Lecture 3 - Data Manipulation

Sajid Bashir, PhD

08 February, 2020

Contents

Agenda **2**

More on Data Frames **2**
Package Description 2
Adding Columns 3
Contigency Tables 3

Factors in R **4**
Creating Factors 4
Changing Levels 5

Lists in R **7**
Creating Lists 7
Names and Referencing 7

Functions in R **8**
Creating functions 8
Returning a list 9

If-else statements **9**

Reminders **10**

Agenda

- More on Data Frames
- Factors
- Lists
- Writing functions in R
- If-else statements

More on Data Frames

Package Description

Typically, a package includes code, documentation and various functions, some tests to check everything works as it should, and some data sets.

Basic information about a package is provided in the `DESCRIPTION file` and it answers following questions at minimum:

- What the package does?
- Who is the author?
- What version the documentation belongs to?
- What is the date of it's release?
- What type of license its uses?
- What are the package dependencies?

Following are equivalent commands that when executed through console will open the description of package named `MASS` (package details):

```
help(package = "MASS")
packageDescription("MASS")
```

We will use the `Cars93` data Set that contains data from 93 Cars on sale in the USA in 1993. Load the package and view the details of first 3 cars:

```
library(MASS)
head(Cars93, 3)
```

```
##   Manufacturer   Model    Type Min.Price Price Max.Price MPG.city MPG.highway
## 1        Acura Integra   Small      12.9  15.9      18.8       25          31
## 2        Acura  Legend Midsize      29.2  33.9      38.7       18          25
## 3         Audi      90 Compact      25.9  29.1      32.3       20          26
##              AirBags DriveTrain Cylinders EngineSize Horsepower  RPM
## 1               None      Front         4        1.8        140 6300
## 2 Driver & Passenger      Front         6        3.2        200 5500
## 3        Driver only      Front         6        2.8        172 5500
##   Rev.per.mile Man.trans.avail Fuel.tank.capacity Passengers Length Wheelbase
## 1         2890             Yes               13.2          5    177       102
## 2         2335             Yes               18.0          5    195       115
## 3         2280             Yes               16.9          5    180       102
##   Width Turn.circle Rear.seat.room Luggage.room Weight  Origin          Make
## 1    68          37           26.5           11   2705 non-USA Acura Integra
## 2    71          38           30.0           15   3560 non-USA  Acura Legend
## 3    67          37           28.0           14   3375 non-USA       Audi 90
```

Adding Columns

1. `transform()` returns a new data frame with columns modified or added as specified by the function call. We can add ot our data frame two new columns, giving the fuel consumption within the city and on highway in km/l.

```
Cars93.metric <- transform(Cars93,
                           KMPL.city = 0.425 * MPG.city,
                           KMPL.highway = 0.425 * MPG.highway)
tail(names(Cars93.metric))
```

```
## [1] "Luggage.room" "Weight"       "Origin"       "Make"         "KMPL.city"
## [6] "KMPL.highway"
```

   Note: `tail` returns the last parts of a vector, matrix, table, data frame or function.

2. Another approach

```
# Add a new column called KMPL.city.2
Cars93.metric$KMPL.city.2 <- 0.425 * Cars93$MPG.city
tail(names(Cars93.metric))
```

```
## [1] "Weight"       "Origin"       "Make"         "KMPL.city"    "KMPL.highway"
## [6] "KMPL.city.2"
```

Comparing the results of both the approaches:

```
identical(Cars93.metric$KMPL.city, Cars93.metric$KMPL.city.2)
```

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

Contigency Tables

We can use `Contigency tables` to summarize the information given in a data frame. A contingency table (also known as a `cross tabulation` or `crosstab`) is a type of table in a matrix format that displays the (multivariate) frequency distribution of the variables.

1. The `table()` function builds **contingency tables** showing counts at each combination of factor levels e.g. distribution based on the airbags in cars available for sale.

```
table(Cars93$AirBags)
```

```
##
## Driver & Passenger        Driver only               None
##                 16                 43                 34
```

   An interesting comparison can be between airbags and the origin of cars:

```
table(Cars93$Origin)
```

```
##
##     USA non-USA
##      48      45
```

```
table(Cars93$AirBags, Cars93$Origin)
```

```
##
##                      USA non-USA
##   Driver & Passenger   9       7
##   Driver only         23      20
```

```
##   None                16      18
```

```
 > Looks like US and non-US cars had about the same distribution of AirBag types
 > Later in the class we'll learn how to do a hypothesis tests on this kind of data
```

