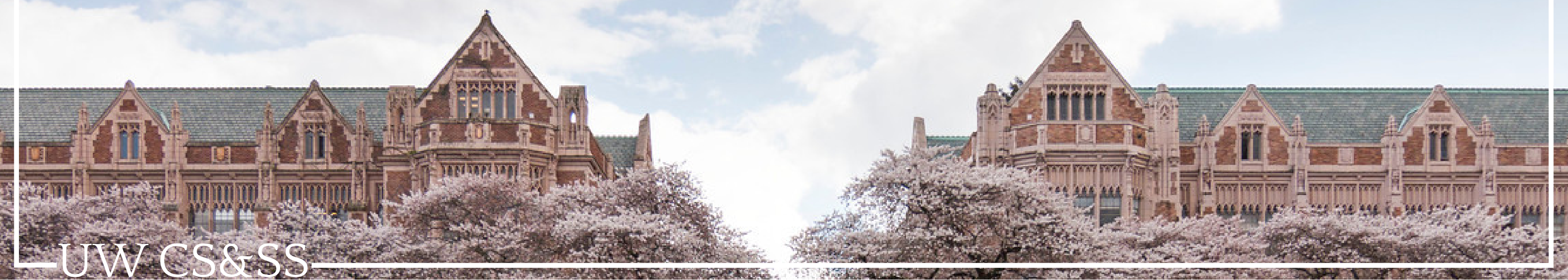
CSSS508, Week 4

R Data Structures

Chuck Lanfear

Oct 21, 2020

Updated: Nov 11, 2020



UW CS&SS

R Data Types

So far we've been manipulating data frames, making visuals, and summarizing. This got you pretty far!

Now we get more in the weeds of programming.

Today is all about *types of data* in R.

Up Until Now

A data frame is really a **list** of **vectors**, where each vector is a column of the same length (number of rows).

But data frames are not the only object we want to have in R (e.g. linear regression output).

We need to learn about **vectors**, **matrices**, and **lists** to do additional things we can't express with `dplyr` syntax.

Vectors

Making Vectors

In R, we call a set of values of the same type a **vector**. We can create vectors using the `c()` function ("c" for **c**ombine or **c**oncatenate).

```
c(1, 3, 7, -0.5)
```

```
## [1]  1.0  3.0  7.0 -0.5
```

Vectors have one dimension: **length**

```
length(c(1, 3, 7, -0.5))
```

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

All elements of a vector are the same type (e.g. numeric or character)!

If you mix character and numeric data, the resulting vector will be *character*.

Element-wise Vector Math

When doing arithmetic operations on vectors, R handles these *element-wise*:

```
c(1, 2, 3) + c(4, 5, 6)
```

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

```
c(1, 2, 3, 4)^3 # exponentiation with ^
```

```
## [1]  1  8 27 64
```

Common operations: `*`, `/`, `exp()` = e^x, `log()` = $\log_e(x)$

Vector Recycling

If we work with vectors of different lengths, R will **recycle** the shorter one by repeating it to make it match up with the longer one:

```
c(0.5, 3) * c(1, 2, 3, 4)
```

```
## [1]  0.5  6.0  1.5 12.0
```

```
c(0.5, 3, 0.5, 3) * c(1, 2, 3, 4) # same thing
```

```
## [1]  0.5  6.0  1.5 12.0
```

Scalars as Recycling

A special case of recycling involves arithmetic with **scalars** (a single number). These are vectors of length 1 that are recycled to make a longer vector:

```
3 * c(-1, 0, 1, 2) + 1
```

```
## [1] -2  1  4  7
```

Warning on Recycling

Recycling doesn't work so well with vectors of incommensurate lengths:

```
c(1, 2, 3, 4) + c(0.5, 1.5, 2.5)
```

```
## Warning in c(1, 2, 3, 4) + c(0.5, 1.5, 2.5): longer object length is
## not a multiple of shorter object length
```

```
## [1] 1.5 3.5 5.5 4.5
```

Try not to let R's recycling behavior catch you by surprise!

