2, X3, X4, X5
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 6 x 5
##   X1         X2        X3          X4           X5
##   <chr>      <chr>     <chr>       <chr>        <chr>
## 1 wavenumber EPDM      Polystyrene Polyethylene Sample: Shopping bag
## 2 550.0952   0.2119556 0.07463058  0.000873196  0.02364882
## 3 550.5773   0.2124079 0.07455246  0.000834192  0.02382648
## 4 551.0594   0.2128818 0.07450471  0.000819447  0.02387163
## 5 551.5415   0.2133267 0.07449704  0.000825491  0.02391921
## 6 552.0236   0.2137241 0.07452058  0.000868397  0.02396947
```

Note in the example above that since the headers are now considered data, and are composed of a string of chatacters, the entire column is then interpreted as character values. This will happen if a single non-numeric character is introduced in the column, so beware of typos when recording data! If we wanted to skip rows (i.e. to avoid blank rows at the top of our `.csv`), we can use the `skip = n` to skip n rows:

```
head(read_csv("data/ATR_plastics.csv", col_names = FALSE, skip = 1))
```

```
## Rows: 7157 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (5): X1, X2, X3, X4, X5
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 6 x 5
##      X1    X2     X3       X4     X5
##   <dbl> <dbl>  <dbl>    <dbl>  <dbl>
## 1  550. 0.212 0.0746 0.000873 0.0236
## 2  551. 0.212 0.0746 0.000834 0.0238
## 3  551. 0.213 0.0745 0.000819 0.0239
## 4  552. 0.213 0.0745 0.000825 0.0239
## 5  552. 0.214 0.0745 0.000868 0.0240
## 6  553. 0.214 0.0746 0.000949 0.0240
```

Note in the example above that we skipped our headers, so `read_csv()` created

placeholder headers (`X1`, `X2`, etc.).

Another useful function to inspect data is `tail()`, which displays the last six rows of a dataframe. Similarly, it accepts an optional `n` argument to specify the number of rows you want to view.

```
tail(atr_plastics)
```

```
## # A tibble: 6 x 5
##   wavenumber  EPDM Polystyrene Polyethylene `Sample: Shopping bag`
##        <dbl> <dbl>       <dbl>        <dbl>                  <dbl>
## 1      3998. 0.113      0.0706  0                          0.0664
## 2      3998. 0.113      0.0706  0.000000582                0.0664
## 3      3999. 0.113      0.0706  0.0000242                  0.0664
## 4      3999. 0.113      0.0706  0.000067                   0.0664
## 5      4000. 0.113      0.0706  0.000125                   0.0664
## 6      4000. 0.113      0.0706  0.000195                   0.0663
```

10.2.1 Tibbles vs. data frames

Quick eyes will notice the first line outputted above is `# A tibble: 6 x 5`. `tibbles` are a variation of `data.frames` introduced in section one, but built specifically for the `tidyverse` family of packages. While `data.frames` and `tibbles` are often interchangeable, it's important to be aware of the difference in case you do run into a rare conflict. In these situations you can readily transform a `tibble` into a `data.frame` by coercion with the `as.data.frame()` function, and vice-versa with the `as_tibble()` function.

```
class(as.data.frame(atr_plastics))
```

```
## [1] "data.frame"
```

10.3 Importing other data types

There are other functions to import different types of tabular data which all function like `read_csv`, such as `read_tsv` for tab-separate value files (`.tsv`) and `read_excel` and `read_xlsx` from the `readxl` package to import *Excel* files. Note most *Excel* files have probably been formatted for legibility (i.e. merged columns), which can lead to errors when importing into R. If you plan on importing *Excel* files, it's probably best to open them in *Excel* to remove any formatting, and then save as `.csv` for smoother importing into R.

10.4 Saving data

As you progress with your analysis you may want to save intermediate or final datasets. This is readily accomplished ussing the `write_csv()` from the `readr` package. Similar rules apply to how we used `read_csv`, but now the second

argument specifies the save location and file name, while the first argument is which `tibble`/`data.frame` we're saving. Note that R *will not* create a folder this way, so if you're saving to a sub-folder you'll have to make sure it exists or create it yourself.

```
write_csv(atr_plastics, "data/ATRSaveExample.csv")
```

A benefit of `write_csv` is that it will always save in `UTF-8` encoding and ISO8601 time format. This standardization makes it easier to share your `.csv` files with collaborators/yourself.

10.5 Further Reading

See Chapters 10 and 11 of *R for Data Science* for some more details on `tibbles` and `read_csv`.

10.6 Exercise

There is a set of exercises available for this chapter!

Not sure how to access and work on the exercise Rmd files?

- Refer to Chapter 3.3 for step-by-step instructions on accessing the exercises and working within the UofT JupyterHub's RStudio environment.
- Alternatively, if you'd like to simply access the individual files, you can download them directly from this repository.

Always remember to save your progress regularly and consult the textbook's guidelines for submitting your completed exercises.

Chapter 11

Tidying Your Data

You might not have explicitly thought about how you store your data, whether working in *Excel* or elsewhere. Data is data after all. But having your data organized in a systematic manner that is conducive to your goal is paramount for working not only with R, but all of your experimental data. This chapter will introduce the concept of *tidy* data, and how to use some of the tools in the *dplyr* package to get there. Lastly we'll offer some tips for how you should record *your* data in the lab. A bit of foresight and consistency can eliminate hours of tedious work down the line.

11.1 What is tidy data?

Tidy data has "…each variable in a column, and each observation in a row…" (Hadley Wickham 2014) This may seem obvious to you, but let's consider how data is often recorded in lab, as exemplified in Figure **??**A. Here the instrument response of two chemicals (*A* and *B*) for two samples (*blank* and *unknown*) are recorded. Note how the samples are on each row and the chemical are columns. However, someone else may record the same data differently as shown in Figure **??**B, with the samples occupying distinct columns, and the chemical rows. Either layout may work well, but analyzing both would require re-tooling your approach. This is where the concept of *tidy* data comes into play. By reclassifying our data into *observations* and *variables* we can restructure out data into a common format: the *tidy* format (Figure **??**C).

In the *tidy* or *long* format, we reclassified out data into three variables (*Sample*, *Chemical*, and *Reading*). This makes the observations clearer as now we know we measured two chemicals (*A* and *B*) in two samples (*blank* and *unknown*) and we've explicitly declared the *Reading* variable for our measured instrument response, which was only implied in the original layouts. Moreover, we can read across a row to get the gist of one data point (i.e. "Our blank has a

A.

Sample	Chemical A	Chemical B
blank	0	0
unknown	1	2

B.

Chemical	blank	unknown
A	0	1
B	0	2

C.

Sample	Chemical	Reading
blank	A	0
blank	B	0
unknown	A	1
unknown	B	2

Figure 11.1: (A and B) The same data can be recorded in multiple formats. (C) The same data in the tidy format. Note how the tidy data typically has more rows, hence why it's sometimes refered to as 'long' data.

reading of 0 for Chemical A"). Again we haven't changed any information, we've simply reorganized our data to be clearer, consistent, and compatible with the `tidyverse` suit of tools.

This might seem pedantic now, but as you progress you'll want to reuse code you've previously written. This is greatly facilitated by making every data set as consistently structured as possible, and the *tidy* format is an ideal starting place.

11.2 Tools to tidy your data

Now one of the more laborious parts of data science is tidying your data. If you can follow the tips in the Tips for recording data section, but the truth is you often won't have control. To this end, the `tidyverse` offers several tools, notable `dplyr` (pronounces 'd-pliers'), to help you get there.

Let's revisit our spectroscopy data from the previous chapter:

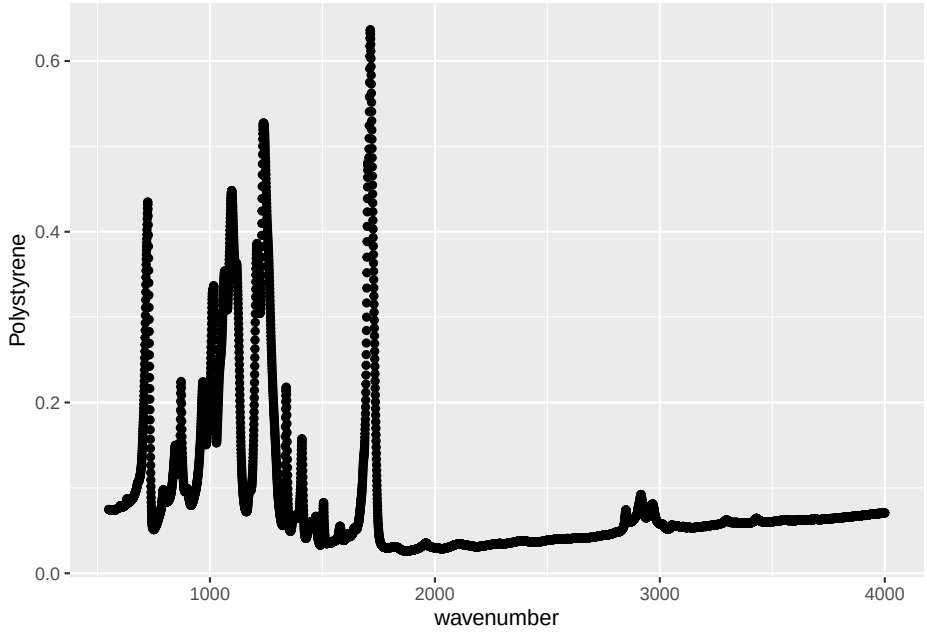
```
atr_plastics <- read_csv("data/ATR_plastics.csv")

# This just outputs a table you can explore within your browser
DT::datatable(atr_plastics)
```

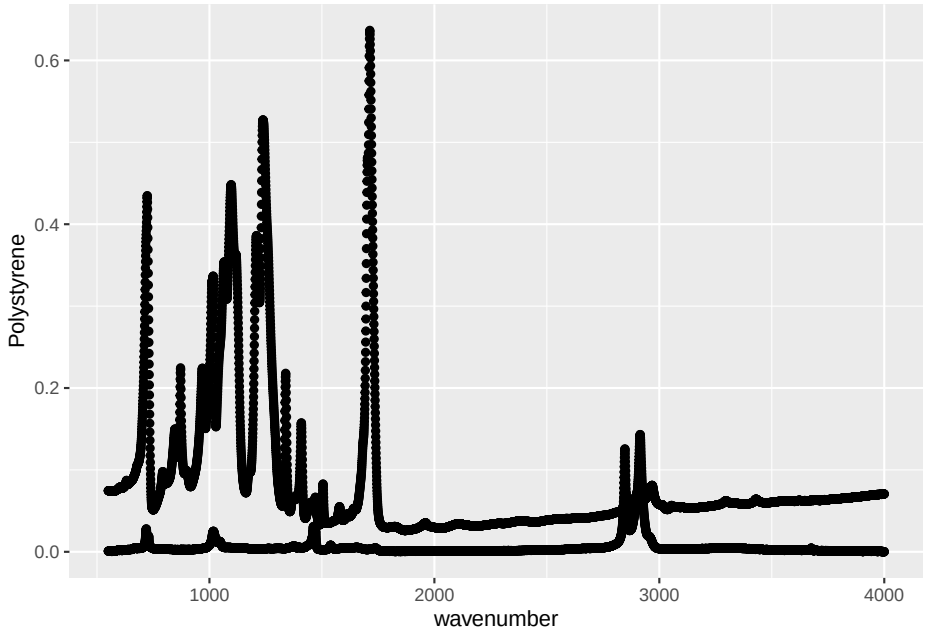
As we can see this our ATR spectroscopy results of several plastics, as recorded for a *CHM 317* lab, is structured similarly to the example in Figure **??**A. The ATR absorbance spectra of the four plastics are recorded in separate columns. Again, this format makes intuitive sense when recording in the lab, and for

working in Excel, but isn't the friendliest with R. When making plots with `ggplot`, we can only specify one `y` variable. In the example plot below it's the absorbance spectrum of `Polystyrene`. However, if wanted to plot the other spectra for comparison, we'd need to repeat our `geom_point` call.

