Introducing R

CJ 702: Advanced Criminal Justice Statistics

Thomas Bryan Smith*

February 03, 2025

Contents

1 Working Directories and Libraries

```
# Find the current working directory (cwd):
getwd()
```

```
## [1] "c:/Users/Tom Smith/Documents/GitHub/CJStatsOM/1 Introducing R"
```

```
# ========================================================================== #

# If the current working directory does not match the location of the folder
# within which you want to work, you can change the directory with the
# following code:

# setwd("C:/the/working/directory")
# Windows style formatting (note the forward slash)

# setwd("/the/working/directory")
```

*University of Mississippi, tbsmit10@olemiss.edu

```r
# Apple style formatting

# To run this code, you will need to replace the text between the
# speech marks, "", with your own working directory. This will
# depend on where you have saved your scripts, data, etc.

# The leading hashtag, #, "comments out" the code so that it is not
# interpreted by R. If you remove it, the code becomes "active"
# and will be interpreted.

# ============================================================================ #

# Having set your working directory, next you will want to install any packages
# you want to work with in your project(s). You do so with the following code:
# install.packages("tidyverse")

# Once the package is installed, you need to load it. You will need to do
# this in EVERY session, so the best practice is to include it at the
# beginning of each script.
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.3.3

## Warning: package 'ggplot2' was built under R version 4.3.3

## Warning: package 'tidyr' was built under R version 4.3.3

## Warning: package 'readr' was built under R version 4.3.3

## Warning: package 'dplyr' was built under R version 4.3.3

## Warning: package 'stringr' was built under R version 4.3.3

## Warning: package 'lubridate' was built under R version 4.3.3

## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.4     v tidyr     1.3.1
## v purrr     1.0.2
## -- 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 become error
```

2 Objects

```r
# R is an inherently mathematical language, and operates in a way that resembles
# algebra. To "store" anything within R (text, single numbers, data, etc.)
# you need to "assign" it to an "object". Once assigned, this object represents
# the assigned value or data until it is overwritten, deleted, or you close the
# session.

# Assign the numeric value, 8, to an object by the name "x":
x <- 8

# Print the "x" object:
print(x)
```

```
## [1] 8
```

```r
# This can also be achieved by simply entering the object:
x
```

```
## [1] 8
```

```r
# You can also enter and print the assigned value simultaneously:
(name <- "Mark")
```

```
## [1] "Mark"
```

```r
# Depending on the type of the object,
# you can perform mathematical operations:
x + 2
```

```
## [1] 10
```

```r
# These output of these mathematical operations can be assigned:
(result <- x + 2)
```

```
## [1] 10
```

```r
# Then you can continue to work with the new object:
(result / 2)
```

```
## [1] 5
```

```r
(result * 5)
```

```
## [1] 50
```

```r
# You can save objects with the save() function:
save(x, result, file = "myobjects.rda")
# Note that this will save them to your current working directory!

# You can view what is currently stored in your
# "workspace" (all of your objects) with the ls() function:
ls()
```

```
## [1] "name"   "result" "x"
```

```
# Then the rm() function can be used to remove objects from that workspace:
rm(x, result, name)

# You can also completely wipe your workspace of all objects:
rm(list = ls())

# Now your workspace will be empty:
ls()
```

```
## character(0)
```

```
# ...and you will not be able to "call" any of these objects:
# result

# ...but you can load in any saved objects with the load() function:
load(file = "myobjects.rda")

# The saved objects are now back in the workspace:
ls()
```

```
## [1] "result" "x"
```

```
# Before moving on, let's wipe the workspace again:
rm(list = ls())
```

