PS 312: Programming with R Course Notes

Abhijit Dasgupta, PhD

Last updated: March 24, 2019

Welcome

This course is an introduction to the statistical programming language R and various applications. We will cover the entire data analytics pipeline from data ingestion to data wrangling, summarizing, modeling, visualizing and reporting, all using tools found within the R ecosystem.

The version of these notes you are reading now was built on 2019-03-24.

Reproducibility

These notes are written with `bookdown`, a R package for writing books using `rmarkdown`. All code in these notes were developed on R version 3.5.0 (2018-04-23), using the same packages pre-installed in your virtual machines. When you're on your own, you will need to install a recent version of R, and also install the corresponding packages, on your computer, for all the code to work. A listing of all the packages used in this course will be available as an appendix.

To build these notes locally, clone or download the Github repo hosting these notes, unzip it if necessary, and double-click on `FSI_Book.Rproj`. Assuming you have RStudio installed, this will open this project (more on *RStudio Projects* later). You can then go to the console and enter the following code:

```
bookdown::render_book("index.Rmd") # to build these notes
browseURL("_book/index.html") # to view it
```

Starting up

Chapter 1

What is R?

R is the most popular[1] open source *statistical programming language* in the world. It allows you to

1. read datasets written in a wide variety of formats,
2. clean and process the data,
3. derive summaries,
4. run analytics,
5. visualize
6. create automated reports, presentations, websites, dashboards and interactive applications

R is not just a language, but an ecosystem comprising over 15,000 user- and corporation-developed *packages* or modules, all written in the R language for a variety of purposes. It is a very flexible and customizable language, which is why it is used by an estimated 2 million users worldwide for data analytics. The question R users often ask is not "Can it be done?" but rather "How can it be done?". R is used in areas as varied as healthcare, economics, forestry, oceanography, pharmaceuticals, artificial intelligence and natural language processing.

Why is R so widely used? Some reasons are:

- R is open source, so it is accessible to anyone with a computer
- Since the code in R and all its packages are open, the community of users can help debug it and make it more reliable and robust
- The R ecosystem is very rich in tools for doing data analytics in particular, so there is almost certainly something available for almost any task
- The community of R users worldwide is a very strong, well-connected group who are welcoming, ready to help, cooperative and inclusive. Many users find this community to be one of the most attractive things about R
- R produces really nice customizable visualizations with relatively little effort, which was one of the first reasons for popularity.

[1] https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

A note on coding and programming

R does not have a *point-and-click* interface that you are probably more familiar with from Excel, Word or other computer applications. It requires you to *code*, i.e. write instructions for the computer to, in the case of R, read, analyze, graph and report on datasets.

R is first and foremost a **language**. So, instead of thinking that this is some geeky thing that "programmers" and "IT people" do, think of it as learning a language. You will see that, like any language, it has nouns, verbs, adjectives and adverbs, and you can create "sentences" that start with data and end in something useful like a table, graph or document. With a traditional spoken and written language like French, Arabic, Farsi or Japanese, you learn it to be able to interact with people at different posts around the world. With a programming language like R, you will be able to interact with **data**, to make sense of it, to describe it, and to present it.

Coding

Coding is writing explicit instructions to a very literal, and in some ways, stupid machine. The machine takes our code literally, and will do **exactly** what you tell it to do in the code. If you are getting unexpected results, it's almost certainly your code that needs to be checked, not the machine.

R, the language

As we will see, R has many elements of a language.

- **Objects**: These are the *nouns*. We will act on objects to create new objects. Each object has a *name* which we will treat as the nouns in our code.
- **Functions**: These are the *verbs*. Functions will act on objects to create new objects.
- **The `%>%` operator**: This acts like the conjunction "then" to create "sentences" called *pipes* or *chains*.
- **Optional function arguments**: These are adverbs which modify the action of the function (verb).

While writing code in R, we should be aware that R is **case-sensitive**, so `mydata` is a different object than `myData` which is also different from `Mydata` and `My_Data` and `MyData` and `my_data`, and `mean`, which is a function in R, is different from `Mean` which is not defined in R.

> You have to name all the objects you create in R if you want to see them again. Try and pick a naming system that is simple yet descriptive, rather than `data1`. Two typical conventions that are used are CamelCase and pothole_case. So you could name a dataset of operational budgets for January, 2019 as `operations_budget_2019_jan` or `OperationsBudget2019Jan` or really anything you want, as long as it's clear to you and doesn't include some forbidden characters like `-`, `@`,`$` which are reserved for other purposes, or doesn't start with a number.
>
> Some people have a system where data objects (which are called `data.frame` or `tibble` or `vector` or `matrix`) should be capitalized, while function names should not. Data objects should probably be nouns and functions verbs, since that reminds us of their functions. There are different opinions. Some influential ones are here and here. As they say, finding a good name is hard, but often worth the effort.

The ultimate goal for every script file is to create a "story" using the language of R, starting from data to create descriptions, understand patterns through visualization and modeling, and analyzing the data in general. Scripts make this story reproducible, and also transferable to different data sets.

Of course, as with any beginner writer, your coding will be sloppy at first, will suffer many stops and starts and strike-throughs and modifications and throwing things into the proverbial trash. With practice, this will become easier and smoother and more effective and more expressive. This workshop is designed to give you an initial push towards that goal.

So, let's start this journey.

1.1 A short introduction to R objects

1.1.1 Objects

The broad categorization of R objects in my mind are functions (verbs) and data objects (nouns). Data objects are in turn of different types:

- `data.frame` or `tibble`: These are rectangular data sets much like you would see in a spreadsheet
- `vector`: This is a 1-dimensional list of numbers or strings (words in the language sense), but all must be of the same kind (number or string)
- `matrix`: This is a 2-dimensional list of numbers or strings, once again all of the same type
- A single number or word
- `list`: This is a catch-all bucket. Each element of a list can be literally any valid R object. So they could be `tibble`'s, or functions, or matrices, and different elements can be of different types.

Most objects we'll use in this workshop are going to be `data.frame` or `tibble` objects. (In case you're wondering, they're basically the same thing, but `tibble`'s have some modest additional functionality). R comes with a bunch of built-in datasets stored as `data.frames`.

```
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
```

```
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

A `data.frame` can be acted upon by different functions to help describe it and extract elements from it. For example, to see the size of the data, we use

```r
dim(mtcars)
```

```
## [1] 32 11
```

Data sets often have row names and column names. These can be extracted by the functions `rownames` and `colnames` or `names`:

```r
rownames(mtcars)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

```r
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

Both of these are valid R objects that are vectors of strings. You could save them for future use by *assigning* them a name using the assignment operator `<-`. So if you wanted to store the row names, which are the makes and models of the cars in this data set (this structure is not desirable, as we'll discuss later), you could run

```
car_names <- rownames(mtcars)
```

You can see that this is stored in R for current use, either by typing `ls()` in the console (for "list") or by looking in the `Environment` pane in RStudio.

> The output of any function is a valid R object, and so you can always store the results of the function by assigning it a name, as above.

1.1.2 Extracting elements from objects

We can see the structure of any object by using the function `str`.

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

This tells us that `mtcars` is a `data.frame` with 32 observations (rows) and 11 variables (columns). Each variable has a name, and all the variables are numeric.

> `data.frame` objects are like lists, in that each column can be of a different type. This is a very powerful structure, since we can keep all sorts of data together, and can load spreadsheets with diverse kinds of data easily into R

To extract the `mpg` variable from this data set, there are a few equivalent methods. My preferred method is `mtcars[,'mpg']`, i.e., extract the *column* named "mpg" from this data set. Notice that we're using `[]` while functions use `()`. This format of extraction will work when you're extracting more than one variable, as we'll see below. Other ways include `mtcars[['mpg']]` and `mtcars$mpg`, which are the `list` way and a `data.frame`-specific shortcut.

> You can also extract elements by position, either using the `[,]` or `[[]]` forms. So, to extract an element in the 2nd row and 4th column, you'd have to use the matrix notation as `mtcars[2,4]`. To extract the 4th column, you could use either `mtcars[,4]` or `mtcars[[4]]`. To extract the 2nd row, you'd again use the matrix notation as `mtcars[2,]`.

If we want to extract the mpg, cyl and disp variables at once to create a new `data.frame`, you can use either the matrix notation `mtcars[,c('mpg','cyl','disp')]` or the list notation `mtcars[[c('mpg','cyl','disp')]]`. The `c()` function stands for *concatenate* and is the function used to **create** vectors. We'll actually see a much more user-friendly way of doing this in the data munging section (Chapter 4).

> **Advanced note:** A `data.frame` object is really a `list` object where all the elements are vectors of the same length, and which happen to have names assigned to them. The object also looks like a matrix or 2-dimensional array visually. So both notations were allowed to be valid for `data.frame` objects.

1.2 R Packages

Packages are modules of R code that enhance the capabilities of R. Many packages are well established and curated, and have to go through a strict software compatibility review before allowed on CRAN.

Installing packages

Installing packages on your computer can be done from the RStudio menu (`Tools > Install Packages`), or by running the command `install.packages(<package name>, repos = "https://cran.rstudio.com")`. For example, to install the `readxl` package, which we will use shortly, we would run the code

