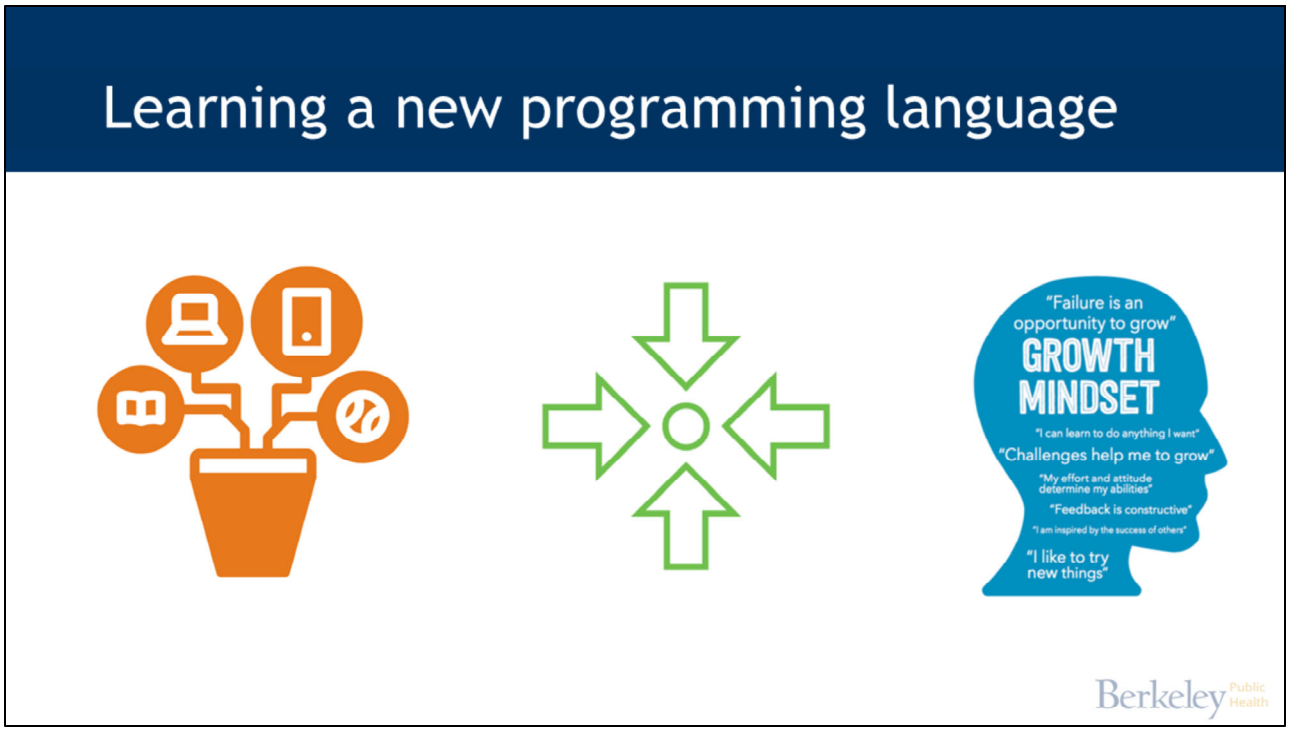
PHW290 Week 1 Reader

Overview

Topic 1: Getting Started with R

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 the semester learning a new programming language. So why use R in public health? Why do we think it's important? And why are we excited to be here to teach it?

First of all, R is really shareable. It's open source, which means that it's available for anyone to download and utilize. 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 it's free or cheap, and that no license is needed. And that's different than some 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 all over the world, but in all different sectors. People doing data analysis and data science are using R, which is really cool when you see online the different folks that are doing the same things. There's also ways to really promote collaboration and version control in R, which helps with working with groups and also just keeping track of changes you've made to your code over time. The next is that R is very powerful. It provides tools for data management and managing and cleaning very large data sets. It allows for high-power calculations, like you would do with a calculator, and/or you can do high-power calculations on data sets, which is great. It allows for robust analysis,

statistical analyses, and also offers some specific tools for epidemiology.

We're also excited about the flexible reporting that's available through R. This might be a little forward-thinking, but there's a lot of options for automation. The graphics that are produced in R are high quality, especially compared to some of the other analytic tools out there. And some of those outputs and graphics can actually be made interactive. There's also the ability to produce HTML, PDFs, and web applications straight from R, in addition to being able to export to different Microsoft products. So overall, we feel like R is a super shareable, approachable, powerful, and flexible tool that's becoming used more and more in public health, and really makes someone marketable as they enter or continue on in their career.

So now, shift a little bit and talk about learning a new programming language. So myself included, when I've been at different points in my education or career, where I've had a new opportunity to learn a new programming language, it can feel challenging and daunting, but it can also feel kind of exciting. So we wanted to offer a few different thoughts on how to approach learning R or a new programming language in general.

So first, I really want to encourage everyone to just really utilize all of the resources that are available to you. Programming is not meant to be done in a tunnel with no help or external resources. You have your peers, this learning community. You have us, your teaching staff or instructors. There's the internet, Google, Stack Overflow. The R and Rstudio Community also has a ton of resources. So we encourage you to approach us or your peers with questions, but to also try out Googling questions or trouble you're running into as well.

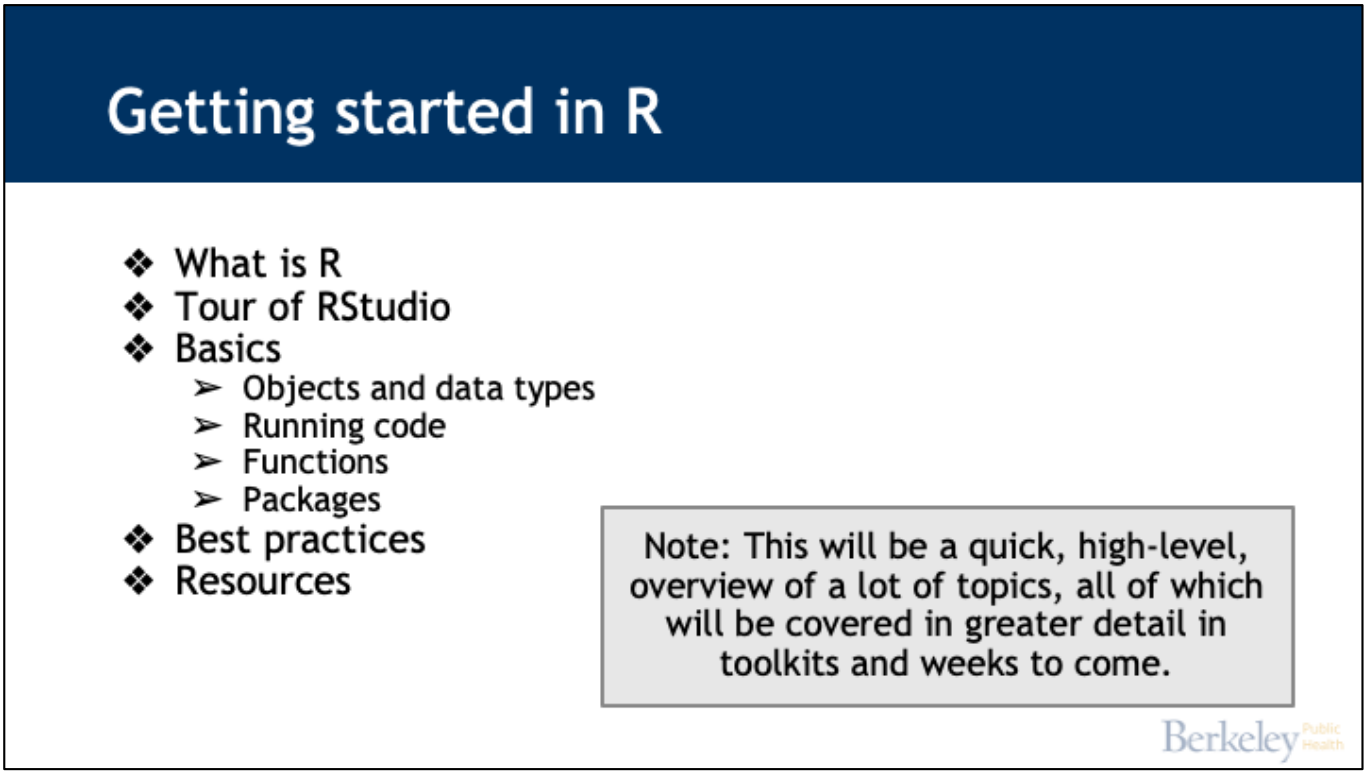
Another other piece of programming that I think is really important is this idea that there's really not always one way to do something. There might be many ways that have different pros and cons of doing one thing. And I think programming allows for some creativity and allows for a user or programmer to come up with a method or a system that works best for them. Might look different from their peer, but as long as their code is doing what they expect it to, and they're getting the end result they desire, then great. I think we can learn from each other. And I think for me, as I've learned more R over the years, there's times where I'll go back and look at old code and realize I've learned a lot of new ways to do things, and I can replace and make my old code more efficient. And I think there's something kind of exciting about that. It might also feel frustrating that there's not one clear, rigid path to do something. But I think we're hoping to offer some tools to build a base knowledge of using R. Then you can decide how to approach it in the future.

The last piece is really encouraging everyone to approach this semester with a growth mindset. And if you haven't heard that term before, it's a term developed by Carol Dweck. And the idea is a 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 this idea of, I can't learn new things, I'm never going to be good at this, I can't do it. We just encourage you to try to lean to the side of the growth mindset. I think it'll make this semester more enjoyable. It will also likely make you more successful.

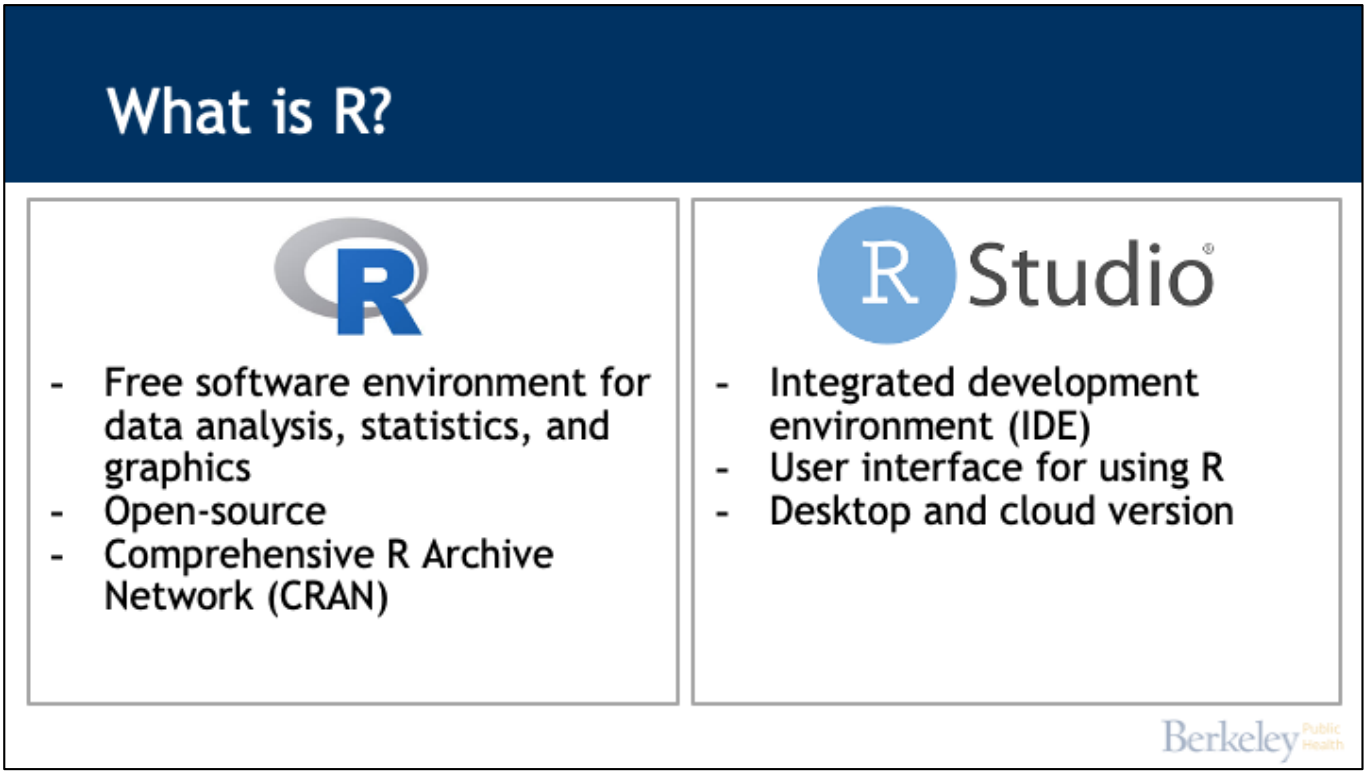
I think related to this is just this idea that programming is really about the path and the journey, and that sounds kind of cheesy, but it's not about the end result exactly. It's about learning how to do things, developing a process, learning how to troubleshoot and debug code, in addition to learning how to write code. So just keeping that in mind, that everything we do and learn this semester is contributing to this base knowledge you have of programming and R and you will know more about R by the end of the semester, no matter what. So we just encourage you to take that attitude as we move forward. Remember that we're here for you, you're not alone during the semester, as we try to build some new skills and apply those skills to public health problems in R. So we're excited to get started with actually using R.

Hi everyone, welcome to week 2, where we are going to get started using R!