Vector-Wise Math

Some functions operate on an entire vector and return *one number* rather than working element-wise:

```r
sum(c(1, 2, 3, 4))
```

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

```r
max(c(1, 2, 3, 4))
```

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

Some others: `min()`, `mean()`, `median()`, `sd()`, `var()`

You've seen these used with `dplyr::summarize()`.

Example: Standardizing Data

Let's say we had some test scores and we wanted to put these on a standardized scale:

$$z_i = \frac{x_i - \text{mean}(x)}{\text{SD}(x)}$$

```r
x <- c(97, 68, 75, 77, 69, 81, 80, 92, 50, 34, 66, 83, 62)
z <- (x - mean(x)) / sd(x)
round(z, 2)
```

```
##  [1]  1.49 -0.23  0.19  0.31 -0.17  0.54  0.48  1.19 -1.30 -2.24 -0.35
## [12]  0.66 -0.58
```

The `scale()` function performs the above operation!

Types of Vectors

`class()` or `str()` will tell you what kind of vector you have. There are a few common types of vectors:

- **numeric**: `c(1, 10*3, 4, -3.14)` [1]
  - **integer**: `0:10`
- **character**: `c("red", "blue", "yellow", "blue")`
- **factor**: `factor(c("red", "blue", "yellow", "blue"))`
- **logical**: `c(FALSE, TRUE, TRUE, FALSE)`

[1] R is perfectly happy with you including a calculation--or any other valid object--as an element!

Generating Numeric Vectors

Numeric vectors contain only numbers, with any number of decimal places.

There are shortcuts for generating common kinds of vectors:

```
seq(-3, 6, by = 1.75) # Sequence from -3 to 6, increments of 1.75
```

```
## [1] -3.00 -1.25  0.50  2.25  4.00  5.75
```

```
rep(c(-1, 0, 1), times = 3) # Repeat c(-1,0,1) 3 times
```

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

```
rep(c(-1, 0, 1), each = 3) # Repeat each element 3 times
```

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

Generating Integer Vectors

Integer vectors are a special case of numeric vectors where all the values are whole numbers.

We can produce them using the `:` shortcut:

```
n <- 12
1:n
```

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

```
n:4
```

```
## [1] 12 11 10  9  8  7  6  5  4
```

You can also specify a single integer using a whole number followed by `L`:

```
class(9L)
```

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

Character Vectors

Character vectors store data as text and typically come up when dealing names, addresses, and IDs:

```
first_names <- c("Andre", "Beth", "Carly", "Dan")
class(first_names)
```

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

Note you can store numbers as character data as well, but you cannot perform math on them unless you convert them to numeric.

Factor Vectors

Factors are categorical data that encode a (modest) number of **levels**, like for sex, experimental group, or geographic region:

```
sex <- factor(c("M", "F", "F", "M"))
sex
```

```
## [1] M F F M
## Levels: F M
```

Character data usually can't go directly into a statistical model[1], but factor data can. It has an *underlying numeric representation*:

```
as.numeric(sex)
```

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

[1] Most R models will automatically convert character data to factors. The default reference is chosen alphabetically.

Logical Vectors

Logical vectors take only TRUE and FALSE values, and are typically the product of logical tests (e.g. `x==5`). We can make logical vectors by defining binary conditions to check for. For example, we can look at which of the first names has at least 4 letters:

```
name_lengths <- nchar(first_names) # number of characters
name_lengths
```

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

```
name_lengths >= 4
```

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

Logical Vectors as Numeric

You can do math with logical vectors, because `TRUE`=1 and `FALSE`=0:

```
name_lengths >= 4
```

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

```
mean(name_lengths >= 4)
```

```
## [1] 0.75
```

What did this last line do?

It told us the *proportion* of name lengths greater than or equal to four!

