CSSS 508, Lecture 4

Data Structures

Michael Pearce
(based on slides from Chuck Lanfear)

October 20, 2022

Preamble: R Data Structures

So far, we've been manipulating, summarizing, and making visuals out of data. That's pretty great!!

But now, we need to get more into the weeds of programming...

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

Topics

Last time, we learned about,

1. Subsetting data
2. Modifying data
3. Summarizing data
4. Joining (merging) data

Today, we will cover,

1. Types of Data
2. Vectors
3. Matrices
4. Lists
5. Data Frames and Tibbles

1. Types of Data

- `numeric`
- `character`
- `factor`
- `logical`
- `NA`, `NaN`, and `Inf`

How is my data stored?

Under the hood, R stores different types of data in different ways.

- e.g., R knows that `4.0` is a number, and that `"Michael"` is not a number.

So what exactly are the common data types, and how do we know what R is doing?

Data Types

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

Note on Factor Vectors

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

```
test_group <- factor(c("Treatment", "Placebo",
                       "Placebo", "Treatment"))
test_group
```

```
## [1] Treatment Placebo   Placebo   Treatment
## Levels: Placebo Treatment
```

Why use `factor` instead of `character`? Because `factor` data can go into a statistical model.[1]

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

Note on Logical Vectors

Remember that `logical` data in R takes on boolean `TRUE` or `FALSE` values.

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

```
my_booleans <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
sum(my_booleans)
```

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

```
mean(my_booleans)
```

```
## [1] 0.4
```

Missing or Infinite Data Types

Sometimes, your data may be missing or infinite (basically, not one of the types I just showed):

1. **Not Applicable** `NA`
    - Used when data simply is missing or "not available"
2. **Not a Number** `NaN`
    - Used when you try to perform a bad math operation, e.g., `0 / 0`
3. **Infinite** `Inf`, `-Inf`
    - Used when you divide by 0, e.g., `-5/0` or `5/0`

Checking Data Types

`class()` tells us what type of data we have:

```
class4    <- class(4)
classAB   <- class(c("A","B"))
classABFac<- class(factor("A","B"))
classTRUE <- class(TRUE)

c(class4,classAB,classABFac,classTRUE)
```

```
## [1] "numeric"   "character" "factor"    "logical"
```

Testing Data Types

There are also functions to **test** for certain data types:

```
c(is.numeric(5),  is.character("A"))
```

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

```
is.logical(TRUE)
```

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

```
c(is.infinite(-Inf),  is.na(NA),  is.nan(NaN))
```

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

Warning: `NA` is not `NaN`!!!

Mini-Check

In each case, what will R return?

- `is.numeric(3.14)`
  - `TRUE`
- `is.numeric(pi)`
  - `TRUE`
- `is.logical(FALSE)`
  - `TRUE`
- `is.nan(NA)`
  - `FALSE`

2. Vectors

- Creating Vectors
- Vector Math
- Subsetting 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, it will convert everything to *characters*!

Generating Numeric Vectors

There are shortcuts for generating numeric vectors:

```
1:10
```

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

```
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
```

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
```

Be careful!!

Example: Standardizing Data

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

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

```
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
```

Math with Missing Values

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

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

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

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

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

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("Andre","Brady","Cecilia",
                 "Danni","Edgar","Francie")
first_names[1]
```

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

```
first_names[c(1,2)]
```

```
## [1] "Andre" "Brady"
```

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

```
first_names[-3]
```

```
## [1] "Andre"   "Brady"   "Danni"   "Edgar"   "Francie"
```

Mini-Check

- What does `sum(c(1,2,NA))` output?
  - `NA`. The code `sum(c(1,2,NA),na.rm=TRUE)` would output `3`.
- What does `rep(c(0,1),times=2)` output?
  - `c(0,1,0,1)`
- I want to get the first and second elements of my vector, `a_vector`. What's wrong with the code `a_vector[1,2]` ?
  - `a_vector[c(1,2)]`

3. Matrices

Matrices: Two Dimensions

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

R fills in a matrix column-by-column (**not** row-by-row!)

```
(a_matrix <- matrix(first_names, nrow=2, ncol=3))
```

```
##      [,1]    [,2]      [,3]
## [1,] "Andre" "Cecilia" "Edgar"
## [2,] "Brady" "Danni"   "Francie"
```

Binding Vectors

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

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

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

```
(c_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] "Cecilia"
```

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

```
## [1] "Cecilia" "Edgar"
```

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

```
dim(a_matrix)
```

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

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.

```
a_matrix[, 1]
```

```
## [1] "Andre" "Brady"
```

You can prevent this behavior using `drop=FALSE`.

```
a_matrix[, 1, drop=FALSE]
```

```
##      [,1]
## [1,] "Andre"
## [2,] "Brady"
```

Matrix Data Type Warning

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

```
(bad_matrix <- cbind(1:2, c("Michael","Pearce")))
```

```
##      [,1] [,2]
## [1,] "1"  "Michael"
## [2,] "2"  "Pearce"
```

In this case, everything was converted to characters!

