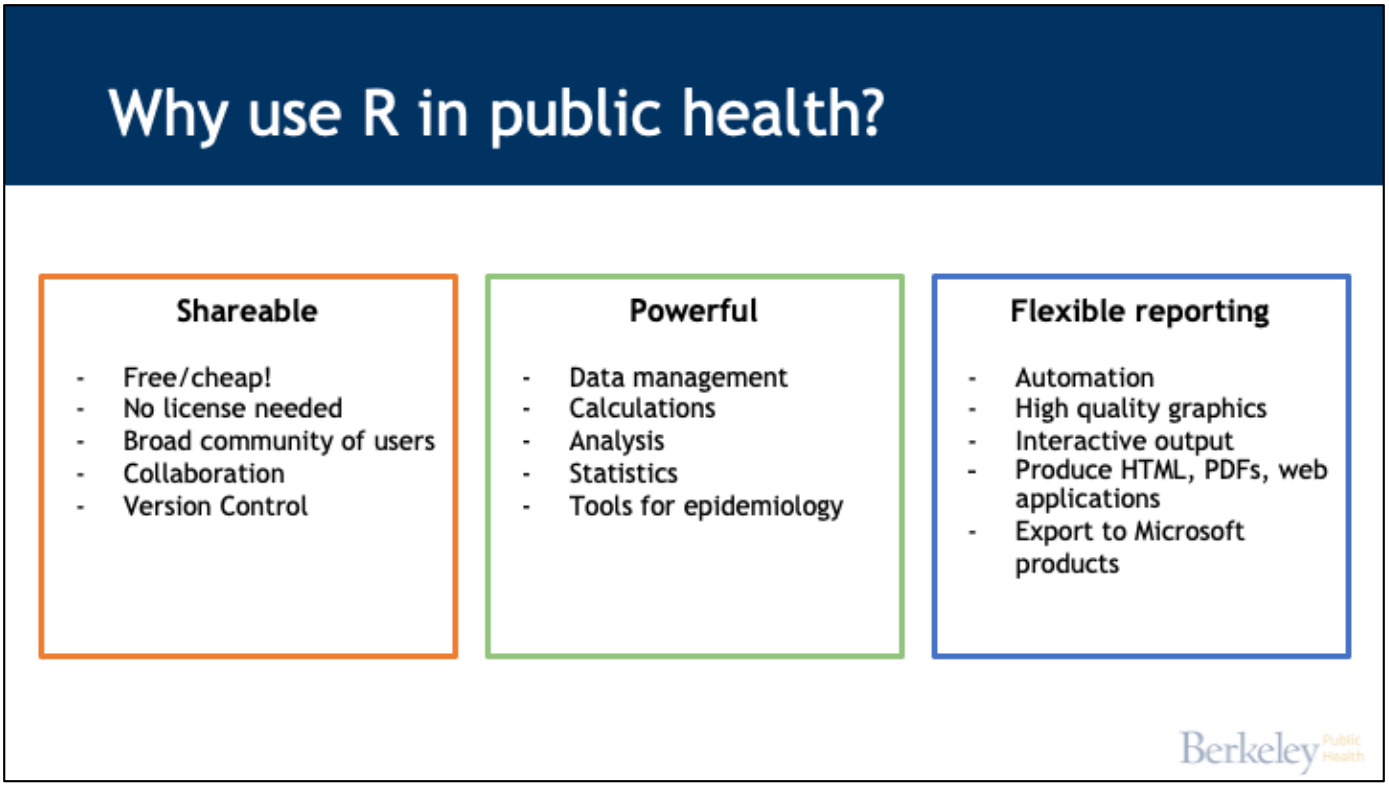
PHW251 Week 1 Reader

Topic 1: Why Learn R?

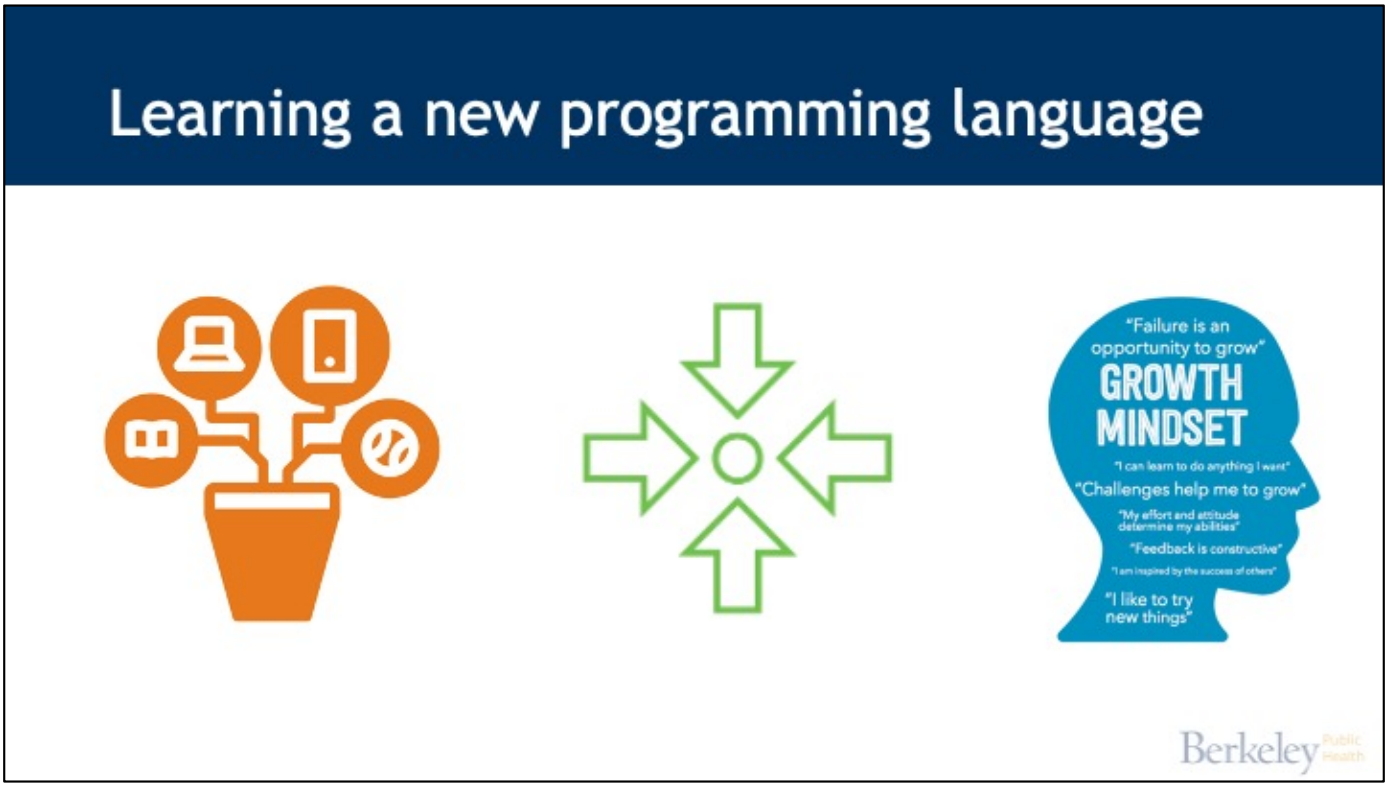
Topic 2: Getting Started with R

Lecture 1: Why Learn R?



Hi, everyone. Before we get started on the main content for this week, which is an introduction to R and getting started using R. I wanted to take a couple of minutes and talk about why we think using R is so important for public health and offer some advice and thoughts we have as you start this course to learn or re equate yourself with a programming language. Why do we use R in public health? Why do we think it's important to learn? And why are we excited to be here to teach it? First of all, R is designed to be easily sharable. It's open source, which means that it's available for anyone to download and use. It also means that anyone can create content that others can use within R, making a whole bunch of different tools available for use in R. Being open source also means that it's either free or very cheap and that no license is needed, and that's different than a lot of the other programming tools that we commonly use in public health like SAS or Stata. There's also a broad community of users, not only from all over the world, but in all settings within the public, private non-profit and academic sectors. Our users are doing data analysis, data science and data reporting using R, which is really cool when you get to see this huge community of different folks that are doing things similar as you are, or even better things you want to be doing or want to try out. There are also ways that promote collaboration and version control in R, which helps with working in groups and also just helps to keep track of changes you made to your code over time. We'll talk a lot in this course about being kind to your future self by making sure that you're keeping track of

versions over time. The next feature of R is that it's very powerful. It provides tools for data management in managing and cleaning very large datasets. It also allows for high powered calculations like what you would do with a calculator and also high powered calculations on datasets, which is really helpful. It allows for robust analysis, statistical analysis, and offers some specific tools for public health and epidemiology. The third feature we want to highlight is that R offers a lot of flexibility in ways for you to report your analytic findings and also gives you a lot of options for automating data wrangling and reporting tasks. The graphics that are used in R are high quality, especially compared to some of the other analytic tools out there and some of those outputs and graphics programmed to be interactive. That means if you have to do a report on 15 different sites, instead of producing 15 different reports, you could just program one report and let the end user choose which site they want to look at. There's also the ability to produce HTML versions, PDFs, web applications straight from R, in addition to being able to export reports in different Microsoft products. Overall, we feel like R is very sharable, approachable, powerful, and a flexible tool that's becoming used more and more in public health, and that really makes someone marketable having those skills as they enter or continue on in their career.



Now let's shift a little bit and talk about learning a new programming language. When I've been at different points in my career or my education or have had an opportunity or a need to learn a new programming language. I have absolutely felt daunted and strained under the challenge, but it can also feel pretty exciting. When you get that spark of joy because you've just figured something out, that's what makes it feel all worthwhile. That's what we're chasing. We wanted to offer a few different thoughts on how to approach learning R or a new programming language in general. First, I really want to encourage everyone to make sure to use all of the resources that are available to you. Programming doesn't need to be or it isn't meant to be done in a tunnel or a vacuum with no help from external resources. You have your peers who are also taking this course with you. This is a learning community, and you should take advantage of that. You also have support from the teaching team or instructors, and there's the Internet, Google, Stack Overflow, Open AI, all of these resources. The R and R Studio posit community also have a ton of resources. We encourage you to approach us or your peers with questions and also try Googling or using other search tools to identify solutions to the problems you're running into. Another piece of programming that I think is really important, is this idea that there isn't always one way to do something. There might be a ton of different ways, and each of them have different pros and cons for doing the one thing you need to do. I think programming allows for a fair bit of creativity and allows for a user programmer to come up with a method or a system that works best for them. That might look different from their peer but as long as their code is doing what they expect it to, and it makes sense and they're getting the end result that they desire, then that's really all we need. I think we

can learn from each other and I think for me, as I've learned more over the years, there's times when I'll go back and look at the old code I've written and realize I've learned a lot of new things and can go back and make my old code more efficient. I think again, something really exciting about that. It might also feel frustrating that there's no one clear, easy, right answer to anything, because there's always multiple ways of doing it but I think we're hoping to offer some tools for you to build a base of knowledge in using R, and then you can decide how to approach it in the future. The last piece I want to mention is just to really encourage everyone to approach this course with a growth mindset. If you haven't heard that term before, it's a term developed by Carol Dweck. The idea is the comparison of the growth mindset and the fixed mindset, where the growth mindset is the idea of approaching a new challenge is really something that will help you grow. You can do anything. You can learn anything. No matter what happens, at the end of the semester, you will have more programming knowledge than you had before. And the fixed mindset is the idea that you're stuck. You can't learn things. You're never going to be good enough at this and to just be overwhelmed by the challenge. We just encourage you to lean towards the side of the growth mindset. I think this will make this course more enjoyable and also likely make you more successful. I think related to this is just the idea that programming is really about the path and the journey, but it's not about the end result exactly. It's about learning how to do things, trying out different approaches and developing a process for learning and trying out new approaches and learning how to troubleshoot, debug your own code in addition to learning the specific syntax. Just keep that in mind that everything we do and learn this semester is contributing to this base knowledge that you have about programming and about programming in R and no matter what happens, you will know more about R and be able to use those new skills by the end of the course. We just encourage you to take that attitude as we move forward. Remember that we're here for you. You're not alone in this course, as we try to altogether build some new skills and apply those skills to public health problems. We're excited that you chose to join this course, and let's get started.

Topic 2: Getting Started with R

Lecture 2.1: Key Common Skills

Getting started with R

What you should already know (from Week 0):

- How to access the RStudio environment through Datahub

What is included in this lecture:

- Key common skills
  - Assigning values
  - Naming objects
  - Running Code
  - Commenting Code
- Data types
- Data Structures/Objects
  - Vectors
  - Matrices
  - Lists
  - Dataframes (briefly)
  - Creating objects using functions
- Calculations and Comparisons

In this video, we're going to talk about getting started with R. What you should already know coming into this is how to access the R Studio environment through data hub, through your desktop or some other means. What we're going to talk about in this lecture are some key common skills, which include assigning values, naming objects, running code, and commenting code. Then we'll talk about data types, data structures and objects. That includes vectors, matrices, lists. We'll talk about data frames briefly, although that will mostly be covered in the rest of the course and then creating objects using functions. Then we'll round things out with calculations and comparisons. This is just a light overview on some major concepts that will cover in this course. We will get into much more depth during the rest of the course, but this is just sort of an introduction.

Key Common Skills

Assigning a value to an object

To assign an object a value (or values), use "<-": Object <- value(s)

```
#use "<-" as the operator to assign the value of 5 to the object named "x"
x <- 5
#return the value of x
x
```

```
[1] 5
```

```
#assign and return
(x <- 5)
```

```
[1] 5
```

So first, we'll talk about key common skills and the first of those is assigning a value to an object. So to assign an object a value in R, we use this combination of symbols. This is the less than symbol here, followed by a dash. What this means, typically, what you would say is x equal five. But what we try to say in R is that we're assigning the value of 5 to x. So that's why it looks like a pointer. We're saying that we're assigning a value to an object, and we're naming that object x. So if we run this, we can see down in the console that we have run that line, and we have assigned x to the value of five, and we can test that by running the code calling x, and when we call x, we're referring to that object, and because we have assigned that value, it returns the value of five.

By putting parentheses around this line, you can both assign the value, so we can basically do these two steps as one. You're essentially creating a very simple function and you're just both assigning that value and returning it. You can see that both of those things happened in the console. So then moving on to naming objects.

Naming objects

- Names cannot start with numbers or symbols
- R is case sensitive!!
- Best practices:
  - Use lower case
  - Use underscores to separate words in names

There are a few rules that are unique to R. Names cannot start with numbers or symbols. They have to start with letters and in R, named objects are case sensitive. So it's not the same as in SAS. In SAS, you can call something with all capital letters like this. You could call something this value with the U and E capitalized and in SAS, that would be the same as SAS would recognize if you spelled it like this as the same object or spelled it with all caps as the same object. It does not differentiate as long as the letters are there in whatever case they are, then it's the same object. Whereas in R, those three things are all separate. You want to keep this in mind, especially when you're first beginning and you're trying to debug your code. You need to be aware of this because you might start a program and call it something like this in all lower case and then, not touch that object again until 300 lines later, and you might call it all capitals or put a capital letter in there somewhere, and R is not going to register that as being the same. It's going to say, well, this is a brand new thing. It's not assigned value, and so I'm going to error out.

So just be aware that R treats things as sensitive, and it's a little bit of something to get used to, especially if you're coming from a different programming language. But there are significant benefits, and you'll get used to it.

```r
#some users prefer what is called 'camelCase' which uses a capital letter
#to indicate a new word

camelCase <- "camel"
# CamelCase
camelCase
```

```
[1] "camel"
```

```r
#we recommend using all lowercase and underscores to separate words
snake_case <- "snake"
```

The other thing we talk about when we talk about naming is the conventions that can be used in order to name things. So you may have heard about things like camel case means basically that if you're naming something, you can name things with multiple words. The first letter should be lower case, and all subsequent words in the name should have the first letter as uppercase. So if you had something like Camel case here, you can see first letter is lower, then uppercase C and uppercase H, and it's called camel because these are like camels humps.

Another version of this is snake case. So snake cases all lower case letters and words are separated with an underscore. Words or numbers can also be part of that. So this would be an example of snake case, and this is what the code that you will see as part of the course. We have written all in snake case and we recommend in order to stay consistent at least during this course to also use snake case, but of course, you're welcome to do whichever you would like, it may be that you eventually go off and graduate and join up with a group that prefers some other standard and they're all perfectly fine. It is helpful to choose one because then when you're creating an object, you know the standard way of creating something, you all have to go back and look at how you've named things previously. So it helps to have that convention of naming things.

There's also a kebab case, which is basically just snake case except dashes instead of underscores, so it looks like a meat or vegetables on a kebab and there are a few others out there. These are the main three that we're aware of. You can just choose one and start using it.

Running Code

- To run one line:
  - Click within line or highlight line
  - Click “Run” (shortcut: ctrl + enter)
- To run several lines:
  - Highlight lines
  - Click “Run” (shortcut: ctrl + enter)

```
assign_this <- "this"

assign_that <- "that"
paste0(assign_this, " and ", assign_that)
```

```
[1] "this and that"
```

- To run directly in console:
  - Enter code directly into the console
  - Press Enter
  - **Good option when there is no need to save code

Running code. There are multiple ways of doing this. You can run one line by clicking within the line anywhere within the line. You can be in the middle, at the end or at the beginning, and you can just hit this run button up here, run selected line, and that will run that single line. You can also highlight it if you prefer, or I just typically put my cursor in there somewhere. So you can hit this run button or the other option that I prefer is you can just use the keyboard shortcut, which is control and enter. So you can put your cursor anywhere in a line, hit control enter, which I’ve just done on my keyboard, and it runs it.

You can also run multiple lines, very similarly, except to run multiple lines, you have to highlight both of those lines. You can’t just put your cursor there and likewise, you could either click the run button up here or hit Control Enter, and it will run both of those lines. When you’re creating either R markdown files or quarto files, which will be covered in the R markdown and quarto video.

You can also run things by running the current chunk, and that basically this gray area is the chunk, and that will just run everything in that chunk. But you can also do line by line, like I just explained.

And then the third way of running code is entering code directly in the console. So I could take this line here, copy it. I’m just going to type it and I’ll need this. So it’s exactly the same as running it on a sheet up here. It’s just running in the console directly. However, running in the console directly, it saves the code in the history, but it doesn’t save it like a document like this. So it’s a good option when there’s no need to save the code, if you just want to figure out how a function is going to work before you put it in your actual code, you could just run it in the console and ephemeral in that way.

Commenting Code

Use “#” to comment out code

Shortcut: Ctrl + Shift + C (Command + Shift + C on macOS)

**Comment multiple lines at a time by highlighting and using keyboard shortcut

```
# This is very simple commenting on a very simple example

assign_this <- "this"

assign_that <- "that"
paste0(assign_this, " and ", assign_that)
```

```
[1] "this and that"
```

```
assign_other <- ", or the other"
put_all_together <- paste0(assign_this, ", ", assign_that, assign_other)
put_all_together
```

```
[1] "this, that, or the other"
```

Next, I’ll talk about commenting code. In R, you use the pound or hash symbol to comment out code. You can also comment out full lines of code by highlighting them and then hitting Control Shift C or command shift plus C on the MacOS. What that does is comments all of the lines together, basically just puts a hash in front of all of those lines and then if I want to comment, I can just highlight those again and hit Control Shift C, and it will uncomment all of them.

Commenting best practices:

- Add comments to describe purpose of code or explain specific or complicated processes
- Err on the side of over-commenting
- While developing code, comment out (rather than delete) sections that you may need to revisit

Commenting best practices. Generally, it's good practice to describe the purpose of your code, or especially if something is really complicated that you're doing, you want to describe how you're approaching what each line of code is doing or what chunks of code are doing. So that for two purposes, you can remember if you write a code and you comment it and then don't see it again for three months, it's easier to get back to that space of what you were thinking at the time and what the code actually does.

Likewise, if you're working with a lot of colleagues or even one colleague, it's good to comment what you're doing so that they can understand what you're doing. We recommend erring on the side of over commenting, and then the other good use of commenting is if you are writing code and you want to try something different, you might come out the original lines of code that you had first written while you're trying something new out. If that doesn't work, then you can just delete the new way of doing it and uncomment the old way of doing it. That way, you're not deleting anything, and if you want to revert back to it, you're not stuck because it's still there.

Lecture 2.2: Data Types

Five Main Data types in R

Character values

Type	Examples
character	“ph”, “ucb”, “ucb ph”
numeric	290, 290.5
integer	290L (the L tells R to store this as an integer)
logical	TRUE (or T), FALSE (or F)
complex	1+4i (complex numbers with real and imaginary parts)

I’m going to shift now to the five main data types in R. Those are character, numeric, integer, logical, and complex. We’re going to focus on this course exclusively on the first four. We don’t really get into complex data types. Character data type is basically letters. It can be letters, numbers, and symbols, but they are treated as text. You put either two quotes around them or one quote around them, and they are text.

