Introduction to R Studio

Chelsea L. Faber, PhD

Introduction

This document is written in a Quarto Markdown (.qmd) file. It allows you to combine code with annotation and documentation that can be rendered to .pdf, .html, .doc, or even .ppt, for publishing.

{r} indicates a code block. This will be recognized by R Studio as executable code. The rest of the text is recognized as Markdown, a text editing language that allows you to use specific **syntax** (e.g., #, ##, *, **) to control how your words appear in the rendered document.

'#' Header 1

'##' Header 2

* *italic* *

** **bold** **

Whether these characters are visible or not depends on whether you are in Source or Visual Editing Mode.

Your first code block

The section below is a **code block**. This means that it contains code that will be recognized and executed by R if you run it. There are a few ways to run code contained within this document:

1. Click the green arrow in the top right of the code block to run the entire chunk at once.
2. Place your cursor anywhere on a single line you would like to run and type `Ctrl` / `Cmd + Enter`.

3. Copy and paste the content of the code block into your R console and type `Enter`.

```
# Welcome to your first code block!

# Within a code block, and generally anywhere else in R that is outside of a
# .qmd or .md file, '#' means something different.

# Everything following a '#' is a COMMENT. Comments will not be executed.
# Use comments for the humans reading your code - including future You!

# Comments are often helpful before a long code block, or after  individual lines
# to clarify the functionality of that code.
```

Operators

Basic math in R

Operators are used to perform operations on variables and values. R uses the default arithmetic operators you already know: `+`, `-`, `*`, `/`, `^`, `%%` (modulus), and `%/%` (integer division)

```
# To run these, either type into the Console and hit Enter, or, if working with a
# script in the Editor window, typing Ctrl/Cmd + Enter will run the line where
# your cursor is located.

1 + 1       # simple addition
```

```
[1] 2
```

```
4 / 3       # division
```

```
[1] 1.333333
```

```
4 %% 3      # remainder
```

```
[1] 1
```

```
4*2^2       # order of operations applies
```

```
[1] 16
```

```r
1e3         # 1 * 10^3
```

```
[1] 1000
```

```r
log10(1e3)  # log base 10
```

```
[1] 3
```

```r
exp(3)      # e^3
```

```
[1] 20.08554
```

```r
log(exp(3)) #ln(e^3)
```

```
[1] 3
```

R assignment operators assign values to variables

A variable is an object that will be saved to your R Environment that holds the values you assign it. While there are other ways to create a variable, it is best practice to use the left assignment operator, `<-`. You can see the values of your variables by looking in your Environment window, or by typing the variable name into the Console and hitting `Enter`.

```r
x  <- 2     # left assignment operator
y  <- 4
z  <- x*y^2
z  <- x^2 * y^2
a  <- 14

# Check your assignments
x
```

```
[1] 2
```

```r
y
```

```
[1] 4
```

```r
z
```

```
[1] 64
```

Immediately print new variables to the console at creation by wrapping the assignment expression in parentheses `()`.

```r
(z <- (x*y)^2)
```

```
[1] 64
```

You can check what variables exist in your environment with `ls()`.

```r
ls()
```

```
[1] "a"               "has_annotations" "x"               "y"
[5] "z"
```

Use the `rm()` function to remove variables from your environment. Use `rm(list = ls())` to remove all variables.

```r
rm(a) # remove a single variable
```

R comparison and logical operators

Comparison operators aptly compare two values, giving an output of `TRUE` (numerically represented as `1`) or `FALSE` (numerically represented as `0`). They include: `==`, `!=`, `>`, `<`, `>=`, and `<=`. Use `&` or `|` to combine multiple comparisons.

```r
x == y # x is equal to y
```

```
[1] FALSE
```

```r
x != y # x is not equal to y
```

```
[1] TRUE
```

```r
x < y  # x is less than y
```

```
[1] TRUE
```

```r
(x < y) & (x < z) # x is less than y AND x is less than z
```

```
[1] TRUE
```

```r
(x == z) | (x < z) # x equals z OR x is less than z
```

```
[1] TRUE
```

```r
x <= z # more succinctly than above, x is less than or equal to z
```

```
[1] TRUE
```