```
install.packages("readxl", repos="https://cran.rstudio.com")
```

> You can set the default repository in RStudio, in `Tools > Global Options`.
>
> Be aware that everything here is case-sensitive

Another way to install packages is to go to the `Packages` pane in RStudio, use the search bar there to find the package you want to install, and then click the checkbox beside the name. This is convenient, but not very reproducible if you have to move to a different computer, so it's generally discouraged.

How do you find packages? Glad you asked. The easiest way to find packages on CRAN is actually through the RStudio `Packages` pane, where the entire set of available packages are listed with a brief, top-line description. You can click on the package name to see a much more detailed overview of the packages, and many packages do have vignettes which give more information. However, once you've found the package you want, you should really code it up with `install.packages`, so that you can save the script for later when you might need to remember it again.

1.2.1 Loading packages in R

We will use several packages to enhance our experience and get going faster. However, to use a package, you must first load it into R. To load a package into R, you use the function `library` (ironically).

The first package we will load is the `tidyverse` package. This is actually a meta-package, which in turn loads a bunch of other packages. These form a core group of useful packages that are widely used, including

- readr (reading data from text files)
- tidyr (Manipulation, pivoting)
- dplyr (summarize, aggregate, filter)
- ggplot2 (visualization)
- purrr (functions applied across data structures, meta-programming)

- stringr (string manipulation)
- forcats (categorical data)

In addition, we'll load the `readxl` package for reading Excel files.

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

1.3 R Resources

There are many high quality resources for learning R available online. This is a selection of what I find most useful.

1. CRAN Task Views: These are curated lists of R packages for various purposes, ranging from econometrics to mathematics, finance, imaging, social sciences, time series, spatial analyses and more.
2. RStudio Cheatsheets: These are high-quality cheatsheets about different aspects of the R analytic pipeline.
3. StackOverflow #r: The `r` tag on StackOverflow is *the* place to find answers about R
4. Twitter #rstats: The who's who of R hang out at the #rstats hashtag, and questions can get answered very quickly. Also a way to find out what new packages are coming up
5. R-Bloggers: A blog aggregator which collects a few hundred R-related blogs in one place (including mine, in the interests of disclosure)
6. RSeek: When one realizes that R is just a letter in the alphabet, Google searches can be a bit difficult. RSeek has created a custom search targeted at R-related topics, sites and packages on the web.

Chapter 2

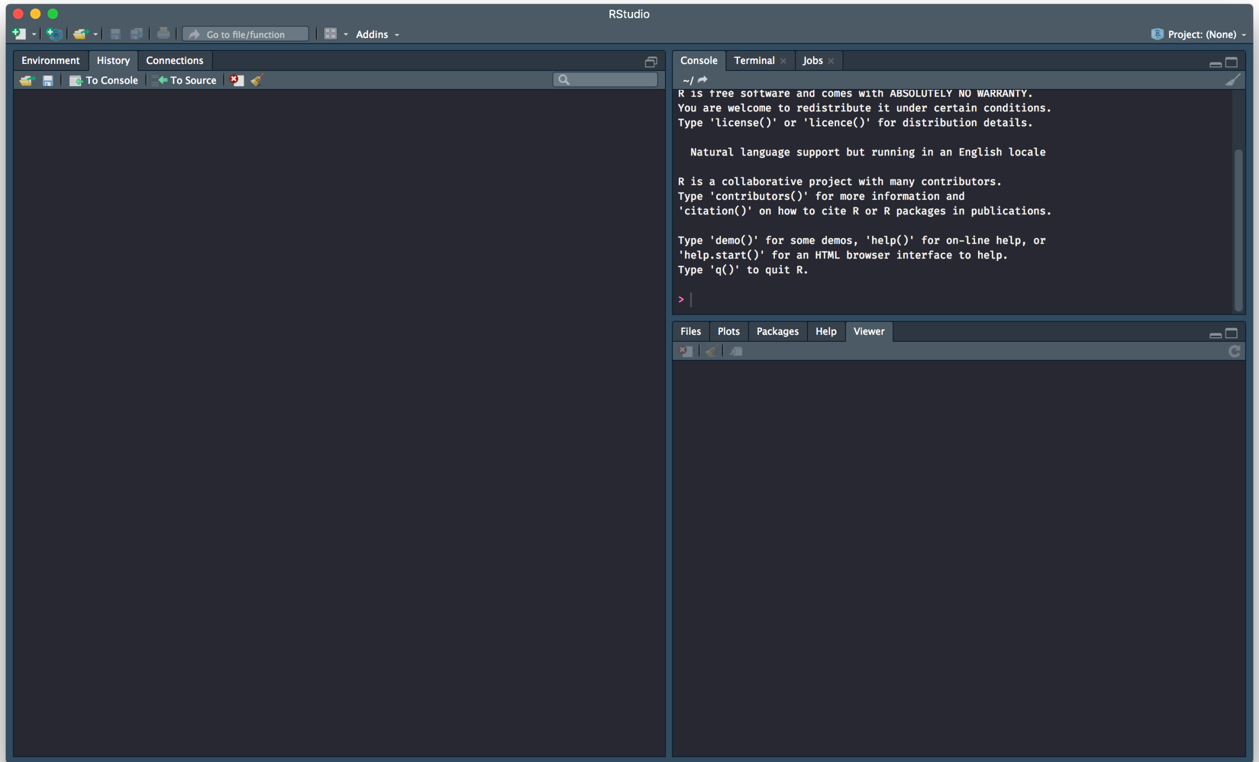
RStudio, your development environment (authoring program)

While R is the language we will learn, RStudio is the interface (or *integrated development environment*) we will use to write it, interact with it, and see our results. RStudio provides by far the most user-friendly interface to R (though it's not point-and-click).

This is the "journal" or "notebook" in which you will start your writing journey in R

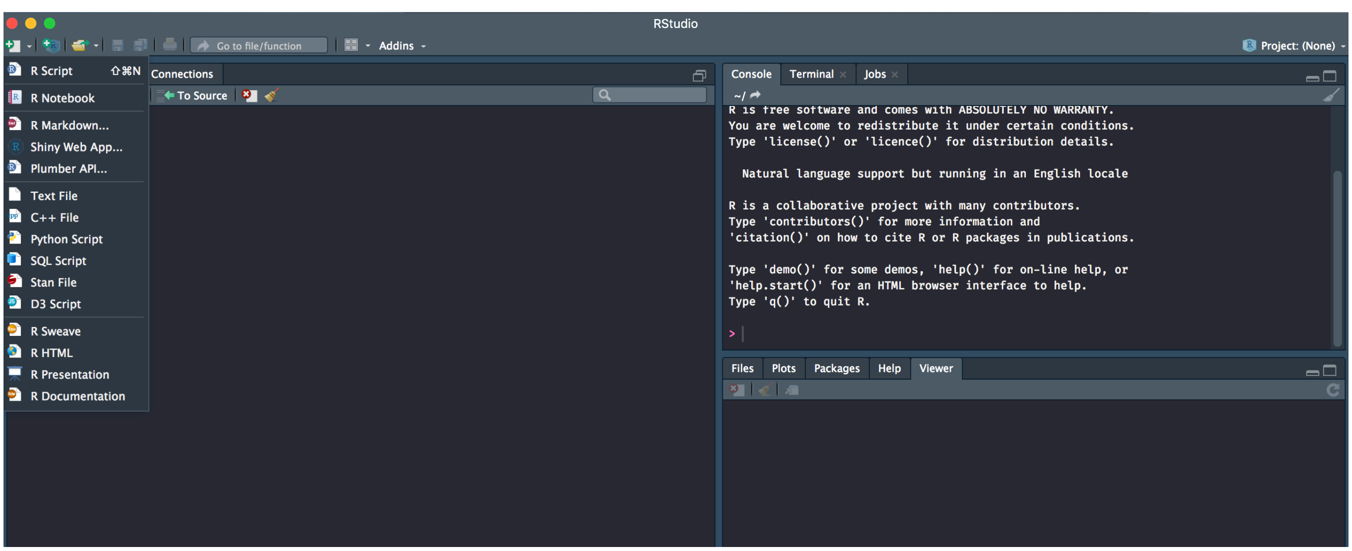
Starting RStudio

When you open RStudio, either from your desktop or from the Start menu, you'll see something like this:

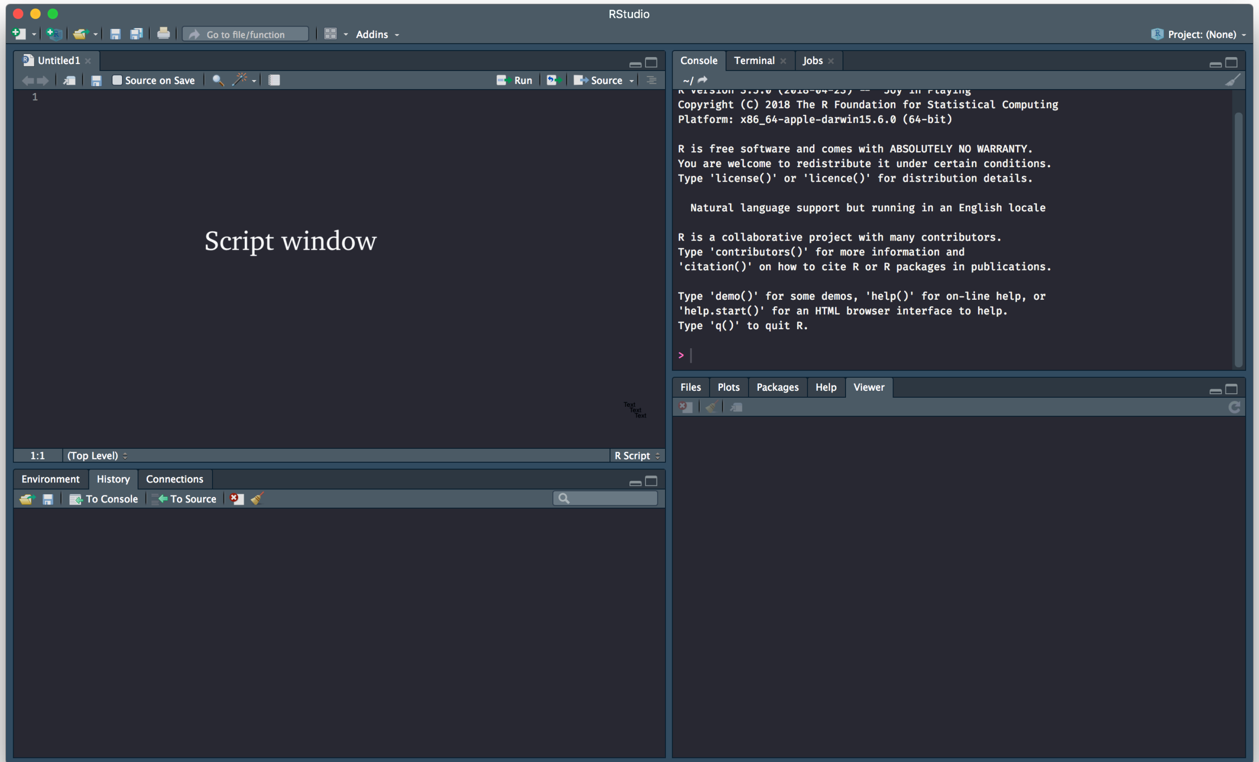


> I'll note that I've done some customization to my console, which you can also do by going to `Tools > Global Options`. Your screen will most likely have a white rather than a dark background

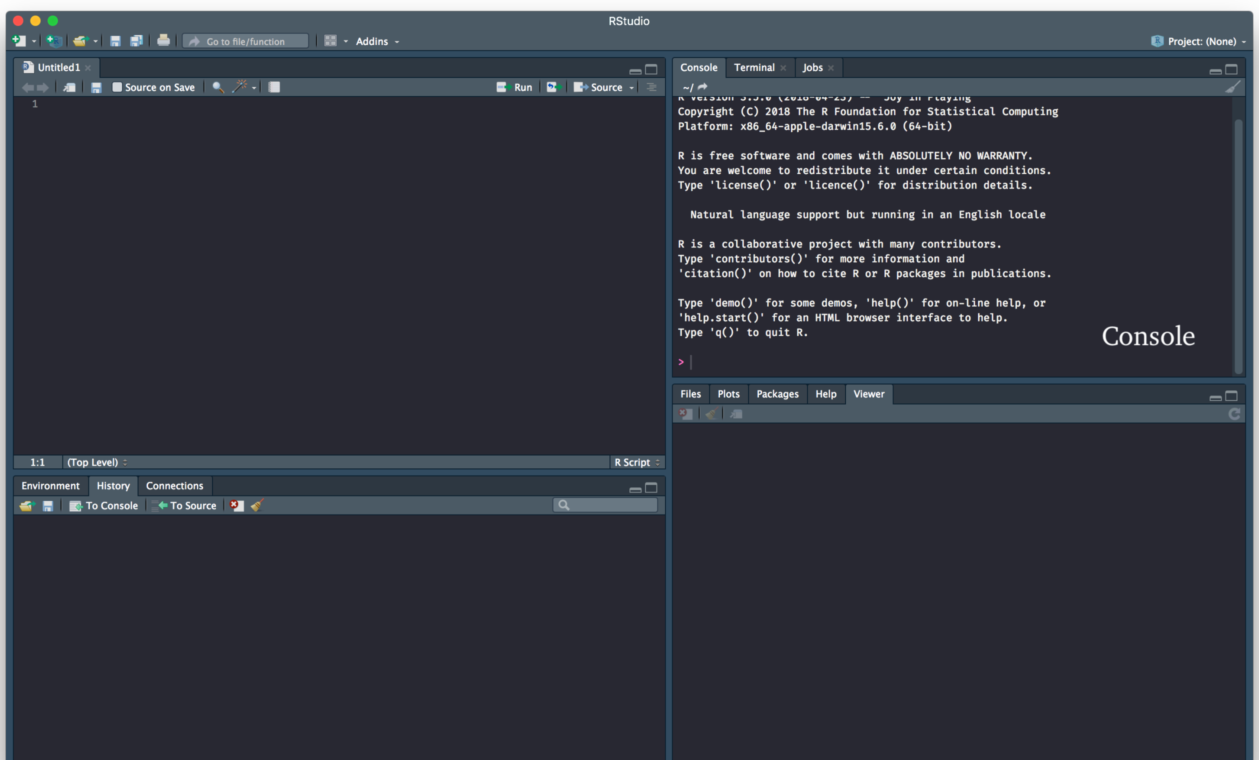
You can open a new panel for an R script using either `File > New File > R Script` or using the button at the top left of the window:



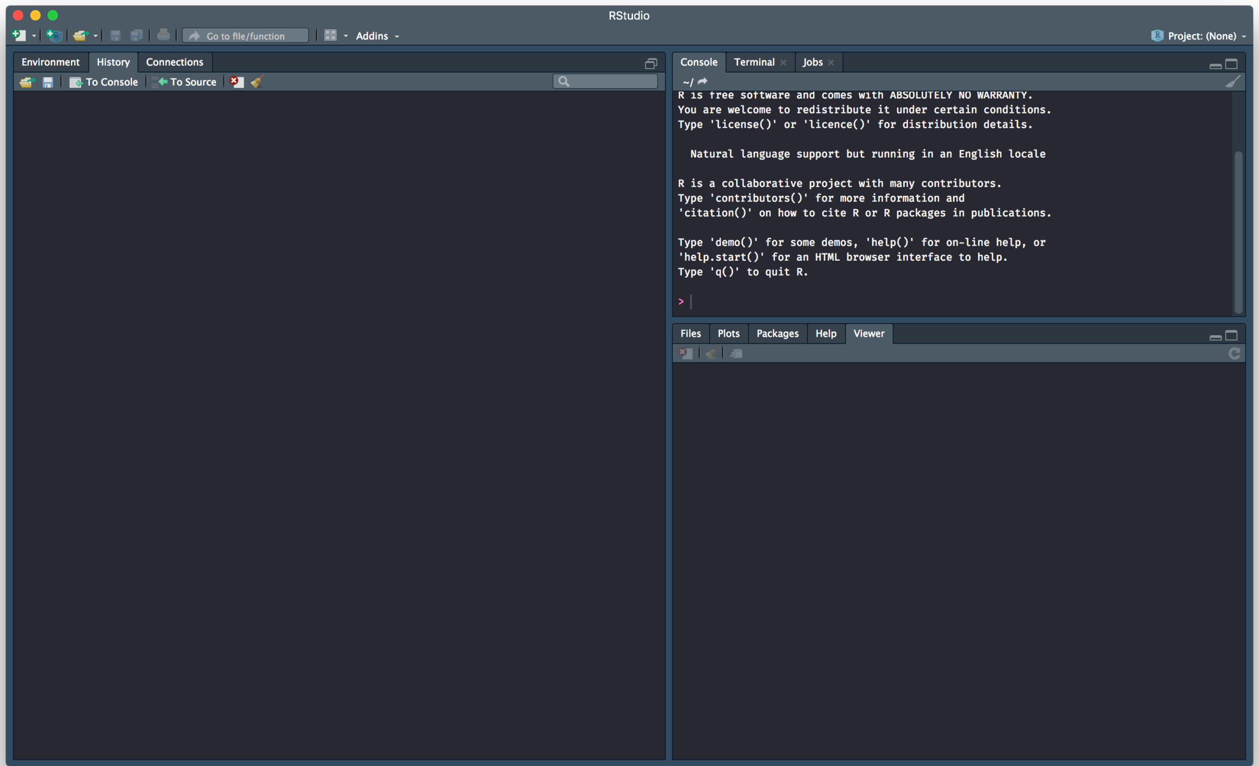
This opens up an R script file that you can edit and save. You will mainly be writing in this panel within a R script (see 2.1 for more details).