3 Vectors and Matrices

```
# To manually creating a numeric vector, we use the
# 'combine', c(), function:
(x <- c(1, 2, 3, 4))
```

```
## [1] 1 2 3 4
```

```
# In this case, we can do the same vector by generating a sequence:
y <- 1:4

# or, using the seq() function:
z <- seq(from = 1, to = 4)

# The seq() function can also be used to generated more complex numeric vectors:
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```
# Then the rep() function can be used to generate vectors with repeating values:
rep(2, times = 10)
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2
```

```r
rep(1:2, times = 10)
```

```
##  [1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
```

```r
rep(1:2, length.out = 8)
```

```
## [1] 1 2 1 2 1 2 1 2
```

```r
# Sometimes it can be helpful to create a sequence, starting at 1,
# that iterates along another vector that does not start at 1:
(x <- 51:65)
```

```
##  [1] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
```

```r
seq_along(x)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```r
# Vectors are not always numeric, they can be character vectors:
(x <- c("Benjamin", "Kathryn", "Jean", "Jonathan"))
```

```
## [1] "Benjamin" "Kathryn"  "Jean"     "Jonathan"
```

```r
# ...or they can be boolean (TRUE, FALSE) vectors:
(x <- c(TRUE, FALSE, FALSE, TRUE))
```

```
## [1]  TRUE FALSE FALSE  TRUE
```

```r
# ============================================================================ #

# Mathematical operations function differently for vectors of length > 1.
# Here are some examples:
(x <- 1:4)
```

```
## [1] 1 2 3 4
```

```r
(y <- 0:3)
```

```
## [1] 0 1 2 3
```

```r
# [1 2 3 4] + 1
1:4 + 1
```

```
## [1] 2 3 4 5
```

```
x + 1
```

```
## [1] 2 3 4 5
```

```
# [1 2 3 4] + [0 1]
1:4 + 0:1
```

```
## [1] 1 3 3 5
```

```
x + 0.1
```

```
## [1] 1.1 2.1 3.1 4.1
```

```
# [1 2 3 4] + [0 1 2 3]
1:4 + 0:3
```

```
## [1] 1 3 5 7
```

```
x + y
```

```
## [1] 1 3 5 7
```

```
# These rules also apply to other mathematical operations:
x - y
```

```
## [1] 1 1 1 1
```

```
x * y
```

```
## [1]  0  2  6 12
```

```
x / y
```

```
## [1]      Inf 2.000000 1.500000 1.333333
```

```
# Working with matrices is a little more complicated, and requires
# knowledge of matrix algebra. Being as most of you do not work
# with relational data, focusing more on single variables (vectors),
# we will not discuss mathematical operations for matrices.

# ============================================================================ #

# Now that you understand vectors, let's move on to matrices.
# Creating a simple matrix:
(mat <- matrix(c(0,1,0, 15,18,10, 2,6,0, 126,75,50), nrow = 3))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0   15    2  126
## [2,]    1   18    6   75
## [3,]    0   10    0   50
```

```r
# Note that the matrix is populated column-by-column. The nrow
# option defines the number of rows in the matrix. By splitting
# the input vector into chunks of 3, we can easily recognize
# what each column will look like ahead of time.
# When you change the number of rows, it helps to change the input:
(mat <- matrix(c(0,1,0,15, 18,10,2,6, 0,126,75,50), nrow = 4))
