PHW251 Week 3 Reader

Topic 1: Data Frames

Topic 2: Importing and Exporting Data

Topic 3: File Environment

Topic 1: Data Frames

Lecture 1.1: Data Frames, Part 1 – Overview and Constructing Data Frames

Overview

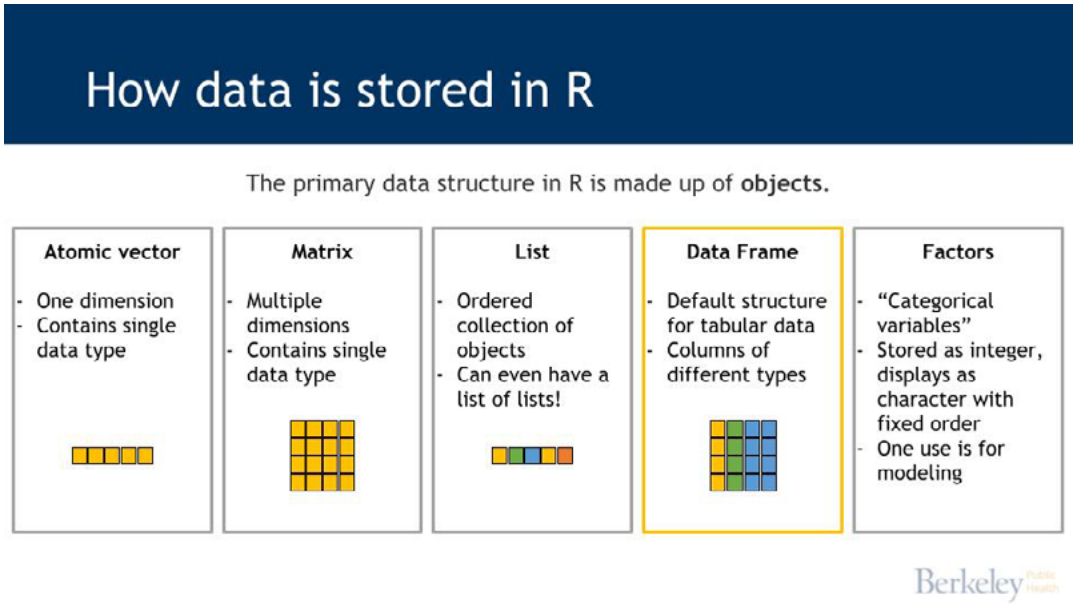
- Review: How data is stored in R
- Introduction to data frames and tibbles
    - Creating data frames
    - Modifying data frames
    - Exploring data frames
    - Tibbles

Objectives

1. Understand default data storage in R (data frames)
2. Create a data frame from scratch
3. Utilize functions available for exploring your data frame
4. Index and extract elements from data frames in base R
5. Add rows and columns to data frames using base R
6. Understand difference between data frames and tibbles
7. Access built-in data frames

Hi, everyone. In this video we're going to be talking a lot about DataFrames which are the primary way that tabular data is stored in R. We'll do a little bit of a review of how data is stored in R and the different types of objects then we'll focus in on DataFrames, creating them, modifying and exploring them and we'll also talk a little bit about tables which is just another way that tabular data is stored.

We have several objectives for this video: learning more about the default data storage in R or DataFrame, how to create a data frame from scratch, utilizing functions available to explore your DataFrame, index and extract elements from DataFrames and base R, odd rows and columns to data frames using base R, understanding the difference between DataFrames and tables, and also accessing some built-in DataFrames in R.



To start, let's revisit how data is stored in R. There's several different object types. For data storage in R, we have the atomic vector which basically is just one dimension. It has a single datatype in it, so all values have to be numeric or all have to be character, or all have to be logical.

Matrix is similar to a vector although it allows for multiple dimensions, but can still only contain one single datatype. A list is an ordered collection of objects so it can have different types of data in it. You could have one character value, one numeric, etc. You can even have a list of lists.

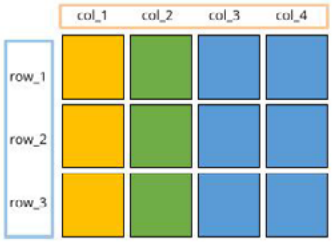
Then a DataFrame is the default structure for tabular data and allows for columns of different types and then we also have factors which are like categorical variables stored as integer but display as a character to provide fixed order. We'll talk about factors more later this semester.

Data Frames

- Table or two-dimensional array-like structure
- Can hold multiple types of data
- Creating data frames
  - Vectors (all of the same length)
  - Built-in data in R
  - Imported data files

Columns
- → Each column has a unique name
  - ◆ Be descriptive
  - ◆ Names are set at creation/import, but are modifiable
- → A column has a single data type

Rows
- → Each row name is unique
- → Defaults to numbers 1:n



Berkeley Public Health

But let's really dive into the DataFrame, become good friends of the DataFrame. To orient, a DataFrame is a two-dimensional array-like structure. It can hold multiple types of data.

To create a DataFrame you can do that from a set of vectors that are all of the same length. You can utilize built-in data in R which we'll talk about later in this video or you can also import data files which we will be talking about in a separate video.

A DataFrame has columns and rows. For columns, each column must have a unique name. They also can't have spaces or any special characters that start the column name. It needs to be a letter or a number that starts the column. The names are set at creation or import but are modifiable and a column has a single datatype. All the values in the column they would be stored as numeric or character or logical, and then similarly we have rows.

Each row name is unique. We focus less on those than the column names, but they do exist and the row names will default to just numbers one through n, n being the number of rows in your DataFrame.

Creating a data frame using data.frame()

Let's create our first data frame from scratch. We will create four county's testing data with the following variables / columns:

1. county
2. total tests
3. positive tests

data.frame() has two main parts: (1) including vectors of equal length and (2) options for how to format the data frame. You can read more details about the function design and options by running the following code:

```
?data.frame
```

Let's first create our three vectors of length four. *Note*: The length corresponds to the number of rows. We have four counties and therefore four rows.

```
c("Alameda", "Contra Costa", "Marin", "Mendocino")
c(500, 745, 832, 301)
c(43, 32, 30, 25)
```

Let's start doing some examples. I think one of the easiest ways to start is to create a DataFrame using the data.frame() function which is what this is. Let's create a DataFrame from scratch. We will create a DataFrame that has four county's testing data with the following variables and columns. We'll have one for the county, two for total tests, and three for positive tests. This is a made-up example, so the condition or disease that we're testing for is really irrelevant but just an example.

DataFrame the function has two main parts. You must include vectors of equal length and then you can also include some options for how to format the DataFrame. You can read more details about the DataFrame function by running this piece of code which I will run it now and just remind you that using the question mark before a term or a function name will populate this help view with some additional information. There's a lot of different arguments here that we're not going to cover, but you could certainly check that out.

To start, let's create three vectors all of length 4, then the length of the vector will correspond to the number of rows. You can consider these vectors each to be a set of values for each column. Well, I'll show you as we go forward, but we're just making three vectors here. This is an unnecessary step, but you can see how these vectors can then easily be plugged into the DataFrame() function as we move on.

I'm going to just run these each we're creating vectors. We're not saving them or anything so they will print below, and we have four values here. It's important that the vectors each correspond to the row that you're wanting to create. For example, our first row will be for Alameda with 500 total tests and 43 positive tests. I know that's not really clear that this is total test here and this is positive, but let's keep moving on.

We can plug these values directly into our data.frame() call and assign the output to an object called `testing`. On the left side of the equal sign is our **column name,** while on the right side is the **value** we are assigning.

```
testing <- data.frame(
  county = c("Alameda", "Contra Costa", "Marin", "Mendocino"),
  total_tests = c(500, 745, 832, 301),
  pos_tests = c(43, 32, 30, 25)
)

# testing_list <- list(
#   county = c("Alameda", "Contra Costa", "Marin", "Mendocino"),
#   total_tests = c(500, 745, 832, 301),
#   pos_tests = c(43, 32, 30, 25)
# )

testing # calling our object to output the results
```

```
        county total_tests pos_tests
1      Alameda         500        43
2 Contra Costa         745        32
3       Marin          832        30
4    Mendocino         301        25
```

Pretty neat, right?!

We can plug those vectors once we confirm that they each have four values they're storing as we want them to. We can plug those values directly into our DataFrame call and assign the output to an object that we're going to call testing. We're creating the object testing. We are using the DataFrame call or function, and we will create three different columns. This line is creating a column called county and the values in county are that vector that we ran above that had the four county names in it.

We'll create a second column called total_tests and you can see that has that second vector and the third has positive tests which has the third vector. What's important here especially if we're creating our own DataFrame for any reason is to just make sure these stay in the same order. So 500 is for Alameda, 745 for Contra Costa, 832 for Marin, and 301 for Mendocino, and same for positive tests. Let's run just this portion and you can see we've created a DataFrame here called testing.

If we open that up, we'll be able to see it in R pure here. You can also just call testing and it will print the DataFrame here and you can see the description says DF, that stands for DataFrame. We have four rows and three columns and when we print it here you can see a little bit of this structure here we have our county which are being stored as characters and then the tests are both being sort as double which is a type of numeric value in R. That's pretty cool.

Open the `testing` data frame from the environment tab. Do you see the row names on the left-hand side? We can make the row names more clear than 1 through 4. Let's change the row names to correspond to the row's county by using an option of `data.frame()`: row.names.

```
testing_rownames <- data.frame(
  county = c("Alameda", "Contra Costa", "Marin", "Mendocino"),
  total_tests = c(500, 745, 832, 301),
  pos_tests = c(43, 32, 30, 25),
  row.names = "county" # specifying the row name to use the `county` column
)

testing_rownames
```

```
             total_tests pos_tests
Alameda              500        43
Contra Costa         745        32
Marin                832        30
Mendocino            301        25
```

We'll move on in this course pretty quickly to importing data from other sources so you don't have to go through this step but it is definitely handy to know how to create these. These are really helpful for maybe small examples or reproducible examples.

We have the testing DataFrame open from our Environment tab and in here you can see that we have our column names that we specified in our DataFrame call. We also have some row names here and these are just row names 1 through 4, so they're not super helpful.

There is an option to change the names of the rows, and so in this example here we have a new DataFrame called testing_rownames where we have this same data being used to create a DataFrame but then we're utilizing the row name's argument to say we want the row names to be equal to the county, so this will utilize the information in the county column in order to populate the row names. Let's just run this.

The table that prints here doesn't look too different although it doesn't have a column name here, but if we also open this up from our Environment tab you'll see that it looks a little different than our original testing DataFrame where we had numeric row names.

Here you can see these have been moved over the county names. They don't have the county title and they're being stored differently. They're not being stored as part of the DataFrame values, rather the row names.

I think one of the use cases for this is if you want to print something and just have it look a little cleaner, it's not as practical if you have a DataFrame that you're doing a bunch of operations on. We aren't going to focus too much on the row names but can be handy for certain reasons.

Lecture 1.2: Data Frames, Part 2 – Exploring Data Frames in R

Exploring data frames in R

Useful functions

Now that we have our data frame, let's walk through a few useful functions for exploring our `testing` data.

Function	Returns	Example
nrow()	# of rows	4
ncol()	# of columns	3
dim()	Dimensions (row columns)	4 3
names()	Column names	"county" "total_tests" "pos_tests"
rownames()	Row names	"1" "2" "3" "4"
typeof()	Type of object	"list"
class()	Class	"data.frame"

str()	Structure	`'data.frame': 4 obs. of 3 variables:`
		`$ county : chr "Alameda" "Contra Costa" "Marin" "Mendocino"`
		`$ total_tests: num 500 745 832 301`
		`$ pos_tests : num 52 61 89 100`

```
# try them out!
nrow(testing)
```

Now that we have our testing DataFrames, so back to this original one, there are several functions in R that can help us explore our testing data. Many of them are listed here. There's more and there's more packages that provide even more options.

Here's a table that has the function and what it returns and then for our specific testing example, what the value is and so we have number of rows and number of columns, which you can see her foreign three dimensions will return the number of rows and the number of columns so you have four and three, names will return column names, row names, row names.

Then there's a couple of different ways to see what we're looking at. You'll notice that we have this type of which actually returns a list, which you might think we didn't just create the list, we created a DataFrame. But really what did DataFrame is on the back-end is just a list of vectors that's displayed in a much easier way.

Let's actually go back up and see what happens if we create this testing DataFrame as a list. Instead of DataFrame, we can actually just replace that with list. Let's run just this line. You can actually see it pop up over here on the right. It looks a little different.

Our other DataFrame say for OBS of three variables, this list of three. If we actually just click on it to open, you can see that we have all the same information in there, but this isn't usable in a way that we want to use the table. It's a way to store the information, but it's not helpful for our needs here. But on the backend, this can be easily converted into a DataFrame. I don't want to cause confusion by that, but I do want to highlight why it may be popping up as a list. I'm going to go ahead and comment this part out. We can go back.

But if you do class, you actually see that it is a DataFrame. We have also the option to do structure. Structure will give us more information than anything above, will see the dimensions. We see that it's a DataFrame and then we see for each column what type of information is each. We have our character into numeric fields.

```
# try them out!
nrow(testing)
ncol(testing)
dim(testing)
names(testing)
rownames(testing)
rownames(testing_rownames) # check out the difference!
typeof(testing)
class(testing)
str(testing)
```

All of these are listed here. You can go into this and try them out yourselves, but I'll do a couple just for example, you could run all of them at once but it's just going to spit out a long list of the results. I'm going to just run one at a time so it's easier.

We did number of rows and it returns four. We can do row names on our testing DataFrame and we get one, two, three, and four. We can also check it out with our testing row names DataFrame, the one where we changed the row names to the counties, and then we can also look at the structure. It's a little easier to see here then above.

These are all super handy. I definitely recommend anytime you've created a DataFrame or are loading a new one to use some of these to just check it out and make sure it looks like you expect.

Summary, table, plot

summary()

```
summary(testing)
```

```
    county           total_tests      pos_tests
 Length:4           Min.   :301.0   Min.   :25.00
 Class :character   1st Qu.:450.2   1st Qu.:28.75
 Mode  :character   Median :622.5   Median :31.00
                    Mean   :594.5   Mean   :32.50
                    3rd Qu.:766.8   3rd Qu.:34.75
                    Max.   :832.0   Max.   :43.00
```

table()

```
table(testing$total_tests)
```

```
301 500 745 832
  1   1   1   1
```

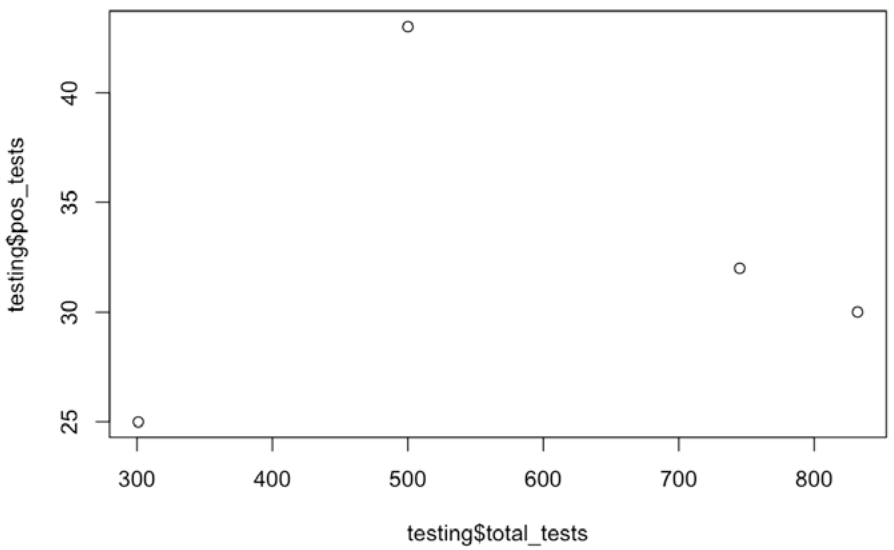
There's a couple additional tools that can be used as well that provide more information. Summary is a great function. If you run that, you will return information about each column. We have county. Because County is a character column, it's not super helpful in here where the summary function really helps us with numeric. We can see the min, the max, the quartiles, and the median and mean. That definitely comes in handy.

We also have the table function which will print out the number of rows with each value. If we do for total tests, we'll see that there's one row or one county that has each value.

For this example, the table is not a super helpful function but is if you have a really long dataset, maybe you're trying to see the frequency of age or something like that. You can use table on a full table but it gets pretty hard to interpret. Because now it's trying to basically do a cross tabulation between three columns, county total tests and positive tests. You can see here, it's just looking at the records that have positive tests equal 25 and that's just Mendocino and aligns with the total tests for three or one. Again, not super helpful for this example, but it is an option. But I'm going to change this back to huddle. They seem to have messed this up as well.

plot()

```
plot(testing$total_tests, testing$pos_tests)
```



We can also do plot. We'll talk a lot more about plotting and how to make beautiful plots later. But for now, you can use this basic plot function to indicate what something you want to compare on the x and y-axis. The first argument will be your x-axis and the second argument will be your y. Again, with this DataFrame of four, this isn't the most compelling visual, but it is an option especially for larger datasets.

Lecture 1.3: Data Frames, Part 3 – Indexing, Extracting, and Modifying Data Frames in Base R

Indexing & extracting using Base R

- Create a new data frame from an existing one using [ ]
- Create a vector of values using [[ ]] or $

	Example	Description
[ ]	`my_df[c("col_1","col_2")]` `my_df[1:2]`	Return data frame of specified columns based on name or number
[,]	`my_df[1:2,c(1,3)]`	Return data frame of specified rows and columns based on name or number
[[ ]]	`my_df[["col_1"]]`	Return vector of specified column values based on name or number
$	`my_df$col_1`	Return vector of specified column values based on name

Berkeley

Let's move on to talking a little bit about indexing and extracting using Base R, and by that I mean taking a subset of columns or rows from a DataFrame. You can pull values into a new vector using double brackets or a dollar sign. If you use double brackets, it might look something like this. You'll have the name of your DataFrame and then the name of the column.

You can also use the position of the column so in this case, I'm assuming since this is col 1, it would be the first position so you could just use the number one or you can use this dollar sign where on the left side you indicate the DataFrame and on the right side you indicate the column.

