Activity Worksheet
Data Structures 2

Dan Maxwell

2018-10-12

A MyDataStory / THE CARPENTRIES Learning Experience

Introduction

At this point, you've seen it all - in the last chapter 2 activities worksheet, we toured all the basic data types and data structures in R. Everything you do will be a manipulation of those tools. But a whole lot of the time, the star of the show is going to be the data frame - the table that we created by loading information from a csv file. In this lesson, we'll learn a few more things about working with data frames.

Adding columns and rows in a data frame

We learned last time that the columns in a data frame were vectors, so that our data are consistent in type throughout the column. As such, if we want to add a new column, we need to start by making a new vector:

```
age <- c(2,3,5,12)
cats
```

```
##     coat weight likes_string
## 1 calico    2.1            1
## 2  black    5.0            0
## 3  tabby    3.2            1
```

We can then add this as a column via:

```
cats <- cbind(cats, age)
```

```
## Error in data.frame(..., check.names = FALSE): arguments imply differing number of rows: 3, 4
```

Why didn't this work? Of course, R wants to see one element in our new column for every row in the table:

```
cats
```

```
##     coat weight likes_string
## 1 calico    2.1            1
## 2  black    5.0            0
## 3  tabby    3.2            1
```

```
age <- c(4,5,8)
cats <- cbind(cats, age)
cats
```

```
##     coat weight likes_string age
## 1 calico    2.1            1   4
## 2  black    5.0            0   5
## 3  tabby    3.2            1   8
```

Now how about adding rows - in this case, we saw last time that the rows of a data frame are made of lists:

```
newRow <- list("tortoiseshell", 3.3, TRUE, 9)
cats <- rbind(cats, newRow)
```

```
## Warning in `[<-.factor`(`*tmp*`, ri, value = "tortoiseshell"): invalid
## factor level, NA generated
```

Factors

Another thing to look out for has emerged - when R creates a factor, it only allows whatever is originally there when our data was first loaded, which was 'black', 'calico' and 'tabby' in our case. Anything new that doesn't fit into one of these categories is rejected as nonsense (becomes NA).

The warning is telling us that we unsuccessfully added 'tortoiseshell' to our *coat* factor, but 3.3 (a numeric), TRUE (a logical), and 9 (a numeric) were successfully added to *weight*, *likes_string*, and *age*, respectfully, since those values are not factors. To successfully add a cat with a 'tortoiseshell' *coat*, explicitly add 'tortoiseshell' as a *level* in the factor:

```
levels(cats$coat)
```

```
## [1] "black"  "calico" "tabby"
```

```
levels(cats$coat) <- c(levels(cats$coat), 'tortoiseshell')
cats <- rbind(cats, list("tortoiseshell", 3.3, TRUE, 9))
```

Alternatively, we can change a factor column to a character vector; we lose the handy categories of the factor, but can subsequently add any word we want to the column without babysitting the factor levels:

```
str(cats)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ coat        : Factor w/ 4 levels "black","calico",..: 2 1 3 NA 4
##  $ weight      : num  2.1 5 3.2 3.3 3.3
##  $ likes_string: int  1 0 1 1 1
##  $ age         : num  4 5 8 9 9
```

```
cats$coat <- as.character(cats$coat)
str(cats)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ coat        : chr  "calico" "black" "tabby" NA ...
##  $ weight      : num  2.1 5 3.2 3.3 3.3
##  $ likes_string: int  1 0 1 1 1
##  $ age         : num  4 5 8 9 9
```

Removing rows

We now know how to add rows and columns to our data frame in R - but in our first attempt to add a 'tortoiseshell' cat to the data frame we've accidentally added a garbage row:

```
cats
```

```
##            coat weight likes_string age
## 1        calico    2.1            1   4
## 2         black    5.0            0   5
## 3         tabby    3.2            1   8
## 4          <NA>    3.3            1   9
## 5 tortoiseshell    3.3            1   9
```

We can ask for a data frame minus this offending row:

```
cats[-4,]
```

```
##            coat weight likes_string age
```

```
## 1        calico    2.1            1   4
## 2         black    5.0            0   5
## 3         tabby    3.2            1   8
## 5 tortoiseshell    3.3            1   9
```

Notice the comma with nothing after it to indicate we want to drop the entire fourth row.

Note: We could also remove both new rows at once by putting the row numbers inside of a vector: `cats[c(-4,-5),]`

Alternatively, we can drop all rows with `NA` values:

```
na.omit(cats)
```

```
##            coat weight likes_string age
## 1        calico    2.1            1   4
## 2         black    5.0            0   5
## 3         tabby    3.2            1   8
## 5 tortoiseshell    3.3            1   9
```

Let's reassign the output to `cats`, so that our changes will be permanent:

```
cats <- na.omit(cats)
```

Appending to a data frame

The key to remember when adding data to a data frame is that *columns are vectors or factors, and rows are lists.* We can also glue two data frames together with `rbind`:

```
cats <- rbind(cats, cats)
cats
```

```
##             coat weight likes_string age
## 1         calico    2.1            1   4
## 2          black    5.0            0   5
## 3          tabby    3.2            1   8
## 5  tortoiseshell    3.3            1   9
## 11        calico    2.1            1   4
## 21         black    5.0            0   5
## 31         tabby    3.2            1   8
## 51 tortoiseshell    3.3            1   9
```

