Introduction to R

To do our statistical modeling, we need to have a software to do so. I’ll demonstrate R code and output through the use of quarto documents.

* R is a great open source language with the best statistical learning packages
* RStudio is an interface for programming in R (or python)
* quarto is the next generation of R Markdown (program that weaves word processing, code, and output) that makes it easier to use multiple languages and programmatically render documents

## RStudio

* We can type code directly into the **console** for evaluation

#simple math operations  
# <-- is a comment - code not evaluated  
3 + 7

[1] 10

10 \* exp(3) #exp is exponential function

[1] 200.8554

log(pi^2) #log is natural log by default

[1] 2.28946

mean(cars$speed)

[1] 15.4

* We write our code in a ‘script’, R markdown document, or quarto document
* Let’s jump into RStudio and open a new quarto document

## quarto Basics

We want to make all of our data processing, modeling, visualizing, etc. steps reproducible! A first step is to use a notebook environment.

We’ll go through the basics of quarto to get you started. Much more is available on the [quarto docs page](https://quarto.org/docs/get-started/hello/rstudio.html), the [RStudio quarto integration page](https://docs.posit.co/ide/user/ide/guide/documents/quarto-project.html), and in the [R for Data Science book](https://r4ds.hadley.nz/quarto).

### Markdown Idea

Markdown is a simpler version of a markup language. HTML is the most commonly known markup language (HTML = Hypertext markup language). With HTML you use tags to specify things that a web browser like chrome interprets. For instance,

<h1>My first level header</h1>  
<a href = "https://www.google.com">Link to a search engine.</a>

The syntax for Markdown is really easy to pick up. Below you’ll find some commonly used markdown syntax:

* # R Markdown First level header
* ## Next Second level header
* \*\*Knit\*\* or \_\_Knit\_\_ Bold font (**Knit**)
* \*italic\* or \_italic\_ Italic font (*italic*)
* \*\_\_both\_\_\* Bold and italic (***both***)
* <https://rstudio.github.io/cheatsheets/quarto.pdf> A hyperlink: <https://rstudio.github.io/cheatsheets/quarto.pdf>
* [Cheat Sheet link](https://rstudio.github.io/cheatsheets/quarto.pdf) [Cheat Sheet link](https://rstudio.github.io/cheatsheets/quarto.pdf)

Check [this site for markdown basics](https://quarto.org/docs/authoring/markdown-basics.html). (Headers can be used to easily create a table of contents (and is useful for accessibility of documents).)

Where quarto goes beyond HTML is in the ability to weave R code into the equation!

* You can include **code chunks** in your .qmd file.
* You then render the markdown through RStudio (or the command line).
* The code runs and output is included in the final document!

Let’s jump into RStudio and create a .qmd file.

## R Objects

* We store **data/info/function/etc.** in R objects
* Create an R object via <- (recommended) or =

#save for later  
avg <- (5 + 7 + 6) / 3  
#call avg object  
avg

[1] 6

#strings (text) can be saved as well  
words <- c("Hello there!", "How are you?")  
words

[1] "Hello there!" "How are you?"

* Built-in objects exist like letters and cars don’t show automatically

library(knitr) #gives access to the kable function  
letters

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"  
[20] "t" "u" "v" "w" "x" "y" "z"

head(cars, n = 3) |>  
 kable()

| speed | dist |
| --- | --- |
| 4 | 2 |
| 4 | 10 |
| 7 | 4 |

* There are many functions to help understand an R Object. str() is one of the best

str(cars)

'data.frame': 50 obs. of 2 variables:  
 $ speed: num 4 4 7 7 8 9 10 10 10 11 ...  
 $ dist : num 2 10 4 22 16 10 18 26 34 17 ...

str(avg)

num 6

* This is essentially equivalent to looking at the Environment area in RStudio

## Common Data Structures

It is important to understand three types of data structures in R.