Then there is also the option to just create a smaller DataFrame from your larger DataFrame. If you're just interested in subsetting columns, you can use single brackets with no comma or anything. If you want to subset based on rows and columns, you would use a comma with the row information on the left and the column information on the right. Let's do some examples because I think that will make it clearer.

Indexing

As you can see, there are several ways to explore a data frame. Let's see the different ways we can pull out the counties in our data. In general, there is no one "right" way, but you may find yourself using these different methods at some point.

```
        testing[["county"]]
```

```
[1] "Alameda"      "Contra Costa" "Marin"        "Mendocino"
```

```
        testing[[1]] # the first column's index is 1
```

```
[1] "Alameda"      "Contra Costa" "Marin"        "Mendocino"
```

```
        testing$county
```

```
[1] "Alameda"      "Contra Costa" "Marin"        "Mendocino"
```

There are several ways you can pull out information from your DataFrame. If we're thinking about indexing, which in this sense means pulling out all information from a single column. There's different ways to do it. There's not necessarily one right way that you might have a preference. Let's say in our testing DataFrame, this one we've been looking at so far, we want to just extract the county names and we want to do that into a vector.

Maybe we want to use that vector somewhere else, or who knows, but there's a few different ways to do that. You can use these double brackets and specifically write out the column name so in this case county, you need to use quotes here to do this correctly.

You could also use the position so in this testing Data-Frame, county is the first column, so you can just use one or you can use the dollar sign and specify county. In this case, you don't have to use quotes. I know it's confusing when you're using the dollar sign, you never have to use quotes. If you're using brackets and you want to reference the column by name, you do need to use quotes.

Let's just quickly run through these one by one and you see they extract a vector with four elements that represent the county names. We'll do the second line. Same thing third line same thing. Again, these are all doing the same thing.

If you wanted to index that column, pull out that information and save it for use somewhere else, you can assign those values to a new object so in this case we have a new object called county_vector, and we're using some of the code from above.

Extracting

Pulling the column vector out of the data frame and assigning to a new object.

```
        county_vector <- testing[["county"]]
        county_vector
```

```
[1] "Alameda"      "Contra Costa" "Marin"        "Mendocino"
```

Outputting a data frame of the counties column.

```
        county_df <- testing["county"]
        county_df
```

```
        county
1      Alameda
2 Contra Costa
3        Marin
4    Mendocino
```

```
        county_df2 <- testing[1]
        county_df2
```

```
        county
1      Alameda
2 Contra Costa
3        Marin
4    Mendocino
```

Again, you can use any three of these to create a vector of just those county values so let's run that and we can see we have a vector that showed up on our environment panel called county_vector and we also see it printed below. That's pulling it into a vector.

You could also pull it into a DataFrame. Very similar except one small change, when we're pulling into a DataFrame, we will use single brackets. I don't have an explanation as to why this is. It is just how things work in Base R.

If we want to create a DataFrame, we can run this line and then can see that showed up to the right and we just have a small DataFrame with the four county values. You can print it here as well if you want to see it. Then to make the same point as above, you could also use just the value of the column index rather than the name. County DataFrame 2 should look exactly like county DataFrame. Those are some examples of maybe pulling out one column into a vector or a DataFrame.

Subsetting based on the column name or index.

```
        testing[ , c("total_tests","pos_tests")] # by name
```

```
  total_tests pos_tests
1         500        43
2         745        32
3         832        30
4         301        25
```

```
        testing[, c(2,3)] # by index
```

```
  total_tests pos_tests
1         500        43
2         745        32
3         832        30
4         301        25
```

Subsetting based on the row number.

```
        testing[c(1, 3), ] # rows 1 and 3
```

```
   county total_tests pos_tests
1 Alameda         500        43
3   Marin         832        30
```

Subsetting based on the row AND column number.

```
        testing[c(1, 3), c(2,3)] # rows 1 and 3 with columns 2 and 3
```

```
  total_tests pos_tests
1         500        43
3         832        30
```

We can expand that a little bit and we could create a new DataFrame that just includes two of the columns. In this case we're using the single brackets. We have a comma in here, we're not including anything to the left because we don't want to necessarily subset any of the rows. We're just interested in the columns which go to the right, and so here we're creating a vector by using C and then in parentheses listing the two columns that we're interested in. If you run this line, it will print out a new DataFrame with just total tests and positive tests.

I will note since we're not creating a new object, like we don't have a new DataFrame we're creating, this is just going to print that subset. It's not going to save it into a new DataFrame. But you could, if you wanted to do test subset and run, that you'll get a test subset over here of just those two columns and you take this out, but that's how you would create a new DataFrame. Then to make a similar point to previously, if you run this using the column index values, you'll get the same thing.

On the flip side of that, we can also subset based on row in a similar way, although we don't necessarily have the same opportunity to use the row names. Especially because we don't typically have row names in our DataFrame besides just the row index so we'll use the row index here. This is saying we just want to keep Rows 1 and 3. We're keeping Alameda and Marin, you can see the original row number still show up there I'm not going to confirm that it did that correctly.

We can also subset based on row and column. This is a combination of the two examples above. We want to keep Row 1 and 3 and Columns 2 and 3 so if we run that, we'll get just total tests and positive tests for Rows 1 and 3. Again, there's a lot of different ways to do this and we'll cover more within the Tidyverse that might be more intuitive, but this is how you do it in Base R and some people prefer to work within Base R.

Reassigning

What if we needed to fix a value in a data frame? We can do that with reassignment.

Here's an example of modifying Contra Costa to Yolo. We are modifying the cell value of row 2 and column 1: [2, 1]

```
testing[2, 1] <- "Yolo"
testing
```

```
     county total_tests pos_tests
1   Alameda         500        43
2      Yolo         745        32
3     Marin         832        30
4 Mendocino         301        25
```

Update the positive tests.

```
testing$pos_tests <- c(52, 61, 89, 100)
testing
```

```
     county total_tests pos_tests
1   Alameda         500        52
2      Yolo         745        61
3     Marin         832        89
4 Mendocino         301       100
```

One other modification that might be made is we want to actually fix a value in a DataFrame. We can do that with a reassignment.

In this example, we're going back to our testing DataFrame. What if someone said, I made a data entry error and Contra Costa really supposed to be Yolo. In that case, we could modify these specific cell. We are in Row 2, Column 1 so we can do testing, index that exact cell Row 2, Column 1 and say we want it to be Yolo. If we do this, it will actually change the value within that county column within our testing DataFrame.

We could do something similar if we wanted to change the values of the positive tests, we could use a slightly different method, we could say for the testing DataFrame, the pos_tests column, we can supply a new vector and this has to be the same length, then it will assign these in order so it'd be 52 first and 61, 89 and 100 so let's go ahead and do this. You can see those values are placed in the positive test column.

I do just want to highlight this type of thing will actually update your DataFrame even if you feel like you're not creating a new object, this is injecting into the existing DataFrame new value so should be used with caution.

Modifying data frames (in base R)

Rows

We just received new testing data from Lassen County and want to add to the `testing` data frame. Let's create a list object because a list allows for *different* types of data. We'll then combine the list with our `testing` data frame using `rbind()`.

```
# create our lassen list object
lassen <- list("Lassen", 1200, 73)

# combine the data frame with the list
# note: since we're reassigning `testing` we are overwriting the previous object'
testing <- rbind(testing, lassen)

testing
```

```
     county total_tests pos_tests
1   Alameda         500        52
2      Yolo         745        61
3     Marin         832        89
4 Mendocino         301       100
5    Lassen        1200        73
```

We can also row bind with two data frames.

```
# create our second testing data frame
testing2 <- data.frame(
  county = c("Imperial", "Riverside"),
  total_tests = c(410, 760),
  pos_tests = c(84, 57)
)

# combine the two data frames
testing <- rbind(testing, testing2)
testing
```

```
     county total_tests pos_tests
1   Alameda         500        52
2      Yolo         745        61
3     Marin         832        89
4 Mendocino         301       100
5    Lassen        1200        73
6  Imperial         410        84
7 Riverside         760        57
```

Now we have modified data frames in a way, extracting or inserting a new value into a specific cell or replacing column values but we can also modify data frames by adding new columns or rows, and that's probably more common than needing to update what's already in there. Hopefully data entry errors are addressed upstream. We want to start with creating a new row.

Let's say we got additional information from Lassen County and we want to add to the testing data frame. We can create a list. We have the county name. We want to create the list in the same order as our data frame. We have our county name, total tests and positive tests, so we'll go ahead and run this line, and you can see we have a list called Lassen within our environment.

Then we can go ahead and add this to our testing data frame using rbind, which is short for row bind, and we'll supply the data frame we're wanting to update and then the list that we want to add to the data frame.

In this case we are actually going to be overwriting the previous objects data by setting this rbind equal to testing. It will overwrite what we already had in testing, so let's go ahead and run that and we can just print it here. But you see we have what we had before plus a row for Lassen. You can also do this row bind with two data frames. Here there's a second testing data frame with additional information from Imperial and Riverside. Run that.

You can see that shows up over here, and we're going to go ahead and do the same thing where we overwrite our original data frame and specify the original data frame and the new one, and when we run that we now have seven rows of data in our data frame.

I will add you don't have to create this new object with the same name, you could call it something else like testing3 and then you would still have your original testing, testing2 and testing3 which might be the right way to go if you're testing something out. But this also keeps your environment clean if you don't mind having your original item overwritten.

Columns

Let's add another column `id`. We can create a vector called `id` and use `cbind()` to column bind the vector to the data frame.

```
id <- 1:7
cbind(testing, id)
```

```
     county total_tests pos_tests id
1   Alameda         500        52  1
2      Yolo         745        61  2
3     Marin         832        89  3
4 Mendocino         301       100  4
5    Lassen        1200        73  5
6  Imperial         410        84  6
7 Riverside         760        57  7
```

We can also directly add a column to the data frame.

```
testing$id <- 1:7
testing
```

```
     county total_tests pos_tests id
1   Alameda         500        52  1
2      Yolo         745        61  2
3     Marin         832        89  3
4 Mendocino         301       100  4
5    Lassen        1200        73  5
6  Imperial         410        84  6
7 Riverside         760        57  7
```

What if we wanted to calculate another column based on other columns? We can calculate percent positivity with the following formula: `100 * positive tests / total test`.

```
testing$pct_pos <- round(100 * testing$pos_tests / testing$total_tests, 1)
testing
```

```
     county total_tests pos_tests id pct_pos
1   Alameda         500        52  1    10.4
2      Yolo         745        61  2     8.2
3     Marin         832        89  3    10.7
4 Mendocino         301       100  4    33.2
5    Lassen        1200        73  5     6.1
6  Imperial         410        84  6    20.5
7 Riverside         760        57  7     7.5
```

Let's add a couple of new columns. First we can add another column called id. This is repetitive of what is in there for the row name, but that's okay. We'll use cbind which is column bind to add the vector into the data frame. First we want to create our vector. We're going to create a vector called id. We can actually see it show up over to the right and has values one through seven and if we do cbind to testing, it will add in the id values.

I will say using cbind like this, it doesn't update our original data frame. It's just going to print what that's going to look like.

But we can also do it this other way which we will do to actually update our data frame. We can use the dollar sign to say which data frame we're working on, what we want our column name to be, and the values for that column. This injects one through seven straight into our data frame, so if we just run this line and reopen testing we have a new id column with values one through seven. I'll go ahead and just put that here.

Finally, if we wanted to calculate another column based on the data already in our table we could do something similar. We can specify what that new column is. We have our testing data frame. We want the new column to be percent pos and there should be a unique column name. If it's the same as another column name in the table it's going to overwrite that column. We're using this formula so we have percent positive calculation here.

Then we're using the round function to round our data to just one digit. But if we run this you'll see that we get this new percent positive column. You can see it in our data frame up here as well. That's pretty handy. We'll learn other ways to add new columns using other tools later but this is definitely a valid option.

Restructuring

Sometimes we need to modify our data frames by re-ordering or removing columns and rows.

We can re-order columns by column indexes.

```
# note that we have 5 columns
# let's change the order to: id, county, pos_test, total_test, pct_pos
testing <- testing[, c(4, 1, 3, 2, 5)]
testing
```

```
  id    county pos_tests total_tests pct_pos
1  1   Alameda        52         500    10.4
2  2      Yolo        61         745     8.2
3  3     Marin        89         832    10.7
4  4 Mendocino       100         301    33.2
5  5    Lassen        73        1200     6.1
6  6  Imperial        84         410    20.5
7  7 Riverside        57         760     7.5
```

We can also re-order based on values in a specified column, such as by ordering by county alphabetically.

```
testing <- testing[order(testing$county), ]
testing
```

```
  id    county pos_tests total_tests pct_pos
1  1   Alameda        52         500    10.4
6  6  Imperial        84         410    20.5
5  5    Lassen        73        1200     6.1
3  3     Marin        89         832    10.7
4  4 Mendocino       100         301    33.2
7  7 Riverside        57         760     7.5
2  2      Yolo        61         745     8.2
```

We might also want to restructure our data frame. If you look at how we have our data frame now it's in a weird order. We can do that a couple of different ways.

We could use the data frame extraction tool that we used before where we use the single brackets and we say within our testing data frame we want to keep all the rows but we want our columns in this order, so this is basically putting the fourth column, the id column first, and then switching the order of total tests and positive tests. If we run that, you can see that we did switch the order.

You could also do this using column names as long as they're in quotes. Then we can also reorder based on the values in a specific column so this would reorder the rows. You can see up here we have our rows the same, but if we wanted to put our data in alphabetical order by county, we could use the order function.

We have testing single brackets and we're operating on the rows here, so we use order and what order will do is it will just take the values in that county column and sort them alphabetically and then we're not doing anything to the column so to the right of the comma is blank here.

You can see we have our row names and id which are repetitive but we have county positive tests, total tests, and then the percent positive last and it also changed it in our environment as well.

Similarly, we can remove columns by index.

```
        testing[, c(1, 3, 4, 5)] # removing county
```

```
  id pos_tests total_tests pct_pos
1  1        52         500    10.4
6  6        84         410    20.5
5  5        73        1200     6.1
3  3        89         832    10.7
4  4       100         301    33.2
7  7        57         760     7.5
2  2        61         745     8.2
```

A shorthand way to remove by index is by "subtracting" or placing "-" before the column index

```
        testing[, c(-2)]
```

```
  id pos_tests total_tests pct_pos
1  1        52         500    10.4
6  6        84         410    20.5
5  5        73        1200     6.1
3  3        89         832    10.7
4  4       100         301    33.2
7  7        57         760     7.5
2  2        61         745     8.2
```

Related to that we could also remove columns by indexing them. We could say we want the testing data frame, all the rows, but then we only want these four columns and so if we run that we actually remove the county column.

Another way to do this especially irrelevant if you have a data frame with a lot of columns, you could do, this will do the same thing. Up here we're saying 1, 3, 4, and 5, if you say, I want just a -2 that will drop the second column. Since we're not assigning these two new objects, this is not actually changing in our saved data frame but those are some examples of things you could do for clean up if you wanted.

Lecture 1.4: Data Frames, Part 4 – Tibbles

Tibbles (tidyverse)

Tibbles are the modern data frame and are a core feature of the tidyverse packages. Tibbles were created to improve flexibility:

- Removed row names
- Allow for spaces within column names
- Allow for column names to start with special characters
- Can handle vector of length 1 by automatically assigning the value to all rows
- Can reference columns within a tibble easily
- Improved readability when outputting to the console

We've talked a lot about DataFrames and base R. DataFrames really are the core way for storing tabular data.

But do want to mention, especially as we move forward in the course, we'll talk more about the Tidyverse and tibbles are really a more modern DataFrame and they're really a core feature of the Tidyverse package.

If you're within Tidyverse and creating tables, they're likely actually tibbles, and they're really created to improve some flexibility. They remove row names. They allow for spaces within column names which is not allowed in DataFrames. They allow for column names to start with special characters. They can handle vectors of length 1 by automatically assigning the value to all rows, which we'll look at an example of shortly. They can reference columns within a tibble a little easier.

You don't necessarily have to use the dollar sign or the brackets, and then often improved readability when outputting to the console.

Creating a tibble is similar to a data frame, but notice how we can:

1. Assign "CA" to all rows of the state column
2. Create a variable Percent Positive Tests that has spaces in the name
3. Reference pos_tests and total_tests before these variables are created

```
library(tidyverse)

testing_tib <- tibble(
  county = c("Alameda", "Contra Costa", "Marin"),
  state = "CA", # assigning one value across all rows
  total_tests = c(500, 745, 832),
  pos_tests = c(43, 32, 30),
  `Percent Positive Tests` = round(100 * pos_tests / total_tests, 1) # spaces in
)

testing_tib
```

```
# A tibble: 3 × 5
  county       state total_tests pos_tests `Percent Positive Tests`
  <chr>        <chr>       <dbl>     <dbl>                    <dbl>
1 Alameda      CA            500        43                      8.6
2 Contra Costa CA            745        32                      4.3
3 Marin        CA            832        30                      3.6
```

Let's create a tibble. We're basically replicating our original testing DataFrame, but with a couple of additional features. We have our county column, and I'll note tibble just has replaced where we use the DataFrame called before. But otherwise this looks really similar.

We have our county, column, we have a state column, and because everything in our DataFrame is from California, we can actually just say state equals California and that will assign that value to all rows. We have our total tests and positive test columns as well. Then we've created a percent positive column that's similar to the one we created above.

But there's two notes here really, you can create a column name with a space. You do need to use these tick marks. These are not single quotes there ticks and typically, or at least on my keyboard that is just to the left of the one on the numbers at the top of your keyboard. But we're able to create this column name, that looks a little more cleaned up. Then we're also able to create a column base on these two columns above, which is something we can't do in DataFrames until after the DataFrame is initially created.

Let's just go ahead and run and print this and you can see how this looks. We'll say allowing spaces in the column name is not super practical for programming because then you have to refer to the column name within the ticks every time. But it is really helpful if you're wanting to easily create a cleaned up table that has a nice-looking column name. That's my personal recommendation.