Other useful operators

More often than not, you will be working with vectors, data frames, or lists, rather than variables containing a single value. A few useful operators for working with vectors are `:`, and `%in%`.

```r
x <- 1:10   # creates a vector sequence
4 %in% x    # finds whether element is within vector - returns boolean
```

```
[1] TRUE
```

Functions and Packages

Functions

A function is a preset command that automatically performs a specific process or task on inputs you designate (also known as `arguments`). To call a function, enter the function name followed by parentheses. Let's start with one that takes no arguments, `getwd()`. `getwd()` (for get working directory) tells you where your current R session is running within your file system.

```
getwd() # get current working directory
```

```
[1] "C:/Users/kaspe/R/Projects/Biostats_Workshop"
```

```
getwd # notice what happens when you forget the parentheses
```

```
function ()
.Internal(getwd())
<bytecode: 0x00000218d03255c0>
<environment: namespace:base>
```

You can change your working directory with `setwd()`, but we won't go into that too much just yet. Just know that it is important for you to explicitly tell R where to look for files.

```
my_directory <- "C:\\file\\path\\here"
setwd(my_directory)
```

Most functions require arguments, however. Let's learn another useful function `c()`, to combine values into a vector, then get some information about the vector with `range()`, `length()`, and `summary()`.

```
my_vector <- c(4,15,9,3,4)
my_vector
```

```
[1]  4 15  9  3  4
```

```
length(my_vector)
```

```
[1] 5
```

```r
range(my_vector)
```

```
[1]  3 15
```

```r
summary(my_vector)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
      3       4       4       7       9      15 
```

R is very powerful in working with vectors. Most functions in R are optimally employed with vectors.

```r
my_new_vector <- 4*my_vector
my_new_vector
```

```
[1] 16 60 36 12 16
```

Getting help

Getting help in R is easy. If you don't understand how a function works, or what its arguments and outputs are, simply enter `?function`.

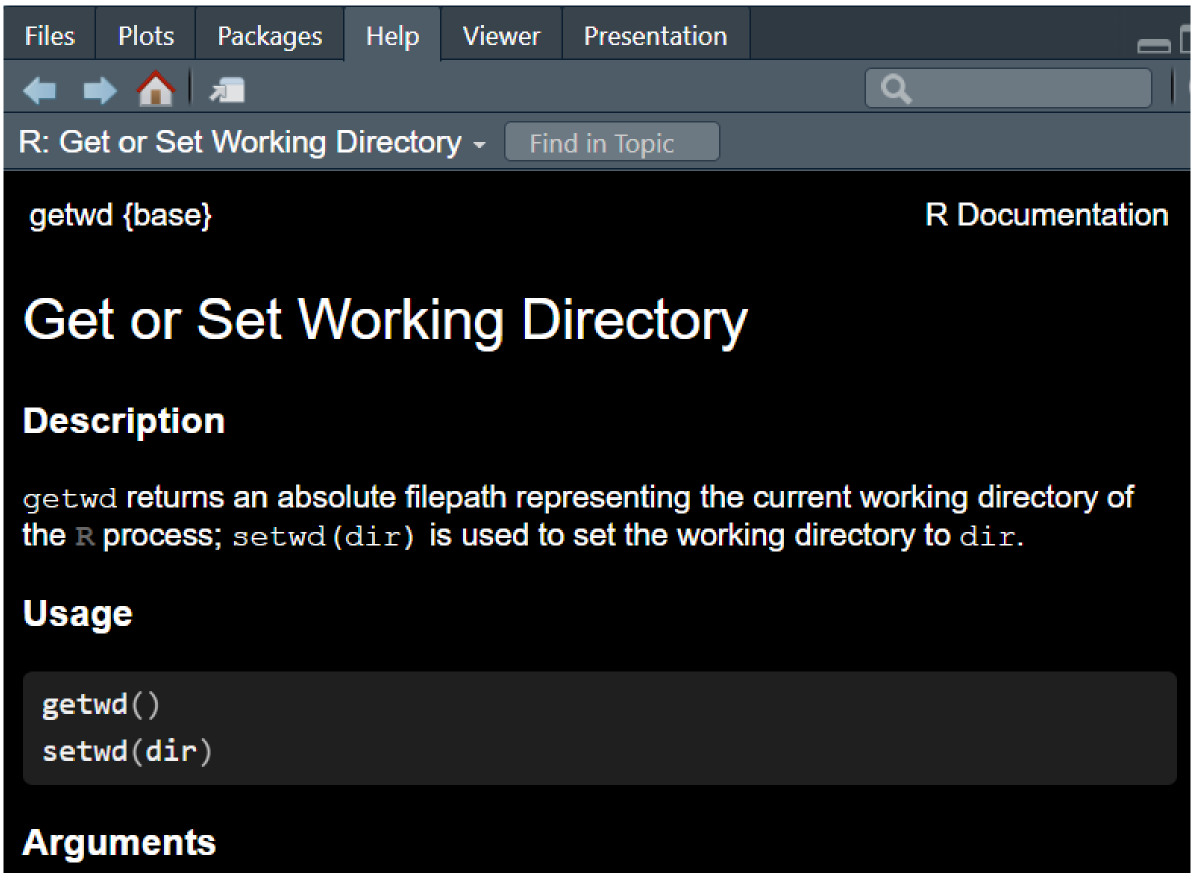
This will open the Help panel, where you can read the R Documentation.

```r
?getwd()
```