```

```
##      [,1] [,2] [,3]
## [1,]    0   18    0
## [2,]    1   10  126
## [3,]    0    2   75
## [4,]   15    6   50
```

```r
# ...or, following best practice:
(mat <- matrix(c(0, 1, 0,         # column 1
                 15, 18, 10,      # column 2
                 2, 6, 0,         # column 3
                 126, 75, 50),    # column 4
               nrow = 3))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0   15    2  126
## [2,]    1   18    6   75
## [3,]    0   10    0   50
```

```r
# If you want to find the number of rows and columns of the matrix:
nrow(mat)
```

```
## [1] 3
```

```r
ncol(mat)
```

```
## [1] 4
```

```r
# The dim() function finds both and outputs them as a vector,
# starting with the rows (4) and ending with the columns (3):
dim(mat)
```

```
## [1] 3 4
```

```r
# Remember, the output of ANY function can be assigned to an object:
(x <- dim(mat))
```

```
## [1] 3 4
```

```r
# Note that, right now, your matrix has no row and column names:
mat
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0   15    2  126
## [2,]    1   18    6   75
## [3,]    0   10    0   50
```

```r
rownames(mat)
```

```
## NULL
```

```r
colnames(mat)
```

```
## NULL
```

```r
# You can assign vectors of names to these rows and columns:
rownames(mat) <- c("Benjamin", "Kathryn", "Jonathan")
colnames(mat) <- c("sex", "age", "delinquencies", "ses")

# Now, when you print the matrix, it will show the names:
mat
```

```
##          sex age delinquencies ses
## Benjamin   0  15             2 126
## Kathryn    1  18             6  75
## Jonathan   0  10             0  50
```

```r
# Using the as.data.frame() function you can convert this matrix
# into a 'data frame', R's version of a dataset:
as.data.frame(mat)
```

```
##          sex age delinquencies ses
## Benjamin   0  15             2 126
## Kathryn    1  18             6  75
## Jonathan   0  10             0  50
```

```r
# However, I would recommend learning to work with
# 'tibbles', the tidyverse equivalent of R's data frame:
as_tibble(mat)
```

```
## # A tibble: 3 x 4
##     sex   age delinquencies   ses
##   <dbl> <dbl>         <dbl> <dbl>
## 1     0    15             2   126
## 2     1    18             6    75
## 3     0    10             0    50
```

```r
# Let's assign this tibble to an object named 'df':
df <- as_tibble(mat, rownames = "name")

# Now, we can save these data as a comma-separated values (.csv) file:
write.csv(df, file = "my_data.csv")

# or as an excel spreadsheet (.xlsx):
# install.packages("readxl")
library(openxlsx)
```

```
## Warning: package 'openxlsx' was built under R version 4.3.3
```

```r
write.xlsx(df, file = "my_data.xlsx")

# or as a Stata dataset (.dta):
# install.packages("haven")
library(haven)
```

```
## Warning: package 'haven' was built under R version 4.3.3
```

```r
write_dta(df, "my_data.dta")

# or as a Feather (.feather):
# install.packages("feather")
library(feather)
```

```
## Warning: package 'feather' was built under R version 4.3.3
```

```r
write_feather(df, "my_data.feather")
```

