R for Environmental Chemists

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

2021-06-15

Contents

Howdy

Howdy,

This website is more-or-less the living results of a collaborative project between the four of us. Our ultimate goal is not to be an exhaustive resource for all environmental chemist. Rather, we're focused on developing broadly applicable data science course content (tutorials and recipes) based in `R` for undergraduate environmental chemistry courses. Note that none of this has been reviewed yet and is not implemented in any capacity in any curriculum.

Authors

If you have any questions/comments/suggestions/concerns please email:

- Dave at davidross.hall@mail.utoronto.ca
- Steven at steven.kutarna@mail.utoronto.ca
- Kristen at kristen.yeh@mail.utoronto.ca
- Dr. D'eon at jessica.deon@utoronto.ca

Chapter 1

Introduction

- What you'll learn (and won't learn)
  - learn = basic understanding of R
  - won't learn = *L33T hax0r Skillz*
- How book is organized
  - code is covered in chunks or monospaced (`like this`)
- Prerequisite software
- getting help

1.1 Prerequisite software

In order to use this book you will first need to download and install **R** on your computer. The latest build as of May 2021 is v4.1.0. Download the appropriate version of R for your Operating System here.

We recommend using the default settings for installation (i.e. just keep clicking "Next"). *Note that you will not need shortcuts for launching R, as you will always be using Rstudio to run provided code.*

Once R is installed on your computer, you will need to download and install RStudio. Download the appropriate version of RStudio for your Operating System here

Again, just use the default installation settings.

1.2 Where to get help

While it's often tempting to contact your TA or Professor at the sign of first trouble, it's often better to try and resolve your issues on your own, especially if they're related to technical issues in R. 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 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:

- Stack overflow
- Using built-in documentation (`?help`)
- reference book such as the invaluable *R for Data Science*, which inspired this entire project.
- All else fails, holler at your TA/profs.

Getting Setup in R

Chapter 2

Running R Code

2.1 Navigating RStudio

Now you should have both R and RStudio downloaded and installed. When you open RStudio for the first time, this is what you should see:

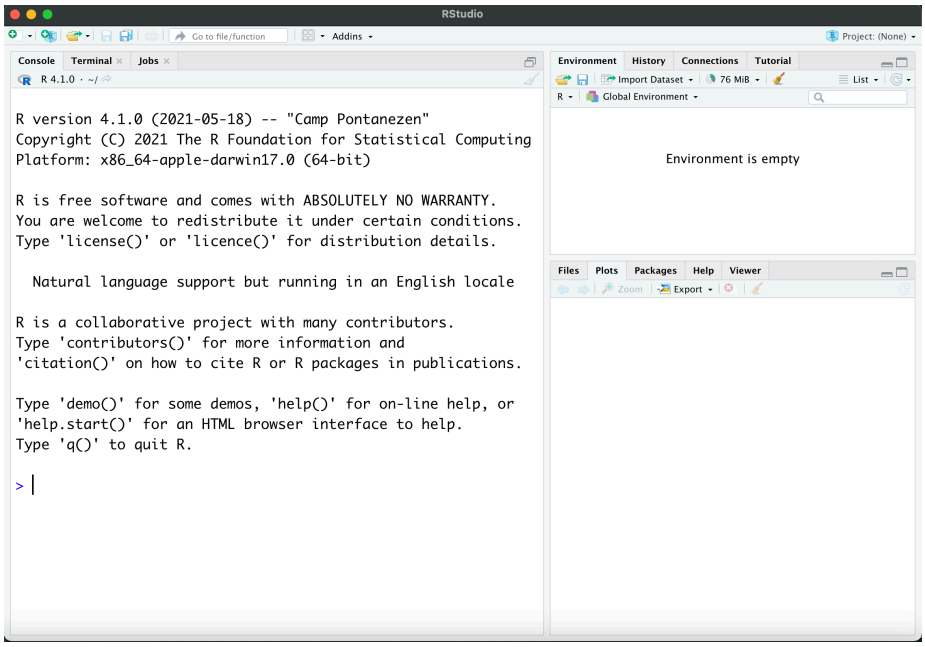


Figure 2.1: Figure 2.1: RStudio default startup view.

The version of R you just installed should appear in the main window, or console,

here:

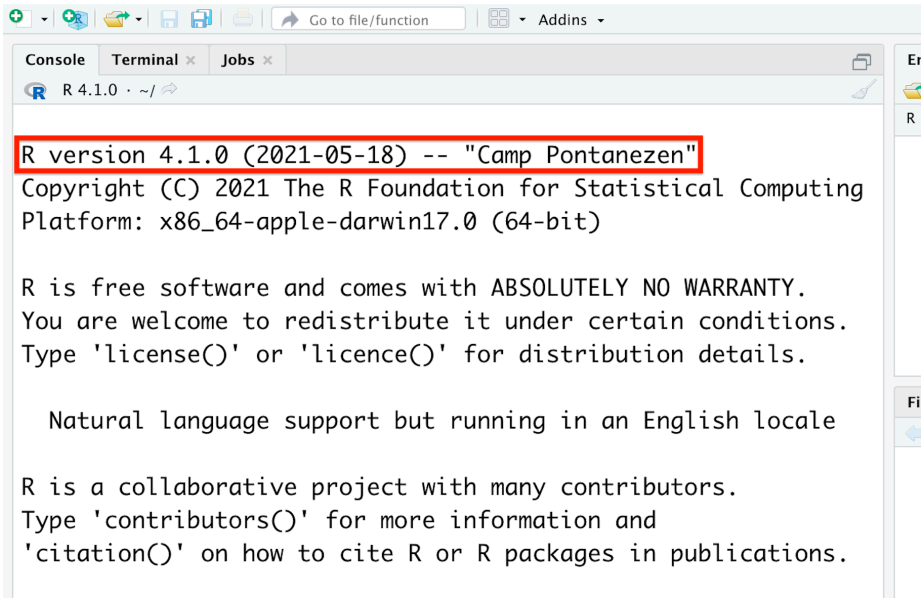


Figure 2.2: Figure 2.2: Locating R version in RStudio.

(If this message does not appear, go to *Tools->Global Options* and make sure that the "R version" box is set to the correct folder.)

It is important to be aware of the version of R you are using, especially when using R packages which may not be compatible with outdated versions of R (more on this in subsequent sections).

Currently, the RStudio interface has three key regions, as highlighted below.

2.1.1 Console

You can run code directly by typing your R code into the console pane. The console is mainly used for installing packages (more on that later), but can also be used for quick arithmetic. Try typing "2+2" into the console, then hit Enter.

```
2+2
```

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

While the console is useful for short-and-sweet commands, you will mainly be using the Scripts window to run chunks of code (see **2.2 Where to run code in RStudio**).

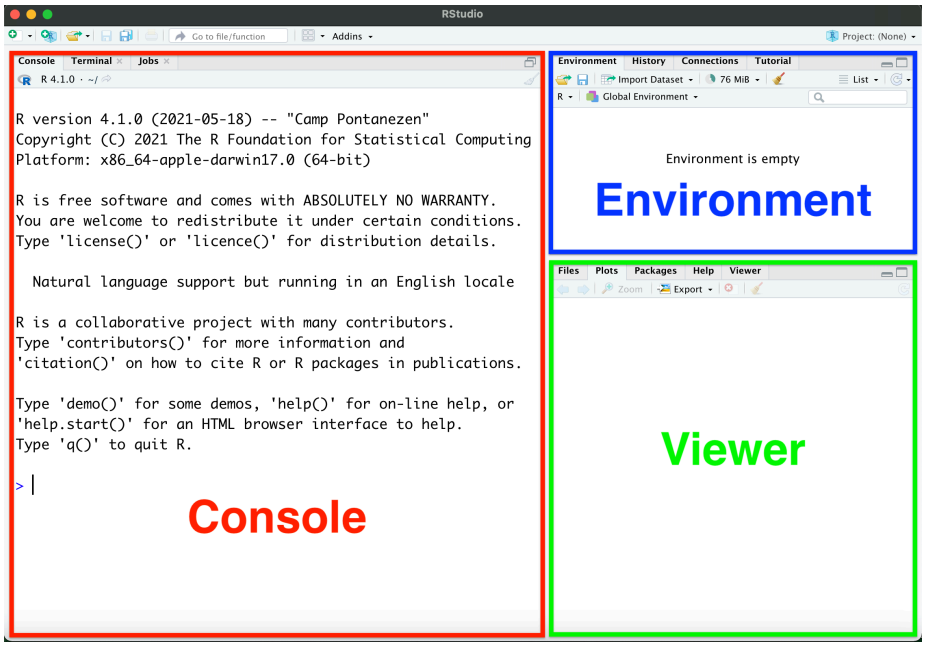


Figure 2.3: Figure 2.3: Important regions in the RStudio interface.

2.1.2 Environment

The environment window lists all variables, packages, and functions which you have run since opening RStudio. This window is particularly useful if you want to view your data in a format similar to an Excel spreadsheet directly from RStudio, which we will discuss in Chapter 3 of this book.

2.1.3 Viewer

The Viewer window has a couple of useful tabs. We will mainly use it to export plots (more on that in later chapters), but you can also open code files from the "Files" tab without having to leave RStudio (expanded on in Chapter 3).

2.1.4 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.

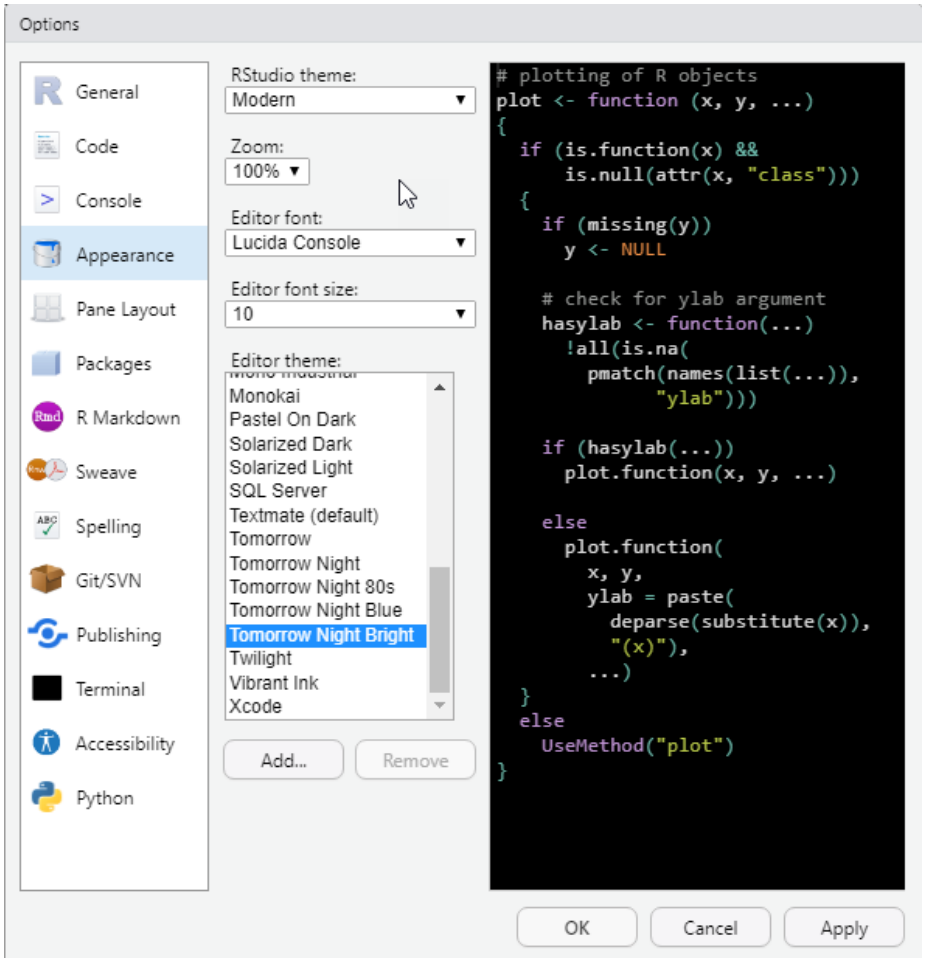


Figure 2.4: Figure 2.4: RStudio Appearance customization window.

2.2 Where to run code in RStudio

Files of R code are called scripts, which are saved in the `.R` format. 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.