You will also have a Console panel where the code will actually run in R.



Other panes



There are several other panes in RStudio that we will see in due course.

- **Environment**: This shows all the objects ("words") in your current environment
- **History**: This gives a history of the commands you have run. This is searchable. Though you do have a stored history, see 2.1 for why you shouldn't fall to temptation to just code in the console.
- **Files**: This is exactly like File Explorer in Windows, and lets you see the contents of a folder/directory
- **Plot**: These is where the plots will show up. See **??** for more details on how to create plots
- **Packages**: This gives a listing of installed packages. You *can* click on the tick boxes to load packages into your environment, but I prefer coding it in (see section **??**) to make it reproducible and verifiable.
- **Help**: This will show help files once they are evoked
- **Viewer**: This pane shows results when they are produced as HTML documents. This pane will also come into play once we start with interactive visualizations in section **??**.

Feel free to explore these different panes and understand their functionalities.

2.1 Rstudio workflow

As we've seen, RStudio has both a scripting pane to write code, and a console pane to run code. Of course, you can write code directly into the console, but it is **not** a good practice. You will tend to get sloppy, lose the "story", and generally have less reproducible code.

Writing the program ("story") is just more reliable if you write into the script file and the send it to the console to run. Sending it to the console can be acheieved with a keyboard shortcut, Ctrl-Enter (or Cmd-Enter on a Mac). This is something that will be second nature while coding in RStudio.

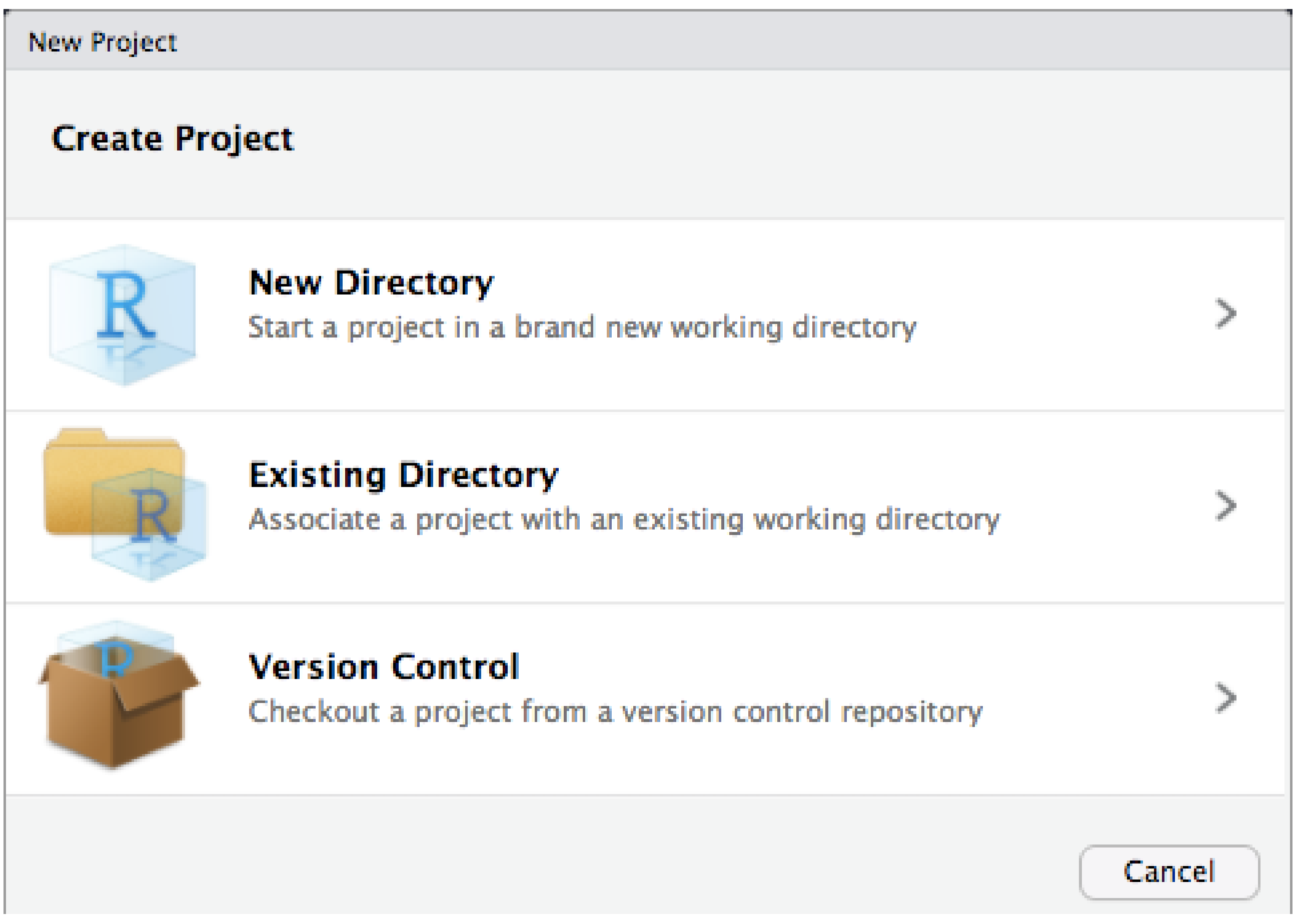
When you write code, be sure to comment your code liberally. In R, any line or any phrase starting with # is considered a comment and is ignored by the program. This allows you to comment your code, explain your ideas to yourself and generally make your code more readable. To further this goal, write your code in differnt lines, and indent, to make it more readable; R ignores white space in your file.

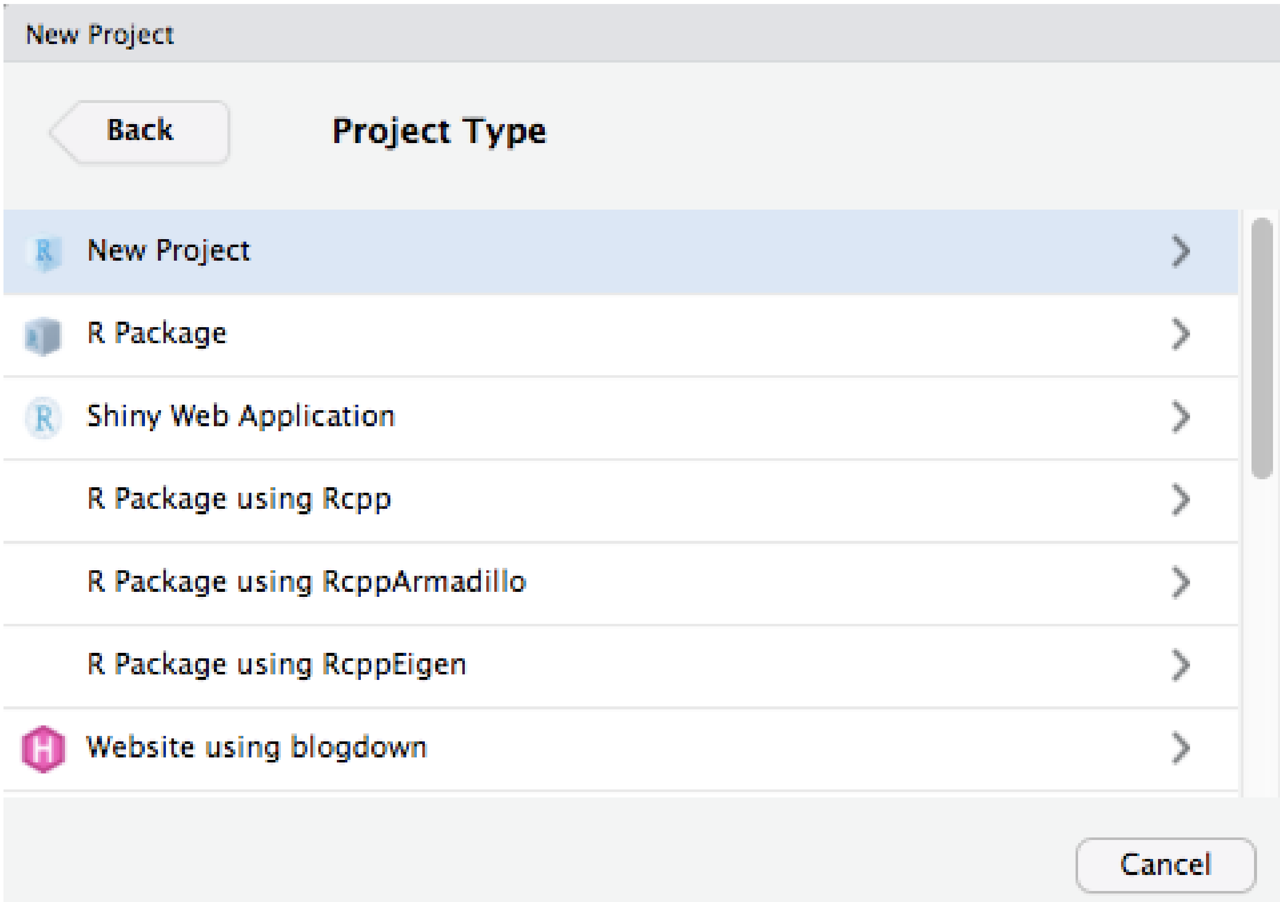
Why bother doing this? Basically because the most likely next person to see your code is going to be you in 6 months, and you don't want to be scratching your head wondering what you were doing earlier (been there, done that, don't like it). You certainly can't phone your earlier self, so the best strategy is to write comments for your future self to minimize future grief.

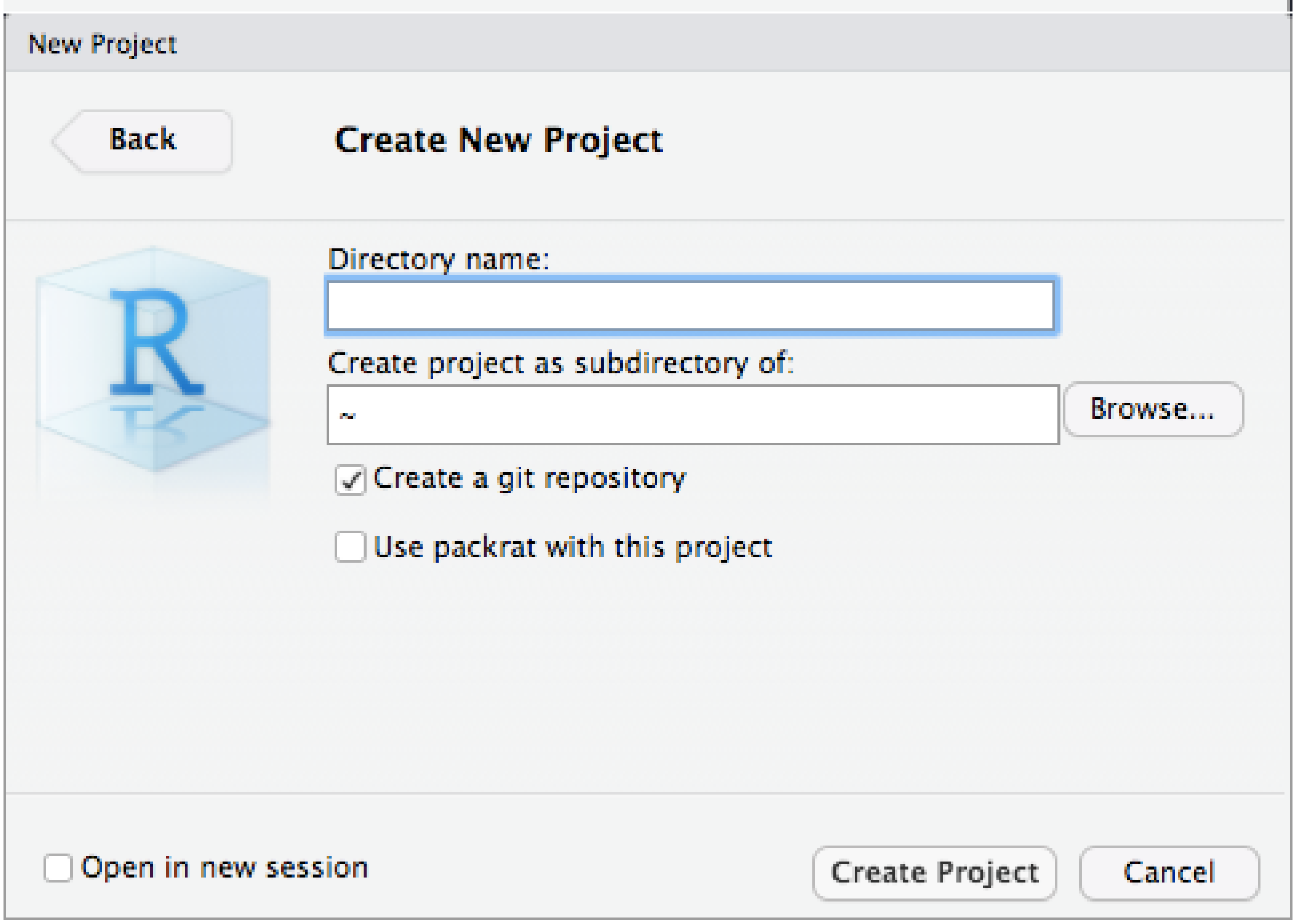
2.2 RStudio Projects

Projects are a nice way to organize your data projects within RStudio. Projects keep together input data, R scripts, analytical results, and figures, and keeps different projects separate. Each project can run on its own independent R session (no worries about cross-hybridization), and you can have several projects open concurrently without risk of cross-pollinating them.

To make a new project, click `File > New Project`, then:




New Project
Back
Project Type
New Project
R Package
Shiny Web Application
R Package using Rcpp
R Package using RcppArmadillo
R Package using RcppEigen
Website using blogdown
Cancel


New Project
Back
Create New Project
Directory name:
Create project as subdirectory of:
~
Browse...
Create a git repository
Use packrat with this project
Open in new session
Create Project
Cancel

Note that I always create a git repository for version control. If you're not doing that, untick the box in the window above.

Chapter 3

Loading data into R

R can access data files from a wide variety of sources. These include

1. Text files (csv, tsv, fixed-width)
2. Microsoft Excel files
3. Microsoft Access databases
4. SQL-based databases (MySql, Postgresql, SQLite, Amazon Redshift)
5. Enterprise databases (SAP, Oracle)

The R package `rio` can help read and write to many file types that are single files, and the package `rodbc` can do the same for the databases.

Exercise: Install the R package `rio` into your R installation

