Data frames

UA R Users Group

May 14, 2019

Contents

Learning Objectives 1

Setup **1**

Looking at data **1**

What are data frames? **2**

Inspecting `data.frame` objects **3**

Indexing `data.frame` objects 4

Factors 6

Learning Objectives

- load external data (CSV files) in memory
- understand the concept of a `data.frame`
- know how to access any element of a `data.frame`
- understand `factors` and how to manipulate them

Setup

The file required for this lesson can be downlaoded by clicking on this link (It will take you to the excel spreadsheet if you have Box synced on your computer. Otherwise you can simply download the entire folder on your computer.)

- Move the downloaded file into the directory for this workshop

To view your current working directory use the `getwd()` command

To set you working directory, use the `setwd()` command. We want to set the working directory to the location of our project. For example:

```r
setwd("~/Workshops/UA-R-Users-Group")
```

Looking at data

You are now ready to load the data. We are going to use the R function `read.csv()` to load the data file into memory (as a `data.frame`). In this case, our data is in a subdirectory called “data”.

```r
cats <- read.csv(file = 'data/herding-cats-small.csv')
```

This statement doesn't produce any output because assignment doesn't display anything. If we want to check that our data has been loaded, we can print the variable's value: `cats`.

```r
cats
```

```
##                  street    coat    sex   age weight fixed wander_dist
## 1        Los Robles Way   tabby female 3.003  3.993     0       0.040
## 2       242 Harding Ave maltese female 8.234 12.368     1       0.033
## 3     201 Hollywood Ave   brown female 4.601  3.947     1       0.076
## 4         140 Robin Way   black female 7.172  8.053     1       0.030
## 5        135 Charles St  calico   male 4.660  6.193     1       0.085
## 6   130 Vista Del Campo   tabby female 3.796  3.860     1       0.085
## 7   115 Via Santa Maria   brown   male 6.917  5.626     1       0.097
## 8       303 Harding Ave   brown   male 3.713  3.982     1       0.033
## 9      16982 Kennedy Rd   black female 2.851  3.291     0       0.065
## 10  16528 Marchmont Dr maltese female 4.594  6.994     0       0.059
##    roamer cat_id
## 1      no    321
## 2      no    250
## 3      no    219
## 4      no    182
## 5     yes    107
## 6      no    234
## 7     yes    196
## 8      no    311
## 9      no    130
## 10     no    349
```

However, if our dataset was larger, we probably wouldn't want to print the whole thing to our console. Instead, we can use the `head` command to view the first six lines or the `View` command to open the dataset in a spreadsheet-like viewer.

```r
head(cats)
View(cats)
```

We've just done two very useful things.
1. We've read our data in to R, so now we can work with it in R
2. We've created a data frame (with the `read.csv` command) the standard way R works with data.

What are data frames?

`data.frame` is the *de facto* data structure for most tabular data and what we use for statistics and plotting.

A `data.frame` is actually a `list` of vectors of identical lengths. Each vector represents a column, and each vector can be of a different data type (e.g., characters, integers, factors). The `str()` function is useful to inspect the data types of the columns.

A `data.frame` can be created by the functions `read.csv()` or `read.table()`, in other words, when importing spreadsheets from your hard drive (or the web).

By default, `data.frame` converts (= coerces) columns that contain characters (i.e., text) into the `factor` data type. Depending on what you want to do with the data, you may want to keep these columns as `character`. To do so, `read.csv()` and `read.table()` have an argument called `stringsAsFactors` which can be set to `FALSE`:

Let's now check the structure of this `data.frame` in more details with the function `str()`:

```
str(cats)
```

```
## 'data.frame':    10 obs. of  9 variables:
##  $ street     : Factor w/ 10 levels "115 Via Santa Maria",..: 10 8 7 4 3 2 1 9 6 5
##  $ coat       : Factor w/ 5 levels "black","brown",..: 5 4 2 1 3 5 2 2 1 4
##  $ sex        : Factor w/ 2 levels "female","male": 1 1 1 1 2 1 2 2 1 1
##  $ age        : num  3 8.23 4.6 7.17 4.66 ...
##  $ weight     : num  3.99 12.37 3.95 8.05 6.19 ...
##  $ fixed      : int  0 1 1 1 1 1 1 1 0 0
##  $ wander_dist: num  0.04 0.033 0.076 0.03 0.085 0.085 0.097 0.033 0.065 0.059
##  $ roamer     : Factor w/ 2 levels "no","yes": 1 1 1 1 2 1 2 1 1 1
##  $ cat_id     : int  321 250 219 182 107 234 196 311 130 349
```

Inspecting `data.frame` objects

We already saw how the functions `head()` and `str()` can be useful to check the content and the structure of a `data.frame`. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data.