```
# Plotting Polystyrene absorbance spectra
ggplot(data = atr_plastics,
       aes( x = wavenumber,
            y = Polystyrene)) +
  geom_point()
```




```
# Plotting Polystyrene and Polyethylene absorbance spectra
ggplot(data = atr_plastics,
       aes( x = wavenumber,
            y = Polystyrene)) +
  geom_point() +
  geom_point(data = atr_plastics,
             aes(x = wavenumber,
                 y = Polyethylene))
```



11.2.1 Selection helpers

As you've already seen, there are multiple ways to select columns and variables with the `dplyr` package. For a complete rundown of other useful helper functions please see Subset columns using their names and types. `starts_with()` for selecting columns from a prefix, and `contains()` for selecting columns that contain a string are two of the most useful.

11.2.2 Seperating columns

Sometimes your data has already been recorded in a tidy-ish fashion, but there may be multiple observations recorded under one apparent variable, something like `1 mM` for concentration. As it stands we cannot easily access the numerical value in the concentration recording because R will encode this as a string due to the `mM`. We can **separate** data like this using the `seperate` function, which operates similarly to how `pivot_longer` breaks up headers.

```
# Example with multiple encoded observations
sep_example
```

```
##          sample reading
## 1  Toronto_O3_1      10
## 2  Toronto_O3_2      22
## 3 Toronto_NO2_1      30
```

The example above is something you'll come across in the lab, most often with

the sample names you'll pass along to your TA where you crammed as much information as possible into that name so you and your TAs know exactly what's being analyzed. In this example, the sample name contains the location (`Toronto`), the chemical measured (`O3` or `NO2`) and the replicate number (i.e. `1`). Using the `seperate` function we can split up these three observations so we can properly group our data later on in our analysis.

```
# Separating observations

sep_data <- separate(sep_example,
    col = sample,
    into = c("location", "chemical", "replicateNum"),
    sep = "_",
    remove = TRUE,
    convert = TRUE)

sep_data
```

```
##   location chemical replicateNum reading
## 1  Toronto       O3            1      10
## 2  Toronto       O3            2      22
## 3  Toronto      NO2            1      30
```

Again, let's break down what we did with the `separate` function:

1. `col = sample` specifies we're selecting the `sample` column
2. `into = c(...)` specifies what columns we're separating our name into.
3. `sep = "_"1` specifies that each element is separated by an underscore (_); you can use `sep = " "` if they were separated by spaces.
4. `remove = TRUE` removes the original sample column, no need for duplication; setting this to `FALSE` would keep the original column.
5. `convert = TRUE` converts the new columns to the appropriate data format. In the original column ,the replicate number is a character value because it's part of a string, `convert` ensures that it'll be converted to a numerical value.

Another example why it's paramount to **be consistent when recording data**.

11.2.3 Uniting/combining columns

The opposite of the `separate` function is the `unite` function. You'll use it far less often, but you should be aware of it as it may come in handy. You can use it for combining strings together, or prettying up tables for publication/presentations as shown in Summarizing Data.

```
# Uniting observations

united_data <- unite(sep_data,
```

```
                      col=sample_reunited,
                      location:chemical:replicateNum,
                      sep = "_",
                      remove = TRUE)

united_data
```

```
##   sample_reunited reading
## 1    Toronto_O3_1      10
## 2    Toronto_O3_2      22
## 3   Toronto_NO2_1      30
```

You can read more about the `unite` function here.

11.2.4 Renaming columns/headers

Sometimes a name is lengthy, or cumbersome to work with in R. While something like `This_is_a_valid_header` is valid and compatible with R and tidyverse functions, you may want to change it to make it easier to work with (i.e. less typing). Simply use the `rename()` function:

Inspect the original column names:

```
colnames(badHeader)
```

```
## [1] "UVVis_Wave_Length_nM" "Absorbance"
```

Use `rename()` to change the column name and save the result to a new dataframe:

```
renamed_data <- rename(badHeader, wavelength_nM = UVVis_Wave_Length_nM)

# Inspect the column names of the renamed dataframe
colnames(renamed_data)
```

```
## [1] "wavelength_nM" "Absorbance"
```

11.2.5 Chaining multiple operations

So far we learned some standalone functions that can tidy up your data. But what if you want to do multiple of these operations to a dataset?

Let's start by talking about the seemingly intuitive but tedious approach. We can transform data by breaking down the process into individual steps:

```
selected_data <- select(atr_plastics, wavenumber, EPDM, `Sample: Shopping bag`)
longer_data <- pivot_longer(selected_data, cols = -wavenumber, names_to = "Material_Ty
atr_plastics_transformed <- rename(longer_data, Wave_Num = wavenumber)

atr_plastics_transformed
```

```
## # A tibble: 14,314 x 3
##    Wave_Num Material_Type         Value
##       <dbl> <chr>                 <dbl>
##  1     550. EPDM                 0.212
##  2     550. Sample: Shopping bag 0.0236
##  3     551. EPDM                 0.212
##  4     551. Sample: Shopping bag 0.0238
##  5     551. EPDM                 0.213
##  6     551. Sample: Shopping bag 0.0239
##  7     552. EPDM                 0.213
##  8     552. Sample: Shopping bag 0.0239
##  9     552. EPDM                 0.214
## 10     552. Sample: Shopping bag 0.0240
## # i 14,304 more rows
```

Now, let's see how we can transform the same `atr_plastics` tibble using the `%>%` operator by chaining operations.

```
atr_plastics_transformed <- atr_plastics %>%
    select(wavenumber, EPDM, `Sample: Shopping bag`) %>%
    pivot_longer(cols = -wavenumber, names_to = "Material_Type", values_to = "Value") %>%
    rename(Wave_Num = wavenumber)

atr_plastics_transformed
```

```
## # A tibble: 14,314 x 3
##    Wave_Num Material_Type         Value
##       <dbl> <chr>                 <dbl>
##  1     550. EPDM                 0.212
##  2     550. Sample: Shopping bag 0.0236
##  3     551. EPDM                 0.212
##  4     551. Sample: Shopping bag 0.0238
##  5     551. EPDM                 0.213
##  6     551. Sample: Shopping bag 0.0239
##  7     552. EPDM                 0.213
##  8     552. Sample: Shopping bag 0.0239
##  9     552. EPDM                 0.214
## 10     552. Sample: Shopping bag 0.0240
## # i 14,304 more rows
```

We will formally introduce the unfamiliar operator `%>%` (pipe) in the next chapter (check out 12.6 if interested). For now, just remember that there is a way to chain your functions like the above example!

11.3 Tips for recording data

In case you haven't picked up on it, tidying data in R is much easier if the data is recorded consistently. You can't always control how your data will look, but in the event that you can (i.e. your inputting the instrument readings into *Excel* on the bench top) here are some tips to make your life easier:

- *Be consistent.* If you're naming your samples make sure they all contain the same elements in the same order. The sample names `Toronto_O3_1` and `Toronto_O3_2` can easily be broken up as demonstrated in [Separating columns]; `O3_Toronto_1`, `TorontoO32`, and `Toronto_1` can't be.
- *Use as simple as possible headers.* Often you'll be pasting instrument readings into one `.csv` using *Excel* on whatever computer records the instrument readings. In these situations it's often much easier to paste things in columns. Recall the capabilities of `pivot_longer` and how you can break up names as described in [Making data 'longer']. `Chemical_A_1` and `Chemical_B_2` are headers that are descriptive for your sample and can be easily pivoted into their own columns. `Chemical A 1 ( I think?!)` is a header that isn't.
- *Make sure data types are consistent within a column.* This harks back to the [Importing data into R] chapter, but a single non-numeric character can cause R to misinterpret an entire column leading to headaches down the line.
- *Save your data in UTF-8 format.* Excel and other programs often allow you to export your data in a variety of `.csv` encodings, but this can affect how R reads when importing your data. Make sure you select `UTF-8` encoding when exporting your data.

11.4 Further reading

As always, the *R for Data Science* book goes into more detail on all of the elements discussed above. For the material covered here you may want to read Chapter 9: Tidy Data.

11.5 Exercise

There is a set of exercises available for this chapter!

Not sure how to access and work on the exercise Rmd files?

- Refer to Chapter 3.3 for step-by-step instructions on accessing the exercises and working within the UofT JupyterHub's RStudio environment.
- Alternatively, if you'd like to simply access the individual files, you can download them directly from this repository.

Always remember to save your progress regularly and consult the textbook's guidelines for submitting your completed exercises.

Chapter 12

Resturcturing Your Data

In the realm of data analysis, the structure of your data can be just as crucial as the data itself. When dealing with complex datasets in R, or any other analytical environment, the way you organize and reshape your data can significantly impact the efficiency and clarity of your analyses. This chapter delves into the art of restructuring data in R, focusing on two powerful functions: `pivot_longer` and `pivot_wider`. These tools, part of the `tidyr` package, are essential for transforming data into a format that aligns perfectly with your analytical objectives.

Understanding and mastering these functions will equip you with the skills to seamlessly toggle between different data layouts. Whether you need to condense wide datasets into longer, more detailed formats using `pivot_longer`, or expand long datasets into a wider, more summarized form with `pivot_wider`, this chapter will guide you through each step with practical examples and insights. By the end of this chapter, you'll not only be adept at manipulating your data's structure in R but also appreciate how such transformations can unveil new perspectives and insights in your data analysis journey.

12.1 Making data longer

Let's revisit our spectroscopy data from the previous chapter:

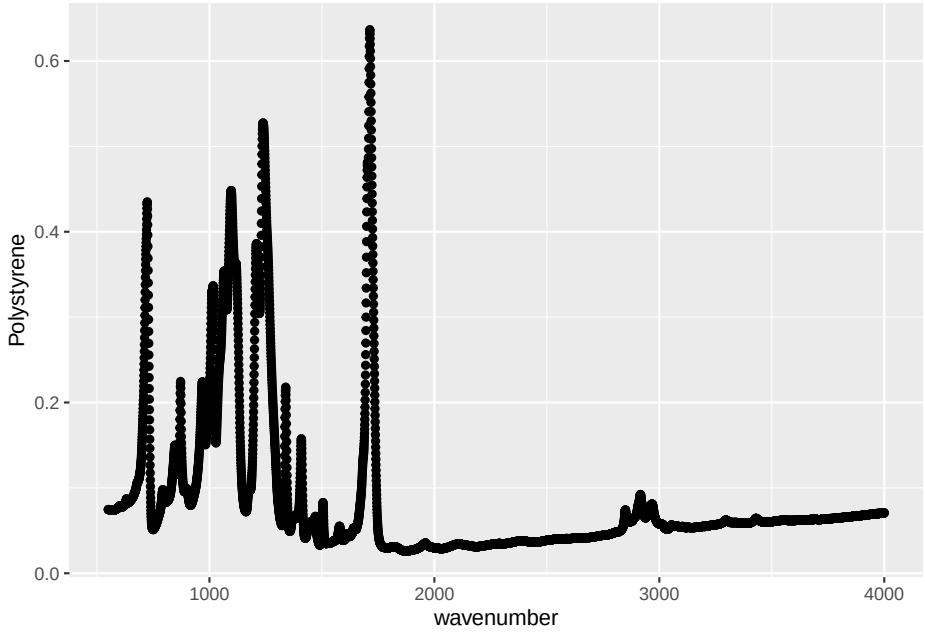
```
atr_plastics <- read_csv("data/ATR_plastics.csv")