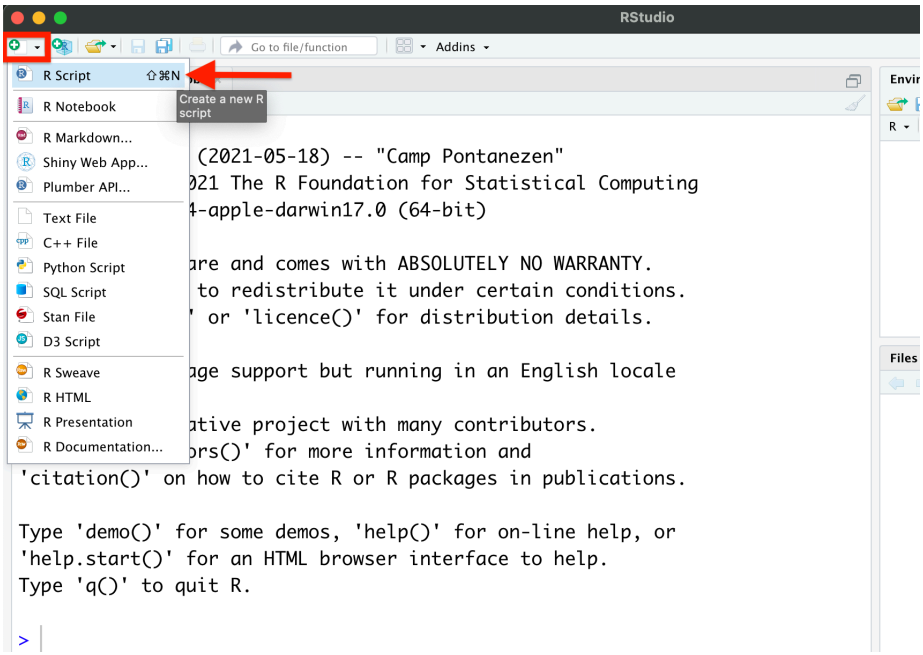


Figure 2.5: Figure 2.5: Opening a new script in RStudio.

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

Whenever you copy code blocks from this website (or other online sources), you should paste them into the Scripts window. You can then run the 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.

2.2.1 Scripts vs. console

Using the Scripts window to write, edit, and run your code has many advantages over using the console directly. Mainly, writing your code in scripts allows you to save your code in the R file format, as mentioned previously. This means you can open old files of code and easily run commands which you've previously written.

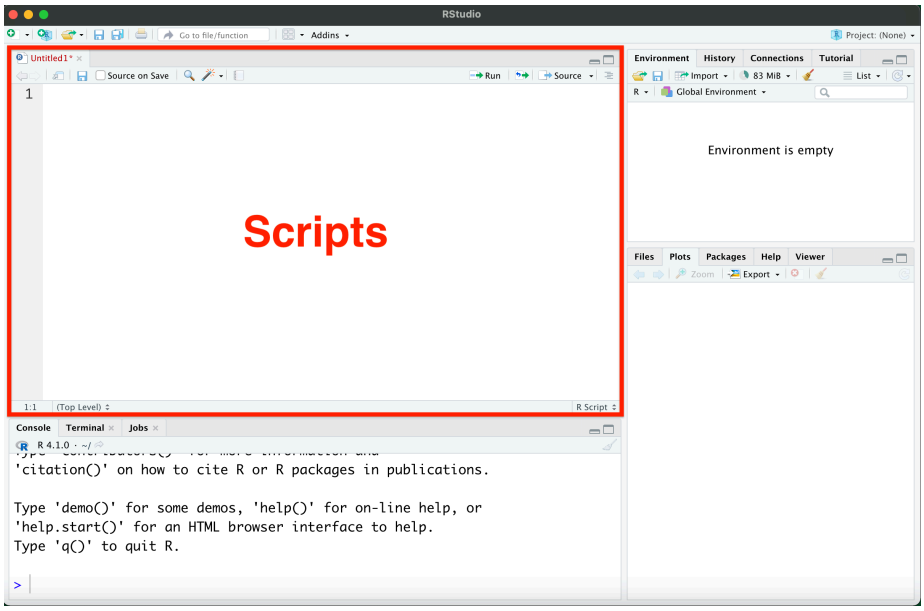


Figure 2.6: Figure 2.6: Scripts window in RStudio.

Additionally, the Scripts window allows you to review and edit your code without the risk of accidentally running an incomplete command. When typing directly into the console, any time you hit Enter, R will try to execute the line of code you have entered, whether or not you have completed the command. This is not an issue when typing your code into the Scripts window, as code in this window is only executed when the "Run" button is clicked, or when the code has been highlighted and you have pressed Ctrl+Enter (Cmd+Enter on Mac).

Scripts are also very useful when you are coding repetitive tasks. Let's say you want to run the same function on 10 different data sets. In the console, you would have to write out that function 10 times, and alter the input data set each time. In the Scripts window, you can write out the function once, then copy-paste it 9 times to subsequent lines in the script. You can then use the *Find/Replace* button, highlighted in the following image, to adjust the input data set in each call of the function. You would then run the code by highlighting all 10 lines at once, and hitting Ctrl+Enter (Cmd+Enter on Mac).

While the value of typing your code into the Scripts window is hard to articulate without getting into some examples of the coding language, you will come to appreciate using scripts and saving them as `.R` files as you become more familiar with coding in the RStudio interface.

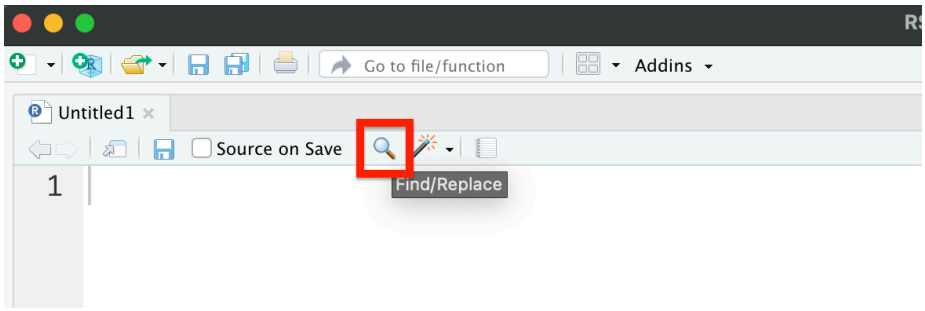


Figure 2.7: Figure 2.7: Find and replace button in the Scripts window of RStudio.

2.3 Coding building blocks

Now that you know how to navigate RStudio, and the different places we can enter code, let's learn some basic coding building blocks.

As mentioned earlier, R can be used as a calculator.

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

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

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

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

2.3.1 Variable assignment

You can assign the outputs of your arithmetic to variables using `<-`, as shown below. To view the contents of the variable, simply type your assigned variable name and press Enter. 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`.

```
x <- 10 / 2
x
```

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

You can also assign single values, character strings (text) or logical statements (`TRUE` or `FALSE`) to variables using the same notation.

```
fifty <- 50
fifty
```

```
## [1] 50
```

```
howdy <- "Howdy world!"
howdy
```

```
## [1] "Howdy world!"
```

```
t <- TRUE
t
```

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

These are examples of assignment statements. We will cover more assignments of more complex objects in the following sections and in subsequent chapters.

2.3.2 Naming variables

In the previous section we learned how to assign variables to names of your choice. Before you create variables of your own, it is a good idea to review what makes a good variable name and what makes a bad variable name, as well as some forbidden names in R.

Variable names can consist of letters, numbers, dots (`.`) and/or underlines (`_`). These names must begin with a letter or with the dot character, as long as not followed by a number (i.e., ".4five" is not a valid name). Variable names cannot begin with a number.

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.

Forbidden variable names in R are words which are reserved for other functions or coding purposes. These include, but are not limited to, `if`, `else`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NA`, and `NaN`.

2.3.3 Basic data structures

R has several data structures which will be briefly introduced here. 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 structures* chapter of *Advanced R* by Hadley Wickham, one of the authors of the *R for Data Science* referenced in the Introduction.

Data structures in R include vectors, lists, matrices, and data frames. Vectors and lists are examples of one dimensional data structures, while matrices and data frames are examples of two dimensional data structures.

Vectors contain multiple elements of the same type; either numeric, character (text), logical, or integer. Vectors are created using `c()`, which is short for combine. 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(T, T, T, F, F, F)
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("Hello", c(5,10,15,20), c(T, T, F))
str(hi)
```

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

```
hi
```

```
## [[1]]
## [1] "Hello"
##
## [[2]]
## [1]  5 10 15 20
##
## [[3]]
## [1]  TRUE  TRUE FALSE
```

Matrices and **data frames** are two dimensional data structures, similar to tables you would create in Excel. Matrices are created using `matrix()`, and three internal arguments; `data`, `ncol`, and `nrow`. An example of a matrix is shown below.

```
mat <- matrix(data = c(3, 4, 1, 6, 2, 9), ncol = 3, nrow = 2)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    3    1    2
## [2,]    4    6    9
```

Data frames are lists of equal-length vectors. While they are two dimensional data structures like matrices, data frames also share properties with one dimensional lists. Data frames are created using `data.frame()`.

2.3.4 R packages and functions

While there are many functions and operations available in base R, sometimes you may need to perform a task which base R functions do not cover. Fortunately, there are many R packages freely available online, each with their own suite of functions and capabilities. Some commonly used packages for data visualization and organization include `ggplot2` and `dplyr`. These are both included in the `tidyverse` package, which will be used in subsequent chapters.

In order to use an R package, you need to install the package in R. Usually this is done in the console, using the command `install.packages()`. You would then load the package into R, using the command `library()`. The `ggplot2` package is installed and loaded below.

```
install.packages("ggplot2")
library(ggplot2)
```

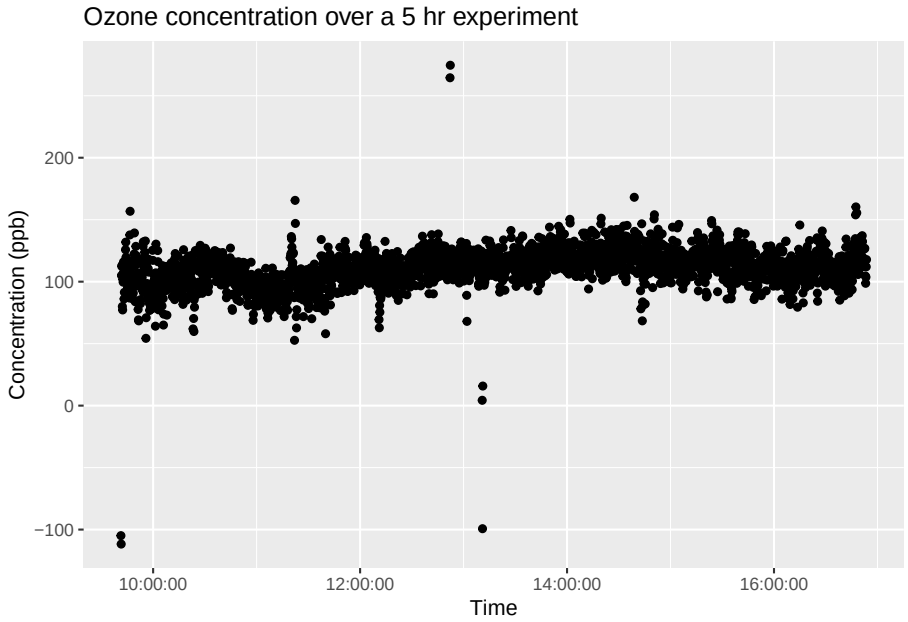
While you only need to install a package once, you will need to reload it every time you start a new R session or open a new R project. We will expand on R sessions and R projects Chapter 3.

Now that we have installed and loaded `ggplot2`, we can use the `ggplot` function to to visualize some data. First, we're going load the `readr` package, and use the `read_csv()` function to import some ozone concentration data. We will then use `ggplot()` with `geom_point()` to plot the ozone concentration over time in a scatter plot.

```
#load relevant packages
library(readr)
library(ggplot2)

#import ozone data set
ozone <- read_csv("./data/Ozone_oxidationexp.csv")

#plot ozone concentration vs. time
ggplot(data = ozone, aes(x = Time, y = Concentration)) +
  geom_point() + ggtitle("Ozone concentration over a 5 hr experiment") +
  ylab ("Concentration (ppb)")
```



2.4 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 your 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: Ozone time series script
#Author: Georgia Green
#Date: January 8, 2072

#import packages
library(dplyr)
library(ggplot2)
```

The rest of your script should be dedicated to executable code. It is good practice to include text comments throughout the script, in between different chunks of code, to remind yourself what the different sections of code are for (i.e., `#import 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. Headings are written with a single hashtag, followed by a space, as shown below. Subheadings are written with two hashtags, followed by a space, also shown below.

```
# Heading Text
## Subheading Text
```

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.

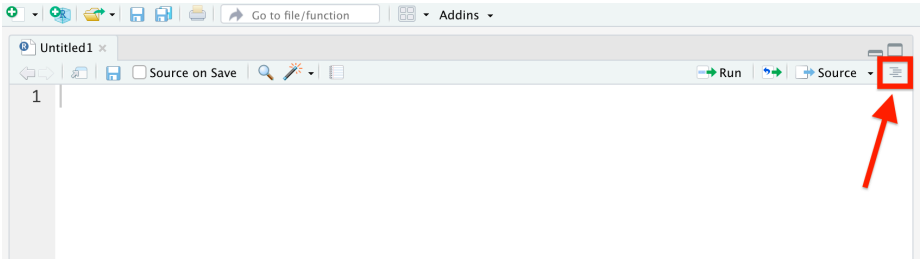


Figure 2.8: Figure 2.8: How to open the Document Outline box.