Combining Logical Conditions

Suppose we are interested in which names have an even number of letters:

```
even_length <- (name_lengths %% 2 == 0)
# %% is the modulo operator: gives remainder when dividing
even_length
```

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

or whose second letter is "a":

```
second_letter_a <- (substr(first_names, start=2, stop=2) == "a")
# substr: substring (portion) of a char vector
second_letter_a
```

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

Logical Operators

- `&` is **AND** (both conditions must be `TRUE` to be `TRUE`):

```
even_length & second_letter_a
```

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

- `|` is **OR** (at least one condition must be `TRUE` to be `TRUE`):

```
even_length | second_letter_a
```

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

- `!` is **NOT** (switches `TRUE` and `FALSE`):

```
!(even_length | second_letter_a)
```

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

Subsetting Vectors

We can **subset** a vector in a number of ways:

- Passing a single index or vector of entries to **keep**:

```
first_names[c(1, 4)]
```

```
## [1] "Andre" "Dan"
```

- Passing a single index or vector of entries to **drop**:

```
first_names[-c(1, 4)]
```

```
## [1] "Beth"  "Carly"
```

Subsetting Vectors

- Passing a logical vector (`TRUE`=keep, `FALSE`=drop):

```
first_names[even_length | second_letter_a]
```

```
## [1] "Beth"  "Carly" "Dan"
```

```
first_names[sex != "F"] # != is "not equal to"
```

```
## [1] "Andre" "Dan"
```

Some Logical/Subsetting Functions

`%in%` lets you avoid typing a lot of logical ORs (`|`):

```
first_names %in% c("Andre", "Carly", "Dan")
```

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

`which()` gives the *indices* of `TRUE`s in a logical vector:

```
which(first_names %in% c("Andre", "Carly", "Dan"))
```

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

Missing Values

Missing values are coded as `NA` entries without quotes:

```
vector_w_missing <- c(1, 2, NA, 4, 5, 6, NA)
```

Even one `NA` "poisons the well": You'll get `NA` out of your calculations unless you remove them manually or with the extra argument `na.rm = TRUE` (in some functions):

```
mean(vector_w_missing)
```

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

```
mean(vector_w_missing, na.rm=TRUE)
```

```
## [1] 3.6
```

Finding Missing Values

WARNING: You can't test for missing values by seeing if they "equal" (`==`) `NA`:

```
vector_w_missing == NA
```

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

Instead, use the `is.na()` function:

```
is.na(vector_w_missing)
```

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

```
mean(vector_w_missing[!is.na(vector_w_missing)])
```

```
## [1] 3.6
```

This is the same as using `mean(na.rm=T)`

`NA` and `%in%`

When testing logical conditions, `NA` will produce an `NA` rather than `TRUE` or `FALSE`:

```
vector_w_missing == 5
```

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

It is noteworthy, however, that `%in%` will handle NAs:

```
vector_w_missing %in% 5
```

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

```
vector_w_missing %in% NA
```

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

It is still usually best to handle NAs *directly*, however!

`Inf` and `NaN`

Sometimes we might get positive or negative infinity (`Inf`, `-Inf`) or `NaN` (**N**ot **A N**umber) from our calculations:

```
c(-2, -1, 0, 1, 2) / 0
```

```
## [1] -Inf -Inf  NaN  Inf  Inf
```

You can check for these using functions like `is.finite()` or `is.nan()`.[1]

```
is.finite(c(-2, -1, 0, 1, 2) / 0)
```

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

```
is.nan(c(-2, -1, 0, 1, 2) / 0)
```

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

[1] Infinity is a number... but isn't finite.

Previewing Vectors

Like with data frames, we can use `head()` and `tail()` to preview vectors:

```
head(letters) # letters is a built-in vector
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
head(letters, 10)
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
tail(letters)
```

```
## [1] "u" "v" "w" "x" "y" "z"
```

Named Vector Entries

We can also index vectors by assigning **names** to the entries.

```
a_vector <- 1:26
names(a_vector) <- LETTERS # capital version of letters
head(a_vector)
```

```
## A B C D E F
## 1 2 3 4 5 6
```

```
a_vector[c("R", "S", "T", "U", "D", "I", "O")]
```

```
##  R  S  T  U  D  I  O
## 18 19 20 21  4  9 15
```

Names are nice for subsetting because they don't depend on your data being in a certain order.

