1.3.2: Data Wrangling

(Asynchronous-Online)

Session Objectives

1. To read in data.
2. To view the Data.
3. To subset the data - select and filter.
4. To create and modify variables.
5. To get data summary and descriptive statistics
6. Exporting/Saving Data

---

0. Prework - Before You Begin

Install Packages

Before you begin, please go ahead and install the following packages - these are all on CRAN, so you can install them using the RStudio Menu "Tools/Install" Packages interface:

- `readr` on CRAN and `readr` package website
- `readxl` on CRAN and `readxl` package website
- `haven` on CRAN and `haven` package website
- `dplyr` on CRAN and `dplyr` package website
- `Hmisc` on CRAN and `Hmisc`package website
- `psych` on CRAN and `psych` package website
- `arsenal` on CRAN and `arsenal` package website
- `gtsummary` on CRAN and `gtsummary` package website
- `tableone` on CRAN
- `gmodels` on CRAN
- `pkgsearch` on CRAN
- `palmerpenguins` on CRAN

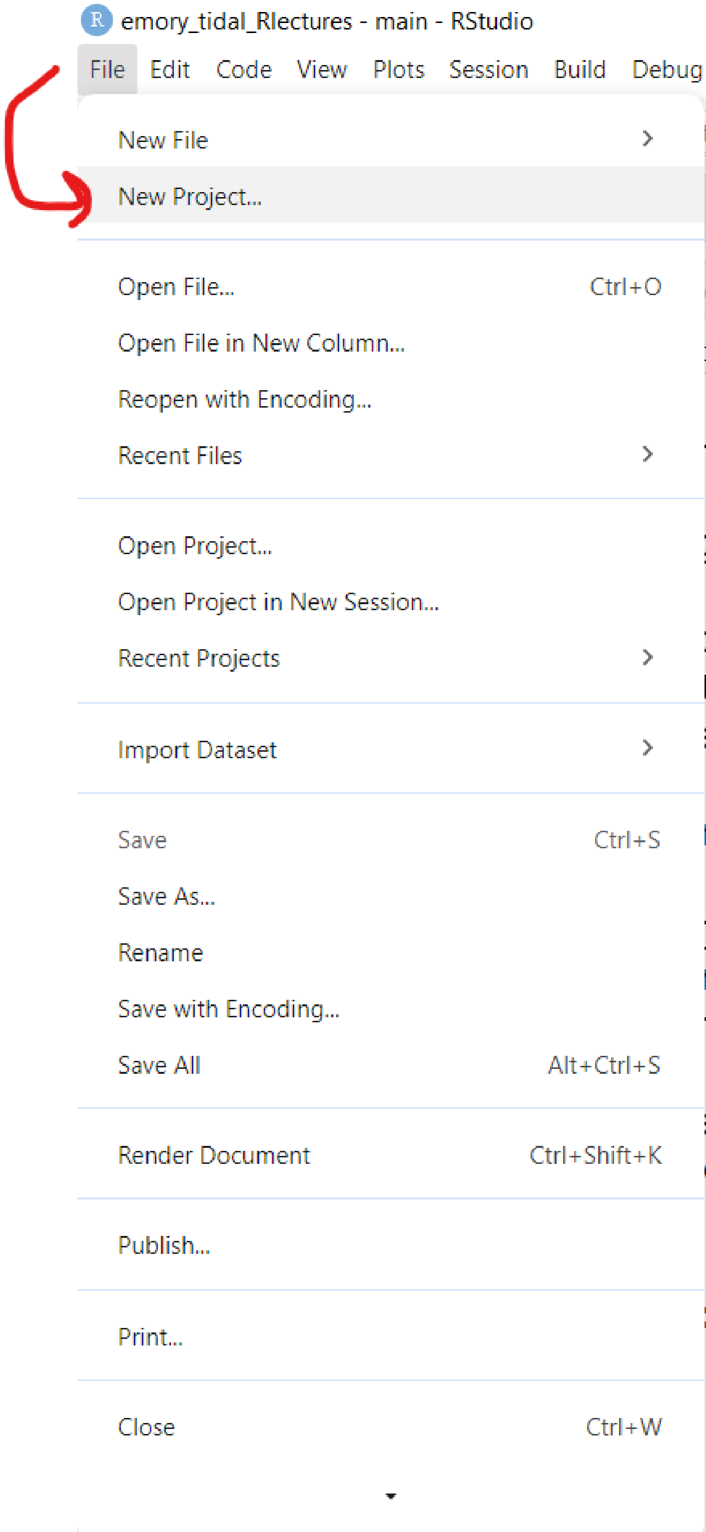
See Module 1.3.1 on Installing Packages

1. To read in data.

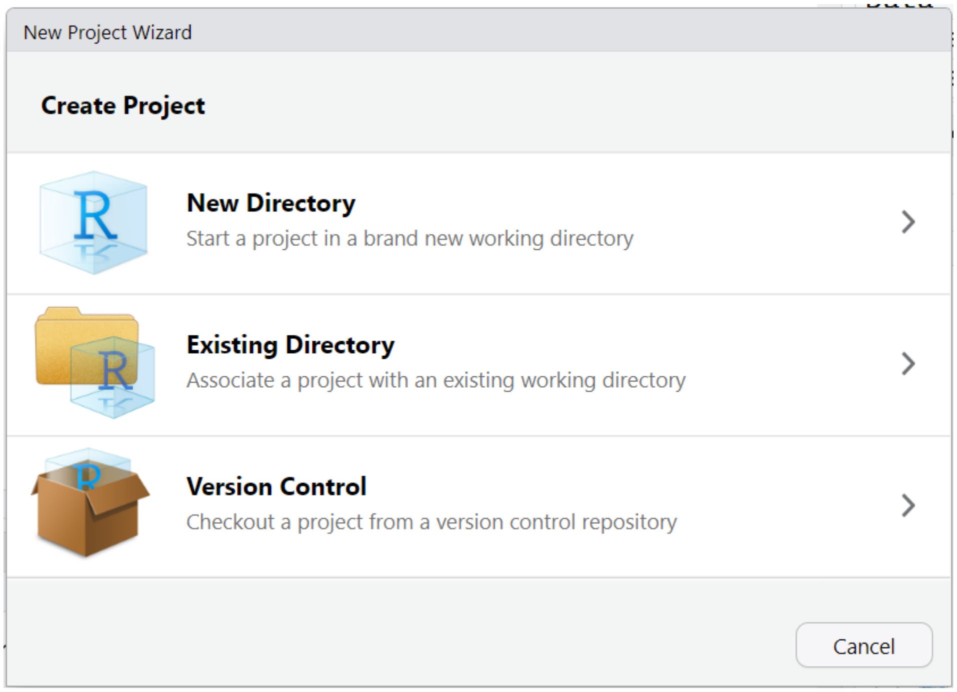
Begin with a NEW RStudio Project

Let's begin with a new RStudio Project.

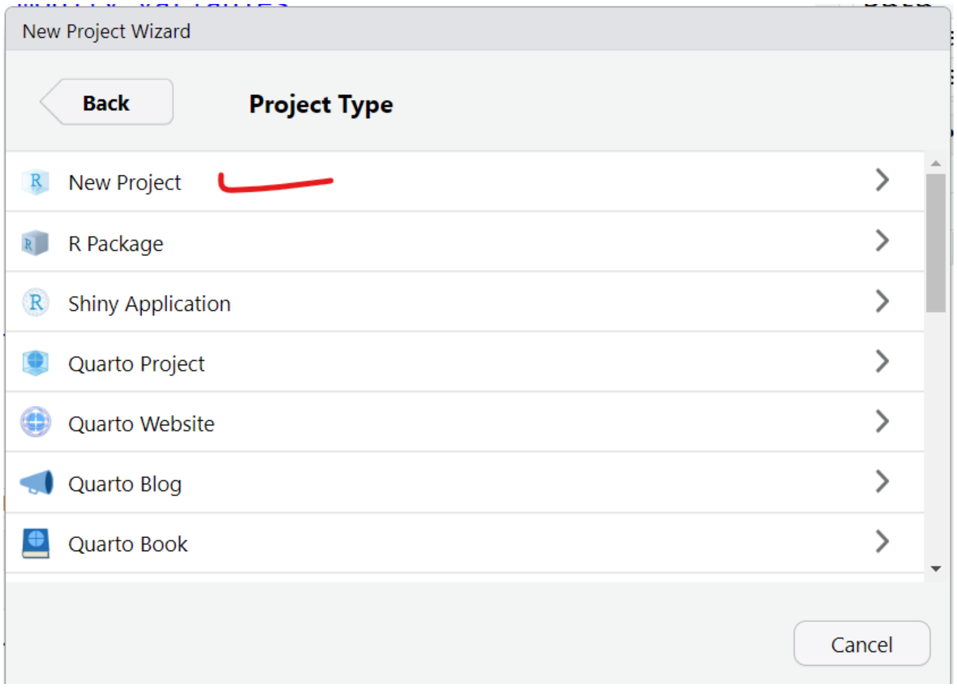
1. First click on the menu at top for "File/New Project":


emory_tidal_Rlectures - main - RStudio
File Edit Code View Plots Session Build Debug
New File
New Project...
Open File... Ctrl+O
Open File in New Column...
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Rename
Save with Encoding...
Save All Alt+Ctrl+S
Render Document Ctrl+Shift+K
Publish...
Print...
Close Ctrl+W

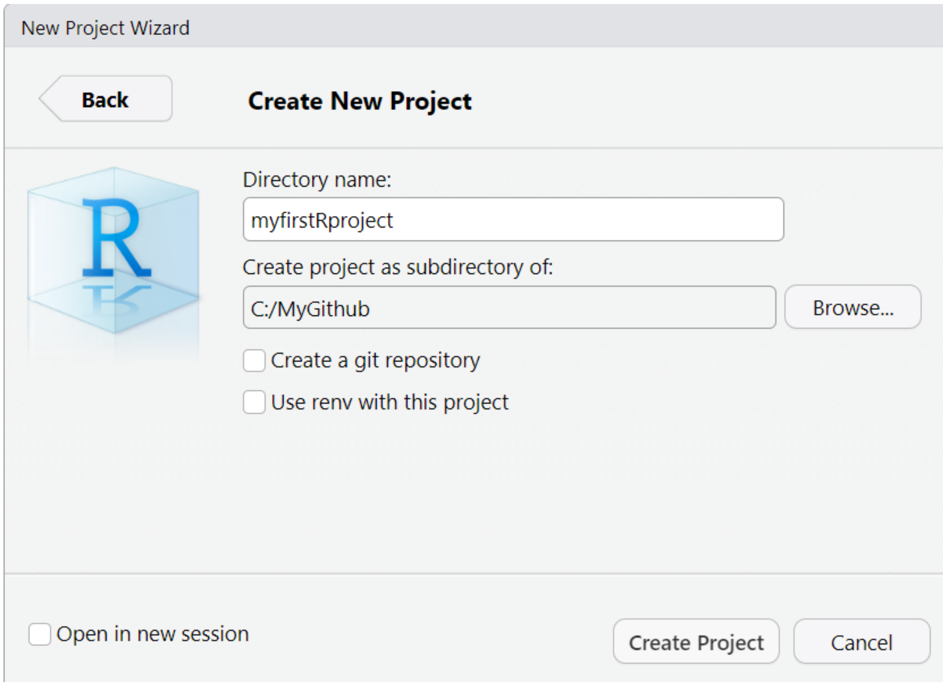
2. Next choose either an "Existing Directory" or "New Directory" depending on whether you want to use a folder that already exists on your computer or you want to create a new folder.



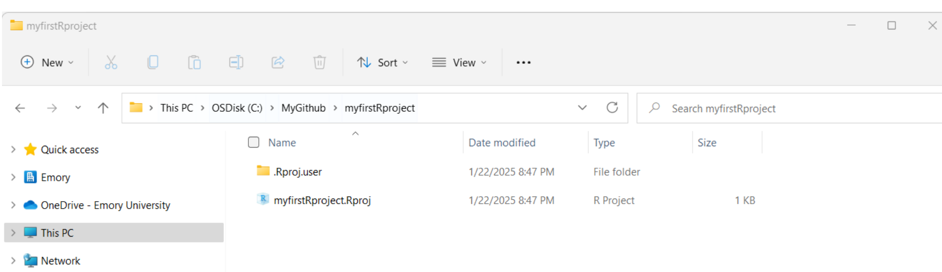
3. For now, let's choose a "New Directory" and then select "New Project"


New Project Wizard
Back
Project Type
New Project
R Package
Shiny Application
Quarto Project
Quarto Website
Quarto Blog
Quarto Book
Cancel

4. When the next window opens, as an example, I'm creating a new project folder called `myfirstRproject` for my RStudio project under my parent directory, `C:\MyGithub`. *Your folder names and directories will most likely be different than mine.*



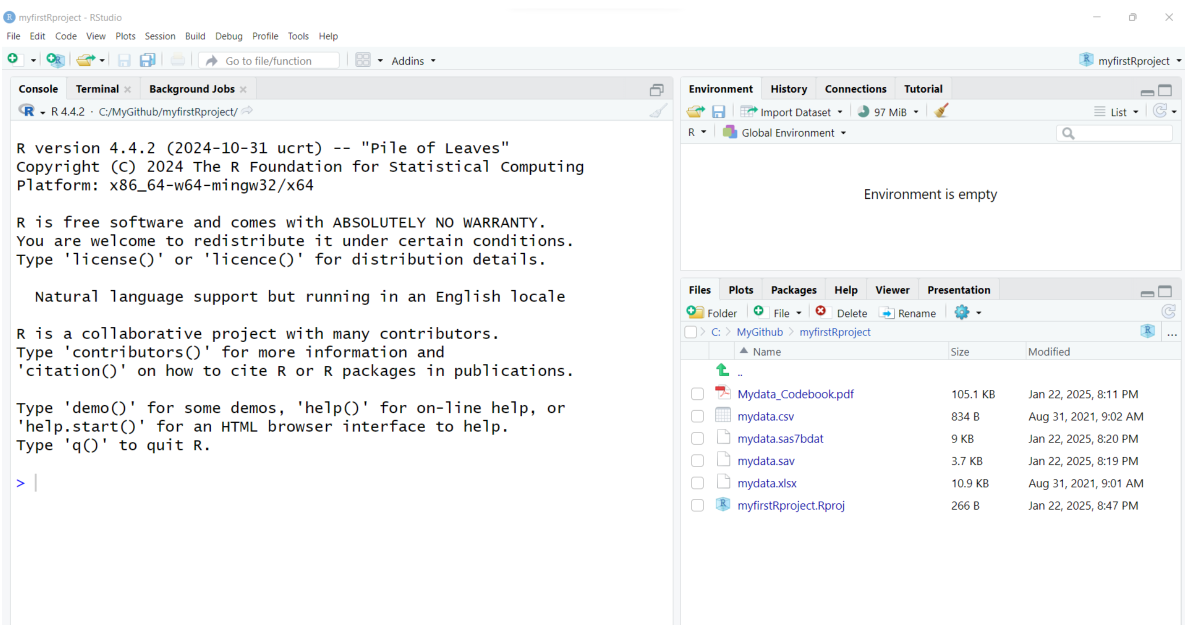
5. So, if I look back on my computer in my file manager (I'm on a computer with Windows 11 operating system) - I can now see this new folder on my computer for `C:\MyGithub\myfirstRproject`.



6. Now let's put some data into this folder. Feel free to move datasets of your own into this new RStudio project directory. But here are some test datasets you can download and place into this new directory on your computer - choose at least one to try out - right click on each link and use "Save As" to save the file on your computer in your new project folder.

- `mydata.csv` - CSV (comma separated value) formatted data
- `mydata.xlsx` - EXCEL file
- `mydata.sav` - SPSS Dataset
- `mydata.sas7bdat` - SAS Dataset
- `Mydata_Codebook.pdf` - Codebook for "mydata" dataset

7. After putting these files into your new RStudio project folder, you should see something like this now in your RStudio Files Listing (bottom right window pane):



Importing Data

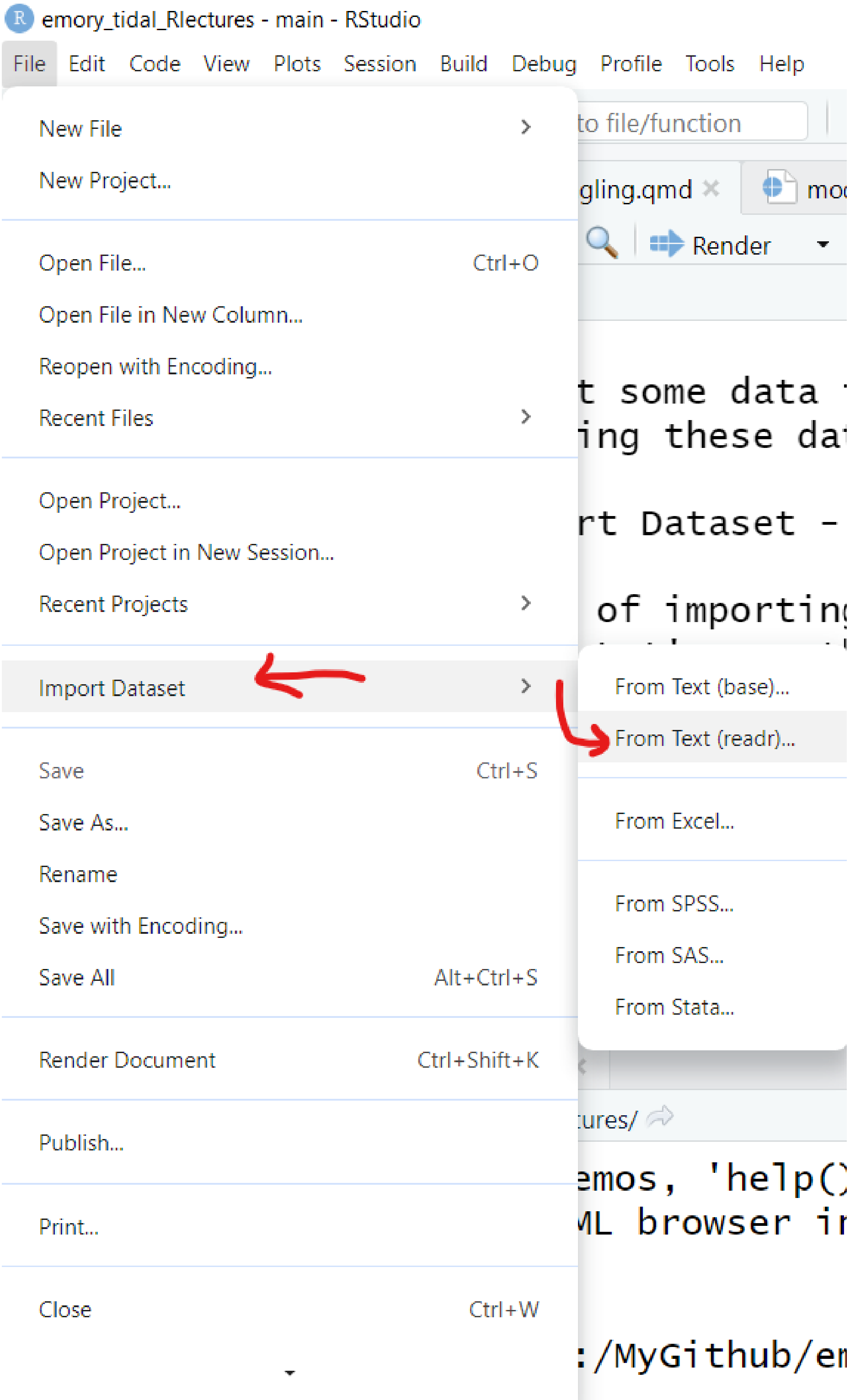
Now that you've got some data in your RStudio project folder, let's look at options for importing these datasets into your RStudio computing session.