```
install.packages("rio", repos = "https://cran.rstudio.com") # Note the quotes
```

The `rio` package has a common way of reading data (using the `import` function). Importing the data will create an object called a data.frame, but if you just import data, it is not saved since it doesn't yet have a name.

```
library(rio) # activate the package
import('data/HR_Data.csv') # can use single or double quotes
```

So every time you import data, you have to name it. You do this using the `<-` operator.

```
hr_data <- import('data/HR_Data.csv')
```

Now, if you type `hr_data` in the console, you will see the data you imported.

```
head(hr_data) # This just displays the first 10 lines of the data
```

```
##                                                  Bureau Gender Grade
## 1    Comptroller and Global Financial Services (CGFS) female   N/A
## 2                East Asian and Pacific Affairs (EAP) female   N/A
## 3                 Overseas Buildings Operations (OBO)   male  FS-5
## 4         Conflict and Stabilization Operations (CSO)   male   N/A
## 5                               Consular Affairs (CA) female  FS-5
## 6 Management Policy, Rightsizing and Innovation (PRI) female  FS-2
```

```
##               Name
## 1  Katrina Lilly
## 2          Keene
## 3 Garrett Murphy
## 4     Jim Rhodes
## 5    Anita Myers
## 6 Vivian Einhorn
##
## 1                                  Hydrology, Research, Design, human resources, Ac
## 2                                                                          Sharepoi
## 3               interagency, Portuguese, Management, Foreign Policy, Economics, Hum
## 4                  education, seo, German, Finance, design, portuguese, disease res
## 5 Healthcare, training, German, french, Sharepoint, Marketing, Data Analysis, Econom
## 6                data analysis, Web Development, Hydrology, IT, SEO, Disease Respor
##   YearsService
## 1           16
## 2           21
## 3            5
## 4            4
## 5           23
## 6           19
```

Seeing the data like this is certainly a bit awkward, especially for large datasets. In RStudio, you can see the data somewhat like a spreadsheet with the following command:

```
View(hr_data)
```

This results in a new pane in RStudio.

README.md _bookdown.yml _output.yml index.Rmd 01-intro.Rmd 02-DataIngestion.Rmd* hr_data

Filter

	Bureau	Gender	Grade	Name	Skills	YearsService
1	Comptroller and Global Financial Services (CGFS)	female	N/A	Katrina Lilly	Hydrology, Research, Design, human resources, Admi...	16
2	East Asian and Pacific Affairs (EAP)	female	N/A	Keene	Sharepoint, Planning	21
3	Overseas Buildings Operations (OBO)	male	FS-5	Garrett Murphy	interagency, Portuguese, Management, Foreign Policy,...	5
4	Conflict and Stabilization Operations (CSO)	male	N/A	Jim Rhodes	education, seo, German, Finance, design, portuguese,...	4
5	Consular Affairs (CA)	female	FS-5	Anita Myers	Healthcare, training, German, french, Sharepoint, Mar...	23
6	Management Policy, Rightsizing and Innovation (PRI)	female	FS-2	Vivian Einhorn	data analysis, Web Development, Hydrology, IT, SEO, ...	19
7	Management Policy, Rightsizing and Innovation (PRI)	male	FS1	Buddy Vescio	Editing, Powerpoint, Communications, data analysis, ...	6
8	Global Youth Issues (GYI)	female	GS-10	Mccormick	marketing, Data Analysis, Network Security, Tourism, ...	2
9	Executive Secretariat (S/ES)	female	FS-1	Claudia Tremblay	Graphics, Marketing, climatology, Data Analysis	21
10	Population, Refugees, and Migration (PRM)	male	GS-8	Alba	Spanish, economics, planning, Graphics, Data Analysi...	16
11	Management Policy, Rightsizing and Innovation (PRI)	male	FS 6	Bebout	Japanese, Sharepoint, Editing, Outlook, interagency, H...	15
12	Inspector General (OIG)	male	FS-6	Grant Nightingale	Operations, Government, graphics, Human Resources...	10
13	Allowances (A/OPR/ALS)	female	FS-3	Margarita Casteel		24
14	Counterterrorism (CT)	female	FS-3	Mary Fegueroa	Japanese, Interagency, Powerpoint, Hydrology, Planni...	15
15	Executive Secretariat (S/ES)	male	FS-8	Francis Vereen	Interagency, Government, disease response, Research...	18
16	Executive Secretariat (S/ES)	male	GS-10	Glen Davis	editing	5
17	Consular Affairs (CA)	female	FS-7	Murray	interagency, Cloud Computing, Email, Portuguese, Go...	20
18	Executive Secretariat (S/ES)	male	N/A	John Small	portuguese, IT, Sharepoint, German, healthcare, engl...	3
19	Protocol (S/CPR)	female	FS-5	Sarah Castillo	outlook, Climatology, French, Operations	11
20	Global AIDS Coordinator (S/GAC)	female	N/A	Myrna Chew	Education, Spanish, Tourism, German, Administration...	13
21	Counterterrorism (CT)	male	FS-1	Gregory Baker	Engineering, Email, disaster response, cloud computi...	25
22	Democracy, Human Rights, and Labor (DRL)	male	GS10	Walker	Social Media	12
23	Director General of the Foreign Service and Director o...	female	FS-7	Alison Ungerecht	Graphics, Design, sharepoint, Cloud Computing, Editi...	10

Showing 1 to 24 of 10,000 entries

3.1 Finer control of CSV imports

We can provide finer control over importing text files using additional options ("adverbs") to the `import` function ("verb"). For example, it might be good to check if all the column names are unique, and to make them not have spaces (which are awkward in terms of typing and functionality). You can add the option `check.names = TRUE` to the command:

```
hr_data <- import('data/HR_Data.csv', check.names = TRUE)
```

Similarly, if you're using European data, where the decimal point is denoted by a comma, you can add the following option:

```
hr_data <- import('data/HR_Data.csv', check.names = TRUE, dec = ',')
```

You can see most of the options in the help file for import, which you can access either from the `Help` pane, or by typing `?import` or `help(import)` in the console

3.2 Finer control of Excel imports

You can specify sheet names or sheet positions for import from an Excel file. If you know the sheet name, you can specify it using the `which` option:

```
dos_data <- import('data/simulatedDOS.xlsx', which='Staffing_by_Bureau')
```

You can also grab the same sheet by position:

```
dos_data <- import('data/simulatedDOS.xlsx', which = 2)
```

We'll talk about how to grab multiple sheets together into a list in the data munging section4.

3.3 Importing data from databases

If you have data in an Access database, you can read it in pretty easily using the RODBC package. To import one particular table from Access, you can use

```
library(RODBC) # activate package, case-sensitive
channel <- odbcConnectAccess('C:/Documents/Name_of_Access_Database') # change to your
mydata <- sqlQuery(channel, paste("select * from Name_of_table_in_database"))
```

For other databases, the connection can be made using the odbc package. You can connnect to a MySQL database, for example, using

```
library(odbc)
con <- dbConnect(odbc(),
                 Driver   = "[your driver's name]",
                 Server   = "[your server's path]",
                 Database = "[your database's name]",
                 UID      = rstudioapi::askForPassword("Database user"),
                 PWD      = rstudioapi::askForPassword("Database password"),
                 Port     = 1433)
```

and you can load a table into R using

```
dat <- dbGetQuery(con, 'select * from <table name>')
```

You'll notice that it is a bit more complicated to call data from databases, though once it's set up, it works beautifully. For more details about this process for different databases, see RStudio's tutorial.

Chapter 4

Data Munging

Data munging refers to the work of transforming data to make it usable for a computer. Data unfortunately comes in all shapes and sizes, with all sorts of issues, so this process can take a while. Often a rule of thumb is that making a data set ready for analysis takes about 80% of the time of a project.

4.1 Tidy data

There is a principle of making data "tidy", promoted by Dr. Hadley Wickham. This tidying of data makes computer programs happy, since these data can be most easily digested. A dataset can be messy or tidy depending on how the rows, columns and tables you're using align with observations, variables and types.

The properties of a tidy dataset are:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table.

This forms a standardized way to structure a dataset, and so makes it easy for the analyst to develop standard pipelines.

A dataset can be messy in many many ways. Many of the more common issues are listed below:

- Column names contain values, not just variable names
- Multiple variables are stored in one column
- Variables are stored in both rows and columns
- Multiple types of observational types are stored in the same table
- A single observational unit is stored in multiple tables

Sometimes the messier format is better for data entry, but bad for data analyses.

We'll show a few examples here, but a more detailed discussion is available online.

The workhorse for this tidying activity is the `tidyr` package, part of the `tidyverse` meta-package. We'll tend to start every analysis by loading the `tidyverse` package, so we are covered.

4.1.1 Variable in column names

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------
## v ggplot2 3.1.0          v purrr   0.3.2
## v tibble  2.0.1          v dplyr   0.8.0.9009
## v tidyr   0.8.3          v stringr 1.4.0
## v readr   1.3.1          v forcats 0.4.0
## Warning: package 'tibble' was built under R version 3.5.2
## Warning: package 'tidyr' was built under R version 3.5.2
## Warning: package 'stringr' was built under R version 3.5.2
## -- Conflicts ------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
pew <- import('data/pew.csv')
head(pew)
```

```
##              religion <$10k $10-20k $20-30k $30-40k $40-50k $50-75k
## 1            Agnostic    27      34      60      81      76     137
## 2             Atheist    12      27      37      52      35      70
## 3            Buddhist    27      21      30      34      33      58
## 4            Catholic   418     617     732     670     638    1116
## 5 Don’t know/refused    15      14      15      11      10      35
## 6   Evangelical Prot   575     869    1064     982     881    1486
##   $75-100k $100-150k >150k Don't know/refused
## 1      122       109    84                 96
## 2       73        59    74                 76
## 3       62        39    53                 54
## 4      949       792   633               1489
## 5       21        17    18                116
## 6      949       723   414               1529
```

This dataset has actual data in the column headers, rather than variable names. This information needs to be captured into a column. We should ideally have 3 columns in this dataset: religion, income and frequency. We can achieve this using a function called `gather` which takes a wide dataset and makes it tall. We will do this by forming a pipeline (think of this as a sentence), starting with the dataset.

```
pew %>%
  gather(income, frequency, -religion)
```

```
##                      religion            income frequency
## 1                    Agnostic             <$10k        27
## 2                     Atheist             <$10k        12
## 3                    Buddhist             <$10k        27
```

```
## 4                  Catholic              <$10k     418
## 5        Don’t know/refused              <$10k      15
## 6          Evangelical Prot              <$10k     575
## 7                     Hindu              <$10k       1
## 8   Historically Black Prot              <$10k     228
## 9         Jehovah's Witness              <$10k      20
## 10                   Jewish              <$10k      19
## 11            Mainline Prot              <$10k     289
## 12                   Mormon              <$10k      29
## 13                   Muslim              <$10k       6
## 14                 Orthodox              <$10k      13
## 15          Other Christian              <$10k       9
## 16             Other Faiths              <$10k      20
## 17    Other World Religions              <$10k       5
## 18             Unaffiliated              <$10k     217
## 19                 Agnostic            $10-20k      34
## 20                  Atheist            $10-20k      27
## 21                 Buddhist            $10-20k      21
## 22                 Catholic            $10-20k     617
## 23       Don’t know/refused            $10-20k      14
## 24         Evangelical Prot            $10-20k     869
## 25                    Hindu            $10-20k       9
## 26  Historically Black Prot            $10-20k     244
## 27        Jehovah's Witness            $10-20k      27
## 28                   Jewish            $10-20k      19
## 29            Mainline Prot            $10-20k     495
## 30                   Mormon            $10-20k      40
## 31                   Muslim            $10-20k       7
## 32                 Orthodox            $10-20k      17
## 33          Other Christian            $10-20k       7
## 34             Other Faiths            $10-20k      33
## 35    Other World Religions            $10-20k       2
## 36             Unaffiliated            $10-20k     299
## 37                 Agnostic            $20-30k      60
## 38                  Atheist            $20-30k      37
## 39                 Buddhist            $20-30k      30
## 40                 Catholic            $20-30k     732
## 41       Don’t know/refused            $20-30k      15
## 42         Evangelical Prot            $20-30k    1064
## 43                    Hindu            $20-30k       7
## 44  Historically Black Prot            $20-30k     236
## 45        Jehovah's Witness            $20-30k      24
## 46                   Jewish            $20-30k      25
## 47            Mainline Prot            $20-30k     619
## 48                   Mormon            $20-30k      48
## 49                   Muslim            $20-30k       9
## 50                 Orthodox            $20-30k      23
```

```
## 51         Other Christian           $20-30k        11
## 52            Other Faiths           $20-30k        40
## 53   Other World Religions           $20-30k         3
## 54            Unaffiliated           $20-30k       374
## 55                Agnostic           $30-40k        81
## 56                 Atheist           $30-40k        52
## 57                Buddhist           $30-40k        34
## 58                Catholic           $30-40k       670
## 59      Don’t know/refused           $30-40k        11
## 60        Evangelical Prot           $30-40k       982
## 61                   Hindu           $30-40k         9
## 62 Historically Black Prot           $30-40k       238
## 63       Jehovah's Witness           $30-40k        24
## 64                  Jewish           $30-40k        25
## 65           Mainline Prot           $30-40k       655
## 66                  Mormon           $30-40k        51
## 67                  Muslim           $30-40k        10
## 68                Orthodox           $30-40k        32
## 69         Other Christian           $30-40k        13
## 70            Other Faiths           $30-40k        46
## 71   Other World Religions           $30-40k         4
## 72            Unaffiliated           $30-40k       365
## 73                Agnostic           $40-50k        76
## 74                 Atheist           $40-50k        35
## 75                Buddhist           $40-50k        33
## 76                Catholic           $40-50k       638
## 77      Don’t know/refused           $40-50k        10
## 78        Evangelical Prot           $40-50k       881
## 79                   Hindu           $40-50k        11
## 80 Historically Black Prot           $40-50k       197
## 81       Jehovah's Witness           $40-50k        21
## 82                  Jewish           $40-50k        30
## 83           Mainline Prot           $40-50k       651
## 84                  Mormon           $40-50k        56
## 85                  Muslim           $40-50k         9
## 86                Orthodox           $40-50k        32
## 87         Other Christian           $40-50k        13
## 88            Other Faiths           $40-50k        49
## 89   Other World Religions           $40-50k         2
## 90            Unaffiliated           $40-50k       341
## 91                Agnostic           $50-75k       137
## 92                 Atheist           $50-75k        70
## 93                Buddhist           $50-75k        58
## 94                Catholic           $50-75k      1116
## 95      Don’t know/refused           $50-75k        35
## 96        Evangelical Prot           $50-75k      1486
## 97                   Hindu           $50-75k        34
```

```
## 98  Historically Black Prot          $50-75k       223
## 99        Jehovah's Witness          $50-75k        30
## 100                  Jewish          $50-75k        95
## 101            Mainline Prot          $50-75k      1107
## 102                  Mormon          $50-75k       112
## 103                  Muslim          $50-75k        23
## 104                Orthodox          $50-75k        47
## 105         Other Christian          $50-75k        14
## 106            Other Faiths          $50-75k        63
## 107   Other World Religions          $50-75k         7
## 108            Unaffiliated          $50-75k       528
## 109                Agnostic         $75-100k       122
## 110                 Atheist         $75-100k        73
## 111                Buddhist         $75-100k        62
## 112                Catholic         $75-100k       949
## 113      Don’t know/refused         $75-100k        21
## 114        Evangelical Prot         $75-100k       949
## 115                   Hindu         $75-100k        47
## 116 Historically Black Prot         $75-100k       131
## 117       Jehovah's Witness         $75-100k        15
## 118                  Jewish         $75-100k        69
## 119            Mainline Prot         $75-100k       939
## 120                  Mormon         $75-100k        85
## 121                  Muslim         $75-100k        16
## 122                Orthodox         $75-100k        38
## 123         Other Christian         $75-100k        18
## 124            Other Faiths         $75-100k        46
## 125   Other World Religions         $75-100k         3
## 126            Unaffiliated         $75-100k       407
## 127                Agnostic        $100-150k       109
## 128                 Atheist        $100-150k        59
## 129                Buddhist        $100-150k        39
## 130                Catholic        $100-150k       792
## 131      Don’t know/refused        $100-150k        17
## 132        Evangelical Prot        $100-150k       723
## 133                   Hindu        $100-150k        48
## 134 Historically Black Prot        $100-150k        81
## 135       Jehovah's Witness        $100-150k        11
## 136                  Jewish        $100-150k        87
## 137            Mainline Prot        $100-150k       753
## 138                  Mormon        $100-150k        49
## 139                  Muslim        $100-150k         8
## 140                Orthodox        $100-150k        42
## 141         Other Christian        $100-150k        14
## 142            Other Faiths        $100-150k        40
## 143   Other World Religions        $100-150k         4
## 144            Unaffiliated        $100-150k       321
```

```
## 145                Agnostic              >150k       84
## 146                 Atheist              >150k       74
## 147                Buddhist              >150k       53
## 148                Catholic              >150k      633
## 149      Don’t know/refused              >150k       18
## 150        Evangelical Prot              >150k      414
## 151                   Hindu              >150k       54
## 152 Historically Black Prot              >150k       78
## 153       Jehovah's Witness              >150k        6
## 154                  Jewish              >150k      151
## 155           Mainline Prot              >150k      634
## 156                  Mormon              >150k       42
## 157                  Muslim              >150k        6
## 158                Orthodox              >150k       46
## 159         Other Christian              >150k       12
## 160            Other Faiths              >150k       41
## 161   Other World Religions              >150k        4
## 162            Unaffiliated              >150k      258
## 163                Agnostic Don't know/refused       96
## 164                 Atheist Don't know/refused       76
## 165                Buddhist Don't know/refused       54
## 166                Catholic Don't know/refused     1489
## 167      Don’t know/refused Don't know/refused      116
## 168        Evangelical Prot Don't know/refused     1529
## 169                   Hindu Don't know/refused       37
## 170 Historically Black Prot Don't know/refused      339
## 171       Jehovah's Witness Don't know/refused       37
## 172                  Jewish Don't know/refused      162
## 173           Mainline Prot Don't know/refused     1328
## 174                  Mormon Don't know/refused       69
## 175                  Muslim Don't know/refused       22
## 176                Orthodox Don't know/refused       73
## 177         Other Christian Don't know/refused       18
## 178            Other Faiths Don't know/refused       71
## 179   Other World Religions Don't know/refused        8
## 180            Unaffiliated Don't know/refused      597
```

Let’s parse this out. First we see this new operator `%>%`, which you can think of as the word “then”. So we start with the dataset pew, “then” we gather its columns into two columns, `income` and `frequency`. We don’t want the `religion` column to be part of this operation, so we “minus” it out, which says, don’t do this `gather` operation on the `religion` column, but use everything else. The `religion` column gets repeated as needed.