# This just outputs a table you can explore within your browser
DT::datatable(atr_plastics)
```

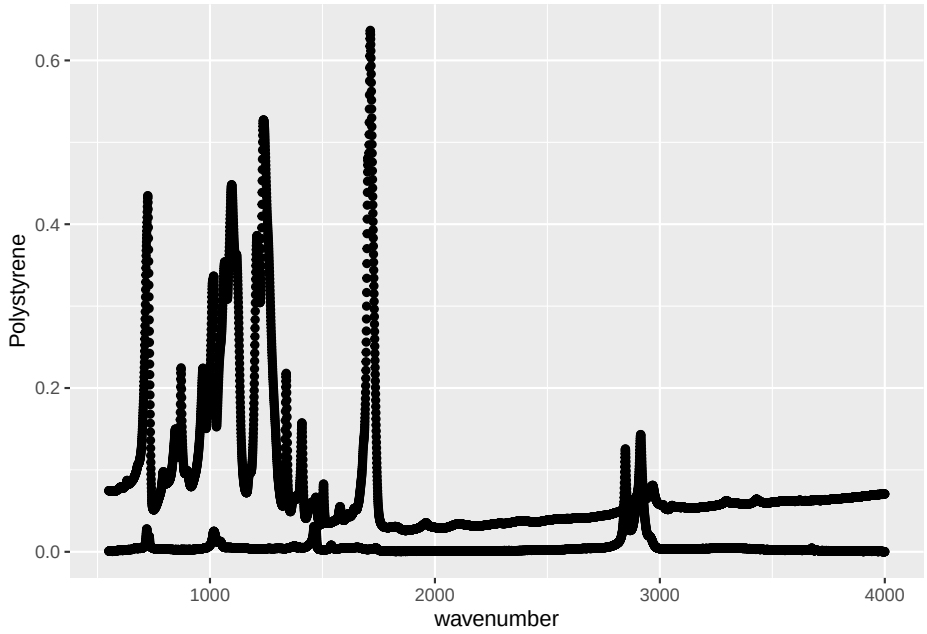
As we can see this our ATR spectroscopy results of several plastics, as recorded for a *CHM 317* lab, is structured similarly to the example in Figure **??**A. The ATR absorbance spectra of the four plastics are recorded in separate columns.

Again, this format makes intuitive sense when recording in the lab, and for working in Excel, but isn't the friendliest with R. When making plots with `ggplot`, we can only specify one `y` variable. In the example plot below it's the absorbance spectrum of `Polystyrene`. However, if wanted to plot the other spectra for comparison, we'd need to repeat our `geom_point` call.

```
# Plotting Polystyrene absorbance spectra
ggplot(data = atr_plastics,
       aes( x = wavenumber,
            y = Polystyrene)) +
  geom_point()
```




```
# Plotting Polystyrene and Polyethylene absorbance spectra
ggplot(data = atr_plastics,
       aes( x = wavenumber,
            y = Polystyrene)) +
  geom_point() +
  geom_point(data = atr_plastics,
             aes(x = wavenumber,
                 y = Polyethylene))
```



While the code above works, it's not particularly handy and undermines much of the utility of `ggplot` because the data *isn't* tidy. Fortunately the `pivot_longer` function can easily restructure our data into the *long* format to work with `ggplot`. Let's demonstrate that:

```
atr_long <- pivot_longer(atr_plastics, cols = -wavenumber,
                names_to = "sample",
                values_to = "absorbance")

# head() only prints the first couple of lines
head(atr_long)
```

```
## # A tibble: 6 x 3
##   wavenumber sample               absorbance
##        <dbl> <chr>                     <dbl>
## 1       550. EPDM                    0.212
## 2       550. Polystyrene             0.0746
## 3       550. Polyethylene            0.000873
## 4       550. Sample: Shopping bag    0.0236
## 5       551. EPDM                    0.212
## 6       551. Polystyrene             0.0746
```

Let's break down the code we've executed via the `pivot_longer` function:

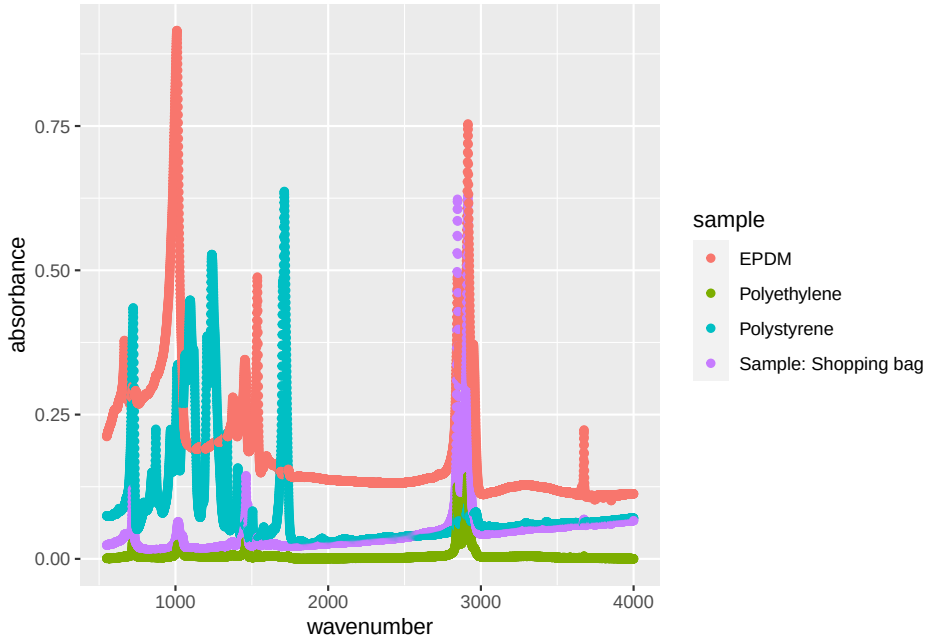
1. `cols = -wavenumber` specifies that we're selecting every other column *but* wave number.
    - we could have just as easily specified each column individually using

`cols = c("EPDM",...)` but it's easier to use `-` to specify what we *don't* want to select.

2. `names_to = "sample"` specifies that the column header (i.e. names) be converted into an observation under the `sample` column.
3. `values_to = "absorbance"` specifies that the absorbance values under each of the selected headers be placed into the `aborsbance` column.

Now that we've reclassified out data into the 'longer', we can exploit the explicitly introduced *sample* variable to easily plot all of our spectra:

```
ggplot(data = atr_long,
       aes(x = wavenumber,
           y = absorbance,
           colour = sample)
       ) +
  geom_point()
```



We'll talk more about `ggplot` in the [Visualizations] chapter, but for now you can see how our code could scale to accommodate any number of different samples, whereas the previous attempt to plot the "wide" data would require an explicit call to each column.

`pivot_longer` has many other features that you can take advantage of. We highly recommend reading the examples listed on the pivot_longer page to get a better sense of the possibilities. For example it's common to record multiple observations in a single column header, i.e. `Chemical_A_0_mM`. We can exploit common naming conventions like this to easily split up these observations as

shown below.

```
head(example)
```

```
##   wavelength_nm Chemical_A_0_mM Chemical_A_1_mM Chemical_B_0_mM Chemical_B_1_mM
## 1           488               0               1               2              NA
## 2           572               0               5               7              20
```

```
example_long <- pivot_longer(example,
   cols = starts_with("Chemical"),
   names_prefix = "Chemical_",
   names_to = c("Chemical", "Concentration", "Conc_Units"),
   names_sep = "_",
   values_to = "Absorbance",
   values_drop_na = TRUE
 )

head(example_long)
```

```
## # A tibble: 6 x 5
##   wavelength_nm Chemical Concentration Conc_Units Absorbance
##           <dbl> <chr>    <chr>         <chr>           <dbl>
## 1           488 A        0             mM                  0
## 2           488 A        1             mM                  1
## 3           488 B        0             mM                  2
## 4           572 A        0             mM                  0
## 5           572 A        1             mM                  5
## 6           572 B        0             mM                  7
```

12.2 Making Data Wider

In data analysis, specific requirements or packages, such as `matrixStats` and `matrixTests`, often necessitate reshaping your data into a matrix or 'wide' format. The `pivot_wider` function from the `tidyr` package is a robust tool for this transformation, operating as the inverse of the `pivot_longer` function we discussed earlier. Essentially, `pivot_wider` is used to spread key-value pairs across a dataset, transforming it from a long to a wide format. This is especially useful when you need your data structured in a wide matrix for certain analytical procedures or visual presentations. To learn more about the `pivot_wider` function, I recommend reading the documentation here.

For example, consider a dataset where we need to compare absorbance data across different samples. With `pivot_wider`, we can transform this data so that each sample is represented in its own column. This transformation is executed as follows:

```
atr_wide <- pivot_wider(atr_long,
                        names_from = sample,
                        values_from = absorbance)