Packages

Packages are a collection of functions that you can save to your device in a library. Packages increase the power of R by organizing and annotating base R functions that perform similar tasks. There are *thousands* of packages available (and more every day). Most packages are available through the Comprehensive R Archive Network (CRAN) here. Many packages (and their source code) are also published by their authors on GitHub.

Check which packages you already have with `library()`, or navigate to the Packages tab of the Help pane.. Load a specific package with `library(package_name)`. Install a new package from CRAN with `install.packages("package_name")`. See where your packages are saved on your device with `.libPaths()`.


Files
Plots
Packages
Help
Viewer
Presentation
R: Get or Set Working Directory
Find in Topic
getwd {base}
R Documentation
Get or Set Working Directory
Description
getwd returns an absolute filepath representing the current working directory of the R process; setwd(dir) is used to set the working directory to dir.
Usage
getwd()
setwd(dir)
Arguments

```r
library() # check your already installed packages
.libPaths() # check the file path(s) where your packages are installed
```

```
[1] "C:/Users/kaspe/AppData/Local/R/win-library/4.2"
[2] "C:/Program Files/R/R-4.2.2/library"
```

Let's install and load the tidyverse, a collection of packages that you will likely use heavily in your R journey. The Tidyverse is my personal favorite collection of functions, with excellent and intuitive functionality and thorough documentation.

Install packages with the `install.packages()` function.

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

This command downloads and saves all of the functions within the Tidyverse collection of packages into your R package library. **In order to use a certain package's functions during your R session, you have to tell R explicitly to load them** with the `library()` function, with the name of the package as an argument.

```r
library(tidyverse) # load tidyverse
```

```
Warning: package 'tidyverse' was built under R version 4.2.3

Warning: package 'ggplot2' was built under R version 4.2.3

Warning: package 'tibble' was built under R version 4.2.3

Warning: package 'dplyr' was built under R version 4.2.3

-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.1     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

R prints some helpful information in the console that confirms you have loaded the package, and warnings (under **Conflicts**) about altered functionality that may occur. This is usually due to *shared function names* between base R and a loaded package. **The most recently loaded function will supersede ("mask") a previously loaded function.**

When you begin writing your own functions, it is best practice to give your functions unique names to avoid conflicts.

You now have access to the hundreds of useful functions within the tidyverse collection of packages. A good place to find what functions you have is in the Packages > Library tab. Alternatively, you can find these in the console with `lsf.str("package:your-package-here")`, but this output is less readable. You can generally rely on finding helpful documentation for most packages online.

Working with Data

Now that we've learned some of the basics of working in R, let's talk about how to work with data. There are 5 main data types in R:

Data types

Variables can contain different types of data. These include:

- `Numeric`: 14.5, 11.97
- `Integer`: 1L, 4L, 17L, where the letter L indicates an integer
- `Complex`: 1 + 4i, where i indicates the imaginary component
- `Character` / `string`: "Hello, world", "This is a string", "So is this."
- `Logical`: Boolean, `TRUE` or `FALSE`

You can check what data type a variable is using the `class()` function.