- Size:
  - `dim()` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow()` - returns the number of rows
  - `ncol()` - returns the number of columns
- Content:
  - `head()` - shows the first 6 rows
  - `tail()` - shows the last 6 rows
- Names:
  - `names()` - returns the column names (synonym of `colnames()` for `data.frame` objects)

  - `rownames()` - returns the row names

- Summary:
  - `str()` - structure of the object and information about the class, length and content of each column

  - `summary()` - summary statistics for each column

Note: most of these functions are "generic", they can be used on other types of objects besides `data.frame`.

Indexing `data.frame` objects

Our cats data frame has rows and columns (it has 2 dimensions), if we want to extract some specific data from it, we need to specify the "coordinates" we want from it. Row numbers come first, followed by column numbers (i.e. [row, column]).

```
cats[1, 2]   # first element in the 2nd column of the data frame
cats[1, 6]   # first element in the 6th column
cats[1:3, 7] # first three elements in the 7th column
cats[3, ]    # the 3rd element for all columns
cats[, 7]    # the entire 7th column
head_meta <- cats[1:6, ] # Row 1-6 which is the same as head
```

For larger datasets, it can be tricky to remember the column number that corresponds to a particular variable. (Are species names in column 5 or 7? oh, right… they are in column 6). In some cases, in which column the variable will be can change if the script you are using adds or removes columns. It's therefore often better to use column names to refer to a particular variable, and it makes your code easier to read and your intentions clearer.

You can do operations on a particular column, by selecting it using the `$` sign. In this case, the entire column is a vector. You can use `names(cats)` or `colnames(cats)` to remind yourself of the column names. For instance, to extract all the cats' weight information from our dataset:

```
cats$weight
```

```
##  [1]  3.993 12.368  3.947  8.053  6.193  3.860  5.626  3.982  3.291  6.994
```

In some cases, you may way to select more than one column. You can do this using the square brackets, passing in a vector of the columns to select. Suppose we wanted weight and coat information:

```
cats[ , c("weight", "coat")]
```

```
##    weight    coat
## 1   3.993   tabby
## 2  12.368 maltese
## 3   3.947   brown
## 4   8.053   black
## 5   6.193  calico
## 6   3.860   tabby
## 7   5.626   brown
## 8   3.982   brown
## 9   3.291   black
## 10  6.994 maltese
```

You can even access columns by column name *and* select specific rows of interest. For example, if we wanted the weight and coat of just rows 4 through 7, we could do:

```
cats[4:7, c("weight", "coat")]
```

```
##   weight   coat
## 4  8.053  black
## 5  6.193 calico
## 6  3.860  tabby
## 7  5.626  brown
```

We can can also use logical statements to select and filter items from a `data.frame`. For example, to select all rows with black cats we could use the following statement

```r
cats[cats$coat == "black", ]
```

```
##               street  coat    sex   age weight fixed wander_dist roamer
## 4    140 Robin Way black female 7.172  8.053     1       0.030     no
## 9 16982 Kennedy Rd black female 2.851  3.291     0       0.065     no
##   cat_id
## 4    182
## 9    130
```

let's break this down a bit. The logical statement in the brackets returns a vector of `TRUE` and `FALSE` values.

```r
cats$coat == "black"
```

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

These `booleans` allow us to select which records we want from our `data.frame`

Another way to do this is with the function `which()`. `which()` finds the indexes of records meeting a logical statement

```r
which(cats$coat == "black")
```

```
## [1] 4 9
```

So, we could also write

```r
cats[which(cats$coat == "black"), ]
```

```
##               street  coat    sex   age weight fixed wander_dist roamer
## 4    140 Robin Way black female 7.172  8.053     1       0.030     no
## 9 16982 Kennedy Rd black female 2.851  3.291     0       0.065     no
##   cat_id
## 4    182
## 9    130
```

But that's getting really long and ugly. R is already considered somewhat of an ugly duckling among programming languages, so no reason to play into the stereotype.