Otherwise, you can do similar things with tibbles as described above. But as we get more into the Tidyverse, we'll talk about ways for subsetting and extracting that look a little different than base R.

Lecture 1.5: Data Frames, Part 5 – Built-in Data

Built in data

As you're learning R, it's helpful to use built in data rather than creating your own from scratch.

```
# to see built in data
data()

# to load one of these datasets, such as admissions to UC Berkeley
data("UCBAdmissions")

UCBAdmissions

UCBAdmissions_df <- as.data.frame(UCBAdmissions)
```

You can also access data through packages.

```
# an example of a package that's popular for data exploration & visualization
library(palmerpenguins)

penguins_tib <- penguins
```

To end this video, I want to just talk a little bit about built-in data in R. This whole video has involved creating a DataFrame from scratch. But one cool thing about R is that there is built-in data. You can actually see all the built-in data if you run this just data call.

All of these are available in R for use and you can load them by using data and then specifying which datasets you're interested in. There's a UC Berkeley admissions dataset which seems appropriate to take a peek at. If you run this data, UCB admissions, it will load it into your environment. It says promise here, that means it hasn't been fully loaded, but it's there for you to access.

If we were to just try to print this, you actually see that it doesn't print out in a way that super helpful. It looks like when we use the table function on a DataFrame.

One thing you can do here is I'm going to just use the same name and you can do as.data.frame and run this line. Will show up as a DataFrame, which is much easier if you open it up. You can see, Oh, that's what I was trying to be shown when I printed that before. We have admission status, gender, department, and basically the number admitted and rejected for each gender by department. There's a ton of datasets you can see there.

You can also access data through packages. There's one package that's pretty popular for data exploration and visualization and that is the penguins or palmer penguins library, but there's a penguins DataFrame. You can see this actually loads as a table because it's part of this library. It's handled different than some of the automatic datasets here, but yeah, this is available for you to use.

You'll notice it doesn't really show up in our environment. You could say you want to save a version that you can actually work off of. Let's run this. You can see penguins tip showed up over here and I did tip because I had seen it loaded as a table, but you can do whatever you want.

Happy exploring. I definitely encourage looking at some of these datasets that are available in R and we will be using them in some of our future lectures for sure.

28 A field guide to base R

28.1 Introduction

To finish off the programming section, we're going to give you a quick tour of the most important base R functions that we don't otherwise discuss in the book. These tools are particularly useful as you do more programming and will help you read code you'll encounter in the wild.

This is a good place to remind you that the tidyverse is not the only way to solve data science problems. We teach the tidyverse in this book because tidyverse packages share a common design philosophy, increasing the consistency across functions, and making each new function or package a little easier to learn and use. It's not possible to use the tidyverse without using base R, so we've actually already taught you a **lot** of base R functions: from `library()` to load packages, to `sum()` and `mean()` for numeric summaries, to the factor, date, and POSIXct data types, and of course all the basic operators like `+`, `-`, `/`, `*`, `|`, `&`, and `!`. What we haven't focused on so far is base R workflows, so we will highlight a few of those in this chapter.

After you read this book, you'll learn other approaches to the same problems using base R, data.table, and other packages. You'll undoubtedly encounter these other approaches when you start reading R code written by others, particularly if you're using StackOverflow. It's 100% okay to write code that uses a mix of approaches, and don't let anyone tell you otherwise!

In this chapter, we'll focus on four big topics: subsetting with `[`, subsetting with `[[` and `$`, the apply family of functions, and `for` loops. To finish off, we'll briefly discuss two essential plotting functions.

28.1.1 Prerequisites

This package focuses on base R so doesn't have any real prerequisites, but we'll load the tidyverse in order to explain some of the differences.

```
library(tidyverse)
```

28.2 Selecting multiple elements with `[`

`[` is used to extract sub-components from vectors and data frames, and is called like `x[i]` or `x[i, j]`. In this section, we'll introduce you to the power of `[`, first showing you how you can use it with vectors, then how the same principles extend in a straightforward way to two-dimensional (2d) structures like data frames. We'll then help you cement that knowledge by showing how various dplyr verbs are special cases of `[`.

28.2.1 Subsetting vectors

There are five main types of things that you can subset a vector with, i.e., that can be the `i` in `x[i]`:

24 **vector of positive integers**. Subsetting with positive integers keeps the elements at those positions:

```
x <- c("one", "two", "three", "four", "five")
x[c(3, 2, 5)]
#> [1] "three" "two"   "five"
```

By repeating a position, you can actually make a longer output than input, making the term "subsetting" a bit of a misnomer.

```
x[c(1, 1, 5, 5, 5, 2)]
#> [1] "one"  "one"  "five" "five" "five" "two"
```

2. **A vector of negative integers**. Negative values drop the elements at the specified positions:

```
x[c(-1, -3, -5)]
#> [1] "two"  "four"
```

3. **A logical vector**. Subsetting with a logical vector keeps all values corresponding to a `TRUE` value. This is most often useful in conjunction with the comparison functions.

```
x <- c(10, 3, NA, 5, 8, 1, NA)

# All non-missing values of x
x[!is.na(x)]
#> [1] 10  3  5  8  1

# All even (or missing!) values of x
x[x %% 2 == 0]
#> [1] 10 NA  8 NA
```

Unlike `filter()`, `NA` indices will be included in the output as `NA`s.

4. **A character vector**. If you have a named vector, you can subset it with a character vector:

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
#> xyz def
#>   5   2
```

As with subsetting with positive integers, you can use a character vector to duplicate individual entries.

5. **Nothing**. The final type of subsetting is nothing, `x[]`, which returns the complete `x`. This is not useful for subsetting vectors, but as we'll see shortly, it is useful when subsetting 2d structures like tibbles.

28.2.2 Subsetting data frames

There are quite a few different ways[1] that you can use `[` with a data frame, but the most important way is to select rows and columns independently with `df[rows, cols]`. Here `rows` and `cols` are vectors as described above.

For example, `df[rows, ]` and `df[, cols]` select just rows or just columns, using the empty subset to preserve the other dimension.

Here are a couple of examples:

```
df <- tibble(
  x = 1:3,
  y = c("a", "e", "f"),
  z = runif(3)
)

# Select first row and second column
df[1, 2]
#> # A tibble: 1 × 1
#>   y
#>   <chr>
#> 1 a

# Select all rows and columns x and y
df[, c("x" , "y")]
#> # A tibble: 3 × 2
#>       x y
#>   <int> <chr>
#> 1     1 a
#> 2     2 e
#> 3     3 f

# Select rows where `x` is greater than 1 and all columns
df[df$x > 1, ]
#> # A tibble: 2 × 3
#>       x y         z
#>   <int> <chr> <dbl>
#> 1     2 e     0.834
#> 2     3 f     0.601
```

We'll come back to `$` shortly, but you should be able to guess what `df$x` does from the context: it extracts the `x` variable from `df`. We need to use it here because `[` doesn't use tidy evaluation, so you need to be explicit about the source of the `x` variable.

There's an important difference between tibbles and data frames when it comes to `[`. In this book, we've mainly used tibbles, which *are* data frames, but they tweak some behaviors to make your life a little easier. In most places, you can use "tibble" and "data frame" interchangeably, so when we want to draw particular attention to R's built-in data frame, we'll write `data.frame`. If `df` is a `data.frame`, then `df[, cols]` will return a vector if `col` selects a single column and a data frame if it selects more than one column. If `df` is a tibble, then `[` will always return a tibble.

```
df1 <- data.frame(x = 1:3)
df1[, "x"]
#> [1] 1 2 3
```

```
df2 <- tibble(x = 1:3)
df2[, "x"]
#> # A tibble: 3 × 1
#>       x
#>   <int>
#> 1     1
#> 2     2
#> 3     3
```

One way to avoid this ambiguity with `data.frame`s is to explicitly specify `drop = FALSE`:

```
df1[, "x" , drop = FALSE]
#>   x
#> 1 1
#> 2 2
#> 3 3
```

28.2.3 dplyr equivalents

Several dplyr verbs are special cases of `[`:

- `filter()` is equivalent to subsetting the rows with a logical vector, taking care to exclude missing values:

  ```
  df <- tibble(
    x = c(2, 3, 1, 1, NA),
    y = letters[1:5],
    z = runif(5)
  )
  df |> filter(x > 1)

  # same as
  df[!is.na(df$x) & df$x > 1, ]
  ```

  Another common technique in the wild is to use `which()` for its side-effect of dropping missing values: `df[which(df$x > 1), ]`.

- `arrange()` is equivalent to subsetting the rows with an integer vector, usually created with `order()`:

  ```
  df |> arrange(x, y)

  # same as
  df[order(df$x, df$y), ]
  ```

  You can use `order(decreasing = TRUE)` to sort all columns in descending order or `-rank(col)` to sort columns in decreasing order individually.

- Both `select()` and `relocate()` are similar to subsetting the columns with a character vector:

```
df |> select(x, z)

# same as
df[, c("x", "z")]
```

Base R also provides a function that combines the features of `filter()` and `select()`[2] called `subset()`:

```
df |>
  filter(x > 1) |>
  select(y, z)
#> # A tibble: 2 × 2
#>   y           z
#>   <chr>   <dbl>
#> 1 a     0.157
#> 2 b     0.00740
```

```
# same as
df |> subset(x > 1, c(y, z))
```

This function was the inspiration for much of dplyr's syntax.

28.2.4 Exercises

1. Create functions that take a vector as input and return:

    a. The elements at even-numbered positions.
    b. Every element except the last value.
    c. Only even values (and no missing values).

2. Why is `x[-which(x > 0)]` not the same as `x[x <= 0]`? Read the documentation for `which()` and do some experiments to figure it out.

28.3 Selecting a single element with `$` and `[[`

`[`, which selects many elements, is paired with `[[` and `$`, which extract a single element. In this section, we'll show you how to use `[[` and `$` to pull columns out of data frames, discuss a couple more differences between `data.frames` and tibbles, and emphasize some important differences between `[` and `[[` when used with lists.

28.3.1 Data frames

`[[` and `$` can be used to extract columns out of a data frame. `[[` can access by position or by name, and `$` is specialized for access by name:

```
tb <- tibble(
  x = 1:4,
  y = c(10, 4, 1, 21)
```

```
)

# by position
tb[[1]]
#> [1] 1 2 3 4

# by name
tb[["x"]]
#> [1] 1 2 3 4
tb$x
#> [1] 1 2 3 4
```

They can also be used to create new columns, the base R equivalent of `mutate()`:

```
tb$z <- tb$x + tb$y
tb
#> # A tibble: 4 × 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1    10    11
#> 2     2     4     6
#> 3     3     1     4
#> 4     4    21    25
```

There are several other base R approaches to creating new columns including with `transform()`, `with()`, and `within()`. Hadley collected a few examples at https://gist.github.com/hadley/1986a273e384fb2d4d752c18ed71bedf.

Using `$` directly is convenient when performing quick summaries. For example, if you just want to find the size of the biggest diamond or the possible values of `cut`, there's no need to use `summarize()`:

```
max(diamonds$carat)
#> [1] 5.01

levels(diamonds$cut)
#> [1] "Fair"      "Good"      "Very Good" "Premium"   "Ideal"
```

dplyr also provides an equivalent to `[[`/`$` that we didn't mention in Chapter 4: `pull()`. `pull()` takes either a variable name or variable position and returns just that column. That means we could rewrite the above code to use the pipe:

```
diamonds |> pull(carat) |> max()
#> [1] 5.01

diamonds |> pull(cut) |> levels()
#> [1] "Fair"      "Good"      "Very Good" "Premium"   "Ideal"
```

There are a couple of important differences between tibbles and base `data.frame`s when it comes to `$`. Data frames match the prefix of any variable names (so-called **partial matching**) and don't complain if a column doesn't exist:

```
df <- data.frame(x1 = 1)
df$x
#> [1] 1
df$z
#> NULL
```

Tibbles are more strict: they only ever match variable names exactly and they will generate a warning if the column you are trying to access doesn't exist:

```
tb <- tibble(x1 = 1)

tb$x
#> Warning: Unknown or uninitialised column: `x`.
#> NULL
tb$z
#> Warning: Unknown or uninitialised column: `z`.
#> NULL
```

For this reason we sometimes joke that tibbles are lazy and surly: they do less and complain more.

28.3.3 Lists

`[[` and `$` are also really important for working with lists, and it's important to understand how they differ from `[`. Let's illustrate the differences with a list named `l`:

```
l <- list(
  a = 1:3,
  b = "a string",
  c = pi,
  d = list(-1, -5)
)
```

- `[` extracts a sub-list. It doesn't matter how many elements you extract, the result will always be a list.

```
str(l[1:2])
#> List of 2
#>  $ a: int [1:3] 1 2 3
#>  $ b: chr "a string"

str(l[1])
#> List of 1
#>  $ a: int [1:3] 1 2 3

str(l[4])
```

```
#> List of 1
#>  $ d:List of 2
#>   ..$ : num -1
#>   ..$ : num -5
```

Like with vectors, you can subset with a logical, integer, or character vector.

- `[[` and `$` extract a single component from a list. They remove a level of hierarchy from the list.

```
str(l[[1]])
#>  int [1:3] 1 2 3

str(l[[4]])
#> List of 2
#>  $ : num -1
#>  $ : num -5

str(l$a)
#>  int [1:3] 1 2 3
```

The difference between `[` and `[[` is particularly important for lists because `[[` drills down into the list while `[` returns a new, smaller list. To help you remember the difference, take a look at the unusual pepper shaker shown in Figure 28.1. If this pepper shaker is your list `pepper`, then, `pepper[1]` is a pepper shaker containing a single pepper packet. `pepper[2]` would look the same, but would contain the second packet. `pepper[1:2]` would be a pepper shaker containing two pepper packets. `pepper[[1]]` would extract the pepper packet itself.

Figure 28.1: (Left) A pepper shaker that Hadley once found in his hotel room. (Middle) `pepper[1]`. (Right) `pepper[[1]]`

This same principle applies when you use 1d `[` with a data frame: `df["x"]` returns a one-column data frame and `df[["x"]]` returns a vector.

28.3.4 Exercises

1. What happens when you use `[[` with a positive integer that's bigger than the length of the vector? What happens when you subset with a name that doesn't exist?

2. What would `pepper[[1]][1]` be? What about `pepper[[1]][[1]]`?

28.4 Apply family

In Chapter 27, you learned tidyverse techniques for iteration like `dplyr::across()` and the map family of functions. In this section, you'll learn about their base equivalents, the **apply family**. In this context apply and map are synonyms because another way of saying "map a function over each element of a vector" is "apply a function over each element of a vector". Here we'll give you a quick overview of this family so you can recognize them in the wild.

The most important member of this family is `lapply()`, which is very similar to `purrr::map()`[3]. In fact, because we haven't used any of `map()`'s more advanced features, you can replace every `map()` call in Chapter 27 with `lapply()`.

There's no exact base R equivalent to `across()` but you can get close by using `[` with `lapply()`. This works because under the hood, data frames are lists of columns, so calling `lapply()` on a data frame applies the function to each column.

```
df <- tibble(a = 1, b = 2, c = "a", d = "b", e = 4)

# First find numeric columns
num_cols <- sapply(df, is.numeric)
num_cols
#>     a     b     c     d     e
#>  TRUE  TRUE FALSE FALSE  TRUE

# Then transform each column with lapply() then replace the original values
df[, num_cols] <- lapply(df[, num_cols, drop = FALSE], \(x) x * 2)
df
#> # A tibble: 1 × 5
#>       a     b c     d         e
#>   <dbl> <dbl> <chr> <chr> <dbl>
#> 1     2     4 a     b         8
```

The code above uses a new function, `sapply()`. It's similar to `lapply()` but it always tries to simplify the result, hence the `s` in its name, here producing a logical vector instead of a list. We don't recommend using it for programming, because the simplification can fail and give you an unexpected type, but it's usually fine for interactive use. purrr has a similar function called `map_vec()` that we didn't mention in Chapter 27.

Base R provides a stricter version of `sapply()` called `vapply()`, short for **v**ector apply. It takes an additional argument that specifies the expected type, ensuring that simplification occurs the same way regardless of the

Topic 2: Importing and Exporting Data

Lecture 2: Importing/Exporting Data

1 Overview

- Importing data (.csv, .xlsx, RDS)
- Exporting data (.csv, .xlsx, RDS)

1.1 Objectives

1. Learn how to import data into R
2. Learn how to export data from R

In this video, we're going to be talking about importing data into R and exporting data from R, really going to focus on CSV files as that will likely be the most common, especially for this course. But there are also some resources in here for other types of data files. Now run this library call.

But basically R can read in a variety of source datatypes and automatically store these data as data frames or tibbles. They will be stored as a data frame using some of these functions and as a tibble using others. But here's some of the popular packages and their relevant functions. You can see for text CSV, TSV files, there are some options in base R, readr is part of the tidyverse.

There's also a package called data table that has some functions for reading in data. For Excel, there is the readxl package. For SAS and Stata data, there's a package called haven, and then for RDS, which is the native R data structure, you can typically just use base R. Again, we're really going to focus on CSVs here, but there are some additional examples at the end.

1.2.1 readr

We could use base R's read.csv(), but `readr::read_csv()` is an improved version.

- Faster
- Creates a tibble
  - Prevents character conversion to a factor
  - Avoids row names
  - Less picky about column names
- Reproducibility - code for readr works the same on everyone's computer

1.2.2 readr example

