Chapter 2: Silly things

Andrew Flowers

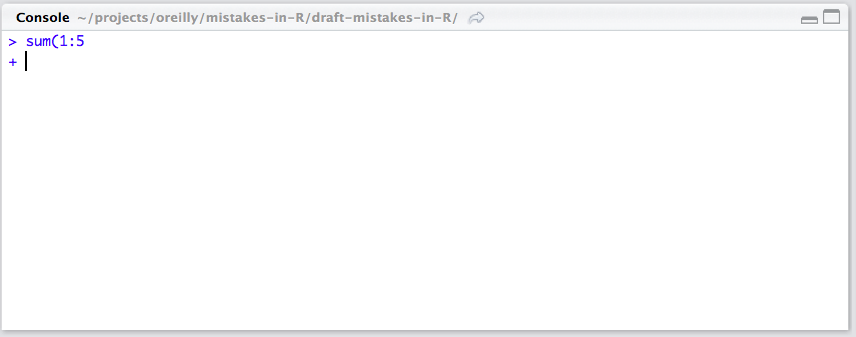
2/27/2017

Now you should be setup for success in R -- you're using RStudio, can install packages and are working through RStudio projects. But there will still be lots of frustrating quirks to learn about R. And often your mistakes, you'll come to find, are silly little things. In this chapter we'll cover those common screwups in actual R coding

### Basics

R is very picky. Every ( must be paired with a corresponding ). Variable names most be spelled correctly, and when passing a file name or other string to an R function, it must be surrounded by quotes (e.g. 'dir/file-name.csv').

Usually, it's best to work in the RStudio Console to interactively run R commands. And while sometimes you'll get an error, other types the console will leave you hanging with an "+" while your cursor is blinking. Like here when I forget the second ):



This means your command is incomplete and RStudio is waiting. Just press the "esc" key.

### Strings as factors

Every programming language has basic data structures for numbers, strings and booleans. R is no different: there are double, character and logical data types. There is also the list for heterogeneous data. But the best known -- and most influential -- R data structrue is data.frame. A dataframe contains tabular data, like a spreadsheet. More precisely, it's a list of equal-length vectors.

However, one annoying default with R is how it treats strings (or character) data in dataframes. Unless you specify otherwise, it interprets this data as a factor. A factor can be a useful data structure. A factor is a vector with only **pre-defined** values, and so it works well with categorical data (like male/female or democrat/republican). Yet often you do *not* want your non-numeric data to be factors. It's better to have them as plain-old characters.

This problem is often encountered when reading in data from an external data file. Let's import the file common\_names.csv, which can be found at the [GitHub repository for this book](https://github.com/andrewflowers/how-to-make-mistakes-in-R). This is data on the 100 most common names in America, and came from a [story](https://fivethirtyeight.com/features/whats-the-most-common-name-in-america/) I wrote with my former colleague Mona Chalabi for [FiveThirtyEight](fivethirtyeight.com). The data has two columns, one for the name and the other (perct\_pop) for it's share of the population.

Below we'll import the data using the "base R" function read.csv and then we'll attempt to calculate the number of characters in each name using the nchar() function. But it will fail. That's because the column name is not a charater data type. It's a factor, as we'll find out by using the helpful str() command, which tell us the structure of our dataframe.

common\_names <- read.csv("common\_names.csv")  
# Attempt to calculate characters in each name  
nchar(common\_names$name)

## Error in nchar(common\_names$name): 'nchar()' requires a character vector

str(common\_names)

## 'data.frame': 100 obs. of 2 variables:  
## $ name : Factor w/ 100 levels "Aaron","Adam",..: 70 39 44 83 24 99 67 21 48 82 ...  
## $ perct\_pop: num 0.01158 0.01022 0.00967 0.00949 0.00894 ...

This is frustrating and annoying. My advice is to have R, by default, interpret strings as charaters, not factors. There are several ways to accomplish this, a common one being to set the stringsAsFactors setting to FALSE. You can do this globally with the options() function, but I would **NOT** recommend that. Why? Becaue it can cause problems when sharing your code with others. You'd either be changing *their* global settings, which isn't polite; or your code could break when they run it.

Scratching that, you have two options. You can set stringsAsFactors == FALSE directly in your function call, if that function allows it (like read.csv does). Once we do this and run the str() function, notice how the data type of name is now a character, not a factor. See here:

common\_names <- read.csv("common\_names.csv", stringsAsFactors = FALSE)  
str(common\_names)

## 'data.frame': 100 obs. of 2 variables:  
## $ name : chr "Michael" "James" "John" "Robert" ...  
## $ perct\_pop: num 0.01158 0.01022 0.00967 0.00949 0.00894 ...

nchar(common\_names$name)

## [1] 7 5 4 6 5 7 4 11 6 7 6 6 7 8 7 7 8 5 4 9 6 6 6  
## [24] 5 5 7 7 5 5 7 4 7 4 4 5 5 7 6 6 4 5 8 8 6 8 8  
## [47] 5 6 6 6 7 9 5 7 4 6 7 5 6 5 4 7 7 5 7 5 3 8 7  
## [70] 6 5 6 5 9 8 6 8 7 8 5 6 5 6 6 5 5 7 6 4 9 7 5  
## [93] 9 6 6 9 5 6 4 4

But I'd argue there is an even better approach: use the readr package developed by Hadley Wickham. readr was installed with the tidyverse package we loaded earlier. It has lots of great functions for reading data into R. The analogous readr function to read.csv is read\_csv. In addition to being faster, it interprets string data as characters (and not factors) **by default**. So no more typing stringsAsFactors == FALSE. Just use readr.

library(readr)  
common\_names <- read\_csv("common\_names.csv")