4 Logical Operations

```r
# Logical operations are useful for identifying specific case(s) in your data.
# So, let's load our data back into R:
read_csv("my_data.csv")
```

```
## New names:
## Rows: 3 Columns: 6
## -- Column specification
## -------------------------------------------------------- Delimiter: "," chr
## (1): name dbl (5): ...1, sex, age, delinquencies, ses
## 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.
## * `` -> `...1`

## # A tibble: 3 x 6
##    ...1 name       sex   age delinquencies   ses
##   <dbl> <chr>    <dbl> <dbl>         <dbl> <dbl>
## 1     1 Benjamin     0    15             2   126
## 2     2 Kathryn      1    18             6    75
## 3     3 Jonathan     0    10             0    50
```

```r
read_dta("my_data.dta")
```

```
## # A tibble: 3 x 5
##   name       sex   age delinquencies   ses
##   <chr>    <dbl> <dbl>         <dbl> <dbl>
## 1 Benjamin     0    15             2   126
## 2 Kathryn      1    18             6    75
## 3 Jonathan     0    10             0    50
```

```r
read_feather("my_data.feather")
```

```
## # A tibble: 3 x 5
##   name       sex   age delinquencies   ses
##   <chr>    <dbl> <dbl>         <dbl> <dbl>
## 1 Benjamin     0    15             2   126
## 2 Kathryn      1    18             6    75
## 3 Jonathan     0    10             0    50
```

```r
# What have we forgotten to do here? Assign it to an object!
df <- read.xlsx("my_data.xlsx")

# The openxlsx package will read the data as a data frame, not a tibble.
df
```

```
##       name sex age delinquencies ses
## 1 Benjamin   0  15             2 126
## 2  Kathryn   1  18             6  75
## 3 Jonathan   0  10             0  50
```

```r
# You will need to convert it back into a tibble manually. You can do this
# in the same way you did previously, using the as_tibble() function.
df <- as_tibble(df)

# You can combine these functions using the "pipe operator" (%>% or |>).
# This operator functions a little like a mathematical operation,
# but it is instead used to chain multiple functions.
# The chained functions are executed from left to right:
read.xlsx("my_data.xlsx") %>% as_tibble()   # tidyverse (magrittr)
```

```
## # A tibble: 3 x 5
##   name       sex   age delinquencies   ses
##   <chr>    <dbl> <dbl>         <dbl> <dbl>
## 1 Benjamin     0    15             2   126
## 2 Kathryn      1    18             6    75
## 3 Jonathan     0    10             0    50
```

```r
read.xlsx("my_data.xlsx") |> as_tibble()    # base R
```

```
## # A tibble: 3 x 5
##   name       sex   age delinquencies   ses
##   <chr>    <dbl> <dbl>         <dbl> <dbl>
## 1 Benjamin     0    15             2   126
## 2 Kathryn      1    18             6    75
## 3 Jonathan     0    10             0    50
```

```r
# Now, let's read the .xlsx file, convert it into a tibble, assign
# the tibble to an object, and print the tibble. All at the same time!
(df <- read.xlsx("my_data.xlsx") |> as_tibble())
```

```
## # A tibble: 3 x 5
##   name       sex   age delinquencies   ses
##   <chr>    <dbl> <dbl>         <dbl> <dbl>
## 1 Benjamin     0    15             2   126
## 2 Kathryn      1    18             6    75
## 3 Jonathan     0    10             0    50
```

```
# ============================================================================ #

# Having loaded in our dataset, let's perform some logical operations.
# First, we extract the variable on which we want to perform the operations:
(age <- df$age)