Click on "File/Import Dataset" - and then choose the file format you want.

Import a CSV file

> **What is a CSV file?**
>
> CSV stands for "comma separated value" format. This format is what you would think - each value for a different column (or variable) is separated by a column and each new row represents a new record in the dataset.
>
> CSV is widely accepted as a "universal" standard as a data format for easy exchange between different software and databases.
>
> - Wikipedia Page on CSV
> - Library of Congress Page on CSV
> - There is even a conference on CSV

Here is an example of importing the `mydata.csv` - CSV formatted data. Let's use the `From Text (readr)` option.

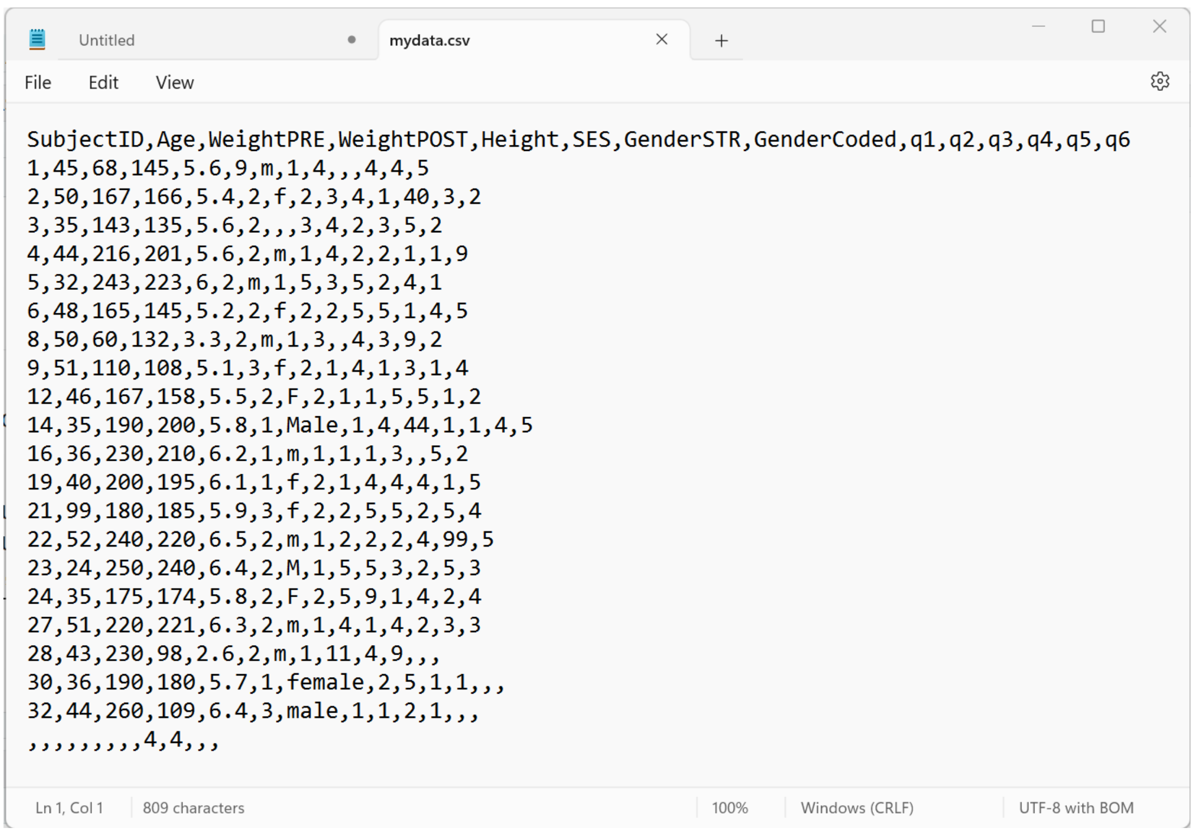

emory_tidal_Rlectures - main - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
New File
New Project...
Open File... Ctrl+O
Open File in New Column...
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Rename
Save with Encoding...
Save All Alt+Ctrl+S
Render Document Ctrl+Shift+K
Publish...
Print...
Close Ctrl+W
From Text (base)...
From Text (readr)...
From Excel...
From SPSS...
From SAS...
From Stata...
Render

Why should we use the "from text" option? Why do I not see a CSV option?

Technically the CSV format is TEXT. You can open a CSV file in a text editor and easily read it - even if you do not have proprietary software like Excel, Access, SPSS, SAS, etc. Here is a screen shot of what the "mydata.csv" file looks like in my text editor "Notepad" on my Windows 11 computer:

Notice that:

- The first row has text labels for the "variables" (columns) in the dataset - there are 14 column labels with each value separated by a `,` comma.
- The remaining rows are the "data" for the dataset.
- After the 1st row of labels, there are 21 rows of data.
- Take a minute and notice there are some odd values, and odd patterns of missing data (two commas `,,` together indicate that value is missing for that column (variable)). *We'll explore these issues further in later lesson modules.*

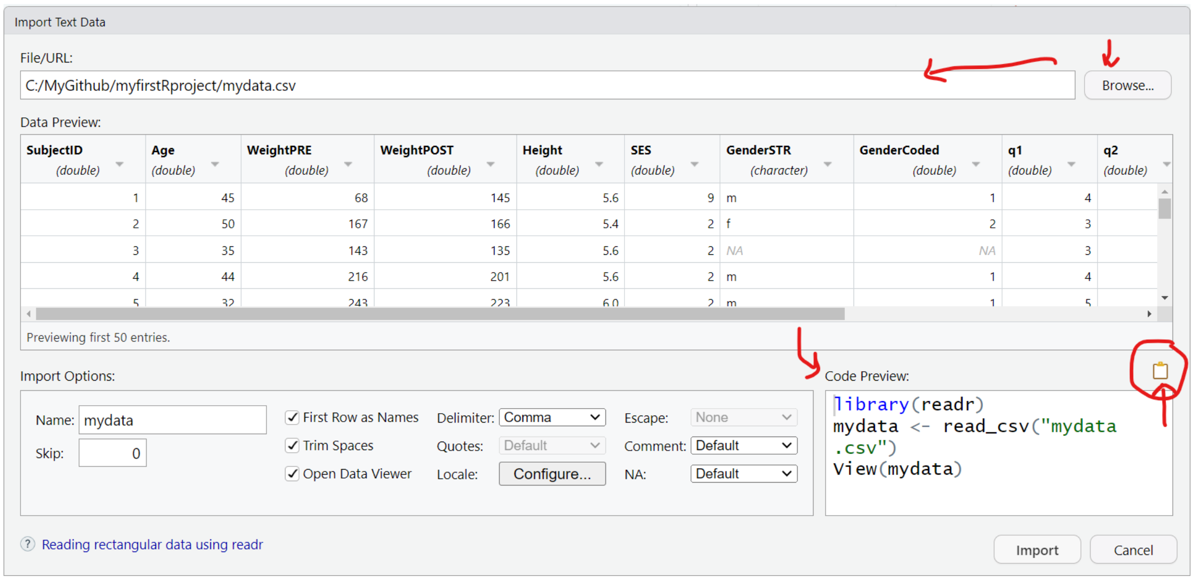



```
SubjectID,Age,WeightPRE,WeightPOST,Height,SES,GenderSTR,GenderCoded,q1,q2,q3,q4,q5,q6
1,45,68,145,5.6,9,m,1,4,,,4,4,5
2,50,167,166,5.4,2,f,2,3,4,1,40,3,2
3,35,143,135,5.6,2,,,3,4,2,3,5,2
4,44,216,201,5.6,2,m,1,4,2,2,1,1,9
5,32,243,223,6,2,m,1,5,3,5,2,4,1
6,48,165,145,5.2,2,f,2,2,5,5,1,4,5
8,50,60,132,3.3,2,m,1,3,,4,3,9,2
9,51,110,108,5.1,3,f,2,1,4,1,3,1,4
12,46,167,158,5.5,2,F,2,1,1,5,5,1,2
14,35,190,200,5.8,1,Male,1,4,44,1,1,4,5
16,36,230,210,6.2,1,m,1,1,1,3,,5,2
19,40,200,195,6.1,1,f,2,1,4,4,4,1,5
21,99,180,185,5.9,3,f,2,2,5,5,2,5,4
22,52,240,220,6.5,2,m,1,2,2,2,4,99,5
23,24,250,240,6.4,2,M,1,5,5,3,2,5,3
24,35,175,174,5.8,2,F,2,5,9,1,4,2,4
27,51,220,221,6.3,2,m,1,4,1,4,2,3,3
28,43,230,98,2.6,2,m,1,11,4,9,,,
30,36,190,180,5.7,1,female,2,5,1,1,,,
32,44,260,109,6.4,3,male,1,1,2,1,,,
,,,,,,,,,4,4,,,
```

Once the "File/Import Data/From Text (readr)" opens, click on "Browse" and choose the `mydata.csv` file. Assuming all goes well, this window will read the top of the datafile and show you a quick "Data Preview" to check that the import will work.

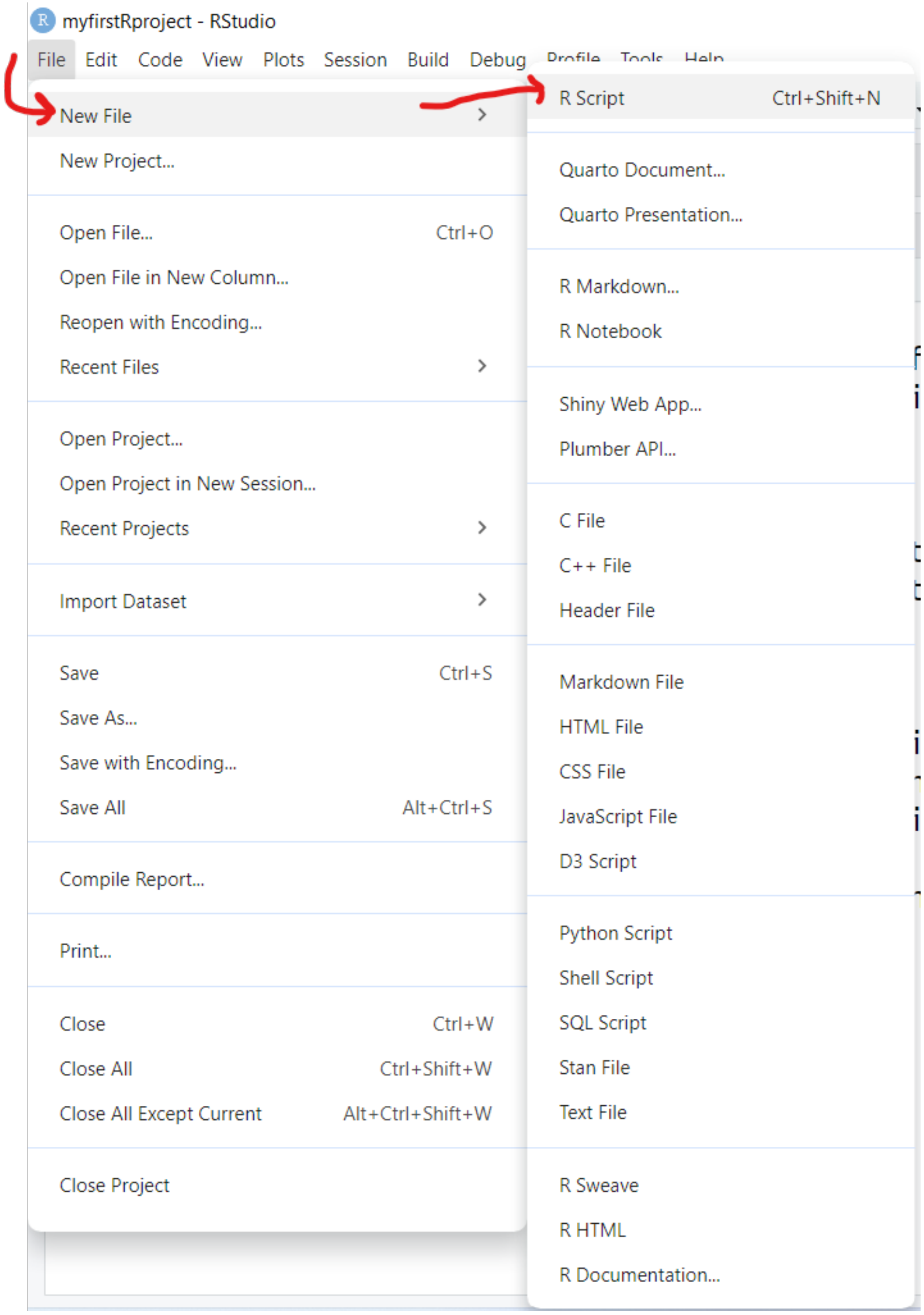
And on the bottom right, the "Code Preview" shows you the R code commands needed to import this dataset. You can then click on the little "clipboard" on the bottom right to copy this R code to your "clipboard", *(the R code option will be explained below)*.

OR You can also just click "Import" and the R code will be executed for you and the dataset brought into your R computing session *(but this is NOT a good practice for reproducible research!)*.



The better way is to save the R code commands to import the data so you will be able to reproduce all steps in your data analysis workflow using code as opposed to non-reproducible point-and-click steps.

Once you copied the R code above to your clipboard, go to "File/New File/R Script" to open a script programming window:



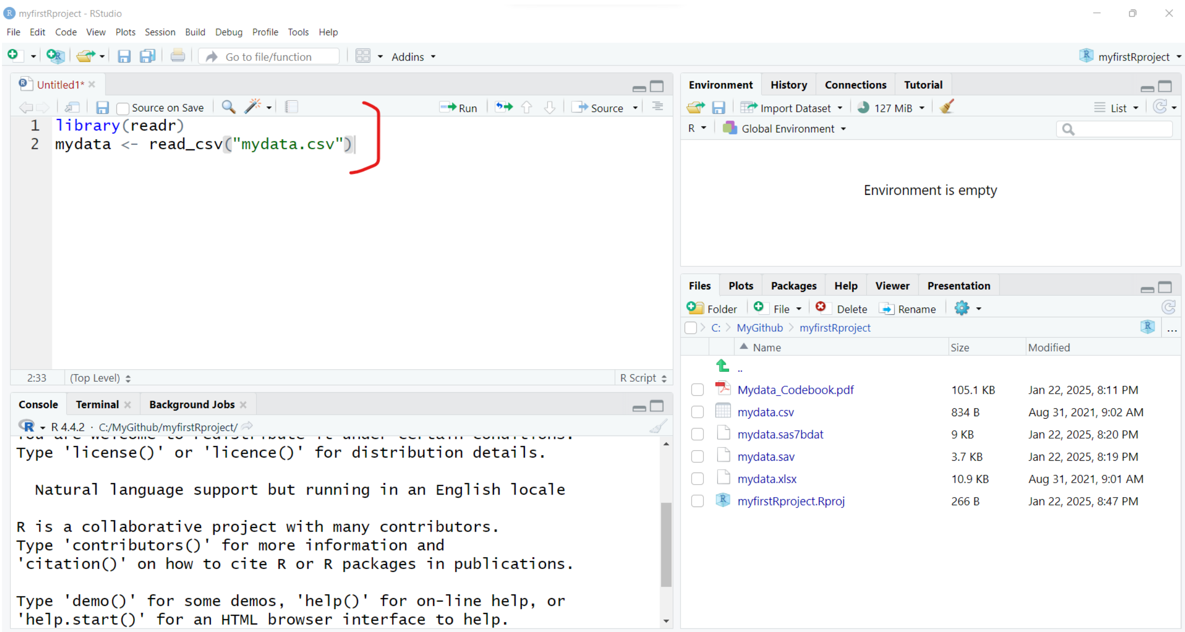
And then "paste" your R code into this window.