1. Atomic Vector (1d)
2. Data Frame (2d)
3. List (1d)

### (Atomic) Vector

* 1D group of elements with an ordering that starts at 1

|  |
| --- |
| Visuals of Two Vectors |

* **Elements** must be same the same ‘type’ (homogeneous). The most common types of data are:
  + logical, integer, double, and character
* c() “combines” values together. Simply separate the values with a comma

#vectors (1 dimensional) objects  
#all elements of the same 'type'  
x <- c(1, 3, 10, -20, sqrt(2))  
x

[1] 1.000000 3.000000 10.000000 -20.000000 1.414214

* Let’s create another vector y with strings stored in it

y <- c("cat", "dog", "bird", "floor")  
y

[1] "cat" "dog" "bird" "floor"

* We can combine two vectors together using c() as well!

z <- c(x, y)  
z

[1] "1" "3" "10" "-20"   
[5] "1.4142135623731" "cat" "dog" "bird"   
[9] "floor"

* R does **element-wise** math by default

x

[1] 1.000000 3.000000 10.000000 -20.000000 1.414214

x + 3

[1] 4.000000 6.000000 13.000000 -17.000000 4.414214

#### Accessing Elements of a Vector

When thinking about accessing (or subsetting) a vector’s elements, remember that vectors are 1D. We can place the numbers corresponding to the positions of the elements we want inside of [] at the end of the vector to return them.

* Return vector elements using square brackets [] at the end of a vector.

letters #built-in vector

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"  
[20] "t" "u" "v" "w" "x" "y" "z"

letters[1] #R starts counting at 1!

[1] "a"

letters[26]

[1] "z"

letters[1:4]

[1] "a" "b" "c" "d"

letters[c(5, 10, 15, 20, 25)]

[1] "e" "j" "o" "t" "y"

x <- c(1, 2, 5)  
letters[x]

[1] "a" "b" "e"

We’d call x above an *indexing vector*

### Data Frame

A 2D objects where the columns are homogenous but can be of different types

|  |
| --- |
| Visual of a Data Frame |

Let’s check out the ‘built-in’ iris data frame

str(iris)

'data.frame': 150 obs. of 5 variables:  
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...  
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...  
 $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...  
 $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...  
 $ Species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

* This is a 2D structure and we can access it with [,]

iris[1:4, 2:4] |> #returns a data frame  
 kable()

| Sepal.Width | Petal.Length | Petal.Width |
| --- | --- | --- |
| 3.5 | 1.4 | 0.2 |
| 3.0 | 1.4 | 0.2 |
| 3.2 | 1.3 | 0.2 |
| 3.1 | 1.5 | 0.2 |

iris[1, ] |> #returns a data frame  
 kable()

| Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
| --- | --- | --- | --- | --- |
| 5.1 | 3.5 | 1.4 | 0.2 | setosa |

iris[1:10, 1] |> #returns a vector  
 kable()

| x |
| --- |
| 5.1 |
| 4.9 |
| 4.7 |
| 4.6 |
| 5.0 |
| 5.4 |
| 4.6 |
| 5.0 |
| 4.4 |
| 4.9 |

We’ll see easy tidyverse functions for dealing with tibbles which are special data frames.

### List

* A vector that can have differing elements! (still **1D**)

|  |
| --- |
| Visual of a List |

* An ordered set of objects (ordering starts at 1)
* Useful for more complex types of data like the output of a model fit

lm\_fit <- lm(Sepal.Length ~ Sepal.Width, data = iris)  
str(lm\_fit) |>  
 kable()