```

```
## [1] 15 18 10
```

```
# Now, let's check to see if any of the respondents are 18 years old:
age == 18
```

```
## [1] FALSE  TRUE FALSE
```

```
# Are any respondents NOT 18 years old?
age != 18
```

```
## [1]  TRUE FALSE  TRUE
```

```
# Are any respondents older than 10 years?
age > 10
```

```
## [1]  TRUE  TRUE FALSE
```

```
# Are any respondents less than or equal to 15 years?
age <= 15
```

```
## [1]  TRUE FALSE  TRUE
```

```
# Are any respondents less than 12 years OR greater than 17 years?
age < 12 | age > 17
```

```
## [1] FALSE  TRUE  TRUE
```

```
# Are any respondents greater than 12 AND less than 17?
age > 12 & age < 17
```

```
## [1]  TRUE FALSE FALSE
```

```
# So far, all of these logical operations have produced
# logical (TRUE / FALSE) vectors. Instead, we might want to
# know the position within the data. The which() function
# can be used to return a vector of rows (indices) that
# fulfil the conditions of the logical operation:
which(age > 12 & age < 17)
```

```
## [1] 1
```

```r
# You can invert (negate) these logical operations with a
# leading exclamation mark (!):
!(age > 12 & age < 17)
```

```
## [1] FALSE  TRUE  TRUE
```

```r
# This also works if you assign the logical vector:
(logic <- age > 12 & age < 17)
```

```
## [1]  TRUE FALSE FALSE
```

```r
!logic
```

```
## [1] FALSE  TRUE  TRUE
```

```r
# The %in% logical operator is unique in that it can be used
# to compare multiple vectors of length > 1.

# Is the number '20' in the age vector?
20 %in% age
```

```
## [1] FALSE
```

```r
# What about the number '18'?
18 %in% age
```

```
## [1] TRUE
```

```r
# Are any of the values in the age vector also in
# a vector consisting of the numbers 10 and 18?
age %in% c(10, 18)
```

```
## [1] FALSE  TRUE  TRUE
```

```r
# These logical operators can be combined to create
# increasingly complicated logical statements.
which(!(age %in% c(10, 18)))
```

```
## [1] 1
```

```r
# Just make sure you interpret them carefully and correctly!
# In this case, we checked to see WHICH respondents' ages
# do NOT (!) appear in the vector: [10, 18].

# Logical vectors can be interpreted as a binary variable.
# They are simply encoded as "FALSE" and "TRUE", rather than 0 and 1.
# As a result, they are easily converted into a binary numeric vector:
as.numeric(age == 18)
```

```
## [1] 0 1 0
```

```r
# So, if you want to count how many respondents fulfill the condition:
sum(age == 18)
```

```
## [1] 1
```

```r
# Or, if you want to find the proportion of respondents who fulfill
# the condition:
mean(age == 18)
```

```
## [1] 0.3333333
```

```r
# For a percentage, just multiply by 100:
mean(age == 18) * 100
```

```
## [1] 33.33333
```

```r
# For a rate per 1,000, just multiply by 1,000:
mean(age == 18) * 1000
```

```
## [1] 333.3333
```

```r
# ============================================================================ #

# For the following exercises and the next section, we are going to load
# in one of R's "built-in" practice datasets, "USArrests", a data frame
# reporting violent crime rates in each US state:
data("USArrests")
# In this special case, we do not need to assign the data() function, because
# this function is simple importing the USArrests object from R's files.

# Let's look at the first 6 observations using the head() function:
head(USArrests)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