Matrix Dimension Names

We can access dimension names or name them ourselves:

```
rownames(bad_matrix) <- c("First", "Last")
colnames(bad_matrix) <- c("Number", "Name")
bad_matrix
```

```
##       Number Name
## First "1"    "Michael"
## Last  "2"    "Pearce"
```

```
bad_matrix[ ,"Name", drop=FALSE]
```

```
##       Name
## First "Michael"
## Last  "Pearce"
```

Matrix Arithmetic

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

```
matrix(c(2,4,6,8),nrow=2,ncol=2) / matrix(c(2,1,3,1),nrow=2,ncol=2)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    4    8
```

"Proper" Matrix Math

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

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

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

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

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

```
##      [,1] [,2]
## [1,]   14   32
## [2,]   32   77
```

"Proper" Matrix Math (cont.)

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

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

```
##            [,1]       [,2]
## [1,]  1.4259259 -0.5925926
## [2,] -0.5925926  0.2592593
```

Check Your Understanding

- Write code to create the following matrix:

```
##      [,1] [,2] [,3]
## [1,] "A"  "B"  "C"
## [2,] "D"  "E"  "F"
```

- Write a line of code to extract the second column. Ensure the output is still a *matrix*.

```
##      [,1]
## [1,] "B"
## [2,] "E"
```

My Answers

- Write code to create the following matrix:

```
mat_test <- matrix(c("A","B","C","D","E","F"),nrow=2,byrow=TRUE)
```

- Write a line of code to extract the second column. Ensure the output is still a *matrix*.

```
mat_test[,2,drop=FALSE]
```

4. Lists

What are Lists?

Lists are objects that can store multiple types of data.

```
my_list <- list("first_thing"  = 1:5,
                "second_thing" = matrix(8:11, nrow = 2))
my_list
```

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

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[[1]]
```

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

```
my_list$first_thing
```

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

Why Two Brackets `[[ ]]`?

Double brackets get *the actual element*—as whatever data type it is stored as—in that location in the list.

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

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

If you use single brackets to access list elements, you get a **list** back.

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

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

`names()` and List Elements

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

```
names(my_list)
```

```
## [1] "first_thing"  "second_thing"
```

Example: Regression Output

When you perform linear regression in R, the output is a list!

```
lm_output <- lm(speed~dist,data=cars)
is.list(lm_output)
```

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

```
names(lm_output)
```

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

```
lm_output$coefficients
```

```
## (Intercept)        dist
##   8.2839056   0.1655676
```

5. Matrices, Data Frames, and Tibbles

- `matrix`
- `data.frame`
- `tibble`

Matrices, Data Frames, and Tibbles

All of these structures display data in two dimensions

- `matrix`
  - Base R
  - Single data type allowed
- `data.frame`
  - Base R (default for data storage)
  - Stores multiple data types
- `tibbles`
  - tidyverse
  - Stores multiple data types
  - Displays nicely

In practice, data.frames and tibbles are very similar!

Check Your Understanding

1. Complete the following sentence: "Lists are to vectors, what data frames are to..."

2. Create a list that contains 3 elements: "ten_numbers" (integers between 1 and 10), "my_name" (your name as a character), and "booleans" (vector of `TRUE` and `FALSE` alternating three times).

My Solution

1. Complete the following sentence: "Lists are to vectors, what data frames are to...**Matrices!**" Lists and data frames can contain mixed data types, while vectors and matrices can only contain one data type.

2. Create a list that contains 3 elements:

```
my_new_list <- list("ten_numbers"=1:10,
                    "my_name"="Michael Pearce",
                    "booleans"=rep(c(TRUE,FALSE),times=3))
my_new_list
```

```
## $ten_numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $my_name
## [1] "Michael Pearce"
##
## $booleans
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

Homework 4

For Homework 4, you will fill in an RMarkdown template on my website that walks you through the process of creating, accessing, and manipulating R data structures. Enter values in the RMarkdown 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*.

On the due date, I will provide a key for the written answers. You will grade those answers as part of your peer review. In addition, you'll be asked to comment on the style of your peer's code and what you yourself did similarly/different. Please remember to provide a numerical grade (0-3), as always.