Matrices

Matrices: Two Dimensions

Matrices extend vectors to two **dimensions**: **rows** and **columns**. We can construct them directly using `matrix`.

Note the `byrow=` argument which determines whether the data fill the matrix by row or by column.

```
(a_matrix <- matrix(letters[1:6], nrow=2, ncol=3))
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "c"  "e"
## [2,] "b"  "d"  "f"
```

```
(b_matrix <- matrix(letters[1:6], nrow=2, ncol=3, byrow=TRUE))
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c"
## [2,] "d"  "e"  "f"
```

Binding Vectors

We can also make matrices by *binding* vectors together with `rbind()` (**r**ow **bind**) and `cbind()` (**c**olumn **bind**).

```r
(c_matrix <- cbind(c(1, 2), c(3, 4), c(5, 6)))
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
(d_matrix <- rbind(c(1, 2, 3), c(4, 5, 6)))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Subsetting Matrices

We subset matrices using the same methods as with vectors, except we index them with `[rows, columns]`:

```
a_matrix[1, 2] # row 1, column 2
```

```
## [1] "c"
```

```
a_matrix[1, c(2, 3)] # row 1, columns 2 and 3
```

```
## [1] "c" "e"
```

We can obtain the dimensions of a matrix using `dim()`.

```
dim(a_matrix)
```

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

Note that using `length()` on a matrix will not give you the number of rows or columns but rather the number of elements!

Matrices Becoming Vectors

If a matrix ends up having just one row or column after subsetting, by default R will make it into a vector. You can prevent this behavior using `drop=FALSE`.

```
a_matrix[, 1] # all rows, column 1, becomes a vector
```

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

```
a_matrix[, 1, drop=FALSE] # all rows, column 1, stays a matrix
```

```
##      [,1]
## [1,] "a"
## [2,] "b"
```

Matrix Data Type Warning

Matrices can be numeric, integer, factor, character, or logical, just like vectors. Also like vectors, *all elements must be the same data type*.

```
(bad_matrix <- cbind(1:2, LETTERS[c(6,1)]))
```

```
##      [,1] [,2]
## [1,] "1"  "F"
## [2,] "2"  "A"
```

```
typeof(bad_matrix)
```

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

In this case, everything was converted to character so as not to lose information.

Matrix Dimension Names

We can access dimension names or name them ourselves:

```r
rownames(bad_matrix) <- c("Wedge", "Biggs")
colnames(bad_matrix) <- c("Pilot grade", "Mustache grade")
bad_matrix
```

```
##       Pilot grade Mustache grade
## Wedge "1"         "F"
## Biggs "2"         "A"
```

```r
bad_matrix["Biggs", , drop=FALSE]
```

```
##       Pilot grade Mustache grade
## Biggs "2"         "A"
```

Matrix Arithmetic

Matrices of the same dimensions can have math performed entry-wise with the usual arithmetic operators:

```r
cbind(c_matrix, d_matrix) # look at side by side
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    1    2    3
## [2,]    2    4    6    4    5    6
```

```r
3 * c_matrix / d_matrix
```

```
##      [,1] [,2] [,3]
## [1,]  3.0  4.5    5
## [2,]  1.5  2.4    3
```

Matrix Transposition and Multiplication

To do matrix transpositions, use `t()`.

```
(e_matrix <- t(c_matrix))
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

To do actual matrix multiplication (not entry-wise), use `%*%`.

```
(f_matrix <- d_matrix %*% e_matrix)
```

```
##      [,1] [,2]
## [1,]   22   28
## [2,]   49   64
```

Matrix Inversion

To invert an invertible square matrix, use `solve()`.

```
(g_matrix <- solve(f_matrix))
```

```
##             [,1]       [,2]
## [1,]  1.777778 -0.7777778
## [2,] -1.361111  0.6111111
```

```
f_matrix %*% g_matrix
```

```
##      [,1]          [,2]
## [1,]    1 -3.552714e-15
## [2,]    0  1.000000e+00
```

Note the floating point imprecision: The off-diagonals are *very close to zero* rather than actually zero!