## Parsed with column specification:  
## cols(  
## name = col\_character(),  
## perct\_pop = col\_double()  
## )

str(common\_names)

## Classes 'tbl\_df', 'tbl' and 'data.frame': 100 obs. of 2 variables:  
## $ name : chr "Michael" "James" "John" "Robert" ...  
## $ perct\_pop: num 0.01158 0.01022 0.00967 0.00949 0.00894 ...  
## - attr(\*, "spec")=List of 2  
## ..$ cols :List of 2  
## .. ..$ name : list()  
## .. .. ..- attr(\*, "class")= chr "collector\_character" "collector"  
## .. ..$ perct\_pop: list()  
## .. .. ..- attr(\*, "class")= chr "collector\_double" "collector"  
## ..$ default: list()  
## .. ..- attr(\*, "class")= chr "collector\_guess" "collector"  
## ..- attr(\*, "class")= chr "col\_spec"

### Subsetting

Another common source of R mistakes comes when subsetting your data. Say you want to select a column from a data frame. Well, you have a few options. You can use the $ or [ operators. But a common mistake is to use the $ operator with a *variable* for the column yuo want to select. That is why you should use the [ operator with variables. Let's try this out on the classic toy dataset mtcars, part of the datasets package installed with "base R."

mtcars$mpg # This works

## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2  
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4  
## [29] 15.8 19.7 15.0 21.4

column\_to\_select <- 'wt'  
mtcars$column\_to\_select # This does NOT work

## NULL

mtcars['wt'] # This is how you use variables to subset

## wt  
## Mazda RX4 2.620  
## Mazda RX4 Wag 2.875  
## Datsun 710 2.320  
## Hornet 4 Drive 3.215  
## Hornet Sportabout 3.440  
## Valiant 3.460  
## Duster 360 3.570  
## Merc 240D 3.190  
## Merc 230 3.150  
## Merc 280 3.440  
## Merc 280C 3.440  
## Merc 450SE 4.070  
## Merc 450SL 3.730  
## Merc 450SLC 3.780  
## Cadillac Fleetwood 5.250  
## Lincoln Continental 5.424  
## Chrysler Imperial 5.345  
## Fiat 128 2.200  
## Honda Civic 1.615  
## Toyota Corolla 1.835  
## Toyota Corona 2.465  
## Dodge Challenger 3.520  
## AMC Javelin 3.435  
## Camaro Z28 3.840  
## Pontiac Firebird 3.845  
## Fiat X1-9 1.935  
## Porsche 914-2 2.140  
## Lotus Europa 1.513  
## Ford Pantera L 3.170  
## Ferrari Dino 2.770  
## Maserati Bora 3.570  
## Volvo 142E 2.780

Notice how subsetting mtcars with $ vs. [ has a different output; the latter includes rownames. That's because $ returns the column as a vector, whereas the [ returns the column of the dataframe as what it is -- a list. If what you want is not a list, then you'll need to use the third type of subsetting operator, [[. Also, in addition to the str() function to examine the object, you can use typeof() too.

# Subsetting for a list  
mtcars['wt']

## wt  
## Mazda RX4 2.620  
## Mazda RX4 Wag 2.875  
## Datsun 710 2.320  
## Hornet 4 Drive 3.215  
## Hornet Sportabout 3.440  
## Valiant 3.460  
## Duster 360 3.570  
## Merc 240D 3.190  
## Merc 230 3.150  
## Merc 280 3.440  
## Merc 280C 3.440  
## Merc 450SE 4.070  
## Merc 450SL 3.730  
## Merc 450SLC 3.780  
## Cadillac Fleetwood 5.250  
## Lincoln Continental 5.424  
## Chrysler Imperial 5.345  
## Fiat 128 2.200  
## Honda Civic 1.615  
## Toyota Corolla 1.835  
## Toyota Corona 2.465  
## Dodge Challenger 3.520  
## AMC Javelin 3.435  
## Camaro Z28 3.840  
## Pontiac Firebird 3.845  
## Fiat X1-9 1.935  
## Porsche 914-2 2.140  
## Lotus Europa 1.513  
## Ford Pantera L 3.170  
## Ferrari Dino 2.770  
## Maserati Bora 3.570  
## Volvo 142E 2.780

typeof(mtcars['wt'])

## [1] "list"

# Subsetting for a vector  
mtcars[['wt']]

## [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440  
## [12] 4.070 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520  
## [23] 3.435 3.840 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780

typeof(mtcars[['wt']])

## [1] "double"

### Missing values

R handles missing values with an NA, but it's often a source of frustration. Take the example below. Note: we're using the c() function -- short for "combine" -- to create a vector of numbers.

sample\_data <- c(23, 7, 19, NA, -4, 12, 7)  
  
# Test for NAs  
is.na(sample\_data)

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

sum(sample\_data)

## [1] NA

mean(sample\_data)

## [1] NA

As you see, some R functions return NA by default if one element is an NA. To fix this, one must either filter out the NA values or set the na.rm parameter to TRUE, which removes them.

# Filter out NAs  
sum(sample\_data[!is.na(sample\_data)])

## [1] 64

# Tell function to remove NAs before calculating  
sum(sample\_data, na.rm = TRUE)

## [1] 64

One relevant warning about NA values: they don't appear by default in the useful table() function, which tallies the results of a single vector, factor or list. To have table() return NA values you must specify the useNA paramaeter to be 'ifany' (in quotes).

table(sample\_data) # Notice the NA is missing

## sample\_data  
## -4 7 12 19 23   
## 1 2 1 1 1

table(sample\_data, useNA = 'ifany')

## sample\_data  
## -4 7 12 19 23 <NA>   
## 1 2 1 1 1 1

Enough of these silly mistakes. Let's move on to how to write more elegant R code.