R for Environmental Chemists

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

2021-06-07

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
- 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 March 2021 is v4.0.5. 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/Prof 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… and they've complained about it and someone else has solved it! An oft 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*)[https://r4ds.had.co.nz/index.html], which inspired this entire project.
- All else fails,holler at your TA/profs.

Getting Setup in R

Chapter 2

Running R Code

- Where to run code in RStudio (script vs. console)
- Basic coding building blocks
- Variable assignment (howdy <- “Howdy world”)
- Importing packages

- Calling functions
- R Studio basics
  - Environment windows to inspect workspace data/variables (in conjunction with above); this needs to be explicitly shown as kids never realize they can inspect their data à la Excel in RStudio
  - Plot window
  - Files window (expanded on in R Workflow chapter)
  - Tearable tabs; again, kids don’t realize you can tear off a tab into a new window so you can see your data and code side-by-side (click and drag tab to tear off).

  - Customization (i.e. themes for dark-mode)
- Brief style guide
  - Packages loaded at the top.
  - How to comment code (#I’m cold and there are wolves after me) -Suggestion for variable names (good_name vs. thisNameWillWork_butIsAwfulToType); including forbidden names -Script headers (i.e. # heading1, ## subheading); These are picked-up by RStudio and displayed in the Document Outline box allowing easy navigation of long scripts.

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 Creating an RStudio Project

When you open RStudio for the first time, this is what you should see:

The version of R you just installed should appear in the main window here:

(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.)

Before doing anything else, let's create an **R Project**. This will establish a default folder for RStudio and bundle together all your code files in one place. Go to *File->New Project*. Click *New Directory*, then *New Project*. Next, you'll be asked to choose a subdirectory name and location. The name can be whatever you want, but we highly recommend that you create the directory in the same parent folder as your data. Click *Create Project*, and you should now see your chosen file path displayed in the bottom-right window:

You are now ready to use R! If you're in a hurry, you can skip straight to Chapter 2: Organizing Your Data, but we recommend you finish this chapter to familiarize yourself with RStudio's layout.