> The `%>%` operator can be easily typed in RStudio using the shortcut Ctrl-Shift-M (Cmd-Shift-M on a Mac)

This is now a tidy dataset, since each column is a single variable, each row is a single observation

4.1.2 Multiple variables in column names

```
tb <- import('data/tb.csv') %>% as_tibble()
head(tb)
```

```
## # A tibble: 6 x 22
##   iso2   year   m04  m514  m014 m1524 m2534 m3544 m4554 m5564   m65    mu
##   <chr> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1 AD     1989    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 2 AD     1990    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 3 AD     1991    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 4 AD     1992    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 5 AD     1993    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 6 AD     1994    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## # ... with 10 more variables: f04 <int>, f514 <int>, f014 <int>,
## #   f1524 <int>, f2534 <int>, f3544 <int>, f4554 <int>, f5564 <int>,
## #   f65 <int>, fu <int>
```

Notice that the column names contain both sex and age group data. First we'll `gather` the sex/age columns, as before. Note that there are many missing values in this dataset. These are denoted in R by `NA`.

```
tb %>%
  gather(sex_age, n, -iso2, -year, -fu)
```

```
## # A tibble: 109,611 x 5
##    iso2   year    fu sex_age     n
##    <chr> <int> <int> <chr>   <int>
##  1 AD     1989    NA m04        NA
##  2 AD     1990    NA m04        NA
##  3 AD     1991    NA m04        NA
##  4 AD     1992    NA m04        NA
##  5 AD     1993    NA m04        NA
##  6 AD     1994    NA m04        NA
##  7 AD     1996    NA m04        NA
##  8 AD     1997    NA m04        NA
##  9 AD     1998    NA m04        NA
## 10 AD     1999    NA m04        NA
## # ... with 109,601 more rows
```

Since there are a lot of missing values here, we can drop them in the above step by adding an option.

```
tb %>% gather(sex_age, n, -iso2, -year, -fu, na.rm=T)
```

```
## # A tibble: 35,478 x 5
##    iso2   year    fu sex_age     n
##    <chr> <int> <int> <chr>   <int>
##  1 AD     2005     0 m04         0
##  2 AD     2006     0 m04         0
##  3 AD     2008     0 m04         0
```

```
## 4 AE 2006 NA m04 0
## 5 AE 2007 NA m04 0
## 6 AE 2008 0 m04 0
## 7 AG 2007 NA m04 0
## 8 AL 2005 0 m04 0
## 9 AL 2006 0 m04 1
## 10 AL 2007 0 m04 0
## # ... with 35,468 more rows
```

We can now use the function `separate` to separate the data in the `sex_age` column into sex and age. In this case we have have the data in a fixed width format (the 1st element is the sex data), so we can use that:

```
tb %>%
  gather(sex_age, n, -iso2, -year, -fu, na.rm=T) %>%
  separate(sex_age, c("sex","age"), sep=1)
```

```
## # A tibble: 35,478 x 6
##    iso2   year    fu sex   age       n
##    <chr> <int> <int> <chr> <chr> <int>
##  1 AD     2005     0 m     04        0
##  2 AD     2006     0 m     04        0
##  3 AD     2008     0 m     04        0
##  4 AE     2006    NA m     04        0
##  5 AE     2007    NA m     04        0
##  6 AE     2008     0 m     04        0
##  7 AG     2007    NA m     04        0
##  8 AL     2005     0 m     04        0
##  9 AL     2006     0 m     04        1
## 10 AL     2007     0 m     04        0
## # ... with 35,468 more rows
```

If the data was separated by a symbol, like _, we would use `sep = "_"` instead.

4.1.3 Variables stored in rows and columns

```
weather <- import('data/weather.csv') %>% as_tibble()
weather
```

```
## # A tibble: 22 x 35
##    id     year month element    d1    d2    d3    d4    d5    d6    d7
##    <chr> <int> <int> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 MX17~  2010     1 tmax       NA  NA    NA       NA  NA       NA    NA
##  2 MX17~  2010     1 tmin       NA  NA    NA       NA  NA       NA    NA
##  3 MX17~  2010     2 tmax       NA  27.3  24.1     NA  NA       NA    NA
##  4 MX17~  2010     2 tmin       NA  14.4  14.4     NA  NA       NA    NA
##  5 MX17~  2010     3 tmax       NA  NA    NA       NA  32.1     NA    NA
##  6 MX17~  2010     3 tmin       NA  NA    NA       NA  14.2     NA    NA
##  7 MX17~  2010     4 tmax       NA  NA    NA       NA  NA       NA    NA
```

```
##  8 MX17~  2010     4 tmin        NA  NA     NA      NA  NA      NA    NA
##  9 MX17~  2010     5 tmax        NA  NA     NA      NA  NA      NA    NA
## 10 MX17~  2010     5 tmin        NA  NA     NA      NA  NA      NA    NA
## # ... with 12 more rows, and 24 more variables: d8 <dbl>, d9 <lgl>,
## #   d10 <dbl>, d11 <dbl>, d12 <lgl>, d13 <dbl>, d14 <dbl>, d15 <dbl>,
## #   d16 <dbl>, d17 <dbl>, d18 <lgl>, d19 <lgl>, d20 <lgl>, d21 <lgl>,
## #   d22 <lgl>, d23 <dbl>, d24 <lgl>, d25 <dbl>, d26 <dbl>, d27 <dbl>,
## #   d28 <dbl>, d29 <dbl>, d30 <dbl>, d31 <dbl>
```

Here, for each year and month, the data for each day of the month is stored in columns. For each day, two values are noted – the max (`tmax`) and min (`tmin`) temperature that day, stored as rows.

We start by gathering the extra columns as before:

```
weather %>%
  gather(day, temp, d1:d31)
```

```
## # A tibble: 682 x 6
##    id       year month element day    temp
##    <chr>   <int> <int> <chr>   <chr> <dbl>
##  1 MX17004  2010     1 tmax    d1       NA
##  2 MX17004  2010     1 tmin    d1       NA
##  3 MX17004  2010     2 tmax    d1       NA
##  4 MX17004  2010     2 tmin    d1       NA
##  5 MX17004  2010     3 tmax    d1       NA
##  6 MX17004  2010     3 tmin    d1       NA
##  7 MX17004  2010     4 tmax    d1       NA
##  8 MX17004  2010     4 tmin    d1       NA
##  9 MX17004  2010     5 tmax    d1       NA
## 10 MX17004  2010     5 tmin    d1       NA
## # ... with 672 more rows
```

> Here's a new notation – `d1:d31`. This means all columns starting at `d1` and ending at `d31`. This notation is originally from creating sequences of numbers. See what happens if you type `1:30` in the console.

Now, for each date, we have two *rows* of data. These need to be two *columns* of data. So we need to do the reverse operation from `gather`. This is called `spread`.

```
weather %>%
  gather(date, temp, d1:d31) %>%
  spread(element, temp)
```

```
## # A tibble: 341 x 6
##    id       year month date   tmax  tmin
##    <chr>   <int> <int> <chr> <dbl> <dbl>
##  1 MX17004  2010     1 d1       NA    NA
##  2 MX17004  2010     1 d10      NA    NA
##  3 MX17004  2010     1 d11      NA    NA
##  4 MX17004  2010     1 d12      NA    NA
##  5 MX17004  2010     1 d13      NA    NA
```

```
##  6 MX17004  2010     1 d14      NA    NA
##  7 MX17004  2010     1 d15      NA    NA
##  8 MX17004  2010     1 d16      NA    NA
##  9 MX17004  2010     1 d17      NA    NA
## 10 MX17004  2010     1 d18      NA    NA
## # ... with 331 more rows
```

We tell `spread` which column should form column names and which should provide the data for the columns.

4.2 Data cleaning

The weather data set shows that we still need to do a bit more cleaning to this data to make it workable. Mainly, we need to fix the `dat` column to make it numeric. Note the odd ordering, where `d1` is followed by `d10`. This is an *alphabetical* ordering rather than a *numeric* ordering. We'll now add to our pipeline (sentence) to make this happen:

```
weather %>%
  gather(date, temp, d1:d31) %>%
  spread(element, temp) %>%
  mutate(date = parse_number(date))
```

```
## # A tibble: 341 x 6
##    id       year month  date  tmax  tmin
##    <chr>   <int> <int> <dbl> <dbl> <dbl>
##  1 MX17004  2010     1     1    NA    NA
##  2 MX17004  2010     1    10    NA    NA
##  3 MX17004  2010     1    11    NA    NA
##  4 MX17004  2010     1    12    NA    NA
##  5 MX17004  2010     1    13    NA    NA
##  6 MX17004  2010     1    14    NA    NA
##  7 MX17004  2010     1    15    NA    NA
##  8 MX17004  2010     1    16    NA    NA
##  9 MX17004  2010     1    17    NA    NA
## 10 MX17004  2010     1    18    NA    NA
## # ... with 331 more rows
```

Here we introduce another "verb", `mutate`. This function changes a column, either in-place as we did here, or by creating a new variable.

The data is still not quite in the right format, since the date column is in a weird order. We can add another verb to this pipe to fix that: `arrange`.

```
weather %>%
  gather(date, temp, d1:d31) %>%
  spread(element, temp) %>%
  mutate(date = parse_number(date)) %>%
  arrange(date)
```

```
## # A tibble: 341 x 6
```

```
##    id       year month  date  tmax  tmin
##    <chr>   <int> <int> <dbl> <dbl> <dbl>
##  1 MX17004  2010     1     1    NA    NA
##  2 MX17004  2010     2     1    NA    NA
##  3 MX17004  2010     3     1    NA    NA
##  4 MX17004  2010     4     1    NA    NA
##  5 MX17004  2010     5     1    NA    NA
##  6 MX17004  2010     6     1    NA    NA
##  7 MX17004  2010     7     1    NA    NA
##  8 MX17004  2010     8     1    NA    NA
##  9 MX17004  2010    10     1    NA    NA
## 10 MX17004  2010    11     1    NA    NA
## # ... with 331 more rows
```

Not quite, right? We're not used to seeing all the 1st of the months together, and so forth. We want all the daes for month 1, then all the dates for month two, and so on. This can be done by modifying the `arrange` command, by sorting first by month and then by date (essentially within month).

```
weather %>%
  gather(date, temp, d1:d31) %>%
  spread(element, temp) %>%
  mutate(date = parse_number(date)) %>%
  arrange(month, date)
```

```
## # A tibble: 341 x 6
##    id       year month  date  tmax  tmin
##    <chr>   <int> <int> <dbl> <dbl> <dbl>
##  1 MX17004  2010     1     1    NA    NA
##  2 MX17004  2010     1     2    NA    NA
##  3 MX17004  2010     1     3    NA    NA
##  4 MX17004  2010     1     4    NA    NA
##  5 MX17004  2010     1     5    NA    NA
##  6 MX17004  2010     1     6    NA    NA
##  7 MX17004  2010     1     7    NA    NA
##  8 MX17004  2010     1     8    NA    NA
##  9 MX17004  2010     1     9    NA    NA
## 10 MX17004  2010     1    10    NA    NA
## # ... with 331 more rows
```

Finally, if we want to save this, we need to assign this final product a name.

```
weather2 <- weather %>%
  gather(date, temp, d1:d31) %>%
  spread(element, temp) %>%
  mutate(date = parse_number(date)) %>%
  arrange(month, date)
```

Exercise

The file `data/mbta.xlsx` contains monthly data on number of commuter trips by different modalities on the MBTA system n Boston. It is in a messy format. It also has an additional quirk in that it has a title on the first line that isn't even data. You can avoid loading that in by using the option `skip=1` (i.e. skip the first line) when you import. Work through this process to clean this dataset into tidy form. I'll also note that you can "minus" columns by position as well as name, so `gather(date, avg_trips, -1, -mode)` is valid to not involve the first column and the `mode` column.