List of 12  
 $ coefficients : Named num [1:2] 6.526 -0.223  
 ..- attr(\*, "names")= chr [1:2] "(Intercept)" "Sepal.Width"  
 $ residuals : Named num [1:150] -0.644 -0.956 -1.111 -1.234 -0.722 ...  
 ..- attr(\*, "names")= chr [1:150] "1" "2" "3" "4" ...  
 $ effects : Named num [1:150] -71.566 -1.188 -1.081 -1.187 -0.759 ...  
 ..- attr(\*, "names")= chr [1:150] "(Intercept)" "Sepal.Width" "" "" ...  
 $ rank : int 2  
 $ fitted.values: Named num [1:150] 5.74 5.86 5.81 5.83 5.72 ...  
 ..- attr(\*, "names")= chr [1:150] "1" "2" "3" "4" ...  
 $ assign : int [1:2] 0 1  
 $ qr :List of 5  
 ..$ qr : num [1:150, 1:2] -12.2474 0.0816 0.0816 0.0816 0.0816 ...  
 .. ..- attr(\*, "dimnames")=List of 2  
 .. .. ..$ : chr [1:150] "1" "2" "3" "4" ...  
 .. .. ..$ : chr [1:2] "(Intercept)" "Sepal.Width"  
 .. ..- attr(\*, "assign")= int [1:2] 0 1  
 ..$ qraux: num [1:2] 1.08 1.02  
 ..$ pivot: int [1:2] 1 2  
 ..$ tol : num 1e-07  
 ..$ rank : int 2  
 ..- attr(\*, "class")= chr "qr"  
 $ df.residual : int 148  
 $ xlevels : Named list()  
 $ call : language lm(formula = Sepal.Length ~ Sepal.Width, data = iris)  
 $ terms :Classes 'terms', 'formula' language Sepal.Length ~ Sepal.Width  
 .. ..- attr(\*, "variables")= language list(Sepal.Length, Sepal.Width)  
 .. ..- attr(\*, "factors")= int [1:2, 1] 0 1  
 .. .. ..- attr(\*, "dimnames")=List of 2  
 .. .. .. ..$ : chr [1:2] "Sepal.Length" "Sepal.Width"  
 .. .. .. ..$ : chr "Sepal.Width"  
 .. ..- attr(\*, "term.labels")= chr "Sepal.Width"  
 .. ..- attr(\*, "order")= int 1  
 .. ..- attr(\*, "intercept")= int 1  
 .. ..- attr(\*, "response")= int 1  
 .. ..- attr(\*, ".Environment")=<environment: R\_GlobalEnv>   
 .. ..- attr(\*, "predvars")= language list(Sepal.Length, Sepal.Width)  
 .. ..- attr(\*, "dataClasses")= Named chr [1:2] "numeric" "numeric"  
 .. .. ..- attr(\*, "names")= chr [1:2] "Sepal.Length" "Sepal.Width"  
 $ model :'data.frame': 150 obs. of 2 variables:  
 ..$ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...  
 ..$ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...  
 ..- attr(\*, "terms")=Classes 'terms', 'formula' language Sepal.Length ~ Sepal.Width  
 .. .. ..- attr(\*, "variables")= language list(Sepal.Length, Sepal.Width)  
 .. .. ..- attr(\*, "factors")= int [1:2, 1] 0 1  
 .. .. .. ..- attr(\*, "dimnames")=List of 2  
 .. .. .. .. ..$ : chr [1:2] "Sepal.Length" "Sepal.Width"  
 .. .. .. .. ..$ : chr "Sepal.Width"  
 .. .. ..- attr(\*, "term.labels")= chr "Sepal.Width"  
 .. .. ..- attr(\*, "order")= int 1  
 .. .. ..- attr(\*, "intercept")= int 1  
 .. .. ..- attr(\*, "response")= int 1  
 .. .. ..- attr(\*, ".Environment")=<environment: R\_GlobalEnv>   
 .. .. ..- attr(\*, "predvars")= language list(Sepal.Length, Sepal.Width)  
 .. .. ..- attr(\*, "dataClasses")= Named chr [1:2] "numeric" "numeric"  
 .. .. .. ..- attr(\*, "names")= chr [1:2] "Sepal.Length" "Sepal.Width"  
 - attr(\*, "class")= chr "lm"

* We can use double square brackets [[ ]] (or [ ]) to return a single list element or use the list element name after a $

lm\_fit[[1]] |>  
 kable()

|  | x |
| --- | --- |
| (Intercept) | 6.5262226 |
| Sepal.Width | -0.2233611 |

lm\_fit$coefficients |>  
 kable()

|  | x |
| --- | --- |
| (Intercept) | 6.5262226 |
| Sepal.Width | -0.2233611 |

## tidyverse

One of the big impediments to learning R in the past was the vast ecosystem of packages.