Be careful testing for equality of numbers after calculations--imprecision produces strange results!

Diagonal Matrices

To extract the diagonal of a matrix or make a diagonal matrix (usually the identity matrix), use `diag()`.

```
diag(2)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
diag(g_matrix)
```

```
## [1] 1.7777778 0.6111111
```

Lists

What are Lists?

Lists are an object that can store multiple types of data.

```
(my_list <- list("first_thing"  = 1:5,
                 "second_thing" = matrix(8:11, nrow = 2),
                 "third_thing"  = lm(dist ~ speed, data = cars)))
```

```
## $first_thing
## [1] 1 2 3 4 5
##
## $second_thing
##      [,1] [,2]
## [1,]    8   10
## [2,]    9   11
##
## $third_thing
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Coefficients:
## (Intercept)        speed
##     -17.579        3.932
```

Accessing List Elements

You can access a list element by its name or number in `[[ ]]`, or a `$` followed by its name:

```
my_list[["first_thing"]]
```

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

```
my_list$first_thing
```

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

```
my_list[[1]]
```

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

Why Two Brackets `[[ ]]`?

If you use single brackets to access list elements, you get a **list** back. Double brackets get *the actual element*—as whatever data type it is stored as—in that location in the list.

```
str(my_list[1])
```

```
## List of 1
##  $ first_thing: int [1:5] 1 2 3 4 5
```

```
str(my_list[[1]])
```

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

Note that you can only select a single element at a time using `[[ ]]`, because this would have to return *multiple objects*!

An R function can only return one object at a time—otherwise operations like assignment would be impossible.

Subsetted Lists Can Be of Length > 1

You can use vector-style subsetting to get a sublist of multiple elements:

```
length(my_list[c(1, 2)])
```

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

```
str(my_list[c(1, 2)])
```

```
## List of 2
##  $ first_thing : int [1:5] 1 2 3 4 5
##  $ second_thing: int [1:2, 1:2] 8 9 10 11
```

Regression Output is a List!

```
str(my_list[[3]], list.len=7) # Displaying on first 7 elements
```

```
## List of 12
##  $ coefficients : Named num [1:2] -17.58 3.93
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "speed"
##  $ residuals    : Named num [1:50] 3.85 11.85 -5.95 12.05 2.12 ...
##   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:50] -303.914 145.552 -8.115 9.885 0.194 ...
##   ..- attr(*, "names")= chr [1:50] "(Intercept)" "speed" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:50] -1.85 -1.85 9.95 9.95 13.88 ...
##   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:50, 1:2] -7.071 0.141 0.141 0.141 0.141 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:50] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "speed"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.14 1.27
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##   [list output truncated]
##  - attr(*, "class")= chr "lm"
```

`names()` and List Elements

You can use `names()` to get a vector of list element names:

```
names(my_list[[3]])
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

Data Frames are Lists!

data frames are lists of equal-length vectors.

```r
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 ...
```

```r
length(cars)
```

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

```r
length(cars$dist) # should be same as nrow(cars)
```

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

You Can Treat Data Frames like a Matrix

```r
cars[1, ]
```

```
##   speed dist
## 1     4    2
```

```r
cars[1:5, "speed", drop = FALSE]
```

```
##   speed
## 1     4
## 2     4
## 3     7
## 4     7
## 5     8
```

Base R vs. `dplyr`

Two ways of calculating the same thing: which do you like better?

Classic R:

```
mean(swiss[swiss$Education > mean(swiss$Education), "Education"])
```

`dplyr`:

```
library(dplyr)
swiss %>%
    filter(Education > mean(Education)) %>%
    summarize(mean = mean(Education))
```

Tibbles

`tidyverse` functions often use a type of data frame called a *tibble*. You can create them manually with `tibble()` as with `data.frame()` or convert existing data frames into tibbles using `as_tibble()`. Tibbles display better than data frames: they truncate output and include column types. They also do not convert strings to factors!

Because the `tidyverse` has abolished row names, `tibbles()` have none. You can convert row names to columns using `tibble::rownames_to_column()` or with the `rownames=` argument in `as_tibble()`.