Numeric are numbers and there are multiple different forms of numbers, including ones with decimal points, and then integers are a special form that are basically just whole numbers. By including that L, it tells R to store this as an integer. Then the logical data type is binary. It’s either true or false. It must be written as all caps, the full word two or a full word false, or a T or a F.

- Indicated by quotation marks (“” or ‘’); best practice is to use double quotes
- Can contain spaces, characters, symbols, and numbers

```
#create a character value using double quotes
ch_double <- "dog"
#try using single quotes - this works, too
ch_single <- 'dog'

ch_double2 <- "turtle"
```

```
#the code below will not work, as R is looking for an object named "dog"
#(since there are no quotations around it)
# ch_no <- dog

#if we create an objected named "dog"
dog <- "puppy"

#then re-run the line with no quotes, the value of "ch_no" will take on the
#value of the object "dog" which makes ch_no = "puppy"
ch_no <- dog
ch_no
```

```
[1] "puppy"
```

```
#character values can be as long as you want
ch_long <- "this is a really really long string that i want to save"
ch_long
```

```
[1] "this is a really really long string that i want to save"
```

Characters are, as I said, indicated by quotation marks. The best practice is to use double quotes. It’s better to be consistent. If you’re mixing double quotes and single quotes, you might have a situation where you’re starting with a double quote and ending with a single quote and R will consider that an ending string and get really confused about what you’re trying to do. Here’s an example we’ll create character value using double quotes. I’m going to call this dog.

I’m going to assign the ch_double object, the value of dog. Then if we try the same thing with single quotes, that does work. Doesn’t need to be a dog related character, and so we can call ch_double assign that the character turtle. So be mindful that when you don’t include quotes around a character, what you’re actually doing is assigning ch_no, the object dog. These are two objects. If you haven’t assigned dog any value, then ch_no will also be assigned no value. Because we’ve not assigned a value to dog, it doesn’t yet exist, and so R is going to show that error that dog has not found. At that point, you’d want to decide, am I trying to assign ch_no character value or am I trying to assign an object.

In this situation, we’re going to assign the character puppy. Then at that point, basically what we’re doing is we’re assigning the ch_no value of the dog object. Then we do the same thing as we had up here, and it will work this time because we’ve assigned dog that value. We can double check that, and ch_no is now the value of the dog was assigned, which is puppy. Character values can be as long as you want as long as your computer has enough memory to store it. I have an example here of a really long string and R handles that, no problem.

Numbers can be stored in three ways.

1. Numeric - both whole numbers or decimals
2. Integer - similar to whole number by indicated with an “L”
3. Complex

```
#numeric objects can be whole or decimal
num_whole <- 290
num_dec <- 290.9

#integers are indicated by adding an "L"
int <- 290.5L

#complex
complex <- 2+4i
```

Moving on to numbers. Numeric can be stored in three ways as numerics, which contain both whole numbers and decimals as integers. That’s similar to whole numbers except it’s indicated with an L, and it will not allow decimals and complex. Numeric objects can be without an L, specifying an integer can be whole numbers or contain decimals.

Again, integers are a special form that do not contain decimals, so if I tried to add 290.5 with an L, it will say that it contains a decimal, and so those are incongruent. So you can’t have that. I think if I try to call that in this case, it is defaulting to what is possible. So 290.5 as an integer is not possible, so it is creating that as a regular and numeric instead. Then complex numbers, you can assign in a similar way. They just have to be complex numbers.

Logical: Use all caps - TRUE or T, FALSE or F

```
#two options for assigning a logical value to an object
logical <- TRUE
logical_1 <- T

logical_2 <- TRUE
#this does not save as a true logical value, rather it saves the string
#"true" as a character
logical_lower <- "TRUE"
```

Then the last data type that we'll talk about is logical. Again, that is either true or false. There are two options. You can either do, as I said, all caps true or caps T, and those are the same value. If I put quotes around it, then that becomes a character value instead of a logical value. It may look the same. You can see logical lower returns. The value of true, but note that it's in quotes and so it's not an actual logical value. You'll note that when you don't include quotes, R recognizes this as a special logical value and will make it blue instead of green, which it does for character values.

Logical values are important. When we get to talk about comparisons, basically, anytime you're trying to make a comparison like is five greater than 10, what R is going to return to you when you ask it to evaluate that is a true or false. There are a lot of functions that depend on logical values being returned in order for conditional programming, you'll use true and false a lot. It's important to recognize the difference there.

Extra details -

- Dates stored as numbers (the number of days (for dates) or seconds (for date/times) from January 1, 1970
- There are several "constants" available in Base R (i.e. today's date)
- Missing values are stored as **NA**

The extra details about data types, dates in R are stored as numbers, the number of days for dates or the number of seconds for date times from January 1st, 1970. They're stored those numbers, but they're displayed in a way that we can read. Basically, year month day is the standard way of R date formats. Then there are several constants in base R, including things like today's date and many others. Then missing values and R are stored as NA, and we will get into that in future weeks.

Lecture 2.3: Data Structures/Objects

Data Structures in R

The primary data structure in R is made up of objects.

We'll go into detail on these below except for

Factors - These are a little complicated, but the idea is there are fixed categories, that will display in a fixed order; this is especially useful for forcing output to display in a fixed manner, or modeling. These will be covered in detail later in the course.

Now I'll talk about data structures in R. The primary data structure in R is made up of objects, and there are five types of objects. There are a lot of packages like spatial packages have specific objects that are specifically related to those packages. There are more, but these are kind of the standard base R structures.

The first is an atomic vector, which is a single dimension and must contain only a single data type. That's one dimension that can contain a whole bunch of spots. This one contains five, as an example. But they all must be the same type, so they all must be either numeric character or logical.

Matrix is very similar in that it needs to also contain a single data type, but it can contain multiple dimensions. This looks like a standard tabular data set. But because it is all one data type, it can be stored as a matrix. Both atomic vectors and matrices have advantages in that you can apply functions to all of the values in either of them all at once. R can operate more efficiently because it knows that all of those are of the same data type, and so it doesn't have to do the compute work of figuring out what's in that field before doing those calculations. There are big advantages in using those, especially if you're doing a lot of calculations.

List is an ordered collection of objects, and they don't need to be of the same type. You can see here, this is one type, this is another. Doesn't have to be consistent. It's nice to have that flexibility, and you can even create a list of lists, which we will get into later when we talk about lists.

The fourth type is a data frame. That is what you are going to be most familiar with, in Excel spreadsheets are essentially like this or if you're familiar with databases. It's a structure of columns that contain different data types. It is similar to a matrix, and that is multi dimensional. But the column 1 has all of the same data type, column 2 has all of the same data type, but they don't have to be the same.

Then the fifth category that we'll talk about are factors. They're categorical variables, but they're stored as an integer. When you store things as an integer, R can process them faster and more efficiently. But sometimes integers don't mean anything to humans reading it. Factors are a way to assign a character value or like a classification like an age category. It will show you the age category when you see it on the screen, but it will be storing that age category as a number behind the scenes. This is frequently used in modeling. Let's see it in building visualizations and even in building tables for display. We'll go into some detail below.

Vectors

(Atomic) Vectors - this is a one dimensional object that can only contain a single data type (character, numeric, logical). Even a single value is stored as a vector in R (basically a vector of length 1)

- Multiple ways to be created:
  - c() function
  - Using : operator for a vector of consecutive numbers

```
#create a numeric vector using the c() function
vec_num <- c(1,5,6,94)
vec_num
```

```
[1]  1  5  6 94
```

```
#create a numeric vector using the : operator
vec_num2 <- 1:10
vec_num2
```

```
[1]  1  2  3  4  5  6  7  8  9 10
```

```
#create a character vector using the c() function
vec_char <- c("dog","cat","mouse")

#important note - a single value in R is stored as a vector of length 1

vec_one <- 100
class(vec_one)
```

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

```
#try creating a vector with multiple data types - this will force 290 to be
#stored as "290"
vec_multi <- c(290,"ph")
str(vec_multi)
```

```
chr [1:2] "290" "ph"
```

We'll talk about atomic vectors first. There are a couple of different ways you can create a vector. One most common way is using the C function. When I'll run here, is just I'm using the C function and creating a vector containing 1, 5, 6, and 94 and then assigning it to vec_num object. You can see this is what a vector looks like. It is a single dimension containing those four numbers.

We can also create a numeric vector in a sequence by using the colon in between two values. I'm assigning vec_num2 to be a vector one through 10. It contains all of the numbers 1-10 using that colon. Vectors can also be character. They can also be logical, and I can create both of those using the C function. This is I'm creating vector containing dog cat and mouse. I'll just run that. You can see that.

An important thing to note is any time you store anything in R, even a single variable, like we've been doing in the previous examples, that is essentially a vector of length of one. Just assigning vec_one v+alue of 100, that is assigning it a single length vector. We can confirm that using the class function, that vec_one is a numeric vector.

We can try creating a vector with different data types. This is creating a vector with a number and trying to add a character to it. What it's going to do is default again to what is possible. It's impossible to convert pH into a number. It's impossible to do that, but it is possible to convert 290 into a character. We can see if we check what did with those. I did what I said. It converted 290 into a character and left pH as a character.

Matrices

- Multi-dimensional, single data type
- Created using matrix() function

```
#create a matrix using the : operator to define the data included
matrix_1 <- matrix(data = 1:12, nrow = 3, ncol = 4, byrow = TRUE, dimnames = NULL
matrix_1
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Lists

- One-dimensional, multiple data types and/or objects
- Created using list() function

```
#use the list function to see what happens if you add items of different types
my_list <- list(290, "290", "ph")
my_list
```

```
[[1]]
[1] 290

[[2]]
[1] "290"

[[3]]
[1] "ph"
```

```
str(my_list)
```

```
List of 3
 $ : num 290
 $ : chr "290"
 $ : chr "ph"
```

More on data frames in an upcoming week!

Matrices are multi dimensional. They need to be a single data type, and they can be created with the matrix function. Here's an example of a matrix containing one through 12. All of the numbers there within those two numbers, and then it's going to create it as having three rows and four columns. It is not going to name the dimensions. It's just going to create that matrix. If we take a look at what that looks like, it is three rows and four columns, and it just assigns those spots within the matrix according to their value.

Again, lists are one dimensional. But they can contain multiple data types, so they're similar to vectors, except that they're more flexible in what data types they can contain. We'll take that example from before up here. We'll have 290 as a numeric, 290 as a character, and we'll add pH in there as well. We can create that as a list using list function, and you can see those values get stored and they're stored in the format that we asked them to and there's no conversion going on. Then we can do a STR a function that just describes an object, and you can see that it's a list, and those are the values.

Then we'll talk about data frames more in depth in the upcoming week.

Using functions to describe objects

A few examples:

- length() - how long is the object?
- class() - what type of object is it?
- typeof() - what data type is the object?

```
#return information about matrix and vectors created above
length(matrix_1)
```

```
[1] 12
```

```
class(matrix_1)
```

```
[1] "matrix" "array"
```

```
typeof(matrix_1)
```

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

```
length(vec_num2)
```

```
[1] 10
```

```
typeof(vec_num2)
```

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

```
length(vec_char)
```

```
[1] 3
```

```
typeof(vec_char)
```

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

Before we move on, I just want to talk a little bit about some functions you can use to describe objects. I just talked about one, which was STR, which just does a general description of what the object is. You can also use a length function to determine how many spots there are in a matrix or a vector or a list. If we look at that matrix_1, you can see that it has a length of 12. It's three rows and four columns. It contains 12 spots.

The class of a matrix is a matrix or array, and then the type of the matrix is integer because that contains all integers. Then if we take a look at the length of vector number 2, that was the one through 10 using the colon. That has a length of 10, and the type of that one is integer also.

Then if we take a look at the type of the vec_char, we can see that that is character. We can also take a look at the length, and that length is three. Those functions work for all different object types, and they are especially helpful when you're coming back to code that you haven't worked with for a while, and you want to check on what objects are length. I use that all the time to determine sometimes I'm doing a calculation. I don't actually know what the length is until after the calculation. I need to know how many operations going to run. Length is really a useful tool there.

Lecture 2.4: Calculations and Comparisons

Calculations and comparisons

Calculations

R can be used as a high-power calculator in the console and in the script.

Calculations can be made on numbers and objects.

Example:

```
> 200+90
[1] 290
> x<-290
> x
[1] 290
> y<-200-90
> y
[1] 110
> x*y
[1] 31900
```

Calculation	Operator	Example
Addition	+	200+90
Subtraction	-	200-90
Multiplication	*	2*90
Division	/	90/2
Exponent	^	90^2
Absolute value	abs()	abs(-90)

...and more!

On to calculations and comparisons. So calculations can be used as a high-power calculator in the console and in the script. You could do all kinds of calculations, even simple ones or complicated ones in R, and it will handle all of that. And the operators you use are plus sign for adding, subtraction is the minus sign, multiplication is the asterisk, division is the forward slash, and then exponent is the caret, which is Shift 6 on a QWERTY keyboard. You can also use functions, so absolute value. There's square root function. There are a lot of functions that you can do to calculate things as well.

```
#calculations can be performed on numbers
54*38743252349
```

```
[1] 2.092136e+12
```

```
#and objects
a <- 4
b <- 75

b/a
```

```
[1] 18.75
```

```
#calculations can be performed on vectors
vec_num2*10
```

```
 [1]  10  20  30  40  50  60  70  80  90 100
```

New objects can be created as result of calculations.

```
matrix_2 <- matrix_1 * 5
matrix_2
```

```
     [,1] [,2] [,3] [,4]
[1,]    5   10   15   20
[2,]   25   30   35   40
[3,]   45   50   55   60
```

```
c <- b-a
c
```

```
[1] 71
```

We can calculate things on numbers so that 54 times that very large number yields a number with scientific notation. You can also run calculations based on objects. I'm assigning a value of four and b the value of 75, and then instead of saying 75/4, I can instead use those objects and just say b/a, and it will give me 18.75. Then calculations can also be performed on vectors so that vec_num 2, which is, I believe 1-10.

What we are doing here is multiplying 10 by each of those values, and it does it in one step. It's not actually iterating through all of those. It's able to calculate that on each vector at once. That's part of that efficiency that I talked about earlier, that R can do things a lot faster than some other programs can do. You can also create new objects, so we have that matrix_1.

If we multiply all of those values and the matrix_1 by five, and then we can assign that to a new object called matrix_2. If we do that, we're multiplying five by each of those, and we have a new matrix that has those results. We can also subtract. We're creating a new object c, which is b-a and so c is now 71.

Using functions for calculations

In addition to the operators above, there are many functions that can be used to do calculations.

```
#example: absolute value
abs(-90)
```

```
[1] 90
```

Then we can also use functions for calculation, as I mentioned before, so the absolute value of -90 is 90. There are all sorts of calculations that you can use functions to get. Another example would be, see if this works. If I want to calculate the mean of of a, b, and c. I'm not sure what exactly happened there, but it does work to calculate a mean of those objects. There are a lot of other operations that we'll get to later in the course.

Comparisons

Two values or objects can be compared to assess if equal (==), unequal (!=), less (< or <=), or greater (> or >=) and **will return a true or false**.

Example:

```
> a<-200
> b<-90
> a==b
[1] FALSE
> a!=b
[1] TRUE
> a<=b
[1] FALSE
> a>=b
[1] TRUE
```

Comparison	Operator
Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

```
5==40
```

```
[1] FALSE
```

```
"dog"=="cat"
```

```
[1] FALSE
```

```
a!=b
```

```
[1] TRUE
```

```
b>c
```

```
[1] TRUE
```

```
d <- (a+b)/c
d
```

```
[1] 1.112676
```

```
d2 <- (a+b)>c
d2
```

```
[1] TRUE
```

Let's move on now to comparisons. Comparisons in R, anytime you want to compare two objects to assess if they are the same, they have equal value. If they have unequal value, if they're less than or less than or equal to or greater or greater than or equal to, anytime you use an operator, it is, so behind the scenes, going to return either a true or false logical value.

A key difference in R, especially from SAS is that equal is symbolized by two equal signs together. Not equal is symbolized by an exclamation point and an equal sign, less than is less than, greater than is greater than, and less than and greater than or equal to, both have an equal sign behind them. We'll take a look at this if we want R to evaluate whether 5 = 40. If we run that, it's going to be false. Likewise, we can do that with character values. Is dog equal to cat? That is also false.

Here, we're using the value of a and b again, so one was four and the other was 75. It should be that we're testing to see if a is not equal to b and in this case, that is true. Or then going to test if b is greater than c, and do you refresh b was 75 and c was 71, so b is greater than c, and so that is true. Then we can also similarly assign a new object of value based on a comparison.

Let's calculate a new variable called d, which is a+b/c, and so d in this case is 1.12. Then we can assign the value of d2, whether or not a+b is greater than c. In that case, we have saved or we have assigned the object d to the value of true. Basically, the result of that comparison. Thank you for listening.

Lecture 2.5: Best Practices for Naming Things (and Coding Style)

Best practices for naming things (and coding style)

Why is naming things important in programming?

Taking a very big step back - naming is important because there is already a lot of complexity in the datasets we use and analyses we perform in public health. We don't need to add more complexity with poorly organized and inconsistent code. Using consistent naming and coding style makes code easier to read and understand, so you can focus more of your attention on the complexity that matters.

It's most important, whatever naming/style convention you choose, to remain consistent with yourself. If you are working with a group (or might possibly share your work with others), it's good practice to settle on a naming/style convention and prompt everyone to keep with it.

Hi everyone. I'm going to talk a little bit today about best practices for naming things and talk a little bit about coding style as well. The reason why these two things are very important in programming is you want to maintain as well organized workspace as you can. It's like having a well-organized desk or a well-organized room where a well-organized file system. It just makes things easier to work with.

The problems that we are trying to deal with epidemiology, biostatistics, etc. In public health are already complex. The datasets are already complex. What we're doing by standardizing the way we're writing code and naming things is taking some of that complexity out of it, simplifying what we have control over and then that makes it easier to focus on the more difficult things.

Then the third reason is it makes it easier for collaborators. That includes, as we often do in this course, includes your future self. If you finish your program, you think it's good and it's working, and you come back to it two months later because either you have an idea to change it or your boss does or somebody does, you want to be able to pick up what you were doing two months ago with relative ease and not have to sort through a bunch of different ways that you're naming things or different ways that you've set up your code. That's why spending a little time on this is important.

I'll just note talking about this before you have really launched into writing code is a little bit backwards, but you can always come back to this video later on in the course if you have questions. I've also developed this document and that will be in the course materials that you can use as a reference point later on in the course once you start writing code.

Naming and style guides

There are lots of naming conventions and style guides out there. If you google "R coding style guides" you'll get many results, but the most common are the Tidyverse Style Guide and the Google R Style Guide (which formed the original structure of the Tidyverse style, but Google has since borrowed parts of how Tidyverse has evolved). It's obviously most straightforward to adopt an existing style, but it's also perfectly acceptable to adapt a style to meet your needs, or come up with something new on your own. If you do so and collaborate with others, it's generally appreciated to document your adapted style so that others can understand.

There are also R packages built to help guide or correct your code according to specific style guides. The styler package will take code that you've written (either a whole file or just a chunk of code) and apply the Tidyverse Style (or a custom style) to it. The lintr package will identify which parts of code don't follow a specified style, but puts the onus on you to make the suggested changes. These are both helpful tools to practice organizing your code.

I'm just going to step through this document. Style guides in particular are a lot of them out there. You can Google R coding styles and you get a lot of results. The two main ones, I think, are the Tidyverse Style Guide and the Google R Style Guide. They've evolved alongside each other. I think Google started an R style guide and tidyverse borrowed from it, and then more recently Google borrowed from that. But there are lots of other ones that people have developed on their own or teams have developed for their specific needs. You can go and see if any speak to you. But we're primarily going to recommend what the Tidyverse Style Guide has.

You're certainly welcome to use whatever style guide you'd prefer but I think that key part of it is that you are using something consistently across the course or across your career just so that you recognize, like I said, when you come upon a program you wrote two months ago, you can pick up things pretty quickly.

There are also a couple of R packages that are built to help guide your code writing style. That's the styler package where you can run a whole file or a chunk of code through styler and it will basically convert whatever you've written to, I think it defaults to the Tidyverse Style Guide, but you can also create a custom style.

The lintr package is more, you can specify a file and it will go through and let you know, give you warnings where your code is outside of the style, but then you have to go in and make those changes. Those are two interesting packages to take a look at, and I've included the links to the documentation here.

General naming/style guidance for this course

The most important principle is to ensure your names are descriptive and meaningful while also as concise as possible. If you find this challenging, you are not alone. But with practice, you'll get figure out how to get there. Here are some tips to help.

Naming code files

For script, R markdown, or quarto file names, make sure they are descriptive enough so that you know what is contained within without having to open up the file. You'll often be trying to figure out which file to open in a folder, so be descriptive as you can.

We recommend **against** using code file names for version control (i.e. idmodel_v20230813.R, idmodel_ww_edits.R, idmodel_final.R). It quickly gets confusing for collaborators (including your future selves) to know which is the right file or which one has the final edits. We encourage you to use git for version control and cover how to do this later in the course!

The other part of this quick talk is about naming code files. Naming is a very challenging balance when you're programming. You want to ensure that your names are as descriptive in meaningful as possible so that you don't have to dig into each line to know what's going on or each file to know what's going on but you also wanted to make it concise as possible. That is the balance that you're dealing with.

Here are some tips to help you out. But understanding that this is something that all programmers find challenging, and it just takes practice to figure out how to do this as you're programming.

For naming code files for R script, like.R files or R markdown files, make sure that they are descriptive enough so that you know, what is going on. What the program in there is doing, what the code in there is doing so that you don't have to open up the file and see. Typically, you'll be trying to find a file to open to modify or something like that. You'll be looking through the R interface. You'll be looking through a folder structure, or on your Windows or Mac, you'll be looking through your File Explorer and trying to find the file that you want to open. If you have descriptive names, it's easier to figure out which one you should actually open without opening them. It's just a tip to make your life easier.

We recommend against using code file names for version control like putting a date at the end, or like Version 1, Version 2, 1.2, 1.3, etc. It gets quickly confusing for collaborators to know which is the right file to run, which one has the final edits, or which one you intended to be just an experimental one versus which one you actually want to go with for the final result.

Later on in the course and a couple of weeks, we're going to introduce you to using Git for version control. That will handle all of the version control management. That makes that a lot easier. You don't have to use the filename for that purpose.