```
a <- 14.94
class(a)
```

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

```
a <- 14L # include L to coerce integer type
class(a)
```

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

```
a <- "14"
class(a)
```

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

```
a <- 14+1i # 1i gives i (sqrt(-1)); i alone will not be recognized
class(a)
```

```
[1] "complex"
```

```
a <- TRUE
class(a)
```

```
[1] "logical"
```

Data Structures

Next, let's go over a few ways to enter data directly into R. It's very useful to learn how to run analyses on smaller (simulated or real) datasets before applying methods to your larger dataset.

Vectors

A **vector** is a one-dimensional data structure that contains a single data type. As you learned already, you can create vectors with `c()` or with the `:` operator.

```
my_vector <- 1:10
class(my_vector)
```

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

```
(my_vector2 <- c("string1","string2",4))
```

```
[1] "string1" "string2" "4"
```

Notice that if you check the class of `my_vector2`, it is "character". Because vectors can only contain one data type, the 3rd item is coerced to being a character vector. If you want to store multiple data types in one R object, **lists** serve that purpose (see below).

Matrices

A matrix is a two-dimensional data structure for a single data type. Let's create an empty 3x4 matrix with the `matrix()` function.

```r
matrix(nrow = 3, ncol = 4)
```

```
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

This first matrix is empty, as indicated by the `NA`s - R's shorthand for missing ("Not Available") values. If you read the R documentation on matrices, you'll notice that we skipped over the first argument, `data`, that accepts a data vector. Let's use the data argument to make a matrix with values from `my_vector`.

```r
matrix(my_vector,nrow=2,ncol=5)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Note that the values are distributed column-wise by default. Use the `byrow` argument to fill values row-wise.

```r
matrix(my_vector,nrow=2,ncol=5,byrow=TRUE)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

Another useful way to create matrices is by combining vectors together by row, with `rbind()` or by column, with `cbind()`.

```
# First, let's clean up our Environment
rm(list = ls())

# Create a vector with random values from standard normal distribution
a <- rnorm(10)
b <- (a^2)/4

# Combine row-wise
(byrow <- rbind(a,b))
```

```
        [,1]       [,2]      [,3]      [,4]       [,5]       [,6]         [,7]
a -1.8286139 0.39775592 0.8296600 1.2587239 0.63018273 -0.7535539 0.0193204239
b  0.8359572 0.03955244 0.1720839 0.3960965 0.09928257  0.1419609 0.0000933197
       [,8]       [,9]     [,10]
a 0.6581423 -1.7480467 0.6539880
b 0.1082878  0.7639168 0.1069251
```

```
# Combine column-wise
(bycol <- cbind(a,b))
```

```
                a            b
 [1,] -1.82861388 0.8359571819
 [2,]  0.39775592 0.0395524429
 [3,]  0.82965996 0.1720839124
 [4,]  1.25872395 0.3960964946
 [5,]  0.63018273 0.0992825678
 [6,] -0.75355388 0.1419608632
 [7,]  0.01932042 0.0000933197
 [8,]  0.65814227 0.1082878131
 [9,] -1.74804671 0.7639168244
[10,]  0.65398803 0.1069250873
```

Notice how the rows and columns are labeled. You can get this with the `rownames()` and `colnames()` functions. You can use these functions to change row and/or column names, as well.

```
rownames(byrow)
```

```
[1] "a" "b"
```

```r
colnames(bycol)
```

```
[1] "a" "b"
```

```r
colnames(bycol) <- c("Variable1","Variable2")
bycol
```

```
       Variable1    Variable2
 [1,] -1.82861388 0.8359571819
 [2,]  0.39775592 0.0395524429
 [3,]  0.82965996 0.1720839124
 [4,]  1.25872395 0.3960964946
 [5,]  0.63018273 0.0992825678
 [6,] -0.75355388 0.1419608632
 [7,]  0.01932042 0.0000933197
 [8,]  0.65814227 0.1082878131
 [9,] -1.74804671 0.7639168244
[10,]  0.65398803 0.1069250873
```