```
swiss %>% select(2:3) %>% head()
```

```
##              Agriculture Examination
## Courtelary          17.0          15
## Delemont            45.1           6
## Franches-Mnt        39.7           5
## Moutier             36.5          12
## Neuveville          43.5          17
## Porrentruy          35.3           9
```

```
swiss %>% select(2:3) %>%
  as_tibble(rownames="Name") %>% head()
```

```
## # A tibble: 6 x 3
##   Name         Agriculture Examination
##   <chr>              <dbl>       <int>
## 1 Courtelary          17            15
## 2 Delemont            45.1           6
## 3 Franches-Mnt        39.7           5
## 4 Moutier             36.5          12
## 5 Neuveville          43.5          17
## 6 Porrentruy          35.3           9
```

An Aside

for People with Statistics Training

Getting Fitted Regression Coefficients

Recall that `my_list[[3]]` is output from a regression model.

```
my_list[[3]][["coefficients"]]
```

```
## (Intercept)       speed
##  -17.579095    3.932409
```

```
(speed_beta <- my_list[[3]][["coefficients"]]["speed"])
```

```
##    speed
## 3.932409
```

Regression Summaries

`summary(lm_object)` is also a list with more information, which has the side effect of printing some output to the console:

```
summary(my_list[[3]]) # this prints output
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511,    Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Getting Standard Errors

```r
summary(my_list[[3]])[["coefficients"]] # a matrix
```

```
##                Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) -17.579095  6.7584402 -2.601058 1.231882e-02
## speed         3.932409  0.4155128  9.463990 1.489836e-12
```

```r
(speed_SE <- summary(my_list[[3]])[["coefficients"]]["speed", "Std. Error"])
```

```
## [1] 0.4155128
```

Example: 95% confidence interval

```r
speed_CI <- speed_beta + c(-qnorm(0.975), qnorm(0.975)) * speed_SE
names(speed_CI) <- c("lower", "upper")
```

Now you can include these values in a Markdown document:

```
A 1 mph increase in speed is associated with a `r
round(speed_beta, 1)` ft increase in stopping distance
(95% CI: (`r round(speed_CI["lower"],1)`,
          `r round(speed_CI["upper"],1)`)).
```

A 1 mph increase in speed is associated with a 3.9 ft increase in stopping distance (95% CI: (3.1, 4.7)).

`qnorm(0.975)` is 1.96

Practice and Homework

Suggested Practice: `swirl`

You can do interactive R tutorials in `swirl` that cover these structure basics.
To set up `swirl`:

1. `install.packages("swirl")`
2. `library(swirl)`
3. `swirl()`
4. Choose `R Programming`, pick a tutorial, and follow directions
5. To get out of `swirl`, type `bye()` in the middle of a lesson, or `0` in the menus

At this point, tutorials 1-8 are appropriate.

Homework: Two Choices

Data Structure Practice (Less Advanced):

Fill in a template R Markdown file that walks you through creating, accessing, and manipulating R data structures. Enter values in the R Markdown document and knit it to check your answers. *Knit after entering each answer*. If you get an error, check to see if undoing your last edit solves the problem; coding an assignment to handle all possible mistakes is really hard! This assignment is also long, so *start early*.

Manual Linear Regression (More Advanced)

Fill in a template R markdown file that walks you through (1) doing linear regression manually and (2) comparing it to the built-in `lm()` function. Includes simulating data and creating and modifying data structures. *Knit after entering each answer*. This assignment does not check answers as you go. This is also long, so *start early*!