# Viewing the first few lines of the transformed dataset
head(atr_wide)
```

```
## # A tibble: 6 x 5
##   wavenumber  EPDM Polystyrene Polyethylene `Sample: Shopping bag`
##        <dbl> <dbl>       <dbl>        <dbl>                  <dbl>
## 1       550. 0.212      0.0746     0.000873                 0.0236
## 2       551. 0.212      0.0746     0.000834                 0.0238
## 3       551. 0.213      0.0745     0.000819                 0.0239
## 4       552. 0.213      0.0745     0.000825                 0.0239
## 5       552. 0.214      0.0745     0.000868                 0.0240
## 6       553. 0.214      0.0746     0.000949                 0.0240
```

Here's a breakdown of this code:

1. `names_from = sample`: This argument specifies which column in our long data will be used to create new column headers in the wide format. Each unique value in the `sample` column becomes a separate column in the resulting wide dataset.
2. `values_from = absorbance`: This tells R that the values filling these new sample columns should be taken from the `absorbance` column.

The result is a more traditional, wide-format dataset where each column represents a different sample's absorbance values, facilitating side-by-side comparisons.

12.2.1 Practical Uses of `pivot_wider`

The `pivot_wider` function is not only useful for converting long data to wide but also for data summarization and creating formats suitable for reports or specific analyses. If you're dealing with summarized data, such as averages or counts, spreading this data into a wide format can make it more interpretable and easier to analyze.

Furthermore, `pivot_wider` can be an essential part of a more complex data transformation process. In many cases, data manipulation might require alternating between widening and lengthening to achieve the desired structure for your analysis.

Understanding both `pivot_longer` and `pivot_wider` equips you with a versatile toolkit for shaping your data. Whether you're preparing data for specific package requirements, like `matrixStats` or `matrixTests`, or simply need to restructure your dataset for clarity and analysis, these functions are invaluable in the R programming environment.

12.3 Further reading

As always, the *R for Data Science* book goes into more detail on all of the elements discussed above. For the material covered here you may want to read Chapter 9: Tidy Data.

12.4 Exercise

There is a set of exercises available for this chapter!

Not sure how to access and work on the exercise Rmd files?

- Refer to Chapter 3.3 for step-by-step instructions on accessing the exercises and working within the UofT JupyterHub's RStudio environment.
- Alternatively, if you'd like to simply access the individual files, you can download them directly from this repository.

Always remember to save your progress regularly and consult the textbook's guidelines for submitting your completed exercises.

Chapter 13

Transform: Data Manipulation

Transformation encompasses any steps you take to manipulate, reshape, refine, or transform your data. We've already touched upon some useful transformation functions in previous example code snippets, such as the `mutate` function for adding columns. This section will explore some of the most useful functionalities of the `dplyr` package, explicitly introduce the pipe operator `%>%`, and showcase how you can leverage these tools to quickly manipulate your data.

The essential `dplyr` functions are :

- `mutate()` to create new columns/variables from existing data
- `arrange()` to reorder rows
- `filter()` to refine observations by their values (in other words by row)
- `select()` to pick variables by name (in other words by column)
- `summarize()` to collapse many values down to a single summary.

We'll go through each of these functions, but we highly recommend you read Chapter 3: Data Transformation from *R for Data Science* to get a more comprehensive breakdown of these functions. Note that the information here is based on a `tidyverse` approach, but this is only one way of doing things. Please see the Further reading section for links to other suitable approaches to data transformation.

Let's explore the functionality of `dplyr` using some flame absorption/emission spectroscopy (FAES) data from a *CHM317* lab. This data represents the emission signal of five sodium (Na) standards measured in triplicate:

```
# Importing using tips from Import chapter
FAES <- read_csv(file = "data/FAESdata.csv") %>% # see section on Pipe
  pivot_longer(cols = -std_Na_conc,
```

```
                names_to = "replicate",
                names_prefix = "reading_",
                values_to = "signal") %>%
  separate(col = std_Na_conc,
           into = c("type", "conc_Na", "units"),
           sep = " ",
           convert = TRUE)

DT::datatable(FAES)
```

Note the use of `convert = TRUE` in the `separate()` call. This runs a type convert on new columns. If we didn't include this, the `conc_Na` column would be of type character because the numbers originated from a string. `convert()` ensures they're converted to numeric. **Always use `convert = TRUE`** when you separate columns.

13.1 Selecting by row or value

`filter()` allows up to subset our data based on observation (row) values.

```
filter(FAES, conc_Na == 0)
```

```
## # A tibble: 3 x 5
##   type  conc_Na units replicate signal
##   <chr>   <dbl> <chr> <chr>      <dbl>
## 1 blank       0 ppm   1           502.
## 2 blank       0 ppm   2           592.
## 3 blank       0 ppm   3           581.
```

Note how we need to pass logical operations to `filter()` to specify which rows we want to select. In the above code, we used `filter()` to get all rows where the concentration of sodium is equal to 0 (`== 0`). Note the presence of two equal signs (`==`). In R one equal sign (`=`) is used to pass an argument, two equal signs (`==`) is the logical operation "is equal" and is used to test equality (i.e. that both sides have the same value). A frequent mistake is to use `=` instead of `==` when testing for equality.

13.1.1 Logical operators

`filter()` can use other *relational* and *logical* operators or combinations thereof. Relational operators compare values and logical operators carry out Boolean operations (TRUE or FALSE). Logical operators are used to combine multiple relational operators… let's just list what they are and how we can use them:

Operator	Type	Description
>	relational	Less than
<	relational	Greater than
<=	relational	Less than or equal to
>=	relational	Greater than or equal to
==	relational	Equal to
!=	relational	Not equal to
&	logical	AND
!	logical	NOT
|	logical	OR
is.na()	function	Checks for missing values, TRUE if NA

- Selecting all signals below a threshold value

```
filter(FAES, signal < 4450)
```

```
## # A tibble: 3 x 5
##   type  conc_Na units replicate signal
##   <chr>   <dbl> <chr> <chr>      <dbl>
## 1 blank       0 ppm   1           502.
## 2 blank       0 ppm   2           592.
## 3 blank       0 ppm   3           581.
```

- Selecting signals between values

```
filter(FAES, signal >= 4450 & signal < 8150)
```

```
## # A tibble: 3 x 5
##   type     conc_Na units replicate signal
##   <chr>      <dbl> <chr> <chr>      <dbl>
## 1 standard     0.1 ppm   1          5656.
## 2 standard     0.1 ppm   2          5654.
## 3 standard     0.1 ppm   3          5667.
```

- Selecting all other replicates other than replicate 2

```
filter(FAES, replicate != 2)
```

```
## # A tibble: 10 x 5
##    type     conc_Na units replicate signal
##    <chr>      <dbl> <chr> <chr>      <dbl>
##  1 blank        0   ppm   1           502.
##  2 blank        0   ppm   3           581.
##  3 standard     0.1 ppm   1          5656.
##  4 standard     0.1 ppm   3          5667.
##  5 standard     0.2 ppm   1          9393.
##  6 standard     0.2 ppm   3          9332.
##  7 standard     0.5 ppm   1         20187.
##  8 standard     0.5 ppm   3         20153.
```

```
##  9 standard     1   ppm   1         30798.
## 10 standard     1   ppm   3         30790.
```

- selecting the first standard replicate OR any of the blanks.

```
filter(FAES, (type == "standard" & replicate == 1) | (type == "blank"))
```

```
## # A tibble: 7 x 5
##   type     conc_Na units replicate signal
##   <chr>      <dbl> <chr> <chr>      <dbl>
## 1 blank        0   ppm   1           502.
## 2 blank        0   ppm   2           592.
## 3 blank        0   ppm   3           581.
## 4 standard     0.1 ppm   1          5656.
## 5 standard     0.2 ppm   1          9393.
## 6 standard     0.5 ppm   1         20187.
## 7 standard     1   ppm   1         30798.
```

- Removing any missing values (`NA`) using `is.na()`. Note there are no missing values in our data set so nothing will be removed, if we removed the NOT operator (`!`) we would have selected all rows *with* missing values.

```
filter(FAES, !is.na(signal))
```

```
## # A tibble: 15 x 5
##    type     conc_Na units replicate signal
##    <chr>      <dbl> <chr> <chr>      <dbl>
##  1 blank        0   ppm   1           502.
##  2 blank        0   ppm   2           592.
##  3 blank        0   ppm   3           581.
##  4 standard     0.1 ppm   1          5656.
##  5 standard     0.1 ppm   2          5654.
##  6 standard     0.1 ppm   3          5667.
##  7 standard     0.2 ppm   1          9393.
##  8 standard     0.2 ppm   2          9363.
##  9 standard     0.2 ppm   3          9332.
## 10 standard     0.5 ppm   1         20187.
## 11 standard     0.5 ppm   2         20141.
## 12 standard     0.5 ppm   3         20153.
## 13 standard     1   ppm   1         30798.
## 14 standard     1   ppm   2         30837.
## 15 standard     1   ppm   3         30790.
```

These are just some examples, but you can combine the logical operators in any way that works for you. Likewise, there are multiple combinations that will yield the same result, it's up to you do figure out which works best for you.

13.2 Arranging rows

`arrange()` reorders the rows based on the value you passed to it. By default it arranges the specified values into ascending order. Let's arrange our data our data by increasing order of signal value:

```
arrange( FAES, signal)
```

```
## # A tibble: 15 x 5
##    type     conc_Na units replicate signal
##    <chr>      <dbl> <chr> <chr>      <dbl>
##  1 blank        0   ppm   1           502.
##  2 blank        0   ppm   3           581.
##  3 blank        0   ppm   2           592.
##  4 standard     0.1 ppm   2          5654.
##  5 standard     0.1 ppm   1          5656.
##  6 standard     0.1 ppm   3          5667.
##  7 standard     0.2 ppm   3          9332.
##  8 standard     0.2 ppm   2          9363.
##  9 standard     0.2 ppm   1          9393.
## 10 standard     0.5 ppm   2         20141.
## 11 standard     0.5 ppm   3         20153.
## 12 standard     0.5 ppm   1         20187.
## 13 standard     1   ppm   3         30790.
## 14 standard     1   ppm   1         30798.
## 15 standard     1   ppm   2         30837.
```

Since our original `FAES` data is already arranged by increasing `conc_Na` and `replicate`, let's inverse that order by arranging `conc_Na` into descending order using the `desc()` function WHILE arranging the `signal` values in ascending order:

```
# Note the order of precedence (left-to-right)
arrange(FAES, desc(conc_Na), signal)
```

```
## # A tibble: 15 x 5
##    type     conc_Na units replicate signal
##    <chr>      <dbl> <chr> <chr>      <dbl>
##  1 standard     1   ppm   3         30790.
##  2 standard     1   ppm   1         30798.
##  3 standard     1   ppm   2         30837.
##  4 standard     0.5 ppm   2         20141.
##  5 standard     0.5 ppm   3         20153.
##  6 standard     0.5 ppm   1         20187.
##  7 standard     0.2 ppm   3          9332.
##  8 standard     0.2 ppm   2          9363.
##  9 standard     0.2 ppm   1          9393.
## 10 standard     0.1 ppm   2          5654.
```

```
## 11 standard     0.1 ppm   1          5656.
## 12 standard     0.1 ppm   3          5667.
## 13 blank        0   ppm   1           502.
## 14 blank        0   ppm   3           581.
## 15 blank        0   ppm   2           592.
```

Just note with `arrange()` that `NA` values will always be placed at the bottom, whether you use `desc()` or not.

13.3 Selecting column name

`select()` allows you to readily select columns by name. Note however that it will always return a tibble, even if you only select one variable/column.

```
select(FAES, signal)
```

```
## # A tibble: 15 x 1
##    signal
##     <dbl>
##  1   502.
##  2   592.
##  3   581.
##  4  5656.
##  5  5654.
##  6  5667.
##  7  9393.
##  8  9363.
##  9  9332.
## 10 20187.
## 11 20141.
## 12 20153.
## 13 30798.
## 14 30837.
## 15 30790.
```

You can also select multiple columns using the same operators and helper functions described in Tidying Your Data:.