2.5 Viewing data and code simultaneously

Before we get into more about coding and workflows, you may want to know how to view your scripts and data side-by-side. You can open a script, plot, or data set in a new window by clicking and dragging the tab in RStudio (may not be compatible with Mac), or by clicking the button highlighted in the image below.

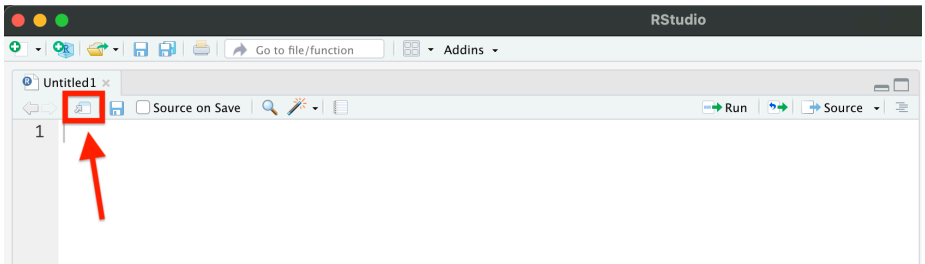


Figure 2.9: Figure 2.9: How to open an R script/plot/data set in a new window.

Now that you're familiar with navigating RStudio and some basic coding building blocks, let's move over to Chapter 3, where we'll review a normal workflow in R.

Chapter 3

R Workflows

Just like there's a common workflow in any chemistry lab (pre-lab, collect reagents, conduct experiment, etc.) there's a workflow when working with R. This is by no means the only way to work, but it's tried and true and will serve you well as you tackle your coursework.

3.1 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()`.

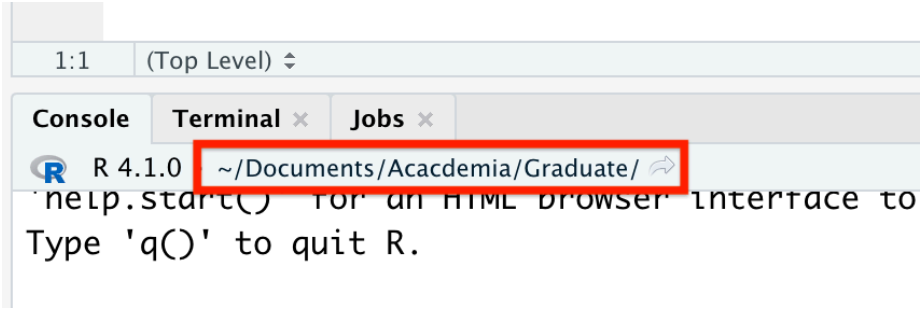


Figure 3.1: Figure 3.1: Working directory path displayed in the RStudio console

By default, R usually sets the working directory to the home directory on your

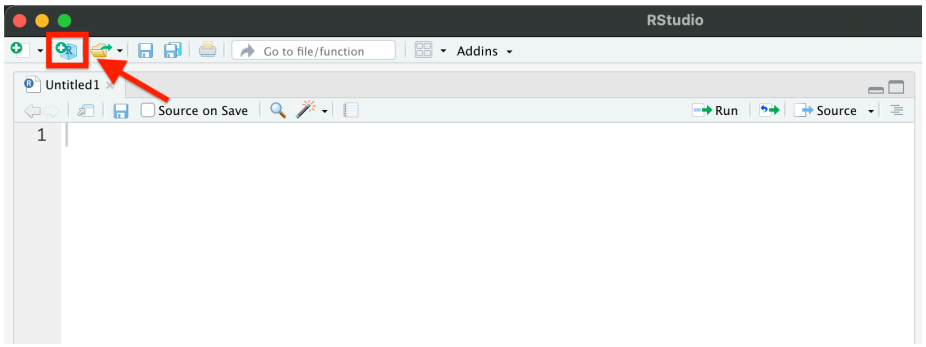
computer. The `~` symbol denotes the home directory, and can be used as a shortcut when writing a path that references the home directory.

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.

3.2 Creating an RStudio project

Let's go ahead and create a new **R 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 subfolder 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 subdirectory name and location. Enter your selected name and choose an appropriate location for the folder on your computer.

Click *Create Project*, and you should now see your chosen file path displayed in the bottom-right window:

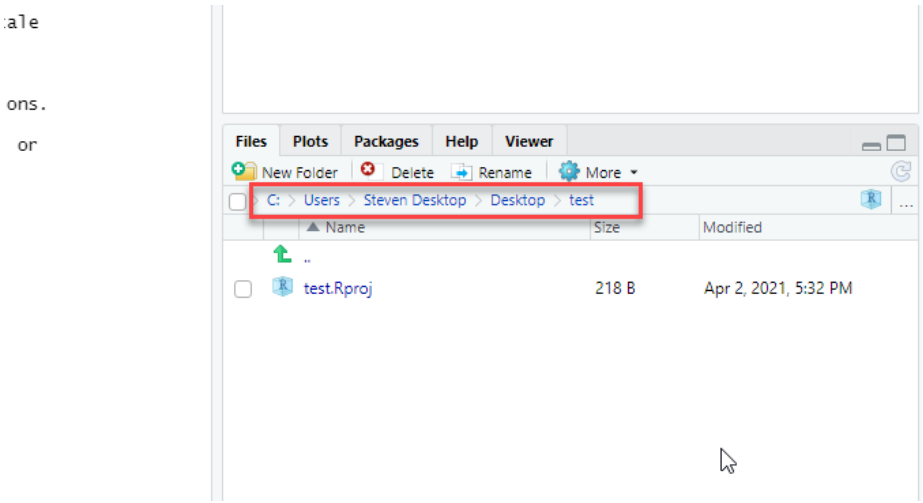


Figure 3.2: Figure 3.3: RStudio Project Folder

As mentioned previously, you can also view the file path to your project directory using `getwd()`. The output should match the file path shown in the image above.

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 subfolders for data and images, however, you may want to avoid making too many nested subfolders, as this will make your paths long and tiresome to type.

3.2.1 The value of R projects

To demonstrate the benefit of working in a project directory rather than using absolute paths, let's review a quick example.

Let's say you have a data set called `absorbance.csv`, which is stored on your computer in the lengthy file path `/Users/Your_Name/Documents/School/Undergrad/Second_Year/CHM210/Assignm` You want to import the contents of this data set into R using `read_csv`.

Improper referencing: When working *outside* of an R project, you would need to reference the full, absolute file path in your scripts in order for R to recognize the file you are looking for. If you wanted to import the file, you would need to type something like this:

```
abs <- read_csv("/Users/Your_Name/Documents/School/Undergrad/Second_Year/CHM210/Assignm
```

While this does the job, it is extremely tedious to type the entire file path without typos, and this also hinders sharing your work with colleagues. Other students/your instructors will not have `absorbance.csv` stored in the same, lengthy file path which you have referenced above. Thus, if they try to run your script, this line of code will throw an error, as there is no file named `absorbance.csv` on their computer at the given file path.

Proper referencing: When working *inside* of an R project, you would set your project directory to the folder `/Assignment1`, using the pop-up windows that appear after clicking *File->New Project->Existing Directory*. By default, whenever you open the R project, the working directory will automatically be set to `/Assignment1`, the folder containing the data set of interest. If you wanted to import the file now, you would write the following command:

```
abs <- read_csv("absorbance.csv")
```

This is much simpler to type, and is much more compatible with sharing your work. Even if you don't share your entire project directory with your colleagues, they should still be able to run this line of code in the script, as long as they have the `absorbance.csv` data set in their current working directory folder.

Not only does working within an R project make your scripts much easier to share with colleagues, TAs, and Profs, but it also makes it easier for you to resume working on your code after you have closed the RStudio application. Think of your scripts as tabs in a web browser. Sometimes a project may require you to have several scripts open at once.

If you are working *outside* of an R project and have multiple scripts open, all of the scripts will close automatically when you quit RStudio. The next time you open RStudio you'll have to manually locate and open up each of the scripts you were working on previously, which can be tedious if they're not stored in convenient locations.

If you are working *inside* of an R project and have multiple scripts open, R will leave the scripts open within the project even after you have quit RStudio. The next time you open RStudio and your project, the script tabs will remain open, allowing you to easily pick up where you left off.

You can try this out for yourself. Open up a new script in your current project in RStudio using *File->New File->R Script*. (If you don't have a project open currently, go to *File->New File*, click *New Directory*, then *New Project*.) Type in whatever you want. If you can't think of anything, here's an example:

```
# R projects are life savers
# wow
# blessed
```

Save the script by going to *File->Save*, or by clicking the button highlighted in the image below. Keep the script open, but close RStudio.

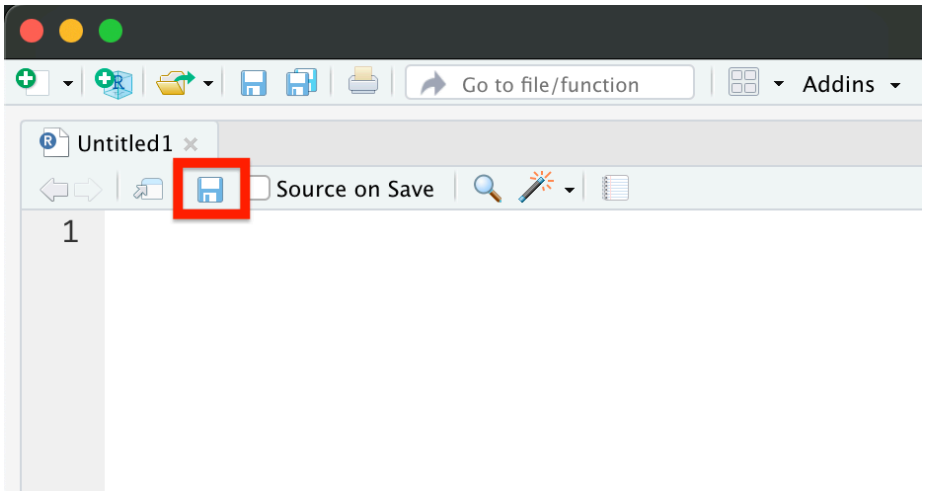


Figure 3.3: Figure 3.4: Save button in RStudio.

When you re-open RStudio, the script will still be there.

Leave the script open. Let's close the R project now. You can close the current project by going to *File->Close Project*, or by clicking the downwards arrow in the top right corner of RStudio, highlighted in the image below. Choose *Close Project* from the drop down menu.

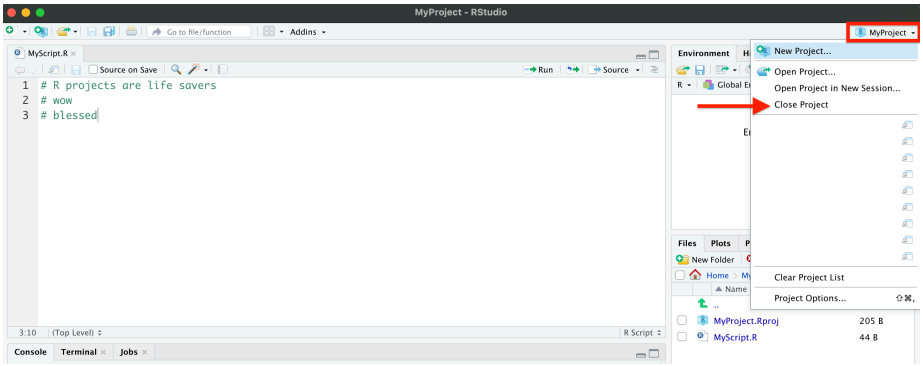


Figure 3.4: Figure 3.5: How to close an R Project.

Now close RStudio. When you open RStudio again, no script will be open! The same is true when you work outside of an R project.

3.3 Saving things in R

As mentioned in the previous section, you can save an R script to a `.R` file by going to *File->Save*, or by clicking the button highlighted in Figure 3.4. Code saved to a `.R` file is considered *real*. Variables, plots, or data sets that only exist in your workspace (shown in the Environment window) are not.

Whenever you close RStudio, any objects in R that are not considered *real* will be lost in that R session. For example, let's say you have some vectors in your workspace. Even if you are working within an R project and have saved your script, those vectors will not be remain in the workspace after you close and re-open RStudio. You would need to re-run the code you used to generate those vectors the next time you open RStudio, in order to replicate the workspace you had before you closed the application.

Let's try this out in practice. Assign something of your choice to a variable (i.e., "text must be in quotations," numbers, TRUE or FALSE), as is done below.

```
real <- "What is real?"
```

The variable should now show up in your environment in RStudio, much like the image below.