```
# Good
climate_data_import.Rmd
disease_data_wrangle.Rmd

# Bad
program1.Rmd # not descriptive
calculations.Rmd # way not descriptive
program1_plus_calculations_v13_final.Rmd # don't use filenames for versioning
```

I included a couple of examples here of good naming. Usually It's combination of nouns and verbs. I start out with the noun that it's climate data, and a verb that I'm importing it. Or maybe there's some specific disease and there's data wrangling going on.

I included some bad examples here that are just not descriptive. Program 1, obviously that means that it happens near the beginning. If you've got a number of different programs that run different steps, that's, I guess somewhat informative, but maybe not descriptive. Calculations is just way not descriptive. Then this is an example of the complications that things can get you to have a Version 1.3 and that's the final but what if there's also a Version 1.4 in the folder, which is the more up-to-date one.

Also if you have files that are in sequence you start a file name with a number. But if there's even a remote chance that the number of files will exceed 9, add a leading 0. This will ensure that the files will sort correctly in a folder.

```
# Good
01_climate_data_setup.Rmd
02_climate_model_run.Rmd
03_climate_model_viz.Rmd
10_climate_id_match.Rmd
20_might_never_have_this_many.Rmd
22_but_you_get_the_point.Rmd
...

# Bad
10_climate_id_match.Rmd
1_climate_data_setup.Rmd
20_might_never_have_this_many.Rmd
2_climate_model_run.Rmd
22_but_you_get_the_point.Rmd
3_climate_model_viz.Rmd
```

Also just a tip. If you have a sequence of programs that you need to run in sequence, you can name them with a leading number. I would just encourage you if there's a remote possibility that you might have 10 or more steps to put a leading zero in because this in your file system, it will sort correctly if you put that leading zero in, but it will sort incorrectly or out of order if you don't. I just encourage you to think about that before starting. That was talking about files.

Naming objects (dataframes, columns, vectors)

This is where a lot of naming happens. When naming objects in R, it is preferred to use:

Use Descriptive Full Words...

Abbreviations can get confusing especially if you forget what you intended them to stand for. So in general avoid abbreviations that are only familiar to you. Abbreviations and acronyms are okay if they are familiar to others and widely used.

... But Also Make Names Short As Possible

The hard part of using full words is you also want to make names as short as possible. If you have a bunch of descriptive words describing the specifics of a dataframe and also specifics for each column, it becomes difficult to keep things tidy and on one line.

```
# here's a ridiculous, but not unheard of example

legionellosis_europe_western_france_positive_hospitalized_2020$cases_severe_ageca
```

What I generally try to do is keep the number of words I use in a name to 3 and no more than 4. If I start to have four-word names, I'll take a step back and see if there's another way I can organize either my dataframes or columns. We'll get into ways of doing this later in the course - but an example with the above dataframe is it's probably okay to not filter out some of those categories. Even if I filter using continent = "Europe" and region == "western" and country == "France", if I'm just interested in French cases, I can just use france in the name. Likewise a hospitalized case infers that the test was positive, so I don't need both. And it's probably okay for me to keep all years in the dataframe; I can always filter out rows I don't need later in the process. So I could probably get away with calling the dataframe legionellosis_france_hospitalized.

Now I'll talk a little bit about naming objects. Typically in R, those are going to be the DataFrames, the columns within a DataFrame or vectors, lists, those sorts of things. This is where a lot of the naming, especially writing code on the fly, where the naming happens. We encourage you to use descriptive full words.

Abbreviations can be useful to shorten things, but gets complicated if you're using abbreviations that really only make sense to you in that moment. They can be confusing later on or confusing to other people. If they're common abbreviations or if a team has come up with glossary of abbreviations to use, that's a good way to shorten code. And you can have that central reference point for abbreviations. Acronyms are similar. You can use them if they're familiar to others and widely used, but would avoid them otherwise.

Then the balance of that is making names as short as possible. This what I call the ridiculous but not unheard of example of just a very long this would be the dataset here. Very specific, very descriptive, but maybe a little too descriptive. Then you have also a very long column. The dataset names and the column names get too long. You're going to have very long code lines like this anytime you're trying to transform or do anything, any wrangling to your dataset. You're going to have these terribly long, messy-looking lines in your code. I believe the tidyverse style guide recommends just having 80 characters on a single line. So there are ways that you could modify this to fit it a little better and make it a little nicer. But generally if you can keep your names shorter, you'll be better for it.

And I have some tips. I generally try and keep names of things down to three categories. I would say, maybe Legionellosis is important to keep in there. But maybe I don't need Europe and Western Europe and France in there. Maybe I don't need positive and hospitalized because, if somebody is hospitalized with Legionella, you don't really need to know that there are also positive. Maybe I can just not specify the year and just filter differently. Include all of those years in the dataset.

Consistent hierarchy

It's also good to employ naming with hierarchy of increasing specificity. The parts of names start with the more general and become more specific moving left to right.

```
# Good
covid_cases_breakthrough
flu_forecast_seir
case_rate_rank

# Bad
yr2022_numerator_mpx_rate
```

Avoid Using Names for Common R Functions

Things will quickly become confusing if you use names of R functions, variables, or values for object names. Again, we're working to remove as much unneeded complexity from your code as possible.

```
# Bad
T <- FALSE
filter <- id_data %>% filter(T == FALSE)
```

Consistent case

We advise using snake case (all lower case separated by underscore i.e airquality_region4_pm25) for names of objects, columns, functions etc. Other possibilities are:

- camel case: airqualityRegion4Pm25
- pascal case: AirqualityRegion4Pm25
- kebab case: airquality-region4-pm25

You might join a group in the future that prefers camel or kebab case. But we recommend (and use!) snake case for its simplicity and consistency.

Another tip, it'd be to use consistent hierarchy. Most generally going from general description at the left to more specific as you go from left to right. If I've got a number of disease conditions that I'm doing analysis on, I want to specify COVID is the most general. Maybe I'm working with COVID test cases and deaths, so I would specify cases for this. And then breakthrough are a subset of those types of cases.

Similarly, we're just saying this is most general rate to be a subset of case and rank would be a subset of the types of analysis or types of information that's contained in that column.

A bad example is just, there's no hierarchy at all. We've gotten numerators earlier on over to the left where it probably makes more sense over to the right. Just something to think about as a way to simplify your naming convention.

Another thing to do is avoid use of common or functions with names, so you would never want to name a DataFrame true or false. Those are protected, those are commonly used values in the R system and you'd never want to also named a DataFrame that. Likewise filter is a big verb in the deep layer package which you'll learn about. Would also not want to name a DataFrame filter. Anything that you would commonly use in R to do calculations or functions or things like that. You would not want to name as a DataFrame because it just gets confusing.

Also consistent case, so we recommend using snake case, which is all lowercase letters separated by an underscore. This is an example, airquality_region4_pm2.5. You can use that for all of the names of objects, columns, and functions. There are other types out there. This is an example of camel case where the first word, first letter is lowercase, but all first letters of subsequent words are capitalized. It's like a camel's back. Pascal case is basically camel except the first letter of all words are capitalized. And kebab case is basically the same as camel, except with dashes instead of underscores.

You might have a job in the future that prefers some other case, but I think we're for consistency and simplicity, we're recommending snake case.

One Strategy to Consider

If all of this seems overly complicated, I don't disagree with you. Naming things, just like any effort to keep things organized, takes effort (refer to the famous quote about naming being one of two difficult things in computer programming). But it will get easier with practice. One strategy I sometimes use so that I avoid getting hung up the naming at the expense of my actual work, is to use names temporarily as I'm programming and then go back and make them more descriptive later.

```
# example specifying 'sandbox' as a reminder to go back and name later
# and using greek letters or nato alphabet (or whatever sequential thing you like
sandbox_legionella_alpha <- step_1
sandbox_legionella_bravo <- step_2
# <some additional steps>
sandbox_legionella_echo_modelA <- model_stepA
sandbox_legionella_echo_modelB <- model_stepB
```

Then when I'm all done and satisfied with the project (or part of the project), I'll go back and change the names to something more meaningful (often using the Find and Replace feature in RStudio - just make sure to use exact match!). You're also more likely to know what meaningful names will be at the end than it is to guess at the beginning. This would also be a good time to run the styler or lintr package on my code to make sure it matches up with my preferred style.

Then I have a strategy to consider, basically saying, you can simplify naming by just temporarily naming everything with a sandbox in front. For a column names, you don't really need to do that. But for DataFrames, you could call that a sandbox, which just lets you know that this is just a temporary name. Then for each DataFrame, if you're building a dataframe with lots of different steps, you can just go through some sequential naming conventions.

This is the nato alphabet, or you could use Greek letters. You're basically saying the stuff that I'm doing around Step 1, I'm just going to generally name it Alpha. And then here's Step 2 and is generally named Bravo. And then I can go through that whole thing, not have to use any mental energy trying to come up with the right names while I'm working on this.

And then later, once I'm satisfied that the code does what it's supposed to do, I can go back in and rename things, especially using Find and Replace. That's a really good way to do that. This might also be a good time to run this styler package to make sure your code fits a style. Then you can be satisfied that your code is neat and tidy, well-named, and well-organized. All right, thanks.

Lecture 2.6: R Markdown and Quarto

R Markdown and Quarto

R markdown and quarto are ways to write your source code and combine it with text and the output of your code into a finished document. These types of documents can be really helpful for doing assignments, keeping track of your progress, documentation, reports, sharing your work with others, teaching, and more. Both R markdown and Quarto work by allowing you to write R code and markdown and then using this to write other code for you to create an output document.

R markdown vs. Quarto

- R markdown files: These end in "**.Rmd**". These files allow you to combine your code, documentation of the code (text), and output all in the same document. You can then "knit" R markdown documents to create PDF, HTML, or other files as output.
    - R markdown cheat sheet
- Quarto files: These end in "**.Qmd**". These are similar to R markdown documents, but have a few more features. You can then "render" quarto documents to create PDF, HTML, or other files as output – and quarto files have more output options
    - Quarto basics in the R for Data Science book

I'm going to step you through today how to use R Markdown and Quarto documents. These two different files are a way to write your source code. That's an R. With Quarto and Markdown you can actually write in other languages as well. But for this course we're just going to be using R. These type of documents can be really helpful for doing your assignments. Documentation, writing reports, sharing your work with others. We're going to use it for teaching, obviously, creating lots of interactive and non interactive visualizations and things like that. Both of these types of documents allow you to write R code and markdown.

Markdown is a specific way of writing content that is a shortcut to HTML, to writing web pages, and you will get some familiarity with how to write markdown. Then you can use this to write other code to create an output document. R Markdown has actually what we've used previously in this class.

Quarto is a more recent file format that RStudio and a number of different collaborators have developed over the past few years. These are similar for our purposes. But Quarto files do have a few more features. You can render Quarto documents similarly to R Markdown to create PDF, HTML or other outputs. But Quarto files have a few more output options, and they can also handle a few additional languages such as Julia.

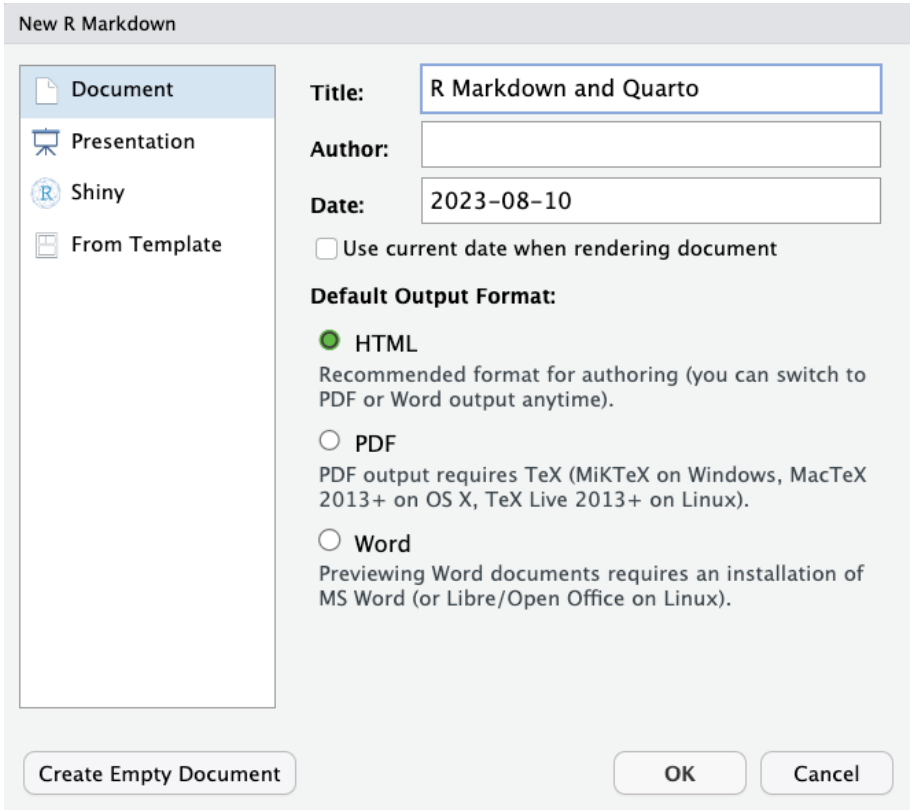
What I'm going to do today is I have on the left side of my screen the script and actually has switch the script. For Quarto documents, you can actually have two different types of ways that you edit the source. You can either edit via this little button here that says source, or this button here that says visual, that gives you a preview of what it's going to look like. Which is a little bit easier when you're writing more narrative or content to do it that way.

But on the left side here I'm going to have the source version of this document, which I'm displaying. What it looks like what the output of this document looks like on the right. These are basically the same thing. Just on the left is what generates what is on the right. I'm going to try to scroll through these so that you can see how this works.

Creating a new document

To create a new R markdown or quarto file:

1. Click the new document button in the upper left.
2. Select the file type.
3. Fill in the information in the popup window. This will set some of the basic properties of the document for you.



4. Remove or edit example text and code as needed.

First of all, to create a new document. What you do is you go to the File menu and click on “New File”. You can see here a number of choices. You can either choose a Quarto document if that’s what you’re interested in creating, or an R Markdown document. What happens next is you get a pop-up window with some options. This isn’t new R Markdown document what this looks like. You can add some title and an author and a date, and then choose one of three outputs, either HTML, PDF, or Word.

We can just open up what the Quarto document looks like. It’s very similar. You have these three options. But on the left here, you can see actually similar options for R Markdown. You can also choose the presentation. If you want to create a PowerPoint or Java version of a PowerPoint slide set, you can choose, Beamer or Reveal JS. Then you can also create a shiny, which we will get to much later in the course. This allows you to do interactive documents. Don’t worry about that right now. We’re going to just focus on the way that we’re going to use this for the most part in the course.

Once you’ve selected the format that you’d like for this class, for the most part, I believe we’re going to be using HTML, some assignments may be submitted via PDF because that’s a little bit easier for us to read. The HTML files can get large. Once you do that, you can click “Okay”, and that will create a new document for you.

Again, I have the script, the source code on the left side. You can see I have, the title is taken from this little section here, which we'll get to in a moment. That gets displayed here. Then I have this first paragraph of content which gets displayed right under the title. This information here is part of the file header so it's giving instructions when you render what's in this document. It takes instructions from here to know what to do and how to render this. Then every code section will have these three ticks. It's actually the tick is under the tilday, which is next to the one key. It does not require a shift so you do three of those. Then curly brace open, and then just choose the programming language. For the most part, we'll be using R and then curly brace close. That will open up this code window, where we're loading Pac-Man packages, which we'll get to later in the course.

Then you can see we just went over this R Markdown versus Quarto. You can see that this is written in script over here, and then that translates to the right. This is how you create a bulleted list, is using this dash, this dash here, it creates this bullet over here. It's just basically taking the content that you have in this script and publishing it into a more pleasant way for your audience to look at.

YAML

The YAML header that explains what type of document to build from your R Markdown file. You can also add details like a title, your name, and the date. The formatting of this header for R markdown is very similar, but has a few minor differences. The easiest ways to get started with writing the YAML header are:

- Follow the above steps for new document creation. This will create a basic YAML header for you.
- Copy an existing YAML header and edit information as needed.

I'm going to skip down now to talk about YAML. YAML is through some file header information that talks about what type of document you want to build, what meta-data you want the document to have. Then what kind of formatting typically is handled in the YAML.

Oftentimes when you're just creating a file from scratch, a basic YAML will automatically be created for you so you don't have to deal with it. Later on in the course, we'll get into more advanced use of R markdown and quarto files, and we'll get into some different options that you can put in the YAML header manually.

Text formatting

The examples below can be used for formatting both R markdown and quarto documents. You can look at the "source" editor to see how this formatting is done with code – or you can use the "visual" editor to make many of these changes.

Header 1

Header 2

Header 3

Header 6

- list
- list
- list
    - sub-list

1. number 1
2. number 2

italics

bold

For text formatting, there are a number of different ways that you can get a document to look like you want it to. This is an example of a header, this is an example obviously of a header. It provides separation, if you're starting a new section of a document, it provides a little bit of separation between the two parts and how you specify a header is just having this pound key or hashtag. Just having one of them produces the largest header so that would be for big chapter level headers and then each one gets smaller.

You can do Header 4 and Header 5 and Header 6, but they just keep getting smaller and a little less bolded. You can create a bulleted list with the dashes I said earlier, and then do a sublist with indented dash. This is indented with two tabs in order to get that sublist.

You can also list with numbers just by including a Number 1 and a dot and two spaces, and then a Number 2 and a dot and two spaces. Then you can specify italics with a single star or asterisk, and bold with a double star, and asterisk. You can see those represented there.

Embedding code

You can also embed code in your R markdown or quarto documents. How this looks and functions between the two document types is a bit different.

Here is an example of adding code:

```
1 + 1
```

```
[1] 2
```

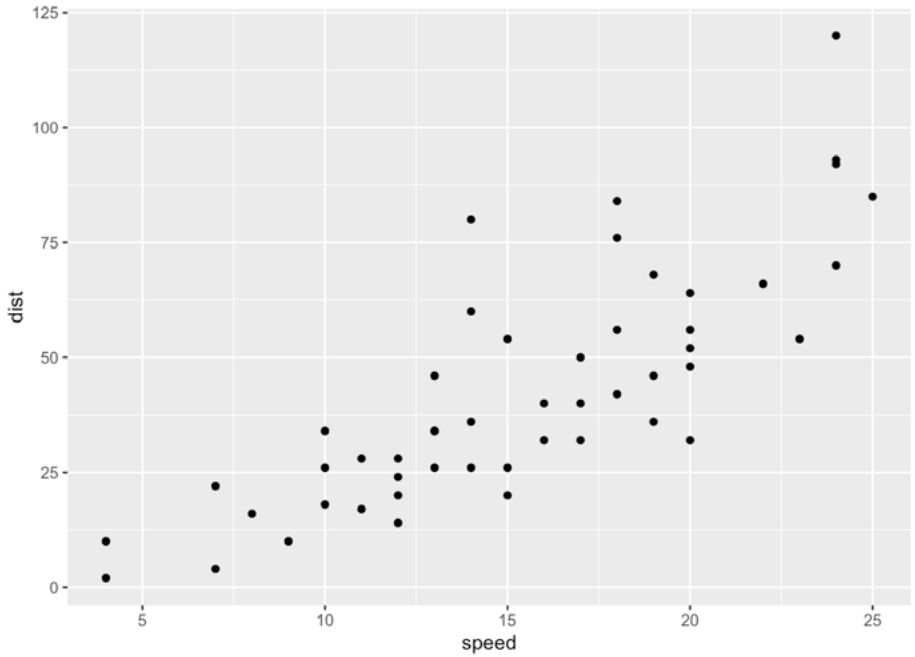
You can add options to executable code like this:

```
[1] "hello world"
```

The `echo: false` option disables the printing of code (only output is displayed).

You can also run code for plots and show the output in the document. Below we are just using some sample data in R to create a basic scatter plot.