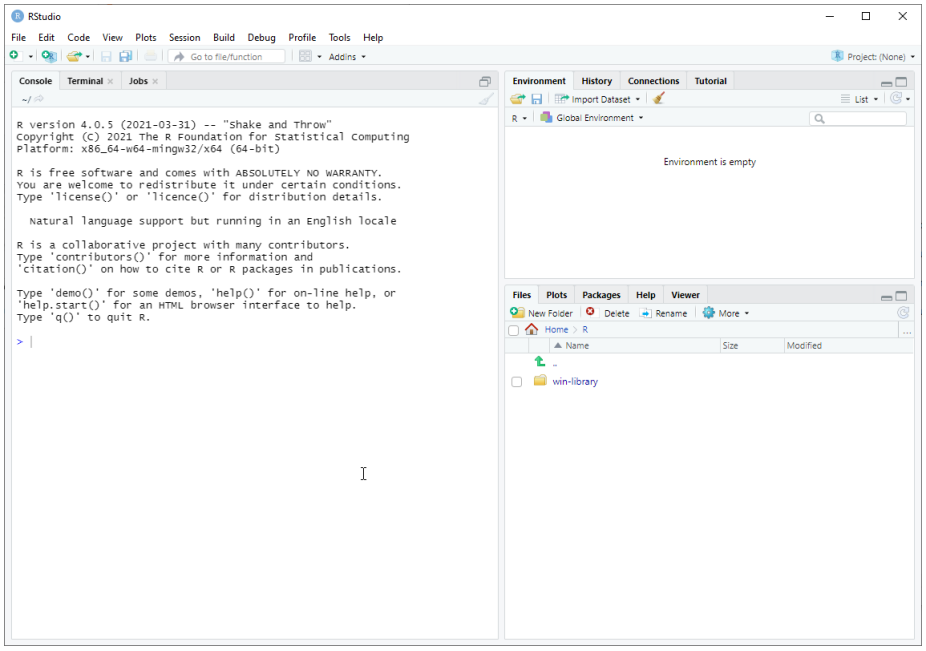

Figure 3.1: RStudio Default View

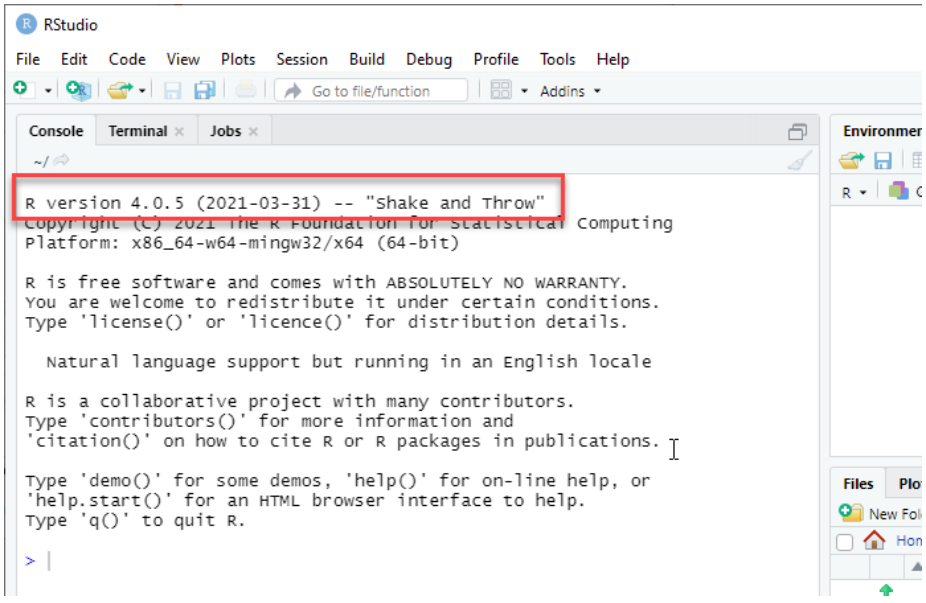

Figure 3.2: RStudio - R Version

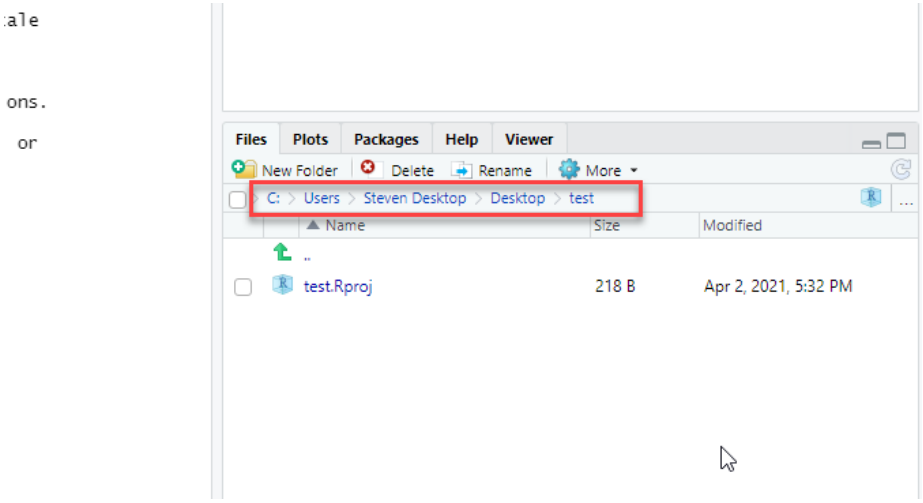

Figure 3.3: RStudio Project Folder

3.2 Navigating RStudio

RStudio has 4 main windows, 3 of which should currently be visible. To see the fourth window, go to *File->New File->R Script*. You should now be able to see all 4 main windows:

3.2.1 R Environment

This window is the least important for our purposes. In brief, it will list all variables, packages, and functions which you have run since opening RStudio.

3.2.2 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.

3.2.3 Console

This is where you can type and run code directly, but you will mainly be using the Scripts window to run code (see below). Your primary use of the console will be for installing packages (more on that later). Another thing that the Console window is useful for is quick arithmetic. Try typing "2+2" in the console, then hit Enter.