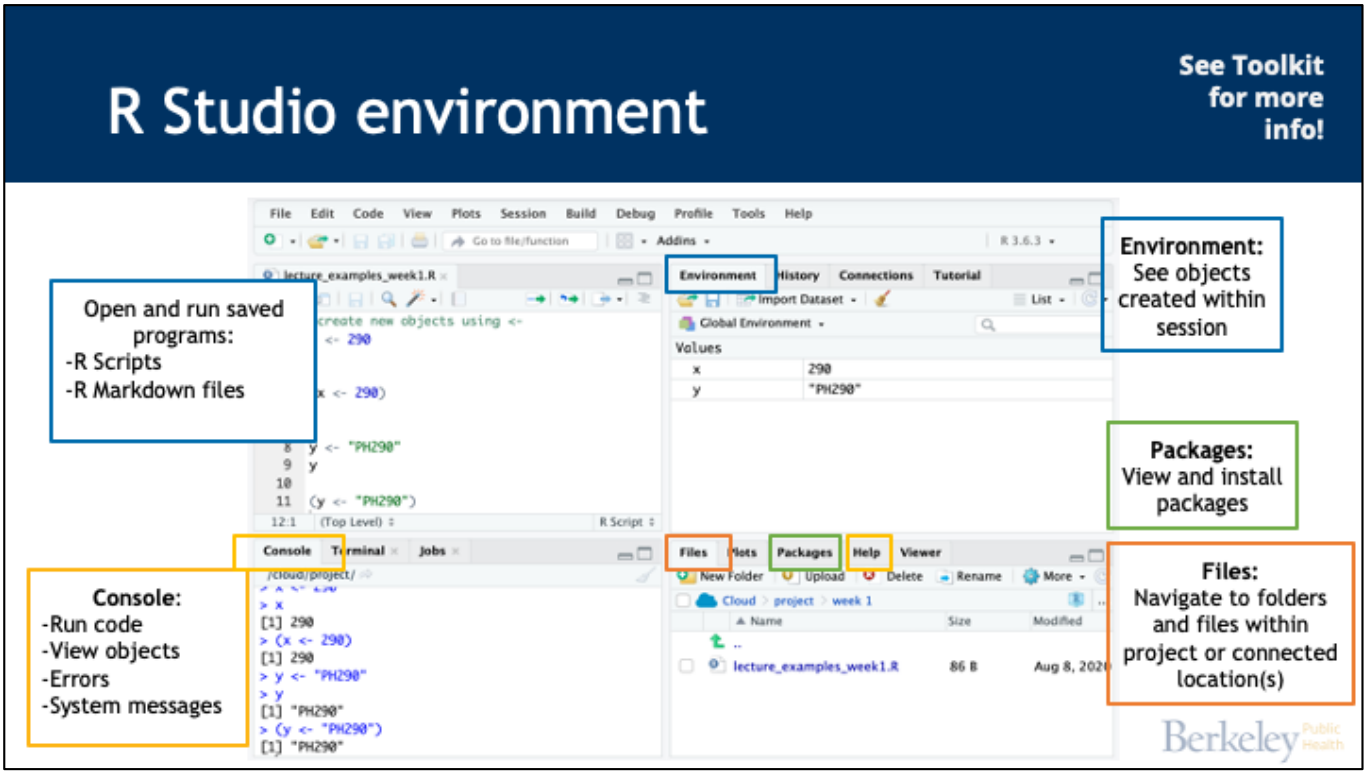
All right, so let's get started using R. So I'm going to talk about several things in the next several slides. I'm going to talk more about what R is, give a tour of RStudio at least to the best of my ability in a slide set.

I'm going to covers some basics, objects and data types, how to run code what real functions play in the use of R, and what role packages play in the use of R. Through all of that, I'm going to sprinkle in some best practices or ideas for getting started. And then we're going to talk about some resources available at the end. And I do want to mention that this really is going to be kind of quick, high-level overview of a lot of topics, all of which will be covered greater detail in the toolkits and in weeks to come.



So what is R other than being just that a capital letter R that sits by itself? It's a free software environment for data analysis, statistics, and graphics. It is open source, which is part of it being free but also means that anyone can develop additional functions or packages for R that are then available for anyone else to use. And the code and documentation for R all live on this comprehensive R archive network, what you call CRAN, and is available worldwide.

RStudio is an integrated development environment. And it's really the user interface for R. And there's a desktop version that's free that you can download for your computer to use RStudio, but there's also a cloud version, which is what we'll be using through data hub for this course.

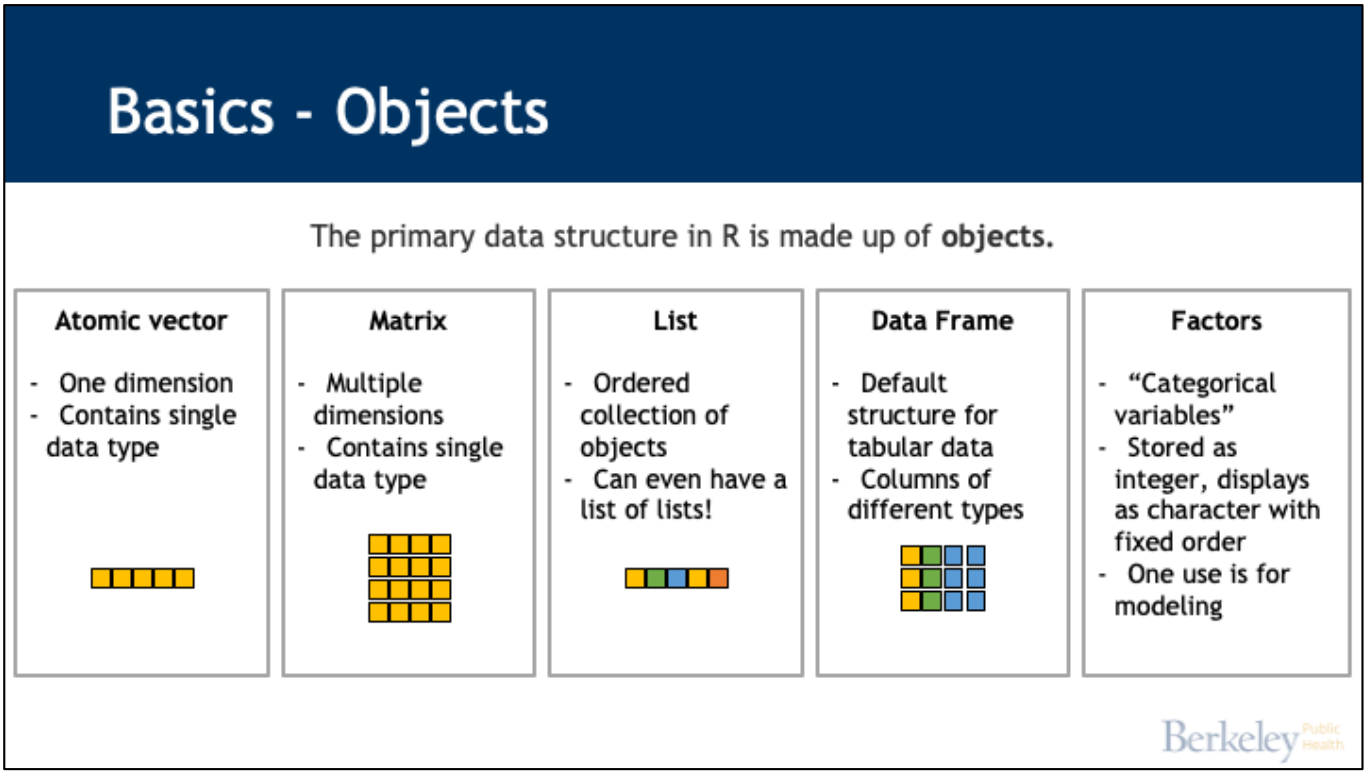


So let's take a peek at RStudio environment. Hopefully, you've already logged into your data hub and this looks familiar. But if not, I'm going to go pretty quickly just through what each of these panes represent. I'm going to start with the top left. So this pane is where you would open and run any saved programs, and that could include R scripts, so files that just have basic R code in it that you would like to run or also R markdown files, which is how you will be doing some of your assignments. And this should be covered in some of the other videos.

The next pane is the console. And the console is a place where you can enter and run code. You can view objects you've created. You would see any errors here and any system messages here as well. And moving up to the top right, there are several tabs, but we're just going to focus on the environment tab. And you can see here any objects that you've created within your session.

So in this session, I really just created two objects, an object named x and y. And each of those have an associated value that you can see there. Moving down, there's, again, several tabs, and we're going to focus on a few of them but.

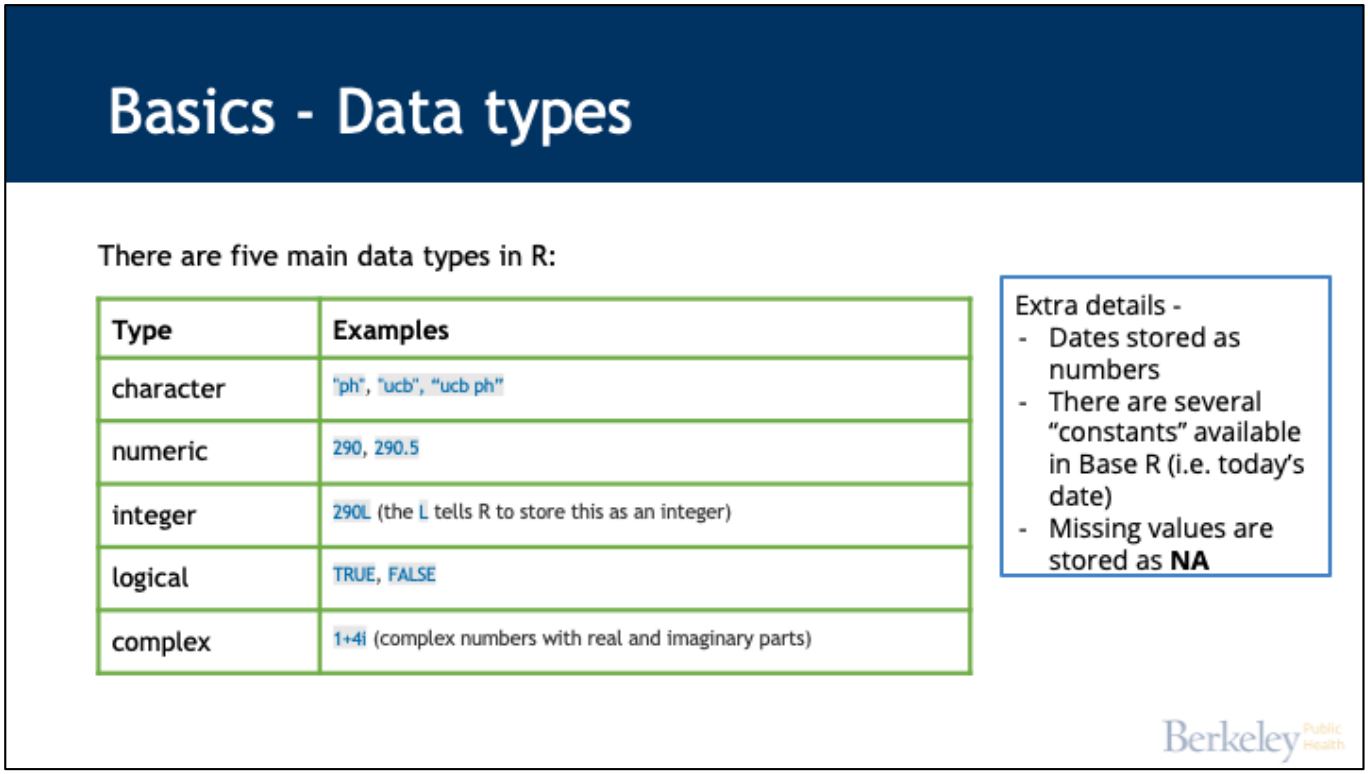
The Files tab is a place that you can navigate to folders and files within a project or connected, kind of very similar to using your file browser on your computer. There's also a tab specific to packages where you can view and install packages, and we'll talk a little bit about packages more later. And then a help tab as well that I'll talk about towards the end of these slides.



So let's start with some basics. The primary data structure in R is made up of objects. So there's five really different categories of objects, and they're all kind of outlined here with some visualization to try to get across to what the different objects mean. So you start with the atomic vector. This is a one-dimensional object that can only contain one single data type, so either numeric, or a character, or something else we'll cover in the next slide. We have a matrix, which similar to the atomic vector contains multiple dimensions, but it can only contain a single data type as well.

We also have a list, which is an ordered collection of objects. And you can even have a list of lists. So you can have objects of different types within a list, which is represented by the different color boxes in this one-dimensional object here. And then we have a data frame, which is really the default structure for tabular data in R. And you can store columns of different types.

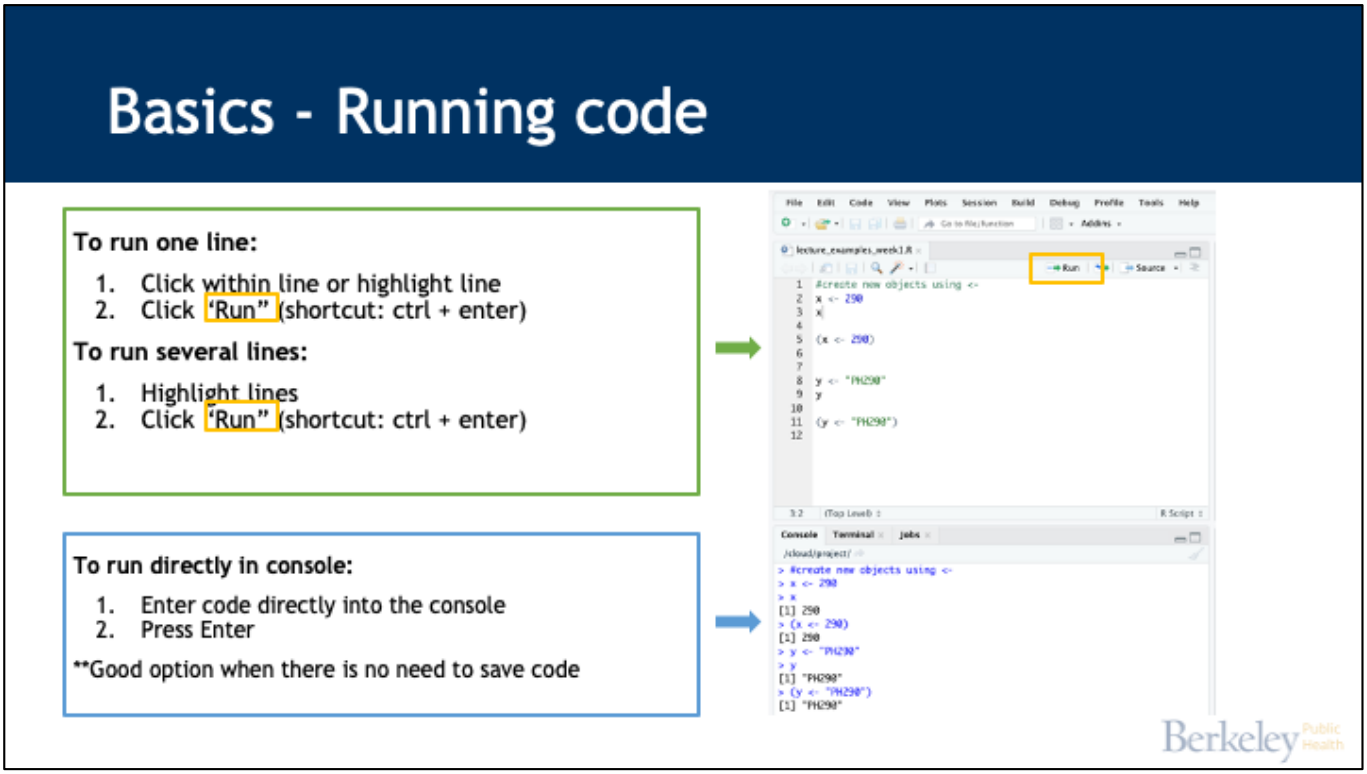
And then the last type of object is a factor. It's essentially a categorical variable that's stored as an integer but displays as character with a fixed order. One use of this is for modeling if you want to make sure that your values stay in a specific order rather than alphabetical order say or can also be used for fixing the order of output on a table or other product.