```
readr:: read_csv(
  file,                                  (1)
  col_names = TRUE,                      (2)
  col_types = NULL,                      (3)
  skip = 0,                              (4)
  n_max = Inf,                           (5)
  guess_max = min(1000, n_max),
  skip_empty_rows = TRUE                 (6)
)
```

(1) Specify a path to your file. When using datahub, it's easier to upload a file to your cloud directory, though you can also download files by specifying a URL.

(2) Specify column names (TRUE will pull names from csv, can also provide a vector of names)

(3) Specify column types

(4) Number of lines to skip before reading data in

(5) Indicate number of rows to read in (start small if file is large)

(6) Should blank rows be ignored altogether?

You can check out examples of Excel and SAS/SPSS/Stata imports in Section 1.5

To load a CSV, we could use Base R's read.csv, but readr::read_csv is really an improved version, so highly recommend just using that. It's faster. It creates a tibble which if you've listened to the data frame video, knows there's some differences and maybe some benefits to a tibble, like preventing character conversion to a factor, rather than just keeping the data as characters, avoiding row names altogether and then being less picky about column names and then also there's an aspect of reproducibility.

Code for readr works the same on pretty much everyone's computer because we're using external package. Here's our readr example. There are within read CSV, or read_delim a lot of different options. I have just the readr reference guide up here.

There's a ton of options, a ton of specification. We're going to highlight what we think is most relevant at this point and we're going to focus on read CSV, but read_delim works very similar. You'll specify a path to your file if you are using DataHub. If you have a file on your home computer or your desktop and you want to get your data into R, it's easiest to go through the process of uploading to some area within your repository and you can browse to identify where maybe you just want to save it, in the home area, and then you can choose a file from your desktop. For this course, we're going to use a lot of files that are available through URLs, whether available through a GitHub repository or some other public open data source.

1.2.2 readr example

```
readr:: read_csv(
  file,                                    (1)
  col_names = TRUE,                        (2)
  col_types = NULL,                        (3)
  skip = 0,                                (4)
  n_max = Inf,                             (5)
  guess_max = min(1000, n_max),
  skip_empty_rows = TRUE                   (6)
)
```

(1) Specify a path to your file. When using datahub, it's easier to upload a file to your cloud directory, though you can also download files by specifying a URL.

(2) Specify column names (TRUE will pull names from csv, can also provide a vector of names)

(3) Specify column types

(4) Number of lines to skip before reading data in

(5) Indicate number of rows to read in (start small if file is large)

(6) Should blank rows be ignored altogether?

You can check out examples of Excel and SAS/SPSS/Stata imports in Section 1.5

This example is going to focus on a URL, but you can also use this file path to direct to places within your R directory here. You will see some examples of that as we move throughout the course. You can specify column names. This argument can actually take the value true, false, or you can provide a vector of names. If you specify true, it will pull the names from the top row in the CSV. Otherwise you can provide a vector of names. You could say false, but then your columns will be nameless. I think true or providing a vector is most relevant.

You can select column types. If you leave this null, it will guess. But you can specify. You can indicate the number of lines to skip before reading data and you might do this for a variety of different reasons. You can indicate the number of rows to read in. That's at this end max. If it's ionized, that's infinite. I'll read in the whole thing, but you could say 10 or 100 if you just want to see what your data looks like. This guess max argument is what is used. If column types aren't specified, it will look throughout your data to a certain extent to try to guess the column type.

Then skip row of zeros equals true will tell R whether or not you want to load in fully blank rows or not. Again, this is a CSV example. At the end of this script, there are some examples of Excel and SAS imports that we'll talk about briefly.

1.3 Importing

1.3.1 Import

Let's first try importing a file from the web. We will read in a .csv file.

```
file_path_csv <- "https://data.chhs.ca.gov/dataset/8eb8839f-52a1-410b-8c4a-5b1c1678bbc2/r

# read in the file with default options
esd <- read_csv(file_path_csv)

# looks good!!
head(esd)
```

```
# A tibble: 6 × 5
  County     Year Percentage Lower95CI Upper95CI
  <chr>     <dbl> <chr>      <chr>     <chr>
1 Alameda    2013 0.54       0.5       0.59
2 Alpine     2013 *          n/a       n/a
3 Amador     2013 0.3        0.17      0.44
4 Berkeley   2013 0.58       n/a       n/a
5 Butte      2013 0.47       0.28      0.66
6 Calaveras  2013 0.43       n/a       n/a
```

Let's try importing a file from the web. Let's read in a CSV file. This is a file that's on the availability of electronic smoking devices by county provided on the open data portal for California Health and Human Services. Then just run this line by line. We're going to save this file path. This just makes it cleaner later on, you don't have to do this. You could insert this directly into the read CSV function. If you are going to be loading something from your working directory, you could specify that in your file path here, and then we'll go ahead and create a new object that actually reads in the file from that path. We're just going to call it ESD. You can see it loaded 186 observations of five variables.

Then we could either open it over here, I like to do that sometimes, or you can use the head function. It will print out the first six rows. It's a little hard to see here because it smashes together, but it's a good option. We have this read in. But as you can see, the column names maybe aren't what we want there. In title case, the competence intervals or maybe not how we would like them labeled.

Let's go ahead and specify the column names. I guess I'll go back up here. We didn't specify any of our arguments of here. That means it runs with all the defaults and the default is for column names to be true, and it will pull from the first row to select the column name.

1.3.2 Adding options

Now let's add options, starting with column names.

```
esd_colnames <- read_csv(
  file_path_csv,
  col_names = c("county", "yr", "pct", "ci_l", "ci_u")
)

head(esd_colnames)
```

```
# A tibble: 6 × 5
  county   yr    pct        ci_l      ci_u
  <chr>    <chr> <chr>      <chr>     <chr>
1 County   Year  Percentage Lower95CI Upper95CI
2 Alameda  2013  0.54       0.5       0.59
3 Alpine   2013  *          n/a       n/a
4 Amador   2013  0.3        0.17      0.44
5 Berkeley 2013  0.58       n/a       n/a
6 Butte    2013  0.47       0.28      0.66
```

Something is wrong! Our previous column headers are now in row 1 due to us specifying column names. We can fix this issue by skipping the first row.

1.3.3 Skipping rows

```
esd_colnames <- read_csv(
  file_path_csv,
  col_names = c("county", "yr", "pct", "ci_cl", "ci_u"),
  skip = 1
)

head(esd_colnames)
```

```
# A tibble: 6 × 5
  county       yr pct   ci_cl ci_u
  <chr>     <dbl> <chr> <chr> <chr>
1 Alameda    2013 0.54  0.5   0.59
2 Alpine     2013 *     n/a   n/a
3 Amador     2013 0.3   0.17  0.44
4 Berkeley   2013 0.58  n/a   n/a
5 Butte      2013 0.47  0.28  0.66
6 Calaveras  2013 0.43  n/a   n/a
```

But down here we can specify, so we'll just say county, year, percent abbreviation in the competence, interval abbreviations. I'm going to go ahead and run this whole thing and then I'll print the first six rows. You can see we have the new column names and we save this as a new object, so it's a little bit easier to see in this window.

You can actually see now that we specify the column names, we're actually loading in what the column names are before into our first row. We don't really want that. We will use the skip argument and then we'll set it equal to one. That means I will skip one row and then start loading the data in. We'll go ahead and run that and you can see now we don't have that extra row. If we go back to this esc colnames, that row is no longer there.

1.3.4 Specifying column types

Let's take a look at the data structure. Notice that the percentage `pct` and confidence interval columns `ci_cl` `ci_u` are read in as a character?

```
str(esd_colnames)
```

```
spc_tbl_ [186 × 5] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ county: chr [1:186] "Alameda" "Alpine" "Amador" "Berkeley" ...
 $ yr    : num [1:186] 2013 2013 2013 2013 2013 ...
 $ pct   : chr [1:186] "0.54" "*" "0.3" "0.58" ...
 $ ci_cl : chr [1:186] "0.5" "n/a" "0.17" "n/a" ...
 $ ci_u  : chr [1:186] "0.59" "n/a" "0.44" "n/a" ...
 - attr(*, "spec")=
  .. cols(
  ..   county = col_character(),
  ..   yr = col_double(),
  ..   pct = col_character(),
  ..   ci_cl = col_character(),
  ..   ci_u = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

We can specify the data type with the col_types option.

```
esd_colnames <- read_csv(
  file_path_csv,
  col_names = c("county", "yr", "pct", "ci_cl", "ci_u"),
  col_types = cols(
    county = col_character(),
    yr     = col_double(),
    pct    = col_number(),
    ci_cl  = col_number(),
    ci_u   = col_number()
    ),
  skip = 1
)
```

Now we have the data in, it looks good. The column names are also a little easier to manage. But we wanted to look at the structure. We will run the STR function. You can see that counties reading in his character year as numeric, but everything else is reading in his character, which we really want it to be numeric. We're going to go ahead and specify what we want the column type options to be.

We have the same arguments as before, but we're going to add in col_types. This is a little bit of a wordy argument, but basically you'll use this cols function. Then for each column, you will want to specify what type of column it is through using this col, col_character, col_double, col_number. These three are numbers because they are decimals and the year is here as a double because it is not a decimal. Then we'll keep R skip = one. We'll go ahead and run this chunk and look at the structure again. Now we see that we have R county as character and then everything else as numeric. That's looking good.

```
Warning: One or more parsing issues, call `problems()` on your data frame for details,
e.g.:
  dat <- vroom(...)
  problems(dat)
```

```
str(esd_colnames)
```

```
spc_tbl_ [186 × 5] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ county: chr [1:186] "Alameda" "Alpine" "Amador" "Berkeley" ...
 $ yr    : num [1:186] 2013 2013 2013 2013 2013 ...
 $ pct   : num [1:186] 0.54 NA 0.3 0.58 0.47 0.43 0.33 0.52 0.5 0.58 ...
 $ ci_cl : num [1:186] 0.5 NA 0.17 NA 0.28 NA NA 0.41 NA 0.48 ...
 $ ci_u  : num [1:186] 0.59 NA 0.44 NA 0.66 NA NA 0.63 NA 0.69 ...
 - attr(*, "spec")=
  .. cols(
  ..   county = col_character(),
  ..   yr = col_double(),
  ..   pct = col_number(),
  ..   ci_cl = col_number(),
  ..   ci_u = col_number()
  .. )
 - attr(*, "problems")=<externalptr>
```

But we did get a warning here. Let's handle that. The warning is around parsing issues. You could leave this. It's honestly probably fine, but you do want to make sure that the data is coming in like you expect and that something's not being nulled out that you want to actually keep. You can check the errors using the problems col. But first, you need to col the data frame with the problems. If we run this, we get a list of the problems here. This isn't super easy to read in browser, so I'm actually going to make a data frame of the problems just because it's a little easier to look at. I'm going to do problems.

1.3.5 Specifying NA values

Our column types were fixed, but we got a warning about parsing issues.

```
# check the errors with problems(); we first need to call the data frame with
# the problems.
esd_colnames
problems()

problems_df <- problems(esd_colnames)
```

Looks like unexpected strings were present in the data, which read_csv() was unsure how to handle. We can specify these strings so that they are changed into NA values.

```
# no warnings!
# save a clean file for exporting
esd_clean <- read_csv(
  file_path_csv,
  col_names = c("county", "yr", "pct", "ci_cl", "ci_u"),
  col_types = cols(
    county = col_character(),
    yr     = col_double(),
    pct    = col_number(),
    ci_cl  = col_number(),
    ci_u   = col_number()
    ),
  skip = 1,
  na = c("", "NA", "*", "n/a")
)

str(esd_clean)
```

```
spc_tbl_ [186 × 5] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ county: chr [1:186] "Alameda" "Alpine" "Amador" "Berkeley" ...
 $ yr    : num [1:186] 2013 2013 2013 2013 2013 ...
 $ pct   : num [1:186] 0.54 NA 0.3 0.58 0.47 0.43 0.33 0.52 0.5 0.58 ...
 $ ci_cl : num [1:186] 0.5 NA 0.17 NA 0.28 NA NA 0.41 NA 0.48 ...
 $ ci_u  : num [1:186] 0.59 NA 0.44 NA 0.66 NA NA 0.63 NA 0.69 ...
 - attr(*, "spec")=
  .. cols(
  ..   county = col_character(),
  ..   yr = col_double(),
  ..   pct = col_number(),
  ..   ci_cl = col_number(),
  ..   ci_u = col_number()
  .. )
 - attr(*, "problems")=<externalptr>
```

The object I'm concerned about. I'm going to run that, and that actually opens us a new data frame and I can just see it a little better. Basically, what this is saying is that's an inventory of all the issues. For row 2, column 3, we were expecting a number, but we got a asterisk. If we go to esd_colnames for row 2, column 3, we actually see an NA now. Because R didn't know what to do with it, it already replaced it with an NA, but we want to know and make sure that everything looks right. It looks like all of these are similar, and it's these N/As or an asterisk that's getting incorrect or R doesn't quite know how to parse it, so it's making it NA by default. We could leave it like that, but we prefer to load it with no warnings.

We can also use this NA argument. You can provide a vector of values that you want to tell it. These values mean NA. Missing can be read in as NA. A string of NA could be read in as NA. In this case, an asterisk, and then also the N/A. Let's load this again and look at the structure. Now we see we don't have the same parsing issues, so we have our esd_clean That likely looks exactly like ESD column names, but we are more secure about that since we've told R how to handle those unexpected strings.

1.5 Additional Examples

1.5.1 Excel Sheets

```
readxl::read_excel(
  path,
  sheet = NULL,                    (1)
  range = NULL,
  col_names = TRUE,                (2)
  col_types = NULL,
  na = "",                         (3)
  trim_ws = TRUE,                  (4)
  skip = 0,                        (5)
  n_max = Inf,                     (6)
  guess_max = min(1000, n_max),
  .name_repair = "unique"
)
```

(1) Specify sheet and/or range of cells within sheet

(2) Control column names and types

(3) Handling NA's

(4) Handling white space

(5) How many rows to skip, starting from the top

(6) Number of max rows

Before we move on to exporting, I want to hop down to the bottom and just note that there are some additional examples. We have an outline of how to load in Excel sheets. Looks pretty similar to the read_csv, but it's for Excel and it has some additional arguments. You can specify the sheet and the range. You can control column names and type. Similar how to handle NAs, how to handle a whitespace, how many rows to skip, the maximum number of rows. It looks pretty similar to read_csv, but it's for Excel. Then haven, here's just an example for SAS. But it offers options for encoding, specifying columns to select if you don't want everything, and then skipping certain rows and the maximum number of rows as well.

We're not going to go through examples of those. I think if we ever use an Excel file in this course, we will provide an example. But I recommend if you're needing to load something in to go through the exercise like we just did with the CSV. Start using the path and don't specify anything else, and go through and just figure out what you need. Of course, if you know you need to use another argument, go ahead and do it. But often it helps to just do it with just the path and then figure out what else do you need to get the data in and the way that you want to use it.

1.4 Exporting

Exporting data is much simpler than importing data. You will want to consider the following before you export:

- Who will use this data?
- Will more analysis need completing or is the intended use for review in excel?

```
# export csv - good if needing flexibility for other analytic programs
readr::write_csv(esd_clean, "esd_clean.csv")

# export to excel - good if others want to look at the data in excel
openxlsx::write.xlsx(esd_clean, "esd_clean.xlsx")

# export RDS - good if you will want to pick up the data back in R
saveRDS(esd_clean, "esd_clean_saved.RDS")
```

In these examples the files were saved to the current working directory. However, you can specify an output location by providing a path. For example if you wanted to place the file one directory backwards, you could write: `../esd_clean.csv`.

The last piece here is exporting data. It is a lot simpler than importing data. There's less customization that needs to be done, but you will want to really consider who is going to be using this data. Will more analysis be needed, or is the intended use for this review and Excel maybe by a non-programmer or someone else? Here are some options, and we're going to go through exporting the esd_clean that we just created. You can use write_csv to create a CSV, and this is helpful or good option if you want flexibility for other analytic programs or to use in R again later.

You can export to Excel. This would be good if you want others to look at the data in Excel. Maybe they don't have access to the same programs or aren't interested in analyzing the data. Then you can export to RDS. This is good if you want to pick the data back up in R. Since this is an R data structure, it won't really be usable in other programs, but it can be used for R. I'm just going to go through and run these really quick. You can actually see that they get written to wherever my current working directory is, which I'm working in a development branch for this week's content. You can see the esd_clean shows up there and same thing if you use Excel. I don't have the openxlsx package apparently. Let's see if I can install that.

This is a good learning moment. Sometimes we might use a package in here that you don't have in your environment. If that happens, you can do what I just did and go to install and type in the package. Typically, it will show up and you can install it. Now if you go back to your viewer here, you can see that openxlsx actually shows up in my list. I don't need to load the library because it's actually called here. Let's try running this again. If you go back to my files, you can see the x. This is actually written in the wrong order, so let's try that one more time and we'll get the right file abbreviation there. Then you can also save as an RDS as well. These are then available in your files on R.

1.4 Exporting

Exporting data is much simpler than importing data. You will want to consider the following before you export:

- Who will use this data?
- Will more analysis need completing or is the intended use for review in excel?

```
# export csv - good if needing flexibility for other analytic programs
readr::write_csv(esd_clean, "esd_clean.csv")

# export to excel - good if others want to look at the data in excel
openxlsx::write.xlsx(esd_clean, "esd_clean.xlsx")

# export RDS - good if you will want to pick up the data back in R
saveRDS(esd_clean, "esd_clean_saved.RDS")
```

In these examples the files were saved to the current working directory. However, you can specify an output location by providing a path. For example if you wanted to place the file one directory backwards, you could write: `../esd_clean.csv`.

If you then wanted to export for someone else to use, you have to go through this process of checking it and then going to this little extra settings. Then you can do export and go through the process, it will download it to your computer. You can reopen these in your session. You can do view file and you'll see the CSV shows up like this. If I open the RDS, it will load it, and it actually loads it as esd_clean_saved. You can see why maybe the RDS, if you know you're coming back to R, you might want to just save whatever file as an RDS so you can really easily load it again. If it's a CSV, you'll have to go through the process of reading in the CSV again.

You can specify output location by providing a path. We didn't really provide a path, so just went to whatever our working directory is. You could place the file one directory backwards by doing a dash with two dots before it. If I go up to my weekly material folder, I'll see that esd_clean_saved. There's ways to navigate around your R folders within these tabs.

8 Data import

8.1 Introduction

Working with data provided by R packages is a great way to learn data science tools, but you want to apply what you've learned to your own data at some point. In this chapter, you'll learn the basics of reading data files into R.

Specifically, this chapter will focus on reading plain-text rectangular files. We'll start with practical advice for handling features like column names, types, and missing data. You will then learn about reading data from multiple files at once and writing data from R to a file. Finally, you'll learn how to handcraft data frames in R.