```
select(FAES, conc_Na:replicate)
```

```
## # A tibble: 15 x 3
##    conc_Na units replicate
##      <dbl> <chr> <chr>
##  1     0   ppm   1
##  2     0   ppm   2
##  3     0   ppm   3
##  4     0.1 ppm   1
```

```
##  5     0.1 ppm   2
##  6     0.1 ppm   3
##  7     0.2 ppm   1
##  8     0.2 ppm   2
##  9     0.2 ppm   3
## 10     0.5 ppm   1
## 11     0.5 ppm   2
## 12     0.5 ppm   3
## 13     1   ppm   1
## 14     1   ppm   2
## 15     1   ppm   3
```

```
# Getting columns containing the character "p"
select(FAES, contains("p"))
```

```
## # A tibble: 15 x 2
##    type     replicate
##    <chr>    <chr>
##  1 blank    1
##  2 blank    2
##  3 blank    3
##  4 standard 1
##  5 standard 2
##  6 standard 3
##  7 standard 1
##  8 standard 2
##  9 standard 3
## 10 standard 1
## 11 standard 2
## 12 standard 3
## 13 standard 1
## 14 standard 2
## 15 standard 3
```

13.4 Deleting Columns or Rows

While the process of selecting and filtering data is pivotal in data analysis, there are instances when you may need to remove specific columns or rows entirely. This is useful especially when you're dealing with redundant or irrelevant data that might clutter your analysis.

13.4.1 Deleting columns

To delete a column, you can use the `select()` function with the `-` sign before the column name you want to remove:

```
# This will remove the 'signal' column from the FAES dataset
head(select(FAES, -signal))
```

```
## # A tibble: 6 x 4
##   type     conc_Na units replicate
##   <chr>      <dbl> <chr> <chr>
## 1 blank        0   ppm   1
## 2 blank        0   ppm   2
## 3 blank        0   ppm   3
## 4 standard     0.1 ppm   1
## 5 standard     0.1 ppm   2
## 6 standard     0.1 ppm   3
```

Multiple columns can be deleted by providing more column names after the - sign:

```
# Deleting both 'signal' and 'replicate' columns from the FAES dataset
head(select(FAES, -c(signal, replicate)))
```

```
## # A tibble: 6 x 3
##   type     conc_Na units
##   <chr>      <dbl> <chr>
## 1 blank        0   ppm
## 2 blank        0   ppm
## 3 blank        0   ppm
## 4 standard     0.1 ppm
## 5 standard     0.1 ppm
## 6 standard     0.1 ppm
```

13.4.2 Deleting rows

To delete rows, the `filter()` function can be used in conjunction with relational or logical conditions that define the rows you wish to exclude:

```
# This will remove rows where 'signal' values are less than 20000
filter(FAES, !(signal < 20000))
```

```
## # A tibble: 6 x 5
##   type     conc_Na units replicate signal
##   <chr>      <dbl> <chr> <chr>      <dbl>
## 1 standard     0.5 ppm   1         20187.
## 2 standard     0.5 ppm   2         20141.
## 3 standard     0.5 ppm   3         20153.
## 4 standard     1   ppm   1         30798.
## 5 standard     1   ppm   2         30837.
## 6 standard     1   ppm   3         30790.
```

The key here is the use of the `!` (NOT) operator which excludes rows that meet

the specified condition.

13.5 Adding new variables

`mutate()` allows you to add new variable (read columns) to your existing data set. It'll probably be the workhorse function you'll use during your data transformation as you can readily pass other functions and mathematical operators to it to transform your data. let's suppose that our standards were diluted by a factor of 10, we can add a new column for this:

```
mutate(FAES, "dil_fct" = 10)
```

```
## # A tibble: 15 x 6
##    type     conc_Na units replicate signal dil_fct
##    <chr>      <dbl> <chr> <chr>      <dbl>   <dbl>
##  1 blank        0   ppm   1           502.      10
##  2 blank        0   ppm   2           592.      10
##  3 blank        0   ppm   3           581.      10
##  4 standard     0.1 ppm   1          5656.      10
##  5 standard     0.1 ppm   2          5654.      10
##  6 standard     0.1 ppm   3          5667.      10
##  7 standard     0.2 ppm   1          9393.      10
##  8 standard     0.2 ppm   2          9363.      10
##  9 standard     0.2 ppm   3          9332.      10
## 10 standard     0.5 ppm   1         20187.      10
## 11 standard     0.5 ppm   2         20141.      10
## 12 standard     0.5 ppm   3         20153.      10
## 13 standard     1   ppm   1         30798.      10
## 14 standard     1   ppm   2         30837.      10
## 15 standard     1   ppm   3         30790.      10
```

We can also create multiple columns in the same `mutate()` call:

```
mutate(FAES,
       dil_fct = 10,
       adj_signal = signal * dil_fct)
```

```
## # A tibble: 15 x 7
##    type     conc_Na units replicate signal dil_fct adj_signal
##    <chr>      <dbl> <chr> <chr>      <dbl>   <dbl>      <dbl>
##  1 blank        0   ppm   1           502.      10      5023.
##  2 blank        0   ppm   2           592.      10      5918.
##  3 blank        0   ppm   3           581.      10      5815.
##  4 standard     0.1 ppm   1          5656.      10     56563.
##  5 standard     0.1 ppm   2          5654.      10     56536.
##  6 standard     0.1 ppm   3          5667.      10     56674.
##  7 standard     0.2 ppm   1          9393.      10     93934.
```

```
##  8 standard     0.2 ppm   2          9363.      10     93627.
##  9 standard     0.2 ppm   3          9332.      10     93320.
## 10 standard     0.5 ppm   1         20187.      10    201869.
## 11 standard     0.5 ppm   2         20141.      10    201405.
## 12 standard     0.5 ppm   3         20153.      10    201530.
## 13 standard     1   ppm   1         30798.      10    307977.
## 14 standard     1   ppm   2         30837.      10    308365.
## 15 standard     1   ppm   3         30790.      10    307898.
```

Couple of things to note:

1. The variable we're creating needs to be in quotation marks, hence `dil_fct` for our dilution factor variable
2. The variables we're referencing do not need to be in quotation marks; hence `signal` because this variable already exist.
3. Note the order of precedence: `dil_fct` is created first so we can reference in the second argument, we would get an error if we swapped the order.

13.5.1 Mutate with a condition

In data analysis, there are often scenarios where we want to categorize or relabel values based on certain conditions. The `case_when()` function, provided by the `dplyr` package, offers a versatile and readable solution for handling these multiple conditions.

The syntax for `case_when()` is straightforward: for each condition, you specify the logical test followed by the tilde (~) operator, and then the value or expression to return if the condition is `TRUE`.

With our FAES data, say you want to label each `conc_Na` as "Low", "Medium", or "High" based on its value. You can use `case_when()` within `mutate()` as follows:

```
mutate(FAES,
       conc_Na_level = case_when(
         conc_Na < 0.2 ~ "Low",
         conc_Na < 0.4 ~ "Medium",
         TRUE ~ "High"))
```

```
## # A tibble: 15 x 6
##    type     conc_Na units replicate signal conc_Na_level
##    <chr>      <dbl> <chr> <chr>      <dbl> <chr>
##  1 blank        0   ppm   1           502. Low
##  2 blank        0   ppm   2           592. Low
##  3 blank        0   ppm   3           581. Low
##  4 standard     0.1 ppm   1          5656. Low
##  5 standard     0.1 ppm   2          5654. Low
##  6 standard     0.1 ppm   3          5667. Low
```

```
##  7 standard     0.2 ppm   1          9393. Medium
##  8 standard     0.2 ppm   2          9363. Medium
##  9 standard     0.2 ppm   3          9332. Medium
## 10 standard     0.5 ppm   1         20187. High
## 11 standard     0.5 ppm   2         20141. High
## 12 standard     0.5 ppm   3         20153. High
## 13 standard     1   ppm   1         30798. High
## 14 standard     1   ppm   2         30837. High
## 15 standard     1   ppm   3         30790. High
```

For those interested in exploring further, there's a similar function called `ifelse()` which provides conditional transformations in R. You can learn more about it here.

13.5.2 Useful mutate function

There are a myriad of functions you can make use of with the mutate function. Here are some of the mathematical operators available in R:

Operator.Function	Definition
+	additon
-	subtraction
*	multiplication
/	division
^	exponent; to the power of...
log()	returns the specified base-log; see also log10() and log2()

13.6 Group and summarize data

`summarize` effectively summarized your data based on functions you've passed to it. Looking at our `FAES` data we'd probably want the mean of the triplicate signals, alongside the standard deviation. Let's see what happens when we apply the summarize function straight up:

```
summarise(FAES, "mean" = mean(signal), "stdDev" = sd(signal))
```

```
## # A tibble: 1 x 2
##     mean stdDev
##    <dbl>  <dbl>
## 1 13310. 11242.
```

This doesn't look like what we wanted. What we got was the mean and standard deviation of *all* of the signals, regardless of the concentration of the standard. Also note how we've lost the other columns/variables and are only left with the mean and stdDev. This is all because we need to **group** our observations by a variable. We can do this by using the `group_by()` function.

```
groupedFAES <- group_by(FAES, type, conc_Na)
summarise(groupedFAES, "mean" = mean(signal), "stdDev" = sd(signal))
```

```
## `summarise()` has grouped output by 'type'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 5 x 4
## # Groups:   type [2]
##   type     conc_Na   mean stdDev
##   <chr>      <dbl>  <dbl>  <dbl>
## 1 blank        0     559.  48.9
## 2 standard     0.1  5659.   7.34
## 3 standard     0.2  9363.  30.7
## 4 standard     0.5 20160.  24.0
## 5 standard     1   30808.  25.0
```

Here we've created a new data set, `groupedFAES`, that we grouped by the variables `type` and `conc_Na` so we could get the mean and standard deviation of each group. Note the multiple levels of grouping. For this data set we could have omitted the `type` variable, but in larger datasets you may have multiple groupings (i.e. from different location), so you can group by multiple variables to get smaller groups.

13.6.1 Useful summarize functions

We've used the `mean()` and `sd()` functions above, but there are a host of other useful functions you can use in conjunction with summarize. See **Useful Functions** in the `summarise()` documentation (enter `?summarise`) in the console. This is also discussed at in more depth in the Summarizing Data chapter.

13.7 The pipe: chaining functions together

With the tools presented here we could do a decent job analyzing our `FAES` data. Let's say we wanted to subtract the mean of the `blank` from each `standard` signal and then get summarize those results. It would look something like this:

```
blank <- filter(FAES, type == "blank")
meanBlank <- summarize(blank, mean(signal))
meanBlank <- as.numeric(meanBlank)

paste("The mean signal from the blank triplicate is:", meanBlank)
```

```
## [1] "The mean signal from the blank triplicate is: 558.5249"
```

```
stds_1 <- filter(FAES, type == "standard")
stds_2 <- mutate(stds_1, "cor_sig" = signal - meanBlank)
stds_3 <- group_by(stds_2, conc_Na)
```

```
stds_4 <- summarize(stds_3, "mean" = mean(cor_sig), "stdDev" = sd(cor_sig))
stds_4
```

```
## # A tibble: 4 x 3
##   conc_Na   mean stdDev
##     <dbl>  <dbl>  <dbl>
## 1     0.1  5101.   7.34
## 2     0.2  8804.  30.7
## 3     0.5 19602.  24.0
## 4     1   30249.  25.0
```

While the code above did it's job, it's certainly wasn't easy to type and certainly not easy to read. At every step of the way we've saved our updated data outputs to a new variable (`stds_1`, `stds_2`, etc.). However, most of these intermediates aren't important, and moreover the repetitive names clutter our code. As the code above is written, we've had to pay special attending to the variable suffix to make sure we're calling the correct data set as our code has progresses. An alternative would be to reassign the outputs back to the original variable name (i.e. `stds_1 <- mutate(stds_1, ...)`), but that doesn't solve the issue of readability as there's still redundant assigning.

A solution for this is the pipe operator `%>%` (pronounced as "then"), an incredibly useful tool for writing more legible and understandable code. The pipe basically changes how you read code to emphasize the functions you're working with by passing the intermediate steps to hidden processes in the background. Re-writing the code above, we'd get something like:

```
meanBlank <- FAES %>%
  filter(type =="blank") %>%
  summarise(mean(signal)) %>%
  as.numeric()

paste("The mean signal from the blank triplicate is:", meanBlank)
```

```
## [1] "The mean signal from the blank triplicate is: 558.5249"
```

Things may look a bit different, but our underlying code hasn't changed much. What's happening is the pipe operator passes the output to the first argument of the next function. So the output of `filter...` is passed to the first argument of `sumamrise...`, and the argument we specified in `summarise` is actually the *second* argument it receives. You're probably wondering how hiding stuff makes your code more legible, but think of `%>%` as being equivalent to "then". We can read our code as:

> "Take the `FAES` dataset, *then* filter for `type == "blank"` *then* collapse the dataset to the mean `signal` value and *then* convert to numeric value *then* pass this final output to the new variable `meanBlank`."

Not only is the pipe less typing, but the emphasis is on the functions so you can better understand what you're doing vs. where all the intermediates are going. Extending our piping to the second batch of code we get:

```
stds <- FAES %>%
  filter(type == "standard") %>%
  mutate("cor_sig" = signal - meanBlank) %>%
  group_by(conc_Na) %>%
  summarize("mean" = mean(cor_sig), "stdDev" = sd(cor_sig))

stds
```

```
## # A tibble: 4 x 3
##   conc_Na   mean stdDev
##     <dbl>  <dbl>  <dbl>
## 1     0.1  5101.   7.34
## 2     0.2  8804.  30.7
## 3     0.5 19602.  24.0
## 4     1   30249.  25.0
```

Same thing. The underlying code hasn't changed much, but it's much more legible and we can clearly see we're subtracting the `meanBlank` value from each measured signal then summarizing the corrected signals.

13.7.1 Notes on piping

The pipe is great and especially useful with *tidyverse* packages, but it does have some limitations:

- You can't easily extract intermediate steps. So you'll need to break up your pipping chain to output any intermediate steps you can.
- The benefit of piping is legibility; this goes away as you increase the number of steps as you lose track of what's going on. Keep the piping short and thematically similar.
- Pipes are linear, if you have multiple inputs or outputs you should consider an alternative approach.

13.8 Further reading

- Chapter 5: Data Transformation of *R for Data Science* for a deeper breakdown of `dplyr` and it's functionality.
- Chapter 18: Pipes of *R for Data Science* for more information on pipes.
- Syntax equivalents: base R vs Tidyverse by Hugo Taveres for a comparison of base-R solutions to tidyverse. This entire book is largely biased towards tidyverse solutions, but there's no denying that certain base-R can be more elegant. Check out this write up to get a better idea.

13.9 Exercise

There is a set of exercises available for this chapter!

Not sure how to access and work on the exercise Rmd files?

- Refer to Chapter 3.3 for step-by-step instructions on accessing the exercises and working within the UofT JupyterHub's RStudio environment.
- Alternatively, if you'd like to simply access the individual files, you can download them directly from this repository.

Always remember to save your progress regularly and consult the textbook's guidelines for submitting your completed exercises.

Chapter 14

ggplot basic visualizations

ggplot2 is loaded by default with the tidyverse suite of packages. Let's revisit our spectroscopy data we encountered in Tidying your data:

```
library(tidyverse)
atr_long <- read_csv("data/ATR_plastics.csv") %>%
    pivot_longer(cols = -wavenumber,
                 names_to = "sample",
                 values_to = "absorbance")
```

```
## Rows: 7157 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (5): wavenumber, EPDM, Polystyrene, Polyethylene, Sample: Shopping bag
##
## 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.
```

```
# First 50 rows of data
DT::datatable(atr_long[1:50, ])
```

14.1 Building plots ups

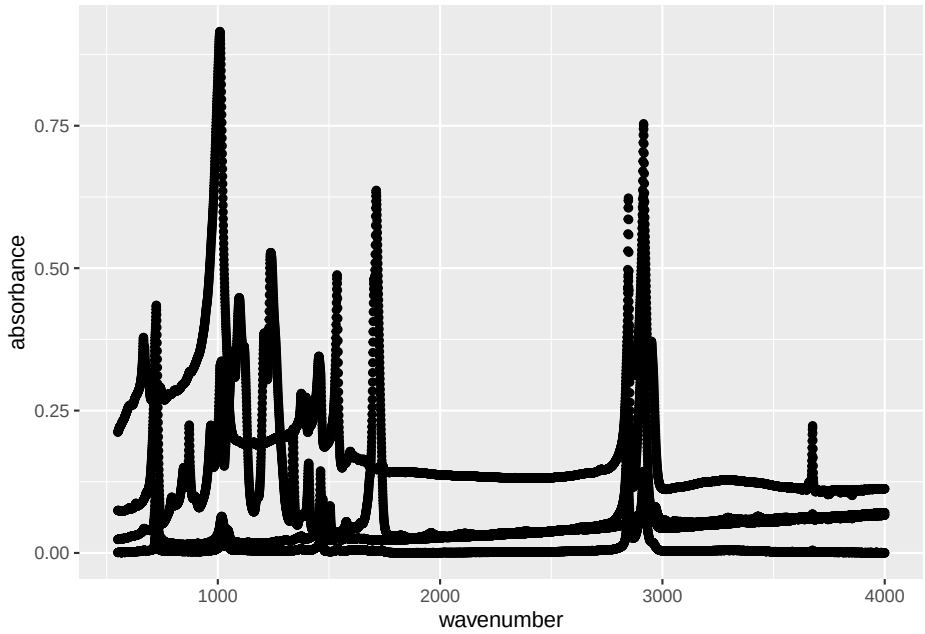
The gg in ggplot2 stands for the *grammar of graphics*,(H. Wickham 2009) and it's a way to break down graphics (plots) into small pieces that can be discussed (hence grammar). We'll take a look at this grammar via geoms (what kind of plot), aes (aesthetic choices), etc. For now, understand that this means we need to build up graphics/plots piece-by-piece and layer-by-layer. This extends beyond code to how we code. No sense in putting lipstick on a pig. Plot often, and discard the useless ones. Take the time to pretty up your plot *after* you're satisfied with the underlying data.

14.2 Basic plotting

`ggplot2` uses `geoms` to specify what type of plot to create. Different plots are used to tell different stories and have different strengths and weakness. We'll explore these more in Visualizations for Env Chem, but for now we'll focus on `geom_point()`, which simply plots data as points on an [x,y] coordinate. In other words, a scatter plot.

Let's plot our tided `atr_long` data:

```
ggplot(data = atr_long,
       aes(x = wavenumber, y = absorbance)) +
  geom_point()
```



Let's ignore the plot for now and look at our code down:

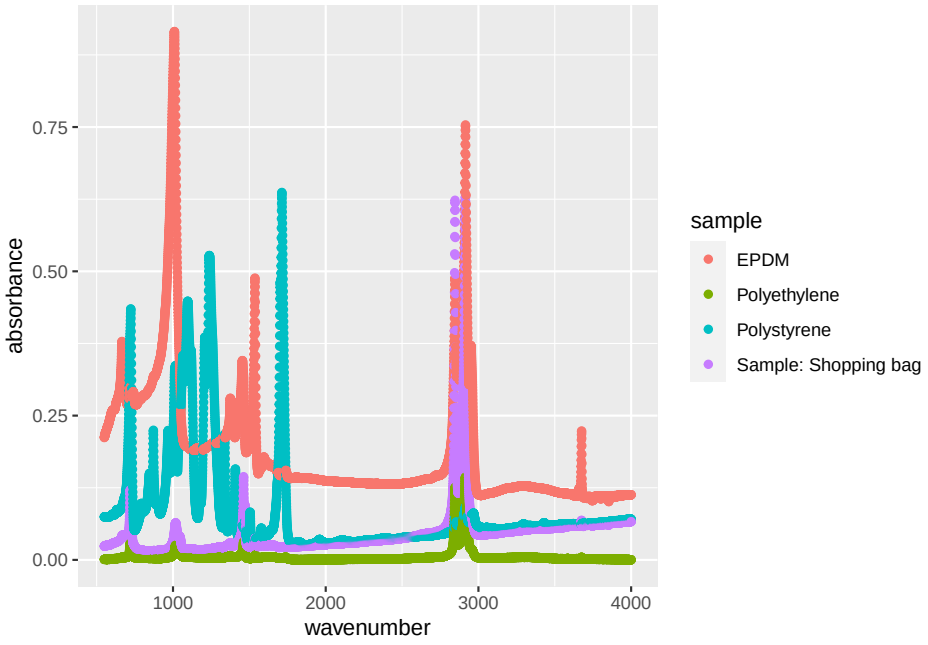
1. `ggplot()` initializes a *ggplot object*, basically an empty plot.
2. We then specified we want to plot data from our `atr_long` dataset (`data = atr_long`).
3. We then specified our *aesthetic mappings* via `aes()`. Here we'll pass information for how we want the plot to look.
4. To our aesthetic mappings we've specified which values from `atr_long` are supposed to be our x-axis values (`x = wavenumber`) and y-axis values (`y = absorbance`).
5. We then add the `geom_point()` layer to create a scatter plot of [x,y] points .

Now let's look at our result. What we see is a point for every recorded ab-

sorbance measurements from our ATR analysis. We can clearly see the spectra of the different plastics in our data, however they're all coloured the same. This is because we've only specified the x and y values. As far as `ggplot()` is concerned, these are the only values that matter, but we know different.

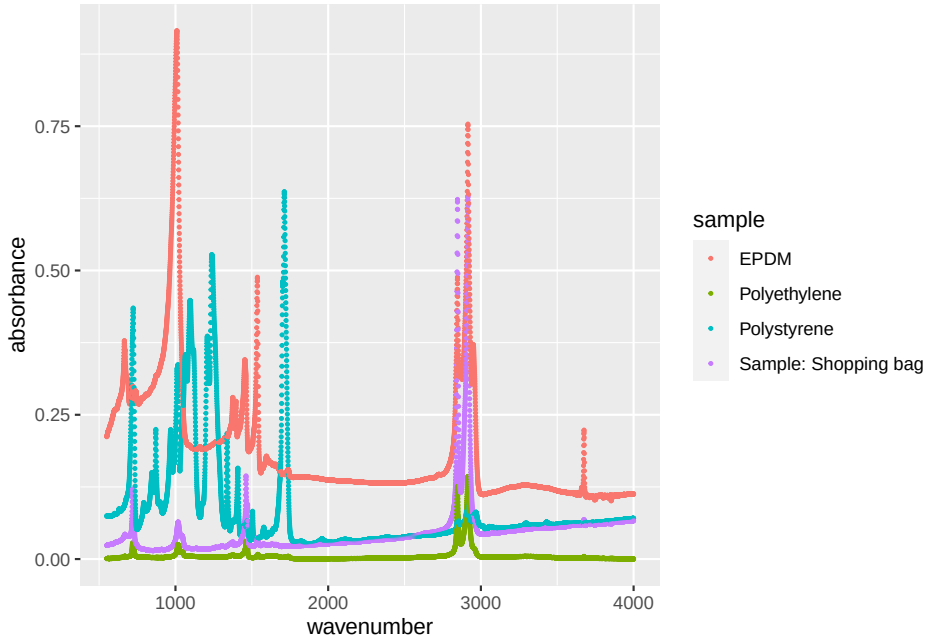
Fortunately you can pass multiple variables to different `aes()` options to enhance our plot. For instance, we can pass the `sample` variable, which specifies which sample a spectrum originates from, to the `colour` option:

```
ggplot(data = atr_long,
       aes(x = wavenumber,
           y = absorbance,
           colour = sample)) +
  geom_point()
```



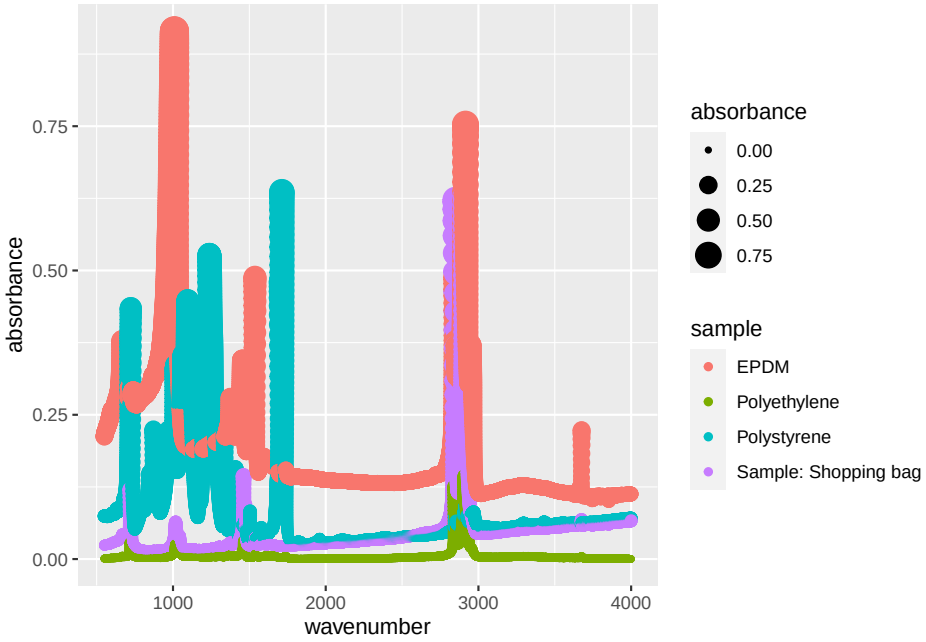
Now we have a legend which clearly specifies which points are associated with each sample. But now the points are too large, potentially masking certain peaks. We can adjust the size of each point as follows:

```
ggplot(data = atr_long,
       aes(x = wavenumber,
           y = absorbance,
           colour = sample)) +
  geom_point(size = 0.5)
```



We specified `size = 0.5` in the `geom_point()` call because it's a constant. We can map `size` to any continuous variable, such as the absorbance:

```
ggplot(data = atr_long,
       aes(x = wavenumber,
           y = absorbance,
           colour = sample,
           size = absorbance)) +
  geom_point()
```

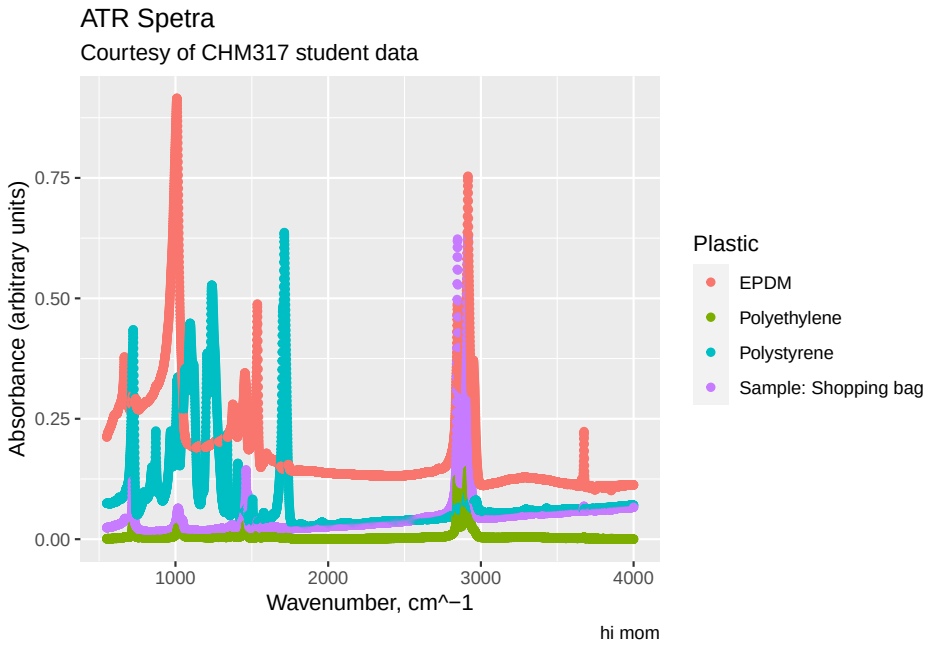


Sometimes this makes sense (i.e. a *bubble chart*) but for our example, having the size of the points increase as the absorbance increases doesn't provide any new information (it actually clutters our plot).

14.3 Changing plot labels

By default `ggplot` uses the header of the columns you passed for the `x` and `y` `aes()` options. Because headers are written for code they're often poor label titles for plots. We can specify new labels and plot titles as follows:

```
ggplot(data = atr_long,
       aes(x = wavenumber,
           y = absorbance,
           colour = sample)) +
  geom_point() +
  labs(title = "ATR Spetra",
       subtitle = "Courtesy of CHM317 student data",
       x = "Wavenumber, cm^-1",
       y = "Absorbance (arbitrary units)",
       caption = "hi mom",
       colour = "Plastic")
```

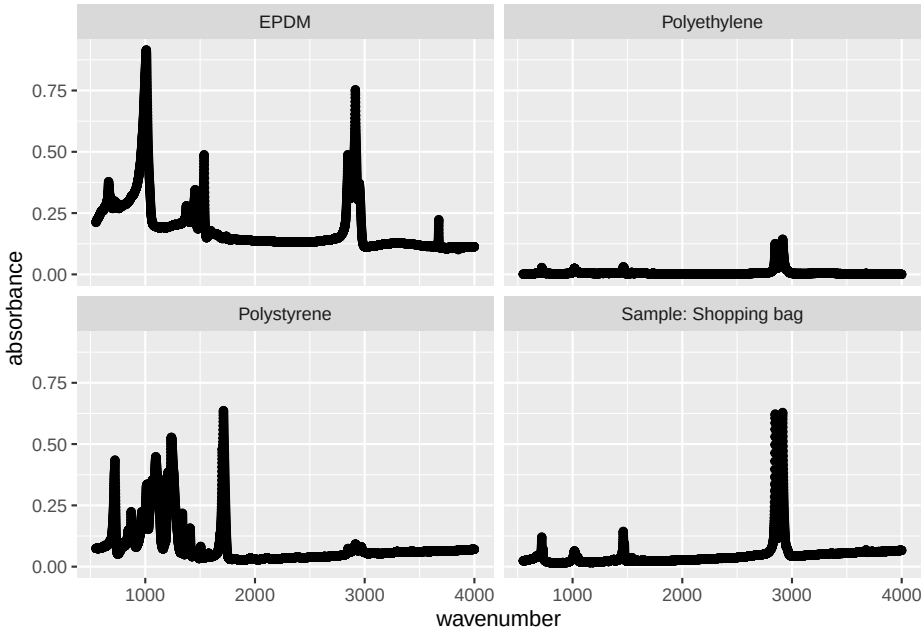


Note how we changed the title of the legend with `colour = "Plastics"`. This is because the legend is generated from our colour aesthetic (`aes(..., colour = sample)`). If our legend was based off of the size aesthetic, we would use `size = "New Title"` to change the title for the size legend.

14.4 Small Multiples

Sometimes your plots become overwhelming, a phenomena called overplotting, which prevent your from comparing graphs or charts. A popular solution is *small multiples*, a series of smilar plots using the same scale and axes. This is readily accomplished in R using `facet_grid()` (which creates a 2-D grid) or `facet_wrap()` (a single 1d ribbon wrapped into 2D space). You simply specy which variable you want to differentiate your plots, for us it's `sample`:

```
ggplot(data = atr_long,
       aes(x = wavenumber,
           y = absorbance)) +
  geom_point() +
  facet_wrap(~sample)
```



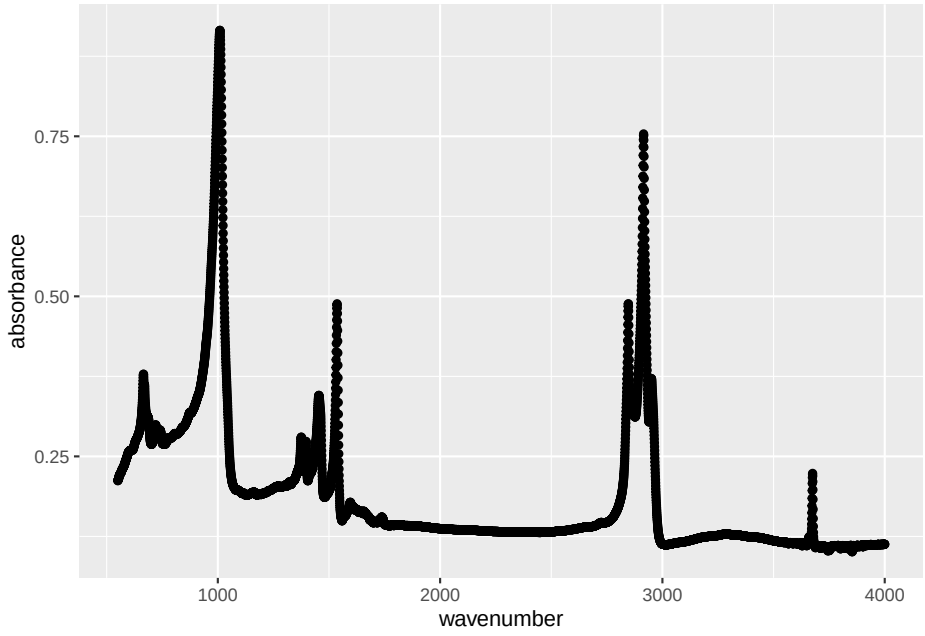
Note the use of the tilde (`~`) in `facet_wrap(~sample)`; in this situation, it's shorthand telling `facet_wrap()` to make small multiples off of the sample variable.

14.5 Plotting subsets of data

Often you won't want to plot everything in your dataset. Rather, you'll want to plot a specific chemical, city, location, etc. To that end you want to plot a *subset* of your data. There are a couple of ways to handle this.

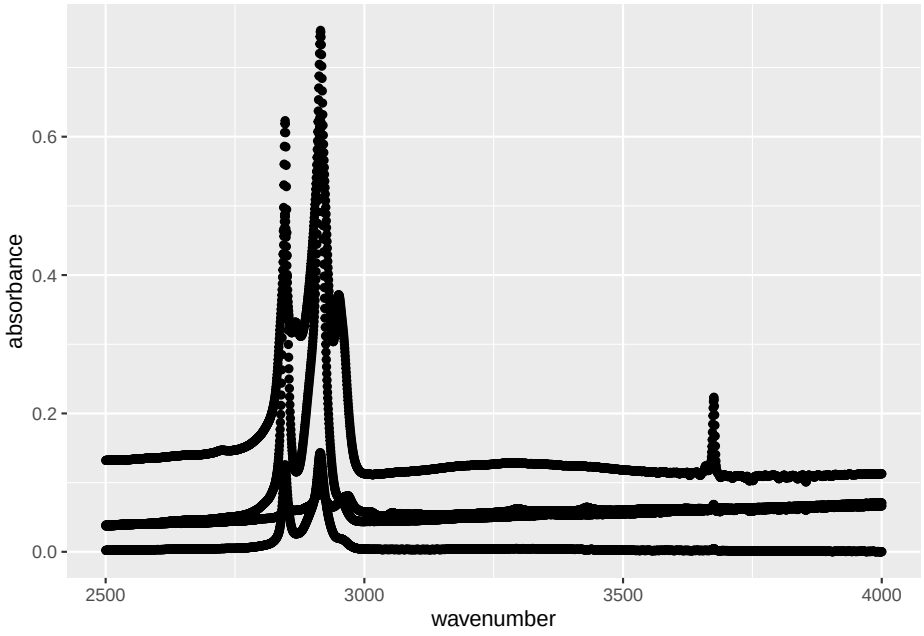
You can subset your data on the fly using `subset()`. This way allows you to specify based off of Logical operators as such:

```
ggplot(data = subset(atr_long, sample == "EPDM"),
       aes(x = wavenumber,
           y = absorbance)) +
  geom_point()
```



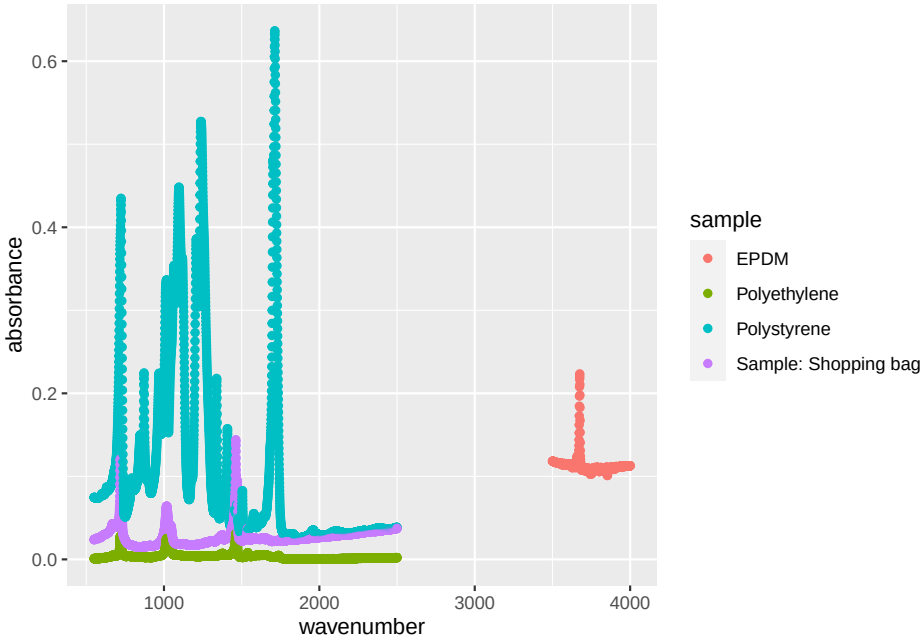
or

```
ggplot(data = subset(atr_long, wavenumber >=2500),
       aes(x = wavenumber,
           y = absorbance)) +
  geom_point()
```



Another approach is to use `filter()` and pipe to `ggplot()`:

```
atr_long %>%
  filter(sample != "EPDM" & wavenumber <= 2500 |sample == "EPDM" & wavenumber >= 3500 ) %>%
  ggplot(., aes(x = wavenumber, y = absorbance, colour = sample)) +
  geom_point()
```



There are pros and cons to either approach. `subset()` on the fly is best for simple task, like plotting a single city, whereas the piping approach is best for more complex sorting.

Chapter 15

Summarizing Data

Summarizing data is what it sounds like. You're reducing the number of rows in your dataset based on some predetermined method. Taking the average of a group of numbers is summarizing the data. Many numbers have been condensed to one: the average. In this chapter we'll go over summarizing data, and some aesthetic changes we can make for publication ready tables.

15.1 Data to play with

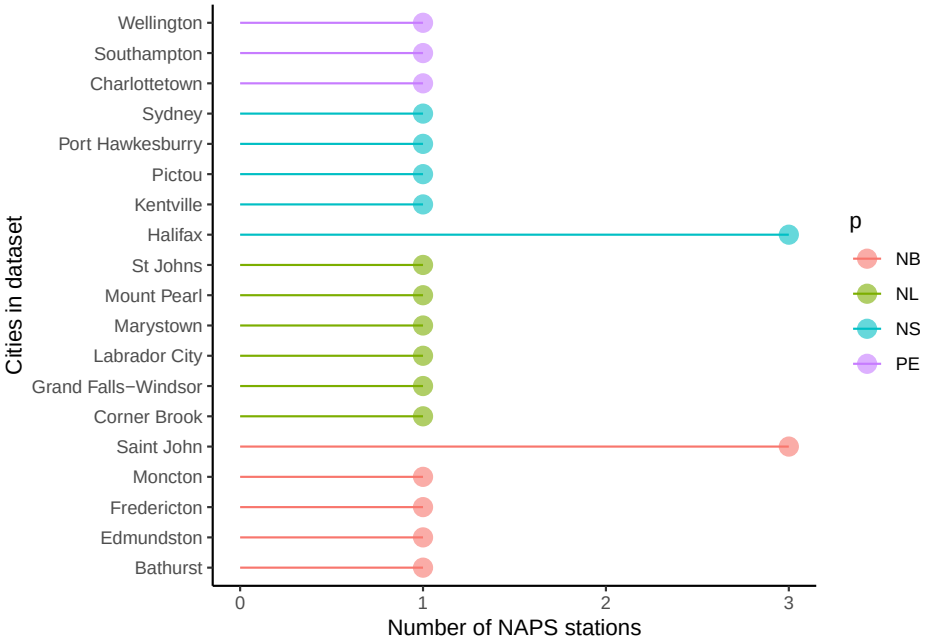
We'll take a look at the 2018 hourly mean NO_2 concentrations for the Atlantic provinces (New Brunswick, Prince Edward Island, Nova Scotia, and Newfoundland). The dataset is available in the *R4EnvChem* Project Template repository. Also if you're keen, you can download any number of atmospheric datasets from *Environment and Climate Change Canada*'s (ECCC) National Airborne Pollution Program's (NAPS) website here

Since ECCC stores their NAPS data in a matrix layout, we need to briefly tidy it up:

```
atlNO2 <- read_csv("data/2018hourlyNO2_Atl.csv", skip = 7, na =c("-999")) %>%
  rename_with(~tolower(gsub("/.*", "", .x))) %>%
  pivot_longer(cols = starts_with("h"),
               names_prefix = "h",
               names_to = "hour",
               names_transform = list(hour = as.numeric),
               values_to = "conc",
               values_transform = list(conc = as.numeric),
               values_drop_na = TRUE)

# First 50 rows of dataset
DT::datatable(atlNO2[1:50, ])
```

Note in our dataset that both Halifax NS and Saint John NB have three NAPS stations each. It won't matter for our aggregation, but if we were exploring this data in more depth this is something we would want to take into account.



15.2 Summarizing data by group

While we can readily summarize an entire dataset, we often want to summarize *groups* within our dataset. In our case, it's [NO_2] in each city. To this end, we need to combine the`group_by()` and `summarize()` functions. `summarise()` also works for the Americans. This approach allows us to (1) specify which groups we want summarized, and (2) how we want them summarized. We'll talk more about point (2) later on, for now, let's look at point (1)

Let's calculate the mean hourly NO_2 concentrations in the 4 provinces in our dataset:

```
sumAtl <- atlNO2 %>%
  group_by(p) %>%
  summarize(mean = mean(conc))

sumAtl
```

```
## # A tibble: 4 x 2
##   p      mean
```

```
##   <chr> <dbl>
## 1 NB    2.86
## 2 NL    2.30
## 3 NS    2.36
## 4 PE    0.975
```

That's it. 186339 unique rows summarized like that. Note that `summarize` produces a *new* data frame, so you'll want to double check on the outputted data types. Let's break down what our code does:

- We're creating a new data frame, so we store it in `sumAtl`.
- We then take our `atlNO2` dataset and group our dataset by province using `group_by(p)`.
- We then summarize our grouped data by summarizing the NO_2 concentration with `summarize(mean = mean(conc)`.
  - Note that since we're creating a new data set, we need to create new columns. This is what `mean = mean(conc)` does. We're creating a column *called mean*, which contains the *numerical mean* 1-hr NO_2 values which were calculated using the `mean()` function. Simple…

Let's dig a little deeper. The *Canadian Ambient Air Quality Standards* stipulates that the annual mean of 1-hour means for NO_2 cannot exceed 17.0 ppb in 2020, and 12.0 ppb in 2025. Let's see if any city in our dataset violated these standards in 2018.

To do this, we'll group by province (`p`) and city (`city`). This will retain our provinces column that we might want to use later on.

```
sumAtl <- atlNO2 %>%
  group_by(p, city) %>%
  summarize(mean = mean(conc))

sumAtl
```

```
## # A tibble: 19 x 3
## # Groups:   p [4]
##    p     city                mean
##    <chr> <chr>              <dbl>
##  1 NB    Bathurst           1.21
##  2 NB    Edmundston         4.52
##  3 NB    Fredericton        1.82
##  4 NB    Moncton            3.37
##  5 NB    Saint John         3.02
##  6 NL    Corner Brook       2.71
##  7 NL    Grand Falls-Windsor 0.918
##  8 NL    Labrador City      2.51
##  9 NL    Marystown          0.277
## 10 NL    Mount Pearl        1.53
```

```
## 11 NL    St Johns            5.33
## 12 NS    Halifax             3.44
## 13 NS    Kentville           0.841
## 14 NS    Pictou              1.19
## 15 NS    Port Hawkesburry    2.53
## 16 NS    Sydney              2.66
## 17 PE    Charlottetown       1.85
## 18 PE    Southampton         0.512
## 19 PE    Wellington          0.455
```

Looks like there aren't any offenders. For tips on visualizing these results please see the [Visualizations] chapter.

15.2.1 Further summarize operations

There are other options we can use to summarize out data. A handy list is provided on the `summarize()` help page. The most common ones you'll need are:

- `mean()` which calculates the arithmetic mean, a.k.a. the average.
- `median()` which calculates the sample median, the value separating the higher 50% of data from the lower 50% of a data sample.
- `sd()` which calculates the sample standard deviation.
- `min()` and `max()` which returns the smallest and largest value in the dataset.
- `n()` which provides the number of entries in a group. Note you don't specify a variable for n.

Let's seem them in action:

```
sumAtl <- atlNO2 %>%
  group_by(p, city) %>%
  summarize(mean = mean(conc),
            sd = sd(conc),
            min = min(conc),
            max = max(conc),
            n = n())

sumAtl
```

```
## # A tibble: 19 x 7
## # Groups:   p [4]
##    p     city                 mean    sd   min   max     n
##    <chr> <chr>               <dbl> <dbl> <dbl> <dbl> <int>
##  1 NB    Bathurst             1.21  1.91     0    27  8755
##  2 NB    Edmundston           4.52  4.82     0    45  8756
##  3 NB    Fredericton          1.82  4.04     0    39  8729
##  4 NB    Moncton              3.37  4.71     0    39  8749
```