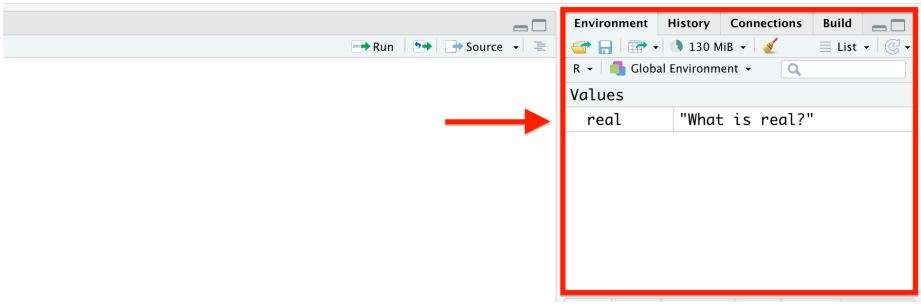


Figure 3.5: Figure 3.6: Items in the workspace/R environment.

When a variable exists within your workspace, you can simply type the variable name into the console to have R print out its contents (examples given in Chapter 2). However, when you close RStudio, and the variable is erased from the workspace, R will give you an error when you try to call your assigned variable. The errors will look a little bit like this:

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

```
## [1] "Hello"
##
## [[2]]
## [1]  5 10 15 20
##
## [[3]]
## [1]  TRUE  TRUE FALSE
```

Where `'hi'` is the name of your variable. This is common notation for errors that occur when you try to run code that references an object (variable, vector, data set, etc.) that might have once existed in your workspace, but currently does not. If you're getting an `object not found` error, this generally means you need to go back and load your data set, or re-run the code used to generate your desired variable or vector. Once you've loaded the data set, or re-created the object in the workspace, the error will go away and your code should run as expected.

While this might not seem like a major issue at this point in time, it certainly will when you start writing lengthy scripts that create variables in the early lines and then reference those variables in commands in later lines. One day you might find yourself opening up RStudio and immediately trying to run some code from the end of one of your scripts. `Object not found` is a common thing to see in response. Don't fret - run the script from beginning to end, in order, and chances are this error will go away (provided that you have included your assignment statements in your saved script).

3.3.1 What should I save?

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

1. R scripts saved to `.R` files are real.
2. Objects in your workspace/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 workspace.

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

It is paramount that you save the scripts you code in, and that you save them regularly. Even if you've made small notation changes to the code, it is always a good idea to save your changes to the script before closing RStudio, as there is a good chance you will not remember the minor differences upon returning.

While it is possible to save your workspace in RStudio, we do not recommend this. It is much easier to recreate your workspace from your R scripts than it is to recreate your scripts from your workspace. That being said, it is integral to get into good script formatting habits.

You want to make sure that even if you lose an object in your environment, your script still contains the code you used to generate that object. You also want to make sure that you generate the object before you call it in part of another command, so that when you run your scripts from top-to-bottom, the variables are generated in the workspace before they are referenced by later commands. Let's look at some examples.

```
# generate vectors
x <- c(1, 2, 3, 5, 76, 8, 2)
y <- c(T, T, T, F, F, F, F)

# turn vectors into data frame
data.frame(x,y)
```

```
##    x     y
## 1  1  TRUE
## 2  2  TRUE
## 3  3  TRUE
## 4  5 FALSE
## 5 76 FALSE
## 6  8 FALSE
## 7  2 FALSE
```

The code above is ordered correctly. Your scripts should be ordered similar to this (with the exception that you will load R packages before you generate your variables), with your objects generated before you call them using a function.

```
# make data frame
data.frame(x,y)

# generate vectors
x <- c(1, 2, 3, 5, 76, 8, 2)
y <- c(T, T, T, F, F, F, F)
```

The code above is ordered *incorrectly*. You can see that using `data.frame()` before generating vectors `x` and `y` has resulted in an error, similar to what we saw in the previous section. You should follow the format from the first example to avoid errors of this sort in your scripts.

3.3.1.1 Saving objects

In some cases, your code may be used to generate large data structures which require quite a bit of input 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 structure to a *real* file that you can simply import the next time you open the application. Fortunately, this is possible thanks to `save()` and `load()`.

The `save()` function saves the input object or objects (i.e., vector, matrix, or data frame) to your working directory as an `.rda` file. Let's try saving our vectors `x` and `y` from the previous examples to a file called `vec.rda`.

```
# save vectors
save(x, y, file = "vec.rda")
```

An `.rda` file should now exist in your project directory, as shown in the image below.

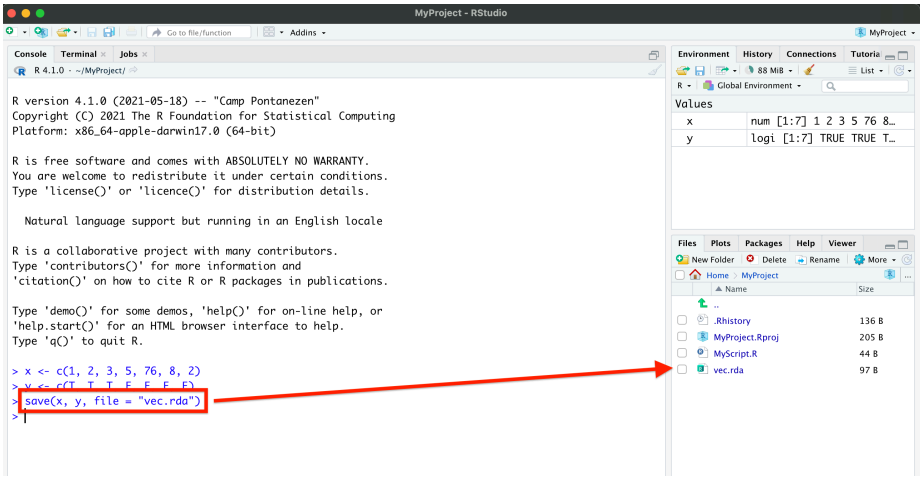


Figure 3.6: Figure 3.7: Generated `.rda` file after using `save()`.

Now, let's try removing our vectors from our workspace and loading them from the `.rda` file we just generated. You can use `rm()` to remove objects from your workspace, as shown below.

```
rm(x, y)
```

Now that our workspace is empty, if you type `x` or `y` into the console, RStudio will not know what to return in response. We have to import the data back into our workspace so that RStudio knows what `x` and `y` contain.You can use `load()` to import the vectors to your current RStudio workspace, using the format below.

```
load(file = "vec.rda")
```

The vectors should now be present in the workspace again, and can be printed to the console if you type in their respective names.

3.4 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 become more familiar with R and start using more complex functions, you will become better acquainted with error messages in R, and how to deal with them accordingly.

We'll go through a few examples of error messages in the following sections, as well as how to read the errors, and how to fix your code to resolve the issues.

3.4.1 Script diagnostics

When writing code in the Script window, 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.

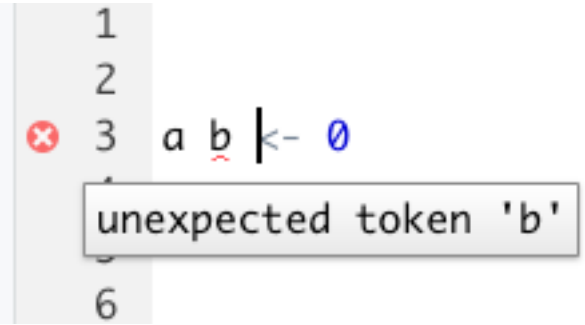


Figure 3.7: Figure 3.8: RStudio highlights syntax errors in the Scripts window.

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 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 = ozone, aes(x = Time, y = Concentration))
```

(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:

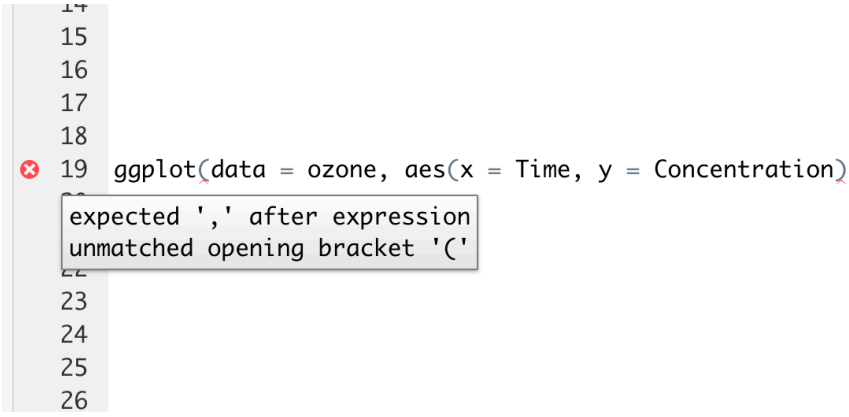


Figure 3.8: Figure 3.9: RStudio highlights unmatched parentheses in the script window.

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

3.4.2 Reading error codes

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. If you try to add, divide or multiply two character strings using arithmetic operations in the console, you will get the same error.

```
"hey" * "hi"
```

```
## Error in "hey" * "hi": non-numeric argument to binary operator
```

It is important to be aware of these error codes as many functions require specific data types as their inputs. You can always look at the required data type by looking at the documentation for the function (generally, this can be viewed by typing `?function` into the console, where `function` is the name of the function). If the function requires numeric data, inputting character strings or logical values will throw the errors shown above. If the function requires logical values, inputting numeric data or character strings will throw the errors shown above.

In order to avoid these errors, make sure that you are using the right type of data in your functions. You can always check your data type using `class()`. Some examples are shown below.

```
class("hi")
```

```
## [1] "character"
```

```
class(10)
```

```
## [1] "numeric"
```

```
class(1L)
```

```
## [1] "integer"
```

```
class(TRUE)
```

```
## [1] "logical"
```

Now that you're familiar with working in RStudio, saving your projects, scripts, and data, let's move over to Chapter 4, where we'll discuss the advantages and disadvantages of using `.Rmd` documents instead of `.R` scripts.

Chapter 4

Using R Markdown

In a nutshell, R Markdown allows you to analyse your data with R and write your report in the same place (this document is written with R Markdown). This has loads of benefits including increased reproducibility, 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:

```
# Look at me go mom
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.

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.

Table 4.1: Example table of airborne pollutant levels used for Figure 1.

temperature	pollutant	concentration	date
-11.7	NO2	41	2018-01-01 19:00:00
-11.7	O3	2	2018-01-01 19:00:00
-11.3	NO2	28	2018-01-01 20:00:00
-11.3	O3	14	2018-01-01 20:00:00
-11.6	NO2	20	2018-01-01 20:59:59

4.1 Let's dig a little deeper

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/Toronto_60433_2018_Jan2to8.csv",
                   na = "-999")
kable(airPol[1:5, ],
      caption = "Example table of airborne pollutant levels used for Figure 1.")
```

```
ggplot(airPol, aes(date, concentration, colour = pollutant)) +
  geom_line() +
  theme_classic()
```

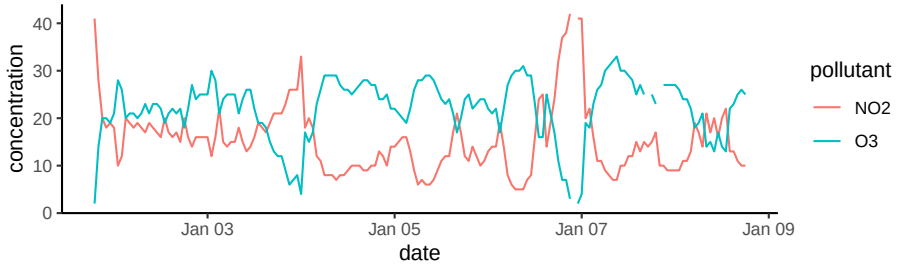


Figure 4.1: Time series of 2018 ambient atmospheric O_3 and NO_2 concentrations (ppb) in downtown Toronto

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.

4.2 How do I get started with R markdown?

As you've already guessed, R markdown documents use R and are most easily written and assembled in the R Studio IDE. If you have not done so, download R from the comprehensive R archive network (CRAN), link here: `http://cran.utstat.utoronto.ca/`, and R Studio, link here: `https://rstudio.com/products/rstudio/download/`). Follow the listed instructions and you should be well on your way. You can also see the accompanying *Working with RStudio* document on Quercus for additional top tips.

Once setup with R and R Studio, we'll need to install the `rmarkdown` and `tinytex` packages. In the console, simply run the following code:

```
install.packages("rmarkdown") # downloaded from CRAN

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

The `rmarkdown` 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 -> Rmarkdown`. 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.

4.2.1 Great, now what's going on with this R markdown document?

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

TA. 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. Let's break down what the R markdown syntax means then let's knit our 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: "Untitled"
author: "Jean Guy Rubberboots"
date: "20/04/2021"
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 "Untitled," it's written by yours truly, on the 24th of August, 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.