So I mentioned data types as part of objects. But just to go over the types of data in R, there's five different types. We have a character, which is denoted by quotation marks. And a character data type can have spaces or not have spaces. There's numeric data, which can be a whole number like 290, or it can also include a decimal.

You have an integer that works similar to the whole numeric. But if you want R to treat it as an integer rather than numeric, you can add a capital L after the value. And we'll get more into why you might want to do that rather than just treat something as numeric later.

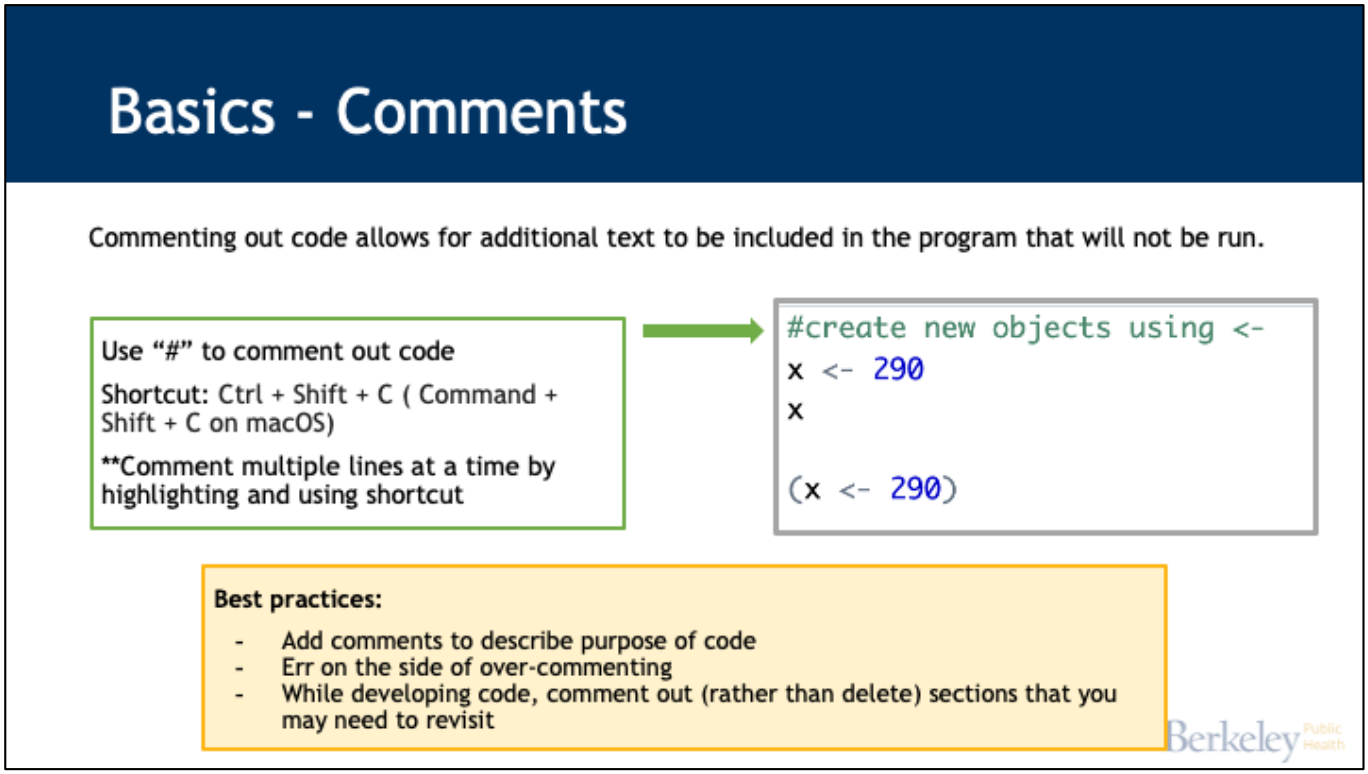
We have logical values, which are true or false, and then complex values, so something like 1 plus 4i. And a few kind of extra details or tidbits related to data types, dates are stored as numbers. They can be a little finicky, so we'll cover those more later. There are also several kind of constant values available in base R, for example, today's date. And then for all of the data types, missing values are stored as N/A.



All right, so now that we have talked a little bit about objects and data types, I want to get in to actually running code in R. On the screen, you'll see a screenshot of the save program and console that we're a part of the larger R environment screenshot below. And there are really two different ways you can run code.

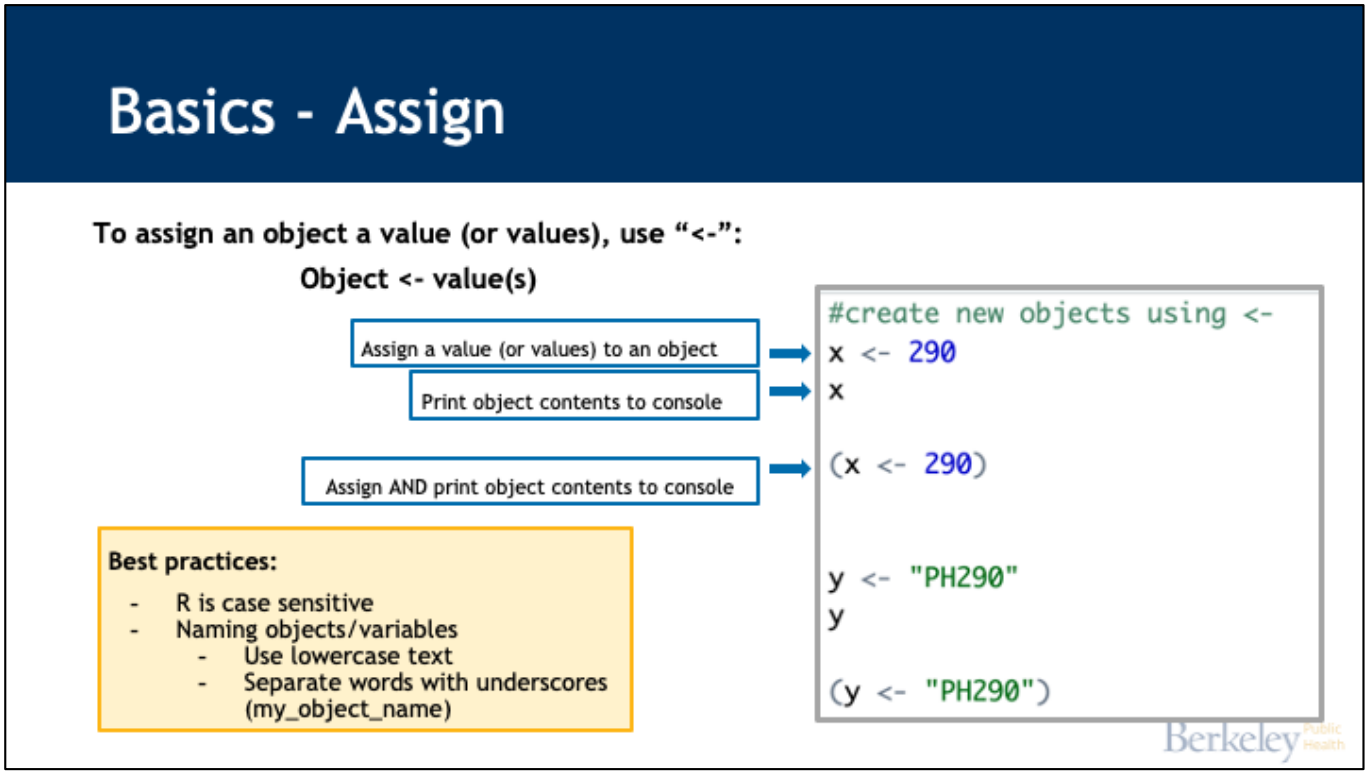
The first is through your saved R script, so code within this top pane. And to run a single line, you can either click with in the line, or highlight the whole line and press Run, or use the shortcut on your keyboard Control-Enter or Command-Enter if you're on a Mac. And similarly, to run several lines, you would highlight all the lines and use the same Run button or the same shortcut.

This is a great option for when you want to save your code, when you want to look back on it, or maybe even just when you're learning or kind of want to see things as they go. The other option is to run code directly in your console. You can enter code after the little blue arrow and then press Enter. And the results will show up automatically in the console. This is a great option when you don't need to save your code. Maybe you're troubleshooting or just doing some basic calculations to see how R works.



So when we're writing code, especially when we're writing the code that we want to save in an R script, it's super important to know how to comment out code. Commenting out code allows for additional text to be included in the program, and that text will not be run as part of the other lines. So in R, the key for commenting out text is to use the hashtag or pound sign before your text at the start of the line. So you can enter the hashtag or pound signed by hand, or you can use this shortcut Control-Shift-C or Command-Shift-C on a Mac. And similar to running code, you can comment multiple lines at a time by highlighting and using the shortcut.

Some considerations or best practices for comments-- we recommend adding comments to describe the purpose of the code and to really err on the side of over-commenting. And while you're developing new code, comment out rather than delete sections that you might want to revisit or that you want to remember you already tried this other method.



All right, so now that we've talked about comments, let's talk a little bit about actually doing meaningful programming, creating some objects in R.

So kind of the key for assigning values in R is this arrow like symbol that looks like it's pointing to the left that's made up of the less than sign and the minus or dash, however you want to look at it. And essentially, the structure is to on the left-hand side name the object that you're wanting to assign a value to, and then include the left-facing arrow, and then include on the right any value or values that you want to assign to the object. These values can be a single number or character string, a logical value. It can be another object or several numbers, and we'll talk more about assigning several values to an object a bit later. But here you can see how you would assign the value of 290 to the object named x.

So running this code will create an object named x with a value of 290. And it will store that. But it will not print that. So if you want then to print to your console, you type the letter x again and run that line of code. Another way to do it if you wanted to do both of those things at once is to use parentheses and assign 290 to x inside parentheses. That will assign and print the object contents to the console.

So some best practices here, especially related to naming objects in R, that R is case sensitive. So if you name an object with lowercase or capital, it must always be referred to in that same case. This is a little bit different than some other programming languages. So we recommend when naming objects or variables to use lowercase text and to separate words with underscores, so something like my_object_name.

Basics - Calculations

- R can be used as a calculator
- Objects can be created based on calculated values

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!

Berkeley Public Health

So to add on to kind of the creation of objects or assigning values, I mentioned earlier that R is capable of being used as a calculator. And you can do that through the console just entering numbers that you want to make a calculation on. But objects can also be created based on calculated values. And then you can also calculate values of combined objects. So for example, there's a few different calculations done here.

This is an example of me typing into the console and getting results there. So you can see I did a basic edition problem and get the result in the next line. So it can also assign x the value of 290. Return x, which comes out as 290. But then I also am creating a value named y that is the result of 200 minus 90.

And so then we return y, and we get the value of 110. But then we can also say, well, what's x times y. So we have x is 290 here. We have y is equal to 110.

When we multiply those together, we get 31,900. So just an example of some of the things you can start doing with calculations in R. And here's a list of some of the basic operators and calculations you can do. There is a ton more that we will provide a list in the resources.