* Many ways to do the same task via competing R packages
* Most packages written by different people
* Different syntax was used in different packages
* Required lots of reading of help pages to understand how to use each package/function

Along came the tidyverse collection of packages! While not the most efficient method for programming, the tidyverse provides a coherent ecosystem for almost all common data tasks! That is,

* (Almost) all packages have functions with the same syntax
* Functions are built to work together
* A plethora of help documentation and vignettes exists

### tidyverse Syntax

As the tidyverse is mostly concerned with the analysis and manipulation of data, the main data object used is a special version of a data frame called a **tibble**.

iris\_tbl <- dplyr::as\_tibble(iris)  
str(iris\_tbl)

tibble [150 × 5] (S3: tbl\_df/tbl/data.frame)  
 $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...  
 $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...  
 $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...  
 $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...  
 $ Species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

### tidyverse Packages

The tidyverse consists of a large number of packages. However, library(tidyverse) loads only the eight core packages (which sometimes load other packages of course). Those are ([from their website](https://www.tidyverse.org/packages/)):

* ggplot2 - ggplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell ggplot2 how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details
* dplyr - dplyr provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges
* tidyr - tidyr provides a set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable
* readr - readr provides a fast and friendly way to read rectangular data (like csv, tsv, and fwf). It is designed to flexibly parse many types of data found in the wild, while still cleanly failing when data unexpectedly changes
* …

### Reading Data

Data comes in many formats such as

* ‘Delimited’ data: Character (such as [‘,’](https://www4.stat.ncsu.edu/~online/datasets/scoresFull.csv) , [‘>’](https://www4.stat.ncsu.edu/~online/datasets/umps2012.txt), or [’ ’]) separated data
* [Excel](https://www4.stat.ncsu.edu/~online/datasets/Dry_Bean_Dataset.xlsx) data
* From a database
* From an Application Programming Interface (API)

We can read delimited data using the readr package. Make sure tidyverse package is installed (this can take a while). This includes readr.

install.packages("tidyverse")

* Load the library into your current session

library(tidyverse)

── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──  
✔ dplyr 1.1.4 ✔ readr 2.1.4  
✔ forcats 1.0.0 ✔ stringr 1.5.0  
✔ ggplot2 3.4.4 ✔ tibble 3.2.1  
✔ lubridate 1.9.2 ✔ tidyr 1.3.0  
✔ purrr 1.0.1   
── Conflicts ────────────────────────────────────────── tidyverse\_conflicts() ──  
✖ dplyr::filter() masks stats::filter()  
✖ dplyr::lag() masks stats::lag()  
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors

* You can see this loads in the eight core packages mentioned previously. The warnings can easily be ignored but we should take care with the conflicts. We’ve overwritten some functions from BaseR. Recall, we can call those functions explicitly if we’d like (stats::filter()).
* Suppose we want to read in the file called bikeDetails.csv available at: <https://www4.stat.ncsu.edu/~online/datasets/bikeDetails.csv>
  + We can download the file and store it locally, reading it in from there
  + Or, for this type of file, we can also read it directly from the web!

We’ll use the read\_csv() function from the readr package. The inputs are:

read\_csv(  
 file,  
 col\_names = TRUE,  
 col\_types = NULL,  
 col\_select = NULL,  
 id = NULL,  
 locale = default\_locale(),  
 na = c("", "NA"),  
 quoted\_na = TRUE,  
 quote = "\"",  
 comment = "",  
 trim\_ws = TRUE,  
 skip = 0,  
 n\_max = Inf,  
 guess\_max = min(1000, n\_max),  
 name\_repair = "unique",  
 num\_threads = readr\_threads(),  
 progress = show\_progress(),  
 show\_col\_types = should\_show\_types(),  
 skip\_empty\_rows = TRUE,  
 lazy = should\_read\_lazy()  
)

We really only need to specify the file argument but we see there are a few others that might be useful.

bike\_details <- read\_csv("https://www4.stat.ncsu.edu/~online/datasets/bikeDetails.csv")