As you can see importing the `mydata.csv` dataset, involves 2 steps:

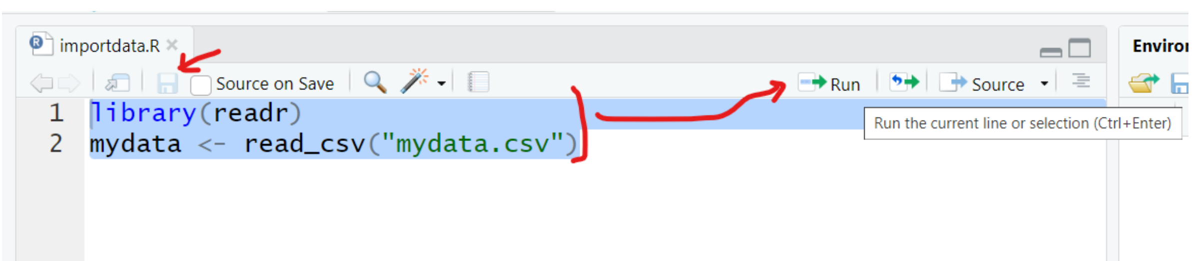
1. Loading the `readr` package into your RStudio computing session, by running

```
library(readr)
```

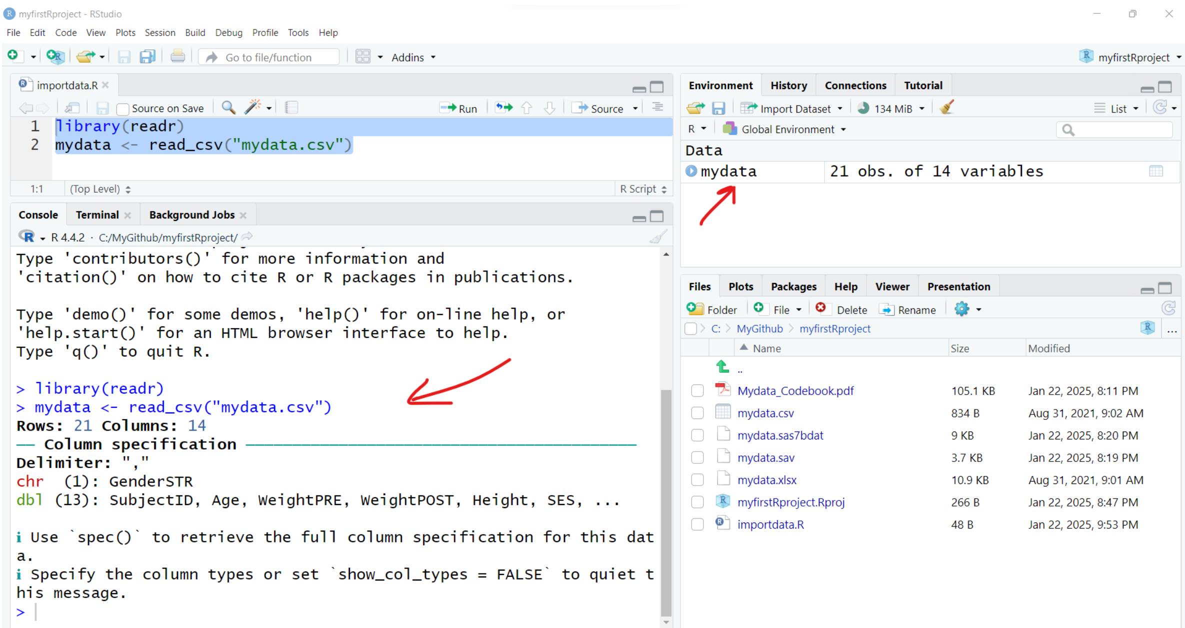
2. Running the `read_csv()` function from the `readr` package and then assigning `<-` this output into a new R data object called `mydata`.



To import the dataset, select these 2 lines of code and then click "Run" to run the R code. And be sure to click "Save" to save your first R program - for example "importdata.R".



After running these 2 lines of code, you should see something like this - the code messages in the bottom left "Console" window pane and a new R data object "mydata" in the top right "Global Environment" window pane.

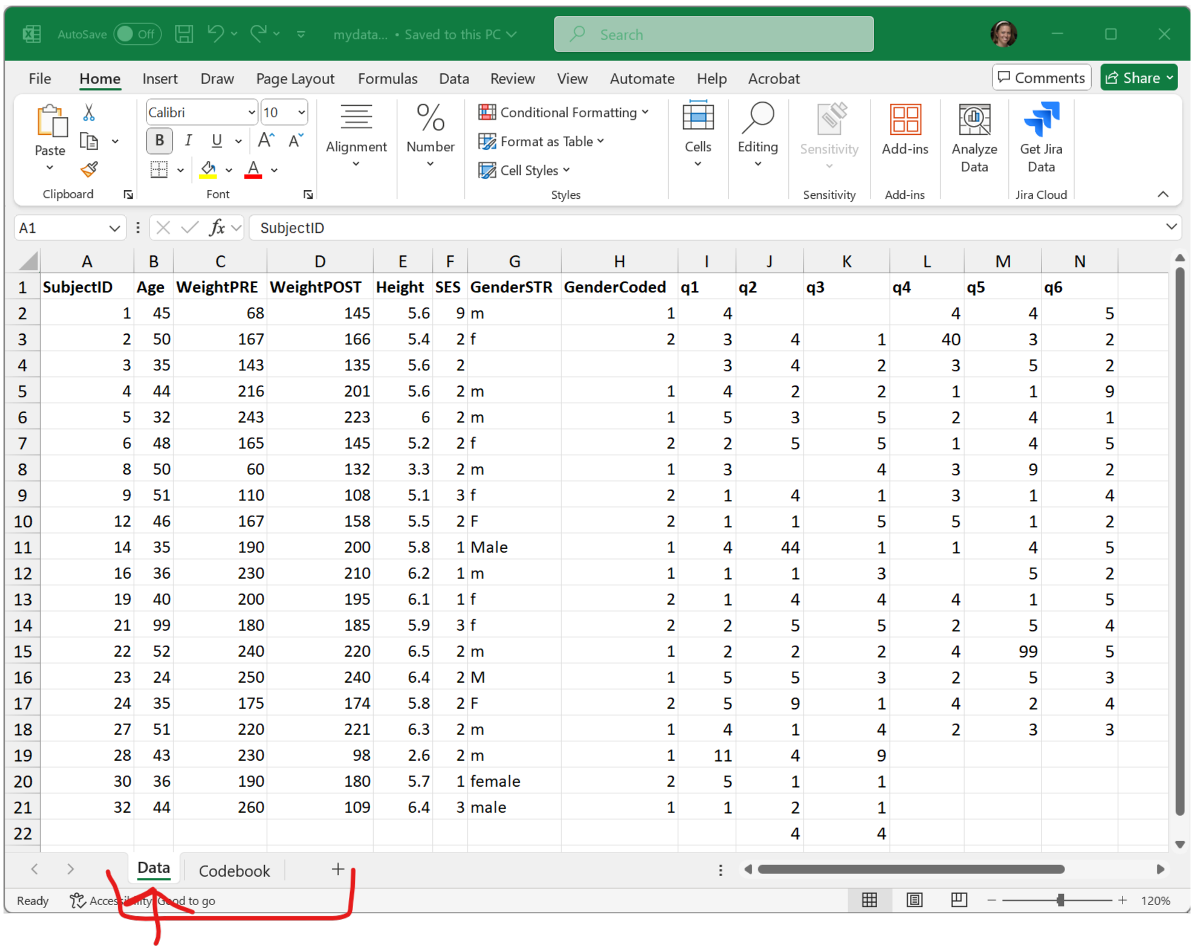


Import an EXCEL file

Let's try another format. While you will probably encounter CSV (comma separated value) data files often (since nearly all data collection platforms, databases and software will be able to export this simple non-proprietary format), many people natively open/read CSV files in the EXCEL software. So you will probably also encounter EXCEL (`*.XLS` or `*.XLSX`) formatted data files.

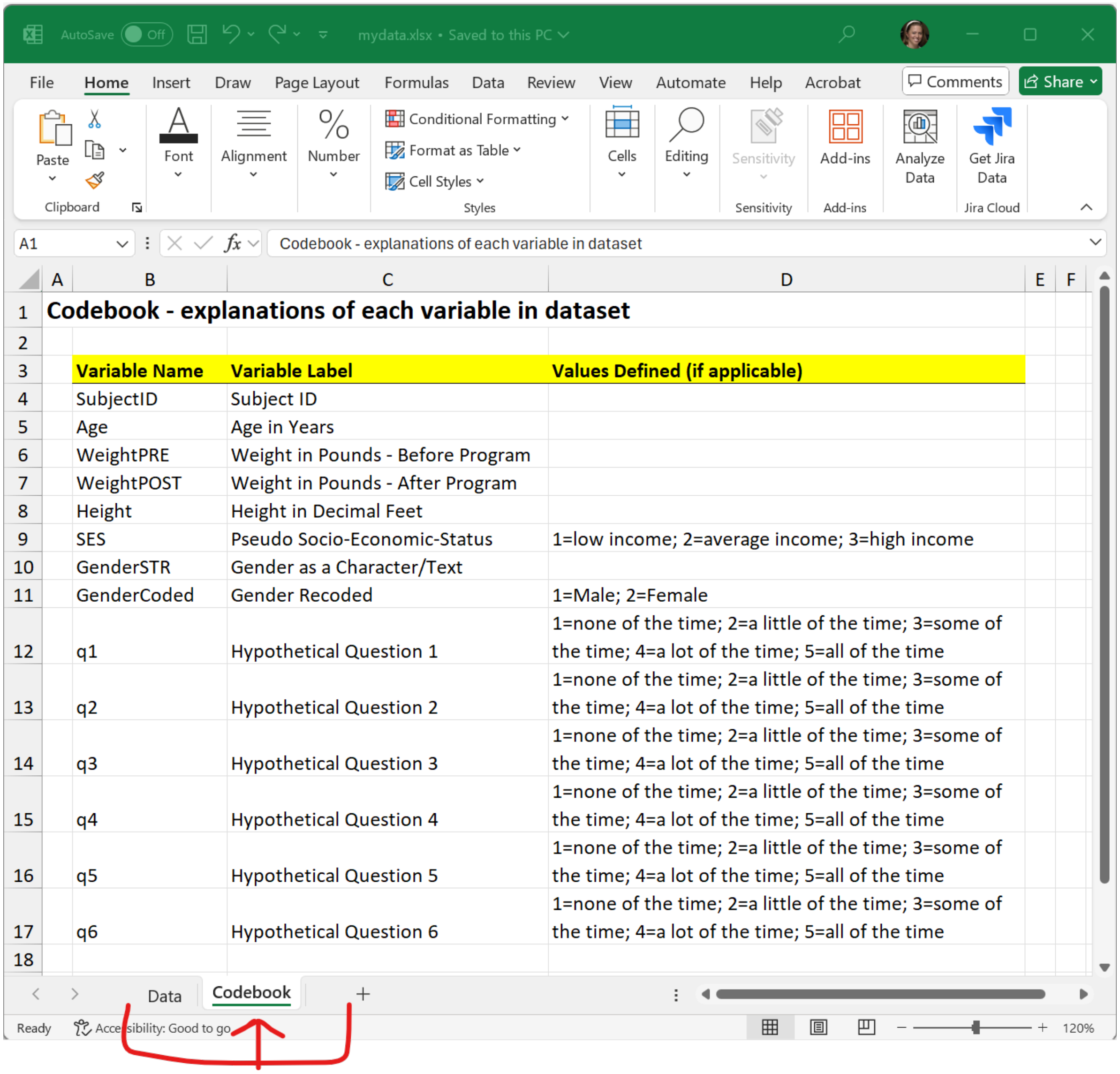
In addition to an EXCEL file using a Microsoft proprietary format, EXCEL files can have formatting (font sizes, colors, borders) and can have multiple TABs (or SHEETs). Here are some screen shots of the `mydata.xlsx` - EXCEL file file.

The first "Data" TAB:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	SubjectID	Age	WeightPRE	WeightPOST	Height	SES	GenderSTR	GenderCoded	q1	q2	q3	q4	q5	q6
2	1	45	68	145	5.6	9	m	1	4			4	4	5
3	2	50	167	166	5.4	2	f	2	3	4	1	40	3	2
4	3	35	143	135	5.6	2			3	4	2	3	5	2
5	4	44	216	201	5.6	2	m	1	4	2	2	1	1	9
6	5	32	243	223	6	2	m	1	5	3	5	2	4	1
7	6	48	165	145	5.2	2	f	2	2	5	5	1	4	5
8	8	50	60	132	3.3	2	m	1	3		4	3	9	2
9	9	51	110	108	5.1	3	f	2	1	4	1	3	1	4
10	12	46	167	158	5.5	2	F	2	1	1	5	5	1	2
11	14	35	190	200	5.8	1	Male	1	4	44	1	1	4	5
12	16	36	230	210	6.2	1	m	1	1	1	3		5	2
13	19	40	200	195	6.1	1	f	2	1	4	4	4	1	5
14	21	99	180	185	5.9	3	f	2	2	5	5	2	5	4
15	22	52	240	220	6.5	2	m	1	2	2	2	4	99	5
16	23	24	250	240	6.4	2	M	1	5	5	3	2	5	3
17	24	35	175	174	5.8	2	F	2	5	9	1	4	2	4
18	27	51	220	221	6.3	2	m	1	4	1	4	2	3	3
19	28	43	230	98	2.6	2	m	1	11	4	9			
20	30	36	190	180	5.7	1	female	2	5	1	1			
21	32	44	260	109	6.4	3	male	1	1	2	1			
22										4	4			

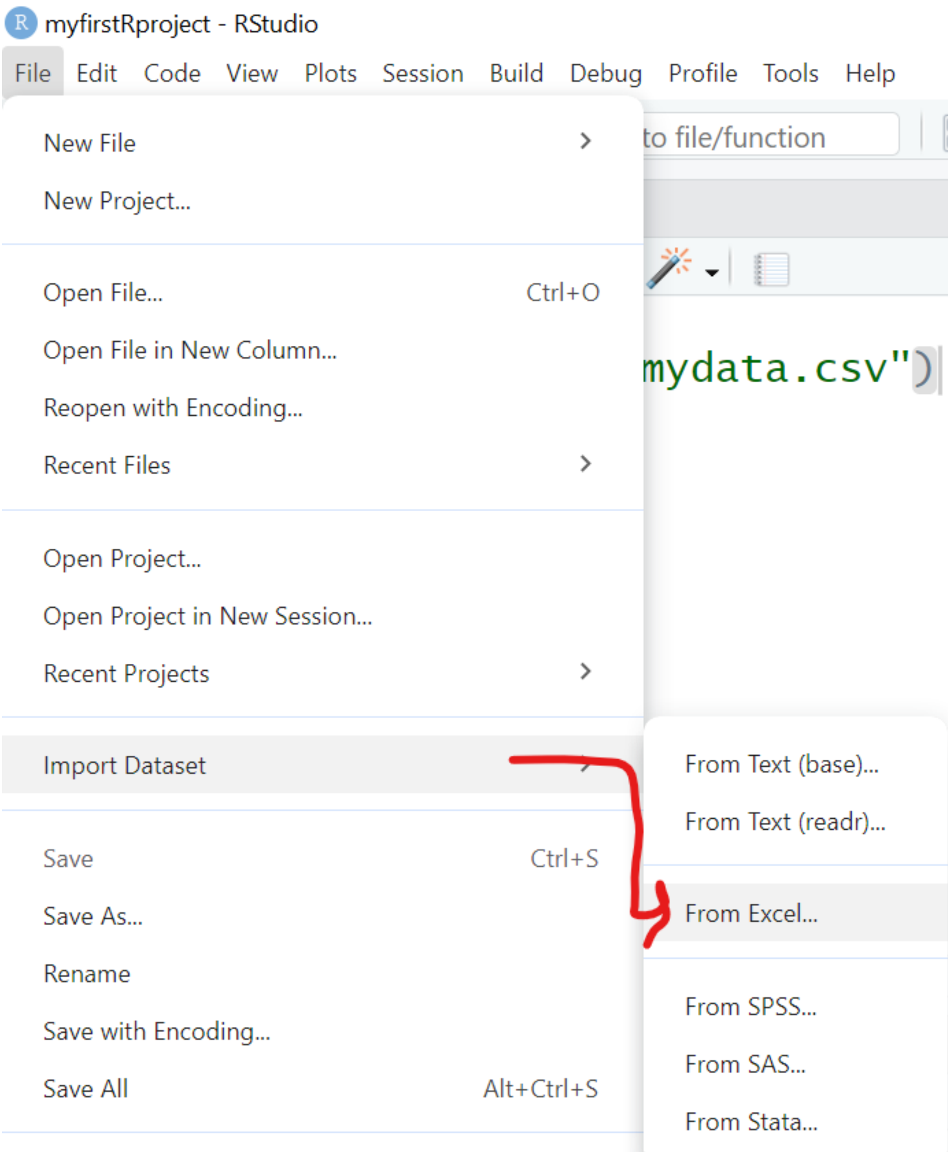
The second "Codebook" TAB:



Codebook - explanations of each variable in dataset

Variable Name	Variable Label	Values Defined (if applicable)
SubjectID	Subject ID	
Age	Age in Years	
WeightPRE	Weight in Pounds - Before Program	
WeightPOST	Weight in Pounds - After Program	
Height	Height in Decimal Feet	
SES	Pseudo Socio-Economic-Status	1=low income; 2=average income; 3=high income
GenderSTR	Gender as a Character/Text	
GenderCoded	Gender Recoded	1=Male; 2=Female
q1	Hypothetical Question 1	1=none of the time; 2=a little of the time; 3=some of the time; 4=a lot of the time; 5=all of the time
q2	Hypothetical Question 2	1=none of the time; 2=a little of the time; 3=some of the time; 4=a lot of the time; 5=all of the time
q3	Hypothetical Question 3	1=none of the time; 2=a little of the time; 3=some of the time; 4=a lot of the time; 5=all of the time
q4	Hypothetical Question 4	1=none of the time; 2=a little of the time; 3=some of the time; 4=a lot of the time; 5=all of the time
q5	Hypothetical Question 5	1=none of the time; 2=a little of the time; 3=some of the time; 4=a lot of the time; 5=all of the time
q6	Hypothetical Question 6	1=none of the time; 2=a little of the time; 3=some of the time; 4=a lot of the time; 5=all of the time

To import an EXCEL file into R, we will use the same process as above, but this time we will select "File/Import Dataset/From Excel":



This process uses the `read_excel()` function from the `readxl` package.