Basics - Relational Comparisons

- R can be used to compare values
- Returns a TRUE or FALSE

Example:

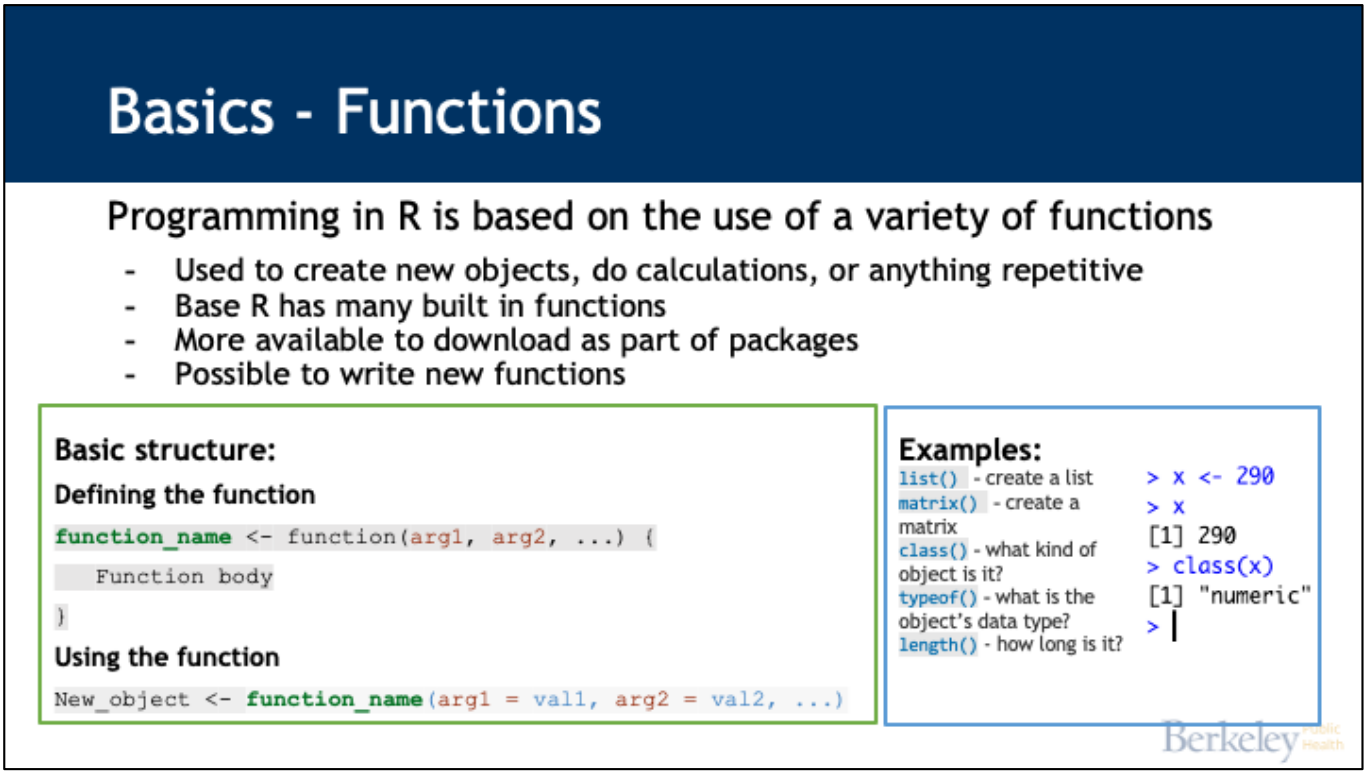
```
> a<-200
> b<-90
> a==b
[1] FALSE
> a!=b
[1] TRUE
> a<=b
[1] FALSE
> a>=b
[1] TRUE
```

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

Berkeley Public Health

So in addition to doing calculations, another basic of R is that there's capacity to do relational comparisons between values or objects. And these comparisons will return a true or false. So you can see some examples here, and I'll go ahead and put some of the operators up because especially the equals operator is a bit different being that it requires two equal signs rather than just one. But over on the left, you can see I've created an object a with a value 200, the object b, the value of 90. I assess if there equal. It returns false because they are not.

But when I assess if they're not equal, it returns true. Then do some less than and greater than comparisons as well. So this will become useful as we move forward as we want to do some conditional actions in R. But good to keep in mind and potentially start playing with as you begin using R.



So to take a little bit of a pivot, wanted to kind of zoom out I guess and talk about some of the bigger components of R. And one of those things are functions. Programming in R is based on the use of a variety of functions. And these functions can be used to create new objects, to do calculations, or really to do anything that's repetitive.

Base R has many built-in functions that we will start out using, but there is also infinitely more available to download as part of different packages. And then ultimately it's possible to write your own functions. And this is something you would want to do if you're doing some programming where you're doing an action more than twice. And rather than having to write out repetitive code, to write a function that replicates your code for you essentially. And we'll cover writing new functions about mid semester.

But as an introduction to the function, I want to talk a little bit about their basic structure and what's required. So there's two pieces really of a function. There's the function definition, which for functions that we are using that are built into R, we won't always pay a lot of attention to. But it's important to know how to navigate that. But then the most important part for us at least when we're beginning is how to use the function.

So a function is set up-- basically, we have a function name kind of with the lowercase underscore. And then we have our left arrow assigner. And then we're noting that this is a function that we're creating. So this would always say function and then indicating which arguments will be passed into the function. So this could be a variety of different things, a data set, a value, et cetera.

And then we define our function within this function body. But then after we've done that and we want to use it, we can create a new object. And we'll left assign it there and call the function name. And then for each of the arguments listed up here, put the argument equals and then add the value of what we're wanting to run in the function.

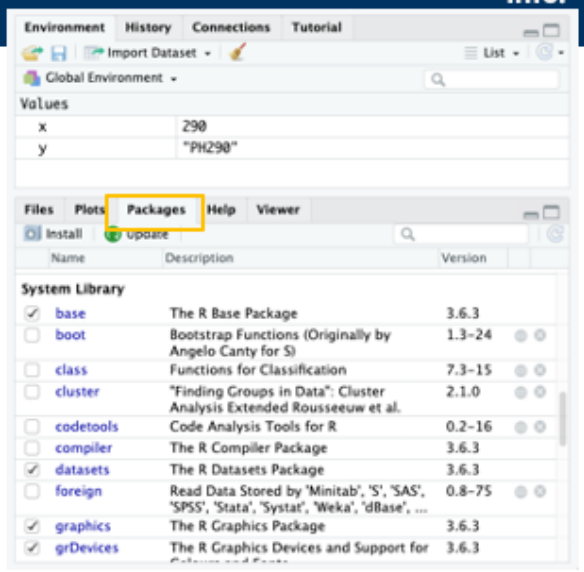
And a few examples. Some functions get really complicated. But to start, functions can be something like how we create a list or a matrix if we're referencing the data objects mentioned earlier. Then there's also functions to kind of get information about the objects we're using in R so that we can get the class, the data type, and the length of the object. So you can see one example over here. We again have this x assigned with the value of 290. We're looking at the class of x, and it's numeric.

Basics - Packages

See Toolkit for more info!

- Expand capabilities of Base R
- Many available through Github and/or CRAN
- Come with thorough documentation
- Install into R environment
- Load in current sessions

Best practices:
- Load packages at start of the script
- All packages are not made equal!



Berkeley Public Health

So zooming kind of even a bit further out from functions is the idea of packages. Packages expand the capabilities of base R. There are many, many available through GitHub and/or CRAN. And packages can be developed by anybody.

RStudio has a lot of packages that their programmers have developed, and they're wonderful. But any R programmer can develop their own package and put it out there for others to download. So they do come with thorough documentation. They need to be installed into the R environment and then loaded during each current session. And they can be installed through this Packages tab within RStudio environment.

For this course because we're using data hub, packages should already be installed. But they can be referenced in the Packages tab. And if you click on the package name, you can get even more information. And we'll cover more about packages in a toolkit video for this week. So check that out.

But I wanted to mention a couple things in terms of best practices. We recommend loading packages that you'll need at the start of the script. That way, that's the first step that runs. And anything you run below it will reference the needed packages. And then just kind of keeping in mind that all packages are not made equal. I think I just mentioned that anyone can create a package, so just being very sure of what the package is doing and what the intention is behind the package to make sure it's doing what you expect it to do rather than just using it blindly.

Tidyverse

- What is the tidyverse?
  - A collection of functions, data, and documentation that extends the capabilities of base R
  - Common philosophy of data and R programming, and are designed to work together naturally
- Data wrangling with dplyr
  - Select rows/columns by value
  - Reorder the rows
  - Create new variables
  - Collapse many values down to a single summary

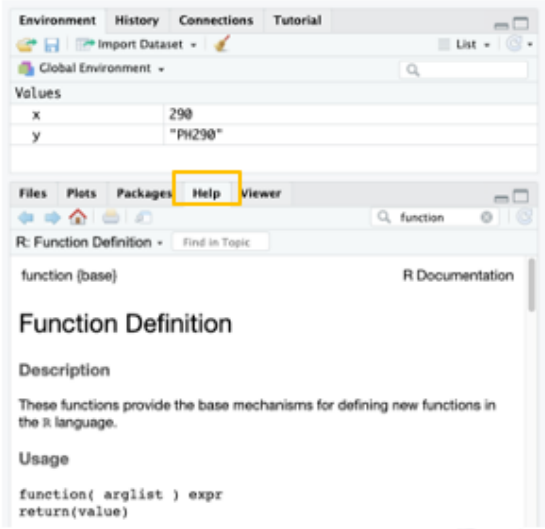
Berkeley Public Health

So I want to mention one specific package or technically a package of packages called the tidyverse, which if you've used R before, you've most likely heard of the tidyverse. If you haven't used R before, you're going to hear a lot about the tidyverse. So the tidyverse is really a collection of functions, data, and documentation that extends the capabilities of base R. And it's all based on a common philosophy of data and R programming. And all of the packages and functions within the tidyverse are designed to work together naturally.

And the tidyverse is something that's developed through RStudio and maintained through RStudio, and there's really fabulous documentation on it. But one piece of the tidyverse that I want to mention and I think will be one of our first super relevant pieces is using the tidyverse for data wrangling with dplyr. And dplyr is just a super user-friendly, intuitive way of like it says wrangling or tidying your data. It provides ways to select rows and columns by value, reorder rows, create new variables, collapse many values down to a single summary to make your data longer or wider, whatever you're looking for. So we'll spend a lot of time talking about dplyr, and the tidyverse, and some of the other packages as part of the tidyverse.

In-session Help

- Help is available within the R Studio environment
- Options:
  - Search within viewer
  - Use help() or ? within the console
  - Example: To learn more about the class() function type `help("class")` or `?class` in the console

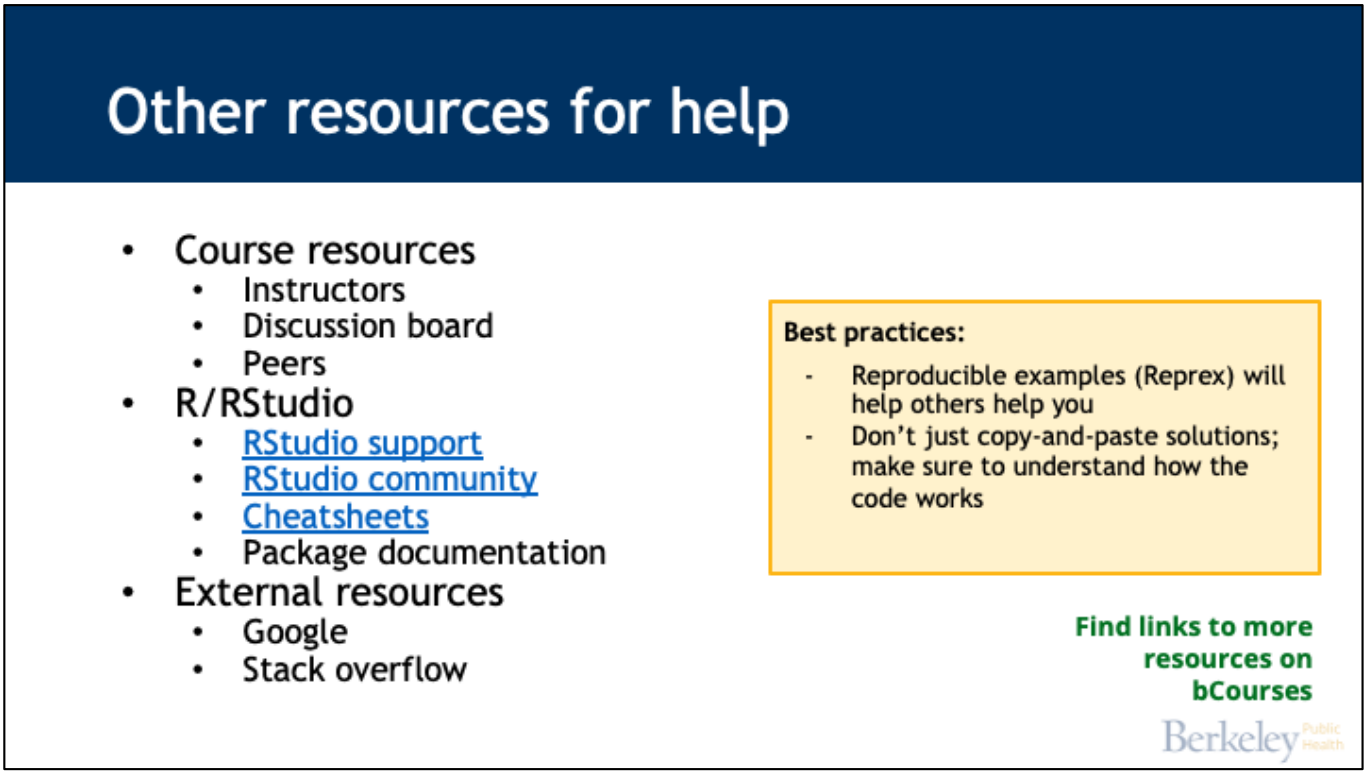


Berkeley Public Health

So with that, I want to end talking a little more about help and resources. So within the RStudio environment, within your session, there is a Help tab on the bottom right pane. And there's a few different ways that you can access help or information while you're in RStudio. You can search within this Help tab viewer.

So here I just searched for a function to get a little more information on what a function is. But you can also use the console. And in the console, you can use the help function or use a question mark to get more information.

So for example, if you want to learn more about the class function, we could type help with class in parentheses and quotes into the console and press Return. Or you could press question mark class into the console, and it would pop up with information on the Help tab. I will say that you have to be pretty precise about what you enter to get a return on the Help tab. But it is often a good place to start, especially if you're looking for information about a specific function or something else very specific.