Rows: 1061 Columns: 7  
── Column specification ────────────────────────────────────────────────────────  
Delimiter: ","  
chr (3): name, seller\_type, owner  
dbl (4): selling\_price, year, km\_driven, ex\_showroom\_price  
  
ℹ Use `spec()` to retrieve the full column specification for this data.  
ℹ Specify the column types or set `show\_col\_types = FALSE` to quiet this message.

bike\_details[1:5, ] |>  
 kable()

| name | selling\_price | year | seller\_type | owner | km\_driven | ex\_showroom\_price |
| --- | --- | --- | --- | --- | --- | --- |
| Royal Enfield Classic 350 | 175000 | 2019 | Individual | 1st owner | 350 | NA |
| Honda Dio | 45000 | 2017 | Individual | 1st owner | 5650 | NA |
| Royal Enfield Classic Gunmetal Grey | 150000 | 2018 | Individual | 1st owner | 12000 | 148114 |
| Yamaha Fazer FI V 2.0 [2016-2018] | 65000 | 2015 | Individual | 1st owner | 23000 | 89643 |
| Yamaha SZ [2013-2014] | 20000 | 2011 | Individual | 2nd owner | 21000 | NA |

* Functions from *readr* and their purpose

| Delimiter | Function |
| --- | --- |
| comma ‘,’ | read\_csv() |
| tab | read\_tsv() |
| space ’ ’ | read\_table() |
| semi-colon ‘;’ | read\_csv2() (This uses ; instead of commas, which is common in many countries) |
| other | read\_delim(…,delim = ,…) |

Consider the umps.txt file available at: <https://www4.stat.ncsu.edu/~online/datasets/umps2012.txt>

* Note that the delimiter is a > sign!
* Note that there are no column names provided:
  + Year Month Day Home Away HPUmpire are the appropriate column names

We can use read\_delim() to read in a generic delimited raw data file! Let’s check the help:

read\_delim(  
 file,  
 delim = NULL,  
 quote = "\"",  
 escape\_backslash = FALSE,  
 escape\_double = TRUE,  
 col\_names = TRUE,  
 col\_types = NULL,  
 col\_select = NULL,  
 id = NULL,  
 locale = default\_locale(),  
 na = c("", "NA"),  
 quoted\_na = TRUE,  
 comment = "",  
 trim\_ws = FALSE,  
 skip = 0,  
 n\_max = Inf,  
 guess\_max = min(1000, n\_max),  
 name\_repair = "unique",  
 num\_threads = readr\_threads(),  
 progress = show\_progress(),  
 show\_col\_types = should\_show\_types(),  
 skip\_empty\_rows = TRUE,  
 lazy = should\_read\_lazy()  
)

We see two arguments we need to worry about right off:

* file (path to file)
* delim the delimiter used in the raw data file
  + Single character used to separate fields within a record.
  + We want to specify a character string with the delimiter for this.

As we don’t have column names we should also consider the col\_names argument. This is set to TRUE by default. The help says:

Either TRUE, FALSE or a character vector of column names.  
If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.  
If col\_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.  
Missing (NA) column names will generate a warning, and be filled in with dummy names …1, …2 etc. Duplicate column names will generate a warning and be made unique, see name\_repair to control how this is done.

* This means we want to set the value to FALSE or supply a character vector with the corresponding names!

ump\_data <- read\_delim("https://www4.stat.ncsu.edu/~online/datasets/umps2012.txt",   
 delim = ">",  
 col\_names = c("Year", "Month", "Day", "Home", "Away", "HPUmpire")  
)

Rows: 2359 Columns: 6  
── Column specification ────────────────────────────────────────────────────────  
Delimiter: ">"  
chr (3): Home, Away, HPUmpire  
dbl (3): Year, Month, Day  
  
ℹ Use `spec()` to retrieve the full column specification for this data.  
ℹ Specify the column types or set `show\_col\_types = FALSE` to quiet this message.