```r
# Let's look at the first 20 observations instead:
head(USArrests, 20)
```

```
##             Murder Assault UrbanPop Rape
## Alabama       13.2     236       58 21.2
## Alaska        10.0     263       48 44.5
## Arizona        8.1     294       80 31.0
```

```
## Arkansas       8.8     190       50 19.5
## California     9.0     276       91 40.6
## Colorado       7.9     204       78 38.7
## Connecticut    3.3     110       77 11.1
## Delaware       5.9     238       72 15.8
## Florida       15.4     335       80 31.9
## Georgia       17.4     211       60 25.8
## Hawaii         5.3      46       83 20.2
## Idaho          2.6     120       54 14.2
## Illinois      10.4     249       83 24.0
## Indiana        7.2     113       65 21.0
## Iowa           2.2      56       57 11.3
## Kansas         6.0     115       66 18.0
## Kentucky       9.7     109       52 16.3
## Louisiana     15.4     249       66 22.2
## Maine          2.1      83       51  7.8
## Maryland      11.3     300       67 27.8
```

```
# "USArrests" is a little long, so let's reassign it to the "df" object
df <- USArrests

# Now we check to see if the data were assigned:
head(df)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

5 Performing Simple Statistical Functions

```
# Say we're interested in calculating descriptive statistics for murder.
# First, let's extract the murder variable from the dataset:
murder <- df$Murder
murder
```

```
##  [1] 13.2 10.0  8.1  8.8  9.0  7.9  3.3  5.9 15.4 17.4  5.3  2.6 10.4  7.2  2.2
## [16]  6.0  9.7 15.4  2.1 11.3  4.4 12.1  2.7 16.1  9.0  6.0  4.3 12.2  2.1  7.4
## [31] 11.4 11.1 13.0  0.8  7.3  6.6  4.9  6.3  3.4 14.4  3.8 13.2 12.7  3.2  2.2
## [46]  8.5  4.0  5.7  2.6  6.8
```

```
# Median
median(murder)
```

```
## [1] 7.25
```

```r
# Mean
mean(murder)
```

```
## [1] 7.788
```

```r
# Standard Deviation
sd(murder)
```

```
## [1] 4.35551
```

```r
# Minimum and Maximum
min(murder)
```

```
## [1] 0.8
```

```r
max(murder)
```

```
## [1] 17.4
```

```r
# Frequency Tables
## Let's create a nominal variable to work with:
sex <- c(rep("male", 15), rep("female", 8))
sex
```

```
##  [1] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"
##  [9] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "female"
## [17] "female" "female" "female" "female" "female" "female" "female"
```

```r
## Now let's tabulate the results:
table(sex)
```

```
## sex
## female   male
##      8     15
```

```r
# Missing Data
# R does not always "know" what to do with missing data.
# Let's introduce some missingness to the murder variable:
murder <- c(murder, rep(NA, 5))
murder
```

```
##  [1] 13.2 10.0  8.1  8.8  9.0  7.9  3.3  5.9 15.4 17.4  5.3  2.6 10.4  7.2  2.2
## [16]  6.0  9.7 15.4  2.1 11.3  4.4 12.1  2.7 16.1  9.0  6.0  4.3 12.2  2.1  7.4
## [31] 11.4 11.1 13.0  0.8  7.3  6.6  4.9  6.3  3.4 14.4  3.8 13.2 12.7  3.2  2.2
## [46]  8.5  4.0  5.7  2.6  6.8   NA   NA   NA   NA   NA
```

```r
# Now let's try to calculate the mean again:
mean(murder)
```

```
## [1] NA
```

```r
# It doesn't work! You have to tell R whether or not to remove
# missing data from the calculation. Under the default setting
# for some functions (e.g., mean), R will try to complete
# calculations with the "NA" values included. As a result,
# the mean cannot be computed. To remove "NA" values, simply
# set the na.rm option to "TRUE":
mean(murder, na.rm = TRUE)
```

```
## [1] 7.788
```

```r
# Similarly, R does not know how to perform logical functions
# on missing values:
murder > 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [13]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
## [25]  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
## [37] FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE
## [49] FALSE  TRUE    NA    NA    NA    NA    NA
```

```r
# You can check for NA's using the is.na() logical function:
is.na(murder)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
# Counting missingness in the variable:
sum(is.na(murder))
```

```
## [1] 5
```

```r
murder |> is.na() |> sum()
```

```
## [1] 5
```

```r
murder %>% is.na %>% sum
```

```
## [1] 5
```

```r
# Proportion of missingness in the variable:
mean(is.na(murder))
```

```
## [1] 0.09090909
```

```r
murder |> is.na() |> mean()
```

```
## [1] 0.09090909
```

```r
murder %>% is.na %>% mean
```

```
## [1] 0.09090909
```

```r
# Percent missingness:
mean(is.na(murder)) * 100
```

```
## [1] 9.090909
```

```r
(murder |> is.na() |> mean()) * 100
```

```
## [1] 9.090909
```

```r
(murder %>% is.na %>% mean) * 100
```

```
## [1] 9.090909
```

6 Creating Custom Functions

```r
# R is a language that is based on functions. mean(), sd(), min(), and every
# other task you have asked R to perform is based on an underlying function
# stored within the base R installation, or an installed R package.

# You can print the underlying function by entering the function call
# without the succeeding parentheses.

# So, to print the 'mean' function, you would simply enter:
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x000002d762aab4f8>
## <environment: namespace:base>
```

```r
# If this looks like nonsense to you, don't worry, it should.
# The base R mean function does not use R syntax.
# So, let's build a function that DOES use R syntax!

mymean <- function(input, rm_na = FALSE) {
  x <- sum(input, na.rm = rm_na)
  if (rm_na == TRUE){
    y <- sum(!is.na(input))
  } else {
```

```r
    y <- length(input)
  }
  result <- x / y
  return(result)
}

# Now let's test the function!
mymean(murder)
```

```
## [1] NA
```

```r
# Oops! Forgot to remove missing values with the "rm_na" option
# we included in the custom function!
mymean(murder, rm_na = TRUE)
```

```
## [1] 7.788
```

```r
# Compare to the native function:
mean(murder, na.rm = TRUE)
```

```
## [1] 7.788
```

```r
# Then, if we want to print our function in the console:
mymean
```

```
## <srcref: file "" chars 14:11 to 23:1>
## <bytecode: 0x000002d76c4e6338>
```