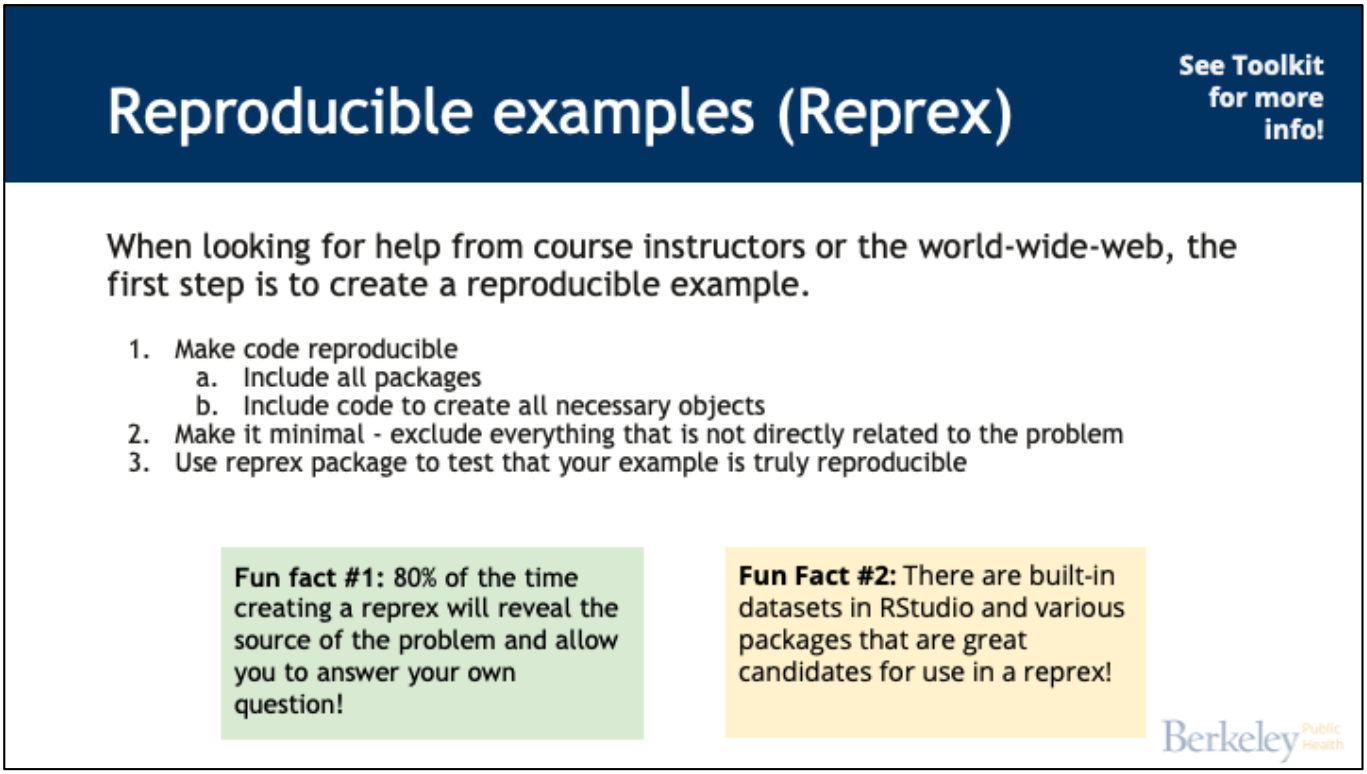
So other than within RStudio environment, there are a lot of other resources for help. There is, of course, our course resources, the instructors and teaching team. There's a discussion board through Piazza and then your peers. We really encourage asking questions on the discussion board, and we'll be available to answer those but also encourage peers to answer questions as well. Then R and RStudio both have a lot of online resources.

There's an RStudio support site. There is an RStudio community where there's forums and questions answered. There's also these things called cheatsheets, which are one or two pages that cover or include a lot of information about getting set up in R or about certain packages or methods that I highly recommend. And we will provide links to all of these resources.

But then there's also package documentation, which is an important place to look for help, especially if you're using a new package to understand how the functions and code work within the package. And then, finally, we encourage everyone to leverage the external resources like Google and Stack Overflow. Type your questions into Google or Stack Overflow. See what comes up. See what people have come up with before you.

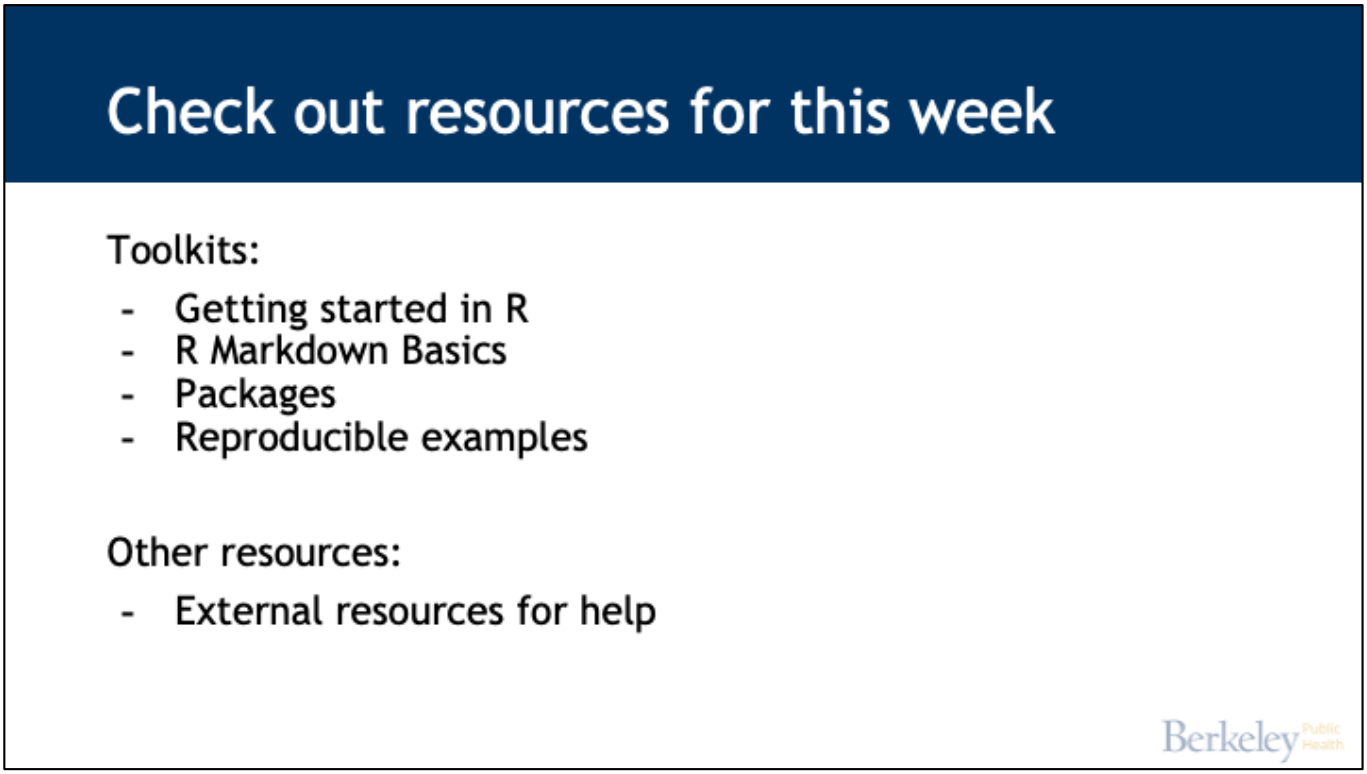
And a couple of best practices with some of these resources for help are really creating reproducible examples will help others help you. I'm going to talk more about that in the next slide. And then kind of similar to what I said about packages, don't just

copy and paste solutions. Make sure to really understand how the code works rather than just blindly using solutions. And you can find links to more of these resources on bCourses.



So before we end, I want to talk a little more about the idea of reproducible examples. It's maybe a little self-explanatory. But the idea is that when looking for help either from course instructors or the internet, the first step is to really create a reproducible example. And that obviously means making code reproducible, which means that you're including references to all needed packages within your code and that you're also creating all necessary objects within the code that you provide for help.

The other key is making the example minimal. Exclude everything that is not directly related to the problem you're having. And then where this reprex comes in is that there is a reprex package where you can test that your example is truly reproducible. A couple of fun facts-- supposedly 80% of the time, creating a reprex will actually reveal the source of the problem and allow you to answer your own question. And there are also some built-in data sets in RStudio and various packages that are really great candidates to use for creating a reproducible example.



So with that, I encourage you to check out the resources for this week. There are several toolkits, a little bit of a deeper dive, getting started with coding and creating objects in R, some basics of R markdown, more on packages and reproducible examples. And then there's some other resources on external resources for help. So with that, good luck, and talk to you all soon.

Toolkit: Getting Started with R

Toolkit - Getting started with R

Lauren Nelson

8/31/2020

Getting started with R

What you should already know:

- How to access the RStudio environment through Datahub
- How to run code (Run button or control/command + Enter)
- How to comment code (# or control/command + shift + C)

What is included in this toolkit:

- Assigning values
  - Naming objects
  - Data types
- Creating objects using functions
  - Vectors
  - Matrices
  - Lists
- Calculations and Comparisons

Assigning a value to an object

```
#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
```

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

```
#some users prefer what is called 'camelCase' which uses a capital letter
#to indicate a new word
camelCase <- "camel"
```

```
# CamelCase
camelCase
```

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

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

Data types

Character values

- 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"
```

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 <- 290L

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

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

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

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

Creating objects using functions

Vectors

(Atomic) Vectors

- One dimensional, single data type
- Multiple ways to be created:
  - c() function
  - Using : operator for a vector of consecutive numbers

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

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

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

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

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

Matrices

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

```r
#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)
```

Lists

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

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

More on data frames next 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"
```

```
typeof(matrix_1)
```

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

```
length(vec_num2)
```

```
## [1] 10
```

```
typeof(vec_num2)
```

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

```
typeof(vec_char)
```

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

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.

```
#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

c <- b-a
```

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
```

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.