8.1.1 Prerequisites

In this chapter, you'll learn how to load flat files in R with the **readr** package, which is part of the core tidyverse.

```
library(tidyverse)
```

8.2 Reading data from a file

To begin, we'll focus on the most common rectangular data file type: CSV, which is short for comma-separated values. Here is what a simple CSV file looks like. The first row, commonly called the header row, gives the column names, and the following six rows provide the data. The columns are separated, aka delimited, by commas.

```
Student ID,Full Name,favourite.food,mealPlan,AGE
1,Sunil Huffmann,Strawberry yoghurt,Lunch only,4
2,Barclay Lynn,French fries,Lunch only,5
3,Jayendra Lyne,N/A,Breakfast and lunch,7
4,Leon Rossini,Anchovies,Lunch only,
5,Chidiegwu Dunkel,Pizza,Breakfast and lunch,five
6,Güvenç Attila,Ice cream,Lunch only,6
```

Table 8.1 shows a representation of the same data as a table.

Table 8.1: Data from the students.csv file as a table.

Student ID	Full Name	favourite.food	mealPlan	AGE
1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4
2	Barclay Lynn	French fries	Lunch only	5
3	Jayendra Lyne	N/A	Breakfast and lunch	7
4	Leon Rossini	Anchovies	Lunch only	NA
5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five
6	Güvenç Attila	Ice cream	Lunch only	6

We can read this file into R using `read_csv()`. The first argument is the most important: the path to the file. You can think about the path as the address of the file: the file is called `students.csv` and that it lives in the `data` folder.

```
students <- read_csv("data/students.csv")
#> Rows: 6 Columns: 5
#> ── Column specification ────────────────────────────────────────────────────
#> Delimiter: ","
#> chr (4): Full Name, favourite.food, mealPlan, AGE
#> dbl (1): Student ID
#>
#> ℹ Use `spec()` to retrieve the full column specification for this data.
#> ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The code above will work if you have the `students.csv` file in a `data` folder in your project. You can download the `students.csv` file from https://pos.it/r4ds-students-csv or you can read it directly from that URL with:

```
students <- read_csv("https://pos.it/r4ds-students-csv")
```

When you run `read_csv()`, it prints out a message telling you the number of rows and columns of data, the delimiter that was used, and the column specifications (names of columns organized by the type of data the column contains). It also prints out some information about retrieving the full column specification and how to quiet this message. This message is an integral part of readr, and we'll return to it in Section 8.3.

8.2.1 Practical advice

Once you read data in, the first step usually involves transforming it in some way to make it easier to work with in the rest of your analysis. Let's take another look at the `students` data with that in mind.

```
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2            2 Barclay Lynn     French fries       Lunch only          5
#> 3            3 Jayendra Lyne    N/A                Breakfast and lunch 7
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6            6 Güvenç Attila    Ice cream          Lunch only          6
```

In the `favourite.food` column, there are a bunch of food items, and then the character string `N/A`, which should have been a real `NA` that R will recognize as "not available". This is something we can address using the `na` argument. By default, `read_csv()` only recognizes empty strings (`""`) in this dataset as `NA`s, we want it to also recognize the character string `"N/A"`.

```
students <- read_csv("data/students.csv", na = c("N/A", ""))
```

```
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2            2 Barclay Lynn     French fries       Lunch only          5
#> 3            3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6            6 Güvenç Attila    Ice cream          Lunch only          6
```

You might also notice that the `Student ID` and `Full Name` columns are surrounded by backticks. That's because they contain spaces, breaking R's usual rules for variable names; they're **non-syntactic** names. To refer to these variables, you need to surround them with backticks, `` ` ``:

```
students |>
  rename(
    student_id = `Student ID`,
    full_name = `Full Name`
  )
#> # A tibble: 6 × 5
#>   student_id full_name        favourite.food     mealPlan            AGE
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

An alternative approach is to use `janitor::clean_names()` to use some heuristics to turn them all into snake case at once[1].

```
students |> janitor::clean_names()
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Another common task after reading in data is to consider variable types. For example, `meal_plan` is a categorical variable with a known set of possible values, which in R should be represented as a factor:

```
students |>
  janitor::clean_names() |>
```

```
  mutate(meal_plan = factor(meal_plan))
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <fct>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Note that the values in the `meal_plan` variable have stayed the same, but the type of variable denoted underneath the variable name has changed from character (`<chr>`) to factor (`<fct>`). You'll learn more about factors in Chapter 17.

Before you analyze these data, you'll probably want to fix the `age` and `id` columns. Currently, `age` is a character variable because one of the observations is typed out as `five` instead of a numeric `5`. We discuss the details of fixing this issue in Chapter 21.

```
students <- students |>
  janitor::clean_names() |>
  mutate(
    meal_plan = factor(meal_plan),
    age = parse_number(if_else(age == "five", "5", age))
  )

students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

A new function here is `if_else()`, which has three arguments. The first argument `test` should be a logical vector. The result will contain the value of the second argument, `yes`, when `test` is `TRUE`, and the value of the third argument, `no`, when it is `FALSE`. Here we're saying if `age` is the character string `"five"`, make it `"5"`, and if not leave it as `age`. You will learn more about `if_else()` and logical vectors in Chapter 13.

8.2.2 Other arguments

There are a couple of other important arguments that we need to mention, and they'll be easier to demonstrate if we first show you a handy trick: `read_csv()` can read text strings that you've created and formatted like a CSV file:

```
read_csv(
  "a,b,c
  1,2,3
  4,5,6"
)
#> # A tibble: 2 × 3
#>       a     b     c
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

Usually, `read_csv()` uses the first line of the data for the column names, which is a very common convention. But it's not uncommon for a few lines of metadata to be included at the top of the file. You can use `skip = n` to skip the first `n` lines or use `comment = "#"` to drop all lines that start with (e.g.) `#`:

```
read_csv(
  "The first line of metadata
  The second line of metadata
  x,y,z
  1,2,3",
  skip = 2
)
#> # A tibble: 1 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3

read_csv(
  "# A comment I want to skip
  x,y,z
  1,2,3",
  comment = "#"
)
#> # A tibble: 1 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
```

In other cases, the data might not have column names. You can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings and instead label them sequentially from `X1` to `Xn`:

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = FALSE
)
#> # A tibble: 2 × 3
#>      X1    X2    X3
```

```
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

Alternatively, you can pass `col_names` a character vector which will be used as the column names:

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = c("x", "y", "z")
)
#> # A tibble: 2 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

These arguments are all you need to know to read the majority of CSV files that you'll encounter in practice. (For the rest, you'll need to carefully inspect your `.csv` file and read the documentation for `read_csv()`'s many other arguments.)

8.2.3 Other file types

Once you've mastered `read_csv()`, using readr's other functions is straightforward; it's just a matter of knowing which function to reach for:

- `read_csv2()` reads semicolon-separated files. These use `;` instead of `,` to separate fields and are common in countries that use `,` as the decimal marker.
- `read_tsv()` reads tab-delimited files.
- `read_delim()` reads in files with any delimiter, attempting to automatically guess the delimiter if you don't specify it.
- `read_fwf()` reads fixed-width files. You can specify fields by their widths with `fwf_widths()` or by their positions with `fwf_positions()`.
- `read_table()` reads a common variation of fixed-width files where columns are separated by white space.
- `read_log()` reads Apache-style log files.

8.2.4 Exercises

1. What function would you use to read a file where fields were separated with "|"?
2. Apart from `file`, `skip`, and `comment`, what other arguments do `read_csv()` and `read_tsv()` have in common?
3. What are the most important arguments to `read_fwf()`?

4. Sometimes strings in a CSV file contain commas. To prevent them from causing problems, they need to be surrounded by a quoting character, like `"` or `'`. By default, `read_csv()` assumes that the quoting character will be `"`. To read the following text into a data frame, what argument to `read_csv()` do you need to specify?

```
"x,y\n1,'a,b'"
```

5. Identify what is wrong with each of the following inline CSV files. What happens when you run the code?

```
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a,b\n\"1")
read_csv("a,b\n1,2\na,b")
read_csv("a;b\n1;3")
```

6. Practice referring to non-syntactic names in the following data frame by:

   a. Extracting the variable called `1`.
   b. Plotting a scatterplot of `1` vs. `2`.
   c. Creating a new column called `3`, which is `2` divided by `1`.
   d. Renaming the columns to `one`, `two`, and `three`.

```
annoying <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

8.3 Controlling column types

A CSV file doesn't contain any information about the type of each variable (i.e. whether it's a logical, number, string, etc.), so readr will try to guess the type. This section describes how the guessing process works, how to resolve some common problems that cause it to fail, and, if needed, how to supply the column types yourself. Finally, we'll mention a few general strategies that are useful if readr is failing catastrophically and you need to get more insight into the structure of your file.

8.3.1 Guessing types

readr uses a heuristic to figure out the column types. For each column, it pulls the values of 1,000[2] rows spaced evenly from the first row to the last, ignoring missing values. It then works through the following questions:

- Does it contain only `F`, `T`, `FALSE`, or `TRUE` (ignoring case)? If so, it's a logical.
- Does it contain only numbers (e.g., `1`, `-4.5`, `5e6`, `Inf`)? If so, it's a number.
- Does it match the ISO8601 standard? If so, it's a date or date-time. (We'll return to date-times in more detail in Section 18.2).
- Otherwise, it must be a string.

You can see that behavior in action in this simple example:

```
read_csv("
  logical,numeric,date,string
  TRUE,1,2021-01-15,abc
  false,4.5,2021-02-15,def
  T,Inf,2021-02-16,ghi
")
#> # A tibble: 3 × 4
#>   logical numeric date       string
#>   <lgl>     <dbl> <date>     <chr>
#> 1 TRUE        1   2021-01-15 abc
#> 2 FALSE       4.5 2021-02-15 def
#> 3 TRUE      Inf   2021-02-16 ghi
```

This heuristic works well if you have a clean dataset, but in real life, you'll encounter a selection of weird and beautiful failures.

8.3.2 Missing values, column types, and problems

The most common way column detection fails is that a column contains unexpected values, and you get a character column instead of a more specific type. One of the most common causes for this is a missing value, recorded using something other than the `NA` that readr expects.

Take this simple 1 column CSV file as an example:

```
simple_csv <- "
  x
  10
  .
  20
  30"
```

If we read it without any additional arguments, `x` becomes a character column:

```
read_csv(simple_csv)
#> # A tibble: 4 × 1
#>   x
#>   <chr>
#> 1 10
#> 2 .
#> 3 20
#> 4 30
```

In this very small case, you can easily see the missing value `.`. But what happens if you have thousands of rows with only a few missing values represented by `.`s sprinkled among them? One approach is to tell readr that `x` is a numeric column, and then see where it fails. You can do that with the `col_types` argument, which takes a named

ere the names match the column names in the CSV file:

```
df <- read_csv(
  simple_csv,
  col_types = list(x = col_double())
)
#> Warning: One or more parsing issues, call `problems()` on your data frame for
#> details, e.g.:
#>   dat <- vroom(...)
#>   problems(dat)
```

Now `read_csv()` reports that there was a problem, and tells us we can find out more with `problems()`:

```
problems(df)
#> # A tibble: 1 × 5
#>     row   col expected actual file
#>   <int> <int> <chr>    <chr>  <chr>
#> 1     3     1 a double .      /tmp/RtmpgnvG6b/file1b0fbc96b45
```

This tells us that there was a problem in row 3, col 1 where readr expected a double but got a `.`. That suggests this dataset uses `.` for missing values. So then we set `na = "."`, the automatic guessing succeeds, giving us the numeric column that we want:

```
read_csv(simple_csv, na = ".")
#> # A tibble: 4 × 1
#>       x
#>   <dbl>
#> 1    10
#> 2    NA
#> 3    20
#> 4    30
```

8.3.3 Column types

readr provides a total of nine column types for you to use:

- `col_logical()` and `col_double()` read logicals and real numbers. They're relatively rarely needed (except as above), since readr will usually guess them for you.
- `col_integer()` reads integers. We seldom distinguish integers and doubles in this book because they're functionally equivalent, but reading integers explicitly can occasionally be useful because they occupy half the memory of doubles.
- `col_character()` reads strings. This can be useful to specify explicitly when you have a column that is a numeric identifier, i.e., long series of digits that identifies an object but doesn't make sense to apply mathematical operations to. Examples include phone numbers, social security numbers, credit card numbers, etc.
- `col_factor()`, `col_date()`, and `col_datetime()` create factors, dates, and date-times respectively; you'll learn more about those when we get to those data types in Chapter 17 and Chapter 18.

- `col_number()` is a permissive numeric parser that will ignore non-numeric components, and is particularly useful for currencies. You'll learn more about it in Chapter 14.
- `col_skip()` skips a column so it's not included in the result, which can be useful for speeding up reading the data if you have a large CSV file and you only want to use some of the columns.

It's also possible to override the default column by switching from `list()` to `cols()` and specifying `.default`:

```
another_csv <- "
x,y,z
1,2,3"

read_csv(
  another_csv,
  col_types = cols(.default = col_character())
)
#> # A tibble: 1 × 3
#>   x     y     z
#>   <chr> <chr> <chr>
#> 1 1     2     3
```

Another useful helper is `cols_only()` which will read in only the columns you specify:

```
read_csv(
  another_csv,
  col_types = cols_only(x = col_character())
)
#> # A tibble: 1 × 1
#>   x
#>   <chr>
#> 1 1
```

8.4 Reading data from multiple files

Sometimes your data is split across multiple files instead of being contained in a single file. For example, you might have sales data for multiple months, with each month's data in a separate file: `01-sales.csv` for January, `02-sales.csv` for February, and `03-sales.csv` for March. With `read_csv()` you can read these data in at once and stack them on top of each other in a single data frame.

```
sales_files <- c("data/01-sales.csv", "data/02-sales.csv", "data/03-sales.csv")
read_csv(sales_files, id = "file")
#> # A tibble: 19 × 6
#>   file              month    year brand  item     n
#>   <chr>             <chr>   <dbl> <dbl> <dbl> <dbl>
#> 1 data/01-sales.csv January  2019     1  1234     3
#> 2 data/01-sales.csv January  2019     1  8721     9
#> 3 data/01-sales.csv January  2019     1  1822     2
#> 4 data/01-sales.csv January  2019     2  3333     1
```

```
#> 5 data/01-sales.csv January  2019     2  2156     9
#> 6 data/01-sales.csv January  2019     2  3987     6
#> # ℹ 13 more rows
```

Once again, the code above will work if you have the CSV files in a `data` folder in your project. You can download these files from https://pos.it/r4ds-01-sales, https://pos.it/r4ds-02-sales, and https://pos.it/r4ds-03-sales or you can read them directly with:

```
sales_files <- c(
  "https://pos.it/r4ds-01-sales",
  "https://pos.it/r4ds-02-sales",
  "https://pos.it/r4ds-03-sales"
)
read_csv(sales_files, id = "file")
```

The `id` argument adds a new column called `file` to the resulting data frame that identifies the file the data come from. This is especially helpful in circumstances where the files you're reading in do not have an identifying column that can help you trace the observations back to their original sources.

If you have many files you want to read in, it can get cumbersome to write out their names as a list. Instead, you can use the base `list.files()` function to find the files for you by matching a pattern in the file names. You'll learn more about these patterns in Chapter 16.

```
sales_files <- list.files("data", pattern = "sales\\.csv$", full.names = TRUE)
sales_files
#> [1] "data/01-sales.csv" "data/02-sales.csv" "data/03-sales.csv"
```

8.5 Writing to a file

readr also comes with two useful functions for writing data back to disk: `write_csv()` and `write_tsv()`. The most important arguments to these functions are `x` (the data frame to save) and `file` (the location to save it). You can also specify how missing values are written with `na`, and if you want to `append` to an existing file.

```
write_csv(students, "students.csv")
```

Now let's read that csv file back in. Note that the variable type information that you just set up is lost when you save to CSV because you're starting over with reading from a plain text file again:

```
students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
```

```
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
write_csv(students, "students-2.csv")
read_csv("students-2.csv")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <chr>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

This makes CSVs a little unreliable for caching interim results—you need to recreate the column specification every time you load in. There are two main alternatives:

1. `write_rds()` and `read_rds()` are uniform wrappers around the base functions `readRDS()` and `saveRDS()`. These store data in R's custom binary format called RDS. This means that when you reload the object, you are loading the *exact same* R object that you stored.

   ```
   write_rds(students, "students.rds")
   read_rds("students.rds")
   #> # A tibble: 6 × 5
   #>   student_id full_name        favourite_food     meal_plan             age
   #>        <dbl> <chr>            <chr>              <fct>               <dbl>
   #> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
   #> 2          2 Barclay Lynn     French fries       Lunch only              5
   #> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
   #> 4          4 Leon Rossini     Anchovies          Lunch only             NA
   #> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
   #> 6          6 Güvenç Attila    Ice cream          Lunch only              6
   ```

2. The arrow package allows you to read and write parquet files, a fast binary file format that can be shared across programming languages. We'll return to arrow in more depth in Chapter 23.

   ```
   library(arrow)
   write_parquet(students, "students.parquet")
   read_parquet("students.parquet")
   #> # A tibble: 6 × 5
   #>   student_id full_name        favourite_food     meal_plan             age
   #>        <dbl> <chr>            <chr>              <fct>               <dbl>
   #> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
   #> 2          2 Barclay Lynn     French fries       Lunch only              5
   #> 3          3 Jayendra Lyne    NA                 Breakfast and lunch     7
   #> 4          4 Leon Rossini     Anchovies          Lunch only             NA
   #> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
   #> 6          6 Güvenç Attila    Ice cream          Lunch only              6
   ```

Parquet tends to be much faster than RDS and is usable outside of R, but does require the arrow package.