ump\_data[1:5, ] |>  
 kable()

| Year | Month | Day | Home | Away | HPUmpire |
| --- | --- | --- | --- | --- | --- |
| 2012 | 4 | 12 | MIN | LAA | D.J. Reyburn |
| 2012 | 4 | 12 | SD | ARI | Marty Foster |
| 2012 | 4 | 12 | WSH | CIN | Mike Everitt |
| 2012 | 4 | 12 | PHI | MIA | Jeff Nelson |
| 2012 | 4 | 12 | CHC | MIL | Fieldin Culbreth |

### Manipulating Data with dplyr

The two major tasks we’ll consider are

* Row manipulations
  + **filtering** or subsetting our observations
* Column manipulations
  + **selecting** a subset of our variables
  + **rename** a column
  + **mutating** our data frame to create a new variable

tidyverse syntax:

* function(tibble, arguments, ...)
  + Allows for easy ‘chaining’ or ‘piping’!

iris |>   
 select(Sepal.Length, starts\_with("Petal")) |>  
 filter(Sepal.Length > 5.4) |>  
 mutate(Petal.Avg = (Petal.Length+Petal.Width)/2) |>  
 kable()

#### Row Manipulations with dplyr

A common task is to only grab certain types of observations (**filter rows**)

|  |
| --- |
| Visual of Filtering a Data Frame |

or rearrange the order of the observations (rows). The two functions from dplyr that help us here are

* filter() - subset **rows**

filter() generally takes a tibble as its first argument and then a logical vector as the next (of the same length as the number of rows):

* Return observations where the number of games played is greater than 50 (the G column):

library(Lahman) #install this package  
batting\_tbl <- as\_tibble(Batting)  
batting\_tbl |>  
 filter(G > 50, yearID == 2018) |>  
 slice(1:5) |>  
 kable()

| playerID | yearID | stint | teamID | lgID | G | AB | R | H | X2B | X3B | HR | RBI | SB | CS | BB | SO | IBB | HBP | SH | SF | GIDP |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| abreujo02 | 2018 | 1 | CHA | AL | 128 | 499 | 68 | 132 | 36 | 1 | 22 | 78 | 2 | 0 | 37 | 109 | 7 | 11 | 0 | 6 | 14 |
| acunaro01 | 2018 | 1 | ATL | NL | 111 | 433 | 78 | 127 | 26 | 4 | 26 | 64 | 16 | 5 | 45 | 123 | 2 | 6 | 0 | 3 | 4 |
| adamewi01 | 2018 | 1 | TBA | AL | 85 | 288 | 43 | 80 | 7 | 0 | 10 | 34 | 6 | 5 | 31 | 95 | 3 | 1 | 1 | 2 | 6 |
| adamsma01 | 2018 | 1 | WAS | NL | 94 | 249 | 37 | 64 | 9 | 0 | 18 | 48 | 0 | 0 | 24 | 55 | 2 | 4 | 0 | 0 | 6 |
| adducji02 | 2018 | 1 | DET | AL | 59 | 176 | 19 | 47 | 8 | 2 | 3 | 21 | 1 | 0 | 6 | 45 | 0 | 0 | 2 | 1 | 2 |

batting\_tbl |>  
 filter(G > 50, yearID %in% c(2018, 2019, 2020)) |>  
 slice(1:5) |>  
 kable()

| playerID | yearID | stint | teamID | lgID | G | AB | R | H | X2B | X3B | HR | RBI | SB | CS | BB | SO | IBB | HBP | SH | SF | GIDP |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| abreujo02 | 2018 | 1 | CHA | AL | 128 | 499 | 68 | 132 | 36 | 1 | 22 | 78 | 2 | 0 | 37 | 109 | 7 | 11 | 0 | 6 | 14 |
| acunaro01 | 2018 | 1 | ATL | NL | 111 | 433 | 78 | 127 | 26 | 4 | 26 | 64 | 16 | 5 | 45 | 123 | 2 | 6 | 0 | 3 | 4 |
| adamewi01 | 2018 | 1 | TBA | AL | 85 | 288 | 43 | 80 | 7 | 0 | 10 | 34 | 6 | 5 | 31 | 95 | 3 | 1 | 1 | 2 | 6 |
| adamsma01 | 2018 | 1 | WAS | NL | 94 | 249 | 37 | 64 | 9 | 0 | 18 | 48 | 0 | 0 | 24 | 55 | 2 | 4 | 0 | 0 | 6 |
| adducji02 | 2018 | 1 | DET | AL | 59 | 176 | 19 | 47 | 8 | 2 | 3 | 21 | 1 | 0 | 6 | 45 | 0 | 0 | 2 | 1 | 2 |