```
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
```

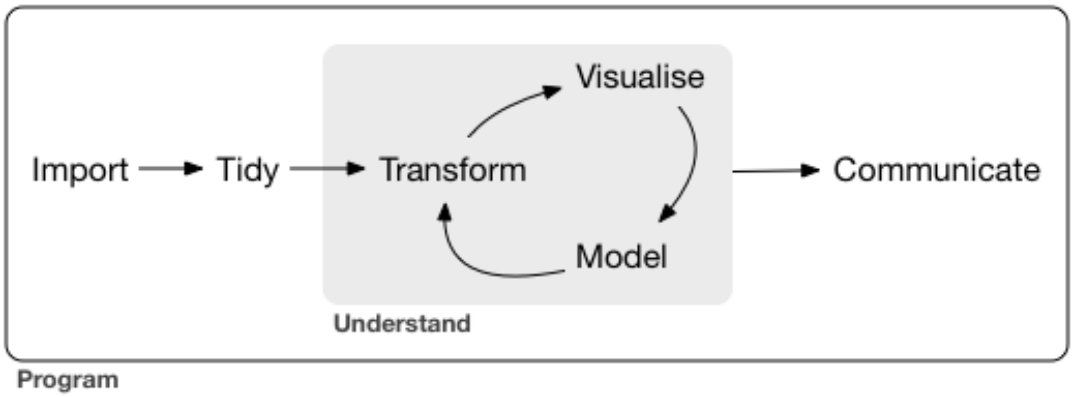
```
## [1] TRUE
```

1 Introduction

Data science is an exciting discipline that allows you to turn 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. After reading this book, you'll have the tools to tackle a wide variety of data science challenges, using the best parts of R.

1.1 What you will learn

Data science is a huge field, and there's no way you can master it by reading a single book. The goal of this book is to give you a solid foundation in the most important tools. Our model of the tools needed in a typical data science project looks something like this:



First you must **import** your data into R. This typically means that you take data stored in a file, database, or web 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 the way 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 struggle on questions about the data, not fighting to get the data into the right form for different functions.

Once you have tidy data, a common first 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: visualisation and modelling. These have complementary strengths and weaknesses so any real analysis will iterate between them many times.

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

Models are complementary tools to visualisation. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are a fundamentally mathematical or computational tool, 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 visualisation 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 every part of the project. You don't need to be an expert programmer to be a 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 for most projects they're not enough. There's a rough 80-20 rule at play; you can tackle about 80% of every project using the tools that 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.

1.2 How this book is organised

The previous description of the tools of data science is organised 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, this is not the best way to learn them:

- Starting with data ingest and tidying is sub-optimal 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 visualisation 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 it.
- Some topics are best explained with other tools. For example, we believe that it's easier to understand how models work if you already know about visualisation, tidy data, and programming.
- Programming tools are not necessarily interesting in their own right, but do allow you to tackle considerably more challenging problems. We'll give you a selection of programming tools in the middle of the book, and then you'll see how they can combine with the data science tools to tackle interesting modelling problems.

Within each chapter, we try and stick to a similar 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. While it's tempting to skip the exercises, there's no better way to learn than practicing on real problems.

1.3 What you won't learn

There are some 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.

1.3.1 Big data

This book proudly 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 learn in this book will easily handle hundreds of megabytes of data, and with a little care you can typically use them to work with 1-2 Gb of data. If you're routinely working with larger data (10-100 Gb, say), you should learn more about data.table. This book doesn't teach data.table because it has

a very concise interface which makes it harder to learn since it offers fewer linguistic cues. But if you're working with large data, the performance payoff is worth the extra effort required to learn it.

If your data is bigger than this, carefully consider if your big data problem might actually be a small data problem in disguise. While the complete data might be big, often the data needed to answer a specific question is small. You might be able to find a subset, subsample, or summary that fits in memory and still allows you to answer the question that you're interested in. The challenge here is finding the right small data, which often requires a lot of iteration.

Another possibility is that your big data problem is actually a large number of small data problems. Each individual problem might fit in memory, but you have millions of them. For example, you might want to fit a model to each person in your dataset. That would be trivial if you had just 10 or 100 people, but instead you have a million. Fortunately each problem is independent of the others (a setup that is sometimes called embarrassingly parallel), so you just need a system (like Hadoop or Spark) that allows you to send different datasets to different computers for processing. Once you've figured out how to answer the question for a single subset using the tools described in this book, you learn new tools like sparklyr, rhipe, and ddr to solve it for the full dataset.

1.3.2 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.

However, we strongly believe that it's best to master one tool at a time. You will get better faster if you dive deep, rather than spreading yourself thinly over many topics. This doesn't mean you should only know one thing, just that you'll generally learn faster if you stick to one thing at a time. You should strive to learn new things throughout your career, but make sure your understanding is solid before you move on to the next interesting thing.

We think R is a great place to start your data science journey because it is an environment designed from the ground up to support data science. R is not just a programming language, but it is also an interactive environment for doing data science. To support interaction, R is a much more flexible language than many of its peers. This flexibility comes with its downsides, but the

big upside is how easy it is to evolve tailored grammars for specific parts of the data science process. These mini languages help you think about problems as a data scientist, while supporting fluent interaction between your brain and the computer.

1.3.3 Non-rectangular data

This book focuses exclusively on rectangular data: collections of values that are each associated with a variable and an observation. There are lots of datasets that do not naturally fit in this paradigm: including images, sounds, trees, and text. But rectangular data frames are extremely common in science and industry, and we believe that they are a great place to start your data science journey.

1.3.4 Hypothesis confirmation

It's possible to divide data analysis into two camps: hypothesis generation and hypothesis confirmation (sometimes called confirmatory analysis). The focus of this book is unabashedly on hypothesis generation, or data exploration. Here you'll look deeply at the data and, in combination with your subject knowledge, generate many interesting hypotheses to help explain why the data behaves the way it does. You evaluate the hypotheses informally, using your scepticism to challenge the data in multiple ways.

The complement of hypothesis generation is hypothesis confirmation. Hypothesis confirmation is hard for two reasons:

1. You need a precise mathematical model in order to generate falsifiable predictions. This often requires considerable statistical sophistication.

2. You can only use an observation once to confirm a hypothesis. As soon as you use it more than once you're back to doing exploratory analysis. This means to do hypothesis confirmation you need to "preregister" (write out in advance) your analysis plan, and not deviate from it even when you have seen the data. We'll talk a little about some strategies you can use to make this easier in modelling.

It's common to think about modelling as a tool for hypothesis confirmation, and visualisation as a tool for hypothesis generation. But that's a false dichotomy: models are often used for exploration, and with a little care you can use visualisation for confirmation. The key difference is

how often do you look at each observation: if you look only once, it's confirmation; if you look more than once, it's exploration.

1.4 Prerequisites

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

There are four things you need 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, the documentation that describes how to use them, and sample data.

1.4.1 R

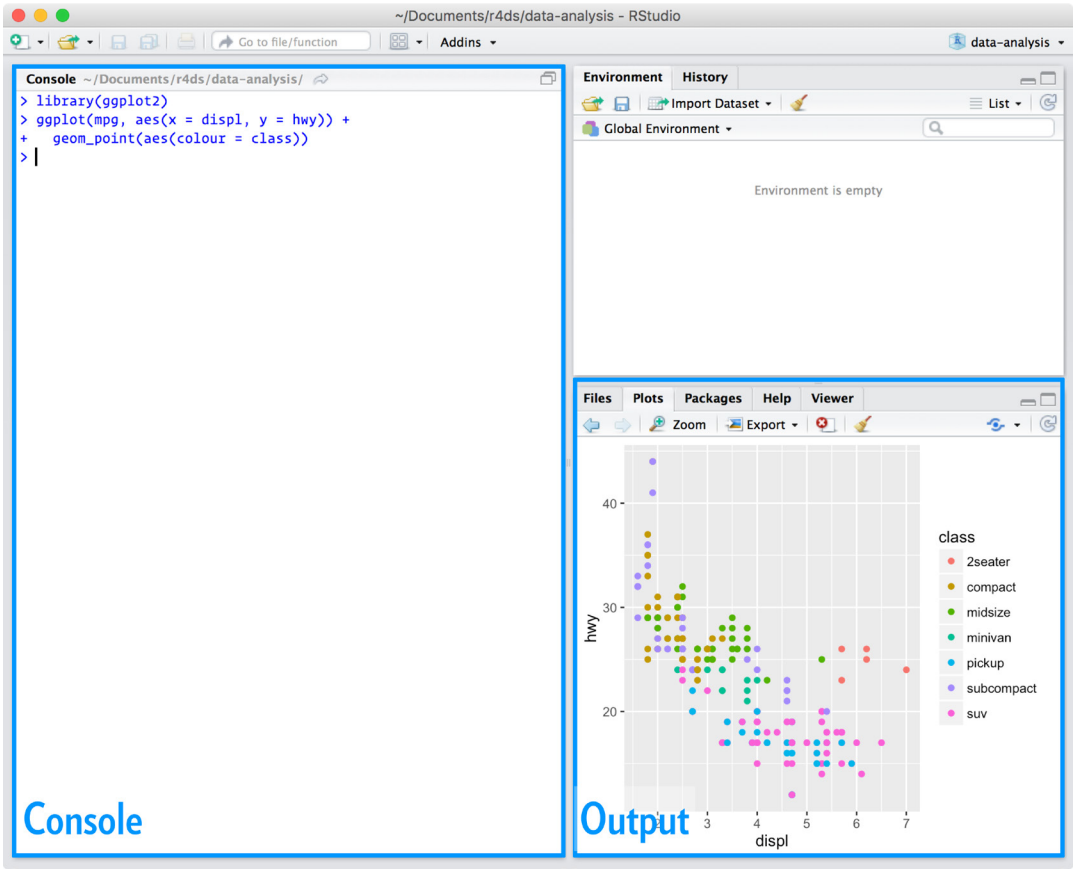
To download R, go to CRAN, the **c**omprehensive **R** **a**rchive **n**etwork. CRAN is composed of a set of mirror servers distributed around the world and is used to distribute R and R packages. Don't try and pick a mirror that's close to you: instead use the cloud mirror, https://cloud.r-project.org, which automatically figures it out for you.

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, which require you to reinstall all your packages, but putting it off only makes it worse.

1.4.2 RStudio

RStudio is an integrated development environment, or IDE, for R programming. Download and install it from http://www.rstudio.com/download. RStudio is updated a couple of times a year. When a new version is available, RStudio will let you know. It's a good idea to upgrade regularly so you can take advantage of the latest and greatest features. For this book, make sure you have at least RStudio 1.0.0.

When you start RStudio, you'll see two key regions in the interface:



For now, all you need to know is that you type R code in the console pane, and press enter to run it. You'll learn more as we go along!

1.4.3 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. The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.

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

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

On your own 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 to your computer. If you have problems installing, make sure that you are connected to the internet, and that https://cloud.r-project.org/ isn't blocked by your firewall or proxy.

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

```
library(tidyverse)
#> ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
#> ✓ ggplot2 3.3.0     ✓ purrr   0.3.4
#> ✓ tibble  3.0.1     ✓ dplyr   0.8.5
#> ✓ tidyr   1.0.3     ✓ stringr 1.4.0
#> ✓ readr   1.3.1     ✓ forcats 0.5.0
#> ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

This tells you that tidyverse is loading the ggplot2, tibble, tidyr, readr, purrr, and dplyr packages. These are considered to be 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, and optionally install them, by running `tidyverse_update()`.

1.4.4 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, just 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.

In this book we'll use three data packages from outside the tidyverse:

```
install.packages(c("nycflights13", "gapminder", "Lahman"))
```

These packages provide data on airline flights, world development, and baseball that we'll use to illustrate key data science ideas.

1.5 Running R code

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

```
1 + 2
#> [1] 3
#> [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, 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 into the console.

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

- Functions are in a code font and followed by parentheses, like `sum()`, or `mean()`.
- Other R objects (like data or function arguments) are in a code font, without parentheses, like `flights` or `x`.
- If we want to make it clear what 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.

1.6 Getting help and learning more

This book is not an island; there is no single resource that will allow you to master R. As you start to apply the techniques described in this book to your own data you will soon find questions that I do not answer. This section describes a few tips on how to get help, and to help you keep

learning.

If you get stuck, start with Google. Typically adding "R" to a query is enough to restrict it to relevant results: if the search isn't useful, it often means that there aren't any R-specific results available. Google is particularly useful for error messages. If you get an error message and you have no idea what it means, try googling it! Chances are that someone else has been confused by it in the past, and there will be help somewhere on the web. (If the error message isn't in English, run `Sys.setenv(LANGUAGE = "en")` and re-run the code; you're more likely to find help for English error messages.)

If Google doesn't help, try stackoverflow. Start by spending a little time searching for an existing answer, including `[R]` to restrict your search to questions and answers that use R. If you don't find anything useful, prepare a minimal reproducible example or **reprex**. A good reprex makes it easier for other people to help you, and often you'll figure out the problem yourself in the course of making it.

There are three things you need to include to make your example reproducible: required packages, data, and code.

1. **Packages** should be loaded at the top of the script, so it's easy to see which ones the example needs. This is a good time to check that you're using the latest version of each package; it's possible you've discovered a bug that's been fixed since you installed the package. For packages in the tidyverse, the easiest way to check is to run `tidyverse_update()`.

2. The easiest way to include **data** in a question is to use `dput()` to generate the R code to recreate it. For example, to recreate the `mtcars` dataset in R, I'd perform the following steps:

    1. Run `dput(mtcars)` in R
    2. Copy the output
    3. In my reproducible script, type `mtcars <-` then paste.

    Try and find the smallest subset of your data that still reveals the problem.

3. Spend a little bit of time ensuring that your **code** is easy for others to read:

    - Make sure you've used spaces and your variable names are concise, yet informative.
    - Use comments to indicate where your problem lies.

    - Do your best to remove everything that is not related to the problem.
      The shorter your code is, the easier it is to understand, and the easier it is to fix.

Finish by checking that you have actually made a reproducible example by starting a fresh R session and copying and pasting your script in.

You should also spend some time preparing yourself to solve problems before they occur. Investing a little time in learning R each day will pay off handsomely in the long run. One way is to follow what Hadley, Garrett, and everyone else at RStudio are doing on the RStudio blog. This is where we post announcements about new packages, new IDE features, and in-person courses. You might also want to follow Hadley (@hadleywickham) or Garrett (@statgarrett) on Twitter, or follow @rstudiotips to keep up with new features in the IDE.

To keep up with the R community more broadly, we recommend reading http://www.r-bloggers.com: it aggregates over 500 blogs about R from around the world. If you're an active Twitter user, follow the `#rstats` hashtag. Twitter is one of the key tools that Hadley uses to keep up with new developments in the community.

1.7 Acknowledgements

This book isn't just the product of Hadley and Garrett, but is the result of many conversations (in person and online) that we've had with the many people in the R community. There are a few people we'd like to thank in particular, because they have spent many hours answering our dumb questions and helping us to better think about data science:

- Jenny Bryan and Lionel Henry for many helpful discussions around working with lists and list-columns.
- The three chapters on workflow were adapted (with permission), from http://stat545.com/block002_hello-r-workspace-wd-project.html by Jenny Bryan.
- Genevera Allen for discussions about models, modelling, the statistical learning perspective, and the difference between hypothesis generation and hypothesis confirmation.
- Yihui Xie for his work on the bookdown package, and for tirelessly responding to my feature requests.
- Bill Behrman for his thoughtful reading of the entire book, and for trying it out with his data science class at Stanford.

- The #rstats twitter community who reviewed all of the draft chapters and provided tons of useful feedback.

- Tal Galili for augmenting his dendextend package to support a section on clustering that did not make it into the final draft.

This book was written in the open, and many people contributed pull requests to fix minor problems. Special thanks goes to everyone who contributed via GitHub:

Thanks go to all contributers in alphabetical order: adi pradhan, Ahmed ElGabbas, Ajay Deonarine, @Alex, Andrew Landgraf, bahadir cankardes, @batpigandme, @behrman, Ben Marwick, Bill Behrman, Brandon Greenwell, Brett Klamer, Christian G. Warden, Christian Mongeau, Colin Gillespie, Cooper Morris, Curtis Alexander, Daniel Gromer, David Clark, Derwin McGeary, Devin Pastoor, Dylan Cashman, Earl Brown, Eric Watt, Etienne B. Racine, Flemming Villalona, Gregory Jefferis, @harrismcgehee, Hengni Cai, Ian Lyttle, Ian Sealy, Jakub Nowosad, Jennifer (Jenny) Bryan, @jennybc, Jeroen Janssens, Jim Hester, @jjchern, Joanne Jang, John Sears, Jon Calder, Jonathan Page, @jonathanflint, Jose Roberto Ayala Solares, Julia Stewart Lowndes, Julian During, Justinas Petuchovas, Kara Woo, @kdpsingh, Kenny Darrell, Kirill Sevastyanenko, @koalabearski, @KyleHumphrey, Lawrence Wu, Matthew Sedaghatfar, Mine Cetinkaya-Rundel, @MJMarshall, Mustafa Ascha, @nate-d-olson, Nelson Areal, Nick Clark, @nickelas, Nirmal Patel, @nwaff, @OaCantona, Patrick Kennedy, @Paul, Peter Hurford, Rademeyer Vermaak, Radu Grosu, @rlzijdeman, Robert Schuessler, @robinlovelace, @robinsones, S'busiso Mkhondwane, @seamus-mckinsey, @seanpwilliams, Shannon Ellis, @shoili, @sibusiso16, @spirgel, Steve Mortimer, @svenski, Terence Teo, Thomas Klebel, TJ Mahr, Tom Prior, Will Beasley, @yahwes, Yihui Xie, @zeal626.

1.8 Colophon

An online version of this book is available at http://r4ds.had.co.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 https://bookdown.org which makes it easy to turn R markdown files into HTML, PDF, and EPUB.

This book was built with:

```
sessioninfo::session_info(c("tidyverse"))
#> ─ Session info ──────────────────────────────────────────────────────────────
#>  setting  value
#>  version  R version 4.0.0 (2020-04-24)
#>  os       Ubuntu 16.04.6 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language en_US.UTF-8
#>  collate  en_US.UTF-8
#>  ctype    en_US.UTF-8
#>  tz       UTC
#>  date     2020-05-08
#>
#> ─ Packages ──────────────────────────────────────────────────────────────────
#>  package      * version  date       lib source
#>  askpass        1.1      2019-01-13 [1] CRAN (R 4.0.0)
#>  assertthat     0.2.1    2019-03-21 [1] CRAN (R 4.0.0)
#>  backports      1.1.6    2020-04-05 [1] CRAN (R 4.0.0)
#>  base64enc      0.1-3    2015-07-28 [1] CRAN (R 4.0.0)
#>  BH             1.72.0-3 2020-01-08 [1] CRAN (R 4.0.0)
#>  broom          0.5.6    2020-04-20 [1] CRAN (R 4.0.0)
#>  callr          3.4.3    2020-03-28 [1] CRAN (R 4.0.0)
#>  cellranger     1.1.0    2016-07-27 [1] CRAN (R 4.0.0)
#>  cli            2.0.2    2020-02-28 [1] CRAN (R 4.0.0)
#>  clipr          0.7.0    2019-07-23 [1] CRAN (R 4.0.0)
#>  colorspace     1.4-1    2019-03-18 [1] CRAN (R 4.0.0)
#>  crayon         1.3.4    2017-09-16 [1] CRAN (R 4.0.0)
#>  curl           4.3      2019-12-02 [1] CRAN (R 4.0.0)
#>  DBI            1.1.0    2019-12-15 [1] CRAN (R 4.0.0)
#>  dbplyr         1.4.3    2020-04-19 [1] CRAN (R 4.0.0)
#>  desc           1.2.0    2018-05-01 [1] CRAN (R 4.0.0)
#>  digest         0.6.25   2020-02-23 [1] CRAN (R 4.0.0)
#>  dplyr        * 0.8.5    2020-03-07 [1] CRAN (R 4.0.0)
#>  ellipsis       0.3.0    2019-09-20 [1] CRAN (R 4.0.0)
#>  evaluate       0.14     2019-05-28 [1] CRAN (R 4.0.0)
```