4.3 Cleaning up data types and values

After you have tidied your data, lets call that tidy dataset `mbta2`.

```
mbta2
```

```
## # A tibble: 638 x 5
##       ..1 mode                year  month avg_trips
##     <dbl> <chr>               <chr> <chr> <chr>
##  1      1 All Modes by Qtr    2007  01    NA
##  2      2 Boat                2007  01    4
##  3      3 Bus                 2007  01    335.819
##  4      4 Commuter Rail       2007  01    142.2
##  5      5 Heavy Rail          2007  01    435.294
##  6      6 Light Rail          2007  01    227.231
##  7      7 Pct Chg / Yr        2007  01    0.02
##  8      8 Private Bus         2007  01    4.772
##  9      9 RIDE                2007  01    4.9
## 10     10 Trackless Trolley 2007  01    12.757
## # ... with 628 more rows
```

We see that there's still some issues. If you look at the top of the dataset, you'll see that year, month and avg_trips are all *character* variables and not *numeric* variables. (You can see this if you converted to a tibble using `as_tibble`. Otherwise, type `str(mbta2)` at the console). Also, there is this odd column with the name `..1` that is just an index of rows. Lastly, the row marked `TOTAL` is not necessary since it's a derived row, and the `All Modes by Qtr` row is missing in many times, and appears inconsistent with `TOTAL`.

First, let's deal with the type issue.

```
mbta2 %>%
  mutate(
    year = parse_number(year),
    month = parse_number(month),
    avg_trips = parse_number(avg_trips)
  )
```

```
## # A tibble: 638 x 5
##       ..1 mode                 year month avg_trips
##     <dbl> <chr>               <dbl> <dbl>     <dbl>
```

```
##   1      1 All Modes by Qtr   2007     1     NA
##   2      2 Boat               2007     1      4
##   3      3 Bus                2007     1    336.
##   4      4 Commuter Rail      2007     1    142.
##   5      5 Heavy Rail         2007     1    435.
##   6      6 Light Rail         2007     1    227.
##   7      7 Pct Chg / Yr       2007     1      0.02
##   8      8 Private Bus        2007     1      4.77
##   9      9 RIDE               2007     1      4.9
##  10     10 Trackless Trolley  2007     1     12.8
## # ... with 628 more rows
```

A more advanced version of this operation would be

```
mbta2 %>%
  mutate_at(vars(year, month, avg_trips), parse_number)
```

Next we want to get rid of that first column. The verb we'll use here is `select`.

```
mbta2 %>%
  mutate(
    year = parse_number(year),
    month = parse_number(month),
    avg_trips = parse_number(avg_trips)
  ) %>%
  select(-1) # Get rid of 1st column
```

```
## # A tibble: 638 x 4
##    mode                year month avg_trips
##    <chr>              <dbl> <dbl>     <dbl>
##  1 All Modes by Qtr    2007     1     NA
##  2 Boat                2007     1      4
##  3 Bus                 2007     1    336.
##  4 Commuter Rail       2007     1    142.
##  5 Heavy Rail          2007     1    435.
##  6 Light Rail          2007     1    227.
##  7 Pct Chg / Yr        2007     1      0.02
##  8 Private Bus         2007     1      4.77
##  9 RIDE                2007     1      4.9
## 10 Trackless Trolley   2007     1     12.8
## # ... with 628 more rows
```

Lastly, we want to get rid of rows where mode equals TOTAL or "All Modes by Qtr". Our verb here is `filter`.

```
mbta3 <- mbta2 %>%
  mutate(
    year = parse_number(year),
    month = parse_number(month),
    avg_trips = parse_number(avg_trips)
  ) %>%
```

```
  select(-1) %>%
  filter(mode != 'TOTAL', mode != "All Modes by Qtr")
```

Note that the strings in quotes have to be exact matches to what you want to look for. The `!=` means "not equals".

We're assigning this to a new variable, mbta3, which is our clean dataset.

> In R, filtering refers to keeping or removing *rows* that meet some criterion; selecting refers to keeping or removing *columns*. The corresponding "verbs" to put into your pipeline are `filter` and `select`.

4.4 Other types of cleaning

There are different functions that you can apply to a dataset for different cleaning purposes. A selection are given below:

1. `distinct()` keeps the unique (non-duplicate) rows of a dataset. Usage: `dataset %>% distinct()`
2. If you want to keep only rows with complete data, you can invoke `drop_na`. Usage: `dataset %>% drop_na()`. You can modify `drop_na` by specifying variables from which you want to drop the missing values.
3. If you want to convert a value to missing (commonly 99 is used for missing data), then you can use `replace_na` within `mutate` to change to missing values on a column-by-column basis. Usage: `dataset %>% mutate(var1 = na_if(var1, 99))`

4.5 Cleaning from Excel files

Excel, being omnipresent, creates its own sets of difficulties. Excel, on top of being a data entry and analysis platform, is also a visual platform, so tables are often created to look good rather than be tidy. Things we commonly find in Excel files include colored rows and columns, multiple lines of headers, multiple rows with variables, typos resulting in numeric data becoming string data, and many others.

4.6 Cleaning from Excel files

Excel, being omnipresent, creates its own sets of difficulties. Excel, on top of being a data entry and analysis platform, is also a visual platform, so tables are often created to look good rather than be tidy. Things we commonly find in Excel files include colored rows and columns, multiple lines of headers, multiple rows with variables, typos resulting in numeric data becoming string data, and many others.

Let's start with a data set that's structured similarly to many government datasets around the world.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	**Table 43. Average daily attendance (ADA) as a percentage of total enrollment, school day length, and school year length in public schools, by school level and state: 2007-08**																
2		Total elementary, secondary, and combined elementary/secondary schools								Elementary schools				Secondary schools			
3	State	ADA as percent of enrollment		Average hours in school day		Average days in school year		Average hours in school year		ADA as percent of enrollment		Average hours in school day		ADA as percent of enrollment		Average hours in school day	
4	1	2		3		4		5		6		7		8		9	
5	**United States ....**	**93.1**	**(0.22)**	**6.6**	**(0.02)**	**180**	**(0.1)**	**1,193**	**(3.1)**	**94.0**	**(0.27)**	**6.7**	**(0.02)**	**91.1**	**(0.43)**	**6.6**	**(0.04)**
6	Alabama	93.8	(1.24)	7.0	(0.07)	180	(0.8)	1,267	(12.3)	93.8	(1.84)	7.0	(0.08)	94.6	(0.38)	7.1	(0.17)
7	Alaska	89.9	(1.22)	6.5	(0.05)	180	(3.4)	1,163	(22.9)	91.3	(1.56)	6.5	(0.05)	93.2	(1.57)	6.2	(0.15)
8	Arizona	89.0	(2.95)	6.4	(0.09)	181	(1.7)	1,159	(14.4)	88.9	(3.91)	6.4	(0.10)	89.0	(3.22)	6.4	(0.25)
9	Arkansas	91.8	(1.35)	6.9	(0.06)	179	(0.2)	1,229	(10.7)	92.1	(2.09)	6.9	(0.08)	90.8	(2.23)	6.8	(0.10)
10	California	93.2	(0.71)	6.2	(0.07)	181	(0.4)	1,129	(12.5)	94.9	(0.75)	6.3	(0.05)	89.4	(1.45)	6.1	(0.20)
11																	
12	Colorado	93.9	(0.44)	7.0	(0.05)	171	(1.0)	1,199	(9.9)	94.5	(0.45)	7.0	(0.07)	91.2	(1.28)	7.0	(0.11)
13	Connecticut	87.9	(2.98)	6.5	(0.09)	181	(0.1)	1,173	(15.9)	87.4	(3.98)	6.5	(0.11)	93.7	(0.68)	6.5	(0.09)
14	Delaware	89.8	(1.75)	6.7	(0.09)	181	(0.8)	1,208	(18.7)	89.4	(2.50)	6.8	(0.06)	‡	(†)	6.5	(0.23)
15	District of Columbia	91.2	(1.27)	6.9	(0.21)	181	(0.4)	1,256	(42.3)	93.9	(0.38)	6.9	(0.10)	‡	(†)	‡	(†)
16	Florida	92.7	(0.74)	6.4	(0.08)	184	(1.2)	1,184	(18.8)	94.0	(0.94)	6.5	(0.05)	89.9	(1.43)	6.3	(0.38)
17																	
18	Georgia	93.3	(1.28)	6.8	(0.06)	181	(1.0)	1,229	(12.4)	93.9	(1.60)	6.8	(0.06)	93.1	(1.23)	6.9	(0.09)
19	Hawaii	90.7	(4.58)	6.3	(0.10)	179	(1.6)	1,118	(12.7)	90.8	(2.63)	6.2	(0.05)	‡	(†)	‡	(†)
20	Idaho	92.4	(2.27)	6.6	(0.09)	173	(2.6)	1,143	(13.0)	92.0	(3.70)	6.6	(0.13)	92.5	(1.54)	6.7	(0.11)
21	Illinois	94.0	(0.71)	6.5	(0.05)	177	(0.3)	1,147	(9.6)	95.3	(0.50)	6.5	(0.06)	90.0	(2.45)	6.5	(0.13)
22	Indiana	95.7	(0.51)	6.8	(0.06)	180	(0.3)	1,222	(11.4)	95.9	(0.61)	6.7	(0.07)	95.0	(0.74)	7.0	(0.09)
23																	
24	Iowa	94.8	(0.65)	6.9	(0.09)	180	(0.2)	1,232	(16.3)	96.3	(0.46)	6.9	(0.07)	90.4	(2.33)	6.6	(0.30)
25	Kansas	95.4	(0.52)	7.0	(0.07)	178	(0.6)	1,240	(14.7)	95.9	(0.53)	7.0	(0.09)	94.6	(0.35)	7.0	(0.07)
26	Kentucky	93.1	(1.89)	6.7	(0.06)	180	(1.1)	1,202	(11.8)	94.0	(2.56)	6.7	(0.08)	91.5	(2.88)	6.7	(0.09)
27	Louisiana	90.3	(2.31)	7.1	(0.08)	178	(1.0)	1,263	(14.9)	91.5	(3.19)	7.0	(0.10)	88.0	(1.82)	7.3	(0.12)

Notice that there are two levels of headers on top, each spanning multiple columns. Data is in paired columns, with the first column being the statistic and the second column being its standard error. The first row is a title, which we don't need, and the 4th row is also not needed.

We'll first tidy-fy this dataset, and then we'll clean it up a bit. Let's first think about the structure we want to get to for tidy-fication. The actual data lies in the statistics and the standard errors, and each of the column groupings represents different variables, so they should be in columns.

You can try to load this data using `import`, but you'll find it's a big mess. There are two powerful packages (by the same group of developers), `tidyxl` and `unpivotr`, that are fantastic tools for "fixing" Excel files for analysis.