Reading further down the default R markdown code, you'll see different blocks of text. In R markdown anything you write will be interpreted as body text (i.e .the stuff you want folks reading like this) in the knitted document. **To actually run R code** you'll need to see the next section.

4.2.2 How to run R code in R Markdown

There's two ways to write `R` code in markdown:

- **Setup a code chunk**. Code chunks start with three back-ticks like this: ```` ```{r} ````, where `r` indicates you're using the R languauge. You end a code chunk using three more backticks like this ```` ``` ````.
  - 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.
- **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 (ex. We found that x is 4)

A screenshot of how this document, the one you're reading, appeared in R Studio is shown in Figure 2.

To actually run your R code you have two options. The first is to run the individual chunks using the *Run current chunk* button (See figure 2). This is a great way to tinker with your code before you compile your document. The second option is to compile your entire document using the *Knit document* button (see Figure 2). 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. **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.

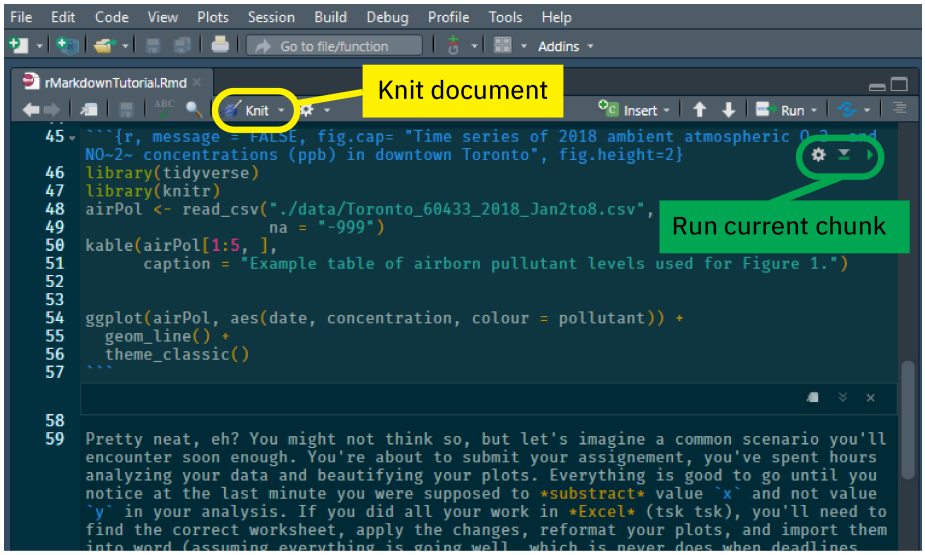


Figure 4.2: How this document, the one you're reading, appeared in RStudio; to see the final results scroll up to Figure 1. Note the "knit" and "run current chunk" buttons.

4.2.3 How do I go from R markdown to something I can hand-in

To create a PDF to hand in you'll need to compile, or knit, your entire markdown document as mentioned above. To knit (or compile) 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.

If you want to test how your code chunks will run, R Studio shows a little green 'play button' on the top right of every code chunk. this is the 'run current chunk' button, and clicking it will run your code chunk and output whatever it would in the final R markdown document. This is a great way to tweak figures and codes as it avoids the need to compile the entire document to check if you managed to change the lines from 'black' to 'blue' in your plot.

4.3 So now what do I do with R Markdown?

You do science and you write it down!

In all seriousness though, this document was only meant to introduce you to R markdown, and to make the case that you should use it for your ENV 316 coursework. A couple of the most useful elements are talked about below, and there is a wealth of helpful resources for formatting your documents. Just remember to keep it simple, there's no need to reinvent the wheel. The default R markdown outputs are plenty fine with us.

4.3.1 R Markdown resources and further reading

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

4.3.2 R code chunk options

You can specify a number of options for an individual R code chunk. You include these at the top of the code chunk. 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." A list of code chunk options is shown below:

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

# some code to generate a plot worth captioning.

```
````

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 ("")

4.3.3 Inserting images into markdown documents

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. Note that compiling to PDF, the LaTeX call will place your image in the "optimal" location, so you might find your image isn't exactly where you though it would be. A quick google search can help you out if this is problem.

```
![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 `%`.

4.3.4 Generating Tables

There's multiple methods to create tables in R markdown. Assuming you want to display results calculated through R code, you can use the `kable()` function. Please consult Chapter 10 of the *R Markdown Cookbook* for additional support.

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

4.3.5 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 with also check your R code; be wary of changing words in your code chunks because you may get an error down the line.

4.3.6 Quick reference on R markdown syntax

- **Inline formatting**; *which* is $_{\text{used}}$ to $^{\text{format}}$ `your text`.

1. Numbered lists

- Normal lists
  - Lists
- **Block-level elements**, i.e. you're section headers

Example R markdown syntax used for formatting shown above:

```
- **Inline formatting**; *which* is ~used~ to ^format^ `your text`.
1. Numbered lists
- Normal lists
  - Lists

- **Block-level elements**, i.e. your section headers

# Headers
## Headers
### Headers
```

Data Analysis in R

Chapter 5

Intro to Data Analysis

This section will teach you **how** to use R to meet your data analysis needs using a common workflow. Whether it takes 10 minutes or 10 hrs, *you'll use this workflow for every data analysis project*. By explicitly understanding the workflow steps, and how to execute them in R, you'll be more than capable of expanding the limited tools learned from this book to any number of data analysis projects you'll soon encounter.

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

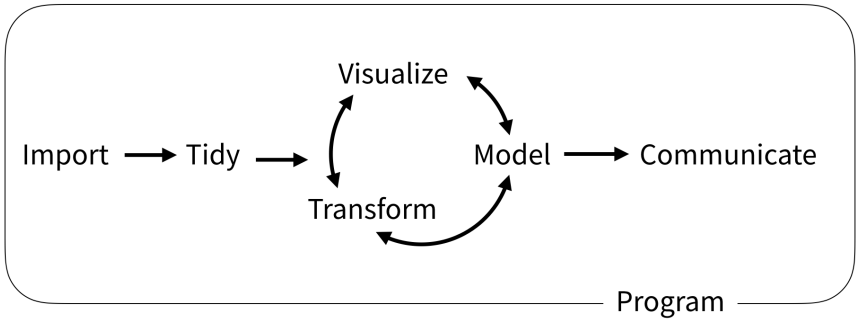


Figure 5.1: Data science workflow describes by Wickham and Grolemund; image from *R for Data Science*, Wickham and Grolemund (2021)

- **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 errors.
- **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.

The *Transform*, *Visualize*, and *Model* cycle exists because these steps often feed into one another. For example, you'll often transform your data, make a quick model, then visualize it to see how it performs. Other times, you'll visualize your data to see what type of model can explain it, and if any transformations are necessary. This is the beauty of R (and coding in general). Once you've setup everything, these steps are fairly simple to execute allowing you to quickly explore your data from a number of different angles. The next section will explore the theory (the **why**) behind these steps, and introduce some tools you can use to better explore your data.

5.1 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 6

Importing 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).

6.1 How data is stored

While there are a myriad of ways data is stored, notably 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.

6.2 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")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   wavenumber = col_double(),
##   EPDM = col_double(),
##   Polystyrene = col_double(),
##   Polyethylene = col_double(),
##   `Sample: Shopping bag` = col_double()
## )
```

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 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
```

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))
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   X1 = col_character(),
##   X2 = col_character(),
##   X3 = col_character(),
##   X4 = col_character(),
##   X5 = col_character()
## )
```

```
## # 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 below that since the headers are now considered data, the entire column is 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))
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   X1 = col_double(),
##   X2 = col_double(),
##   X3 = col_double(),
##   X4 = col_double(),
##   X5 = col_double()
## )
```

```
## # 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
```

6.2.1 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"
```

6.3 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.

6.4 Saving data

As you progress with your analysis you may want to save intermediate or final datasets. This is readily accomplished using the `write.csv` (base R) or `write_csv` (tidyverse) functions. Similar rules apply to how we used `read_csv`, but now the second argument specifies the save location and file name, 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.

6.5 Further Reading

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

Chapter 7

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 some of the tools of 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.

7.1 What is tidy data?

Tidy data has "…each variable in a column, and each observation in a row…" (Wickham 2014) This may seem obvious to you, but let's consider how data is often recorded in lab, as exemplified in Figure 7.1A. 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 7.1B, 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 7.1C).

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

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 7.1: (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.

read across a row to get the gist of one data point (i.e. "Our blank has a 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.

7.2 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:

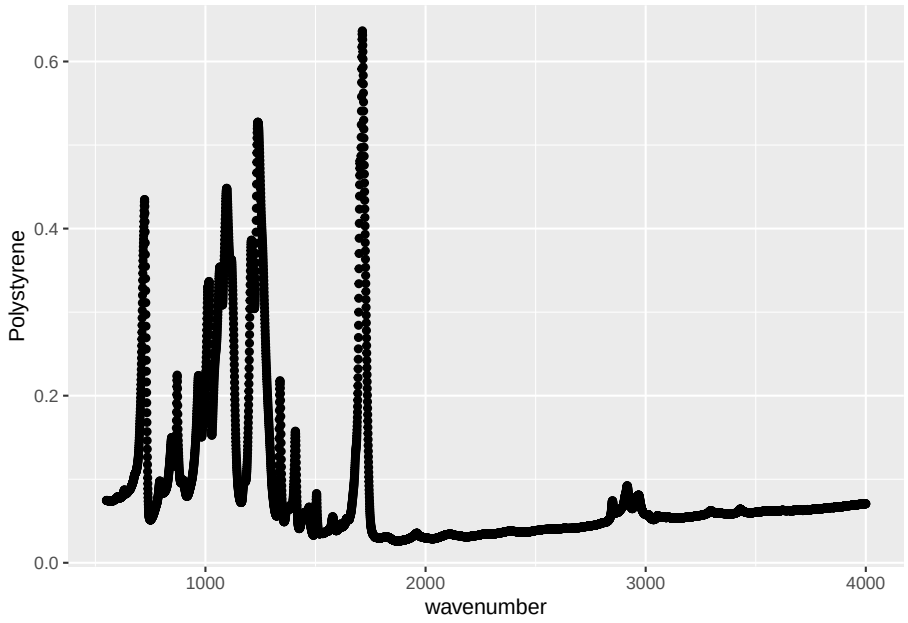
```
atr_plastics <- read_csv("data/ATR_plastics.csv")

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

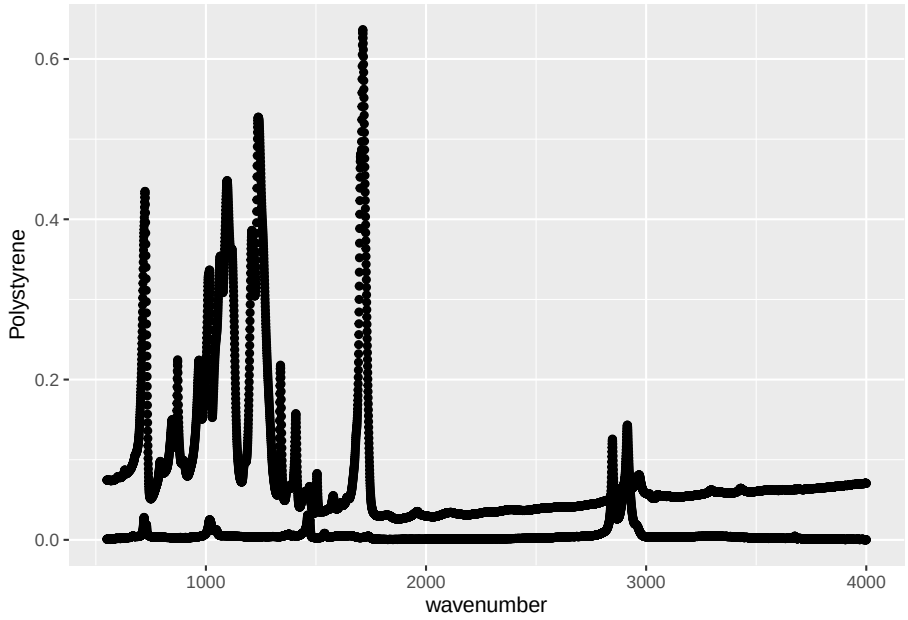
As we can see this our ATR spectroscopy results of several plastics, as recorded for a *CHM 317* lab, is structured similarly to the example in Figure 7.1A. The ATR absorbance spectra of the four plastics are recorded in separate columns. Again, this format makes intuitive sense when recording in the lab, and for working in Excel, but isn't the friendliest with R. In the example below we can

only specify one `y` value for `ggplot` to plot. In our example 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))
```



7.2.1 Making data 'longer'

While 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 <- atr_plastics %>%
  pivot_longer(cols = -wavenumber,
               names_to = "sample",
               values_to = "absorbance")

head(atr_long)
```