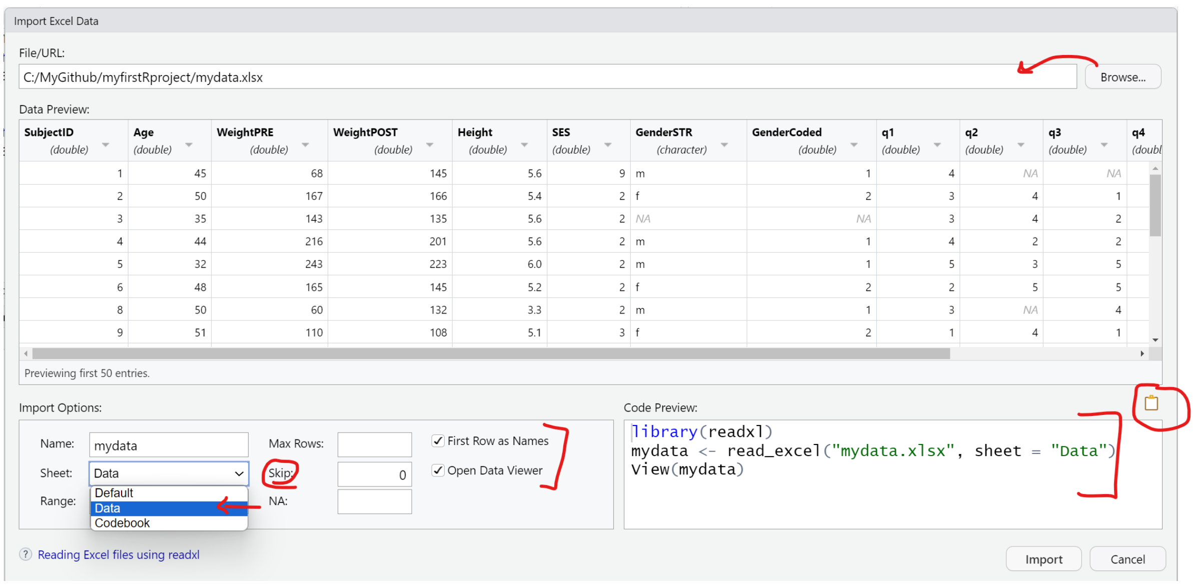
With the `read_excel()` function, we can specify several options including:

- Which TAB do you want to import *(for now we are only importing one data TAB at a time)*. We are selecting the "Data" TAB.
- I'm leaving all of the rest as their defaults which include:

- not changing the "Range",
- leaving "Max Rows" blank,
- and leaving rows to "Skip" as 0, which can be useful if you receive files with a lot of "header" information at the top,,
- leaving the "NA" box blank - but you could put in a value like "99" if you want all 99's treated as missing - but this is applied to the ENTIRE dataset. We will look at these issues for individual variables below.

• Also notice that the checkboxes are selected for "First Row as Names" (which is the usual convention) and "Open Data Viewer", which creates the `View(mydata)` in the "Code Preview" window to the right. You can skip this if you like.

So in the "Code Preview" window to the right, we have specified the name of the data file `"mydata.xlsx"` and the "Data" TAB using the option `sheet = "Data"`. Remember to copy this code to the clipboard and save it in a `*.R` program script.

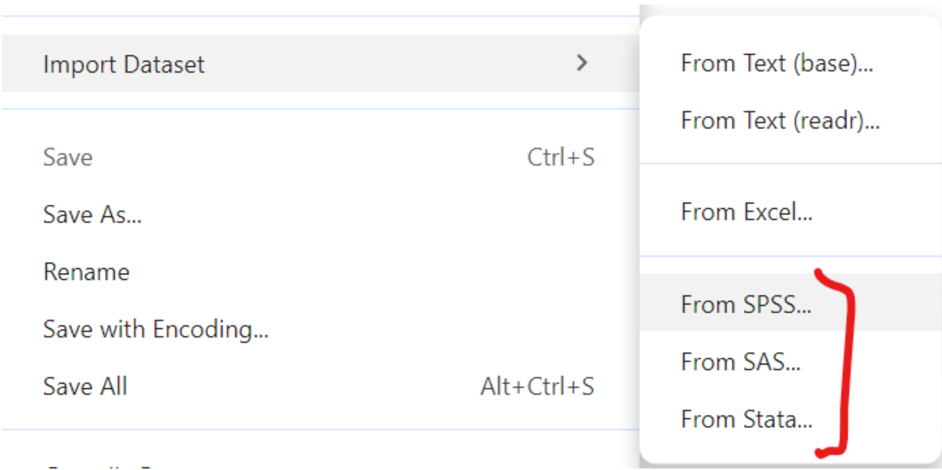


Here is the `importdata.R` program script we have so far for reading in the `"mydata.csv"` and `"mydata.xlsx"` data files. *At the moment, the second time we "create" the* ***`mydata`*** *R data object we are overwriting the previous one in the sequential code steps below.*

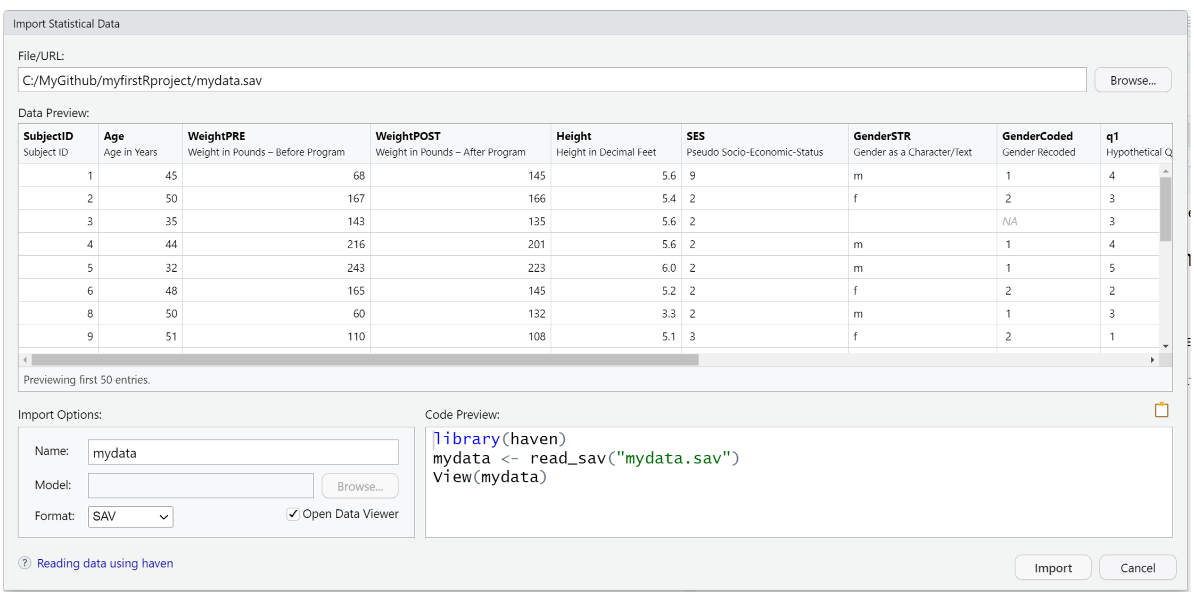
Also notice I have added some comments which start with a `#` hashtag. Any text following a `#` will be ignored by R and not executed.

Import SPSS data

For data files from other "common" statistics software like SPSS, SAS and Stata, we can use the "File/Import Dataset/From SPSS (or From SAS or From Stata)". All of these use `read_xxx()` functions from the `haven` package.



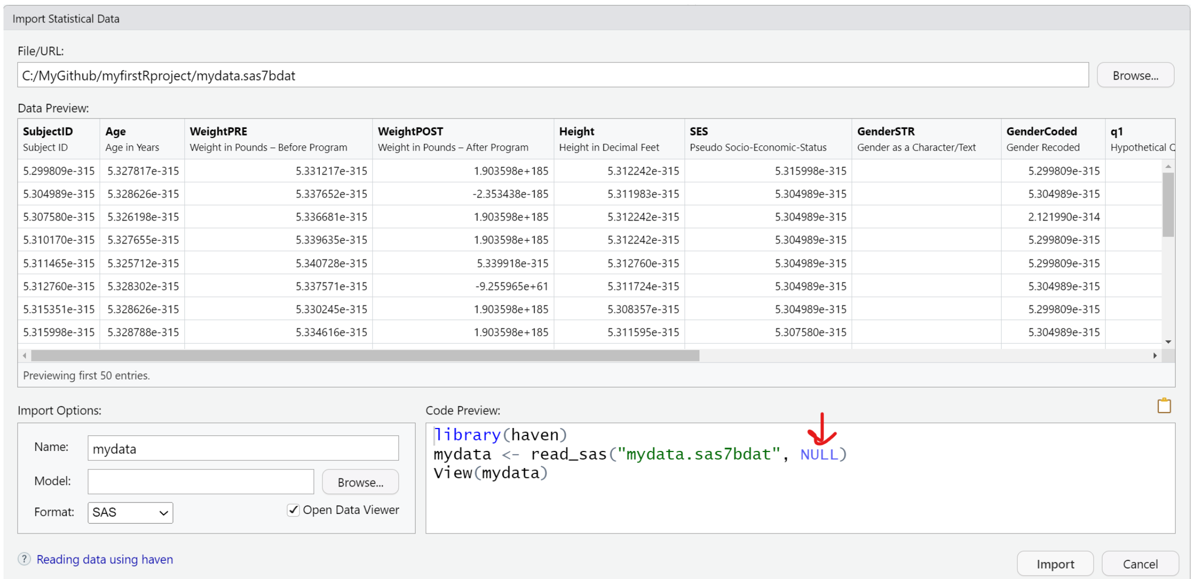
Here is the code generated to import a SPSS datafile:

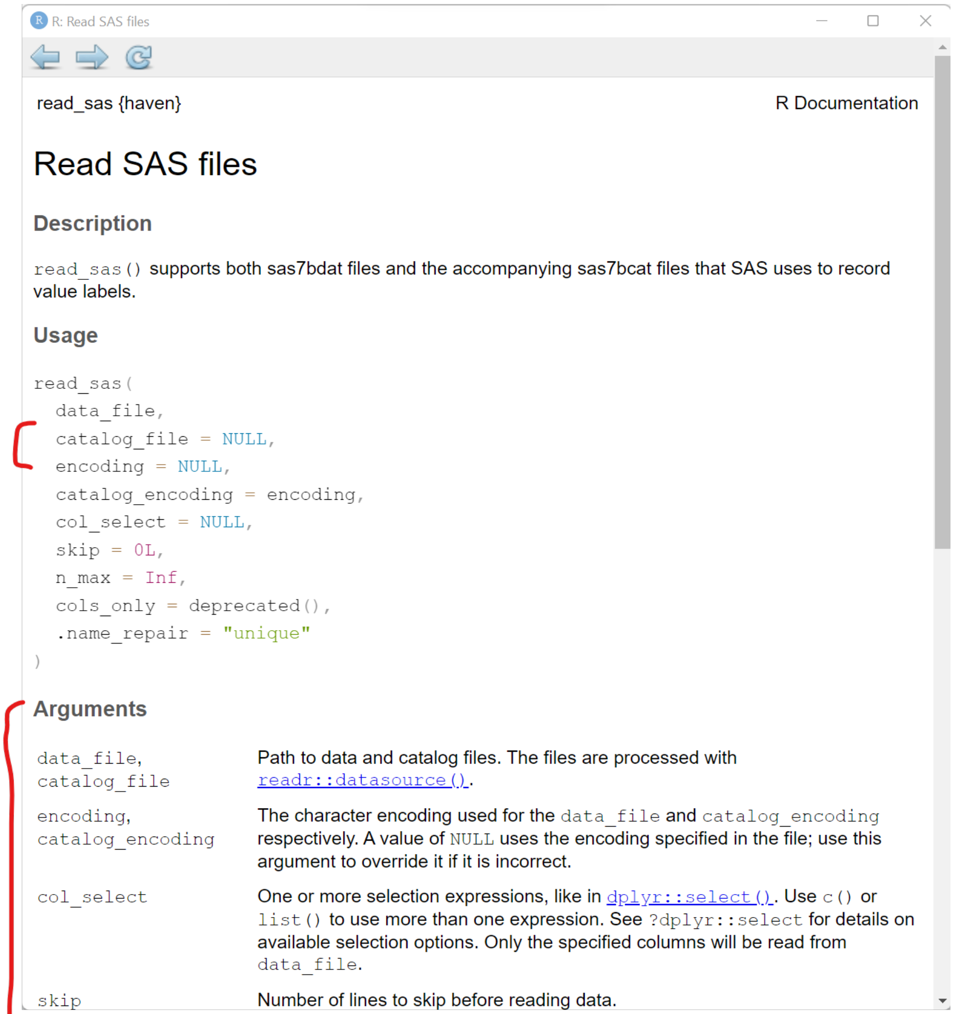


Import SAS data

Importing a `*.sas7bdat` SAS datafile, is similar to SPSS - here is that code.

Notice that in addition to the datafile `"mydata.sas7bdat"`, the `read_sas()` function also shows `NULL`. When reading in a SAS file, you can also add arguments for the catalog file and encoding specifics. You can read more on the Help pages for the `haven::read_sas()` function.





R: Read SAS files

read_sas {haven} R Documentation

Read SAS files

Description

`read_sas()` supports both sas7bdat files and the accompanying sas7bcat files that SAS uses to record value labels.

Usage

```
read_sas(
  data_file,
  catalog_file = NULL,
  encoding = NULL,
  catalog_encoding = encoding,
  col_select = NULL,
  skip = 0L,
  n_max = Inf,
  cols_only = deprecated(),
  .name_repair = "unique"
)
```

Arguments

`data_file`, `catalog_file`	Path to data and catalog files. The files are processed with `readr::datasource()`.
`encoding`, `catalog_encoding`	The character encoding used for the `data_file` and `catalog_encoding` respectively. A value of `NULL` uses the encoding specified in the file; use this argument to override it if it is incorrect.
`col_select`	One or more selection expressions, like in `dplyr::select()`. Use `c()` or `list()` to use more than one expression. See `?dplyr::select` for details on available selection options. Only the specified columns will be read from `data_file`.
`skip`	Number of lines to skip before reading data.

Here is a quick summary of all of the data import codes shown above `importdata.R`:

> **Using = equals for parameter options inside a function**
>
> Notice that we used `sheet = "Data"` inside the `readxl::read_excel()` function. The single `=` equals sign is used to assign a value to a parameter or option inside a function.

```
# Import the CSV file
library(readr)
mydata <- read_csv("mydata.csv")

# Import the EXCEL file
# Choose the "Data" TAB
library(readxl)
mydata <- read_excel("mydata.xlsx", sheet = "Data")

# Import a SPSS file
library(haven)
mydata <- read_sav("mydata.sav")

# Import a SAS file
library(haven)
mydata <- read_sas("mydata.sas7bdat", NULL)
```

> **`haven` and `foreign` packages**
>
> In addition to the `haven` package which is part of `tidyverse` and has been around since 2015, there is also another useful package for importing and exporting other statistical software formats that has been around since 1999 and it still being maintained - the `foreign` package.
>
> In addition to SPSS and Stata, the `foreign` package also can read in other formats like DBF, EPI INFO, Minitab, Octave, SSD (SAS Permanent Datasets via XPORT) SYSTAT, and ARFF.
>
> Compare current downloads of these 2 packages at https://hadley.shinyapps.io/cran-downloads/.
>
> We can also review the history of these 2 packages using the `pkgsearch` package and the `cran_package_history()` function.

```r
# optionally install pkgsearch
# install.packages("pkgsearch")
library(pkgsearch)

# get history of haven package
havenhistory <- cran_package_history("haven")

# get history of foreign package
foreignhistory <- cran_package_history("foreign")

# display the earliest date on CRAN
# for these 2 packages
havenhistory$date[1]
```

```
[1] "2015-03-01T08:18:16+00:00"
```

```r
foreignhistory$date[1]
```

```
[1] "1999-12-17T02:05:13+00:00"
```

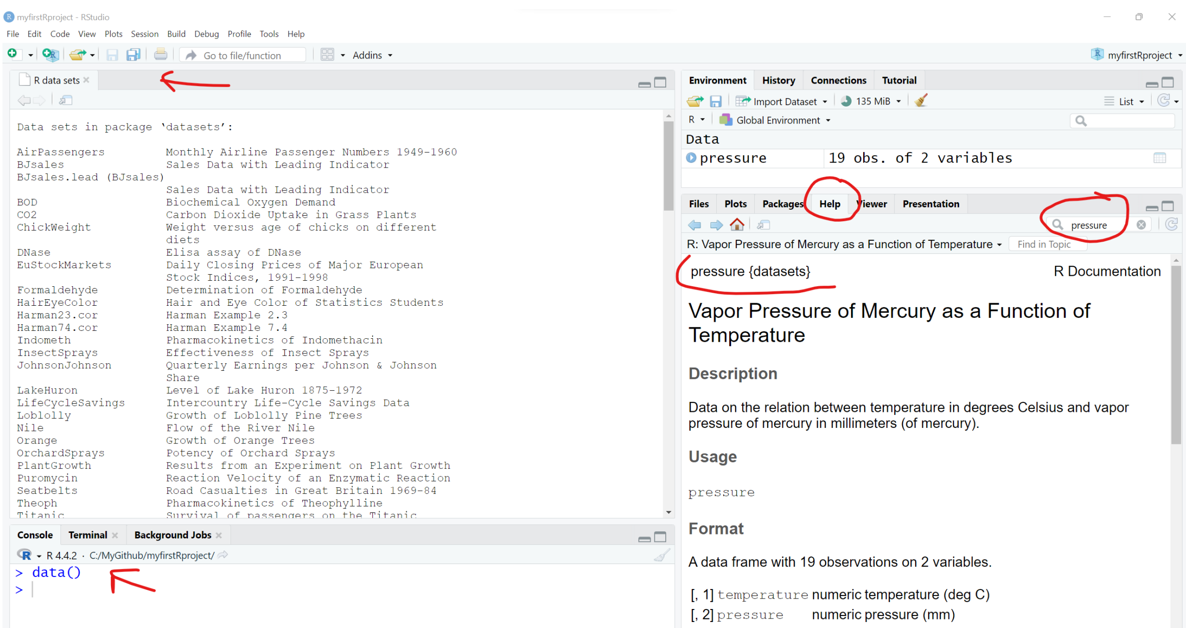
Exploring Built-in Datasets

If you are looking for other datasets to test out functions or just need some data to play around with, the base R packages and other R packages (like `palmerpenguins`) have data built-in to them. You can use these datasets.