```
ggplot(cars, aes(x = speed, y = dist))+
  geom_point()
```



What we will be doing a lot of in this course is embedding code into documents. As I said earlier, you can do this by typing three ticks and then curly brace and I hit their curly brace button in both the open and the close appear and then I just need to specify that it's the R programming language that I want.

Then I can put whatever code I want, as simple as one plus one, and then when I run that by clicking on this Run Current Chunk button, it will calculate that it's 2. That is one plus one. That's represented over here in the document and over here in the source visualization.

You can also add options to execute code like this, so if I want to run a function which is the print function and print hello world, I can just put that code in there and hit Run and it will print hello world. In this situation, what I'm doing is including a line that says echo equal false. What echo equals false does is basically, if I say I want echo to be false, I don't want that code line to show up, I only want the result line to show up.

If I were to do this as true, and render this file, it would include that print hello world line in there but if I don't want to, it depends on if I'm working with a collaborator, I want to show them my work. Show them what I'm doing with my code. I may want to include that code line in there so they can see what I'm doing. If I'm presenting to executives, if you don't want to know the details, they just want the result, I'm more likely to say echo equal false there.

You can also run code for plots or just show outputs and tables, we'll get to that much later in the course for how to output visualizations but all that code can be contained in these R code chunks. That's what these are called. They're called code chunks. Basically here I am running a GG plot, which is a function for creating visualizations using that code there and then basically when I run it, it will load the library GG plot. Down here, it will work. It will show up over here and then when I render it, it will show up in the published version.

Embedding Images

You can also embed images in R markdown and quarto document. You can do this using the source code (view source code here for example), or the visual editor. In this case, we've also added some code in the source code to center the image.

You can also embed images, you have to first put an exclamation point and then an open and close brackets. You can actually put texts in there if you want a image to have a caption.

But in this case I just want to show the quarto image. I can save that quarto image as a JPEG somewhere in my file system, so I have that stored in an image folder here. I have that quarto JPEG there and so I'm referencing that here. I'm basically saying, look in the folder for images and try to find that quarto.JPEG file and then display it here. I'm also in these curly brace here, I'm telling it to align in the center, and if we go back to the viewer, we can see that, that is aligned in the center.

It's also possible to align it to the left and then you can add text wrapping. You can do all sorts of things. But that's how you reference a image.

Further resources

There are a lot of resources online for further formatting and tools that you can use in these documents. In addition, it can be helpful to look at the source code for documents we've provided in the class.

- R markdown cheat sheet
- Quarto basics in the R for Data Science book
- Quarto guide on the quarto website

And then further resources, there are a lot of resources online for how to write R markdown, how to produce exactly what you want to produce. We recommend the R markdown cheat-sheet that has a lot of great tips for R markdown.

And then there is quarto basics in the R for data science book. That's a chapter in the R for data science book and then you can also look at the quarto guide on the quarto website for more information about quarto. Thank you.

Lecture 2.7: Function Basics

Function Basics

What are functions?

Functions are modules of code that accomplish a specific task. Functions usually take in some sort of data structure (value, vector, dataframe etc.), process it, and return a result.

Nearly everything you do in R relies on the use of functions. Functions are available from:

- Base R - thousands of functions are readily available to use in R
- Packages - every R package is comprised of additional functions created by the package authors. Packages are great places to look for functions
- Custom - Users (you!) can create functions that meet your very specific needs. We will cover a simple example below and more complex examples later in the course.

What are functions used for?

In short, functions are used for almost everything in R!

Some common uses include:

- Creation of new objects - vectors, lists, data frames, visualizations
- Operations on objects - creating a new column, calculations
- Applying logic
- Summarize or describe existing objects

In this short video, we are going to be talking about functions. What are functions? Functions are modules of code that accomplish a specific task. Functions usually take in some data structure, like a value, an object, data frame, etc, process it, and then return a result. Nearly everything you will be doing in R relies on the use of functions. If you've used R for anything before, you have likely already used at least one function, but likely many more.

Functions are available from different sources. There are thousands of functions that are readily available in Base-R, that are ready to use without having to load anything else.

There are packages. Every R package is comprised of additional functions created by package authors, and they're a really good place to look for functions that maybe you can't find something that does the same thing in Base R or you're looking for something a little bit more specific or more user friendly or something like that. Then it's also possible to create custom functions. You can create functions that meet your very specific needs.

We'll do a very simple example below. But later in the course, we will talk through more complex examples of creating your own functions.

What are functions used for?

In short, functions are used for almost everything in R!

Some common uses include:

- Creation of new objects - vectors, lists, data frames, visualizations
- Operations on objects - creating a new column, calculations
- Applying logic
- Summarize or describe existing objects

As I said, functions are pretty much used for everything in R. Some common uses would be creating new objects like vectors, lists, data frames, even creating visualizations. Doing operations on objects, like creating a new column, or doing a more complex calculation. Applying logic, something like if then or if else would be a function, and then also summarizing or describing existing objects.

What is the structure of a function?

Understanding the basic structure of a function is important, but becomes more relevant once creating your own functions.

Basic structure:

function_name <- function**(arg1 = 1, arg2 = "Y", ...)** {

  Function body

}

- function_name - should be something short yet descriptive
- arguments (arg1, arg2, etc.) - arguments that are required in order to effectively use the function. Some arguments may be optional, and some functions may not require any arguments.
    - Common use of arguments is to identify the object that you are operating on (vector, list, data frame, column, etc.)
    - Other arguments may be helpful for specifying how the function works (like "options")
    - Many functions will have default values for each argument so that it runs even if not all arguments are defined

Before we get into using functions, it's important to understand the back end structure of a function. Usually, when you use a function, you're just using it, you might not look at the code behind it, and that's fine, but it's helpful to understand how the structure is set up when a function is created.

Here you can see basic structure. If we want to create a new function or for every function we use, at some point, someone did this step, they will declare the function name, and then they'll use the function, saying function a lot to specify that they're creating a new function. Then within the parentheses after function, you can specify arguments, and these arguments will be used in the function body to run desired code. A little bit more about each of those pieces.

The function name should be something short yet descriptive, so it's obvious what the function is doing.

Arguments. You typically wouldn't name them arg1, arg2. You'd name them more meaningfully. But these are arguments that are required in order to effectively use the function. It is possible that some arguments may be optional and some functions may actually not require any arguments, but that's a little more rare.

One common use of arguments is to identify the object you are operating on, like a vector, list, data frame, column. Those are the types of arguments that will typically be required. Other arguments may be helpful for specifying how the function works. You could think of it like options. Some of those options would be maybe more optional, but it really depends function by function.

Then for many functions, they will have default values set for each argument so that it runs even if not all arguments are defined. A default value would just be specified here, so you could say arg1 equals maybe it equals 1 and arg2 equals yes. Those would be the default, but anyone could change that when they use the function themselves. But if they didn't specify arguments, then at least there would be a default value to run through the function.

- function body - this will include R code that is to be run when the function is called
    - This code should be fully self-contained - meaning it does not rely on any previously run code to be successful
    - If there are arguments, they will be referenced within the code chunk

Then the function body is just any R code that is to be run when the function is called. Any operations that are happening on the arguments would be specified here. If we want to print the first argument, we would say print arg1 and automatically whatever supplied for the first argument would be populated there.

To add a couple more things about the function body, this code should really be fully self contained, meaning it does not rely on any previously run code to be successful. Then, again, if there are arguments specified, they will be referenced within the code chunk to tell the function what to do with those arguments.

How are functions called (or used)?

Some functions don't require any arguments, or have default values that don't need to be changed.

function_name()

Some will require argument values to be specified.

function_name(arg_1= x, arg_2 = y)

function_name(x, y)

The functions above will both print results straight to the console. However, the returned value can also be assigned to a new object:

new_object <- function_name(arg_1= x, arg_2 = y)

Now that we know a bit about how they're set up on the back end, R functions called or used. As I mentioned previously, some functions don't require any arguments or they may have default values that don't need to be changed. In that case, we could just call a function by specifying the function name and then open and closed parentheses with nothing inside.

But some will require arguments to be specified. If we wanted to do that, we would call the function name and then within the parentheses, specify the value we want for arg1, which in this case, maybe we're saying x, and for arg2, we want it to be y. You don't have to include the arg1 equals and arg2 equals, as long as you do things in the order that the function is expecting, you could just supply x and y. This is typically okay if there's one or two arguments, but when we get into functions that have a lot more arguments, it's really a best practice to make sure you're specifying which argument equals what, just to be really clear that the function is doing what you want.

If we run either of these fake functions, the results would be printed straight to the console, which there may be a use for that if we're just doing some exploration. It's also possible to return the value to a new object. If we created a new object called new object and then call the function, this would create a new object with the value that's coming out of the function.

Examples - using functions in base R:

Some frequently used functions:

- list() - create a list
- c() - create a vector
- class() - what kind of object is it?
- typeof() - what is the object's data type?
- length() - how long is the object?

Let's talk a little bit more specifically. Some examples of functions that we'll start out using in Base-R are list, which we'll use to create a list. This function that's just a C, it's to create a vector. Then there's some functions, these are just a sampling that can be used to help describe objects. Class tells us what kind of object it is, type of is what's the object data type, and then length could be used to tell how long the object is.

Example - using a function in base R:

Use seq() to generate a numeric sequence from 1 to 20

```
#first let's learn a bit more about the seq function
?seq

#what happens if we don't specify an argument
seq()
```

```
[1] 1
```

Let's try using a function that's available in base R. We're going to use the sequence function or it's seq for short to generate a numeric sequence 1-20. But first, let's learn a little bit more about the function. I'm going to go ahead and run?seq, and you can see over on the right some information is coming up about this function. The function is used to generate regular sequences. It has some default set, which we can see here.

Some of the arguments that you can specify are from, to, by, and length out. From is where our sequence would start, to is where it would end.

By is we can say, we want to go from 1-10 by to, or instead of by, we can specify length out and say, we want to go from 1-10 and have five values in our final sequence, and it will split it evenly.

The values here, from equals 1, to equals 1, our default by has an equation, but essentially it's saying the default is equal to 1.

Let's just see what happens when we run the sequence function without any arguments. So we just print value of one. That's because we're going from 1-1. There's nothing really in between there. You can see if you scroll down over here, there's a bit more information about each of the arguments. We're not going to really pay attention to that along with function. It's not going to be helpful for us in the moment.

There's more information to load that I'm not going to go over, but just an example that there's a lot of documentation for pretty much every function that exists, and if you're wanting to do something new, looking at this documentation is super helpful.

```
#try with arguments
seq(from=1, to=20, by=1)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
seq(from=1, to=20, by=2)
```

```
 [1]  1  3  5  7  9 11 13 15 17 19
```

```
seq(from=1, to=20, length.out=10)
```

```
 [1]  1.000000  3.111111  5.222222  7.333333  9.444444 11.555556 13.666667
 [8] 15.777778 17.888889 20.000000
```

```
a <- 2
b <- 40

new_seq <- seq(from=a,to=b,by=6)
new_seq
```

```
[1]  2  8 14 20 26 32 38
```

```
#let's use some functions to decribe our new object
typeof(new_seq)
```

```
[1] "double"
```

```
class(new_seq)
```

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

```
summary(new_seq)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      2      11      20      20      29      38
```

Let's go back and let's add this function with argument. Our goal is to get a sequence from 1-20. If we just said, we want to start from 1-20 by 1, We would print, we get a vector here that has all the values 1-20 by 1. We could see what that looks like if we change it to by 2. You can see it's giving us every integer by 2 between 1-20. The 20 isn't included because 19 is where it stops.

On the flip side instead of using by, we could try up the length out. So if we said we want a sequence from 1-20 that has 10 values. That's what this is saying. We would get this vector instead. Sequence does work with decimals. That's how we would use sequence.

A couple more things about this. When you're using functions, you can specify using the numbers or whatever exact string object number you're wanting to supply to the function. But functions can also ingest values from other objects.

If we created two objects a with the value of two and b with the value of 40, we could create a new object called new sequence that has a sequence from a-b, so from 2-40 by 6. Let's run that and we see that that doesn't print to the console, but it does show up in our environment. But we can print it by just running the object itself, and we do see a sequence here from 2-40 by 6. Now that we have this new object, we can use other functions to explain it a bit, so if we do type of, this is telling us what type of data is in this vector, which is double.

Another way to say numeric or integer and then class. The object as a whole is numeric, and then summary actually is a handy one that will provide some summary statistics for all of the values in our object. This typically works best if you have multiple entries in your object, which we do, so we can see the min, the max, the median, and mean. Again, a lot of functions with a lot of different helpful tools for us.

Example - creating a simple function:

```
#create a function
add_together <- function(value1 = 1, value2 = 2){

  value1 + value2
}

#use function
add_together()
```

```
[1] 3
```

```
add_together(value1 = 200, value2 = 34)
```

```
[1] 234
```

```
add_together(200,34)
```

```
[1] 234
```

```
add_together(value1 = 200)
```

```
[1] 202
```

Additional resources for learning about specific functions

- Built in RStudio help
- typing ?_keyword_ into the console
- Google/online resources
- rdocumentation.org
    - Example for seq()
- Functions from package libraries
    - Example: read_delim from readr package
        - Package documentation
        - rdocumentation.org

To close, let's create a simple function. So going back to how we create a function, I'm going to create a function called add together. I'm specifying that I'm creating a function by writing function here. Then I'm specifying two arguments, value 1, and value 2. Value 1 has a default of one, and value two has a default of two.

Then within my block of code, I'm just instructing R to add those two values together. Pretty simple function. But if we run that, we see it shows up in our functions header in the environment tab. Then if I run this without specifying any new values, I will get by default three because 1+2 = 3. Then I can change that though. I can specify I want value 1 to equal 200, value 2 to equal 34. I run I get 234.

I can also just provide the numeric values I want for value 1 and value 2. If I do that, I also get 234. Again, I recommend specifying which variable you're changing to what but this is certainly possible and easier to manage if you're using a function that only has one or two arguments.

Then finally, just to demonstrate if you only specify for one value, so say here, we specify value 1 equals 200, but we don't specify value 2, it's going to go ahead and take the default value for value 2, which is 2. We end up with 202. Very simple example of building your own function here. Again, we will talk about this more later in the course.

Lecture 2.8: Packages

Packages

This toolkit will cover what packages are, how they work, how to install them, how to use them, and ways to streamline package management.

What is a package?

R packages are a collection of R functions, compiled code, and sample data. These packages expand the capabilities of base R, and can Improve efficiency or effectiveness of programming.

- Some narrow, some broad
- Some focused on specific industry needs, some focused on refining a specific type of task (import, stats, graphics, etc.)
- Some meant to work with other packages (tidyverse is an example of this)
- Anyone can create a package
- Thousands of packages available
- RStudio does create and maintain several very helpful packages

This is a short toolkit about packages and how they work in the R and RStudio environment, how to install and use them, and a new way to streamline package management.

What is a package? A package is really just a collection of functions that you can use, you're going to install, and load in your session and use to do data tidying, data manipulation, analysis models. Basically, much of what you can do in R, you do through packages. The R programming language has a base R, which are functions that are specific to the programming language. But the packages serve as add-ons for specific things that you can do beyond what is in base R.

There are some that have very narrow scope and some that are extremely broad that you might use every day or every time you use R. Some are focused on specific industry needs or performing a specific type of task in a efficient way.

There are several packages. The tidyverse is a good example of this that are meant to work as a system with other packages. There are thousands and thousands of packages available. Basically, anybody can write a package. There are ways within the R language that you can create a package. Then you can publish those on CRAN, which is the Comprehensive R Archive Network, which we'll talk about in a little bit.

How do packages work in the R environment?

Packages need to be installed and loaded in order to use them in R.

Libraries

Packages are contained within a **library**.

- User library (user or cloud)
- System library (default packages)

There are two main ways to see what packages are already stored in your environment. You can use the "packages" tab in the bottom right pane, or run code. We recommend using code to manage packages – this will be much more efficient as you continue to learn and use R. To see all packages already stored in your environment:

```
# run code below to see packages that are
# available within your environment
library()

#return list with more information to console
# save to an object named "info"
info <- installed.packages()
```

There are a lot of packages already installed because we are using Datahub, which has an extensive list of packages available for student use.

Basically, how packages work in the R environment is you need to have packages installed. They're either installed in the cloud, when we're using DataHub primarily for the course, you will install packages in the cloud.

For those of you who are working on a desktop version, either on a Mac or a PC, those would be installed on your computer and then you load them in a specific session with a library function. There are also packages that you can install your self or there are also system libraries that contain the default packages.

If you're working on DataHub, that's managed by the folks who manage DataHub at Berkeley. They have a number of default packages installed, but then you can also load your own. You can run the code below using the library function to see what packages are available in your environment.

You can also use the Packages tab. Here I will show you where that is, typically on the lower right window and there's a tab called Packages. You can click on that and you can see all of the packages that have been installed that you could potentially load into your session.

You can also create a data frame of your installed packages just by running this code, the install.packages function, and you assign that or that list to an info object.
I will just run that right now. I have created info data frame of all of the packages that I have installed. These are all system library packages, and you can get information about the current version and dependencies, etc.

Finding Packages & Documentation

For this course, we will tell you what packages to use. However, there are many more packages available for various uses in R.

- Many of the packages we use are part of a group of packages called the Tidyverse. These are very commonly used packages for data science.
- A lot of packages have very helpful guides online. For example: dplyr, which is a tidyverse package we will use for data manipulation.
- You can also view a full list of packages available on CRAN
    - All of the packages on CRAN will have a PDF reference manual that documents in detail how to use the package. These all follow the same format. Example: dplyr.
- Within R, you can run the `help()` function to get more information about using a specific package.

```
help(package = "dplyr")
```

There are a number of ways that you can get documentation about packages. That tidyverse system of packages, which has a number of packages that are developed by our studio, now called Posit, they have really nice documentation.

For example, the dplyr package, which I'll click on here, has very nice system of documentation. There homepage is typically a very high-level overview of all that you can do with the package. RStudio also does a cheat sheet for highly used packages that are linked to from the main page.

Then there are usage examples which are usually very helpful for seeing how packages are used. Then there's typically a Get Started link. I'm giving you a little bit more detail about how to start using that package. Then a reference which gets into a lot of depth, basically giving you all of the detail about every function that exists within that package and how to use it. You can click on one of these and see exactly what kind of arguments are included in the function.

Then typically there are good examples at the bottom. RStudio does a really good job of documentation. There's also documentation on CRAN. The CRAN documentation is basically just PDFs source from the package itself. It tends to be definitely has all of the information you need. But if organized a little bit less helpefully, I think I prefer the RStudio documentation if I have a choice. But you can see all of the functions listed here. Click on pages to get more information and they are typically examples as well.

Then thirdly, you can get information about packages by running the help function, which is part of base R. If I wanted to get help information about the dplyr package, I could just run this line here.

In the Help window, usually at the lower right, the documentation, basically it's very similar to what is on CRAN. But the documentation for dplyr shows up there and then you can search through and find answers to your questions.

Installing and Loading Packages

Packages must be installed into your R Studio environment in order to be used. Once a package is installed, you just need to load it each time you use R in order to use the package.

- Use `install.packages()` function to install a package

  Note: Often installing a new package will require the installation of other dependent packages (typically automatic).

```
# you can un-comment the below line to run the code
# install.packages("dplyr")
```

- Use `library()` function to load a package.

  Note: Occasionally, multiple packages will have a function with the same name. Whichever package is loaded last will take priority in this instance, and you will receive a message in the console about these conflicts.

```
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

To install and load a package, those are two tasks that you need to do. Packages must be installed at least once into your RStudio environment in order to be used. You install packages by using the install package function and within the parentheses, you would just include the package that you want to install and then run that. It will install not only the package, but also any dependencies that that package has.

Oftentimes, dplyr will have other packages that it uses and it will install basically everything it needs to run. Then for each session, you will use the library function to load that package. In order to actually use dplyr, I would need to go in basically at the top of any code that I'm running. I load all of the libraries, all of the packages that I want to use for that session. Keeping them at the top just helps me organize things so I know what's all getting installed and what is not.

In order to use dplyr, basically I have to use that library function in order to load that package. Occasionally, there are multiple packages that use the same function. Select is dplyr function, but select is also a function that is used widely by other packages. Whichever package is loaded last will take priority. If you know that there's a conflict, you can put the package for which you're using the function most frequently later in the list.

Or the other thing you can do is reference both the package and the function whenever you're calling that function. If I want to make sure that the select function that I'm using is coming out of the dplyr package, I can reference it by referencing the package, dplyr. Then I use double colon to get all of the dplyr packages, and then I can select. Then I know that it's actually the select function in dplyr that's getting called and not just basing it on what I load last.

Updating Packages

If one or more packages needs to be updated, you have a few options.