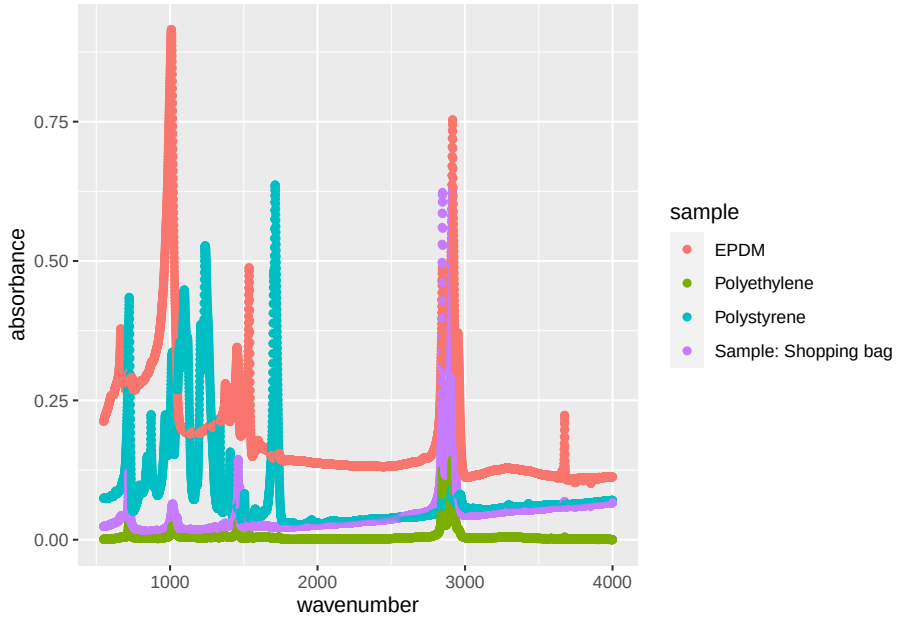
```
## # A tibble: 6 x 3
##   wavenumber sample                absorbance
##        <dbl> <chr>                      <dbl>
## 1       550. EPDM                     0.212
## 2       550. Polystyrene              0.0746
## 3       550. Polyethylene             0.000873
## 4       550. Sample: Shopping 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 understand how our code could scale to accommodate any number of different

samples, whereas the previous attempt 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 <- example %>%
 pivot_longer(
   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
```

7.2.2 Making data 'wider'

Sometimes packages or circumstances will require you reformat your data into a matrix or 'wide' format (notable the `matrixStats` and `matrixTests` packages). You can accomplish this using the `pivot_wider` function, which operates inverse to the `pivot_longer` function described above. For example the input

`names_from` is used to specify which variables are to be converted to headers. You can read up on the `pivot_wider` function here

7.2.3 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. You've crammed as much information as possible into that name so you and them 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_example %>%
  separate(
    col = sample,
    into = c("location", "chemical", "replicateNum"),
    sep = "_",
    remove = TRUE,
    convert = TRUE)
```

```
##   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.

Again it's paramount to **be consistent when recording data**.

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

7.2.5 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:

```
colnames(badHeader)
```

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

```
colnames(rename(badHeader, wavelength_nM = UVVis_Wave_Length_nM))
```

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

7.2.6 Rounding numbers

If you want to round the numbers in your data to account for significant figures or something, you can do so using the `round` function.

```
head(example)
```

```
##   measurement absorbance     conc
## 1           A    123.123 1.100000
## 2           B    300.000 3.000022
## 3           C    175.547 1.750000
```

```
# rounding 'conc' column to 1 decimal.

example %>%
  mutate_at(vars(conc), round, 1)
```

```
##   measurement absorbance conc
## 1           A    123.123  1.1
## 2           B    300.000  3.0
## 3           C    175.547  1.8
```

7.3 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 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.

7.4 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.

Chapter 8

Transform: dplyr and 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 functionailities of the `dplyr` package, explicitly introduce the pipe operator `%>%`, and showcase how you can leverage these tools to quickly manipulate your data.

The benchmark `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 equally 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.

8.1 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 mg/L  1          1349.
## 2 blank       0 mg/L  2          1304.
## 3 blank       0 mg/L  3          1396.
```

Note how we need to pass logical operations to `filter()`. 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.

8.1.1 Logical oeprators

`filter()` can use other *relational* and *logical* operators, or combinations thereof, to improve your sub-setting. 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: 8 x 5
##   type     conc_Na units replicate signal
##   <chr>      <dbl> <chr> <chr>      <dbl>
## 1 blank          0 mg/L  1          1349.
## 2 blank          0 mg/L  2          1304.
## 3 blank          0 mg/L  3          1396.
## 4 standard     0.1 mg/L  1          2947.
## 5 standard     0.1 mg/L  2          2924.
## 6 standard     0.1 mg/L  3          2927.
## 7 standard     0.2 mg/L  1          4446.
## 8 standard     0.2 mg/L  3          4416.
```

- Selecting signals between values

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

```
## # A tibble: 6 x 5
##   type     conc_Na units replicate signal
##   <chr>      <dbl> <chr> <chr>      <dbl>
## 1 standard     0.2 mg/L  2          4453.
## 2 standard     0.3 mg/L  1          6235.
## 3 standard     0.3 mg/L  2          6207.
## 4 standard     0.3 mg/L  3          6267.
## 5 standard     0.4 mg/L  2          8141.
## 6 standard     0.4 mg/L  3          8106.
```

- 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   mg/L  1          1349.
##  2 blank        0   mg/L  3          1396.
##  3 standard     0.1 mg/L  1          2947.
##  4 standard     0.1 mg/L  3          2927.
##  5 standard     0.2 mg/L  1          4446.
##  6 standard     0.2 mg/L  3          4416.
##  7 standard     0.3 mg/L  1          6235.
##  8 standard     0.3 mg/L  3          6267.
##  9 standard     0.4 mg/L  1          8173.
## 10 standard     0.4 mg/L  3          8106.
```

- 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   mg/L  1          1349.
## 2 blank        0   mg/L  2          1304.
## 3 blank        0   mg/L  3          1396.
## 4 standard     0.1 mg/L  1          2947.
## 5 standard     0.2 mg/L  1          4446.
## 6 standard     0.3 mg/L  1          6235.
## 7 standard     0.4 mg/L  1          8173.
```

- 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   mg/L  1          1349.
```

```
##  2 blank        0   mg/L  2          1304.
##  3 blank        0   mg/L  3          1396.
##  4 standard     0.1 mg/L  1          2947.
##  5 standard     0.1 mg/L  2          2924.
##  6 standard     0.1 mg/L  3          2927.
##  7 standard     0.2 mg/L  1          4446.
##  8 standard     0.2 mg/L  2          4453.
##  9 standard     0.2 mg/L  3          4416.
## 10 standard     0.3 mg/L  1          6235.
## 11 standard     0.3 mg/L  2          6207.
## 12 standard     0.3 mg/L  3          6267.
## 13 standard     0.4 mg/L  1          8173.
## 14 standard     0.4 mg/L  2          8141.
## 15 standard     0.4 mg/L  3          8106.
```

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.

8.2 Arranging rows

`arrange()` simple 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 signal in increasing by increasing order:

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

```
## # A tibble: 15 x 5
##    type     conc_Na units replicate signal
##    <chr>      <dbl> <chr> <chr>      <dbl>
##  1 blank        0   mg/L  2          1304.
##  2 blank        0   mg/L  1          1349.
##  3 blank        0   mg/L  3          1396.
##  4 standard     0.1 mg/L  2          2924.
##  5 standard     0.1 mg/L  3          2927.
##  6 standard     0.1 mg/L  1          2947.
##  7 standard     0.2 mg/L  3          4416.
##  8 standard     0.2 mg/L  1          4446.
##  9 standard     0.2 mg/L  2          4453.
## 10 standard     0.3 mg/L  2          6207.
## 11 standard     0.3 mg/L  1          6235.
## 12 standard     0.3 mg/L  3          6267.
## 13 standard     0.4 mg/L  3          8106.
```

```
## 14 standard     0.4 mg/L  2          8141.
## 15 standard     0.4 mg/L  1          8173.
```

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

```
# Note the order of precedence
arrange(FAES, desc(conc_Na), signal)
```

```
## # A tibble: 15 x 5
##    type     conc_Na units replicate signal
##    <chr>      <dbl> <chr> <chr>      <dbl>
##  1 standard     0.4 mg/L  3          8106.
##  2 standard     0.4 mg/L  2          8141.
##  3 standard     0.4 mg/L  1          8173.
##  4 standard     0.3 mg/L  2          6207.
##  5 standard     0.3 mg/L  1          6235.
##  6 standard     0.3 mg/L  3          6267.
##  7 standard     0.2 mg/L  3          4416.
##  8 standard     0.2 mg/L  1          4446.
##  9 standard     0.2 mg/L  2          4453.
## 10 standard     0.1 mg/L  2          2924.
## 11 standard     0.1 mg/L  3          2927.
## 12 standard     0.1 mg/L  1          2947.
## 13 blank        0   mg/L  2          1304.
## 14 blank        0   mg/L  1          1349.
## 15 blank        0   mg/L  3          1396.
```

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

8.3 Selecting by column or variable

`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  1349.
##  2  1304.
##  3  1396.
##  4  2947.
##  5  2924.
##  6  2927.
##  7  4446.
##  8  4453.
##  9  4416.
## 10  6235.
## 11  6207.
## 12  6267.
## 13  8173.
## 14  8141.
## 15  8106.
```

You can also select multiple columns using the same helper functions describes in Importing data into R.

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

```
## # A tibble: 15 x 3
##    conc_Na units replicate
##      <dbl> <chr> <chr>
##  1     0   mg/L  1
##  2     0   mg/L  2
##  3     0   mg/L  3
##  4     0.1 mg/L  1
##  5     0.1 mg/L  2
##  6     0.1 mg/L  3
##  7     0.2 mg/L  1
##  8     0.2 mg/L  2
##  9     0.2 mg/L  3
## 10     0.3 mg/L  1
## 11     0.3 mg/L  2
## 12     0.3 mg/L  3
## 13     0.4 mg/L  1
## 14     0.4 mg/L  2
## 15     0.4 mg/L  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
```

8.4 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   mg/L  1          1349.      10
##  2 blank        0   mg/L  2          1304.      10
##  3 blank        0   mg/L  3          1396.      10
##  4 standard     0.1 mg/L  1          2947.      10
##  5 standard     0.1 mg/L  2          2924.      10
##  6 standard     0.1 mg/L  3          2927.      10
##  7 standard     0.2 mg/L  1          4446.      10
##  8 standard     0.2 mg/L  2          4453.      10
##  9 standard     0.2 mg/L  3          4416.      10
## 10 standard     0.3 mg/L  1          6235.      10
## 11 standard     0.3 mg/L  2          6207.      10
## 12 standard     0.3 mg/L  3          6267.      10
## 13 standard     0.4 mg/L  1          8173.      10
```

```
## 14 standard     0.4 mg/L  2          8141.      10
## 15 standard     0.4 mg/L  3          8106.      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   mg/L  1          1349.      10     13485.
##  2 blank        0   mg/L  2          1304.      10     13041.
##  3 blank        0   mg/L  3          1396.      10     13958.
##  4 standard     0.1 mg/L  1          2947.      10     29473.
##  5 standard     0.1 mg/L  2          2924.      10     29244.
##  6 standard     0.1 mg/L  3          2927.      10     29273.
##  7 standard     0.2 mg/L  1          4446.      10     44464.
##  8 standard     0.2 mg/L  2          4453.      10     44531.
##  9 standard     0.2 mg/L  3          4416.      10     44164.
## 10 standard     0.3 mg/L  1          6235.      10     62352.
## 11 standard     0.3 mg/L  2          6207.      10     62074.
## 12 standard     0.3 mg/L  3          6267.      10     62666.
## 13 standard     0.4 mg/L  1          8173.      10     81731.
## 14 standard     0.4 mg/L  2          8141.      10     81412.
## 15 standard     0.4 mg/L  3          8106.      10     81062.
```

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.

8.4.1 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:

function.	definition
+	additon
-	subtraction
*	multiplication
/	division
^	exponent; to the power off...
log()	returns the specified base-log; see also log10() and log2()

8.5 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 4620.  2475.
```

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` arg
```

```
## # A tibble: 5 x 4
## # Groups:   type [2]
##   type     conc_Na  mean stdDev
##   <chr>      <dbl> <dbl>  <dbl>
## 1 blank        0   1349.   45.9
## 2 standard     0.1 2933.   12.5
## 3 standard     0.2 4439.   19.5
## 4 standard     0.3 6236.   29.6
## 5 standard     0.4 8140.   33.4
```

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.

8.5.1 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.

8.6 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: 1349.4489"
```

```
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 1584.   12.5
## 2     0.2 3089.   19.5
## 3     0.3 4887.   29.6
## 4     0.4 6791.   33.4
```

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 "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: 1349.4489"
```

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 1584.   12.5
## 2     0.2 3089.   19.5
## 3     0.3 4887.   29.6
## 4     0.4 6791.   33.4
```

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.