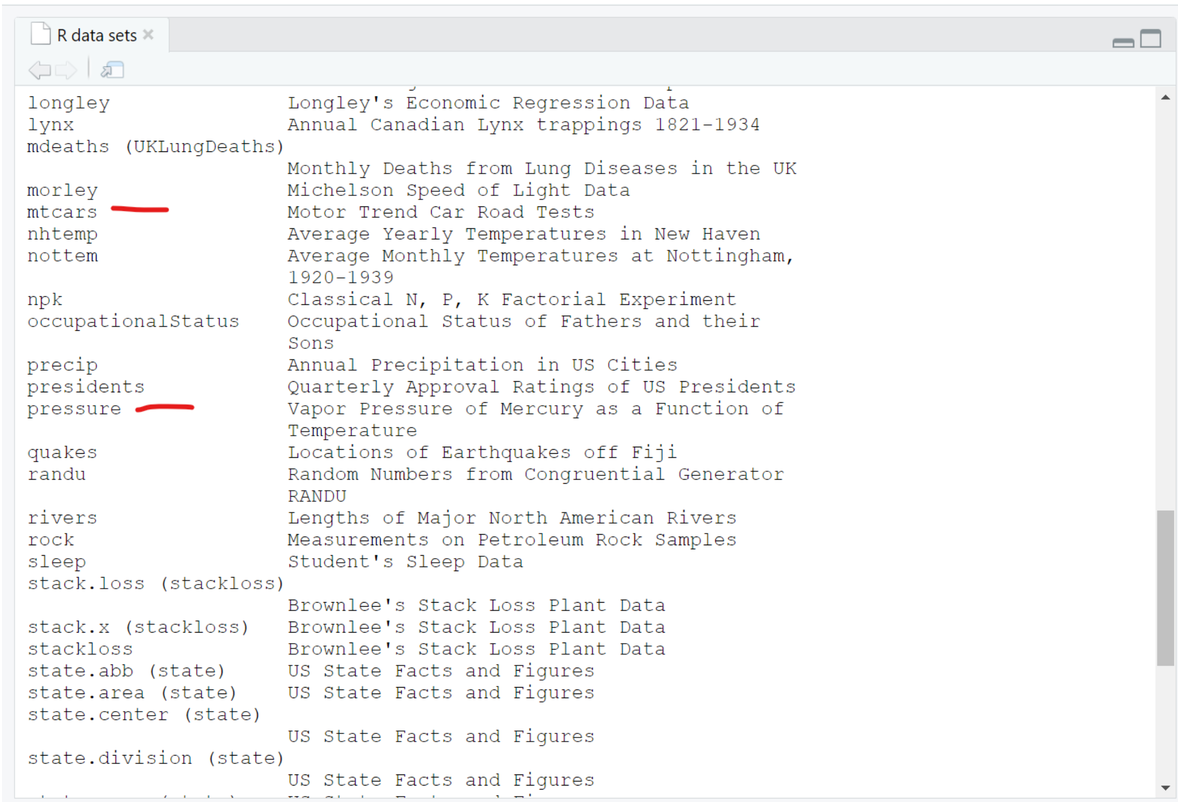
We can take a look at what datasets are available using the `data()` function:

```r
# take a look at the datasets available in the
# "datasets" base R package
data()
```

This will open a viewer window (top left) - also notice that if you search for "Help" on the `pressure` dataset, you get a description of the dataset and the original source and citation. Notice in the "Help" window, the word `pressure` is followed by curly brackets indicating that the `pressure` dataset is in the built-in R package `{datasets}`.



We can see the **pressure** dataset is indeed in the **datasets** package if we keep scrolling down in the viewer window - also notice the **mtcars** dataset which you will often find in R tutorials and coding examples.



R data sets

```
longley                   Longley's Economic Regression Data
lynx                      Annual Canadian Lynx trappings 1821-1934
mdeaths (UKLungDeaths)
                          Monthly Deaths from Lung Diseases in the UK
morley                    Michelson Speed of Light Data
mtcars                    Motor Trend Car Road Tests
nhtemp                    Average Yearly Temperatures in New Haven
nottem                    Average Monthly Temperatures at Nottingham,
                          1920-1939
npk                       Classical N, P, K Factorial Experiment
occupationalStatus        Occupational Status of Fathers and their
                          Sons
precip                    Annual Precipitation in US Cities
presidents                Quarterly Approval Ratings of US Presidents
pressure                  Vapor Pressure of Mercury as a Function of
                          Temperature
quakes                    Locations of Earthquakes off Fiji
randu                     Random Numbers from Congruential Generator
                          RANDU
rivers                    Lengths of Major North American Rivers
rock                      Measurements on Petroleum Rock Samples
sleep                     Student's Sleep Data
stack.loss (stackloss)
                          Brownlee's Stack Loss Plant Data
stack.x (stackloss)       Brownlee's Stack Loss Plant Data
stackloss                 Brownlee's Stack Loss Plant Data
state.abb (state)         US State Facts and Figures
state.area (state)        US State Facts and Figures
state.center (state)
                          US State Facts and Figures
state.division (state)
                          US State Facts and Figures
```

Once you know where to look, you can then explore lots of these datasets. For example, we can take a look at the built-in `pressure` dataset, which includes 19 values showing the relationship between temperature in degrees Celsius and pressure in mm (or mercury). To "see" this built-in data object, just type the name `pressure` to see (or print out) the object.

```
pressure
```

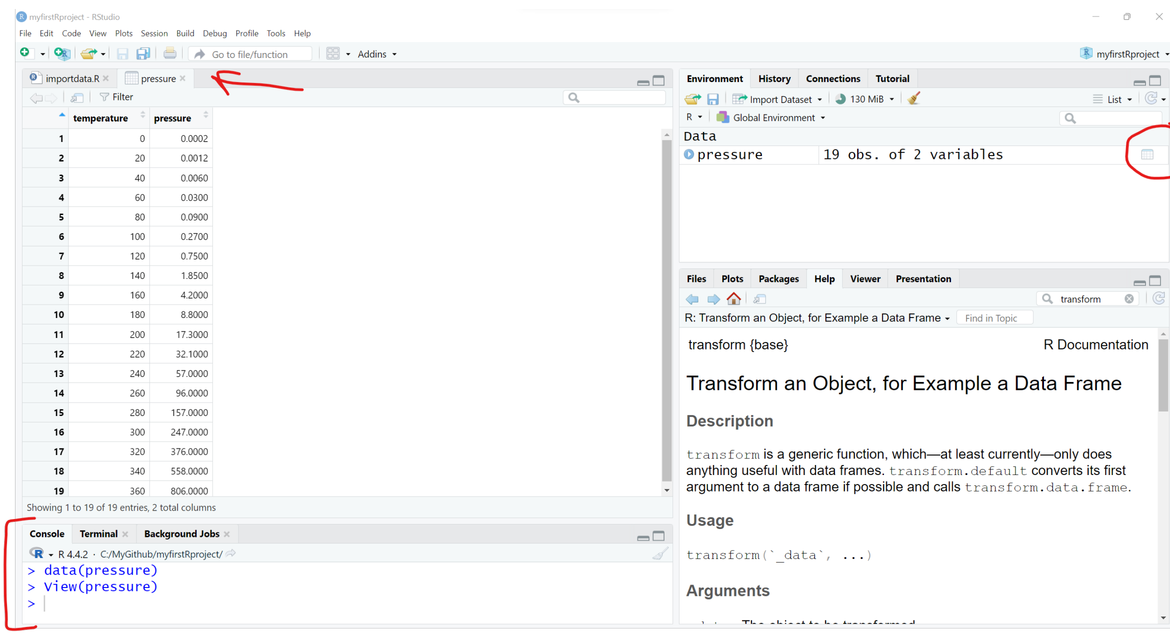
```
   temperature pressure
1            0   0.0002
2           20   0.0012
3           40   0.0060
4           60   0.0300
5           80   0.0900
6          100   0.2700
7          120   0.7500
8          140   1.8500
9          160   4.2000
10         180   8.8000
11         200  17.3000
12         220  32.1000
13         240  57.0000
14         260  96.0000
15         280 157.0000
16         300 247.0000
17         320 376.0000
18         340 558.0000
19         360 806.0000
```

Normally most datasets are much larger than this little dataset. So, I would not advise trying to view most datasets by printing them to the "Console" window pane. Instead you can either click on the object in your "Global Environment" to view it - or you can run the `View()` function to open the viewer window.

You can "load" the built-in `pressure` dataset using the `data(pressure)` function to load the `pressure` dataset to load into your "Global Environment", which loads the dataset into your R session.

If we click on the little “Table icon” all the way to the right of the pressure dataset in the “Global Environment” window - or run `View(pressure)` - we can open the dataset in the Viewer window:

```
data(pressure)
View(pressure)
```



Explore Datasets in R Packages

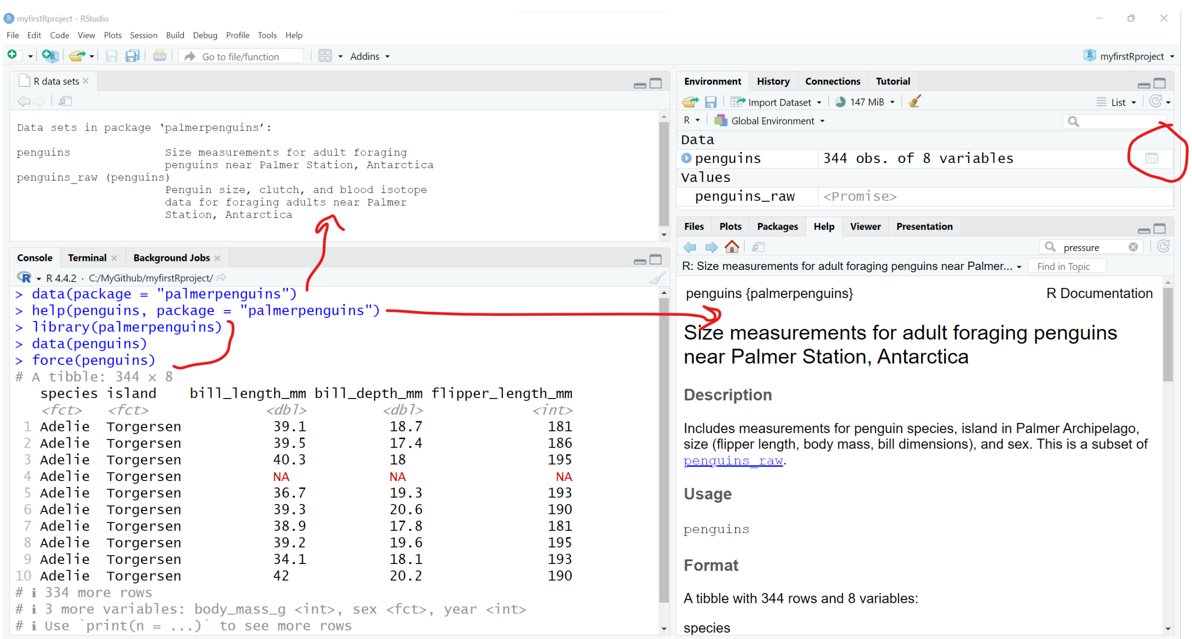
I encourage you to use the `data(package = "xxx")` function to see what, if any, datasets may be built-in to the various packages you may install and load during your R computing sessions.

If you are interested in seeing other datasets in other R packages, go ahead and install the palmerpenguins package and take a look at the **penguins** dataset included:

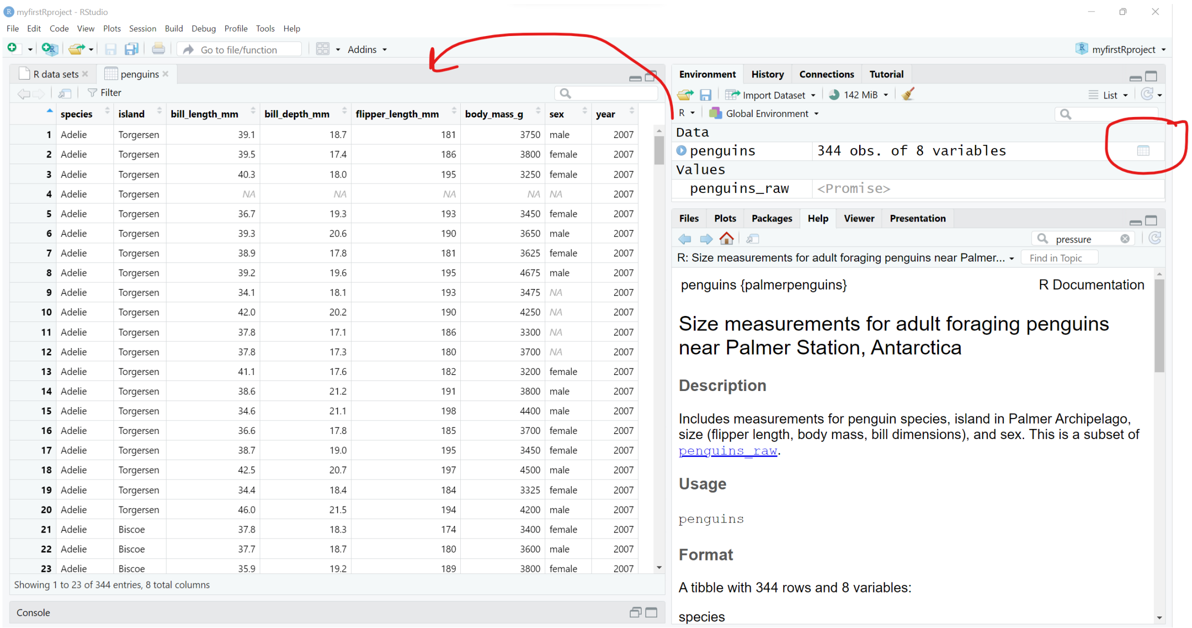
```
# look at datasets included with the
# palmerpenguins dataset
data(package = "palmerpenguins")
```

You can learn more about the **penguins** dataset, by opening up the "Help" page for the dataset. You can also load the **palmerpenguins** package and then load the **penguins** dataset using this code.

```
help(penguins, package = "palmerpenguins")
library(palmerpenguins)
data(penguins)
```



And clicking the the little data table icon after loading the **penguins** dataset into the "Global Environment", you can see the dataset in the viewer window.



2. To view The Data.

Look at small data in Console

Let's work with the `mydata` dataset that we imported above using the `readr::read_csv()` function.

```
# import the mydata.csv dataset
mydata <- readr::read_csv("mydata.csv")
```

This is not a very large dataset - `mydata` has 21 rows (or observations) and 14 variables (or columns). So, we can view the whole thing by printing it to the "Console" window.

You'll notice that depending on the size of your current "Console" window, font size, zoom settings and more, what you see may vary. Since we read this dataset in using the `readr` package, the data object is now a "tibble" dataframe which only shows the columns and rows that will reasonably show up in your "Console" window.

> **What is a "tibble" `tbl_df`?**
>
> As stated on the homepage for the `tibble` package at https://tibble.tidyverse.org/, a "tibble" is
>
> > "… a modern reimagining of the data.frame, keeping what time has proven to be effective, and throwing out what is not."
>
> Also a "tibble" has
>
> > "… an enhanced print() method which makes them easier to use with large datasets containing complex objects."

And the output below also lists what kind of column each variable is. For example,

- `Age` is a `<dbl>` indicating it is a numeric variable saved using double-precision, whereas
- `GenderSTR` is `<chr>` indicating this is a text or character (or "string") type variable.

```
# print the dataset into the Console
mydata
```

```
# A tibble: 21 x 14
   SubjectID   Age WeightPRE WeightPOST Height   SES GenderSTR GenderCoded
   q1
```

```
      <dbl> <dbl>     <dbl>      <dbl>  <dbl> <dbl> <chr>           <dbl>
      <dbl>
 1        1    45        68        145    5.6     9 m                   1
 4
 2        2    50       167        166    5.4     2 f                   2
 3
 3        3    35       143        135    5.6     2 <NA>               NA
 3
 4        4    44       216        201    5.6     2 m                   1
 4
 5        5    32       243        223    6       2 m                   1
 5
 6        6    48       165        145    5.2     2 f                   2
 2
 7        8    50        60        132    3.3     2 m                   1
 3
 8        9    51       110        108    5.1     3 f                   2
 1
 9       12    46       167        158    5.5     2 F                   2
 1
10       14    35       190        200    5.8     1 Male                1
4
# i 11 more rows
# i 5 more variables: q2 <dbl>, q3 <dbl>, q4 <dbl>, q5 <dbl>, q6 <dbl>
```

Look the "structure" of the dataset

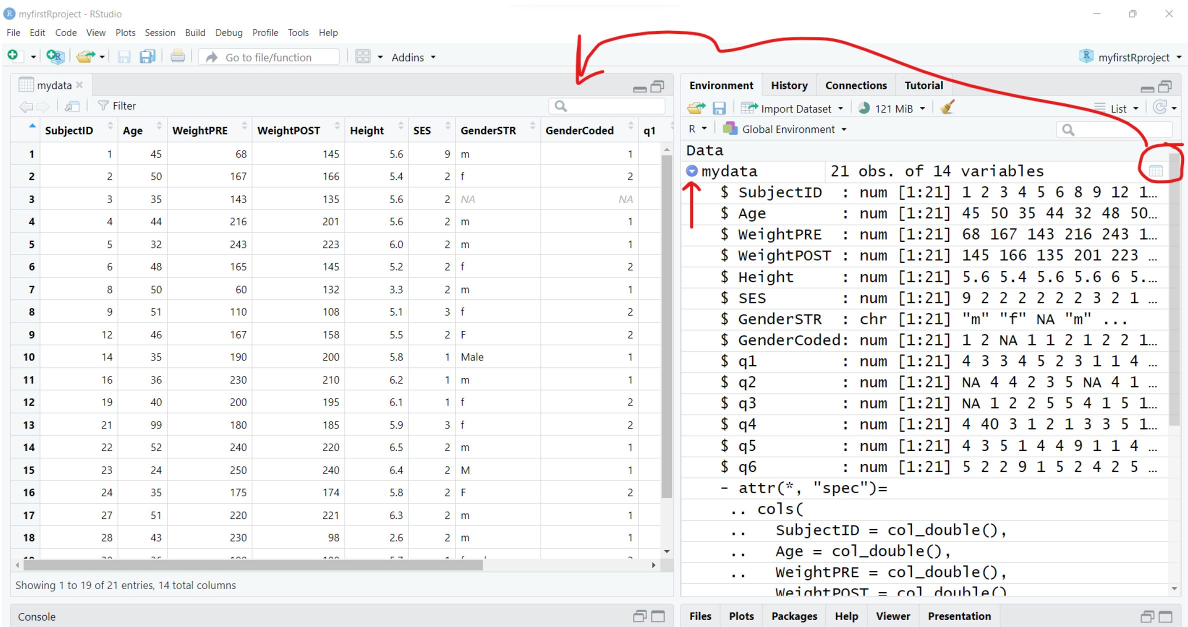
You can also view the different kinds of variables in the dataset using the `str()` or "structure" function - which lists the type of variable, the number of elements in each column [1:21] indicates each column has 21 elements (or 21 rows) and the other values are a quick "peek" at the data inside the dataset. For example, the first 3 people in this dataset are ages 45, 50 and 35.

```
str(mydata)
```

```
spc_tbl_ [21 x 14] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ SubjectID  : num [1:21] 1 2 3 4 5 6 8 9 12 14 ...
 $ Age        : num [1:21] 45 50 35 44 32 48 50 51 46 35 ...
 $ WeightPRE  : num [1:21] 68 167 143 216 243 165 60 110 167 190 ...
 $ WeightPOST : num [1:21] 145 166 135 201 223 145 132 108 158 200 ...
 $ Height     : num [1:21] 5.6 5.4 5.6 5.6 6 5.2 3.3 5.1 5.5 5.8 ...
 $ SES        : num [1:21] 9 2 2 2 2 2 2 3 2 1 ...
 $ GenderSTR  : chr [1:21] "m" "f" NA "m" ...
 $ GenderCoded: num [1:21] 1 2 NA 1 1 2 1 2 2 1 ...
 $ q1         : num [1:21] 4 3 3 4 5 2 3 1 1 4 ...
 $ q2         : num [1:21] NA 4 4 2 3 5 NA 4 1 44 ...
 $ q3         : num [1:21] NA 1 2 2 5 5 4 1 5 1 ...
 $ q4         : num [1:21] 4 40 3 1 2 1 3 3 5 1 ...
 $ q5         : num [1:21] 4 3 5 1 4 4 9 1 1 4 ...
 $ q6         : num [1:21] 5 2 2 9 1 5 2 4 2 5 ...
 - attr(*, "spec")=
  .. cols(
  ..   SubjectID = col_double(),
  ..   Age = col_double(),
  ..   WeightPRE = col_double(),
  ..   WeightPOST = col_double(),
  ..   Height = col_double(),
  ..   SES = col_double(),
  ..   GenderSTR = col_character(),
  ..   GenderCoded = col_double(),
  ..   q1 = col_double(),
  ..   q2 = col_double(),
  ..   q3 = col_double(),
  ..   q4 = col_double(),
  ..   q5 = col_double(),
  ..   q6 = col_double()
  .. )
 - attr(*, "problems")=<externalptr>
```

You can also interactively **`View`** the data by clicking on the data icon and you can also click the little "table" icon to the far right next to the dataset in the "Global Environment"to open the data viewer window on the left.