- Freshly install the package from CRAN using `install.packages()`. This works well when you only need to update one package.
- Use the `update.packages()` function to update all packages in your library. This works well if you need to update many packages.

```
# you can un-comment the below line to run the code
# update.packages()
```

- Alternately, you can check which packages need updating and update only those using `update.packages()`.

```
# return list of packages needing an update
# you can un-comment the below line to run the code
# old_list <- old.packages()

# update only old package2
# you can un-comment the below line to run the code
# update.packages(oldPkgs = old_list, ask = FALSE)
```

- Another often preferred option is to use a package management tool (detailed below)

For updating packages, package authors and RStudio are updating tidyverse packages all the time to bring new functionality, to fix bugs, et cetera. It's a good idea. Unless you have very specific need for a package at a point in time, it's usually a good idea to update them regularly. You can do that individually with each package by just rerunning the install packages in quote.

If I wanted to get an update on dplyr, I would just do install.packages dplyr. Again, I did this at the beginning when I initially installed it, and now running it again will re-install with the more recent version. Or the other thing you can do, if you just want to update all of the installed packages, is just run the update.packages function. It will go through your entire library, identify what packages need updating, and then run those updates.

Alternatively, you can just get the list of packages that are out of date by using this old.packages function. Here I'm going to run that to get our data frame of all of the packages that are out of date in this data frame called old_list.

Then I can run the update.packages function specifying only those packages in the old_list. Then adding this ask equal false just prevents it. Otherwise, you'll get a message for every package asking that you want to update it. Just ask equal false will prevent that message from occurring over and over again.

Package Management

You need to install packages in each computing environment you use, load them each time you want to use them, and packages frequently need to be updated. This can become a lot to manage if you are switching computing environments or working with a group. Luckily, there is also a package to make this easier!

Using pacman

This package is a package management tool that streamlines multiple above steps for you.

1. Make sure pacman is installed in the computing environment you are using. This code checks if the package pacman is already installed. If it's not installed, the code then installs it.

```
        if (!require("pacman")) install.packages("pacman")
```

```
Loading required package: pacman
```

```
        library(pacman)
```

2. Use the `p_load()` function to install and load packages. This checks to see if a package is installed, installs it if it is not installed, and loads it. The `update = TRUE` option will updated all out of date packages.

```
        p_load(dplyr, ggplot2, purrr, update = TRUE)
```

```
The downloaded binary packages are in
    /var/folders/h_/d54d41nn1g565y9k5kwdvp_w0000gn/T//RtmpfZucpu/downloaded_packages
```

This code becomes very helpful when you are using a lot of packages. It also allows you to automate the process of package install, loading, and updating. For example – you can just put the code in this section at the top of all of your code files, and you will then have current versions of all the packages listed installed and loaded wherever you run your code.

Another preferred option that we're using in this course is to use the pacman package. The pacman package is basically a package management tool that allows you to streamline much of what I talked about earlier.

You can make sure that the pacman package is installed in the computing environment that you're using. Using this code here checks to make sure that that package is installed. It's basically saying if the package pacman is not installed, it's what this exclamation point indicates. If that is not installed, then go ahead and install it and then load that into my current session.

Then what pacman does is basically does all of those things that we talked about automatically. You can use this p_load function and list out all of the packages that you want to install for the session.

Then if you want to update them, you would just include this variable update equals true. Then it will go through, identify whether dplyr is out of date, whether ggplot is out of date, and whether purrr is out of date and install updates if they need them. Otherwise, it will just load them into your session.

The other thing that this does is puts all of the package load on one line so you don't have to have multiple lines of libraries over and over again, which tidies up your code quite a bit. Then it just manages that automatically so that, when you start a new session, that runs and you don't have to worry about out-of-date packages. Thank you.

Introduction

Data science is an exciting discipline that allows you to transform raw data into understanding, insight, and knowledge. The goal of "R for Data Science" is to help you learn the most important tools in R that will allow you to do data science efficiently and reproducibly, and to have some fun along the way 😃 . After reading this book, you'll have the tools to tackle a wide variety of data science challenges using the best parts of R.

What you will learn

Data science is a vast field, and there's no way you can master it all by reading a single book. This book aims to give you a solid foundation in the most important tools and enough knowledge to find the resources to learn more when necessary. Our model of the steps of a typical data science project looks something like Figure 1.

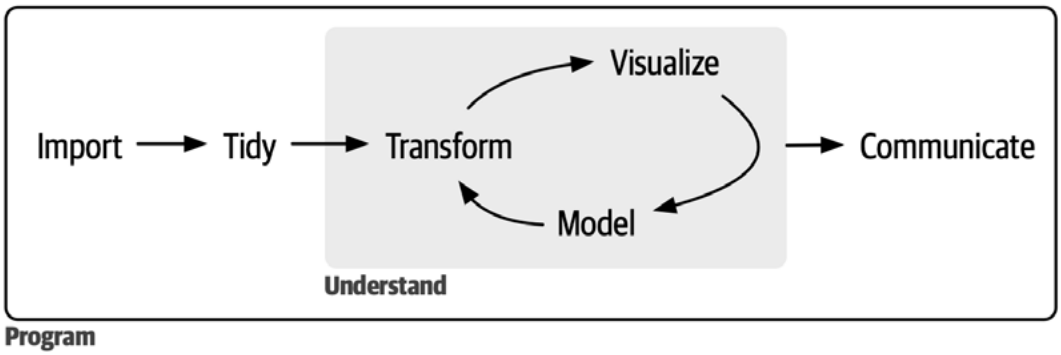


Figure 1: In our model of the data science process, you start with data import and tidying. Next, you understand your data with an iterative cycle of transforming, visualizing, and modeling. You finish the process by communicating your results to other humans.

First, you must **import** your data into R. This typically means that you take data stored in a file, database, or web application programming interface (API) and load it into a data frame in R. If you can't get your data into R, you can't do data science on it!

Once you've imported your data, it is a good idea to **tidy** it. Tidying your data means storing it in a consistent form that matches the semantics of the dataset with how it is stored. In brief, when your data is tidy, each column is a variable and each row is an observation. Tidy data is important because the consistent structure lets you focus your efforts on answering questions about the data, not fighting to get the data into the right form for different functions.

Once you have tidy data, a common next step is to **transform** it. Transformation includes narrowing in on observations of interest (like all people in one city or all data from the last year), creating new variables that are functions of existing variables (like computing speed from distance and time), and calculating a set of summary statistics (like counts or means). Together, tidying and transforming are called **wrangling** because getting your data in a form that's natural to work with often feels like a fight!

Once you have tidy data with the variables you need, there are two main engines of knowledge generation: visualization and modeling. These have complementary strengths and weaknesses, so any real data analysis will iterate between them many times.

Visualization is a fundamentally human activity. A good visualization will show you things you did not expect or raise new questions about the data. A good visualization might also hint that you're asking the wrong question or that you need to collect different data. Visualizations can surprise you, but they don't scale particularly well because they require a human to interpret them.

Models are complementary tools to visualization. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are fundamentally mathematical or computational tools, so they generally scale well. Even when they don't, it's usually cheaper to buy more computers than it is to buy more brains! But every model makes assumptions, and by its very nature, a model cannot question its own assumptions. That means a model cannot fundamentally surprise you.

The last step of data science is **communication**, an absolutely critical part of any data analysis project. It doesn't matter how well your models and visualization have led you to understand the data unless you can also communicate your results to others.

Surrounding all these tools is **programming**. Programming is a cross-cutting tool that you use in nearly every part of a data science project. You don't need to be an expert programmer to be a successful data scientist, but learning more about programming pays off because becoming a better programmer allows you to automate common tasks and solve new problems with greater ease.

You'll use these tools in every data science project, but they're not enough for most projects. There's a rough 80/20 rule at play: you can tackle about 80% of every project using the tools you'll learn in this book, but you'll need other tools to tackle the remaining 20%. Throughout this book, we'll point you to resources where you can learn more.

How this book is organized

The previous description of the tools of data science is organized roughly according to the order in which you use them in an analysis (although, of course, you'll iterate through them multiple times). In our experience, however, learning data importing and tidying first is suboptimal because, 80% of the time, it's routine and boring, and the other 20% of the time, it's weird and frustrating. That's a bad place to start learning a new subject! Instead, we'll start with visualization and transformation of data that's already been imported and tidied. That way, when you ingest and tidy your own data, your motivation will stay high because you know the pain is worth the effort.

Within each chapter, we try to adhere to a consistent pattern: start with some motivating examples so you can see the bigger picture, and then dive into the details. Each section of the book is paired with exercises to help you practice what you've learned. Although it can be tempting to skip the exercises, there's no better way to learn than by practicing on real problems.

What you won't learn

There are several important topics that this book doesn't cover. We believe it's important to stay ruthlessly focused on the essentials so you can get up and running as quickly as possible. That means this book can't cover every important topic.

Modeling

Modeling is super important for data science, but it's a big topic, and unfortunately, we just don't have the space to give it the coverage it deserves here. To learn more about modeling, we highly recommend Tidy Modeling with R by our colleagues Max Kuhn and Julia Silge. This book will teach you the tidymodels family of packages, which, as you might guess from the name, share many conventions with the tidyverse packages we use in this book.

Big data

This book proudly and primarily focuses on small, in-memory datasets. This is the right place to start because you can't tackle big data unless you have experience with small data. The tools you'll learn throughout the majority of this book will easily handle hundreds of megabytes of data, and with a bit of care, you can typically use them to work with a few gigabytes of data. We'll also show you how to get data out of databases and parquet files, both of which are often used to store big data. You won't necessarily be able to work with the entire dataset, but that's not a problem because you only need a subset or subsample to answer the question that you're interested in.

If you're routinely working with larger data (10–100 GB, say), we recommend learning more about data.table. We don't teach it here because it uses a different interface than the tidyverse and requires you to learn some different conventions. However, it is incredibly faster, and the performance payoff is worth investing some time in learning it if you're working with large data.

Python, Julia, and friends

In this book, you won't learn anything about Python, Julia, or any other programming language useful for data science. This isn't because we think these tools are bad. They're not! And in practice, most data science teams use a mix of languages, often at least R and Python. But we strongly believe that it's best to master one tool at a time, and R is a great place to start.

Prerequisites

We've made a few assumptions about what you already know to get the most out of this book. You should be generally numerically literate, and it's helpful if you have some basic programming experience already. If you've never programmed before, you might find Hands on Programming with R by Garrett to be a valuable adjunct to this book.

You need four things to run the code in this book: R, RStudio, a collection of R packages called the **tidyverse**, and a handful of other packages. Packages are the fundamental units of reproducible R code. They include reusable functions, documentation that describes how to use them, and sample data.

R

To download R, go to CRAN, the **c**omprehensive **R** **a**rchive **n**etwork, https://cloud.r-project.org. A new major version of R comes out once a year, and there are 2-3 minor releases each year. It's a good idea to update regularly. Upgrading can be a bit of a hassle, especially for major versions that require you to re-install all your packages, but putting it off only makes it worse. We recommend R 4.2.0 or later for this book.

RStudio

RStudio is an integrated development environment, or IDE, for R programming, which you can download from https://posit.co/download/rstudio-desktop/. RStudio is updated a couple of times a year, and it will automatically let you know when a new version is out, so there's no need to check back. It's a good idea to upgrade regularly to take advantage of the latest and greatest features. For this book, make sure you have at least RStudio 2022.02.0.

When you start RStudio, Figure 2, you'll see two key regions in the interface: the console pane and the output pane. For now, all you need to know is that you type the R code in the console pane and press enter to run it. You'll learn more as we go along![1]

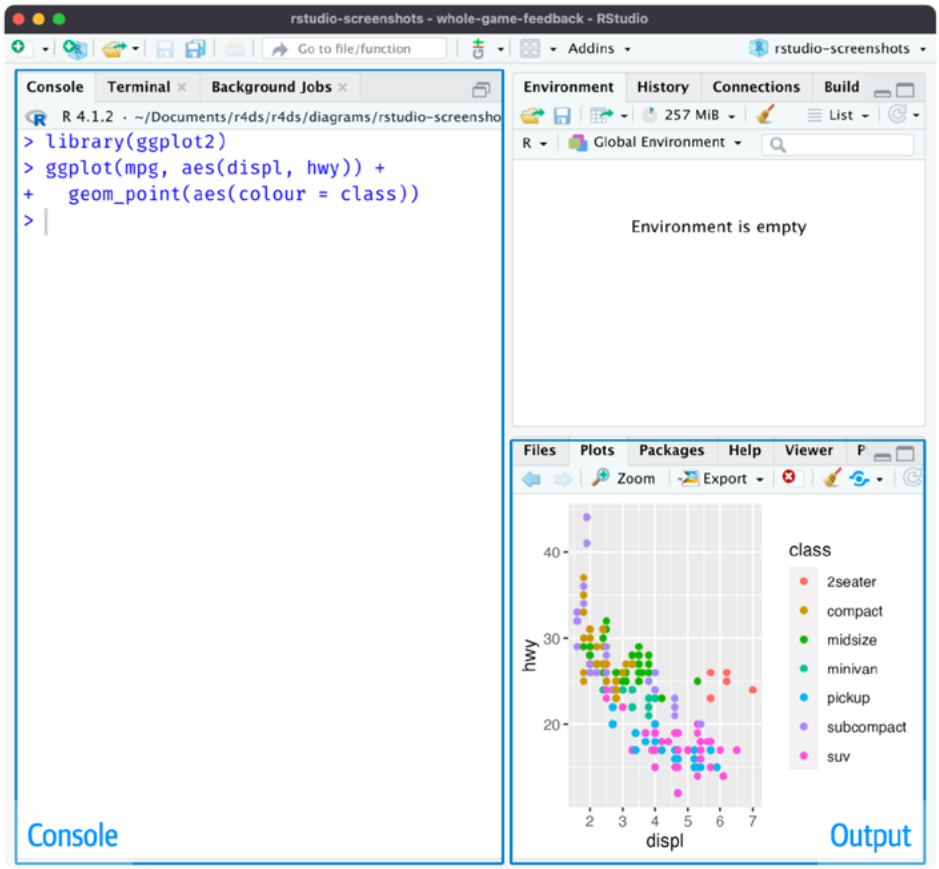


Figure 2: The RStudio IDE has two key regions: type R code in the console pane on the left, and look for plots in the output pane on the right.

The tidyverse

You'll also need to install some R packages. An R **package** is a collection of functions, data, and documentation that extends the capabilities of base R. Using packages is key to the successful use of R. The majority of the

packages that you will learn in this book are part of the so-called tidyverse. All packages in the tidyverse share a common philosophy of data and R programming and are designed to work together.

You can install the complete tidyverse with a single line of code:

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

On your computer, type that line of code in the console, and then press enter to run it. R will download the packages from CRAN and install them on your computer.

You will not be able to use the functions, objects, or help files in a package until you load it with `library()`. Once you have installed a package, you can load it using the `library()` function:

```
library(tidyverse)
#> ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
#> ✔ dplyr     1.1.4     ✔ readr     2.1.5
#> ✔ forcats   1.0.0     ✔ stringr   1.5.1
#> ✔ ggplot2   3.5.1     ✔ tibble    3.2.1
#> ✔ lubridate 1.9.3     ✔ tidyr     1.3.1
#> ✔ purrr     1.0.2
#> ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
#> ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts t
```

This tells you that tidyverse loads nine packages: dplyr, forcats, ggplot2, lubridate, purrr, readr, stringr, tibble, tidyr. These are considered the **core** of the tidyverse because you'll use them in almost every analysis.

Packages in the tidyverse change fairly frequently. You can see if updates are available by running `tidyverse_update()`.

Other packages

There are many other excellent packages that are not part of the tidyverse because they solve problems in a different domain or are designed with a different set of underlying principles. This doesn't make them better or worse; it just makes them different. In other words, the complement to the tidyverse is not the messyverse but many other universes of interrelated packages. As you tackle more data science projects with R, you'll learn new packages and new ways of thinking about data.

We'll use many packages from outside the tidyverse in this book. For example, we'll use the following packages because they provide interesting datasets for us to work with in the process of learning R:

```
install.packages(
  c("arrow", "babynames", "curl", "duckdb", "gapminder",
    "ggrepel", "ggridges", "ggthemes", "hexbin", "janitor", "Lahman",
    "leaflet", "maps", "nycflights13", "openxlsx", "palmerpenguins",
```

```
    "repurrrsive", "tidymodels", "writexl")
  )
```

We'll also use a selection of other packages for one off examples. You don't need to install them now, just remember that whenever you see an error like this:

```
library(ggrepel)
#> Error in library(ggrepel) : there is no package called 'ggrepel'
```

You need to run `install.packages("ggrepel")` to install the package.

Running R code

The previous section showed you several examples of running R code. The code in the book looks like this:

```
1 + 2
#> [1] 3
```

If you run the same code in your local console, it will look like this:

```
> 1 + 2
[1] 3
```

There are two main differences. In your console, you type after the `>`, called the **prompt**; we don't show the prompt in the book. In the book, the output is commented out with `#>`; in your console, it appears directly after your code. These two differences mean that if you're working with an electronic version of the book, you can easily copy code out of the book and paste it into the console.

Throughout the book, we use a consistent set of conventions to refer to code:

- Functions are displayed in a code font and followed by parentheses, like `sum()` or `mean()`.
- Other R objects (such as data or function arguments) are in a code font, without parentheses, like `flights` or `x`.
- Sometimes, to make it clear which package an object comes from, we'll use the package name followed by two colons, like `dplyr::mutate()` or `nycflights13::flights`. This is also valid R code.

Acknowledgments

This book isn't just the product of Hadley, Mine, and Garrett but is the result of many conversations (in person and online) that we've had with many people in the R community. We're incredibly grateful for all the conversations we've had with y'all; thank you so much!

This book was written in the open, and many people contributed via pull requests. A special thanks to all 259 of you who contributed improvements via GitHub pull requests (in alphabetical order by username): @a-rosenberg, Tim Becker (@a2800276), Abinash Satapathy (@Abinashbunty), Adam Gruer (@adam-gruer), adi pradhan