8.6.1 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.

8.7 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.

Chapter 9

Programming with R

Programming is writing instructions that tell the computer what to do. Like most things, learning a little goes a long way. And like most things, it's easy to lose the forest for the trees. That's why we won't focus too much on programming (after all you're chemist not computer scientist) but we will introduce a few simple yet incredibly powerful elements of programming to help you along with your data science quest.

We'll point to several sources for further reading on functions at the end of this chapter.

9.1 Functions

Functions allow you to write general purpose code to automate common tasks. They're a great way to decrease repetition and make your code more legible and reproducible. To crease a function in R you only need `function()`:

```
funSum <- function(x,y){
  z <- x + y
  paste("The sum of", x, "+", y, "is", z, sep =" ")
}

funSum(1, 3)
```

```
## [1] "The sum of 1 + 3 is 4"
```

```
funSum("yes",3)
```

```
## Error in x + y: non-numeric argument to binary operator
```

What we've done is create a function called `funSum` which takes two numeric inputs `x` and `y`, sums the two into `z` and paste an output telling us the sum. A couple of things to note:

- We need to *explicitly* state which arguments are function will take; in this example they are `x` and `y`. Whatever we pass to `x` or `y` will be carried into the function.
- We can't sum non-numeric values, so R returns an error in the second instance
- Functions create their own environment, therefore *any variable* created inside a function only exists inside the function.
  - In the above example, `x`, `y`, and `z` only exist inside the function.
- R automatically returns whichever variable is on the last line of the body of the function; but you can explicitly ask for an output using `return()`

Let's take a look at a more practical function, something that you might actually use. In mass spectrometry, a gauge of accuracy is the *mass error*, a measure of the difference between the observed and theoretical masses, and is reported in parts-per-million (ppm). The formula for calculating mass error is:

$$Mass\ error\ (ppm) = \frac{|mass_{theoretical} - mass_{experimental}|}{mass_{theorical}} \times 10^6$$

The formula is simple enough, but you may need to calculate any number of mass errors, so it behooves us to compose a quick formula to simplify our workload:

```
ppmMS <- function(theoMZ, expMZ){

  ppm <- abs(theoMZ - expMZ)/theoMZ * 1e6
  ppm
}

# Theoretical mass = 1479.63 m/z
# experimental mass = 1480.10 m/z
ppmMS(theoMZ = 1479.63, expMZ = 1480.10)
```

```
## [1] 317.647
```

Pretty useful if you're manually checking something, but we can also use our functions into the pipe to help our data transformation:

```
# Example data
masses <- data.frame("theo" = c(1479.63, 1479.63, 1479.63),
                     "exp" = c(1478.63, 1479.63, 1480.10))

masses %>%
  mutate(massError = ppmMS(theo, exp))
```

```
##      theo     exp massError
## 1 1479.63 1478.63  675.8446
## 2 1479.63 1479.63    0.0000
## 3 1479.63 1480.10  317.6470
```

This last part is critical as *functions make your code more legible*. We can clearly read that the code above is calculating the mass error between the theoretical and experimentally observed masses. This might not be as apparent if we put in a complex mathematical formula in the middle of our pipe.

9.2 Conditional arguments

Are 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.

9.2.1 Piping conditional statements

You can already see the potential for simply 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 having an intensity of 0; 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.

9.3 When to use functions

A good rule when coding is **Don't Repeat yourself!**. In practice, this means don't copy and paste blocks of code to multiple parts of your script. It's more difficult to read (more lines), and if you identify an issue with one block, you'll need to hunt down all the other blocks to rectify the situation (you'll always miss something!). by using functions you'll reduce the number of lines of code, but you'll also only need to check one spot to rectify the issues.

9.4 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.

Chapter 10

Modelling

Modelling is basically math used to describe some type of system, and they are a forte of R, a language tailor made for statistical computing… Every model has assumptions, limitations, and all around tricky bits to working. We'll discuss model fitting and break down popular models you'll encounter in specific chapters in Section 3. For now, we'll introduce the `lm()` function for generalized linear models.

Linear models are the *trend lines* you used all the way back in *CHM135*. However, if you've been exposed to these, it's most likely via *Excel*'s 'add trend line' option. While `lm()` works much the same *mathematically*, unlike *Excel*, R returns *allllll* of the model outputs. Correspondingly, it's easy to get lost between juggling R code, the endless model outputs, and keeping yourself grounded in the real science you're attempting to model.

So let's take our `lm()` function at face value and learn *how* to model in R. Again we'll touch up the details later on, but for now let's import the FAES calibration results we saw in Transform: dplyr and data manipulation. As we've already seen, our data is composed of four standards and a blank analyzed in triplicate. Since we're focusing on modelling, *we'll treat the blank as a standard in our model fitting*:

```
# Importing using tips from Import chapter

FAES <- read_csv(file = "data/FAESdata.csv") %>%
  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) %>%
  mutate(type = "standard")

DT::datatable(FAES)
```

Note model is a general term, in this situation we'll be calculating a **calibration curve**. All calibration curves are models, but not all models are calibration curves.

10.1 Base R Linear Model

R's base `lm()` function for linear regression is excellent, but it's outputs have some messy quirks. It's easier to show that, so let's calculate the linear relationship between the `signal` as a function of `conc_Na`:

```
lm_fit <- lm(signal ~ conc_Na, data = FAES)
lm_fit
```

```
##
## Call:
## lm(formula = signal ~ conc_Na, data = FAES)
##
## Coefficients:
## (Intercept)      conc_Na
##        1243        16885
```

Reading the code above (recall that we're reading it from *right to left* because it's base R):

1. We're taking the FAES data
2. We're comparing `signal` (the dependent, y, variable) to `conc_Na` (the independent, x, variable) via the tilde `~`. The way to read this is: *"Signal depends on concentration"*.
3. We're comparing these two variables using the `lm()` function for generalized linear models.
4. All of the model outputs are stored in the `lm_fit` variable.

As we can see, the model outputs are pretty brief and not more more than *Excel*'s outputs. We can use `summary()` to extract more information to better understand our model:

```
summary(lm_fit)
```

```
##
## Call:
## lm(formula = signal ~ conc_Na, data = FAES)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -203.091  -86.731   -3.761  107.837  176.562
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1242.57      58.05   21.41 1.61e-11 ***
## conc_Na     16884.82     236.98   71.25  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 129.8 on 13 degrees of freedom
## Multiple R-squared:  0.9974, Adjusted R-squared:  0.9972
## F-statistic:  5077 on 1 and 13 DF,  p-value: < 2.2e-16
```

Now we have a lot more information from our model (don't worry about what everything means, it's discussed further in Section 3. For now, understand that it's a hot mess.

10.2 Cleaning up model ouputs

`summary()` provides a decent overview of our model's performance, but the outputs are difficult to work with. Let's turn to the `broom()` package to clean up our model outputs.

```
library(broom)

calCurve <- FAES %>%
  group_by(type) %>%
  nest() %>%
  mutate(fit = map(data, ~lm(signal ~ conc_Na, data = .x)),
         tidied = map(fit, tidy),
         glanced = map(fit, glance)
         )
calCurve
```

```
## # A tibble: 1 x 5
```

```
## # Groups:   type [1]
##   type     data              fit    tidied           glanced
##   <chr>    <list>            <list> <list>           <list>
## 1 standard <tibble [15 x 4]> <lm>   <tibble [2 x 5]> <tibble [1 x 12]>
```

Things look a bit more complicated than our earlier example, so let's break down our code line by line:

1. We're taking the `FAES` dataset that we created earlier.
2. `group_by(type` groups all rows by `type`, in this situation we have only one type: `standard`.
3. `nest()` collapses everything other than the `type` column into smaller dataframes. In this situation, all other information is stored as a `tibble` under the `data` column; this is the data used to calculate the linear model.
4. Withing the `mutate` function, we've created three columns: `fit`, `tidied` and `glanced`.

And it's the the `fit`, `tidied` and `glanced` that contains out cleaned up model outputs. `fit` contains the raw output from the linear regression model for `signal` as a function of `conc_Na` using the `lm()` function. The output is in the form of a list, similar to what `summary()` gave us above. Again, this is exceptionally messy, hence why we used the `tidy()`, and `glance()` function from the `broom` package . `map()` just means we're applying the function `tidy()` to the individual output list created by `lm()` and stored in the `fit` column. Note that the `tidy()` and `glanced()` outputs are tibbles. So we now have a tibble containing specific model output values (i.e. `(Intercept)`), lists (i.e. `fit`), and tibbles (`tidied`). This is known as **nested data*. We're no longer in Kansas anymore…

Anyways, let's take a look at our model results. The `glanced` tibble contains "…a concise one-row summary of the model. This typically contains values such as R^2, adjusted R^2, and residual standard error that are computed once for the entire mode"[1] Because the data is nested, we'll need to use `unnest()` to flatten it back out into regular columns:

```
calCurve %>%
  unnest(glanced)
```

```
## # A tibble: 1 x 16
## # Groups:   type [1]
##   type   data     fit   tidied  r.squared adj.r.squared sigma statistic  p.value
##   <chr>  <list>   <lis> <list>      <dbl>         <dbl> <dbl>     <dbl>    <dbl>
## 1 stand~ <tibble~ <lm>  <tibbl~     0.997         0.997  130.     5077. 3.05e-18
## # ... with 7 more variables: df <dbl>, logLik <dbl>, AIC <dbl>, BIC <dbl>,
## #   deviance <dbl>, df.residual <int>, nobs <int>
```

[1] From the *broom* package vignette.

What you see here is a bit more than what you'd get from *Excel*'s 'line-of-best fit' output. See the section on *Modelling* for a better breakdown of what everything means. But for now, we can see that our `r.squared` of each calibration curve is pretty good, and the `p.value` indicates each model is significant. the `adj.r.squared` is the same as `r.squared` in this situation. This is because `r.squared` will always increase if we add more exploratory variables to our model; the `adj.r.squared` accounts for the number of exploratory variables used in the model. However, in our case we only have one exploratory variable, hence they're the same.

But what about the slope and the intercept? After all, that's what we need to calculate the concentration in our unknowns. Let's take a look at `tidied` from the `tidy()` function "…which constructs a tibble that summarizes the model's statistical findings. This includes coefficients and p-values for each term in a regression…"[2]

```
# storing because we'll use it later on.

tidied <- calCurve %>%
  unnest(tidied)

tidied
```

```
## # A tibble: 2 x 9
## # Groups:   type [1]
##   type   data      fit    term   estimate std.error statistic  p.value glanced
##   <chr>  <list>    <list> <chr>     <dbl>     <dbl>     <dbl>    <dbl> <list>
## 1 stand~ <tibble ~ <lm>   (Inte~    1243.      58.0      21.4 1.61e-11 <tibble ~
## 2 stand~ <tibble ~ <lm>   conc_~   16885.     237.       71.3 3.05e-18 <tibble ~
```

Again, a lot more to unpack compared to *Excel*. That's because the `lm()` function in R calculates a generalized linear model. `lm()` performs a linear regression model, which we normally think of as an equation of the form $y = mx+b$. But, regression models can be expanded to account for multiple variables (hence *multiple linear regression*) of the form

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 ... \beta_p x_p$$

]

where,

- y = dependent variable
- x = exploratory variable; there's no limit how many you can input

[2] From the *broom* package vignette.

- B_0 = y-intercept (constant term)
- B_p = slope coefficient for each explanatory variable

In our situation, we only have one input variable for our model (`conc`), so the above formula collapses down to $y = \beta_0 + \beta_1 x_1$. So looking at our results above, each row corresponds to a model parameter. For each modelling parameter, we're provided an estimate of it's numerical value (`estimate`, the values we'll use to calculate concentration). The other parameters are useful to understand but not necessary at this point (again, check out the *Modelling* section).

And we can extract these values to use in subsequent calculations:

```
intercept <- as.numeric(tidied[1,5])
slope <- as.numeric(tidied[2,5])

paste("The equation of our calibration curve is: y = ", slope, "x + ", intercept, sep="
```

```
## [1] "The equation of our calibration curve is: y = 16884.8167x + 1242.56646666666"
```

10.3 Further reading

The theory and use of these models are explored in greater details in Section 3. Please read up on it for an understanding of the model outputs and how to use them in your analysis. As well, see the section on modelling in *R for Data Science*.

https://www.newyorker.com/magazine/2021/06/21/when-graphs-are-a-matter-of-life-and-death

Chapter 11

Visualizations

- theory undergirding ggplot (focus on geom_point)
  - Geoms
  - aesL x,y, ccolour, shape, size, alpha
  - arranging plots in a grid (grid.arrange) and facets…
- How to plot
  - labels
  - scales
  - annotations
  - themes
- building a plot layer by layer
- saving/exporting plots.