You can also click on the little blue circle to the left of the **`mydata`** dataset to change the arrow from facing right to facing down to see the "structure" of the data in the "Global Environment".



3. To subset the data - select and filter.

Using base R packages and functions

View parts of the dataset

Now let's "explore" the data by viewing sections of it.

Using base R commands, we can use functions like `head()` and `tail()` with each showing either the top or bottom 6 rows of the dataset. We can add a number to the function call to see more or less rows if we wish.

```
# look at top 6 rows of data
head(mydata)
```

```
# A tibble: 6 x 14
  SubjectID   Age WeightPRE WeightPOST Height   SES GenderSTR GenderCoded
  q1
      <dbl> <dbl>     <dbl>      <dbl>  <dbl> <dbl> <chr>           <dbl>
      <dbl>
1         1    45        68        145    5.6     9 m                   1
4
2         2    50       167        166    5.4     2 f                   2
3
3         3    35       143        135    5.6     2 <NA>               NA
3
4         4    44       216        201    5.6     2 m                   1
4
5         5    32       243        223    6       2 m                   1
5
6         6    48       165        145    5.2     2 f                   2
2
# i 5 more variables: q2 <dbl>, q3 <dbl>, q4 <dbl>, q5 <dbl>, q6 <dbl>
```

```
# look at the bottom 10 rows of data
tail(mydata, n=10)
```

```
# A tibble: 10 x 14
   SubjectID   Age WeightPRE WeightPOST Height   SES GenderSTR GenderCoded
   q1
       <dbl> <dbl>     <dbl>      <dbl>  <dbl> <dbl> <chr>           <dbl>
       <dbl>
```

```
1          19    40       200        195    6.1      1 f                 2
1
2          21    99       180        185    5.9      3 f                 2
2
3          22    52       240        220    6.5      2 m                 1
2
4          23    24       250        240    6.4      2 M                 1
5
5          24    35       175        174    5.8      2 F                 2
5
6          27    51       220        221    6.3      2 m                 1
4
7          28    43       230         98    2.6      2 m                 1
11
8          30    36       190        180    5.7      1 female            2
5
9          32    44       260        109    6.4      3 male              1
1
10         NA    NA        NA         NA    NA      NA <NA>              NA
NA
# i 5 more variables: q2 <dbl>, q3 <dbl>, q4 <dbl>, q5 <dbl>, q6 <dbl>
```

What are these wierd NAs?

The NA letters that show up is how R stores missing data. If the dataset you import has a blank cell (for either numeric or character type data), then R interprets that as "not available" which is indicated by NA. NA is a reserved word in R specifically set aside for handling missing values.

You can learn more about NA by running:

```
help(NA, package = "base")
```

You can also view different parts of the data by using square brackets [] to select specific rows and columns using [row, column] index indicators.

```
# Select the values in rows 1-4
# and in columns 1-3
mydata[1:4, 1:3]
```

```
# A tibble: 4 x 3
  SubjectID   Age WeightPRE
      <dbl> <dbl>     <dbl>
```

```
1          1    45         68
2          2    50        167
3          3    35        143
4          4    44        216
```

To select all of a given row or column just leave that index blank.

```
# show all of rows 1-2
mydata[1:2, ]
```

```
# A tibble: 2 x 14
  SubjectID   Age WeightPRE WeightPOST Height   SES GenderSTR GenderCoded
  q1
      <dbl> <dbl>     <dbl>      <dbl>  <dbl> <dbl> <chr>           <dbl>
      <dbl>
1         1    45        68        145    5.6     9 m                   1
4
2         2    50       167        166    5.4     2 f                   2
3
# i 5 more variables: q2 <dbl>, q3 <dbl>, q4 <dbl>, q5 <dbl>, q6 <dbl>
```

```
# show all of columns 3-4
mydata[ ,3:4]
```

```
# A tibble: 21 x 2
   WeightPRE WeightPOST
       <dbl>      <dbl>
 1        68        145
 2       167        166
 3       143        135
 4       216        201
 5       243        223
 6       165        145
 7        60        132
 8       110        108
 9       167        158
10       190        200
# i 11 more rows
```

View variables in dataset by name

We can also select columns from a dataset using the variable (or column) name. To see the names of all of the variables in a dataset, use the `names()` function.

```
# list variable names in mydata
names(mydata)
```

```
 [1] "SubjectID"   "Age"         "WeightPRE"   "WeightPOST"  "Height"
 [6] "SES"         "GenderSTR"   "GenderCoded" "q1"          "q2"
[11] "q3"          "q4"          "q5"          "q6"
```

We can use the `$` "dollar sign" operator to "select" named variables out of a dataset. Let's look at all of the ages in `mydata`.

```
# look at all of the ages
# of the 21 people in mydata
mydata$Age
```

```
 [1] 45 50 35 44 32 48 50 51 46 35 36 40 99 52 24 35 51 43 36 44 NA
```

We can also use these variable names with the `[]` brackets in base R syntax. And we use the `c()` combine function to help us put a list together. Let's look at the 2 weight columns in the dataset. Put the variable names inside `""` double quotes.

```
# show all rows for
# the 2 weight variables in mydata
mydata[ , c("WeightPRE", "WeightPOST")]
```

```
# A tibble: 21 x 2
   WeightPRE WeightPOST
       <dbl>      <dbl>
 1        68        145
 2       167        166
 3       143        135
 4       216        201
 5       243        223
 6       165        145
 7        60        132
 8       110        108
 9       167        158
10       190        200
# i 11 more rows
```

Using `dplyr` functions

Using `tidyverse` packages and functions

As you can see while base R is very powerful on it's own, the syntax is less than intuitive. There is a whole suite of R packages that are designed to work together and use a different syntax that improves programming workflow and readability.

Learn more about the suite of `tidyverse` packages. You've already used two of these, `readr` and `haven` are both part of `tidyverse` for importing datasets.

Another one of these `tidyverse` packages, `dplyr` is a very good package for "data wrangling".

Pick columns using `dplyr::select()`

Instead of using the base R `$` selector, the `dplyr` package has a `select()` function where you simply choose variables using their name. Let's look at `Height` and `q1` from the `mydata` dataset.

> Using package::function() syntax
>
> It is good coding practice, especially when loading several packages at once into your computing session, to make sure you are calling the exact function you want from a specific package. So, I'm using the syntax of `package::function()` to help keep track of which package and which function is being used below.

```
# load dplyr package
library(dplyr)

# select Height and q1 from mydata
dplyr::select(mydata, c(Height, q1))
```

```
# A tibble: 21 x 2
   Height    q1
    <dbl> <dbl>
 1    5.6     4
 2    5.4     3
 3    5.6     3
 4    5.6     4
 5    6       5
 6    5.2     2
 7    3.3     3
 8    5.1     1
```

```
 9    5.5     1
10    5.8     4
# i 11 more rows
```

Workflow using the pipe %>% operator

Another improvement of the `tidyverse` approach of R programming is to use the pipe `%>%` operator. Basically what this syntax does is take the results from "A" and pipe it into –> the next "B" function, e.g. `A %>% B` so we can begin to "daisy-chain" a sequence of programming steps together into a logical workflow that is easy to "read" and follow.

Here is a working example to show the same variable selection process we did above, but now we will be using the `dplyr::select()` function. The code below takes the `mydata` dataset and pipes `%>%` it into the `select()` function. We were also able to drop using the `c()` function here.

```r
# start with mydata and then
# select Height and q1 from mydata
mydata %>% dplyr::select(Height, q1)
```

```
# A tibble: 21 x 2
   Height    q1
    <dbl> <dbl>
 1    5.6     4
 2    5.4     3
 3    5.6     3
 4    5.6     4
 5    6       5
 6    5.2     2
 7    3.3     3
 8    5.1     1
 9    5.5     1
10    5.8     4
# i 11 more rows
```

We could even add the base R `head()` function here. If we put each code step on a separate line, you can now see that we are [1] taking the `mydata` dataset "and then" [2] selecting 2 variables "and then" [3] looking at the top 6 rows of the dataset.

```r
# select Height and q1 from mydata
# and show only the top 6 rows
mydata %>%
  dplyr::select(Height, q1) %>%
```

```
  head()
```

```
# A tibble: 6 x 2
  Height    q1
   <dbl> <dbl>
1    5.6     4
2    5.4     3
3    5.6     3
4    5.6     4
5    6       5
6    5.2     2
```

TL;DR If `%>%` is a pipe, then what is `|>`??

The `%>%` pipe operator is implemented within `tidyverse` from the `magrittr` package which is used by the `tidyverse` packages which started being used quite extensively by R programmers over the last decade.

However, the rest of the R development community *(which is much larger than just those who use the `tidyverse` suite)* also recently added a new base R pipe operator `|>` (since R version 4.1.0).

Learn more in this `tidyverse` blog post from 2023

So, you do have the option to also use the base R `|>` pipe operator.

```
# select Height and q1 from mydata
# and show only the top 6 rows
mydata |>
  dplyr::select(Height, q1) |>
  head()
```

```
# A tibble: 6 x 2
  Height    q1
   <dbl> <dbl>
1    5.6     4
2    5.4     3
3    5.6     3
4    5.6     4
5    6       5
6    5.2     2
```

For now, we will stay with the `%>%` operator for consistency. But be aware that you will see both approaches on the Internet when "Googling" for answers.

Select variables with matching using `starts_with()`

When using `dplyr::select()` to select variables, there are several "helper functions" that are useful for "selection". You can see a list of these functions by running `help("starts_with", package = "tidyselect")`. These "selection helper" functions are actually in the `tidyselect` package which is loaded with the `dplyr` package.

Let's use these functions to pull out all of the Likert-scaled "question" variables that start with the letter `"q"`.

```
mydata %>%
  dplyr::select(starts_with("q"))
```

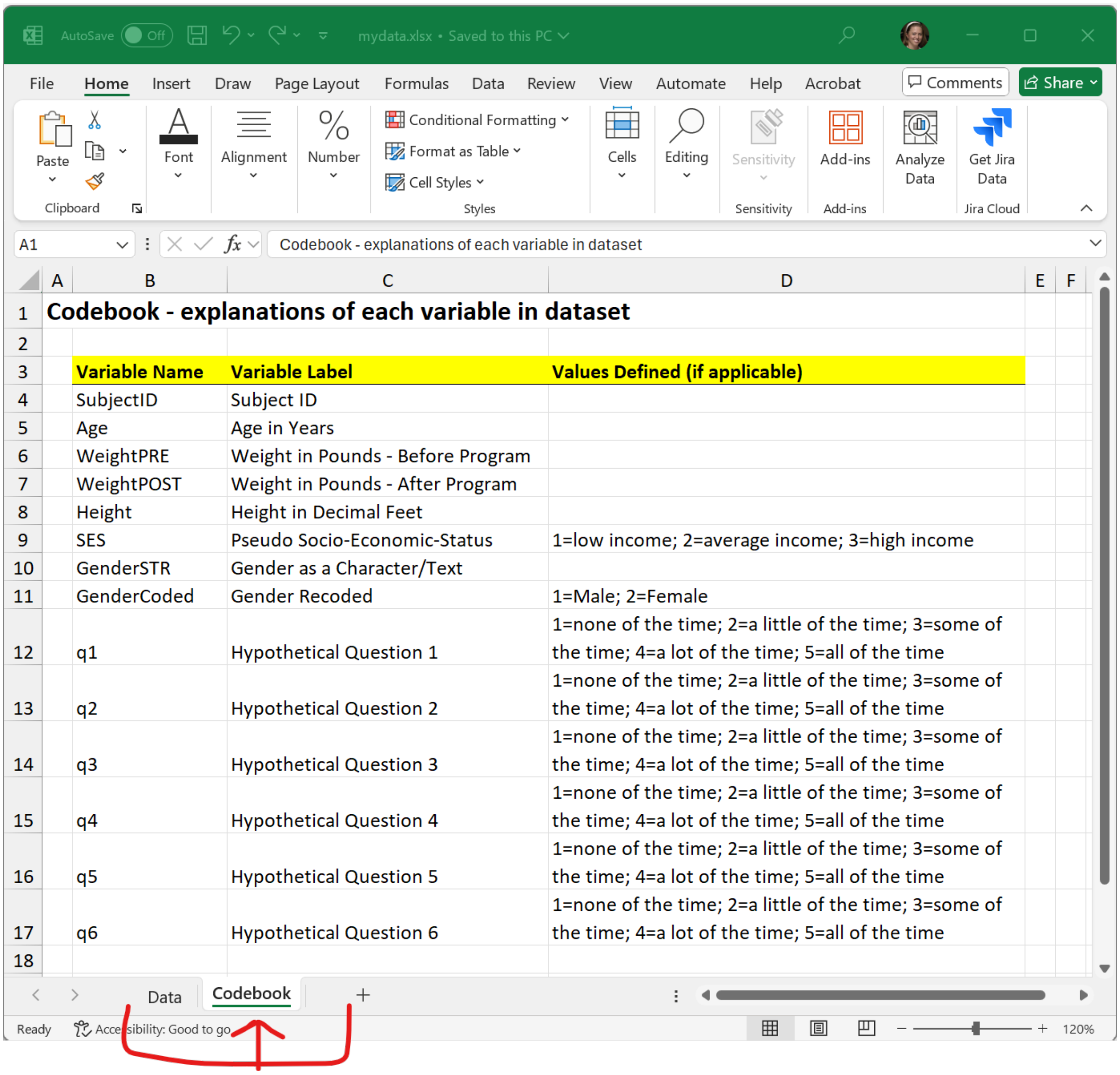
```
# A tibble: 21 x 6
      q1    q2    q3    q4    q5    q6
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1     4    NA    NA     4     4     5
 2     3     4     1    40     3     2
 3     3     4     2     3     5     2
 4     4     2     2     1     1     9
 5     5     3     5     2     4     1
 6     2     5     5     1     4     5
 7     3    NA     4     3     9     2
 8     1     4     1     3     1     4
 9     1     1     5     5     1     2
10     4    44     1     1     4     5
# i 11 more rows
```


Pick rows using `dplyr::filter()`

In addition to selecting columns or variables from your dataset, you can also pull out a subset of your data by "filtering" out only the rows you want.

For example, suppose we only want to look at the `Age`, `WeightPRE` for the Females in the dataset indicates by `GenderCoded` equal to 2.

For reference, take a look at the `mydata` codebook - and here is a screenshot as well:



Notice that:

- I changed the order of the columns, which is OK,
- and to filter out and KEEP only the rows for females, I typed `GenderCoded == 2` using two equal signs `==`. R uses two `==` equal signs to perform a logical operation to ask does the variable `GenderCoded` equal the value of 2, with either a `TRUE` or `FALSE` result. Only the rows with a `TRUE` result are shown.

> **Be careful not to mix up `=` and `==`**
>
> Odds are you will get errors at some point due to typos or other issues, but a common error is to use a single `=` equals sign when trying to perform a logic operation. Remember to use 2 equals signs `==` if you are trying to perform a `TRUE/FALSE` operation and use only 1 equals sign `=` when assigning a value to a function argument.

```r
# select columns from mydata
# and then only show rows for females
mydata %>%
  select(GenderCoded, Age, WeightPRE) %>%
  filter(GenderCoded == 2)
```

```
# A tibble: 8 x 3
  GenderCoded   Age WeightPRE
        <dbl> <dbl>     <dbl>
1           2    50       167
2           2    48       165
3           2    51       110
4           2    46       167
5           2    40       200
6           2    99       180
7           2    35       175
8           2    36       190
```

Here is an example of the error you will get if you use a single `=` sign instead of `==` two.

```r
# select columns from mydata
# and then only show rows for females
mydata %>%
  select(GenderCoded, Age, WeightPRE) %>%
  filter(GenderCoded = 2)
```

```
Error in `filter()`:
! We detected a named input.
```

```
  This usually means that you've used `=` instead of `==`.
  Did you mean `GenderCoded == 2`?
Run `rlang::last_trace()` to see where the error occurred.
```

Filter rows using matching `%in%` operator

Another helpful operator in R is the `%in%` operator used for matching. Let's suppose we wanted to pull out the rows for specific subject IDs - perhaps you want to review only these records.