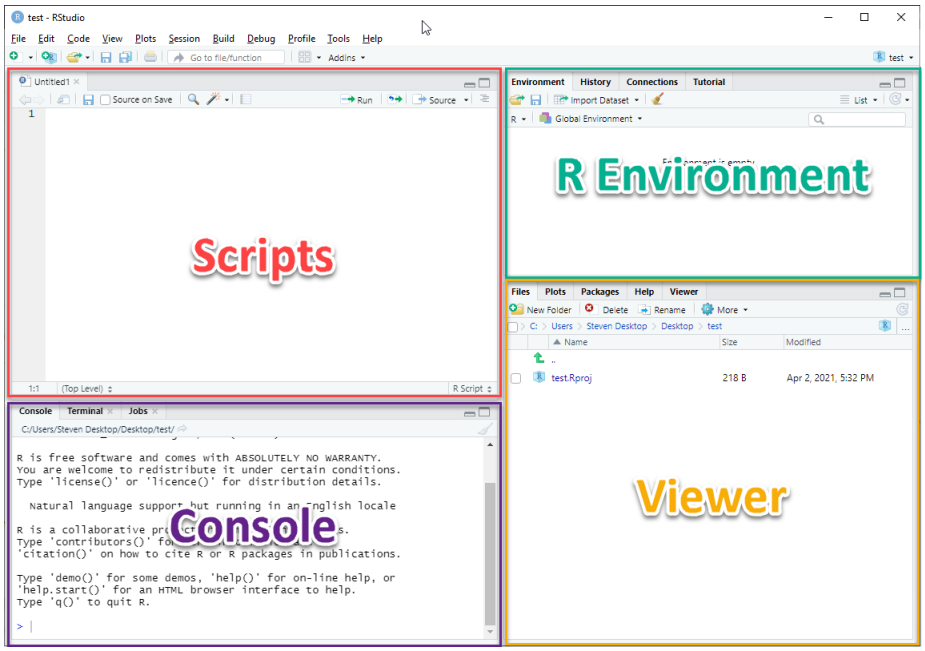

Figure 3.4: RStudio Quadrants

```
2+2
```

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

3.2.4 Scripts

R code files are called scripts, and are saved as *.R files. Whenever you copy code blocks from this website, paste them into the Scripts window. You can then run specific lines by highlighting them and pressing Ctrl+Enter (Cmd+Enter on Mac), or clicking the "Run" button in the top right of this window.

3.3 Dark Mode

Lastly, some of you may be interested in how to set RStudio to Dark Mode. Simply go to *Tools->Global Options*, then click "Appearance" on the left. Then select your preferred Editor Theme from the list. (My personal preference is "Tomorrow Night Bright")