```
#>  fansi          0.4.1    2020-01-08 [1] CRAN (R 4.0.0)
#>  farver         2.0.3    2020-01-16 [1] CRAN (R 4.0.0)
#>  forcats      * 0.5.0    2020-03-01 [1] CRAN (R 4.0.0)
#>  fs             1.4.1    2020-04-04 [1] CRAN (R 4.0.0)
#>  generics       0.0.2    2018-11-29 [1] CRAN (R 4.0.0)
#>  ggplot2      * 3.3.0    2020-03-05 [1] CRAN (R 4.0.0)
#>  glue           1.4.0    2020-04-03 [1] CRAN (R 4.0.0)
#>  gtable         0.3.0    2019-03-25 [1] CRAN (R 4.0.0)
#>  haven          2.2.0    2019-11-08 [1] CRAN (R 4.0.0)
#>  highr          0.8      2019-03-20 [1] CRAN (R 4.0.0)
#>  hms            0.5.3    2020-01-08 [1] CRAN (R 4.0.0)
#>  htmltools      0.4.0    2019-10-04 [1] CRAN (R 4.0.0)
#>  httr           1.4.1    2019-08-05 [1] CRAN (R 4.0.0)
#>  isoband        0.2.1    2020-04-12 [1] CRAN (R 4.0.0)
#>  jsonlite       1.6.1    2020-02-02 [1] CRAN (R 4.0.0)
#>  knitr          1.28     2020-02-06 [1] CRAN (R 4.0.0)
#>  labeling       0.3      2014-08-23 [1] CRAN (R 4.0.0)
#>  lattice        0.20-41  2020-04-02 [3] CRAN (R 4.0.0)
#>  lifecycle      0.2.0    2020-03-06 [1] CRAN (R 4.0.0)
#>  lubridate      1.7.8    2020-04-06 [1] CRAN (R 4.0.0)
#>  magrittr       1.5      2014-11-22 [1] CRAN (R 4.0.0)
#>  markdown       1.1      2019-08-07 [1] CRAN (R 4.0.0)
#>  MASS           7.3-51.5 2019-12-20 [3] CRAN (R 4.0.0)
#>  Matrix         1.2-18   2019-11-27 [3] CRAN (R 4.0.0)
#>  mgcv           1.8-31   2019-11-09 [3] CRAN (R 4.0.0)
#>  mime           0.9      2020-02-04 [1] CRAN (R 4.0.0)
#>  modelr         0.1.7    2020-04-30 [1] CRAN (R 4.0.0)
#>  munsell        0.5.0    2018-06-12 [1] CRAN (R 4.0.0)
#>  nlme           3.1-147  2020-04-13 [3] CRAN (R 4.0.0)
#>  openssl        1.4.1    2019-07-18 [1] CRAN (R 4.0.0)
#>  pillar         1.4.4    2020-05-05 [1] CRAN (R 4.0.0)
#>  pkgbuild       1.0.8    2020-05-07 [1] CRAN (R 4.0.0)
#>  pkgconfig      2.0.3    2019-09-22 [1] CRAN (R 4.0.0)
#>  pkgload        1.0.2    2018-10-29 [1] CRAN (R 4.0.0)
#>  plogr          0.2.0    2018-03-25 [1] CRAN (R 4.0.0)
#>  plyr           1.8.6    2020-03-03 [1] CRAN (R 4.0.0)
```

```
#>  praise         1.0.0    2015-08-11 [1] CRAN (R 4.0.0)
#>  prettyunits    1.1.1    2020-01-24 [1] CRAN (R 4.0.0)
#>  processx       3.4.2    2020-02-09 [1] CRAN (R 4.0.0)
#>  progress       1.2.2    2019-05-16 [1] CRAN (R 4.0.0)
#>  ps             1.3.3    2020-05-08 [1] CRAN (R 4.0.0)
#>  purrr        * 0.3.4    2020-04-17 [1] CRAN (R 4.0.0)
#>  R6             2.4.1    2019-11-12 [1] CRAN (R 4.0.0)
#>  RColorBrewer   1.1-2    2014-12-07 [1] CRAN (R 4.0.0)
#>  Rcpp           1.0.4.6  2020-04-09 [1] CRAN (R 4.0.0)
#>  readr        * 1.3.1    2018-12-21 [1] CRAN (R 4.0.0)
#>  readxl         1.3.1    2019-03-13 [1] CRAN (R 4.0.0)
#>  rematch        1.0.1    2016-04-21 [1] CRAN (R 4.0.0)
#>  reprex         0.3.0    2019-05-16 [1] CRAN (R 4.0.0)
#>  reshape2       1.4.4    2020-04-09 [1] CRAN (R 4.0.0)
#>  rlang          0.4.6    2020-05-02 [1] CRAN (R 4.0.0)
#>  rmarkdown      2.1      2020-01-20 [1] CRAN (R 4.0.0)
#>  rprojroot      1.3-2    2018-01-03 [1] CRAN (R 4.0.0)
#>  rstudioapi     0.11     2020-02-07 [1] CRAN (R 4.0.0)
#>  rvest          0.3.5    2019-11-08 [1] CRAN (R 4.0.0)
#>  scales         1.1.0    2019-11-18 [1] CRAN (R 4.0.0)
#>  selectr        0.4-2    2019-11-20 [1] CRAN (R 4.0.0)
#>  stringi        1.4.6    2020-02-17 [1] CRAN (R 4.0.0)
#>  stringr      * 1.4.0    2019-02-10 [1] CRAN (R 4.0.0)
#>  sys            3.3      2019-08-21 [1] CRAN (R 4.0.0)
#>  testthat       2.3.2    2020-03-02 [1] CRAN (R 4.0.0)
#>  tibble       * 3.0.1    2020-04-20 [1] CRAN (R 4.0.0)
#>  tidyr        * 1.0.3    2020-05-07 [1] CRAN (R 4.0.0)
#>  tidyselect     1.0.0    2020-01-27 [1] CRAN (R 4.0.0)
#>  tidyverse    * 1.3.0    2019-11-21 [1] CRAN (R 4.0.0)
#>  tinytex        0.22     2020-04-17 [1] CRAN (R 4.0.0)
#>  utf8           1.1.4    2018-05-24 [1] CRAN (R 4.0.0)
#>  vctrs          0.2.4    2020-03-10 [1] CRAN (R 4.0.0)
#>  viridisLite    0.3.0    2018-02-01 [1] CRAN (R 4.0.0)
#>  whisker        0.4      2019-08-28 [1] CRAN (R 4.0.0)
#>  withr          2.2.0    2020-04-20 [1] CRAN (R 4.0.0)
#>  xfun           0.13     2020-04-13 [1] CRAN (R 4.0.0)
```

```
#>  xml2          1.3.2    2020-04-23 [1] CRAN (R 4.0.0)
#>  yaml          2.2.1    2020-02-01 [1] CRAN (R 4.0.0)
#>
#> [1] /home/travis/R/Library
#> [2] /usr/local/lib/R/site-library
#> [3] /home/travis/R-bin/lib/R/library
```

4 Workflow: basics

You now have some experience running R code. I 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 will cause it to complain. But while you should expect to be a little frustrated, take comfort in that it's both 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 make sure you've got a solid foundation in running R code, and that you know about some of the most helpful RStudio features.

4.1 Coding basics

Let's review some basics we've so far omitted in the interests of getting you plotting as quickly as possible. You can use R as a calculator:

```
1 / 200 * 30
#> [1] 0.15
(59 + 73 + 2) / 3
#> [1] 44.7
sin(pi / 2)
#> [1] 1
```

You can create new objects with `<-`:

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

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. Don’t be lazy and use `=` : it will work, but it will cause confusion later. Instead, use RStudio’s keyboard shortcut: Alt + - (the minus sign). Notice that RStudio automagically surrounds `<-` with spaces, which is a good code formatting practice. Code is miserable to read on a good day, so giveyoureyesabreak and use spaces.

4.2 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 a convention for multiple words. I recommend **snake_case** where you separate lowercase words with `_` .

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

We’ll come back to code style later, in functions.

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.

Ooops, you made a mistake! `this_is_a_really_long_name` should have value 3.5 not 2.5. Use another keyboard shortcut to help you fix it. Type “this” then press Cmd/Ctrl + ↑. That will list all the commands you’ve typed that start 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
```

There's an implied contract between you and R: it will do the tedious computation for you, but in return, you must be completely precise in your instructions. Typos matter. Case matters.

4.3 Calling functions

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

```
function_name(arg1 = val1, arg2 = val2, ...)
```

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.

Press TAB once more when you've selected the function you want. RStudio will add matching opening (`(`) and closing (`)`) parentheses for you. Type the arguments `1, 10` and hit return.

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

Type this code and notice you get 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 that means you've forgotten either a `"` or a `)`. Either add the missing pair, or press ESCAPE to abort the expression and try again.

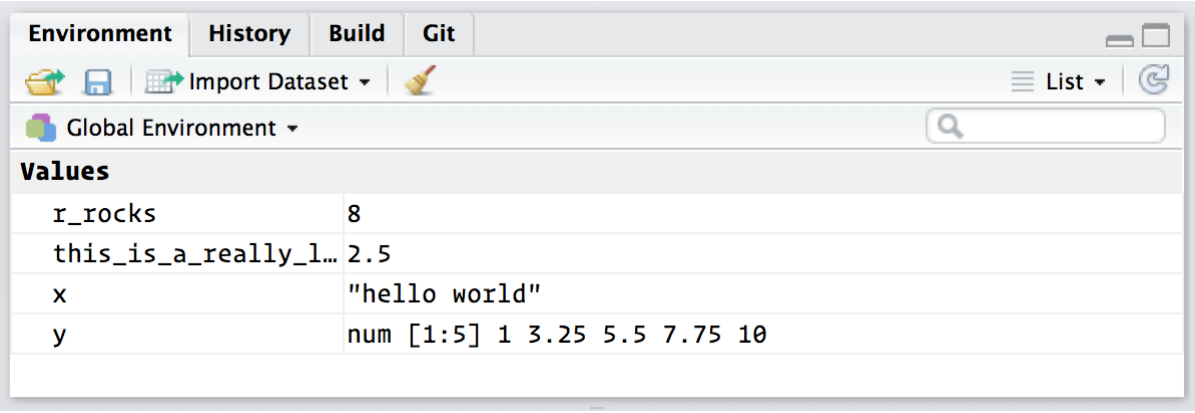
If you make an assignment, you don't get to see the value. You're then tempted to immediately double-check the result:

```
y <- seq(1, 10, length.out = 5)
y
#> [1]  1.00  3.25  5.50  7.75 10.00
```

This common action can be shortened by surrounding the assignment with parentheses, which causes assignment and "print to screen" to happen.

```
(y <- seq(1, 10, length.out = 5))
#> [1]  1.00  3.25  5.50  7.75 10.00
```

Now look at your environment in the upper right pane:



Here you can see all of the objects that you've created.

4.4 Exercises

1. Why does this code not work?

```
my_variable <- 10
my_variable
#> Error in eval(expr, envir, enclos): object 'my_variable' 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:

```
library(tidyverse)

ggplot(dota = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

fliter(mpg, cyl = 8)
filter(diamond, carat > 3)
```

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

27 R Markdown

27.1 Introduction

R Markdown provides an unified authoring framework for data science, combining your code, its results, and your prose commentary. R Markdown documents are fully reproducible and support dozens of output formats, like PDFs, Word files, slideshows, and more.

R Markdown 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.

R Markdown integrates a number of R packages and external tools. This means that help is, by-and-large, not available through `?`. Instead, as you work through this chapter, and use R Markdown in the future, keep these resources close to hand:

- R Markdown Cheat Sheet: *Help* > *Cheatsheets* > *R Markdown Cheat Sheet*,
- R Markdown Reference Guide: *Help* > *Cheatsheets* > *R Markdown Reference Guide*.

Both cheatsheets are also available at http://rstudio.com/cheatsheets.

27.1.1 Prerequisites

You need the **rmarkdown** package, but you don't need to explicitly install it or load it, as RStudio automatically does both when needed.

27.2 R Markdown basics

This is an R Markdown file, a plain text file that has the extension `.Rmd`:

````
---
title: "Diamond sizes"
date: 2016-08-25
output: html_document
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