Let's pull out the data for only IDs 14, 21 and 24. Rather than writing a complicated if-then-else set of code steps, we can search for these IDs and only the rows with these IDs will be kept.

```
mydata %>%
  filter(SubjectID %in% c(14, 21, 24))
```

```
# A tibble: 3 x 14
  SubjectID   Age WeightPRE WeightPOST Height   SES GenderSTR GenderCoded
  q1
      <dbl> <dbl>     <dbl>      <dbl>  <dbl> <dbl> <chr>           <dbl>
      <dbl>
1        14    35       190        200    5.8     1 Male                1
4
2        21    99       180        185    5.9     3 f                   2
2
3        24    35       175        174    5.8     2 F                   2
5
# i 5 more variables: q2 <dbl>, q3 <dbl>, q4 <dbl>, q5 <dbl>, q6 <dbl>
```

Sort/arrange rows using `dplyr::arrange()`

Here is another helpful function from `dplyr`. Suppose we want to find the 5 oldest people in `mydata` and show their IDs.

Let's use the `dplyr::arrange()` function which will sort our data based on the variable we specify in increasing order (lowest to highest) by default. We will add the `desc()` function to sort decreasing from largest to smallest.

Learn more by running `help(arrange, package = "dplyr")`

Note: There was someone with age 99 in this made-up dataset.

```
# take mydata
# select SubjectID and Age
# sort descending by Age
# show the top 5 IDs and Ages
mydata %>%
  select(SubjectID, Age) %>%
  arrange(desc(Age)) %>%
```

```r
head(n=5)
```

```
# A tibble: 5 x 2
  SubjectID   Age
      <dbl> <dbl>
1        21    99
2        22    52
3         9    51
4        27    51
5         2    50
```

The oldest people are subject IDs 21, 22, 9, 27 and 2 who are age 99, 52, 51, 51 and 50 years old respectively.

---

4. To create and modify variables.

To create and add new variables to the dataset, we can use either a base R approach or use the `mutate()` function from the `dplyr` package. Let's take a look at both approaches. In the `mydata` dataset, we have `Height` in decimal feet and we have `WeightPRE` and `WeightPOST` in pounds.

So, let's compute BMI (body mass index) as follows from Height (in inches) and Weight (in pounds):

$$BMI = \left(\frac{weight_{(lbs)}}{(height_{(inches)})^2} \right) * 703$$

Create New Variable - Base R Approach

Create a new variable using the $ selector operator. Then write out the mathematical equation. I also had to multiply the height in decimal feet * 12 to get inches.

```r
# Compute BMI for the PRE Weight
mydata$bmiPRE <-
  (mydata$WeightPRE * 703) / (mydata$Height * 12)^2

# look at result
mydata$bmiPRE
```

```
 [1]  10.58585  27.95901  22.26142  33.62564  32.95312  29.78997  26.89777
 [8]  20.64644  26.95156  27.57341  29.21039  26.23996  25.24418  27.73176
[15]  29.79702  25.39656  27.06041 166.10166  28.54938  30.98891        NA
```

Look at the "Global Environment" or run the `str()` function to see if a new variable was added to `mydata` - which should now have 15 variables instead of only 14.

You can also list the variable names in the updated dataset.

```r
# look at updated data structure
str(mydata)
```

```
spc_tbl_ [21 x 15] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ SubjectID  : num [1:21] 1 2 3 4 5 6 8 9 12 14 ...
 $ Age        : num [1:21] 45 50 35 44 32 48 50 51 46 35 ...
 $ WeightPRE  : num [1:21] 68 167 143 216 243 165 60 110 167 190 ...
 $ WeightPOST : num [1:21] 145 166 135 201 223 145 132 108 158 200 ...
```

```
 $ Height     : num [1:21] 5.6 5.4 5.6 5.6 6 5.2 3.3 5.1 5.5 5.8 ...
 $ SES        : num [1:21] 9 2 2 2 2 2 2 3 2 1 ...
 $ GenderSTR  : chr [1:21] "m" "f" NA "m" ...
 $ GenderCoded: num [1:21] 1 2 NA 1 1 2 1 2 2 1 ...
 $ q1         : num [1:21] 4 3 3 4 5 2 3 1 1 4 ...
 $ q2         : num [1:21] NA 4 4 2 3 5 NA 4 1 44 ...
 $ q3         : num [1:21] NA 1 2 2 5 5 4 1 5 1 ...
 $ q4         : num [1:21] 4 40 3 1 2 1 3 3 5 1 ...
 $ q5         : num [1:21] 4 3 5 1 4 4 9 1 1 4 ...
 $ q6         : num [1:21] 5 2 2 9 1 5 2 4 2 5 ...
 $ bmiPRE     : num [1:21] 10.6 28 22.3 33.6 33 ...
 - attr(*, "spec")=
  .. cols(
  ..   SubjectID = col_double(),
  ..   Age = col_double(),
  ..   WeightPRE = col_double(),
  ..   WeightPOST = col_double(),
  ..   Height = col_double(),
  ..   SES = col_double(),
  ..   GenderSTR = col_character(),
  ..   GenderCoded = col_double(),
  ..   q1 = col_double(),
  ..   q2 = col_double(),
  ..   q3 = col_double(),
  ..   q4 = col_double(),
  ..   q5 = col_double(),
  ..   q6 = col_double()
  .. )
 - attr(*, "problems")=<externalptr>
```

```
# list the variable names in the
# updated dataset
names(mydata)
```

```
 [1] "SubjectID"   "Age"         "WeightPRE"   "WeightPOST"  "Height"
 [6] "SES"         "GenderSTR"   "GenderCoded" "q1"          "q2"
[11] "q3"          "q4"          "q5"          "q6"          "bmiPRE"
```

Create New Variable - `dplyr::mutate()` Approach

In the `dplyr` package, you can create or modify variables using the `mutate()` function.

```r
# Compute BMI for the POST Weight
# use the dplyr::mutate() function
mydata <- mydata %>%
  mutate(
    bmiPOST = (WeightPOST * 703) / (Height * 12)^2
    )

# check updates
str(mydata)
```

```
tibble [21 x 16] (S3: tbl_df/tbl/data.frame)
 $ SubjectID  : num [1:21] 1 2 3 4 5 6 8 9 12 14 ...
 $ Age        : num [1:21] 45 50 35 44 32 48 50 51 46 35 ...
 $ WeightPRE  : num [1:21] 68 167 143 216 243 165 60 110 167 190 ...
 $ WeightPOST : num [1:21] 145 166 135 201 223 145 132 108 158 200 ...
 $ Height     : num [1:21] 5.6 5.4 5.6 5.6 6 5.2 3.3 5.1 5.5 5.8 ...
 $ SES        : num [1:21] 9 2 2 2 2 2 2 3 2 1 ...
 $ GenderSTR  : chr [1:21] "m" "f" NA "m" ...
 $ GenderCoded: num [1:21] 1 2 NA 1 1 2 1 2 2 1 ...
 $ q1         : num [1:21] 4 3 3 4 5 2 3 1 1 4 ...
 $ q2         : num [1:21] NA 4 4 2 3 5 NA 4 1 44 ...
 $ q3         : num [1:21] NA 1 2 2 5 5 4 1 5 1 ...
 $ q4         : num [1:21] 4 40 3 1 2 1 3 3 5 1 ...
 $ q5         : num [1:21] 4 3 5 1 4 4 9 1 1 4 ...
 $ q6         : num [1:21] 5 2 2 9 1 5 2 4 2 5 ...
 $ bmiPRE     : num [1:21] 10.6 28 22.3 33.6 33 ...
 $ bmiPOST    : num [1:21] 22.6 27.8 21 31.3 30.2 ...
```

```r
names(mydata)
```

```
 [1] "SubjectID"   "Age"         "WeightPRE"   "WeightPOST"  "Height"
 [6] "SES"         "GenderSTR"   "GenderCoded" "q1"          "q2"
[11] "q3"          "q4"          "q5"          "q6"          "bmiPRE"
[16] "bmiPOST"
```

Create New Variable - add labels to codes by creating a "factor" type variable

As you probably noticed in the views of the `mydata` dataset above, there was originally a variable where people were allowed to enter their gender using free text (the `GenderSTR` variable). There were entries like "f", "F", "female", "male", "Male" and other variations. So, another variable `GenderCoded` was included where 1=male and 2=female, but when we look at `mydata$GenderCoded` all we see are 1's and 2's and `NA`s.

```
mydata$GenderCoded
```

```
 [1]  1  2 NA  1  1  2  1  2  2  1  1  2  2  1  1  2  1  1  2  1 NA
```

It would be nice if we could add some labels. One way to do this is to convert `GenderCoded` from being a simple "numeric" variable to a new object class called a "factor" which includes both numeric values and text labels.

Here is the base R approach to create a new factor type variable. Learn more by looking at the help page for `factor()`, run `help(factor, package = "base")`.

```
# create a new factor with labels
mydata$GenderCoded.f <-
  factor(mydata$GenderCoded,
         levels = c(1, 2),
         labels = c("Male", "Female"))

# look at new variable
mydata$GenderCoded.f
```

```
 [1] Male   Female <NA>   Male   Male   Female Male   Female Female Male
[11] Male   Female Female Male   Male   Female Male   Male   Female Male
[21] <NA>
Levels: Male Female
```

We can check the type each variable using the `class()` function.

```
class(mydata$GenderCoded)
```

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

```
class(mydata$GenderCoded.f)
```

```
[1] "factor"
```

Another quick way to see these `class` type differences is to use the `table()` function to get the frequencies of each distinct value. I'm also adding the `useNA = "ifany"` option to also get a count of any missing values. Learn more by running `help(table, package = "base")`.

```
# table of frequencies of GenderCoded - numeric class
table(mydata$GenderCoded, useNA = "ifany")
```

```

   1    2 <NA> 
  11    8    2 
```

```
# table of GenderCoded.f - factor class
table(mydata$GenderCoded.f, useNA = "ifany")
```

```

  Male Female   <NA> 
    11      8      2 
```

---

5. To get data summary and descriptive statistics.

Getting summary statistics

summary() function

One of the best functions that is part of base R is the `summary()` function. Let's see what this gives us for the `mydata` dataset.

As you can see for all of the `numeric` class variables, the `summary()` function gives us the min, max, median, mean, 1st quartile and 3rd quartile and a count of the the number of missing `NA`s. So, you can see the mean `Age` is 44.8 and the median `Age` is 44.0.

For the character variable `GenderSTR` all we know is it has a length of 21.

But for the `factor` type variable `GenderCoded.f` we get the number of Males, Females and `NA`s.

```
summary(mydata)
```

```
   SubjectID          Age          WeightPRE        WeightPOST
 Min.   : 1.00   Min.   :24.00   Min.   : 60.0   Min.   : 98.0
 1st Qu.: 5.75   1st Qu.:35.75   1st Qu.:166.5   1st Qu.:142.5
 Median :15.00   Median :44.00   Median :190.0   Median :177.0
 Mean   :15.30   Mean   :44.80   Mean   :185.2   Mean   :172.2
 3rd Qu.:23.25   3rd Qu.:50.00   3rd Qu.:230.0   3rd Qu.:203.2
 Max.   :32.00   Max.   :99.00   Max.   :260.0   Max.   :240.0
 NA's   :1       NA's   :1       NA's   :1       NA's   :1
     Height           SES         GenderSTR          GenderCoded
 Min.   :2.600   Min.   :1.0   Length:21          Min.   :1.000
 1st Qu.:5.475   1st Qu.:2.0   Class :character   1st Qu.:1.000
 Median :5.750   Median :2.0   Mode  :character   Median :1.000
 Mean   :5.550   Mean   :2.3                      Mean   :1.421
 3rd Qu.:6.125   3rd Qu.:2.0                      3rd Qu.:2.000
 Max.   :6.500   Max.   :9.0                      Max.   :2.000
 NA's   :1       NA's   :1                        NA's   :2
       q1              q2                q3              q4
 Min.   : 1.00   Min.   : 1.000   Min.   :1.00   Min.   : 1.000
 1st Qu.: 1.75   1st Qu.: 2.000   1st Qu.:1.00   1st Qu.: 2.000
 Median : 3.00   Median : 4.000   Median :3.00   Median : 3.000
 Mean   : 3.35   Mean   : 5.526   Mean   :3.15   Mean   : 5.062
 3rd Qu.: 4.25   3rd Qu.: 4.500   3rd Qu.:4.25   3rd Qu.: 4.000
 Max.   :11.00   Max.   :44.000   Max.   :9.00   Max.   :40.000
 NA's   :1       NA's   :2        NA's   :1      NA's   :5
       q5               q6               bmiPRE            bmiPOST
```

```
 Min.   : 1.000   Min.   :1.000   Min.   : 10.59   Min.   :12.99  
 1st Qu.: 2.000   1st Qu.:2.000   1st Qu.: 26.03   1st Qu.:25.38  
 Median : 4.000   Median :4.000   Median : 27.65   Median :26.42  
 Mean   : 9.176   Mean   :3.706   Mean   : 33.78   Mean   :29.43  
 3rd Qu.: 5.000   3rd Qu.:5.000   3rd Qu.: 29.79   3rd Qu.:28.71  
 Max.   :99.000   Max.   :9.000   Max.   :166.10   Max.   :70.77  
 NA's   :4        NA's   :4       NA's   :1        NA's   :1      
 GenderCoded.f
 Male  :11    
 Female: 8    
 NA's  : 2    
```

So, the `summary()` function is helpful, but you'll notice we do not get the standard deviation. For some reason that was left out of the original `summary()` statistics function.

There are a few other descriptive statistics functions that can be useful. There is a `describe()` function in both the `Hmisc` package and the `psych` packages.

Hmisc::describe() function

Let's look at `Hmisc::describe()` for a couple of the variables.

You'll notice that this still doesn't give us the standard deviation, but we get the min, max, mean, median, as well as the `.05` *(5th percentile)* and others, and the output includes a summary of the frequency of the distinct values.

```r
mydata %>%
  select(Age, GenderCoded.f, bmiPRE) %>%
  Hmisc::describe()
```

```
.

 3  Variables      21  Observations
--------------------------------------------------------------------------------
Age 
       n  missing distinct     Info     Mean  pMedian      Gmd      .05 
      20        1       14    0.994     44.8       43    13.81    31.60 
     .10      .25      .50      .75      .90      .95 
   34.70    35.75    44.00    50.00    51.10    54.35 
```

```
Value         24   32   35   36   40   43   44   45   46   48   50   51   52
Frequency      1    1    3    2    1    1    2    1    1    1    2    2    1
Proportion 0.05 0.05 0.15 0.10 0.05 0.05 0.10 0.05 0.05 0.05 0.10 0.10 0.05

Value         99
Frequency      1
Proportion 0.05

For the frequency table, variable is rounded to the nearest 0
--------------------------------------------------------------------------------
GenderCoded.f
       n  missing distinct 
      19        2        2 
                        
Value        Male Female
Frequency      11      8
Proportion  0.579  0.421
--------------------------------------------------------------------------------
bmiPRE 
       n  missing distinct     Info     Mean  pMedian      Gmd      .05 
      20        1       20        1    33.78    27.73    18.52    20.14 
     .10      .25      .50      .75      .90      .95 
   22.10    26.03    27.65    29.79    33.02    40.25 

10.5858489229025 (1, 0.05), 20.6464394036482 (1, 0.05), 22.2614175878685 (1,
0.05), 25.2441827061189 (1, 0.05), 25.3965599815035 (1, 0.05),
26.2399593896503
(1, 0.05), 26.8977655341292 (1, 0.05), 26.9515610651974 (1, 0.05),
27.0604126424232 (1, 0.05), 27.5734079799181 (1, 0.05), 27.7317554240631 (1,
0.05), 27.9590096784027 (1, 0.05), 28.5493827160494 (1, 0.05),
29.2103855937103
(1, 0.05), 29.7899716469428 (1, 0.05), 29.7970241970486 (1, 0.05),
30.9889051649305 (1, 0.05), 32.953125 (1, 0.05), 33.6256377551021 (1, 0.05),
166.101660092045 (1, 0.05)

For the frequency table, variable is rounded to the nearest 0
--------------------------------------------------------------------------------
```

psych::describe() function

The `psych::describe()` function only works on numeric data. So, let's look at `Age` and `bmiPRE`. This function now gives us the standard deviation `sd` and even the `mad` which is the mean absolute deviation.

```
mydata %>%
  select(Age, bmiPRE) %>%
  psych::describe()
```

```
       vars  n  mean    sd median trimmed   mad   min   max  range skew
Age       1 20 44.80 14.87  44.00   43.06 10.38 24.00  99.0  75.00 2.21
bmiPRE    2 20 33.78 31.53  27.65   27.79  3.17 10.59 166.1 155.52 3.66
       kurtosis   se
Age        6.10 3.32
bmiPRE    12.53 7.05
```

Base R specific statistics functions

There are many built-in functions in base R for computing specific statistics like `mean()`, `sd()` for standard deviation, `median()`, `min()`, `max()` and `quantile()` to get specific percentiles.

Let get some summary statistics for different variables in `mydata`.

```
# get min, max for Age
min(mydata$Age)
```

```
[1] NA
```

```
max(mydata$Age)
```

```
[1] NA
```

WAIT!? - why did I get `NA`? Since there is missing data in this dataset, we need to tell these R functions how to handle the missing data. We need to add `na.rm=TRUE` to remove the `NA`s and then compute the `min()` and `max()` for the non-missing values.

```
min(mydata$Age, na.rm = TRUE)
```

```
[1] 24
```

```
max(mydata$Age, na.rm = TRUE)
```

```
[1] 99
```

If we want, we could get the non-parametric statistics of median (which is the 50th percentile), 25th and 75th percentiles for the interquartile range. Let's get these statistics for `bmiPRE`.

```
# get median bmiPRE
# and 25th and 75th percentiles for bmiPRE
median(mydata$bmiPRE,
       na.rm = TRUE)
```

```
[1] 27.65258
```

```
quantile(mydata$bmiPRE,
         probs = 0.25,
         na.rm = TRUE)
```

```
     25%
26.02911
```

```
quantile(mydata$bmiPRE,
         probs = 0.75,
         na.rm = TRUE)
```

```
     75%
29.79173
```

We can also get the `mean()` and `sd()` for `Height`.

```
mean(mydata$Height, na.rm = TRUE)
```

```
[1] 5.55
```

```
sd(mydata$Height, na.rm = TRUE)
```

```
[1] 0.9795273
```

dplyr::summarize() function

The `dplyr` package also has a `summarize()` function you can use to get specific statistics of your choosing. For example, let's get the `mean()` and `sd()` for `Age` in one code step.

```
mydata %>%
  dplyr::summarise(
    mean_age = mean(Age, na.rm = TRUE),
    sd_age = sd(Age, na.rm = TRUE)
  )
```

```
# A tibble: 1 x 2
  mean_age sd_age
     <dbl>  <dbl>
1     44.8   14.9
```

We can do this same code again but add the `dplyr::group_by()` function to add a grouping variable to get the statistics by.