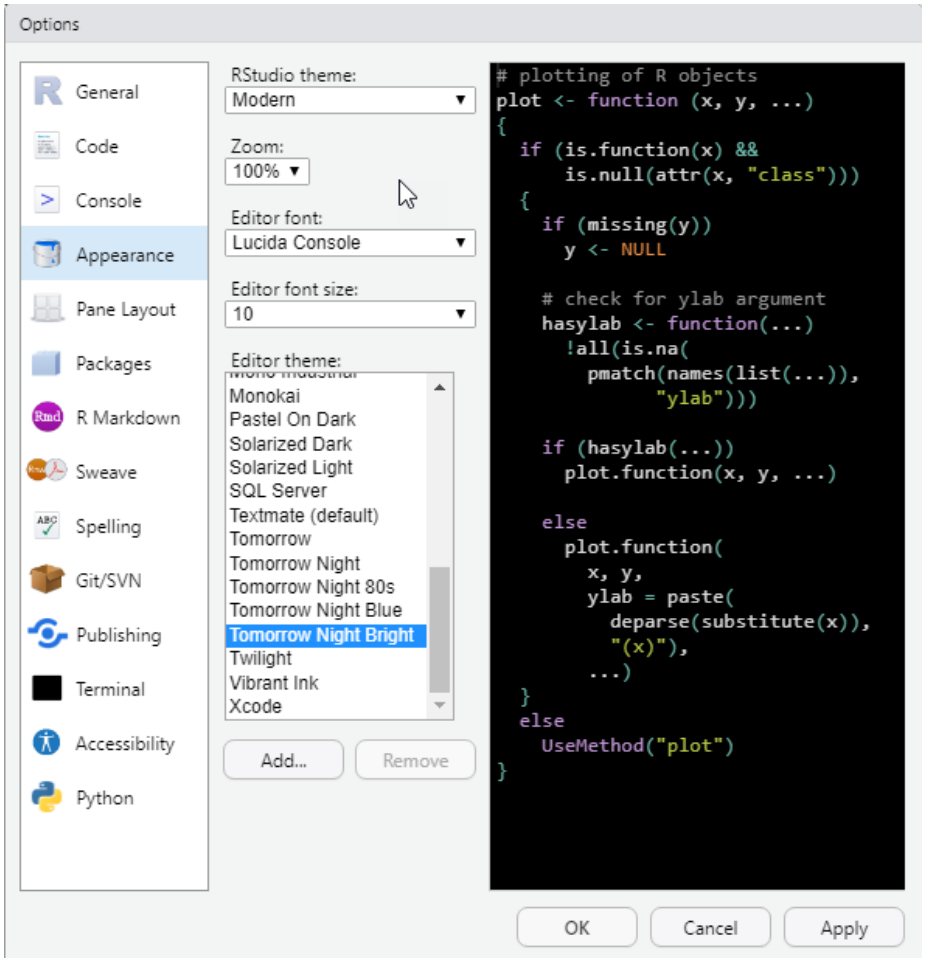

Figure 3.5: RStudio Dark Themes

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()
```

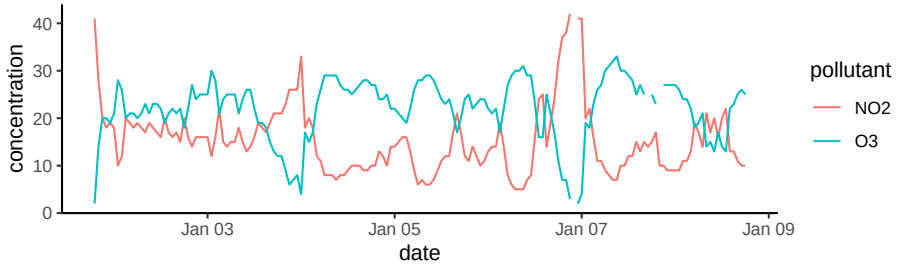

Figure 4.1: Time series of 2018 ambient atmospheric O_3 and NO_2 concentrations (ppb) in downtown Toronto

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.

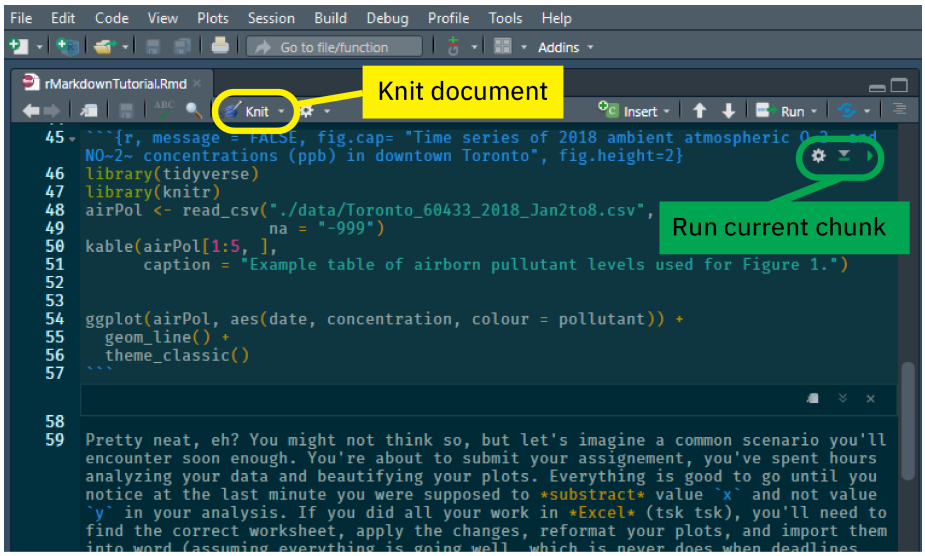

Figure 4.2: How this document, the one you're reading, appeared in RStudio; to see the final results scroll up to Figure 1. Note the "knit" and "run current chunk" buttons.

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:

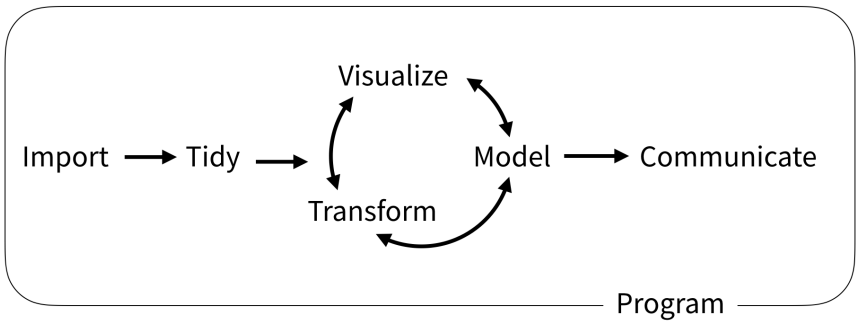

Figure 5.1: Data science workflow describes by Wickham and Grolemund; image from *R for Data Science*, Wickham and Grolemund (2021)

- **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.

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                115.
## 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  115.079
## 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 115.
## 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`.