Lists

Lists are a more flexible way to store data of multiple types and dimensions. Each element in a list can store any type of R object.

```r
(my_first_list <- list(a,b,bycol,byrow))
```

```
[[1]]
 [1] -1.82861388  0.39775592  0.82965996  1.25872395  0.63018273 -0.75355388
 [7]  0.01932042  0.65814227 -1.74804671  0.65398803

[[2]]
 [1] 0.8359571819 0.0395524429 0.1720839124 0.3960964946 0.0992825678
 [6] 0.1419608632 0.0000933197 0.1082878131 0.7639168244 0.1069250873

[[3]]
       Variable1    Variable2
 [1,] -1.82861388 0.8359571819
 [2,]  0.39775592 0.0395524429
 [3,]  0.82965996 0.1720839124
```

```
 [4,]  1.25872395 0.3960964946
 [5,]  0.63018273 0.0992825678
 [6,] -0.75355388 0.1419608632
 [7,]  0.01932042 0.0000933197
 [8,]  0.65814227 0.1082878131
 [9,] -1.74804671 0.7639168244
[10,]  0.65398803 0.1069250873

[[4]]
        [,1]       [,2]      [,3]      [,4]       [,5]       [,6]         [,7]
a -1.8286139 0.39775592 0.8296600 1.2587239 0.63018273 -0.7535539 0.0193204239
b  0.8359572 0.03955244 0.1720839 0.3960965 0.09928257  0.1419609 0.0000933197
       [,8]       [,9]     [,10]
a 0.6581423 -1.7480467 0.6539880
b 0.1082878  0.7639168 0.1069251
```

The elements of your list can be named. Extract and replace names with the `names()` function.

```
names(my_first_list) <- c("Vector_a","Vector_b","bycol","byrow")
my_first_list
```

```
$Vector_a
 [1] -1.82861388  0.39775592  0.82965996  1.25872395  0.63018273 -0.75355388
 [7]  0.01932042  0.65814227 -1.74804671  0.65398803

$Vector_b
 [1] 0.8359571819 0.0395524429 0.1720839124 0.3960964946 0.0992825678
 [6] 0.1419608632 0.0000933197 0.1082878131 0.7639168244 0.1069250873

$bycol
        Variable1    Variable2
 [1,] -1.82861388 0.8359571819
 [2,]  0.39775592 0.0395524429
 [3,]  0.82965996 0.1720839124
 [4,]  1.25872395 0.3960964946
 [5,]  0.63018273 0.0992825678
 [6,] -0.75355388 0.1419608632
 [7,]  0.01932042 0.0000933197
 [8,]  0.65814227 0.1082878131
 [9,] -1.74804671 0.7639168244
[10,]  0.65398803 0.1069250873
```

```
$byrow
        [,1]       [,2]      [,3]      [,4]       [,5]       [,6]         [,7]
a -1.8286139 0.39775592 0.8296600 1.2587239 0.63018273 -0.7535539 0.0193204239
b  0.8359572 0.03955244 0.1720839 0.3960965 0.09928257  0.1419609 0.0000933197
       [,8]       [,9]     [,10]
a 0.6581423 -1.7480467 0.6539880
b 0.1082878  0.7639168 0.1069251
```

Notice how now, printing `my_first_list` to the console shows the new names next to the `$` operator. We'll get to what `$` does in the **Indexing** section below.

Data Frames

Data frames, created with the `data.frame()` function, is a specialized list that can hold multiple data types, but requires each element to have the same length.

```
x <- 1:10
y <- x^2

(df <- data.frame(x,y))
```

```
    x   y
1   1   1
2   2   4
3   3   9
4   4  16
5   5  25
6   6  36
7   7  49
8   8  64
9   9  81
10 10 100
```

```
# Data frames can hold multiple data types. Let's show this by adding another column to a
z <- sample(LETTERS,10,replace=TRUE)
(df <- cbind(df,z))
```

```
    x   y z
1   1   1 A
```

```
2   2   4 U
3   3   9 U
4   4  16 C
5   5  25 X
6   6  36 M
7   7  49 V
8   8  64 D
9   9  81 V
10 10 100 W
```

Indexing

Indexing is a way to access or replace values contained in vectors, matrices, data tables, or lists. Remember the `$` operator we saw above? If you read the Extract documentation from running `help('$')`, R gives you a long list of possible ways to use an extract operator, including `x[i]`, `x[i,j]`, `x[[i,j,]]`, and `x$name`.

The `$` operator allows us to select an element of a list or data frame by name. Type `df$` in your console window. You should see a list of available options (in alphabetical order) to auto complete this expression. You probably noticed that the options are all the column names of `df`. Hit `Tab` then `Enter` to execute the first one.

```
df$x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Whereas `$` selects elements by name, `[]` selects elements by position. With a one-dimensional vector, `[i]` gives the ith term. You can combine indexing with `:` or `c()` to select multiple values, or with the assignment operator `<-` to replace specific elements.