Visualizations have always been an important part of data science and chemsitry. Good graphics illuminate trends and pattersn you may have otherwise missed and allow us to quickly inspect thousands of values. R via the `ggplot2` package is one of, if not the premier, data visualization langauge available. This chapter will formally introduce the `ggplot2` package, explain a bit of the logic undergirding it's operation, and give you some quick examples of how it works. Section 3 will delve deeper into specific visulizations you'll use and encounter in your studies.

`ggplot2` is loaded by default with the `tidyverse` suite of pakacges. 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")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   wavenumber = col_double(),
##   EPDM = col_double(),
##   Polystyrene = col_double(),
##   Polyethylene = col_double(),
##   `Sample: Shopping bag` = col_double()
## )
```

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

```
## Warning in instance$preRenderHook(instance): It seems your data is too big
## for client-side DataTables. You may consider server-side processing: https://
## rstudio.github.io/DT/server.html
```

11.1 Building plots ups

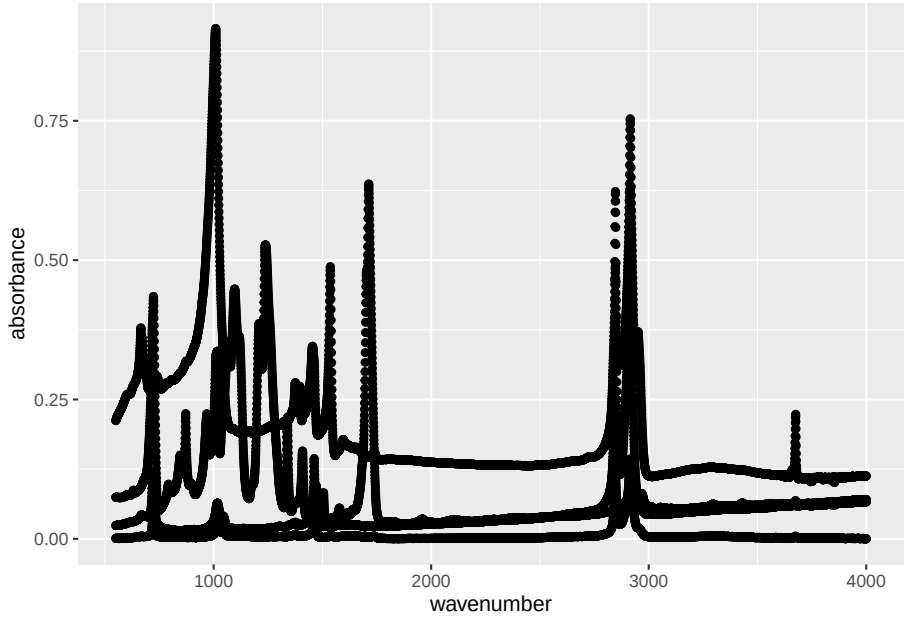
The `gg` in `ggplot2` stands for the `grammar of graphics` (Wickham 2010), and it's a way to break down graphcis (plots) into small pieces that can be discussed (hence grammar). WE'll take a look at this grammar via `geoms` (what kind of plot), `aes` (aesthetic choices), etc. For now, understand that 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 unerlying data.

11.2 Basic plotting

`ggplot2` uses `geoms` to specificyt what type of plot to create. Different plots are used to convey different meanings and have different strenghts and weakness. We'll explore these more in Section 3, but for now we'll focus on `geom_point()`, which simply plots data as poits on an [x,y] coordinate. In otherwords, a scatter plot.

Let's plot our tided `atr_long` data:

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



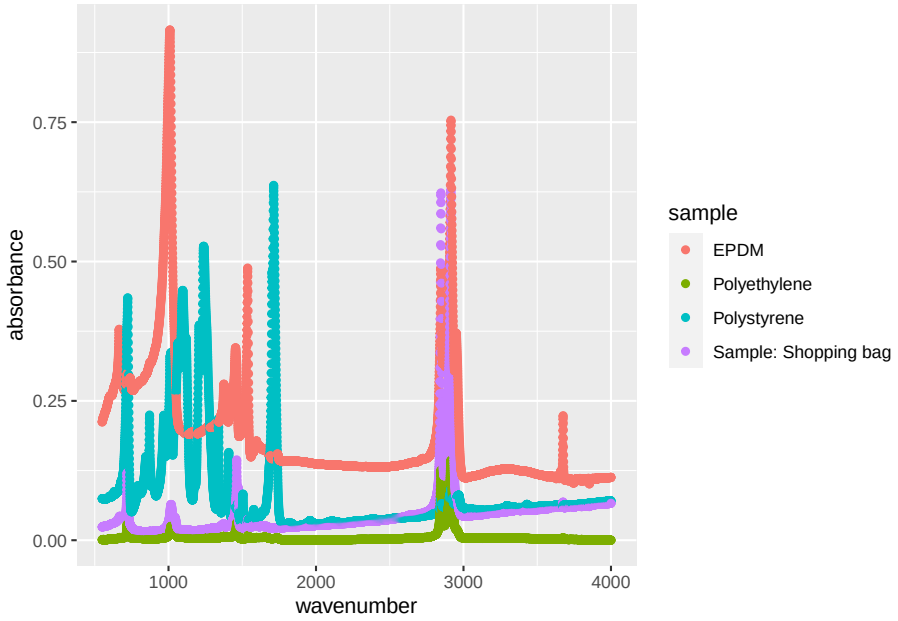
Let's ignore the plot for now and look at our code down:

1. `ggplot()` intilaizes a *ggplot object*, basically an empty plot. To this we've specified out data set (`data = atr_long`).
2. We then specified our *aesthetic mappings* via `aes()`. Here we'll pass information for how we want the plot to look. 3.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`).
3. 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 absorbance measurements from our ATR analysis. We can clearly see the spectra of the different plastics in our data, however they're all colours the same. This is because we've only speciies the x and y values. As far as `ggplot()` is concerned, these are the only values that mattter, but we know different.

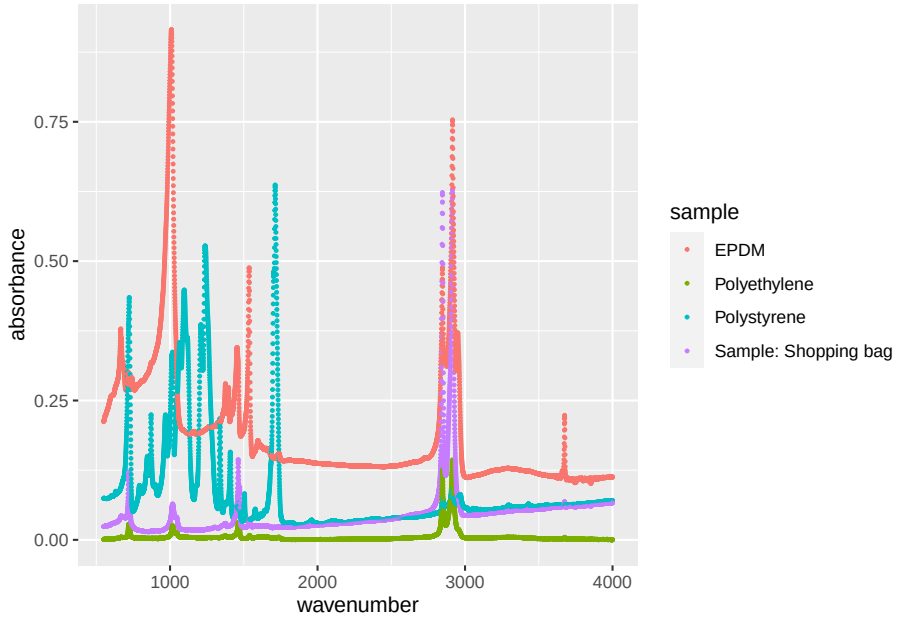
Fortuanely you can pass multiple variables to different `aes()` options to enahce our plot. For instance, we can pass the `sample` variable, which specifies which sample a spectrum originates from, to the `colour` option:

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



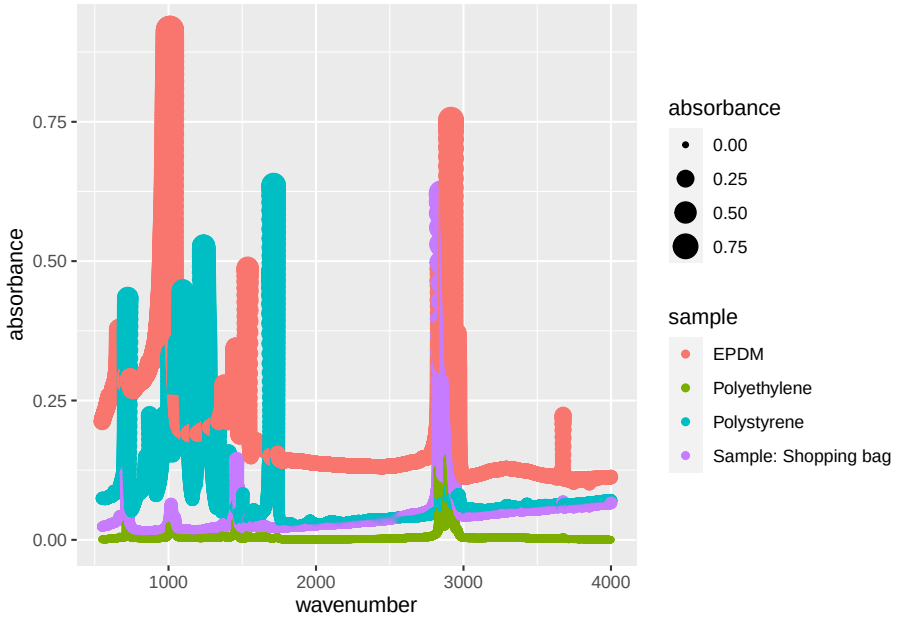
Now we have a legend which clearly specifies which points are assosciated with each sample. But now the points are too large, potentially masking certain peaks. We can adjust the size of each point as follows:

```
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()
```

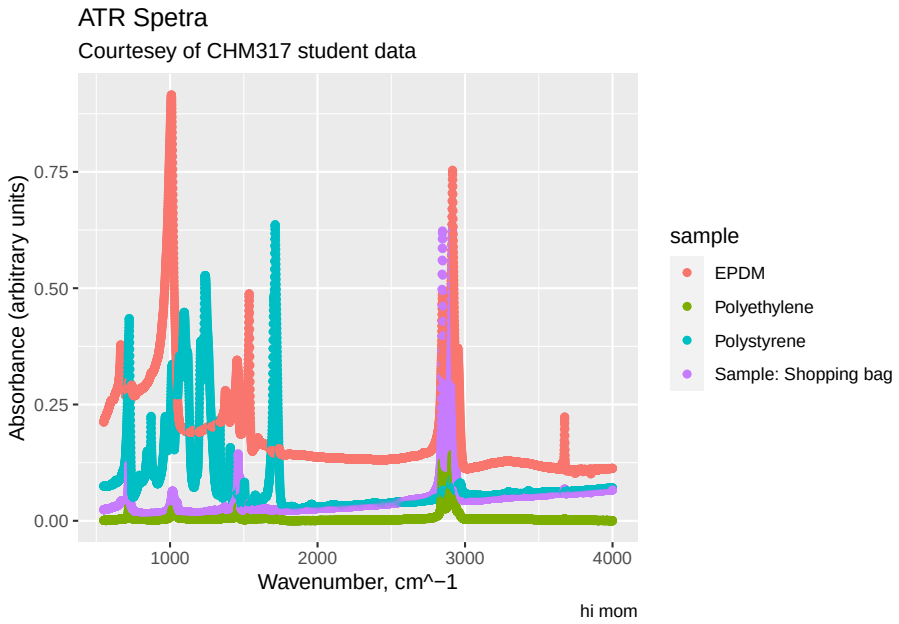


Sometimes this makes sense (i.e. a *bubble chart*) but for our example, having the size of the poitns increase as the absorbance increases doesn't provide any new information (it actually clutters our plot).

11.2.1 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 = "Courtesey of CHM317 student data",
       x = "Wavenumber, cm^-1",
       y = "Absorbance (arbitrary units)",
       caption = "hi mom")
```



11.3 Further reading

There's no shortage of options when playing around with `ggplot` and these will be explored in greater detail in Section 3 (including *when* you should and shouldn't do things).

Chapter 12

Communication

- R markdown
- slides
- exporting
- tips on automating Rmd generation?

Wickham, Hadley. 2010. "A Layered Grammar of Graphics." *Journal of Computational and Graphical Statistics* 19 (1): 328. https://doi.org/10.1198/jcgs.2009.07098.

———. 2014. "Tidy Data." *Journal of Statistical Software* 59 (1): 1–23. https://doi.org/10.18637/jss.v059.i10.