(@adidoit), A. s. (@Adrianzo), Aep Hidyatuloh (@aephidayatuloh), Andrea Gilardi (@agila5), Ajay Deonarine (@ajay-d), @AlanFeder, Daihe Sui (@alansuidaihe), @alberto-agudo, @AlbertRapp, @aleloi, pete (@alonzi), Alex (@ALShum), Andrew M. (@amacfarland), Andrew Landgraf (@andland), @andyhuynh92, Angela Li (@angela-li), Antti Rask (@AnttiRask), LOU Xun (@aquarhead), @ariespirgel, @august-18, Michael Henry (@aviast), Azza Ahmed (@azzaea), Steven Moran (@bambooforest), Brian G. Barkley (@BarkleyBG), Mara Averick (@batpigandme), Oluwafemi OYEDELE (@BB1464), Brent Brewington (@bbrewington), Bill Behrman (@behrman), Ben Herbertson (@benherbertson), Ben Marwick (@benmarwick), Ben Steinberg (@bensteinberg), Benjamin Yeh (@bentyeh), Betul Turkoglu (@betulturkoglu), Brandon Greenwell (@bgreenwell), Bianca Peterson (@BinxiePeterson), Birger Niklas (@BirgerNi), Brett Klamer (@bklamer), @boardtc, Christian (@c-hoh), Caddy (@caddycarine), Camille V Leonard (@camillevleonard), @canovasjm, Cedric Batailler (@cedricbatailler), Christina Wei (@christina-wei), Christian Mongeau (@chrMongeau), Cooper Morris (@coopermor), Colin Gillespie (@csgillespie), Rademeyer Vermaak (@csrvermaak), Chloe Thierstein (@cthierst), Chris Saunders (@ctsa), Abhinav Singh (@curious-abhinav), Curtis Alexander (@curtisalexander), Christian G. Warden (@cwarden), Charlotte Wickham (@cwickham), Kenny Darrell (@darrkj), David Kane (@davidkane9), David (@davidrsch), David Rubinger (@davidrubinger), David Clark (@DDClark), Derwin McGeary (@derwinmcgeary), Daniel Gromer (@dgromer), @Divider85, @djbirke, Danielle Navarro (@djnavarro), Russell Shean (@DOH-RPS1303), Zhuoer Dong (@dongzhuoer), Devin Pastoor (@dpastoor), @DSGeoff, Devarshi Thakkar (@dthakkar09), Julian During (@duju211), Dylan Cashman (@dylancashman), Dirk Eddelbuettel (@eddelbuettel), Edwin Thoen (@EdwinTh), Ahmed El-Gabbas (@elgabbas), Henry Webel (@enryH), Ercan Karadas (@ercan7), Eric Kitaif (@EricKit), Eric Watt (@ericwatt), Erik Erhardt (@erikerhardt), Etienne B. Racine (@etiennebr), Everett Robinson (@evjrob), @fellennert, Flemming Miguel (@flemmingmiguel), Floris Vanderhaeghe (@florisvdh), @funkybluehen, @gabrivera, Garrick Aden-Buie (@gadenbuie), Peter Ganong (@ganong123), Gerome Meyer (@GeroVanMi), Gleb Ebert (@gl-eb), Josh Goldberg (@GoldbergData), bahadir cankardes (@gridgrad), Gustav W Delius (@gustavdelius), Hao Chen (@hao-trivago), Harris McGehee (@harrismcgehee), @hendrikweisser, Hengni Cai (@hengnicai), Iain (@Iain-S), Ian Sealy (@iansealy), Ian Lyttle (@ijlyttle), Ivan Krukov (@ivan-krukov), Jacob Kaplan (@jacobkap), Jazz Weisman (@jazzlw), John Blischak (@jdblischak), John D. Storey (@jdstorey), Gregory Jefferis (@jefferis), Jeffrey Stevens (@JeffreyRStevens), 蒋雨蒙 (@JeldorPKU), Jennifer (Jenny) Bryan (@jennybc), Jen Ren (@jenren), Jeroen Janssens (@jeroenjanssens), @jeromecholewa, Janet Wesner (@jilmun), Jim Hester (@jimhester), JJ Chen (@jjchern), Jacek Kolacz (@jkolacz), Joanne Jang (@joannejang), @johannes4998, John Sears (@johnsears), @jonathanflint, Jon Calder (@jonmcalder), Jonathan Page (@jonpage), Jon Harmon (@jonthegeek), JooYoung Seo (@jooyoungseo), Justinas Petuchovas (@jpetuchovas), Jordan (@jrdnbradford), Jeffrey Arnold (@jrnold), Jose Roberto Ayala Solares (@jroberayalas), Joyce Robbins (@jtr13), @juandering, Julia Stewart Lowndes (@jules32), Sonja (@kaetschap), Kara Woo (@karawoo), Katrin Leinweber (@katrinleinweber), Karandeep Singh (@kdpsingh), Kevin Perese (@kevinxperese), Kevin Ferris (@kferris10), Kirill Sevastyanenko (@kirillseva), Jonathan Kitt (@KittJonathan), @koalabearski, Kirill Müller (@krlmlr), Rafał Kucharski (@kucharsky), Kevin Wright (@kwstat), Noah Landesberg (@landesbergn), Lawrence Wu (@lawwu), @lindbrook, Luke W Johnston (@lwjohnst86), Kara de la Marck (@MarckK), Kunal Marwaha (@marwahaha), Matan Hakim (@matanhakim), Matthias Liew (@MatthiasLiew), Matt Wittbrodt (@MattWittbrodt), Mauro Lepore (@maurolepore), Mark Beveridge (@mbeveridge), @mcewenkhundi, mcsnowface, PhD (@mcsnowface), Matt Herman (@mfherman), Michael Boerman (@michaelboerman), Mitsuo Shiota (@mitsuoxv), Matthew Hendrickson (@mjhendrickson), @MJMarshall, Misty Knight-Finley (@mkfin7), Mohammed Hamdy (@mmhamdy), Maxim Nazarov (@mnazarov), Maria Paula Caldas (@mpaulacaldas), Mustafa Ascha (@mustafaascha), Nelson Areal (@nareal), Nate Olson (@nate-d-olson), Nathanael (@nateaff), @nattalides, Ned Western (@NedJWestern), Nick Clark (@nickclark1000), @nickelas, Nirmal Patel (@nirmalpatel), Nischal Shrestha (@nischalshrestha), Nicholas Tierney (@njtierney), Jakub Nowosad (@Nowosad), Nick Pullen (@nstjhp),

@olivier6088, Olivier Cailloux (@oliviercailloux), Robin Penfold (@p0bs), Pablo E. Garcia (@pabloedug), Paul Adamson (@padamson), Penelope Y (@penelopeysm), Peter Hurford (@peterhurford), Peter Baumgartner (@petzi53), Patrick Kennedy (@pkq), Pooya Taherkhani (@pooyataher), Y. Yu (@PursuitOfDataScience), Radu Grosu (@radugrosu), Ranae Dietzel (@Ranae), Ralph Straumann (@rastrau), Rayna M Harris (@raynamharris), @ReeceGoding, Robin Gertenbach (@rgertenbach), Jajo (@RIngyao), Riva Quiroga (@rivaquiroga), Richard Knight (@RJHKnight), Richard Zijdeman (@rlzijdeman), @robertchu03, Robin Kohrs (@RobinKohrs), Robin (@Robinlovelace), Emily Robinson (@robinsones), Rob Tenorio (@robtenorio), Rod Mazloomi (@RodAli), Rohan Alexander (@RohanAlexander), Romero Morais (@RomeroBarata), Albert Y. Kim (@rudeboybert), Saghir (@saghirb), Hojjat Salmasian (@salmasian), Jonas (@sauercrowd), Vebash Naidoo (@sciencificity), Seamus McKinsey (@seamus-mckinsey), @seanpwilliams, Luke Smith (@seasmith), Matthew Sedaghatfar (@sedaghatfar), Sebastian Kraus (@sekR4), Sam Firke (@sfirke), Shannon Ellis (@ShanEllis), @shoili, Christian Heinrich (@Shurakai), S'busiso Mkhondwane (@sibusiso16), SM Raiyyan (@sm-raiyyan), Jakob Krigovsky (@sonicdoe), Stephan Koenig (@stephan-koenig), Stephen Balogun (@stephenbalogun), Steven M. Mortimer (@StevenMMortimer), Stéphane Guillou (@stragu), Sulgi Kim (@sulgik), Sergiusz Bleja (@svenski), Tal Galili (@talgalili), Alec Fisher (@Taurenamo), Todd Gerarden (@tgerarden), Tom Godfrey (@thomasggodfrey), Tim Broderick (@timbroderick), Tim Waterhouse (@timwaterhouse), TJ Mahr (@tjmahr), Thomas Klebel (@tklebel), Tom Prior (@tomjamesprior), Terence Teo (@tteo), @twgardner2, Ulrik Lyngs (@ulyngs), Shinya Uryu (@uribo), Martin Van der Linden (@vanderlindenma), Walter Somerville (@waltersom), @werkstattcodes, Will Beasley (@wibeasley), Yihui Xie (@yihui), Yiming (Paul) Li (@yimingli), @yingxingwu, Hiroaki Yutani (@yutannihilation), Yu Yu Aung (@yuyu-aung), Zach Bogart (@zachbogart), @zeal626, Zeki Akyol (@zekiakyol).

Colophon

An online version of this book is available at https://r4ds.hadley.nz. It will continue to evolve in between reprints of the physical book. The source of the book is available at https://github.com/hadley/r4ds. The book is powered by Quarto, which makes it easy to write books that combine text and executable code.

1. If you'd like a comprehensive overview of all of RStudio's features, see the RStudio User Guide at https://docs.posit.co/ide/user. ↩

R for Data Science (2e) was written by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund. This book was built with Quarto.

2 Workflow: basics

You now have some experience running R code. We didn't give you many details, but you've obviously figured out the basics, or you would've thrown this book away in frustration! Frustration is natural when you start programming in R because it is such a stickler for punctuation, and even one character out of place can cause it to complain. But while you should expect to be a little frustrated, take comfort in that this experience is typical and temporary: it happens to everyone, and the only way to get over it is to keep trying.

Before we go any further, let's ensure you've got a solid foundation in running R code and that you know some of the most helpful RStudio features.

2.1 Coding basics

Let's review some basics we've omitted so far in the interest of getting you plotting as quickly as possible. You can use R to do basic math calculations:

```
1 / 200 * 30
#> [1] 0.15
(59 + 73 + 2) / 3
#> [1] 44.66667
sin(pi / 2)
#> [1] 1
```

You can create new objects with the assignment operator `<-`:

```
x <- 3 * 4
```

Note that the value of `x` is not printed, it's just stored. If you want to view the value, type `x` in the console.

You can **c**ombine multiple elements into a vector with `c()`:

```
primes <- c(2, 3, 5, 7, 11, 13)
```

And basic arithmetic on vectors is applied to every element of of the vector:

```
primes * 2
#> [1]  4  6 10 14 22 26
primes - 1
#> [1]  1  2  4  6 10 12
```

All R statements where you create objects, **assignment** statements, have the same form:

```
object_name <- value
```

When reading that code, say "object name gets value" in your head.

You will make lots of assignments, and `<-` is a pain to type. You can save time with RStudio's keyboard shortcut: Alt + - (the minus sign). Notice that RStudio automatically surrounds `<-` with spaces, which is a good code formatting practice. Code can be miserable to read on a good day, so giveyoureyesabreak and use spaces.

2.2 Comments

R will ignore any text after `#` for that line. This allows you to write **comments**, text that is ignored by R but read by other humans. We'll sometimes include comments in examples explaining what's happening with the code.

Comments can be helpful for briefly describing what the following code does.

```
# create vector of primes
primes <- c(2, 3, 5, 7, 11, 13)

# multiply primes by 2
primes * 2
#> [1]  4  6 10 14 22 26
```

With short pieces of code like this, leaving a comment for every single line of code might not be necessary. But as the code you're writing gets more complex, comments can save you (and your collaborators) a lot of time figuring out what was done in the code.

Use comments to explain the *why* of your code, not the *how* or the *what*. The *what* and *how* of your code are always possible to figure out, even if it might be tedious, by carefully reading it. If you describe every step in the comments, and then change the code, you will have to remember to update the comments as well or it will be confusing when you return to your code in the future.

Figuring out *why* something was done is much more difficult, if not impossible. For example, `geom_smooth()` has an argument called `span`, which controls the smoothness of the curve, with larger values yielding a smoother curve. Suppose you decide to change the value of `span` from its default of 0.75 to 0.9: it's easy for a future reader to understand *what* is happening, but unless you note your thinking in a comment, no one will understand *why* you changed the default.

For data analysis code, use comments to explain your overall plan of attack and record important insights as you encounter them. There's no way to re-capture this knowledge from the code itself.

2.3 What's in a name?

Object names must start with a letter and can only contain letters, numbers, `_`, and `.`. You want your object names to be descriptive, so you'll need to adopt a convention for multiple words. We recommend **snake_case**, where you separate lowercase words with `_`.

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

We'll return to names again when we discuss code style in Chapter 4.

You can inspect an object by typing its name:

```
x
#> [1] 12
```

Make another assignment:

```
this_is_a_really_long_name <- 2.5
```

To inspect this object, try out RStudio's completion facility: type "this", press TAB, add characters until you have a unique prefix, then press return.

Let's assume you made a mistake, and that the value of `this_is_a_really_long_name` should be 3.5, not 2.5. You can use another keyboard shortcut to help you fix it. For example, you can press ↑ to bring the last command you typed and edit it. Or, type "this" then press Cmd/Ctrl + ↑ to list all the commands you've typed that start with those letters. Use the arrow keys to navigate, then press enter to retype the command. Change 2.5 to 3.5 and rerun.

Make yet another assignment:

```
r_rocks <- 2^3
```

Let's try to inspect it:

```
r_rock
#> Error: object 'r_rock' not found
R_rocks
#> Error: object 'R_rocks' not found
```

This illustrates the implied contract between you and R: R will do the tedious computations for you, but in exchange, you must be completely precise in your instructions. If not, you're likely to get an error that says the object you're looking for was not found. Typos matter; R can't read your mind and say, "oh, they probably meant `r_rocks` when they typed `r_rock`". Case matters; similarly, R can't read your mind and say, "oh, they probably meant `r_rocks` when they typed `R_rocks`".

2.4 Calling functions

R has a large collection of built-in functions that are called like this:

```
function_name(argument1 = value1, argument2 = value2, ...)
```

Let's try using `seq()`, which makes regular **seq**uences of numbers, and while we're at it, learn more helpful features of RStudio. Type `se` and hit TAB. A popup shows you possible completions. Specify `seq()` by typing more (a `q`) to disambiguate or by using ↑/↓ arrows to select. Notice the floating tooltip that pops up, reminding you of the function's arguments and purpose. If you want more help, press F1 to get all the details in the help tab in the lower right pane.

When you've selected the function you want, press TAB again. RStudio will add matching opening (`(`) and closing (`)`) parentheses for you. Type the name of the first argument, `from`, and set it equal to `1`. Then, type the name of the second argument, `to`, and set it equal to `10`. Finally, hit return.

```
seq(from = 1, to = 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

We often omit the names of the first several arguments in function calls, so we can rewrite this as follows:

```
seq(1, 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Type the following code and notice that RStudio provides similar assistance with the paired quotation marks:

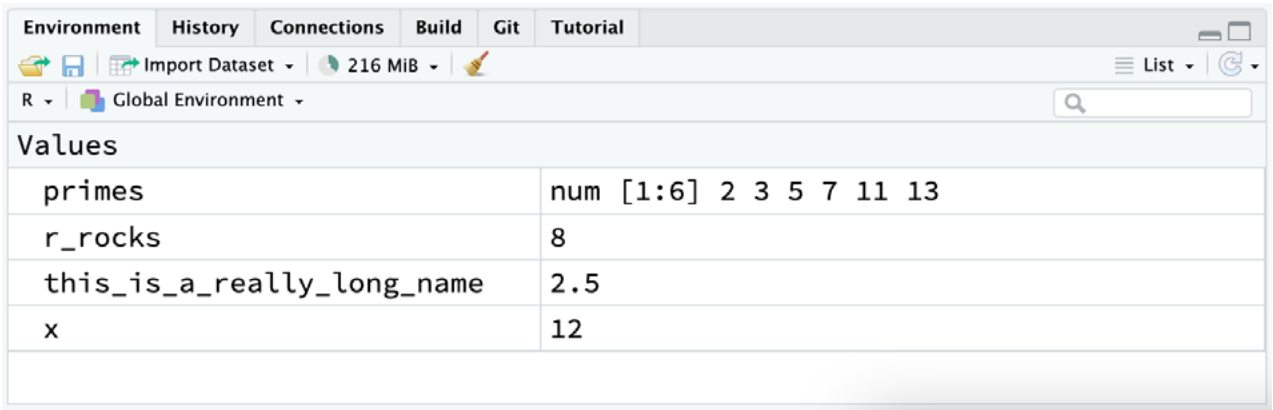
```
x <- "hello world"
```

Quotation marks and parentheses must always come in a pair. RStudio does its best to help you, but it's still possible to mess up and end up with a mismatch. If this happens, R will show you the continuation character "+":

```
> x <- "hello
+
```

The `+` tells you that R is waiting for more input; it doesn't think you're done yet. Usually, this means you've forgotten either a `"` or a `)`. Either add the missing pair, or press ESCAPE to abort the expression and try again.

Note that the environment tab in the upper right pane displays all of the objects that you've created:



2.5 Exercises

1. Why does this code not work?

```
my_variable <- 10
my_varıable
#> Error in eval(expr, envir, enclos): object 'my_varıable' not found
```

   Look carefully! (This may seem like an exercise in pointlessness, but training your brain to notice even the tiniest difference will pay off when programming.)

2. Tweak each of the following R commands so that they run correctly:

```
libary(todyverse)

ggplot(dTA = mpg) +
  geom_point(maping = aes(x = displ y = hwy)) +
  geom_smooth(method = "lm)
```

3. Press Option + Shift + K / Alt + Shift + K. What happens? How can you get to the same place using the menus?

4. Let's revisit an exercise from the Section 1.6. Run the following lines of code. Which of the two plots is saved as `mpg-plot.png`? Why?

```
my_bar_plot <- ggplot(mpg, aes(x = class)) +
  geom_bar()
my_scatter_plot <- ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point()
ggsave(filename = "mpg-plot.png", plot = my_bar_plot)
```

2.6 Summary

Now that you've learned a little more about how R code works, and some tips to help you understand your code when you come back to it in the future. In the next chapter, we'll continue your data science journey by teaching you about dplyr, the tidyverse package that helps you transform data, whether it's selecting important variables, filtering down to rows of interest, or computing summary statistics.

28 Quarto

28.1 Introduction

Quarto provides a unified authoring framework for data science, combining your code, its results, and your prose. Quarto documents are fully reproducible and support dozens of output formats, like PDFs, Word files, presentations, and more.

Quarto files are designed to be used in three ways:

1. For communicating to decision-makers, who want to focus on the conclusions, not the code behind the analysis.
2. For collaborating with other data scientists (including future you!), who are interested in both your conclusions, and how you reached them (i.e. the code).
3. As an environment in which to *do* data science, as a modern-day lab notebook where you can capture not only what you did, but also what you were thinking.

Quarto is a command line interface tool, not an R package. This means that help is, by-and-large, not available through `?`. Instead, as you work through this chapter, and use Quarto in the future, you should refer to the Quarto documentation.

If you're an R Markdown user, you might be thinking "Quarto sounds a lot like R Markdown". You're not wrong! Quarto unifies the functionality of many packages from the R Markdown ecosystem (rmarkdown, bookdown, distill, xaringan, etc.) into a single consistent system as well as extends it with native support for multiple programming languages like Python and Julia in addition to R. In a way, Quarto reflects everything that was learned from expanding and supporting the R Markdown ecosystem over a decade.

28.1.1 Prerequisites

You need the Quarto command line interface (Quarto CLI), but you don't need to explicitly install it or load it, as RStudio automatically does both when needed.

28.2 Quarto basics

This is a Quarto file – a plain text file that has the extension `.qmd`:

```
---
title: "Diamond sizes"
date: 2022-09-12
format: html
```

````
---

```{r}
#| label: setup
#| include: false

library(tidyverse)

smaller <- diamonds |>
  filter(carat <= 2.5)
```

We have data about `r nrow(diamonds)` diamonds.
Only `r nrow(diamonds) - nrow(smaller)` are larger than 2.5 carats.
The distribution of the remainder is shown below:

```{r}
#| label: plot-smaller-diamonds
#| echo: false

smaller |>
  ggplot(aes(x = carat)) +
  geom_freqpoly(binwidth = 0.01)
```
````

It contains three important types of content:

1. An (optional) **YAML header** surrounded by `---`s.
2. **Chunks** of R code surrounded by ```` ``` ````.
3. Text mixed with simple text formatting like `# heading` and `_italics_`.

Figure 28.1 shows a `.qmd` document in RStudio with notebook interface where code and output are interleaved. You can run each code chunk by clicking the Run icon (it looks like a play button at the top of the chunk), or by pressing Cmd/Ctrl + Shift + Enter. RStudio executes the code and displays the results inline with the code.

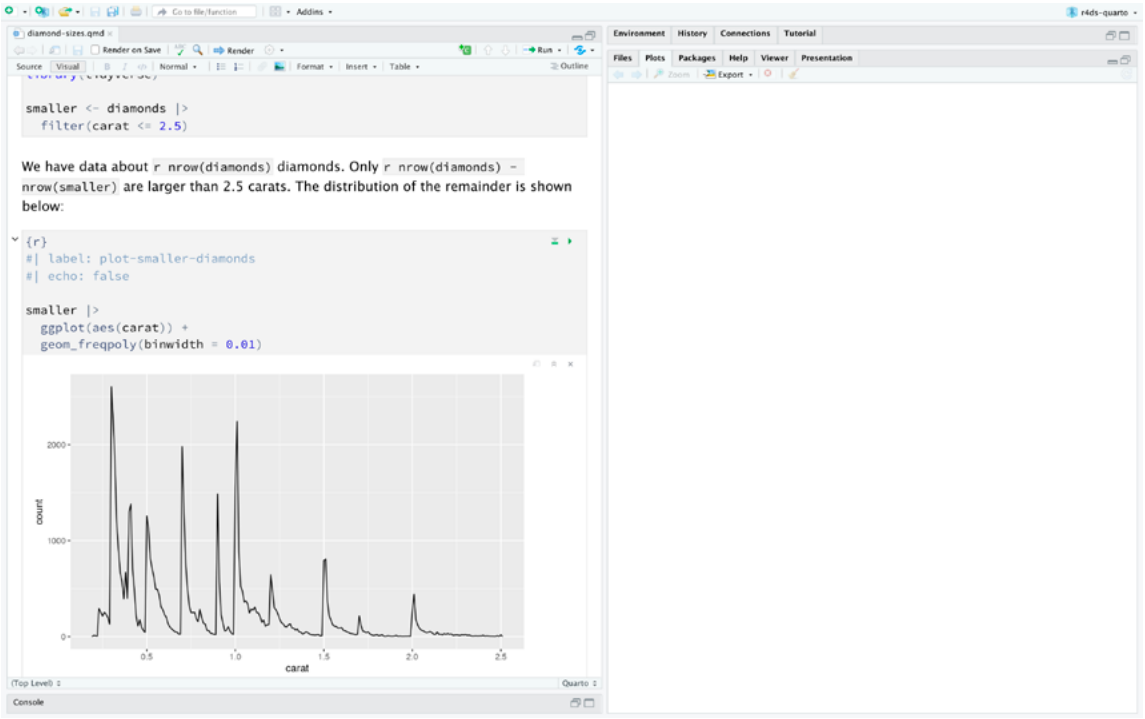


Figure 28.1: A Quarto document in RStudio. Code and output interleaved in the document, with the plot output appearing right underneath the code.

If you don't like seeing your plots and output in your document and would rather make use of RStudio's Console and Plot panes, you can click on the gear icon next to "Render" and switch to "Chunk Output in Console", as shown in Figure 28.2.