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

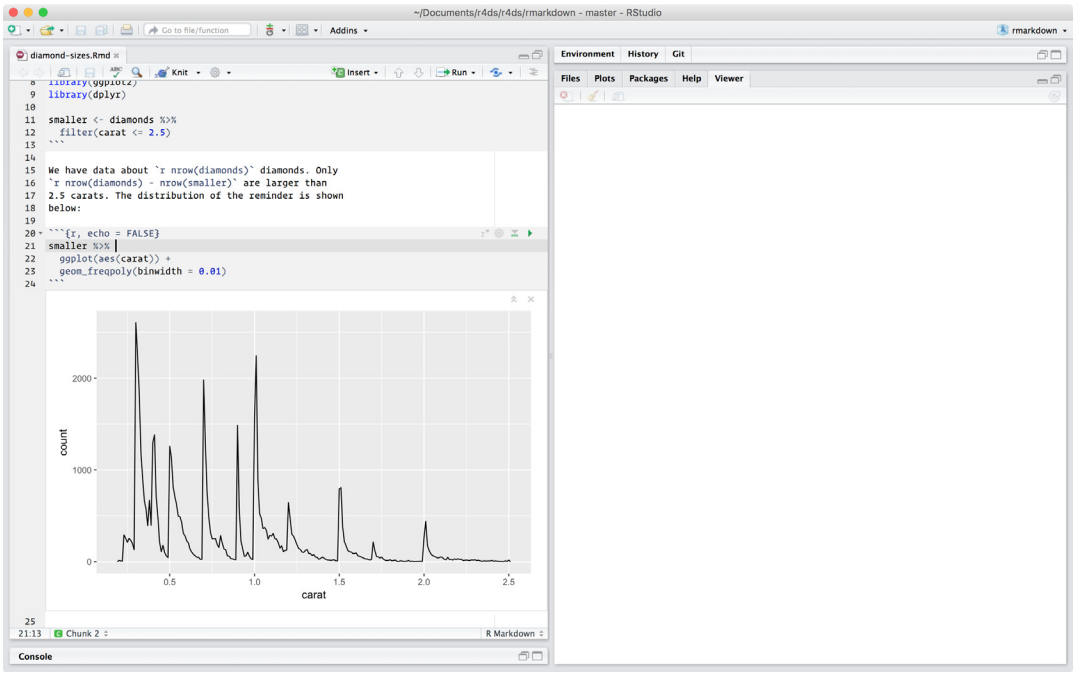
We have data about `r nrow(diamonds)` diamonds. Only
`r nrow(diamonds) - nrow(smaller)` are larger than
2.5 carats. The distribution of the remainder is shown
below:

```{r, echo = FALSE}
smaller %>%
  ggplot(aes(carat)) +
  geom_freqpoly(binwidth = 0.01)
```
````

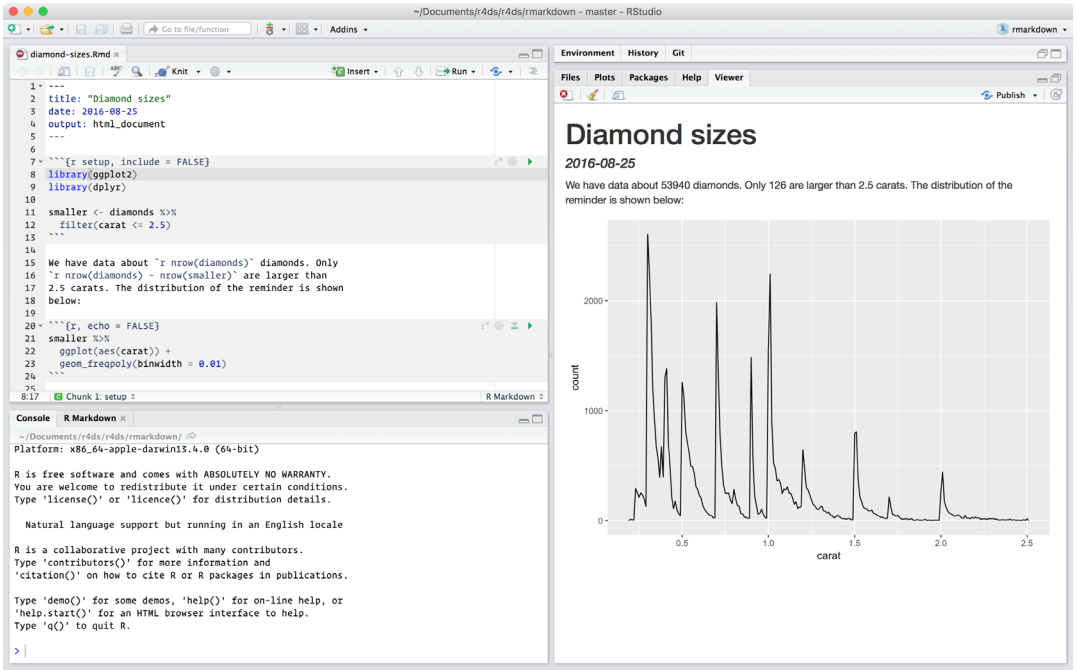
It contains three important types of content:

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_`.

When you open an `.Rmd`, you get a 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:



To produce a complete report containing all text, code, and results, click "Knit" or press Cmd/Ctrl + Shift + K. You can also do this programmatically with `rmarkdown::render("1-example.Rmd")`. This will display the report in the viewer pane, and create a self-contained HTML file that you can share with others.



When you **knit** the document, R Markdown sends the .Rmd file to **knitr**, http://yihui.name/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**, http://pandoc.org/, which is responsible for creating the finished file. 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 R markdown formats.

To get started with your own `.Rmd` file, select *File* > *New File* > *R Markdown…* in the menubar. 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 R Markdown work.

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

27.2.1 Exercises

1. Create a new notebook using *File* > *New File* > *R Notebook*. Read the instructions. Practice running the chunks. Verify that you can modify the code, re-run it, and see modified output.

2. Create a new R Markdown document with *File* > *New File* > *R Markdown…* Knit it by clicking the appropriate button. Knit it by using the appropriate keyboard short cut. Verify that you can modify the input and see the output update.

3. Compare and contrast the R notebook and R markdown files you created above. How are the outputs similar? How are they different? How are the inputs similar? How are they different? What happens if you copy the YAML header from one to the other?

4. Create one new R Markdown document for each of the three built-in formats: HTML, PDF and Word. Knit each of the three documents. How does the output differ? How does the input differ? (You may need to install LaTeX in order to build the PDF output — RStudio will prompt you if this is necessary.)

27.3 Text formatting with Markdown

Prose in `.Rmd` files is written in Markdown, a lightweight set of conventions for formatting plain text files. Markdown is designed to be easy to read and easy to write. It is also very easy to learn. The guide below shows how to use Pandoc's Markdown, a slightly extended version of Markdown that R Markdown understands.

```
Text formatting
------------------------------------------------------------

*italic*  or _italic_
**bold**   __bold__
`code`
superscript^2^ and subscript~2~

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

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

Links and images
------------------------------------------------------------
```

```
<http://example.com>

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

![optional caption text](path/to/img.png)

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*.

27.3.1 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 R Markdown quick reference, figure out how to:

    1. Add a footnote.
    2. Add a horizontal rule.
    3. Add a block quote.

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

27.4 Code chunks

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

1. The keyboard shortcut Cmd/Ctrl + Alt + I
2. The "Insert" button icon in the editor toolbar.
3. By manually typing the chunk delimiters ```` ```{r} ```` and ```` ``` ````.

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

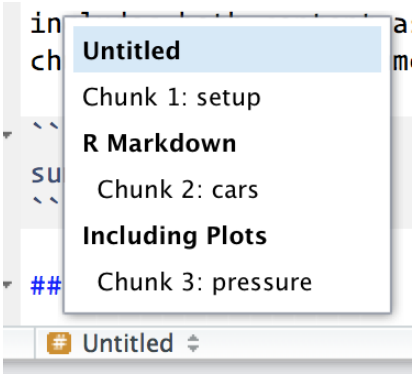
You can continue to run the code using the keyboard shortcut that by now (I hope!) you know and love: Cmd/Ctrl + Enter. However, chunks get a new keyboard shortcut: Cmd/Ctrl + Shift + Enter, which runs all the code in the chunk. Think of a chunk like a function. A chunk should be relatively self-contained,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

The following sections describe the chunk header which consists of ```` ```{r ````, followed by an optional chunk name, followed by comma separated options, followed by ```` } ````. Next comes your R code and the chunk end is indicated by a final ```` ``` ````.

27.4.1 Chunk name

Chunks can be given an optional name: ```` ```{r by-name} ````. 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 other important options.
3. You can set up networks of cached chunks to avoid re-performing expensive computations on every run. More on that below.

There is one chunk name that imbues special behaviour: `setup`. When you're in a notebook mode, the chunk named setup will be run automatically once, before any other code is run.

27.4.2 Chunk options

Chunk output can be customised with **options**, arguments 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 http://yihui.name/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 `.Rmd`. It's also useful if you're teaching R and want to deliberately include an error. The default, `error = FALSE` causes knitting to fail if there is a single error in the document.

The following table summarises which types of output each option supressess:

Option	Run code	Show code	Output	Plots	Messages	Warnings
`eval = FALSE`	-		-	-	-	-
`include = FALSE`		-	-	-	-	-
`echo = FALSE`		-				
`results = "hide"`			-			
`fig.show = "hide"`				-		
`message = FALSE`					-	
`warning = FALSE`						-

27.4.3 Table

By default, R Markdown 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 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4    4
#> Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4    1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.21 19.4  1  0    3    1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  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 27.1.

```
knitr::kable(
  mtcars[1:5, ],
  caption = "A knitr kable."
)
```

Table 27.1: A knitr kable.

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.62	16.5	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.88	17.0	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.6	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.21	19.4	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.0	0	0	3	2

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

There is also a rich set of options for controlling how figures are embedded. You'll learn about these in saving your plots.

27.4.4 Caching

Normally, each knit 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`. 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 raw_data}
rawdata <- readr::read_csv("a_very_large_file.csv")
```


```{r 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 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 `.Rmd` 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.info()`: it returns a bunch of information about the file including when it was last modified. Then you can write:

```
```{r raw_data, cache.extra = file.info("a_very_large_file.csv")}
rawdata <- readr::read_csv("a_very_large_file.csv")
```
```

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

I've used 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.

27.4.5 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 calling `knitr::opts_chunk$set()` in a code chunk. For example, when writing books and tutorials I set:

```
knitr::opts_chunk$set(
  comment = "#>",
  collapse = TRUE
)
```

This uses my preferred comment formatting, and ensures that the code and output are kept closely entwined. On the other hand, if you were preparing a report, you might set:

```
knitr::opts_chunk$set(
  echo = FALSE
)
```

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.

27.4.6 Inline code

There is one other way to embed R code into an R Markdown document: directly into the text, with: `` `r ` ``. This can be very useful if you mention properties of your data in the text. For example, in the example document I used at the start of the chapter I 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 knit, 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 accuracy, and a `big.mark` to make numbers easier to read. I'll often 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"
```

27.4.7 Exercises

1. Add a section that explores how diamond sizes vary by cut, colour, 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.Rmd` from https://github.com/hadley/r4ds/tree/master/rmarkdown. Add a section that describes the largest 20 diamonds, including a table that displays their most important attributes.

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

4. 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.

27.5 Troubleshooting

Troubleshooting R Markdown documents can be challenging because you are no longer in an interactive R environment, and you will need to learn some new tricks. The first thing you should always try is to recreate the problem in an interactive session. Restart R, then "Run all chunks" (either from 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 R markdown environment. You're going to need to systematically explore the options. The most common difference is the working directory: the working directory of an R Markdown 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 R markdown 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.

27.6 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 "yet another markup language", which is designed for representing hierarchical data in a way that's easy for humans to read and write. R Markdown uses it to control many details of the output. Here we'll discuss two: document parameters and bibliographies.

27.6.1 Parameters

R Markdown 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:

````
---
output: html_document
params:
  my_class: "suv"
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

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

# Fuel economy for `r params$my_class`s

```{r, message = FALSE}
ggplot(class, aes(displ, 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 `!r`. This is a good way to specify date/time parameters.

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

In RStudio, you can click the "Knit with Parameters" option in the Knit dropdown menu to set parameters, render, and preview the report in a single user friendly step. You can customise the dialog by setting other options in the header. See

http://rmarkdown.rstudio.com/developer_parameterized_reports.html#parameter_user_interfaces for more details.

Alternatively, if you need to produce many such paramterised reports, you can call `rmarkdown::render()` with a list of `params`:

```
rmarkdown::render("fuel-economy.Rmd", params = list(my_class = "suv"))
```

This is particularly powerful in conjunction with `purrr:pwalk()`. The following example creates a report for each value of `class` found in `mpg`. First we create a data frame that has one row for each class, giving the `filename` of the report and the `params`:

```
reports <- tibble(
  class = unique(mpg$class),
  filename = stringr::str_c("fuel-economy-", class, ".html"),
  params = purrr::map(class, ~ list(my_class = .))
)
reports
#> # A tibble: 7 x 3
#>   class   filename                params
#>   <chr>   <chr>                   <list>
#> 1 compact fuel-economy-compact.html <named list [1]>
#> 2 midsize fuel-economy-midsize.html <named list [1]>
#> 3 suv     fuel-economy-suv.html     <named list [1]>
#> 4 2seater fuel-economy-2seater.html <named list [1]>
#> 5 minivan fuel-economy-minivan.html <named list [1]>
#> 6 pickup  fuel-economy-pickup.html  <named list [1]>
#> # … with 1 more row
```

Then we match the column names to the argument names of `render()`, and use purrr's **parallel** walk to call `render()` once for each row:

```
reports %>%
  select(output_file = filename, params) %>%
  purrr::pwalk(rmarkdown::render, input = "fuel-economy.Rmd")
```

27.6.2 Bibliographies and Citations

Pandoc can automatically generate citations and a bibliography in a number of styles. To use this feature, specify a bibliography file using the `bibliography` field in your file's header. The field should contain a path from the directory that contains your .Rmd file to the file that contains the bibliography file:

```
bibliography: rmarkdown.bib
```

You can use many common bibliography formats including BibLaTeX, BibTeX, endnote, medline.

To create a citation within your .Rmd file, 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 R Markdown 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 I assume that the csl file is in the same directory as the .Rmd file. A good place to find CSL style files for common bibliography styles is http://github.com/citation-style-language/styles.

27.7 Learning more

R Markdown is still relatively young, and is still growing rapidly. The best place to stay on top of innovations is the official R Markdown website: http://rmarkdown.rstudio.com.

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 two free resources that will teach you about Git:

1. "Happy Git with R": a user friendly introduction to Git and GitHub from R users, by Jenny Bryan. The book is freely available online: http://happygitwithr.com

2. The "Git and GitHub" chapter of *R Packages*, by Hadley. You can also read it for free online: http://r-pkgs.had.co.nz/git.html.

I have also not touched on what you should actually write in order to clearly communicate the results of your analysis. To improve your writing, I 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/the-litigation-articles.html. They are aimed at lawyers, but almost everything applies to data scientists too.