* If we want an *or* condition, we use the compound logical operator for that

batting\_tbl |>  
 filter(G > 50 | yearID %in% c(2018, 2019, 2020)) |>  
 slice(1:5) |>  
 kable()

| playerID | yearID | stint | teamID | lgID | G | AB | R | H | X2B | X3B | HR | RBI | SB | CS | BB | SO | IBB | HBP | SH | SF | GIDP |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bechtge01 | 1872 | 1 | NY2 | NA | 51 | 247 | 61 | 74 | 11 | 3 | 0 | 42 | 9 | 1 | 7 | 3 | NA | NA | NA | NA | 2 |
| cummica01 | 1872 | 1 | NY2 | NA | 55 | 249 | 37 | 52 | 9 | 3 | 0 | 26 | 0 | 1 | 4 | 14 | NA | NA | NA | NA | 2 |
| eggleda01 | 1872 | 1 | NY2 | NA | 56 | 290 | 94 | 97 | 20 | 0 | 0 | 19 | 18 | 6 | 8 | 9 | NA | NA | NA | NA | 0 |
| hallge01 | 1872 | 1 | BL1 | NA | 53 | 250 | 69 | 84 | 17 | 6 | 1 | 37 | 8 | 1 | 3 | 1 | NA | NA | NA | NA | 0 |
| hatfijo01 | 1872 | 1 | NY2 | NA | 56 | 288 | 76 | 93 | 15 | 2 | 1 | 47 | 12 | 5 | 9 | 6 | NA | NA | NA | NA | 3 |

#### Column Manipulations with dplyr

We may want to subset our variables, rename them, or create new variables.

##### select() - Subset Columns

We call the subset of our variables **selecting** columns (or variables)

|  |
| --- |
| Visualization of Selecting a Column |

* Suppose we just wanted to look at the playerID, teamID, and hits type variables: H, X2B X3B, and HR of the players in our subset
* We can add in a select() function to our chain

batting\_tbl |>  
 filter(G > 50 | yearID %in% c(2018, 2019, 2020)) |>  
 select(playerID, teamID, H, X2B, X3B, HR) |>  
 slice(1:5) |>  
 kable()

| playerID | teamID | H | X2B | X3B | HR |
| --- | --- | --- | --- | --- | --- |
| bechtge01 | NY2 | 74 | 11 | 3 | 0 |
| cummica01 | NY2 | 52 | 9 | 3 | 0 |
| eggleda01 | NY2 | 97 | 20 | 0 | 0 |
| hallge01 | BL1 | 84 | 17 | 6 | 1 |
| hatfijo01 | NY2 | 93 | 15 | 2 | 1 |

* Where we really gain here is the ability to use helper functions when selecting columns!
  + : to select all contiguous columns, starts\_with(), and ends\_with()

batting\_tbl |>  
 filter(G > 50 | yearID %in% c(2018, 2019, 2020)) |>  
 select(ends\_with("ID"), G, AB, H:HR) |>  
 slice(1:5) |>  
 kable()

| playerID | yearID | teamID | lgID | G | AB | H | X2B | X3B | HR |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bechtge01 | 1872 | NY2 | NA | 51 | 247 | 74 | 11 | 3 | 0 |
| cummica01 | 1872 | NY2 | NA | 55 | 249 | 52 | 9 | 3 | 0 |
| eggleda01 | 1872 | NY2 | NA | 56 | 290 | 97 | 20 | 0 | 0 |
| hallge01 | 1872 | BL1 | NA | 53 | 250 | 84 | 17 | 6 | 1 |
| hatfijo01 | 1872 | NY2 | NA | 56 | 288 | 93 | 15 | 2 | 1 |