NOTE: The `dplyr::group_by()` function must come BEFORE `dplyr::summarise()`.

Let's get the summary stats (`mean` and `sd`) for `Age` by `GenderCoded.f`.

```
mydata %>%
  dplyr::group_by(GenderCoded.f) %>%
  dplyr::summarise(
    mean_age = mean(Age, na.rm = TRUE),
    sd_age = sd(Age, na.rm = TRUE)
  )
```

```
# A tibble: 3 x 3
  GenderCoded.f mean_age sd_age
  <fct>            <dbl>  <dbl>
1 Male              41.5   8.77
2 Female            50.6  20.5
3 <NA>              35    NA
```

! Each code step may result in different object classes

As you work through a series of code steps in an analysis or computational workflow, each step may produce an output object with a different class.

Let's look at each step of the code above to produce a table of means and standard deviations of `Age` by `GenderCoded.f`.

STEP 1 - begin with the dataset

We start with the dataset `mydata` which is a "tibble" "data.frame" since we imported the data using one of the `tidyverse` packages: `readr`, `readxl` or `haven` all of which create a `tbl_df` class object.

```
# Step 1
class(mydata)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

STEP 2 - create a "grouped" data.frame

Notice that as soon as we use the `dplyr::group_by()` function, the result is an updated type of "tibble""data.frame" which is now a `grouped_df` class object. This object class is described at https://dplyr.tidyverse.org/articles/grouping.html.

The `grouped_df` is similar to:

- applying the `SPLIT FILE` command in the SPSS software
- or using the `BY` command in SAS to "work with grouped data"

```
# save the output of step 2
step2 <- mydata %>%
  dplyr::group_by(GenderCoded.f)

class(step2)
```

```
[1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

STEP 3 - after the summarise step

After STEP 3, another `tbl_df` is created.

```
step3 <- mydata %>%
  dplyr::group_by(GenderCoded.f) %>%
  dplyr::summarise(
    mean_age = mean(Age, na.rm = TRUE),
    sd_age = sd(Age, na.rm = TRUE)
  )

class(step3)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Since this saved output object `step3` is a `tbl_df`, we can use it like any other "data.frame" object. For example, we can pull out the `mean_age` column:

```
# pull out the mean_age column using $
step3$mean_age
```

```
[1] 41.45455 50.62500 35.00000
```

```
# pull out the sd_age column using select()
step3 %>%
  select(sd_age)
```

```
# A tibble: 3 x 1
  sd_age
   <dbl>
1   8.77
2  20.5
3  NA
```

Make summary tables

Creating nicely formatted summary tables is an active area of development in the R community. So, I'm sure there are new functions and packages that I may not have shown here. But here are a few packages I use often for making tables of summary statistics. Most of these are designed to work within a Rmarkdown document.

`arsenal` package for tables

The `arsenal` package is useful for making tables - especially with Rmarkdown - to be explained further in a later session Module 1.3.6. Learn more at `tableby()` vignette.

Here is a quick example of some summary statistics for `Age`, `bmiPRE`, and `SES` by `GenderCoded.f` using the `tableby()` function.

First let's add labels for `SES` and create a factor variable.

```
mydata$SES.f <-
  factor(mydata$SES,
         levels = c(1, 2, 3),
         labels = c("low income",
                    "average income",
                    "high income"))
```

```
library(arsenal)
tab1 <- tableby(GenderCoded.f ~ Age + bmiPRE +SES.f,
               data = mydata)
summary(tab1)
```

	Male (N=11)	Female (N=8)	Total (N=19)	p value
Age				0.199
Mean (SD)	41.455 (8.768)	50.625 (20.493)	45.316 (15.089)	
Range	24.000 - 52.000	35.000 - 99.000	24.000 - 99.000	
bmiPRE				0.370
Mean (SD)	40.230 (42.193)	26.347 (2.785)	34.384 (32.274)	
Range	10.586 - 166.102	20.646 - 29.790	10.586 - 166.102	
SES.f				0.625
N-Miss	1	0	1	
low income	2 (20.0%)	2 (25.0%)	4 (22.2%)	

	Male (N=11)	Female (N=8)	Total (N=19)	p value
average income	7 (70.0%)	4 (50.0%)	11 (61.1%)	
high income	1 (10.0%)	2 (25.0%)	3 (16.7%)	

`gtsummary` package for tables

The `gtsummary` package is also useful for making tables. We can even use it to make nicely formatted tables in the "Viewer" window pane or in Rmarkdown. Learn more at `tbl_summary()` vignette.

Here is a quick example of some summary statistics for `Age` and `bmiPRE` by `GenderCoded.f` using the `tbl_summary()` function.

```
library(gtsummary)

mydata %>%
  select(Age, bmiPRE, SES.f, GenderCoded.f) %>%
  tbl_summary(by = GenderCoded.f)
```

Table 2

Characteristic	**Male** N = 11[1]	**Female** N = 8[1]
Age	44 (35, 50)	47 (38, 51)
bmiPRE	29 (27, 33)	27 (25, 28)
SES.f		
low income	2 (20%)	2 (25%)
average income	7 (70%)	4 (50%)
high income	1 (10%)	2 (25%)
Unknown	1	0

[1]Median (Q1, Q3); n (%)

`tableone` package for making summary tables

The output produced from `tableone` is simple text output.

```
library(tableone)

tableone::CreateTableOne(
  data = mydata,
  vars = c("Age", "bmiPRE", "SES.f"),
  strata = "GenderCoded.f"
)
```

```
                    Stratified by GenderCoded.f
                     Male          Female        p      test
  n                     11             8
  Age (mean (SD))    41.45 (8.77)  50.62 (20.49)  0.199
  bmiPRE (mean (SD)) 40.23 (42.19) 26.35 (2.79)   0.370
  SES.f (%)                                       0.625
     low income          2 (20.0)      2 (25.0)
     average income      7 (70.0)      4 (50.0)
     high income         1 (10.0)      2 (25.0)
```

gmodels package for R-x-C tables

If we want to look at a "cross-table" similar to output from SPSS or SAS, the `gmodels` package has the `CrossTable()` function that creates text-based tables similar to these other statistics software packages.

Let's get the frequencies and columns percentages for SES by gender. The first variable is the row variable, the second is the column variable.

```r
library(gmodels)

CrossTable(mydata$SES.f,           # row variable
           mydata$GenderCoded.f,   # column variable
           prop.t = FALSE,         # turn off percent of total
           prop.r = FALSE,         # turn off percent of row
           prop.c = TRUE,          # turn on percent of column
           prop.chisq = FALSE,     # turn off percent for chisq test
           format = "SPSS")        # format like SPSS
```

```
   Cell Contents
|-------------------------|
|                   Count |
|          Column Percent |
|-------------------------|

Total Observations in Table:  18

               | mydata$GenderCoded.f
  mydata$SES.f |      Male  |    Female  | Row Total |
---------------|------------|------------|-----------|
    low income |         2  |         2  |        4  |
               |   20.000%  |   25.000%  |           |
---------------|------------|------------|-----------|
average income |         7  |         4  |       11  |
               |   70.000%  |   50.000%  |           |
---------------|------------|------------|-----------|
   high income |         1  |         2  |        3  |
               |   10.000%  |   25.000%  |           |
---------------|------------|------------|-----------|
  Column Total |        10  |         8  |       18  |
               |   55.556%  |   44.444%  |           |
---------------|------------|------------|-----------|
```

Table Inspiration

Making effective, nicely formatted tables from R and Rmarkdown has been an active area of development these past few years. In fact, I encourage you to check out the winners of the last few Table Contests:

- 2024 Table Contest Winners
- 2022 Table Contest Winners
- 2021 Table Contest Winners

Other Table Resources and Packages include:

- https://bookdown.org/yihui/rmarkdown-cookbook/table-other.html
- https://epirhandbook.com/en/new_pages/tables_descriptive.html
- `gt` package on CRAN and `gt package website`
- `kableExtra` package on CRAN and `kableExtra` package website
- `flextable` package on CRAN and `flextable` package website and `flextable` book
- `huxtable` package on CRAN

6. Exporting/Saving Data

Throughout this lesson we have worked with the `mydata` dataset. We have made some changes and created new variables. Let's save the updates to this little dataset for use in later modules.

Using the `save()` function

Save `mydata` as `*.Rdata` native R binary format

As we move forward in our lesson modules, we will mostly be working with the "native" format for datafiles (and objects) which have the extension of `*.RData` or `*.rda`. These file formats are efficient in terms of saving memory and speed for faster loading of data.

> **R data binary formats registered with Library of Congress**
>
> The "R Data Format Family (.rdata, .rda)" are registered with the Library of Congress under the "Sustainability of Digital Formats". The description summary states:
>
> > "The RData format (usually with extension .rdata or .rda) is a format designed for use with R, a system for statistical computation and related graphics, for storing a complete R workspace or selected"objects" from a workspace in a form that can be loaded back by R. The save function in R has options that result in significantly different variants of the format. This description is for the family of formats created by save and closely related functions. A workspace in R is a collection of typed "objects" and may include much more than the typical tabular data that might be considered a "dataset," including, for example, results of intermediate calculations and scripts in the R programming language. A workspace may also contain several datasets, which are termed "data frames" in R."

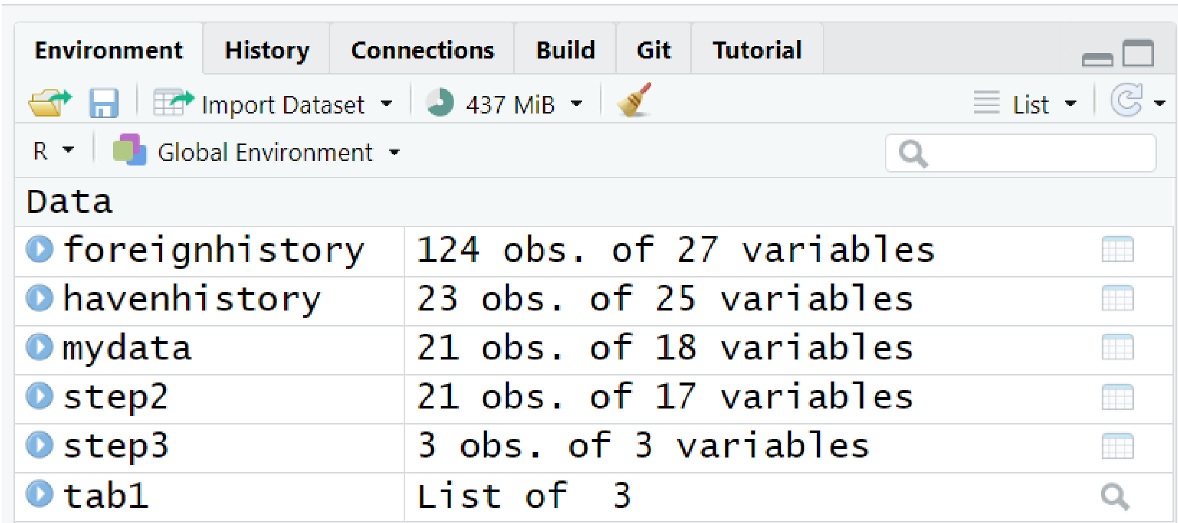
Let's save the `mydata` data.frame object as "mydata.RData", using the `save()` function. See `help(save, package = "base")`.

```r
# save the mydata dataset object
save(mydata,
     file = "mydata.RData")
```

Save All Objects in Global Environment as `*.Rdata`

It is worth noting that the code above specifically ONLY saves the `mydata` object. Assuming that your "Global Environment" was empty at the beginning of your computing session at the beginning of this Module 1.3.2, we have created 6 objects so far:

- `foreignhistory` - created above looking at the CRAN history for the `foreign` package
- `havenhistory` - created above looking at the CRAN history for the `haven` package
- `mydata` - main dataset imported above
- `step2` - created to illustrate the `%>%` stepwise programming workflow
- `step3` - created to illustrate the `%>%` stepwise programming workflow
- `tab1` - created above to make a table using the `arsenal` package



Suppose we want to save ALL of these objects for a future computing session or if you'd like to share all of these objects with someone else on your team.

We can save the whole Global Environment or select objects in the environment also to a `*.RData` file to be read back into a future computing session.

To save all objects in the Global Environment, we can use `save.image()`:

```
# save all objects from module 1.3.2
save.image(file = "module132.RData")
```

Save More than One Object in Global Environment as `*.Rdata`

To save one or more objects for future use - simply list the object names and then save them into an `*.RData` file.

```
# save mydata and tab1
save(mydata, tab1,
     file = "mydata_tab1.RData")
```

Reading Objects Saved as `*.Rdata` Back Into Session

To test and make sure these items were saved as we expect, let's remove all objects from our Global Environment and load them back in.

> ⚠ Be careful using `rm(list= ls())`
>
> The use of the `rm(list= ls())` should NOT be used unless you know you have saved everything up to this point. Once you remove all objects from your Global Environment, it cannot be undone. You can either rerun the R code to recreate these objects, or go through the steps described below to **save** and **re-load** your objects into your session.

```
# remove all objects
rm(list = ls())

# check that global environment is empty
ls()
```

```
character(0)
```

Read back in only the `mydata` file.

```
# read in mydata
load(file = "mydata.RData")

# check objects in global environment
ls()
```

```
[1] "mydata"
```

I'll remove all objects again for to illustrate the next use of `load()` function.

```r
rm(list = ls())
```

Read back in both the `mydata` and `tab1` objects

```r
# read in mydata_tab1
load(file = "mydata_tab1.RData")

# check objects in global environment
ls()
```

```
[1] "mydata" "tab1"
```

```r
rm(list = ls())
```

Read back in all objects saved from Module 1.3.2.

```r
# read in module132.RData
load(file = "module132.RData")

# check objects in global environment
ls()
```

```
[1] "foreignhistory" "havenhistory"   "mydata"         "step2"
[5] "step3"          "tab1"
```

```r
rm(list = ls())
```

Save/export data to other formats

In addition to use the built-in `save()` and `save.image()` functions, we can also export (or save) data objects from R into other formats like CSV and those for specific statistics software like SPSS (`*.sav`), SAS (`*.XPT`)and Stata (`*.dta`).

Export/Write CSV and EXCEL

In the `readr` package, we can use `write_csv()` to save our updated data as a CSV file which can be read by other software like Excel.

I'll load the data back in and then save/export it as other formats.

```
# read in mydata
load(file = "mydata.RData")

# write as CSV
readr::write_csv(mydata,
                 file = "mydata_updated.csv")
```

Export/Write for Other Software (SPSS, SAS, Stata)

We can also use the `haven` package to export/save the updated `mydata` dataset as a SPSS (`*.sav`), SAS (`*.XPT`) or Stata (`*.dta`) file format.

Code to export to SPSS `*.sav` format

```
haven::write_sav(mydata,
                 path = "mydata_updated.sav")
```

Rename variable "GenderCoded.f" and "SES.f" to "GenderCoded_f" and "SES_f" to export to SAS or Stata since the "*.f" won't work in a variable name in these software.

```
# rename GenderCoded.f and SES.f since the
# xxx.f wont work for SAS or Stata
names(mydata)[names(mydata) == "GenderCoded.f"] <-
  "GenderCoded_f"
names(mydata)[names(mydata) == "SES.f"] <-
  "SES_f"
```

Code to export to SAS using the "XPT" format

```
haven::write_xpt(mydata,
                 path = "mydata_updated.xpt")
```

Code to export to Stata `*.dta` format

```
haven::write_dta(mydata,
                 path = "mydata_updated.dta")
```

If you have these other statistical software on your computer, try opening these new exported files into that software to confirm they worked.

Some import/export work better than others

Be aware that many of these import/export functions do work pretty well, but some features of functionality of native formats used by other software packages may not work fully. Read the documentation for each package and function to understand what the limitations may be. For example, importing and exporting SAS `*.sas7bdat` formatted files can be problematic.

R Code For This Module

- `module_132.R`

References

Csárdi, Gábor, and Maëlle Salmon. 2025. *Pkgsearch: Search and Query CRAN r Packages*. https://github.com/r-hub/pkgsearch.

Heinzen, Ethan, Jason Sinnwell, Elizabeth Atkinson, Tina Gunderson, and Gregory Dougherty. 2021. *Arsenal: An Arsenal of r Functions for Large-Scale Statistical Summaries*. https://github.com/mayoverse/arsenal.

Iannone, Richard. 2023. *Fontawesome: Easily Work with Font Awesome Icons*. https://github.com/rstudio/fontawesome.

R Core Team. 2025. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Sjoberg, Daniel D., Joseph Larmarange, Michael Curry, Jessica Lavery, Karissa Whiting, and Emily C. Zabor. 2024. *Gtsummary: Presentation-Ready Data Summary and Analytic Result Tables*. https://github.com/ddsjoberg/gtsummary.

Sjoberg, Daniel D., Karissa Whiting, Michael Curry, Jessica A. Lavery, and Joseph Larmarange. 2021. "Reproducible Summary Tables with the Gtsummary Package." *The R Journal* 13: 570–80. https://doi.org/10.32614/RJ-2021-053.

Warnes, Gregory R., Ben Bolker, Thomas Lumley, Randall C Johnson. Contributions from Randall C. Johnson are Copyright SAIC-Frederick, Inc. Funded by the Intramural Research Program, of the NIH, National Cancer Institute, and Center for Cancer Research under NCI Contract NO1-CO-12400. 2022. *Gmodels: Various r Programming Tools for Model Fitting*. https://doi.org/10.32614/CRAN.package.gmodels.

Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *Dplyr: A Grammar of Data Manipulation*. https://dplyr.tidyverse.org.

Yoshida, Kazuki, and Alexander Bartel. 2022. *Tableone: Create Table 1 to Describe Baseline Characteristics with or Without Propensity Score Weights*. https://github.com/kaz-yos/tableone.

Other Helpful Resources

Other Helpful Resources