But now the row names are unnecessarily complicated. We can remove the rownames, and R will automatically re-name them sequentially:

```
rownames(cats) <- NULL
cats
```

```
##            coat weight likes_string age
## 1        calico    2.1            1   4
## 2         black    5.0            0   5
## 3         tabby    3.2            1   8
## 4 tortoiseshell    3.3            1   9
## 5        calico    2.1            1   4
## 6         black    5.0            0   5
## 7         tabby    3.2            1   8
## 8 tortoiseshell    3.3            1   9
```

Challenge 1

You can create a new data frame right from within R with the following syntax:

```
df <- data.frame(id = c('a', 'b', 'c'),
                 x = 1:3,
                 y = c(TRUE, TRUE, FALSE),
                 stringsAsFactors = FALSE)
```

Make a data frame that holds the following information for yourself:

- first name
- last name
- lucky number

Then use `rbind` to add an entry for your best friend. Finally, use `cbind` to add a column with an answer to the question, "Is it time for coffee break?"

```
# Solution to Challenge 1

df <- data.frame(first = c('Grace'),
                 last = c('Hopper'),
                 lucky_number = c(0),
                 stringsAsFactors = FALSE)
df <- rbind(df, list('Marie', 'Curie', 238) )
df <- cbind(df, coffeetime = c(TRUE,TRUE))
```

Realistic example

So far, you've seen the basics of manipulating data frames with our cat data; now, let's use those skills to digest a more realistic dataset. Lets read in the gapminder dataset that we downloaded previously:

```
gapminder <- read.csv("gapminder.csv")
```

Miscellaneous Tips

- Another type of file you might encounter are tab-separated value files (.tsv). To specify a tab as a separator, use `"\\t"` or `read.delim()`.
- Files can also be downloaded directly from the Internet into a local folder of your choice onto your computer using the `download.file` function. The `read.csv` function can then be executed to read the downloaded file from the download location, for example,

```
download.file("https://raw.githubusercontent.com/danielsmaxwell/intro_to_r/master/data/gapminder.cs
gapminder <- read.csv("gapminder.csv")
```

- Alternatively, you can also read in files directly into R from the Internet by replacing the file paths with a web address in `read.csv`. One should note that in doing this no local copy of the csv file is first saved onto your computer. For example,

```
gapminder <- read.csv("https://raw.githubusercontent.com/danielsmaxwell/intro_to_r/master/data/gapm
```

- You can read directly from excel spreadsheets without converting them to plain text first by using the readxl package.

Let's investigate gapminder a bit; the first thing we should always do is check out what the data looks like with `str`:

```
str(gapminder)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

We can also examine individual columns of the data frame with our `typeof` function:

```
typeof(gapminder$year)
```

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

```
typeof(gapminder$country)
```

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

```
str(gapminder$country)
```

```
##  Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
```

We can also interrogate the data frame for information about its dimensions; remembering that `str(gapminder)` said there were 1704 observations of 6 variables in gapminder, what do you think the following will produce, and why?

```
length(gapminder)
```

```
## [1] 6
```

A fair guess would have been to say that the length of a data frame would be the number of rows it has (1704), but this is not the case; remember, a data frame is a *list of vectors and factors*:

```
typeof(gapminder)
```

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

When `length` gave us 6, it's because gapminder is built out of a list of 6 columns. To get the number of rows and columns in our dataset, try:

```
nrow(gapminder)
```

```
## [1] 1704
```

```
ncol(gapminder)
```

```
## [1] 6
```

Or, both at once:

```
dim(gapminder)
```

```
## [1] 1704    6
```

We'll also likely want to know what the titles of all the columns are, so we can ask for them later:

```
colnames(gapminder)
```

```
## [1] "country"   "year"      "pop"       "continent" "lifeExp"   "gdpPercap"
```

At this stage, it's important to ask ourselves if the structure R is reporting matches our intuition or expectations; do the basic data types reported for each column make sense? If not, we need to sort any problems out now before they turn into bad surprises down the road, using what we've learned about how R interprets data, and the importance of *strict consistency* in how we record our data.

Once we're happy that the data types and structures seem reasonable, it's time to start digging into our data proper. Check out the first few lines:

```
head(gapminder)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

To make sure our analysis is reproducible, we should put the code into a script file so we can come back to it later.

Challenge 2

Go to file -> new file -> R script, and write an R script to load in the gapminder dataset. Put it in your default working directory.

Run the script using the `source` function, using the file path as its argument (or by pressing the "source" button in RStudio).

```
# Solution to Challenge 2
# The contents of `script/load-gapminder.R`:

download.file("https://raw.githubusercontent.com/danielsmaxwell/intro_to_r/master/data/gapminder.csv", d

gapminder <- read.csv(file = "gapminder.csv")

# To run the script and load the data into the `gapminder` variable:

source(file = "load-gapminder.R")
```

Challenge 3

Read the output of `str(gapminder)` again; this time, use what you've learned about factors, lists and vectors, as well as the output of functions like `colnames` and `dim` to explain what everything that `str` prints out for gapminder means.

```
# Solution to Challenge 3

# The object `gapminder` is a data frame with columns
# - `country` and `continent` are factors.
# - `year` is an integer vector.
# - `pop`, `lifeExp`, and `gdpPercap` are numeric vectors.
```