8.6 Data entry

Sometimes you'll need to assemble a tibble "by hand" doing a little data entry in your R script. There are two useful functions to help you do this which differ in whether you layout the tibble by columns or by rows. `tibble()` works by column:

```
tibble(
  x = c(1, 2, 5),
  y = c("h", "m", "g"),
  z = c(0.08, 0.83, 0.60)
)
#> # A tibble: 3 × 3
#>       x y         z
#>   <dbl> <chr> <dbl>
#> 1     1 h      0.08
#> 2     2 m      0.83
#> 3     5 g      0.6
```

Laying out the data by column can make it hard to see how the rows are related, so an alternative is `tribble()`, short for **tr**ansposed t**ibble**, which lets you lay out your data row by row. `tribble()` is customized for data entry in code: column headings start with `~` and entries are separated by commas. This makes it possible to lay out small amounts of data in an easy to read form:

```
tribble(
  ~x, ~y, ~z,
  1, "h", 0.08,
  2, "m", 0.83,
  5, "g", 0.60
)
#> # A tibble: 3 × 3
#>       x y         z
#>   <dbl> <chr> <dbl>
#> 1     1 h      0.08
#> 2     2 m      0.83
#> 3     5 g      0.6
```

8.7 Summary

In this chapter, you've learned how to load CSV files with `read_csv()` and to do your own data entry with `tibble()` and `tribble()`. You've learned how csv files work, some of the problems you might encounter, and how to overcome them. We'll come to data import a few times in this book: Chapter 21 from Excel and Google Sheets, Chapter 22 will show you how to load data from databases, Chapter 23 from parquet files, Chapter 24 from JSON, and Chapter 25 from websites.

We're just about at the end of this section of the book, but there's one important last topic to cover: how to get help. So in the next chapter, you'll learn some good places to look for help, how to create a reprex to maximize your chances of getting good help, and some general advice on keeping up with the world of R.

---

1. The janitor package is not part of the tidyverse, but it offers handy functions for data cleaning and works well within data pipelines that use `|>`.↩

2. You can override the default of 1000 with the `guess_max` argument.↩

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

This book was built with Quarto.

21 Spreadsheets

21.1 Introduction

In Chapter 8 you learned about importing data from plain text files like `.csv` and `.tsv`. Now it's time to learn how to get data out of a spreadsheet, either an Excel spreadsheet or a Google Sheet. This will build on much of what you've learned in Chapter 8, but we will also discuss additional considerations and complexities when working with data from spreadsheets.

If you or your collaborators are using spreadsheets for organizing data, we strongly recommend reading the paper "Data Organization in Spreadsheets" by Karl Broman and Kara Woo: https://doi.org/10.1080/00031305.2017.1375989. The best practices presented in this paper will save you much headache when you import data from a spreadsheet into R to analyze and visualize.

21.2 Excel

Microsoft Excel is a widely used spreadsheet software program where data are organized in worksheets inside of spreadsheet files.

21.2.1 Prerequisites

In this section, you'll learn how to load data from Excel spreadsheets in R with the **readxl** package. This package is non-core tidyverse, so you need to load it explicitly, but it is installed automatically when you install the tidyverse package. Later, we'll also use the writexl package, which allows us to create Excel spreadsheets.

```
library(readxl)
library(tidyverse)
library(writexl)
```

21.2.2 Getting started

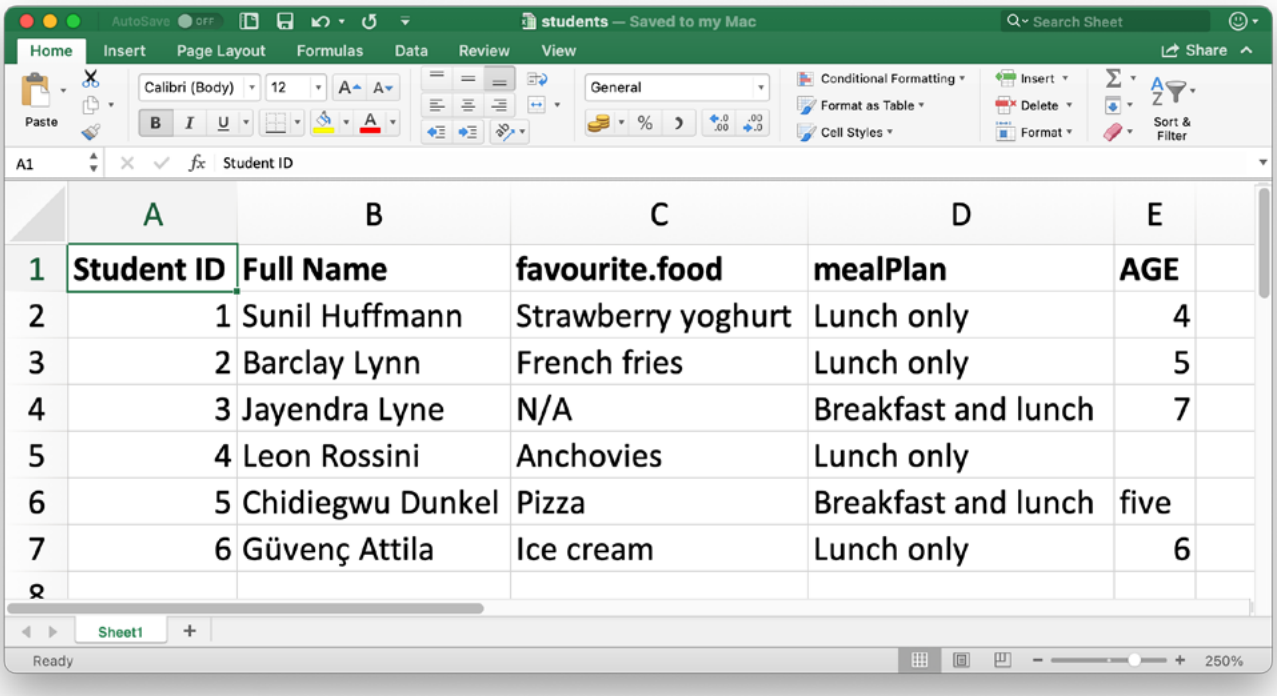
Most of readxl's functions allow you to load Excel spreadsheets into R:

- `read_xls()` reads Excel files with `xls` format.
- `read_xlsx()` read Excel files with `xlsx` format.
- `read_excel()` can read files with both `xls` and `xlsx` format. It guesses the file type based on the input.

These functions all have similar syntax just like other functions we have previously introduced for reading other types of files, e.g., `read_csv()`, `read_table()`, etc. For the rest of the chapter we will focus on using `read_excel()`.

21.2.3 Reading Excel spreadsheets

Figure 21.1 shows what the spreadsheet we're going to read into R looks like in Excel.



	A	B	C	D	E
1	Student ID	Full Name	favourite.food	mealPlan	AGE
2	1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4
3	2	Barclay Lynn	French fries	Lunch only	5
4	3	Jayendra Lyne	N/A	Breakfast and lunch	7
5	4	Leon Rossini	Anchovies	Lunch only	
6	5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five
7	6	Güvenç Attila	Ice cream	Lunch only	6
8					

Figure 21.1: Spreadsheet called students.xlsx in Excel.

The first argument to `read_excel()` is the path to the file to read.

```
students <- read_excel("data/students.xlsx")
```

`read_excel()` will read the file in as a tibble.

```
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2            2 Barclay Lynn     French fries       Lunch only          5
#> 3            3 Jayendra Lyne    N/A                Breakfast and lunch 7
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6            6 Güvenç Attila    Ice cream          Lunch only          6
```

We have six students in the data and five variables on each student. However there are a few things we might want to address in this dataset:

1. The column names are all over the place. You can provide column names that follow a consistent format; we recommend `snake_case` using the `col_names` argument.

```
read_excel(
  "data/students.xlsx",
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age")
)
#> # A tibble: 7 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>   <chr>      <chr>            <chr>              <chr>               <chr>
#> 1 Student ID Full Name        favourite.food     mealPlan            AGE
#> 2 1          Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 3 2          Barclay Lynn     French fries       Lunch only          5
#> 4 3          Jayendra Lyne    N/A                Breakfast and lunch 7
#> 5 4          Leon Rossini     Anchovies          Lunch only          <NA>
#> 6 5          Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 7 6          Güvenç Attila    Ice cream          Lunch only          6
```

Unfortunately, this didn't quite do the trick. We now have the variable names we want, but what was previously the header row now shows up as the first observation in the data. You can explicitly skip that row using the `skip` argument.

```
read_excel(
  "data/students.xlsx",
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1
)
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    N/A                Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

2. In the `favourite_food` column, one of the observations is `N/A`, which stands for "not available" but it's currently not recognized as an `NA` (note the contrast between this `N/A` and the age of the fourth student in the list). You can specify which character strings should be recognized as `NA`s with the `na` argument. By default, only `""` (empty string, or, in the case of reading from a spreadsheet, an empty cell or a cell with the formula `=NA()`) is recognized as an `NA`.

```
read_excel(
  "data/students.xlsx",
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1,
  na = c("", "N/A")
)
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
```

```
#>        <dbl> <chr>           <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn   French fries       Lunch only          5
#> 3          3 Jayendra Lyne  <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini   Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza            Breakfast and lunch five
#> 6          6 Güvenç Attila  Ice cream          Lunch only          6
```

3. One other remaining issue is that `age` is read in as a character variable, but it really should be numeric. Just like with `read_csv()` and friends for reading data from flat files, you can supply a `col_types` argument to `read_excel()` and specify the column types for the variables you read in. The syntax is a bit different, though. Your options are `"skip"`, `"guess"`, `"logical"`, `"numeric"`, `"date"`, `"text"` or `"list"`.

```
read_excel(
  "data/students.xlsx",
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1,
  na = c("", "N/A"),
  col_types = c("numeric", "text", "text", "text", "numeric")
)
#> Warning: Expecting numeric in E6 / R6C5: got 'five'
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <chr>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch    NA
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

However, this didn't quite produce the desired result either. By specifying that `age` should be numeric, we have turned the one cell with the non-numeric entry (which had the value `five`) into an `NA`. In this case, we should read age in as `"text"` and then make the change once the data is loaded in R.

```
students <- read_excel(
  "data/students.xlsx",
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1,
  na = c("", "N/A"),
  col_types = c("numeric", "text", "text", "text", "text")
)

students <- students |>
  mutate(
    age = if_else(age == "five", "5", age),
    age = parse_number(age)
  )
```

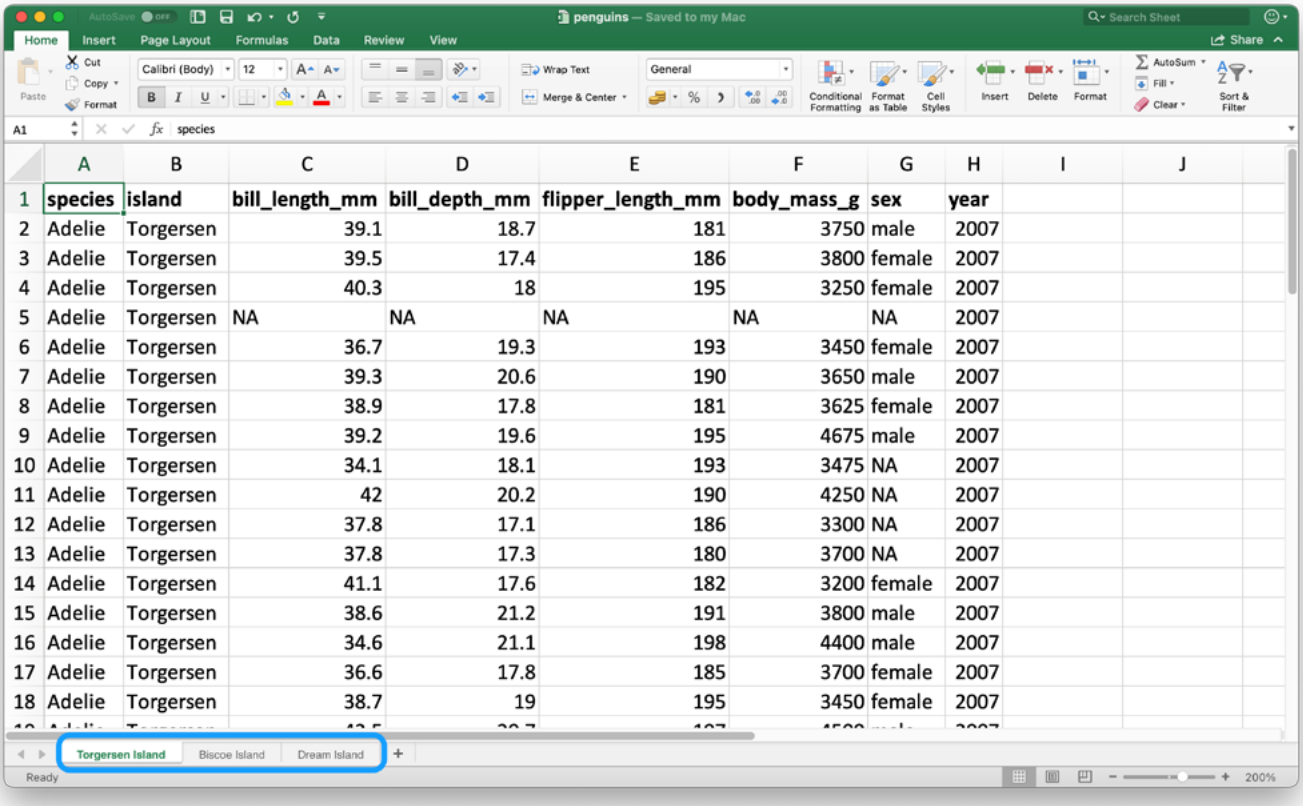
```
students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <chr>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

It took us multiple steps and trial-and-error to load the data in exactly the format we want, and this is not unexpected. Data science is an iterative process, and the process of iteration can be even more tedious when reading data in from spreadsheets compared to other plain text, rectangular data files because humans tend to input data into spreadsheets and use them not just for data storage but also for sharing and communication.

There is no way to know exactly what the data will look like until you load it and take a look at it. Well, there is one way, actually. You can open the file in Excel and take a peek. If you're going to do so, we recommend making a copy of the Excel file to open and browse interactively while leaving the original data file untouched and reading into R from the untouched file. This will ensure you don't accidentally overwrite anything in the spreadsheet while inspecting it. You should also not be afraid of doing what we did here: load the data, take a peek, make adjustments to your code, load it again, and repeat until you're happy with the result.

21.2.4 Reading worksheets

An important feature that distinguishes spreadsheets from flat files is the notion of multiple sheets, called worksheets. Figure 21.2 shows an Excel spreadsheet with multiple worksheets. The data come from the **palmerpenguins** package. Each worksheet contains information on penguins from a different island where data were collected.



	A	B	C	D	E	F	G	H
1	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
2	Adelie	Torgersen	39.1	18.7	181	3750	male	2007
3	Adelie	Torgersen	39.5	17.4	186	3800	female	2007
4	Adelie	Torgersen	40.3	18	195	3250	female	2007
5	Adelie	Torgersen	NA	NA	NA	NA	NA	2007
6	Adelie	Torgersen	36.7	19.3	193	3450	female	2007
7	Adelie	Torgersen	39.3	20.6	190	3650	male	2007
8	Adelie	Torgersen	38.9	17.8	181	3625	female	2007
9	Adelie	Torgersen	39.2	19.6	195	4675	male	2007
10	Adelie	Torgersen	34.1	18.1	193	3475	NA	2007
11	Adelie	Torgersen	42	20.2	190	4250	NA	2007
12	Adelie	Torgersen	37.8	17.1	186	3300	NA	2007
13	Adelie	Torgersen	37.8	17.3	180	3700	NA	2007
14	Adelie	Torgersen	41.1	17.6	182	3200	female	2007
15	Adelie	Torgersen	38.6	21.2	191	3800	male	2007
16	Adelie	Torgersen	34.6	21.1	198	4400	male	2007
17	Adelie	Torgersen	36.6	17.8	185	3700	female	2007
18	Adelie	Torgersen	38.7	19	195	3450	female	2007

Figure 21.2: Spreadsheet called penguins.xlsx in Excel containing three worksheets.

You can read a single worksheet from a spreadsheet with the `sheet` argument in `read_excel()`. The default, which we've been relying on up until now, is the first sheet.

```
read_excel("data/penguins.xlsx", sheet = "Torgersen Island")
#> # A tibble: 52 × 8
#>   species island    bill_length_mm     bill_depth_mm      flipper_length_mm
#>   <chr>   <chr>     <chr>              <chr>              <chr>
#> 1 Adelie  Torgersen 39.1               18.7               181
#> 2 Adelie  Torgersen 39.5               17.399999999999999 186
#> 3 Adelie  Torgersen 40.299999999999997 18                 195
#> 4 Adelie  Torgersen NA                 NA                 NA
#> 5 Adelie  Torgersen 36.700000000000003 19.3               193
#> 6 Adelie  Torgersen 39.299999999999997 20.6               190
#> # ℹ 46 more rows
#> # ℹ 3 more variables: body_mass_g <chr>, sex <chr>, year <dbl>
```

Some variables that appear to contain numerical data are read in as characters due to the character string `"NA"` not being recognized as a true `NA`.

```
penguins_torgersen <- read_excel("data/penguins.xlsx", sheet = "Torgersen Island", na = "

penguins_torgersen
#> # A tibble: 52 × 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm
```

```
#>   <chr>   <chr>              <dbl>         <dbl>             <dbl>
#> 1 Adelie  Torgersen           39.1          18.7               181
#> 2 Adelie  Torgersen           39.5          17.4               186
#> 3 Adelie  Torgersen           40.3          18                 195
#> 4 Adelie  Torgersen           NA            NA                  NA
#> 5 Adelie  Torgersen           36.7          19.3               193
#> 6 Adelie  Torgersen           39.3          20.6               190
#> # ℹ 46 more rows
#> # ℹ 3 more variables: body_mass_g <dbl>, sex <chr>, year <dbl>
```

Alternatively, you can use `excel_sheets()` to get information on all worksheets in an Excel spreadsheet, and then read the one(s) you're interested in.

```
excel_sheets("data/penguins.xlsx")
#> [1] "Torgersen Island" "Biscoe Island"    "Dream Island"
```

Once you know the names of the worksheets, you can read them in individually with `read_excel()`.

```
penguins_biscoe <- read_excel("data/penguins.xlsx", sheet = "Biscoe Island", na = "NA")
penguins_dream  <- read_excel("data/penguins.xlsx", sheet = "Dream Island", na = "NA")
```

In this case the full penguins dataset is spread across three worksheets in the spreadsheet. Each worksheet has the same number of columns but different numbers of rows.