```
# Get the 4th element of vector a
a[4]
```

```
[1] 1.258724
```

```
# Select 1st, 8th, 9th values
a[c(1,8,9)]
```

```
[1] -1.8286139  0.6581423 -1.7480467
```

```r
# Select the 2nd to last value
a[length(a)-1]
```

```
[1] -1.748047
```

```r
# Replace the 4th element with a new value
a[4] <- 9
a
```

```
 [1] -1.82861388  0.39775592  0.82965996  9.00000000  0.63018273 -0.75355388
 [7]  0.01932042  0.65814227 -1.74804671  0.65398803
```

With a two-dimensional matrix or data frame, `[i,j]` returns a vector containing the value in row *i*, column *j*. If you leave *i* or *j* blank (e.g., `[i,]` or `[,j]`), R will return the entire *i*th row or *j*th column. Let's use this to get some specific values from our data frame, `df`.

```r
df[,] # return all rows and all columns
```

```
    x   y z
1   1   1 A
2   2   4 U
3   3   9 U
4   4  16 C
5   5  25 X
6   6  36 M
7   7  49 V
8   8  64 D
9   9  81 V
10 10 100 W
```

```r
df[2,] # return all columns in the second row
```

```
  x y z
2 2 4 U
```

```r
df[,3] # return all rows in the the third column
```

```
 [1] "A" "U" "U" "C" "X" "M" "V" "D" "V" "W"
```

```r
df[1,3] # return the third values of the first row
```

```
[1] "A"
```

Double brackets `[[]]` are a slightly more complicated way to index and are most useful when working with lists. Let's learn by example by indexing the first element of `my_first_list`:

```r
my_first_list[1]
```

```
$Vector_a
 [1] -1.82861388  0.39775592  0.82965996  1.25872395  0.63018273 -0.75355388
 [7]  0.01932042  0.65814227 -1.74804671  0.65398803
```

```r
my_first_list[[1]]
```

```
 [1] -1.82861388  0.39775592  0.82965996  1.25872395  0.63018273 -0.75355388
 [7]  0.01932042  0.65814227 -1.74804671  0.65398803
```

Can you see the difference in the console output? Hint: try checking the `class()` and `length()` of these outputs:

```r
class(my_first_list[1])
```

```
[1] "list"
```

```r
length(my_first_list[1])
```

```
[1] 1
```

```r
class(my_first_list[[1]])
```

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

```r
length(my_first_list[[1]])
```

```
[1] 10
```

`my_first_list[1]` returns the 1st list item of `my_first_list`, but `my_first_list[[1]]` returns the *contents* of that list item. This is a very important distinction, so be sure you know which behavior is appropriate for your indexing goals.

Data Exploration

Using built-in datasets

As a beginneR, it's very useful to learn how to work with data using some of R's many built-in datasets for practice. Get a list of built-in datasets with `data()`.

Let's learn more about data frames by playing with the built-in `mtcars` dataset. Print it out in your console to view the contents of this data frame. (You can also see a more full view with the `View()` function.)

```
mtcars
```

```
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
```

```
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Let’s get some more information on the mtcars dataset by using the `?` operator to get help.

```
?mtcars
```

R: Motor Trend Car Road Tests ▾ Find in Topic

Motor Trend Car Road Tests

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

```
mtcars
```

Format

A data frame with 32 observations on 11 (numeric) variables.

[, 1]	mpg	Miles/(US) gallon
[, 2]	cyl	Number of cylinders
[, 3]	disp	Displacement (cu.in.)
[, 4]	hp	Gross horsepower
[, 5]	drat	Rear axle ratio
[, 6]	wt	Weight (1000 lbs)
[, 7]	qsec	1/4 mile time
[, 8]	vs	Engine (0 = V-shaped, 1 = straight)
[, 9]	am	Transmission (0 = automatic, 1 = manual)
[,10]	gear	Number of forward gears
[,11]	carb	Number of carburetors

How else can we get information about the structure and content of data frames? A few useful functions to know are: `str()`, for getting the structure of an R object, `head()` and `tail()` for getting the first and last n (default 6) rows, `names()`, for getting names of an R object (column names, list item names), and `dim()`, for getting row x column dimensions of an R matrix, array, or data frame.