##### rename()

rename() comes in handy, especially when we have non-standard column names

batting\_tbl |>  
 filter(G > 50 | yearID %in% c(2018, 2019, 2020)) |>  
 select(playerID, teamID, H, X2B, X3B, HR) |>  
 rename("Doubles" = "X2B", "Triples" = "X3B") |>  
 slice(1:5) |>  
 kable()

| playerID | teamID | H | Doubles | Triples | HR |
| --- | --- | --- | --- | --- | --- |
| bechtge01 | NY2 | 74 | 11 | 3 | 0 |
| cummica01 | NY2 | 52 | 9 | 3 | 0 |
| eggleda01 | NY2 | 97 | 20 | 0 | 0 |
| hallge01 | BL1 | 84 | 17 | 6 | 1 |
| hatfijo01 | NY2 | 93 | 15 | 2 | 1 |

##### mutate() to Create New Variables

This function allows us to create one or more variables and append them to our tibble.

* For our dataset from above, suppose we wanted to create an “extra base hits” type column that is the sum of the doubles, triples, and home runs.

batting\_tbl |>  
 filter(G > 50 | yearID %in% c(2018, 2019, 2020)) |>  
 select(playerID, teamID, H, X2B, X3B, HR) |>  
 rename("Doubles" = "X2B", "Triples" = "X3B") |>  
 mutate(Extra\_Base\_Hits = Doubles + Triples + HR) |>  
 slice(1:5) |>  
 kable()

| playerID | teamID | H | Doubles | Triples | HR | Extra\_Base\_Hits |
| --- | --- | --- | --- | --- | --- | --- |
| bechtge01 | NY2 | 74 | 11 | 3 | 0 | 14 |
| cummica01 | NY2 | 52 | 9 | 3 | 0 | 12 |
| eggleda01 | NY2 | 97 | 20 | 0 | 0 | 20 |
| hallge01 | BL1 | 84 | 17 | 6 | 1 | 24 |
| hatfijo01 | NY2 | 93 | 15 | 2 | 1 | 18 |

* If we want to add more than one variable, we just separate the variable definitions with a comma.
  + Let’s add a Singles variable representing the number of hits minus the number of extra base hits

batting\_tbl |>  
 filter(G > 50 | yearID %in% c(2018, 2019, 2020)) |>  
 select(playerID, teamID, H, X2B, X3B, HR) |>  
 rename("Doubles" = "X2B", "Triples" = "X3B") |>  
 mutate(Extra\_Base\_Hits = Doubles + Triples + HR,  
 Singles = H - Extra\_Base\_Hits) |>  
 select(playerID, teamID, Singles, Doubles:HR, H, Extra\_Base\_Hits) |>  
 slice(1:5) |>  
 kable()

| playerID | teamID | Singles | Doubles | Triples | HR | H | Extra\_Base\_Hits |
| --- | --- | --- | --- | --- | --- | --- | --- |
| bechtge01 | NY2 | 60 | 11 | 3 | 0 | 74 | 14 |
| cummica01 | NY2 | 40 | 9 | 3 | 0 | 52 | 12 |
| eggleda01 | NY2 | 77 | 20 | 0 | 0 | 97 | 20 |
| hallge01 | BL1 | 60 | 17 | 6 | 1 | 84 | 24 |
| hatfijo01 | NY2 | 75 | 15 | 2 | 1 | 93 | 18 |

## Recap!

quarto is great for creating documents with code and text

dplyr gives us a ton of functionality for doing common data manipulations

* as\_tibble() - coerce a data frame to a tibble
* filter() - subset rows
* select() - subset/reorder columns
* rename() - rename columns
* mutate() - add new variables to the tibble

The functionality of selecting columns described in the help for select() can be used in many places across the tidyverse and the functions group\_by() and if\_else() are really useful as well!

* [dplyr Cheat Sheet](https://rstudio.github.io/cheatsheets/html/data-transformation.html) (PDF version on the right hand side of the page)