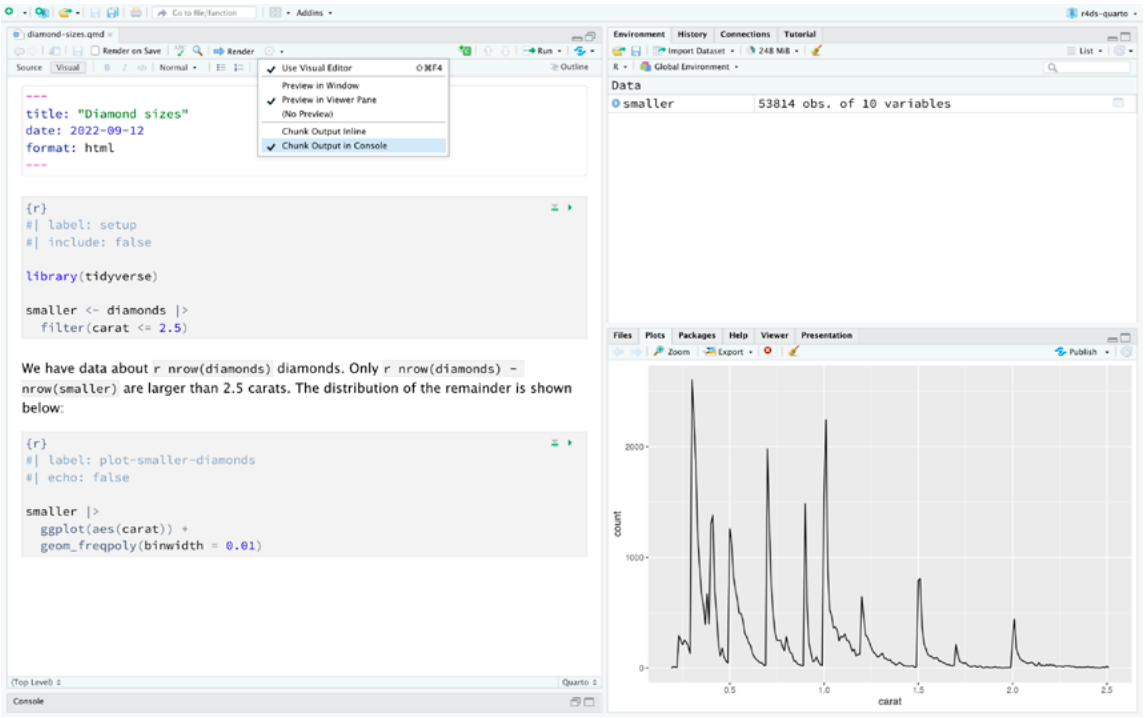


Figure 28.2: A Quarto document in RStudio with the plot output in the Plots pane.

To produce a complete report containing all text, code, and results, click "Render" or press Cmd/Ctrl + Shift + K. You can also do this programmatically with `quarto::quarto_render("diamond-sizes.qmd")`. This will display the report in the viewer pane as shown in Figure 28.3 and create an HTML file.

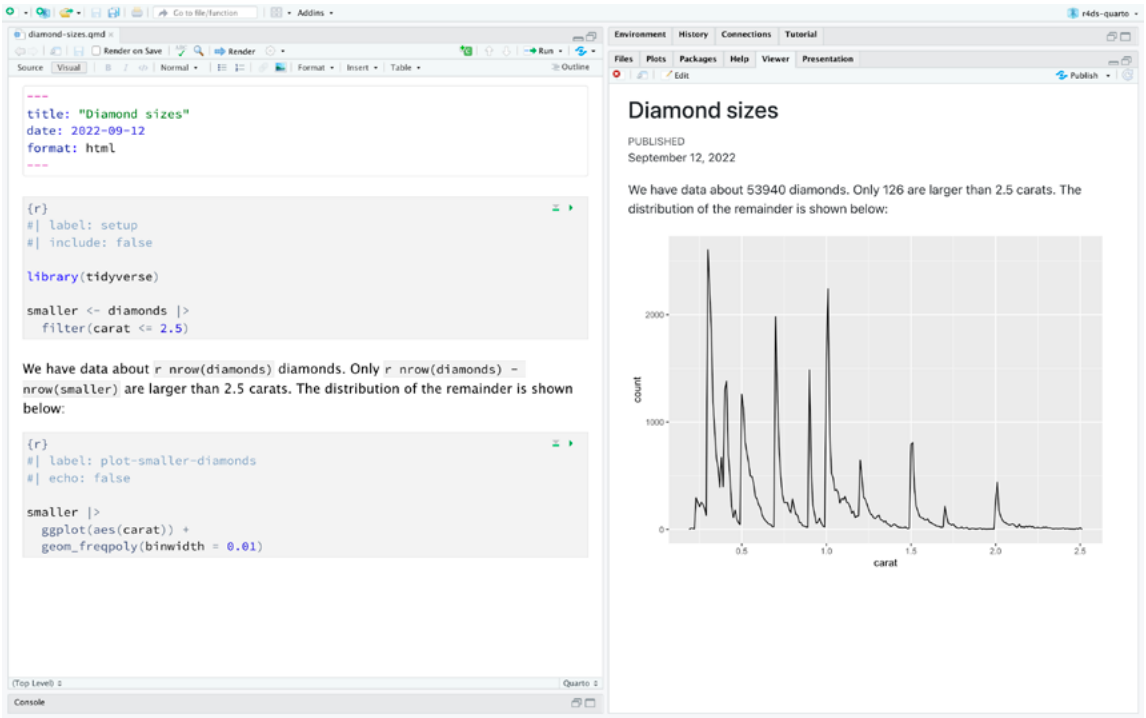


Figure 28.3: A Quarto document in RStudio with the rendered document in the Viewer pane.

When you render the document, Quarto sends the `.qmd` file to **knitr**, https://yihui.org/knitr/, which executes all of the code chunks and creates a new markdown (`.md`) document which includes the code and its output. The markdown file generated by knitr is then processed by **pandoc**, https://pandoc.org, which is responsible for creating the finished file. This process is shown in Figure 28.4. The advantage of this two step workflow is that you can create a very wide range of output formats, as you'll learn about in Chapter 29.

Figure 28.4: Diagram of Quarto workflow from qmd, to knitr, to md, to pandoc, to output in PDF, MS Word, or HTML formats.

To get started with your own `.qmd` file, select *File > New File > Quarto Document…* in the menu bar. RStudio will launch a wizard that you can use to pre-populate your file with useful content that reminds you how the key features of Quarto work.

The following sections dive into the three components of a Quarto document in more details: the markdown text, the code chunks, and the YAML header.

28.2.1 Exercises

1. Create a new Quarto document using *File > New File > Quarto Document*. Read the instructions. Practice running the chunks individually. Then render the document by clicking the appropriate button and then by using the appropriate keyboard short cut. Verify that you can modify the code, re-run it, and see modified output.

2. Create one new Quarto document for each of the three built-in formats: HTML, PDF and Word. Render each of the three documents. How do the outputs differ? How do the inputs differ? (You may need to install LaTeX in order to build the PDF output — RStudio will prompt you if this is necessary.)

28.3 Visual editor

The Visual editor in RStudio provides a WYSIWYM interface for authoring Quarto documents. Under the hood, prose in Quarto documents (`.qmd` files) is written in Markdown, a lightweight set of conventions for formatting plain text files. In fact, Quarto uses Pandoc markdown (a slightly extended version of Markdown that Quarto understands), including tables, citations, cross-references, footnotes, divs/spans, definition lists, attributes, raw HTML/TeX, and more as well as support for executing code cells and viewing their output inline. While Markdown is designed to be easy to read and write, as you will see in Section 28.4, it still requires learning new syntax. Therefore, if you're new to computational documents like `.qmd` files but have experience using tools like Google Docs or MS Word, the easiest way to get started with Quarto in RStudio is the visual editor.

In the visual editor you can either use the buttons on the menu bar to insert images, tables, cross-references, etc. or you can use the catch-all `⌘` + `/` or `Ctrl` + `/` shortcut to insert just about anything. If you are at the beginning of a line (as shown in Figure 28.5), you can also enter just `/` to invoke the shortcut.

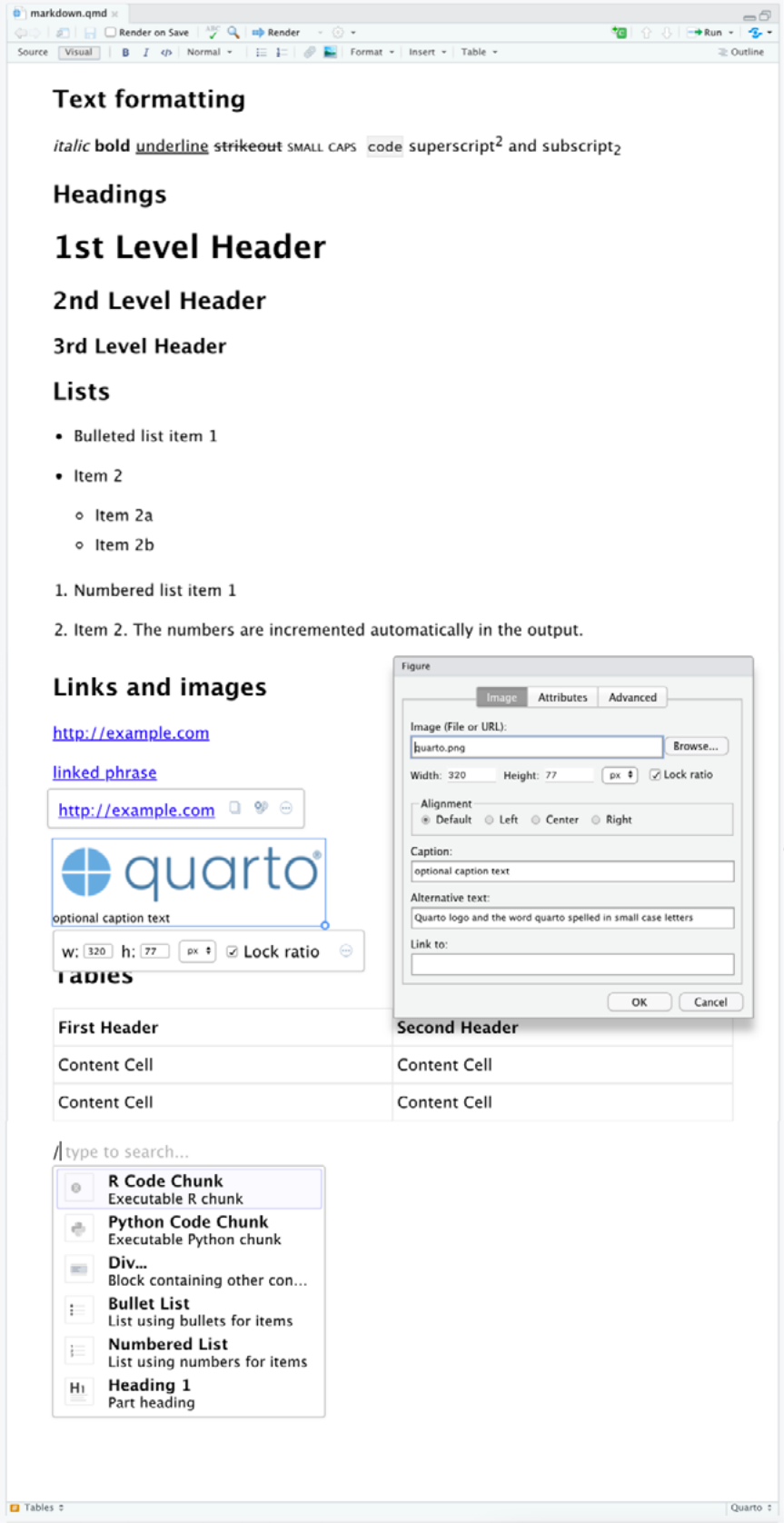


Figure 28.5: Quarto visual editor.

Inserting images and customizing how they are displayed is also facilitated with the visual editor. You can either paste an image from your clipboard directly into the visual editor (and RStudio will place a copy of that image in the project directory and link to it) or you can use the visual editor's Insert > Figure / Image menu to browse to the image you want to insert or paste it's URL. In addition, using the same menu you can resize the image as well as add a caption, alternative text, and a link.

The visual editor has many more features that we haven't enumerated here that you might find useful as you gain experience authoring with it.

Most importantly, while the visual editor displays your content with formatting, under the hood, it saves your content in plain Markdown and you can switch back and forth between the visual and source editors to view and edit your content using either tool.

28.3.1 Exercises

1. Re-create the document in Figure 28.5 using the visual editor.
2. Using the visual editor, insert a code chunk using the Insert menu and then the insert anything tool.
3. Using the visual editor, figure out how to:
    a. Add a footnote.
    b. Add a horizontal rule.
    c. Add a block quote.
4. In the visual editor, go to Insert > Citation and insert a citation to the paper titled Welcome to the Tidyverse using its DOI (digital object identifier), which is 10.21105/joss.01686. Render the document and observe how the reference shows up in the document. What change do you observe in the YAML of your document?

28.4 Source editor

You can also edit Quarto documents using the Source editor in RStudio, without the assist of the Visual editor. While the Visual editor will feel familiar to those with experience writing in tools like Google docs, the Source editor will feel familiar to those with experience writing R scripts or R Markdown documents. The Source editor can also be useful for debugging any Quarto syntax errors since it's often easier to catch these in plain text.

The guide below shows how to use Pandoc's Markdown for authoring Quarto documents in the source editor.

```
## Text formatting

*italic* **bold** ~~strikeout~~ `code`

superscript^2^ subscript~2~

[underline]{.underline} [small caps]{.smallcaps}

## Headings

# 1st Level Header

## 2nd Level Header
```

```
### 3rd Level Header

## Lists

-   Bulleted list item 1

-   Item 2

    -   Item 2a

    -   Item 2b

1.  Numbered list item 1

2.  Item 2.
    The numbers are incremented automatically in the output.

## Links and images

<http://example.com>

[linked phrase](http://example.com)

![optional caption text](quarto.png){fig-alt="Quarto logo and the word quarto spelled in small case letters"}

## Tables

| First Header | Second Header |
|--------------|---------------|
| Content Cell | Content Cell  |
| Content Cell | Content Cell  |
```

The best way to learn these is simply to try them out. It will take a few days, but soon they will become second nature, and you won't need to think about them. If you forget, you can get to a handy reference sheet with *Help > Markdown Quick Reference*.

28.4.1 Exercises

1. Practice what you've learned by creating a brief CV. The title should be your name, and you should include headings for (at least) education or employment. Each of the sections should include a bulleted list of jobs/degrees. Highlight the year in bold.

2. Using the source editor and the Markdown quick reference, figure out how to:

    a. Add a footnote.
    b. Add a horizontal rule.
    c. Add a block quote.

3. Copy and paste the contents of `diamond-sizes.qmd` from https://github.com/hadley/r4ds/tree/main/quarto in to a local R Quarto document. Check that you can run it, then add text after the frequency polygon that describes its most striking features.

4. Create a document in a Google doc or MS Word (or locate a document you have created previously) with some content in it such as headings, hyperlinks, formatted text, etc. Copy the contents of this document and paste it into a Quarto document in the visual editor. Then, switch over to the source editor and inspect the source code.

28.5 Code chunks

To run code inside a Quarto document, you need to insert a chunk. There are three ways to do so:

1. The keyboard shortcut Cmd + Option + I / Ctrl + Alt + I.

2. The "Insert" button icon in the editor toolbar.

3. By manually typing the chunk delimiters ```` ```{r} ```` and ```` ``` ````.

We'd recommend you learn the keyboard shortcut. It will save you a lot of time in the long run!

You can continue to run the code using the keyboard shortcut that by now (we hope!) you know and love: Cmd/Ctrl + Enter. However, chunks get a new keyboard shortcut: Cmd/Ctrl + Shift + Enter, which runs all the code in the chunk. Think of a chunk like a function. A chunk should be relatively self-contained, and focused around a single task.

The following sections describe the chunk header which consists of ```` ```{r} ````, followed by an optional chunk label and various other chunk options, each on their own line, marked by `#|`.

28.5.1 Chunk label

Chunks can be given an optional label, e.g.

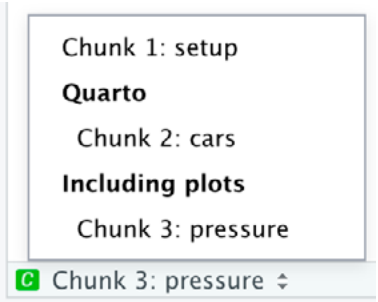
````
```{r}
#| label: simple-addition

1 + 1
```
````

```
#> [1] 2
```

This has three advantages:

1. You can more easily navigate to specific chunks using the drop-down code navigator in the bottom-left of the script editor:



2. Graphics produced by the chunks will have useful names that make them easier to use elsewhere. More on that in Section 28.6.

3. You can set up networks of cached chunks to avoid re-performing expensive computations on every run. More on that in Section 28.8.

Your chunk labels should be short but evocative and should not contain spaces. We recommend using dashes (–) to separate words (instead of underscores, _) and avoiding other special characters in chunk labels.

You are generally free to label your chunk however you like, but there is one chunk name that imbues special behavior: `setup` . When you're in a notebook mode, the chunk named setup will be run automatically once, before any other code is run.

Additionally, chunk labels cannot be duplicated. Each chunk label must be unique.

28.5.2 Chunk options

Chunk output can be customized with **options**, fields supplied to chunk header. Knitr provides almost 60 options that you can use to customize your code chunks. Here we'll cover the most important chunk options that you'll use frequently. You can see the full list at https://yihui.org/knitr/options.

The most important set of options controls if your code block is executed and what results are inserted in the finished report:

- `eval: false` prevents code from being evaluated. (And obviously if the code is not run, no results will be generated). This is useful for displaying example code, or for disabling a large block of code without commenting each line.

- `include: false` runs the code, but doesn't show the code or results in the final document. Use this for setup code that you don't want cluttering your report.

- `echo: false` prevents code, but not the results from appearing in the finished file. Use this when writing reports aimed at people who don't want to see the underlying R code.

- `message: false` or `warning: false` prevents messages or warnings from appearing in the finished file.

- `results: hide` hides printed output; `fig-show: hide` hides plots.

- `error: true` causes the render to continue even if code returns an error. This is rarely something you'll want to include in the final version of your report, but can be very useful if you need to debug exactly what is

going on inside your `.qmd`. It's also useful if you're teaching R and want to deliberately include an error. The default, `error: false` causes rendering to fail if there is a single error in the document.

Each of these chunk options get added to the header of the chunk, following `#|`, e.g., in the following chunk the result is not printed since `eval` is set to false.

````
```{r}
#| label: simple-multiplication
#| eval: false

2 * 2
```
````

The following table summarizes which types of output each option suppresses:

Option	Run code	Show code	Output	Plots	Messages	Warnings
`eval: false`	X		X	X	X	X
`include: false`		X	X	X	X	X
`echo: false`		X				
`results: hide`			X			
`fig-show: hide`				X		
`message: false`					X	
`warning: false`						X

28.5.3 Global options

As you work more with knitr, you will discover that some of the default chunk options don't fit your needs and you want to change them.

You can do this by adding the preferred options in the document YAML, under `execute`. For example, if you are preparing a report for an audience who does not need to see your code but only your results and narrative, you might set `echo: false` at the document level. That will hide the code by default, so only showing the chunks you deliberately choose to show (with `echo: true`). You might consider setting `message: false` and `warning: false`, but that would make it harder to debug problems because you wouldn't see any messages in the final document.

```
title: "My report"
execute:
  echo: false
```

Since Quarto is designed to be multi-lingual (works with R as well as other languages like Python, Julia, etc.), all of the knitr options are not available at the document execution level since some of them only work with knitr and not other engines Quarto uses for running code in other languages (e.g., Jupyter). You can, however, still set these as global options for your document under the `knitr` field, under `opts_chunk`. For example, when writing books and tutorials we set:

```
title: "Tutorial"
knitr:
  opts_chunk:
    comment: "#>"
    collapse: true
```

This uses our preferred comment formatting and ensures that the code and output are kept closely entwined.

28.5.4 Inline code

There is one other way to embed R code into a Quarto document: directly into the text, with: `` `r ` ``. This can be very useful if you mention properties of your data in the text. For example, the example document used at the start of the chapter had:

> We have data about `` `r nrow(diamonds)` `` diamonds. Only `` `r nrow(diamonds) - nrow(smaller)` `` are larger than 2.5 carats. The distribution of the remainder is shown below:

When the report is rendered, the results of these computations are inserted into the text:

> We have data about 53940 diamonds. Only 126 are larger than 2.5 carats. The distribution of the remainder is shown below:

When inserting numbers into text, `format()` is your friend. It allows you to set the number of `digits` so you don't print to a ridiculous degree of precision, and a `big.mark` to make numbers easier to read. You might combine these into a helper function:

```
comma <- function(x) format(x, digits = 2, big.mark = ",")
comma(3452345)
#> [1] "3,452,345"
comma(.12358124331)
#> [1] "0.12"
```

28.5.5 Exercises

1. Add a section that explores how diamond sizes vary by cut, color, and clarity. Assume you're writing a report for someone who doesn't know R, and instead of setting `echo: false` on each chunk, set a global option.