Let's try each of these below:

```r
class(mtcars)
```

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

```r
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
tail(mtcars)
```

```
                mpg cyl  disp  hp drat    wt qsec vs am gear carb
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

```r
str(mtcars)
```

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

```r
names(mtcars)
```

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

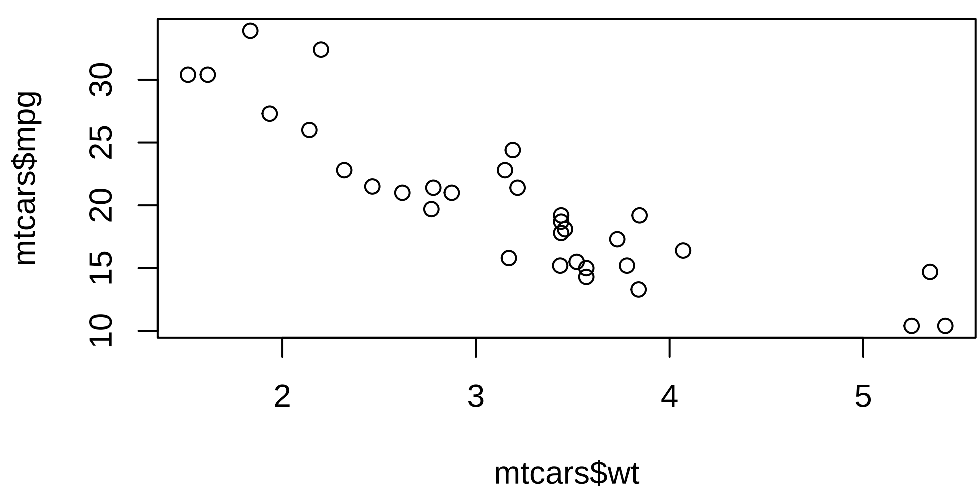
```r
dim(mtcars)
```

```
[1] 32 11
```

You can get a quick view of the relationship between two variables by plotting. We don't have time to get into the nitty gritty of the many ways to generate plots in R, so I suggest reading up on the `ggplot2` package (conveniently, part of your tidyverse installation). For very quick quality checks, the base `plot()` function is a convenient way to see relationships between x and y variables.

Let's plot vehicle weight (`wt`) against miles per gallon (`mpg`) in the `mtcars` dataset.

```r
plot(x = mtcars$wt, y = mtcars$mpg)
```



Loading external data

Most data analysis requires working with data saved in a spreadsheet or text document. Loading your data into R is quick and easy - provided you tell R the correct location to look on your device!

If you downloaded this .Rproj and opened it in your R session, you should have an "example.csv" file saved in your project repository. You can confirm this in your file explorer, or within R by navigating to the Files tab of the help pane. You can also confirm you are in an active project folder with `getwd()`. Print the contents of your current directory with `dir()`.

```r
dir()
```

```
 [1] "2024-05-03_Biostatistics-Workshop-Intro.pptx"
 [2] "Biostats_Workshop.Rproj"
 [3] "example.csv"
 [4] "Figures"
 [5] "Intro_to_R.html"
 [6] "Intro_to_R.pdf"
 [7] "Intro_to_R.qmd"
 [8] "Intro_to_R.rmarkdown"
 [9] "Intro_to_R_files"
[10] "README.md"
```

Let's clear our Environment then load the data in "example.csv" with the `read.csv()` function. (Note: the Tidyverse `readr` package provides an analogous function, `readr::read_csv()`, to load data as a `tibble()`, a tidyverse-specific type of data structure you can read about with `?tibble`.

```r
rm(list=ls())
df <- read.csv("example.csv")
str(df)
```

```
'data.frame':	20 obs. of  4 variables:
 $ Date       : chr  "4/19/2024" "4/19/2024" "4/19/2024" "4/19/2024" ...
 $ Subject    : chr  "sbj001" "sbj002" "sbj003" "sbj004" ...
 $ Body_mass_g: int  23 24 20 20 20 25 24 24 21 21 ...
 $ Treatment  : chr  "Control" "Control" "Control" "Control" ...
```

Other useful functions for loading data in .txt files are `read.table()` and `readr::read_table()`. Functions for loading data from Google Sheets are provided by the `googlesheets4` package, and functions for .xls and .xlsx files by the `readxl` package.