```
dim(penguins_torgersen)
#> [1] 52  8
dim(penguins_biscoe)
#> [1] 168   8
dim(penguins_dream)
#> [1] 124   8
```

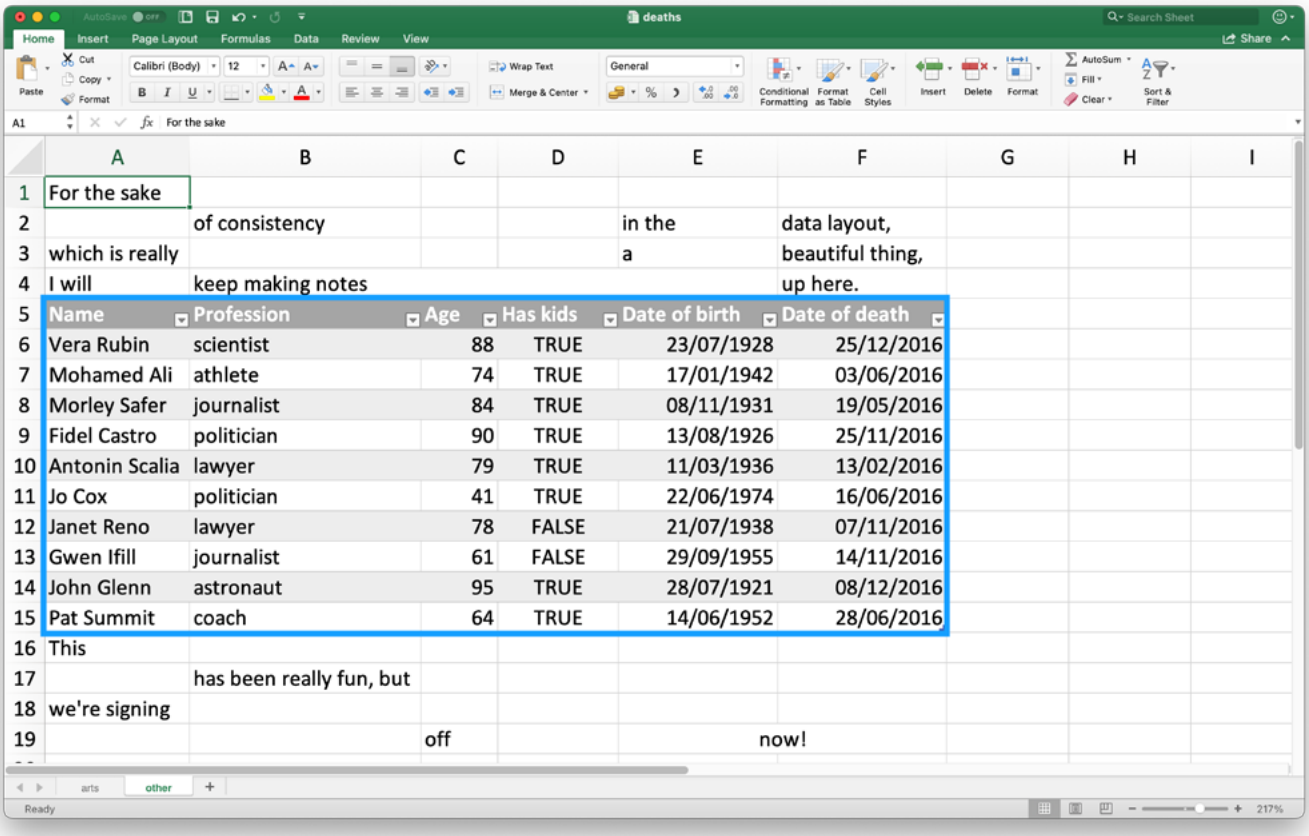
We can put them together with `bind_rows()`.

```
penguins <- bind_rows(penguins_torgersen, penguins_biscoe, penguins_dream)
penguins
#> # A tibble: 344 × 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm
#>   <chr>   <chr>              <dbl>         <dbl>             <dbl>
#> 1 Adelie  Torgersen           39.1          18.7               181
#> 2 Adelie  Torgersen           39.5          17.4               186
#> 3 Adelie  Torgersen           40.3          18                 195
#> 4 Adelie  Torgersen           NA            NA                  NA
#> 5 Adelie  Torgersen           36.7          19.3               193
#> 6 Adelie  Torgersen           39.3          20.6               190
#> # ℹ 338 more rows
#> # ℹ 3 more variables: body_mass_g <dbl>, sex <chr>, year <dbl>
```

 pter 27 we'll talk about ways of doing this sort of task without repetitive code.

21.2.5 Reading part of a sheet

Since many use Excel spreadsheets for presentation as well as for data storage, it's quite common to find cell entries in a spreadsheet that are not part of the data you want to read into R. Figure 21.3 shows such a spreadsheet: in the middle of the sheet is what looks like a data frame but there is extraneous text in cells above and below the data.



	A	B	C	D	E	F	G	H	I
1	For the sake								
2		of consistency			in the	data layout,			
3	which is really				a	beautiful thing,			
4	I will	keep making notes				up here.			
5	Name	Profession	Age	Has kids	Date of birth	Date of death			
6	Vera Rubin	scientist	88	TRUE	23/07/1928	25/12/2016			
7	Mohamed Ali	athlete	74	TRUE	17/01/1942	03/06/2016			
8	Morley Safer	journalist	84	TRUE	08/11/1931	19/05/2016			
9	Fidel Castro	politician	90	TRUE	13/08/1926	25/11/2016			
10	Antonin Scalia	lawyer	79	TRUE	11/03/1936	13/02/2016			
11	Jo Cox	politician	41	TRUE	22/06/1974	16/06/2016			
12	Janet Reno	lawyer	78	FALSE	21/07/1938	07/11/2016			
13	Gwen Ifill	journalist	61	FALSE	29/09/1955	14/11/2016			
14	John Glenn	astronaut	95	TRUE	28/07/1921	08/12/2016			
15	Pat Summit	coach	64	TRUE	14/06/1952	28/06/2016			
16	This								
17		has been really fun, but							
18	we're signing								
19			off			now!			

Figure 21.3: Spreadsheet called deaths.xlsx in Excel.

This spreadsheet is one of the example spreadsheets provided in the readxl package. You can use the `readxl_example()` function to locate the spreadsheet on your system in the directory where the package is installed. This function returns the path to the spreadsheet, which you can use in `read_excel()` as usual.

```
deaths_path <- readxl_example("deaths.xlsx")
deaths <- read_excel(deaths_path)
#> New names:
#> • `` -> `...2`
#> • `` -> `...3`
#> • `` -> `...4`
#> • `` -> `...5`
#> • `` -> `...6`
deaths
#> # A tibble: 18 × 6
#>   `Lots of people`    ...2       ...3  ...4     ...5          ...6
#>   <chr>               <chr>      <chr> <chr>    <chr>         <chr>
```

```
#> 1 simply cannot resi… <NA>       <NA>  <NA>     <NA>          some notes
#> 2 at                  the        top   <NA>     of            their spreadsh…
#> 3 or                  merging    <NA>  <NA>     <NA>          cells
#> 4 Name                Profession Age   Has kids Date of birth Date of death
#> 5 David Bowie         musician   69    TRUE     17175         42379
#> 6 Carrie Fisher       actor      60    TRUE     20749         42731
#> # ℹ 12 more rows
```

The top three rows and the bottom four rows are not part of the data frame. It's possible to eliminate these extraneous rows using the `skip` and `n_max` arguments, but we recommend using cell ranges. In Excel, the top left cell is `A1`. As you move across columns to the right, the cell label moves down the alphabet, i.e. `B1`, `C1`, etc. And as you move down a column, the number in the cell label increases, i.e. `A2`, `A3`, etc.

Here the data we want to read in starts in cell `A5` and ends in cell `F15`. In spreadsheet notation, this is `A5:F15`, which we supply to the `range` argument:

```
read_excel(deaths_path, range = "A5:F15")
#> # A tibble: 10 × 6
#>   Name          Profession   Age `Has kids` `Date of birth`
#>   <chr>         <chr>      <dbl> <lgl>      <dttm>
#> 1 David Bowie   musician      69 TRUE       1947-01-08 00:00:00
#> 2 Carrie Fisher actor         60 TRUE       1956-10-21 00:00:00
#> 3 Chuck Berry   musician      90 TRUE       1926-10-18 00:00:00
#> 4 Bill Paxton   actor         61 TRUE       1955-05-17 00:00:00
#> 5 Prince        musician      57 TRUE       1958-06-07 00:00:00
#> 6 Alan Rickman  actor         69 FALSE      1946-02-21 00:00:00
#> # ℹ 4 more rows
#> # ℹ 1 more variable: `Date of death` <dttm>
```

21.2.6 Data types

In CSV files, all values are strings. This is not particularly true to the data, but it is simple: everything is a string.

The underlying data in Excel spreadsheets is more complex. A cell can be one of four things:

- A boolean, like `TRUE`, `FALSE`, or `NA`.
- A number, like "10" or "10.5".
- A datetime, which can also include time like "11/1/21" or "11/1/21 3:00 PM".
- A text string, like "ten".

When working with spreadsheet data, it's important to keep in mind that the underlying data can be very different than what you see in the cell. For example, Excel has no notion of an integer. All numbers are stored as floating points, but you can choose to display the data with a customizable number of decimal points. Similarly, dates are actually stored as numbers, specifically the number of seconds since January 1, 1970. You can customize how you display the date by applying formatting in Excel. Confusingly, it's also possible to have something that looks like a number but is actually a string (e.g., type `'10` into a cell in Excel).

These differences between how the underlying data are stored vs. how they're displayed can cause surprises when the data are loaded into R. By default readxl will guess the data type in a given column. A recommended workflow is to let readxl guess the column types, confirm that you're happy with the guessed column types, and if not, go back and re-import specifying `col_types` as shown in Section 21.2.3.

Another challenge is when you have a column in your Excel spreadsheet that has a mix of these types, e.g., some cells are numeric, others text, others dates. When importing the data into R readxl has to make some decisions. In these cases you can set the type for this column to `"list"`, which will load the column as a list of length 1 vectors, where the type of each element of the vector is guessed.

> ⓘ Sometimes data is stored in more exotic ways, like the color of the cell background, or whether or not the text is bold. In such cases, you might find the tidyxl package useful. See https://nacnudus.github.io/spreadsheet-munging-strategies/ for more on strategies for working with non-tabular data from Excel.

21.2.7 Writing to Excel

Let's create a small data frame that we can then write out. Note that `item` is a factor and `quantity` is an integer.

```
bake_sale <- tibble(
  item     = factor(c("brownie", "cupcake", "cookie")),
  quantity = c(10, 5, 8)
)

bake_sale
#> # A tibble: 3 × 2
#>   item    quantity
#>   <fct>      <dbl>
#> 1 brownie       10
#> 2 cupcake        5
#> 3 cookie         8
```

You can write data back to disk as an Excel file using the `write_xlsx()` from the writexl package:

```
write_xlsx(bake_sale, path = "data/bake-sale.xlsx")
```

Figure 21.4 shows what the data looks like in Excel. Note that column names are included and bolded. These can be turned off by setting `col_names` and `format_headers` arguments to `FALSE`.

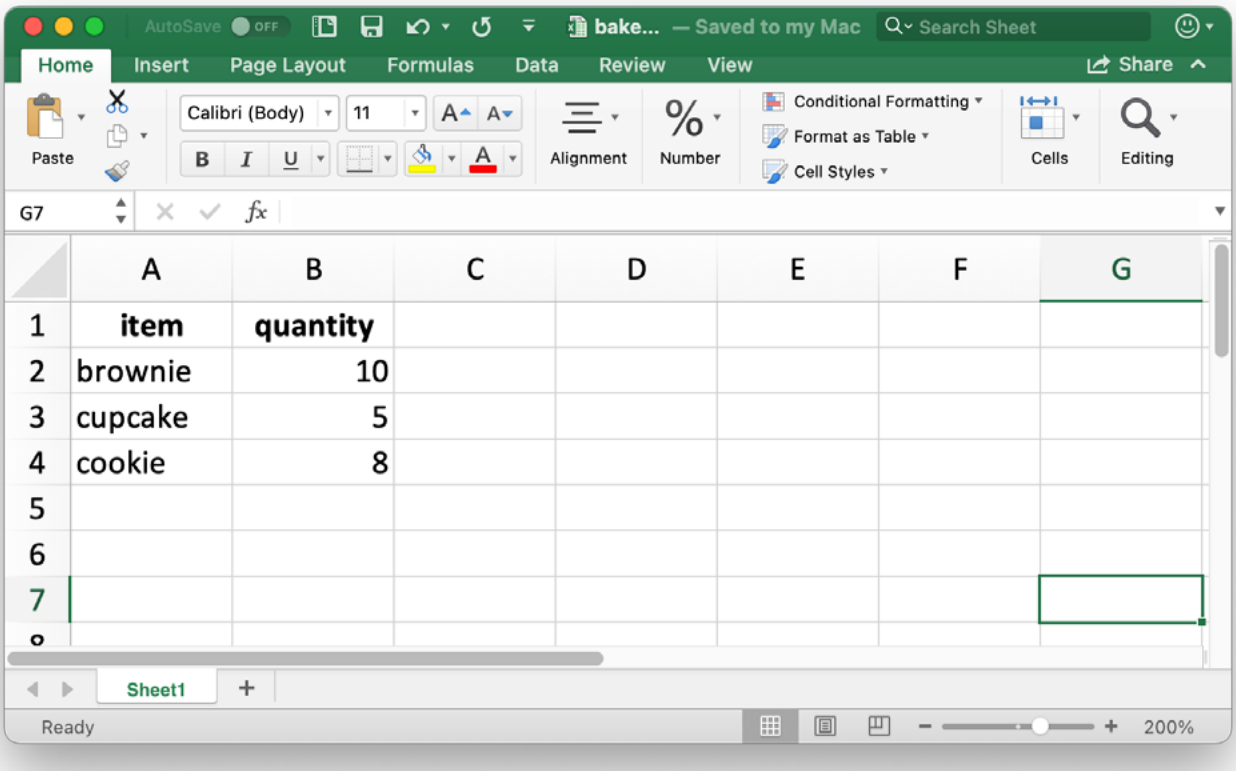


Figure 21.4: Spreadsheet called bake_sale.xlsx in Excel.

Just like reading from a CSV, information on data type is lost when we read the data back in. This makes Excel files unreliable for caching interim results as well. For alternatives, see Section 8.5.

```
read_excel("data/bake-sale.xlsx")
#> # A tibble: 3 × 2
#>   item    quantity
#>   <chr>      <dbl>
#> 1 brownie       10
#> 2 cupcake        5
#> 3 cookie         8
```

21.2.8 Formatted output

The writexl package is a light-weight solution for writing a simple Excel spreadsheet, but if you're interested in additional features like writing to sheets within a spreadsheet and styling, you will want to use the openxlsx package. We won't go into the details of using this package here, but we recommend reading https://ycphs.github.io/openxlsx/articles/Formatting.html for an extensive discussion on further formatting functionality for data written from R to Excel with openxlsx.

Note that this package is not part of the tidyverse so the functions and workflows may feel unfamiliar. For example, function names are camelCase, multiple functions can't be composed in pipelines, and arguments are in a different order than they tend to be in the tidyverse. However, this is ok. As your R learning and usage expands outside of this book you will encounter lots of different styles used in various R packages that you might use to

accomplish specific goals in R. A good way of familiarizing yourself with the coding style used in a new package is to run the examples provided in function documentation to get a feel for the syntax and the output formats as well as reading any vignettes that might come with the package.

21.2.9 Exercises

1. In an Excel file, create the following dataset and save it as `survey.xlsx`. Alternatively, you can download it as an Excel file from here.

	A	B
1	survey_id	n_pets
2	1	0
3	2	1
4	3	N/A
5	4	two
6	5	2
7	6	

   Then, read it into R, with `survey_id` as a character variable and `n_pets` as a numerical variable.

```
#> # A tibble: 6 × 2
#>   survey_id n_pets
#>   <chr>      <dbl>
#> 1 1              0
#> 2 2              1
#> 3 3             NA
#> 4 4              2
#> 5 5              2
#> 6 6             NA
```

2. In another Excel file, create the following dataset and save it as `roster.xlsx`. Alternatively, you can download it as an Excel file from here.

	A	B	C
1	**group**	**subgroup**	**id**
2	1	A	1
3			2
4			3
5		B	4
6			5
7			6
8			7
9	2	A	8
10			9
11		B	10
12			11
13			12

Then, read it into R. The resulting data frame should be called `roster` and should look like the following.

```
#> # A tibble: 12 × 3
#>    group subgroup    id
#>    <dbl> <chr>    <dbl>
#>  1     1 A            1
#>  2     1 A            2
#>  3     1 A            3
#>  4     1 B            4
#>  5     1 B            5
#>  6     1 B            6
#>  7     1 B            7
#>  8     2 A            8
#>  9     2 A            9
#> 10     2 B           10
#> 11     2 B           11
#> 12     2 B           12
```

3. In a new Excel file, create the following dataset and save it as `sales.xlsx`. Alternatively, you can download it as an Excel file from here.

	A	B
1	This file contains information on sales.	
2	Data are organized by brand name, and for each brand, we have the ID number for the item sold, and how many are sold.	
3		
4		
5	Brand 1	n
6	1234	8
7	8721	2
8	1822	3
9	Brand 2	n
10	3333	1
11	2156	3
12	3987	6
13	3216	5

a. Read `sales.xlsx` in and save as `sales`. The data frame should look like the following, with `id` and `n` as column names and with 9 rows.

```
#> # A tibble: 9 × 2
#>   id      n
#>   <chr>   <chr>
#> 1 Brand 1 n
```

```
#> 2 1234    8
#> 3 8721    2
#> 4 1822    3
#> 5 Brand 2 n
#> 6 3333    1
#> 7 2156    3
#> 8 3987    6
#> 9 3216    5
```

b. Modify `sales` further to get it into the following tidy format with three columns (`brand`, `id`, and `n`) and 7 rows of data. Note that `id` and `n` are numeric, `brand` is a character variable.