We can combine logical statements and index statements

```r
cats[cats$coat == "black", c("coat", "weight")]
```

```
##    coat weight
## 4 black  8.053
## 9 black  3.291
```

Finally, we can use `&`, the symbol for "and", and `|`, the symbol for "or", to make logical statements.

```r
cats[cats$coat == "black" & cats$roamer == "no", ]
```

```
##               street  coat    sex   age weight fixed wander_dist roamer
## 4    140 Robin Way black female 7.172  8.053     1       0.030     no
## 9 16982 Kennedy Rd black female 2.851  3.291     0       0.065     no
##   cat_id
## 4    182
## 9    130
```

This statement selects all records with black cats that also like string

Factors

Factors are used to represent categorical data. Factors can be ordered or unordered and are an important class for statistical analysis and for plotting.

Factors are stored as integers, and have labels associated with these unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

In the data frame we just imported, let's do

```r
str(cats)
```

```
## 'data.frame':    10 obs. of  9 variables:
##  $ street     : Factor w/ 10 levels "115 Via Santa Maria",..: 10 8 7 4 3 2 1 9 6 5
##  $ coat       : Factor w/ 5 levels "black","brown",..: 5 4 2 1 3 5 2 2 1 4
##  $ sex        : Factor w/ 2 levels "female","male": 1 1 1 1 2 1 2 2 1 1
##  $ age        : num  3 8.23 4.6 7.17 4.66 ...
##  $ weight     : num  3.99 12.37 3.95 8.05 6.19 ...
##  $ fixed      : int  0 1 1 1 1 1 1 1 0 0
##  $ wander_dist: num  0.04 0.033 0.076 0.03 0.085 0.085 0.097 0.033 0.065 0.059
##  $ roamer     : Factor w/ 2 levels "no","yes": 1 1 1 1 2 1 2 1 1 1
##  $ cat_id     : int  321 250 219 182 107 234 196 311 130 349
```

We can see the names of the multiple columns. And, we see that coat is a `Factor w/ 5 levels`

When we read in a file, any column that contains text is automatically assumed to be a factor. Once created, factors can only contain a pre-defined set values, known as *levels*. By default, R always sorts *levels* in alphabetical order.

You can check this by using the function `levels()`, and check the number of levels using `nlevels()`:

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

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

```r
nlevels(cats$coat)
```

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

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., "low", "medium", "high") or it is required by particular type of analysis. Additionally, specifying the order of the levels allows to compare levels:

```
satisfaction <- factor(c("low", "high", "medium", "high", "low", "medium", "high"))
levels(satisfaction)
```

```
## [1] "high"   "low"    "medium"
```

```
satisfaction <- factor(satisfaction, levels = c("low", "medium", "high"))
levels(satisfaction)
```

```
## [1] "low"    "medium" "high"
```

```
min(satisfaction) ## doesn't work
```

```
## Error in Summary.factor(structure(c(1L, 3L, 2L, 3L, 1L, 2L, 3L), .Label = c("low", : 'min' 
```

```
satisfaction <- factor(satisfaction, levels = c("low", "medium", "high"), ordered = TRUE)
levels(satisfaction)
```

```
## [1] "low"    "medium" "high"
```

```
min(satisfaction) ## works!
```

```
## [1] low
## Levels: low < medium < high
```

In R's memory, these factors are represented by numbers (1, 2, 3). They are better than using simple integer labels because factors are self describing: `"low"`, `"medium"`, and `"high"`" is more descriptive than `1`, `2`, `3`. Which is low? You wouldn't be able to tell with just integer data. Factors have this information built in. It is particularly helpful when there are many levels (like the species in our example data set).

Converting factors

If you need to convert a factor to a character vector, simply use `as.character(x)`.

Converting a factor to a numeric vector is however a little trickier, and you have to go via a character vector. Compare:

```
f <- factor(c(1, 5, 10, 2))
as.numeric(f)               ## wrong! and there is no warning...
```

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

```
as.numeric(as.character(f)) ## works...
```

```
## [1]  1  5 10  2
```

```
as.numeric(levels(f))[f]    ## The recommended way.
```

```
## [1]  1  5 10  2
```