Let's start with `tidyxl`.

```r
library(tidyxl)
```

```
## Warning: package 'tidyxl' was built under R version 3.5.2
```

```r
dataset1 <- xlsx_cells('data/attendance.xlsx')
dataset1
```

```
## # A tibble: 1,173 x 21
##    sheet address   row   col is_blank data_type error logical numeric
##    <chr> <chr>   <int> <int> <lgl>    <chr>     <chr> <lgl>     <dbl>
##  1 Tabl~ A1          1     1 FALSE    character <NA>  NA           NA
##  2 Tabl~ B1          1     2 TRUE     blank     <NA>  NA           NA
##  3 Tabl~ C1          1     3 TRUE     blank     <NA>  NA           NA
```

```
##   4 Tabl~ D1          1      4 TRUE      blank      <NA>  NA             NA
##   5 Tabl~ E1          1      5 TRUE      blank      <NA>  NA             NA
##   6 Tabl~ F1          1      6 TRUE      blank      <NA>  NA             NA
##   7 Tabl~ G1          1      7 TRUE      blank      <NA>  NA             NA
##   8 Tabl~ H1          1      8 TRUE      blank      <NA>  NA             NA
##   9 Tabl~ I1          1      9 TRUE      blank      <NA>  NA             NA
##  10 Tabl~ J1          1     10 TRUE      blank      <NA>  NA             NA
## # ... with 1,163 more rows, and 12 more variables: date <dttm>,
## #   character <chr>, character_formatted <list>, formula <chr>,
## #   is_array <lgl>, formula_ref <chr>, formula_group <int>, comment <chr>,
## #   height <dbl>, width <dbl>, style_format <chr>, local_format_id <int>
```

Notice that this pulls in a lot of meta-data in a tidy form, including information about cell formatting. This will be really useful in many situations.

First, lets get rid of the rows we don't need.

```
dataset1 <- dataset1 %>% filter(row != 1, row != 4, row < 65)
```

Now we could manipulate this dataset using `tidyverse` tools, but `unpivotr` is much mor poweful. First, we are going to pull off the two headers. `unpivotr` does this using the function `behead` (suggestive?), with the first argument being the direction ('N', "S", 'E','W',etc) of the table that the header is present. We will also consider the first column, consisting of state names, as a header on the left.

```
library(unpivotr)
```

```
## Warning: package 'unpivotr' was built under R version 3.5.2
```

```
dataset1 %>%
  behead('N', tophead) %>%
  behead('N', head2) %>%
  behead('W', State) %>%
  select(row, col, data_type, numeric, tophead, head2, State)
```

```
## # A tibble: 960 x 7
##       row   col data_type  numeric tophead                head2     State
##     <int> <int> <chr>        <dbl> <chr>                  <chr>     <chr>
##  1      5     2 numeric    9.31e+1 Total elementary, se~ ADA as p~ "   Unit~
##  2      5     3 numeric    2.19e-1 <NA>                   <NA>      "   Unit~
##  3      5     4 numeric    6.64e+0 <NA>                   Average ~ "   Unit~
##  4      5     5 numeric    1.76e-2 <NA>                   <NA>      "   Unit~
##  5      5     6 numeric    1.80e+2 <NA>                   Average ~ "   Unit~
##  6      5     7 numeric    1.43e-1 <NA>                   <NA>      "   Unit~
##  7      5     8 numeric    1.19e+3 <NA>                   Average ~ "   Unit~
##  8      5     9 numeric    3.09e+0 <NA>                   <NA>      "   Unit~
##  9      5    10 numeric    9.40e+1 Elementary schools     ADA as p~ "   Unit~
## 10      5    11 numeric    2.69e-1 <NA>                   <NA>      "   Unit~
## # ... with 950 more rows
```

We need to separate the statistics and the standard errors from consecutive columns, and also make them headers.

```
dataset1 %>%
  behead('N', tophead) %>%
  behead('N', head2) %>%
  behead('W', State) %>%
  select(row, col, data_type, numeric, tophead, head2, State) %>%
  mutate(header = ifelse(col %% 2 == 0, 'stats','se'))
```

```
## # A tibble: 960 x 8
##      row   col data_type  numeric tophead            head2    State    header
##    <int> <int> <chr>        <dbl> <chr>              <chr>    <chr>    <chr>
##  1     5     2 numeric    9.31e+1 Total elementar~ ADA as ~ "   Uni~ stats
##  2     5     3 numeric    2.19e-1 <NA>               <NA>     "   Uni~ se
##  3     5     4 numeric    6.64e+0 <NA>               Average~ "   Uni~ stats
##  4     5     5 numeric    1.76e-2 <NA>               <NA>     "   Uni~ se
##  5     5     6 numeric    1.80e+2 <NA>               Average~ "   Uni~ stats
##  6     5     7 numeric    1.43e-1 <NA>               <NA>     "   Uni~ se
##  7     5     8 numeric    1.19e+3 <NA>               Average~ "   Uni~ stats
##  8     5     9 numeric    3.09e+0 <NA>               <NA>     "   Uni~ se
##  9     5    10 numeric    9.40e+1 Elementary scho~ ADA as ~ "   Uni~ stats
## 10     5    11 numeric    2.69e-1 <NA>               <NA>     "   Uni~ se
## # ... with 950 more rows
```

The %% operator computes the remainder if the left side is divided by the right side. So the criterion is asking which columns are even. The `ifelse` statement says, if the criterion is met, write "stats", otherwise write "se". This new variable is assigned to the dataset with the variable name "header".

Notice that we have actually tidy-fied this dataset, but there's missing data here, since the column headers span several rows visually but are only credited to the first column it covers. So we need to fill in the entries for the remaining rows with the corresponding entry from the earliest column. There is a function `fill` in `tidyr` that does this general trick, using a method called *last value carried forward*.

```
dataset1 %>%
  behead('N', tophead) %>%
  behead('N', head2) %>%
  behead('W', State) %>%
  select(row, col, data_type, numeric, tophead, head2, State) %>%
  fill(tophead) %>%
  fill(head2)
```

```
## # A tibble: 960 x 7
##      row   col data_type  numeric tophead                  head2      State
##    <int> <int> <chr>        <dbl> <chr>                    <chr>      <chr>
##  1     5     2 numeric    9.31e+1 Total elementary, se~ ADA as p~ "   Unit~
##  2     5     3 numeric    2.19e-1 Total elementary, se~ ADA as p~ "   Unit~
##  3     5     4 numeric    6.64e+0 Total elementary, se~ Average ~ "   Unit~
##  4     5     5 numeric    1.76e-2 Total elementary, se~ Average ~ "   Unit~
##  5     5     6 numeric    1.80e+2 Total elementary, se~ Average ~ "   Unit~
##  6     5     7 numeric    1.43e-1 Total elementary, se~ Average ~ "   Unit~
##  7     5     8 numeric    1.19e+3 Total elementary, se~ Average ~ "   Unit~
```

```
## 8     5     9 numeric    3.09e+0 Total elementary, se~ Average ~ "   Unit~
## 9     5    10 numeric    9.40e+1 Elementary schools    ADA as p~ "   Unit~
## 10    5    11 numeric    2.69e-1 Elementary schools    ADA as p~ "   Unit~
## # ... with 950 more rows
```

To make this really tidy, we need to make two columns titled `stats` and `se` from this. We've seen this using `spread`, but there is a slightly more robust method from `unpivotr` called `spatter` which is meant for this unique structure.

```
tidy_dataset <- dataset1 %>%
  behead('N', tophead) %>%
  behead('N', head2) %>%
  behead('W', State) %>%
  select(row, col, data_type, numeric, tophead, head2, State) %>%
  mutate(header = ifelse(col %% 2 == 0, 'stats','se')) %>%
  fill(tophead) %>%
  fill(head2) %>%
  select(row, numeric, tophead, head2, State, header) %>%
  spatter(header, numeric) %>%
  select(-row)
tidy_dataset
```

```
## # A tibble: 480 x 5
##    tophead                          head2          State              se  stats
##    <chr>                            <chr>          <chr>           <dbl>  <dbl>
##  1 Elementary schools               ADA as percen~ "    United ~ 0.269   9.40e1
##  2 Elementary schools               Average hours~ "    United ~ 0.0160 6.66e0
##  3 Secondary schools                ADA as percen~ "    United ~ 0.432   9.11e1
##  4 Secondary schools                Average hours~ "    United ~ 0.0403 6.59e0
##  5 Total elementary, secondary, ~ ADA as percen~ "    United ~ 0.219   9.31e1
##  6 Total elementary, secondary, ~ Average days ~ "    United ~ 0.143   1.80e2
##  7 Total elementary, secondary, ~ Average hours~ "    United ~ 0.0176 6.64e0
##  8 Total elementary, secondary, ~ Average hours~ "    United ~ 3.09    1.19e3
##  9 Elementary schools               ADA as percen~ Alabama ...~ 1.84    9.38e1
## 10 Elementary schools               Average hours~ Alabama ...~ 0.0759 7.04e0
## # ... with 470 more rows
```

We have to clean the `State` variable. We'll use the methods in the `stringr` package, which is already loaded with the `tidyverse`.

```
tidy_dataset <- tidy_dataset %>%
  mutate(State = str_remove(State, '\\.+')) %>%
  mutate(State = str_trim(State))
```

The first verb removes all the `.` in the variable, using something called a regular expression. This particular expression means that we want to look for sequences of dots, and remove them. The `\\` before the `.` tells R that we really mean `.`, since the `.` has a different meaning in regular expressions.

The second verb trims away blank spaces before and after each entry.

We'll hold on to this dataset for the visualization section. Just to be safe, let's save it.

```
saveRDS(tidy_dataset, file = 'data/attendance.rds')
```

This saves the data in an R-specific format that will allow us to load it quickly.

> The RDS format is an open standard and so it can be called from other programs if the appropriate programs are written.

Dealing with visual formating (colors)

The dataset we'll use for this has identifiable information, so I will not expose it publicly. It is available in your files as `data/classlist.xlsx`.

Since we're interested in background and font colors here, which are informative, we also need to load the format information into R.

```
library(tidyxl)
library(unpivotr)

dataset2 <- xlsx_cells('data/classlist.xlsx')
formats <- xlsx_formats('data/classlist.xlsx')

format_id <- dataset2$local_format_id
dataset2$font_color <- formats$local$font$color$rgb[format_id]
dataset2$bg_color <- formats$local$fill$patternFill$fgColor$rgb[format_id]
```

```
unique(dataset2$font_color)
```

```
## [1] "FF000000" "FF0563C1" "FFFF0000"
```

```
unique(dataset2$bg_color)
```

```
## [1] "FFFFC000" NA         "FFE7E6E6"
```

So we can filter rows based on these two colors if we want.

To tidy-fy this dataset, we realize that there are really two interweaved datasets. The odd rows are one dataset and the even rows are another dataset.

```
dat1 <- dataset2 %>%
  filter( row %% 2 == 1) %>% # odd rows
  behead('N', header) %>%
  mutate(row = (row+1)/2) # make the row numbers sequential

dat2 <- dataset2 %>%
  filter(row %% 2 == 0) %>%  # even rows
  behead('N', header) %>%
  mutate(row = row/2) %>% # make row numbers sequential
  mutate(col = col+4) # These will be the last 4 cols of new data

tidy_dataset2 <-
```

```
  rbind(dat1, dat2) %>% # Put datsets on top of each other
  select(row, data_type, numeric, character, header) %>%
  spatter(header) %>%
  select(-row, -numeric)
```

We'll do a couple of finesse things to finish. First, we'll make the names with no spaces (they're a pain to write otherwise) and put the student name on the first column.

```
tidy_dataset2 <- tidy_dataset2 %>%
  set_names(make.names(names(.))) %>%
  select(Student.Name, everything())
```

`make.names` changes a vector of names into "approved" space-free format, replacing the space with `.`. One shortcut I've used is using `.` as an argument to `names`, which means that the `.` is replaced by the "noun" that is being acted on by the "verbs". You will notice that I can also do multiple verbs together to work sequentially.