```
#> # A tibble: 7 × 3
#>   brand      id     n
#>   <chr>   <dbl> <dbl>
#> 1 Brand 1  1234     8
#> 2 Brand 1  8721     2
#> 3 Brand 1  1822     3
#> 4 Brand 2  3333     1
#> 5 Brand 2  2156     3
#> 6 Brand 2  3987     6
#> 7 Brand 2  3216     5
```

4. Recreate the `bake_sale` data frame, write it out to an Excel file using the `write.xlsx()` function from the openxlsx package.

5. In Chapter 8 you learned about the `janitor::clean_names()` function to turn columns names into snake case. Read the `students.xlsx` file that we introduced earlier in this section and use this function to "clean" the column names.

6. What happens if you try to read in a file with `.xlsx` extension with `read_xls()`?

21.3 Google Sheets

Google Sheets is another widely used spreadsheet program. It's free and web-based. Just like with Excel, in Google Sheets data are organized in worksheets (also called sheets) inside of spreadsheet files.

21.3.1 Prerequisites

This section will also focus on spreadsheets, but this time you'll be loading data from a Google Sheet with the **googlesheets4** package. This package is non-core tidyverse as well, you need to load it explicitly.

```
library(googlesheets4)
library(tidyverse)
```

A quick note about the name of the package: googlesheets4 uses v4 of the Sheets API v4 to provide an R interface to Google Sheets, hence the name.

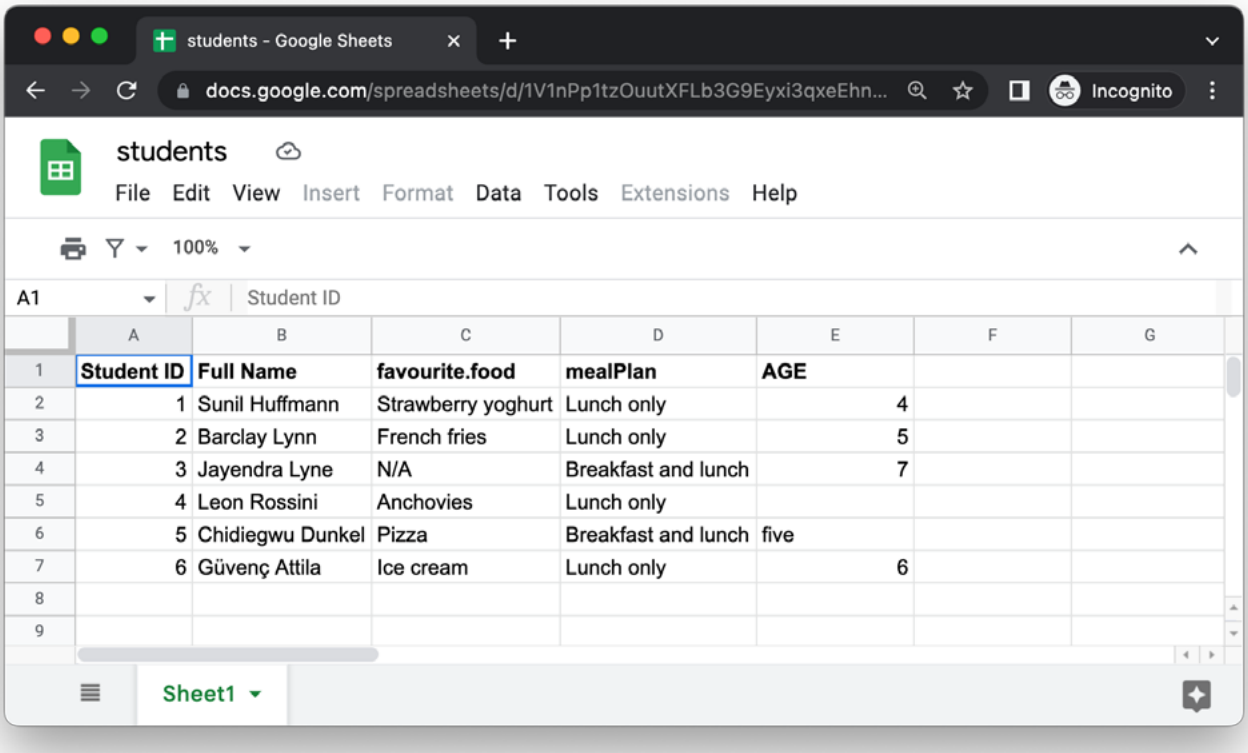
The main function of the googlesheets4 package is `read_sheet()`, which reads a Google Sheet from a URL or a file id. This function also goes by the name `range_read()`.

You can also create a brand new sheet with `gs4_create()` or write to an existing sheet with `sheet_write()` and friends.

In this section we'll work with the same datasets as the ones in the Excel section to highlight similarities and differences between workflows for reading data from Excel and Google Sheets. readxl and googlesheets4 packages are both designed to mimic the functionality of the readr package, which provides the `read_csv()` function you've seen in Chapter 8. Therefore, many of the tasks can be accomplished with simply swapping out `read_excel()` for `read_sheet()`. However you'll also see that Excel and Google Sheets don't behave in exactly the same way, therefore other tasks may require further updates to the function calls.

21.3.3 Reading Google Sheets

Figure 21.5 shows what the spreadsheet we're going to read into R looks like in Google Sheets. This is the same dataset as in Figure 21.1, except it's stored in a Google Sheet instead of Excel.



	A	B	C	D	E	F	G
1	Student ID	Full Name	favourite.food	mealPlan	AGE		
2	1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4		
3	2	Barclay Lynn	French fries	Lunch only	5		
4	3	Jayendra Lyne	N/A	Breakfast and lunch	7		
5	4	Leon Rossini	Anchovies	Lunch only			
6	5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five		
7	6	Güvenç Attila	Ice cream	Lunch only	6		
8							
9							

Figure 21.5: Google Sheet called students in a browser window.

The first argument to `read_sheet()` is the URL of the file to read, and it returns a tibble: https://docs.google.com/spreadsheets/d/1V1nPp1tzOuutXFLb3G9Eyxi3qxeEhnOXUzL5_BcCQ0w. These URLs are not pleasant to work with, so you'll often want to identify a sheet by its ID.

```
students_sheet_id <- "1V1nPp1tzOuutXFLb3G9Eyxi3qxeEhnOXUzL5_BcCQ0w"
students <- read_sheet(students_sheet_id)
```

```
#> ✔ Reading from students.
#> ✔ Range Sheet1.
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
#>          <dbl> <chr>            <chr>              <chr>               <list>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          <dbl>
#> 2            2 Barclay Lynn     French fries       Lunch only          <dbl>
#> 3            3 Jayendra Lyne    N/A                Breakfast and lunch <dbl>
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NULL>
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch <chr>
#> 6            6 Güvenç Attila    Ice cream          Lunch only          <dbl>
```

Just like we did with `read_excel()`, we can supply column names, NA strings, and column types to `read_sheet()`.

```
students <- read_sheet(
  students_sheet_id,
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1,
  na = c("", "N/A"),
  col_types = "dcccc"
)
#> ✔ Reading from students.
#> ✔ Range 2:10000000.

students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Note that we defined column types a bit differently here, using short codes. For example, “dcccc” stands for “double, character, character, character, character”.

It’s also possible to read individual sheets from Google Sheets as well. Let’s read the “Torgersen Island” sheet from the penguins Google Sheet:

```
penguins_sheet_id <- "1aFu8lnD_g0yjF5O-K6SFgSEWiHPpgvFCF0NY9D6LXnY"
read_sheet(penguins_sheet_id, sheet = "Torgersen Island")
#> ✔ Reading from penguins.
#> ✔ Range ''Torgersen Island''.
#> # A tibble: 52 × 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm
#>   <chr>   <chr>     <list>         <list>        <list>
```

```
#> 1 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 2 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 3 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 4 Adelie  Torgersen <chr [1]>      <chr [1]>     <chr [1]>
#> 5 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 6 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> # ℹ 46 more rows
#> # ℹ 3 more variables: body_mass_g <list>, sex <chr>, year <dbl>
```

You can obtain a list of all sheets within a Google Sheet with `sheet_names()`:

```
sheet_names(penguins_sheet_id)
#> [1] "Torgersen Island" "Biscoe Island"    "Dream Island"
```

Finally, just like with `read_excel()`, we can read in a portion of a Google Sheet by defining a `range` in `read_sheet()`. Note that we're also using the `gs4_example()` function below to locate an example Google Sheet that comes with the googlesheets4 package.

```
deaths_url <- gs4_example("deaths")
deaths <- read_sheet(deaths_url, range = "A5:F15")
#> ✔ Reading from deaths.
#> ✔ Range A5:F15.
deaths
#> # A tibble: 10 × 6
#>   Name          Profession   Age `Has kids` `Date of birth`
#>   <chr>         <chr>      <dbl> <lgl>      <dttm>
#> 1 David Bowie   musician      69 TRUE       1947-01-08 00:00:00
#> 2 Carrie Fisher actor         60 TRUE       1956-10-21 00:00:00
#> 3 Chuck Berry   musician      90 TRUE       1926-10-18 00:00:00
#> 4 Bill Paxton   actor         61 TRUE       1955-05-17 00:00:00
#> 5 Prince        musician      57 TRUE       1958-06-07 00:00:00
#> 6 Alan Rickman  actor         69 FALSE      1946-02-21 00:00:00
#> # ℹ 4 more rows
#> # ℹ 1 more variable: `Date of death` <dttm>
```

21.3.4 Writing to Google Sheets

You can write from R to Google Sheets with `write_sheet()`. The first argument is the data frame to write, and the second argument is the name (or other identifier) of the Google Sheet to write to:

```
write_sheet(bake_sale, ss = "bake-sale")
```

If you'd like to write your data to a specific (work)sheet inside a Google Sheet, you can specify that with the `sheet` argument as well.

```
write_sheet(bake_sale, ss = "bake-sale", sheet = "Sales")
```


21.3.5 Authentication

While you can read from a public Google Sheet without authenticating with your Google account, reading a private sheet or writing to a sheet requires authentication so that googlesheets4 can view and manage *your* Google Sheets.

When you attempt to read in a sheet that requires authentication, googlesheets4 will direct you to a web browser with a prompt to sign in to your Google account and grant permission to operate on your behalf with Google Sheets. However, if you want to specify a specific Google account, authentication scope, etc. you can do so with `gs4_auth()`, e.g., `gs4_auth(email = "mine@example.com")`, which will force the use of a token associated with a specific email. For further authentication details, we recommend reading the documentation googlesheets4 auth vignette: https://googlesheets4.tidyverse.org/articles/auth.html.

21.3.6 Exercises

1. Read the `students` dataset from earlier in the chapter from Excel and also from Google Sheets, with no additional arguments supplied to the `read_excel()` and `read_sheet()` functions. Are the resulting data frames in R exactly the same? If not, how are they different?

2. Read the Google Sheet titled survey from https://pos.it/r4ds-survey, with `survey_id` as a character variable and `n_pets` as a numerical variable.

3. Read the Google Sheet titled roster from https://pos.it/r4ds-roster. The resulting data frame should be called `roster` and should look like the following.

```
#> # A tibble: 12 × 3
#>    group subgroup    id
#>    <dbl> <chr>    <dbl>
#>  1     1 A            1
#>  2     1 A            2
#>  3     1 A            3
#>  4     1 B            4
#>  5     1 B            5
#>  6     1 B            6
#>  7     1 B            7
#>  8     2 A            8
#>  9     2 A            9
#> 10     2 B           10
#> 11     2 B           11
#> 12     2 B           12
```

21.4 Summary

Microsoft Excel and Google Sheets are two of the most popular spreadsheet systems. Being able to interact with data stored in Excel and Google Sheets files directly from R is a superpower! In this chapter you learned how to read data into R from spreadsheets from Excel with `read_excel()` from the readxl package and from Google Sheets with `read_sheet()` from the googlesheets4 package. These functions work very similarly to each other and have similar arguments for specifying column names, NA strings, rows to skip on top of the file you're reading. Additionally, both functions make it possible to read a single sheet from a spreadsheet as well.

On the other hand, writing to an Excel file requires a different package and function (`writexl::write_xlsx()`) while you can write to a Google Sheet with the googlesheets4 package, with `write_sheet()`.

In the next chapter, you'll learn about a different data source and how to read data from that source into R: databases.

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

This book was built with Quarto.

Topic 3: File Environment

Lecture 3: File Environments

File environments

- Working directory
- Referencing files
    - Absolute paths
    - Relative paths

The **"working directory"** refers to the specific folder or directory where you're currently operating. In an ideal scenario, if you're working on a project, all the related data and code would be contained within that project's specific directory. However, this isn't always practical, as sometimes files from outside the working directory need to be accessed.

How do you find out what your current working directory is?

To determine your current working directory in R, use the following command:

```
getwd()
```

```
[1] "/home/rstudio/PHW251_Fall2023/weekly_material/week_3"
```

Hi, everyone. This is a really quick video to talk a little bit more about file environments in R. This is specifically relevant to when we're wanting to read in or export data files to different areas within R working environment. This is less relevant to the situations where you may be reading in data from an API or a URL.

In this short video, we're going to talk a little bit about the working directory in R environment, we're going to talk about referencing files through absolute paths and relative paths. We'll start with a working directory, this is a specific folder or directory that you're currently operating in. Since I'm working on this file_environment.qmd file, that's within our course repository, weekly material, and Week 3. This should be my working directory.

In an ideal scenario, if you're working on a project, all of the related data and code would be contained within that project-specific directory. Maybe it's within sub-folders or maybe it's just all within the main folder we're working in. But there are certainly times when that might not be practical and there may be files saved elsewhere that you want to pull into whatever you're working on.

To start, we can find out what R current working directory is by running this getwd(). If we run this, we'll see that my current working directory is this home/rstudio/PHW251_Fall2023/weekly_material/week_3 folder, which aligns with what we expect since I'm working on this file environment, QMD script.

Referencing Files

Within the Working Directory: To access a file within your working directory, you can simply use the filename enclosed in quotes:

```
library(readr)

data <- read_csv("homeless_impact.csv")
```

```
Rows: 14517 Columns: 7
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (1): county
dbl  (5): rooms, rooms_occupied, trailers_requested, trailers_delivered, don...
date (1): date

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Using Absolute Paths: The "absolute path" is a complete path starting from the root directory of your system (or from the root on a server, like DataHub) to the specific file. You will need to replace the working directory in `[REPLACE]` based on what you see when running `getwd()`.

```
data2 <- read_csv("/home/rstudio/PHW251_Fall2023/weekly_material/week_3/homeless_impact.c
```

Within the working directory, to access a file that you need, you can simply use the filename enclosed in quotes. We can read in this homeless_impact.csv by just using the read CSV function, homeless_impact.csv. If we run that, we do see that it loaded. We can do the same using absolute paths. This is a complete path starting from the root directory of your system. So in datahub, that would be home. But if you're working on a different system, it might look a little bit different, but it's the whole path from the root until the specific file. You'll need to replace this little place that's in brackets with the working directory that you see when you run getwd.

I'm going to go ahead and copy what I had here and put it where the replace packet is, and make sure there's only a single slashes, and everything looks okay, everything's within a single string, and changes to that read CSV function, that's part of read R, and I'm going to try this again. You can see we also loaded the same dataset using this entire path.

Really absolute paths are what we will recommend using, this is the biggest fail-safe if you are using data that needs to live somewhere else besides where your script is or say you're moving your script somewhere else or you're sharing with someone else. I think this just makes it the clearest, this is where the data that I'm importing is saved, whereas this line is not as obvious.

Using Relative Paths: Relative paths are a way of navigating the directory structure in relation to the current location. For example, `../` moves up one level in the directory, while specific folder names like "week_1" enter that directory. Here's how to change the working directory using relative paths:

```
setwd("../week_0")
getwd()
```

```
[1] "/home/rstudio/PHW251_Fall2023/weekly_material/week_0"
```

```
data3 <- read_csv("data/sample_data.csv")
```

```
Rows: 10 Columns: 2
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
dbl (2): ID, test_results

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

However, that only changed the working directory for that chunk. The next chunk will reset to the folder this file is saved in.

```
getwd()
```

```
[1] "/home/rstudio/PHW251_Fall2023/weekly_material/week_3"
```

Note

While working directories can be convenient, they can sometimes be tricky, particularly in Quarto / R Markdown. **We recommend using absolute paths when referencing files because they're more reliable.**

But we also do want to talk about relative paths, though, this is another way of navigating the directory structure in relation to your current location. For example, this dot, dot or period, period forward slash moves up one level in the directory, while the specific folder names like Week 1 or Week 0 enter that directory.

Here's how to change the working directory using relative paths, and just to demonstrate what the period, period, forward slash is. If R current working directory is this Week 3 folder, using the period, period, forward slash moves up to the weekly material. If we use that, then you see we have our choice of Week 0, Week 1, Week 2, or Week 3. We're going to try out setting our working directory to Week 0 using the period, period, forward slash. If we run that, it will set our working directory, it does yell at us to let us know that we did this.

Actually, it changed it back. That's interesting. But if we set the working directory, you know why, it's because they didn't run the chunk in full. What this error is saying is that it will change the working directory inside a notebook chunk only. If we run this as a chunk, we can actually pull in some of the data from Week 0. You can see I have R working directory change to Week 0, and then there's this data folder, and I'm going to pull in sample data. So let's run this whole chunk.

When we do this, you can see this is the setting of the working directory. This is that get working directory, and then we've created a new dataset with that sample data CSV. But as I mentioned that only change the working directory for that specific chunk, the next chunk will reset to the folder the file is saved in. If we run getwd again, we're back with our working directory in Week 3.

I know I already said this, but want to just highlight that while working directories can be really convenient, they can be tricky and they're especially maybe tricky for your future self or for others you might be working with. They're extra tricky in Quarto and R Markdown. We highly recommend using absolute paths when referencing any files because they are more reliable. Just for a refresher, this is what the absolute path looks like, it has our entire pathway from our root directory to the file we're importing.