2. Download `diamond-sizes.qmd` from https://github.com/hadley/r4ds/tree/main/quarto. Add a section that describes the largest 20 diamonds, including a table that displays their most important attributes.

3. Modify `diamonds-sizes.qmd` to use `label_comma()` to produce nicely formatted output. Also include the percentage of diamonds that are larger than 2.5 carats.

28.6 Figures

The figures in a Quarto document can be embedded (e.g., a PNG or JPEG file) or generated as a result of a code chunk.

To embed an image from an external file, you can use the Insert menu in the Visual Editor in RStudio and select Figure / Image. This will pop open a menu where you can browse to the image you want to insert as well as add alternative text or caption to it and adjust its size. In the visual editor you can also simply paste an image from your clipboard into your document and RStudio will place a copy of that image in your project folder.

If you include a code chunk that generates a figure (e.g., includes a `ggplot()` call), the resulting figure will be automatically included in your Quarto document.

28.6.1 Figure sizing

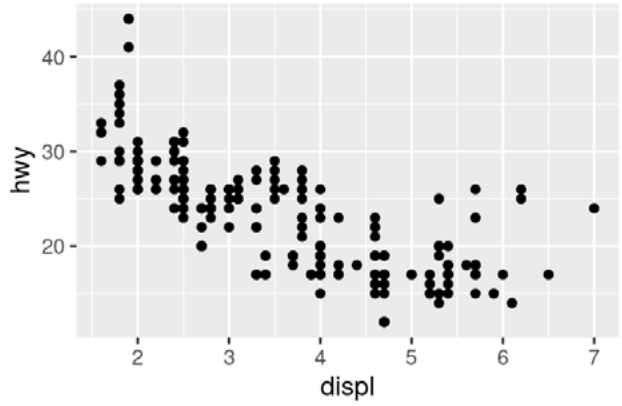
The biggest challenge of graphics in Quarto is getting your figures the right size and shape. There are five main options that control figure sizing: `fig-width`, `fig-height`, `fig-asp`, `out-width` and `out-height`. Image sizing is challenging because there are two sizes (the size of the figure created by R and the size at which it is inserted in the output document), and multiple ways of specifying the size (i.e. height, width, and aspect ratio: pick two of three).

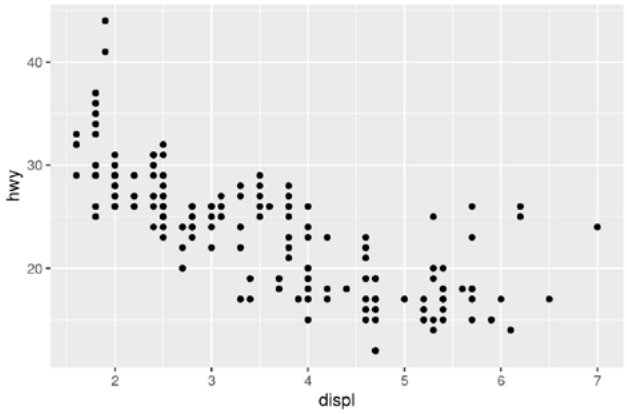
We recommend three of the five options:

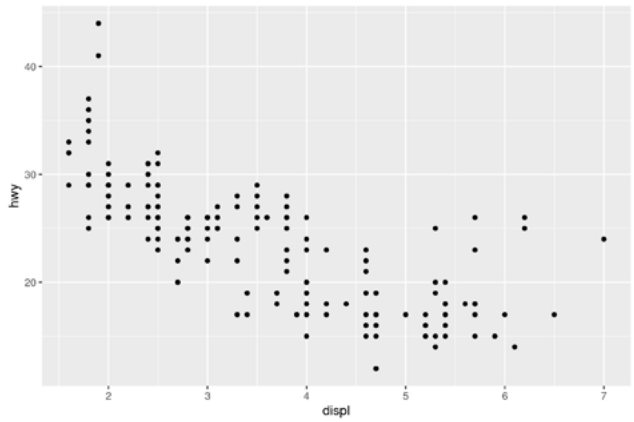
- Plots tend to be more aesthetically pleasing if they have consistent width. To enforce this, set `fig-width: 6` (6") and `fig-asp: 0.618` (the golden ratio) in the defaults. Then in individual chunks, only adjust `fig-asp`.
- Control the output size with `out-width` and set it to a percentage of the body width of the output document. We suggest to `out-width: "70%"` and `fig-align: center`.

  That gives plots room to breathe, without taking up too much space.
- To put multiple plots in a single row, set the `layout-ncol` to 2 for two plots, 3 for three plots, etc. This effectively sets `out-width` to "50%" for each of your plots if `layout-ncol` is 2, "33%" if `layout-ncol` is 3, etc. Depending on what you're trying to illustrate (e.g., show data or show plot variations), you might also tweak `fig-width`, as discussed below.

If you find that you're having to squint to read the text in your plot, you need to tweak `fig-width`. If `fig-width` is larger than the size the figure is rendered in the final doc, the text will be too small; if `fig-width` is smaller, the text will be too big. You'll often need to do a little experimentation to figure out the right ratio between the `fig-width` and the eventual width in your document. To illustrate the principle, the following three plots have `fig-width` of 4, 6, and 8 respectively:







If you want to make sure the font size is consistent across all your figures, whenever you set `out-width`, you'll also need to adjust `fig-width` to maintain the same ratio with your default `out-width`. For example, if your default `fig-width` is 6 and `out-width` is "70%", when you set `out-width: "50%"` you'll need to set `fig-width` to 4.3 (6 * 0.5 / 0.7).

Figure sizing and scaling is an art and science and getting things right can require an iterative trial-and-error approach. You can learn more about figure sizing in the taking control of plot scaling blog post.

28.6.2 Other important options

When mingling code and text, like in this book, you can set `fig-show: hold` so that plots are shown after the code. This has the pleasant side effect of forcing you to break up large blocks of code with their explanations.

To add a caption to the plot, use `fig-cap`. In Quarto this will change the figure from inline to "floating".

If you're producing PDF output, the default graphics type is PDF. This is a good default because PDFs are high quality vector graphics. However, they can produce very large and slow plots if you are displaying thousands of points. In that case, set `fig-format: "png"` to force the use of PNGs. They are slightly lower quality, but will be much more compact.

It's a good idea to name code chunks that produce figures, even if you don't routinely label other chunks. The chunk label is used to generate the file name of the graphic on disk, so naming your chunks makes it much easier to pick out plots and reuse in other circumstances (e.g., if you want to quickly drop a single plot into an email).

28.6.3 Exercises

1. Open `diamond-sizes.qmd` in the visual editor, find an image of a diamond, copy it, and paste it into the document. Double click on the image and add a caption. Resize the image and render your document. Observe how the image is saved in your current working directory.
2. Edit the label of the code chunk in `diamond-sizes.qmd` that generates a plot to start with the prefix `fig-` and add a caption to the figure with the chunk option `fig-cap`. Then, edit the text above the code chunk to add a cross-reference to the figure with Insert > Cross Reference.
3. Change the size of the figure with the following chunk options, one at a time, render your document, and describe how the figure changes.

    a. `fig-width: 10`

    b. `fig-height: 3`

    c. `out-width: "100%"`

    d. `out-width: "20%"`

28.7 Tables

Similar to figures, you can include two types of tables in a Quarto document. They can be markdown tables that you create directly in your Quarto document (using the Insert Table menu) or they can be tables generated as a result of a code chunk. In this section we will focus on the latter, tables generated via computation.

By default, Quarto prints data frames and matrices as you'd see them in the console:

```
mtcars[1:5, ]
#>                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

If you prefer that data be displayed with additional formatting you can use the `knitr::kable()` function. The code below generates Table 28.1.

```
knitr::kable(mtcars[1:5, ], )
```

Table 28.1: A knitr kable.

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2

Read the documentation for `?knitr::kable` to see the other ways in which you can customize the table. For even deeper customization, consider the **gt**, **huxtable**, **reactable**, **kableExtra**, **xtable**, **stargazer**, **pander**, **tables**, and **ascii** packages. Each provides a set of tools for returning formatted tables from R code.

28.7.1 Exercises

1. Open `diamond-sizes.qmd` in the visual editor, insert a code chunk, and add a table with `knitr::kable()` that shows the first 5 rows of the `diamonds` data frame.
2. Display the same table with `gt::gt()` instead.
3. Add a chunk label that starts with the prefix `tbl-` and add a caption to the table with the chunk option `tbl-cap`. Then, edit the text above the code chunk to add a cross-reference to the table with Insert > Cross Reference.

28.8 Caching

Normally, each render of a document starts from a completely clean slate. This is great for reproducibility, because it ensures that you've captured every important computation in code. However, it can be painful if you have some computations that take a long time. The solution is `cache: true`.

You can enable the Knitr cache at the document level for caching the results of all computations in a document using standard YAML options:

```
---
title: "My Document"
execute:
  cache: true
---
```

You can also enable caching at the chunk level for caching the results of computation in a specific chunk:

```
```{r}
#| cache: true

# code for lengthy computation...
```
```

When set, this will save the output of the chunk to a specially named file on disk. On subsequent runs, knitr will check to see if the code has changed, and if it hasn't, it will reuse the cached results.

The caching system must be used with care, because by default it is based on the code only, not its dependencies. For example, here the `processed_data` chunk depends on the `raw-data` chunk:

```
```{r}
#| label: raw-data
#| cache: true

rawdata <- readr::read_csv("a_very_large_file.csv")
```
```

```
```{r}
#| label: processed_data
#| cache: true

processed_data <- rawdata |>
  filter(!is.na(import_var)) |>
  mutate(new_variable = complicated_transformation(x, y, z))
```
```

Caching the `processed_data` chunk means that it will get re-run if the dplyr pipeline is changed, but it won't get rerun if the `read_csv()` call changes. You can avoid that problem with the `dependson` chunk option:

```
```{r}
#| label: processed-data
#| cache: true
#| dependson: "raw-data"

processed_data <- rawdata |>
  filter(!is.na(import_var)) |>
  mutate(new_variable = complicated_transformation(x, y, z))
```
```

`dependson` should contain a character vector of *every* chunk that the cached chunk depends on. Knitr will update the results for the cached chunk whenever it detects that one of its dependencies have changed.

Note that the chunks won't update if `a_very_large_file.csv` changes, because knitr caching only tracks changes within the `.qmd` file. If you want to also track changes to that file you can use the `cache.extra` option.

This is an arbitrary R expression that will invalidate the cache whenever it changes. A good function to use is `file.mtime()`: it returns when it was last modified. Then you can write:

````
```{r}
#| label: raw-data
#| cache: true
#| cache.extra: !expr file.mtime("a_very_large_file.csv")

rawdata <- readr::read_csv("a_very_large_file.csv")
```
````

We've followed the advice of David Robinson to name these chunks: each chunk is named after the primary object that it creates. This makes it easier to understand the `dependson` specification.

As your caching strategies get progressively more complicated, it's a good idea to regularly clear out all your caches with `knitr::clean_cache()`.

28.8.1 Exercises

1. Set up a network of chunks where `d` depends on `c` and `b`, and both `b` and `c` depend on `a`. Have each chunk print `lubridate::now()`, set `cache: true`, then verify your understanding of caching.

28.9 Troubleshooting

Troubleshooting Quarto documents can be challenging because you are no longer in an interactive R environment, and you will need to learn some new tricks. Additionally, the error could be due to issues with the Quarto document itself or due to the R code in the Quarto document.

One common error in documents with code chunks is duplicated chunk labels, which are especially pervasive if your workflow involves copying and pasting code chunks. To address this issue, all you need to do is to change one of your duplicated labels.

If the errors are due to the R code in the document, the first thing you should always try is to recreate the problem in an interactive session. Restart R, then "Run all chunks", either from the Code menu, under Run region or with the keyboard shortcut Ctrl + Alt + R. If you're lucky, that will recreate the problem, and you can figure out what's going on interactively.

If that doesn't help, there must be something different between your interactive environment and the Quarto environment. You're going to need to systematically explore the options. The most common difference is the working directory: the working directory of a Quarto is the directory in which it lives. Check the working directory is what you expect by including `getwd()` in a chunk.

Next, brainstorm all the things that might cause the bug. You'll need to systematically check that they're the same in your R session and your Quarto session. The easiest way to do that is to set `error: true` on the chunk causing the problem, then use `print()` and `str()` to check that settings are as you expect.

28.10 YAML header

You can control many other "whole document" settings by tweaking the parameters of the YAML header. You might wonder what YAML stands for: it's "YAML Ain't Markup Language", which is designed for representing hierarchical data in a way that's easy for humans to read and write. Quarto uses it to control many details of the output. Here we'll discuss three: self-contained documents, document parameters, and bibliographies.

28.10.1 Self-contained

HTML documents typically have a number of external dependencies (e.g., images, CSS style sheets, JavaScript, etc.) and, by default, Quarto places these dependencies in a `_files` folder in the same directory as your `.qmd` file. If you publish the HTML file on a hosting platform (e.g., QuartoPub, https://quartopub.com/), the dependencies in this directory are published with your document and hence are available in the published report. However, if you want to email the report to a colleague, you might prefer to have a single, self-contained, HTML document that embeds all of its dependencies. You can do this by specifying the `embed-resources` option:

```yaml
format:
  html:
    embed-resources: true
```

The resulting file will be self-contained, such that it will need no external files and no internet access to be displayed properly by a browser.

28.10.2 Parameters

Quarto documents can include one or more parameters whose values can be set when you render the report. Parameters are useful when you want to re-render the same report with distinct values for various key inputs. For example, you might be producing sales reports per branch, exam results by student, or demographic summaries by country. To declare one or more parameters, use the `params` field.

This example uses a `my_class` parameter to determine which class of cars to display:

````
---
format: html
params:
  my_class: "suv"
---

```{r}
#| label: setup
#| include: false

library(tidyverse)

class <- mpg |> filter(class == params$my_class)
```

# Fuel economy for `r params$my_class`s
````

```
```{r}
#| message: false

ggplot(class, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
```
```

As you can see, parameters are available within the code chunks as a read-only list named `params`.

You can write atomic vectors directly into the YAML header. You can also run arbitrary R expressions by prefacing the parameter value with `!expr`. This is a good way to specify date/time parameters.

```
params:
  start: !expr lubridate::ymd("2015-01-01")
  snapshot: !expr lubridate::ymd_hms("2015-01-01 12:30:00")
```

28.10.3 Bibliographies and Citations

Quarto can automatically generate citations and a bibliography in a number of styles. The most straightforward way of adding citations and bibliographies to a Quarto document is using the visual editor in RStudio.

To add a citation using the visual editor, go to Insert > Citation. Citations can be inserted from a variety of sources:

1. DOI (Document Object Identifier) references.
2. Zotero personal or group libraries.
3. Searches of Crossref, DataCite, or PubMed.
4. Your document bibliography (a `.bib` file in the directory of your document)

Under the hood, the visual mode uses the standard Pandoc markdown representation for citations (e.g., `[@citation]`).

If you add a citation using one of the first three methods, the visual editor will automatically create a `bibliography.bib` file for you and add the reference to it. It will also add a `bibliography` field to the document YAML. As you add more references, this file will get populated with their citations. You can also directly edit this file using many common bibliography formats including BibLaTeX, BibTeX, EndNote, Medline.

To create a citation within your .qmd file in the source editor, use a key composed of '@' + the citation identifier from the bibliography file. Then place the citation in square brackets. Here are some examples:

```
Separate multiple citations with a `;`: Blah blah [@smith04; @doe99].

You can add arbitrary comments inside the square brackets:
Blah blah [see @doe99, pp. 33-35; also @smith04, ch. 1].

Remove the square brackets to create an in-text citation: @smith04
```

```
says blah, or @smith04 [p. 33] says blah.

Add a `-` before the citation to suppress the author's name:
Smith says blah [-@smith04].
```

When Quarto renders your file, it will build and append a bibliography to the end of your document. The bibliography will contain each of the cited references from your bibliography file, but it will not contain a section heading. As a result it is common practice to end your file with a section header for the bibliography, such as `# References` or `# Bibliography`.

You can change the style of your citations and bibliography by referencing a CSL (citation style language) file in the `csl` field:

```
bibliography: rmarkdown.bib
csl: apa.csl
```

As with the bibliography field, your csl file should contain a path to the file. Here we assume that the csl file is in the same directory as the .qmd file. A good place to find CSL style files for common bibliography styles is https://github.com/citation-style-language/styles.

28.11 Workflow

Earlier, we discussed a basic workflow for capturing your R code where you work interactively in the *console*, then capture what works in the *script editor*. Quarto brings together the console and the script editor, blurring the lines between interactive exploration and long-term code capture. You can rapidly iterate within a chunk, editing and re-executing with Cmd/Ctrl + Shift + Enter. When you're happy, you move on and start a new chunk.

Quarto is also important because it so tightly integrates prose and code. This makes it a great **analysis notebook** because it lets you develop code and record your thoughts. An analysis notebook shares many of the same goals as a classic lab notebook in the physical sciences. It:

- Records what you did and why you did it. Regardless of how great your memory is, if you don't record what you do, there will come a time when you have forgotten important details. Write them down so you don't forget!

- Supports rigorous thinking. You are more likely to come up with a strong analysis if you record your thoughts as you go, and continue to reflect on them. This also saves you time when you eventually write up your analysis to share with others.

- Helps others understand your work. It is rare to do data analysis by yourself, and you'll often be working as part of a team. A lab notebook helps you share not only what you've done, but why you did it with your colleagues or lab mates.

Much of the good advice about using lab notebooks effectively can also be translated to analysis notebooks. We've drawn on our own experiences and Colin Purrington's advice on lab notebooks (https://colinpurrington.com/tips/lab-notebooks) to come up with the following tips:

- Ensure each notebook has a descriptive title, an evocative file name, and a first paragraph that briefly describes the aims of the analysis.

- Use the YAML header date field to record the date you started working on the notebook:

  ```
  date: 2016-08-23
  ```

  Use ISO8601 YYYY-MM-DD format so that's there no ambiguity. Use it even if you don't normally write dates that way!

- If you spend a lot of time on an analysis idea and it turns out to be a dead end, don't delete it! Write up a brief note about why it failed and leave it in the notebook. That will help you avoid going down the same dead end when you come back to the analysis in the future.

- Generally, you're better off doing data entry outside of R. But if you do need to record a small snippet of data, clearly lay it out using `tibble::tribble()`.

- If you discover an error in a data file, never modify it directly, but instead write code to correct the value. Explain why you made the fix.

- Before you finish for the day, make sure you can render the notebook. If you're using caching, make sure to clear the caches. That will let you fix any problems while the code is still fresh in your mind.

- If you want your code to be reproducible in the long-run (i.e. so you can come back to run it next month or next year), you'll need to track the versions of the packages that your code uses. A rigorous approach is to use **renv**, https://rstudio.github.io/renv/index.html, which stores packages in your project directory. A quick and dirty hack is to include a chunk that runs `sessionInfo()` — that won't let you easily recreate your packages as they are today, but at least you'll know what they were.

- You are going to create many, many, many analysis notebooks over the course of your career. How are you going to organize them so you can find them again in the future? We recommend storing them in individual projects, and coming up with a good naming scheme.

28.12 Summary

In this chapter we introduced you to Quarto for authoring and publishing reproducible computational documents that include your code and your prose in one place. You've learned about writing Quarto documents in RStudio with the visual or the source editor, how code chunks work and how to customize options for them, how to include figures and tables in your Quarto documents, and options for caching for computations. Additionally, you've learned about adjusting YAML header options for creating self-contained or parametrized documents as well as including citations and bibliography. We have also given you some troubleshooting and workflow tips.

While this introduction should be sufficient to get you started with Quarto, there is still a lot more to learn. Quarto is still relatively young, and is still growing rapidly. The best place to stay on top of innovations is the official Quarto website: https://quarto.org.

There are two important topics that we haven't covered here: collaboration and the details of accurately communicating your ideas to other humans. Collaboration is a vital part of modern data science, and you can make your life much easier by using version control tools, like Git and GitHub. We recommend "Happy Git with R", a user friendly introduction to Git and GitHub from R users, by Jenny Bryan. The book is freely available online: https://happygitwithr.com.

We have also not touched on what you should actually write in order to clearly communicate the results of your analysis. To improve your writing, we highly recommend reading either *Style: Lessons in Clarity and Grace* by Joseph M. Williams & Joseph Bizup, or *The Sense of Structure: Writing from the Reader's Perspective* by George Gopen. Both books will help you understand the structure of sentences and paragraphs, and give you the tools to make your writing more clear. (These books are rather expensive if purchased new, but they're used by many English classes so there are plenty of cheap second-hand copies). George Gopen also has a number of short articles on writing at https://www.georgegopen.com/litigation-articles.html. They are aimed at lawyers, but almost everything applies to data scientists too.

R for Data Science (2e) was written by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund. This book was built with Quarto.