2. Alternative syntax: When `table()` is supplied a data frame, it produces contingency tables for all combinations of *factors*:

```
head(Cars93[c("AirBags", "Origin")], 3)
```

```
##              AirBags  Origin
## 1               None non-USA
## 2 Driver & Passenger non-USA
## 3        Driver only non-USA
```

```
table(Cars93[c("AirBags", "Origin")])
```

```
##                     Origin
## AirBags              USA non-USA
##   Driver & Passenger   9       7
##   Driver only         23      20
##   None                16      18
```

Factors in R

Creating Factors

`Factors` are special vectors to store categorical variables. Factors can be used for different porposes:

1. Although not recommended but these character vectors can be used to store `nominal data` (data used to label variables without providing any quantitative value). A factor of blood groups with Levels i.e. set of all possible categories.

```
blood <- factor(c("O", "AB", "A"), levels = c("A", "B", "AB", "O"))
blood
```

```
## [1] O  AB A
## Levels: A B AB O
```

2. Store `labels` only once and reduce the memory size e.g. store 1, 1, 2 instead of MALE, MALE, FEMALE.

```
#Creating Factors
gender <- factor(c("male","female","male"))
gender
```

```
## [1] male   female male
## Levels: female male
```

```
#User defined levels
levels(gender) <- c("1","2") # assign levels in alphabetical order
gender
```

```
## [1] 2 1 2
## Levels: 1 2
```

3. Factors are very important when dealing with `ordinal data`. In such cases, we set the parameter `ordered` to `TRUE` e.g. describing severity of flue in increasing order.

```
symptoms <- factor(c("SEVERE", "MILD", "MODERATE"),
                   levels = c("MILD", "MODERATE", "SEVERE"),
                   ordered = TRUE)
symptoms
```

```
## [1] SEVERE   MILD     MODERATE
## Levels: MILD < MODERATE < SEVERE
```

Logical Test - Whether symptoms are greater than MODERATE?

```
symptoms > "MODERATE"
```

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

Changing Levels

The list of manufacturers in

```
manufacturer <- Cars93$Manufacturer
head(manufacturer, 10)
```

```
##  [1] Acura    Acura    Audi     Audi     BMW      Buick    Buick    Buick
##  [9] Buick    Cadillac
## 32 Levels: Acura Audi BMW Buick Cadillac Chevrolet Chrylser Chrysler ... Volvo
```

1. We'll use the `mapvalues(x, from, to)` function from the `plyr` library which is part of `tidyverse` package. How to find the list of loaded packages:

   ```
   (.packages())
   ```

   ```
   ## [1] "MASS"      "stats"     "graphics"  "grDevices" "utils"     "datasets"
   ## [7] "methods"   "base"
   ```

   Load the `plyr` library

   ```
   library(plyr)
   (.packages())
   ```

   ```
   ## [1] "plyr"      "MASS"      "stats"     "graphics"  "grDevices" "utils"
   ## [7] "datasets"  "methods"   "base"
   ```

   Changing the levels:

   ```
   # Map Chevrolet, Pontiac and Buick to GM
   manufacturer.combined <- mapvalues(manufacturer,
                                      from = c("Chevrolet", "Pontiac", "Buick"),
                                      to = rep("GM", 3))

   head(manufacturer.combined, 10)
   ```

   ```
   ##  [1] Acura    Acura    Audi     Audi     BMW      GM       GM       GM
   ##  [9] GM       Cadillac
   ## 30 Levels: Acura Audi BMW GM Cadillac Chrylser Chrysler Dodge Eagle ... Volvo
   ```

2. **Another Example:** A lot of data comes with integer encodings of `levels`. You may want to convert the integers to more meaningful values for the purpose of your analysis. Let's assume that in the students survey `Program` was coded as an integer with 1 = CS, 2 = DS and 3 = Other,

```
survey <- read.table("survey_students.csv", header = TRUE, sep = ",")
survey <- transform(survey, Program = as.numeric(Program))
head(survey,10)
```

```
##    Program                PriorExp      Rexperience OperatingSystem TVhours
## 1        1         Some experience       Never used        Mac OS X       2
## 2        1 Never programmed before       Never used         Windows      15
## 3        1         Some experience Basic competence        Mac OS X      16
## 4        3         Some experience Basic competence        Mac OS X       0
## 5        1 Never programmed before       Never used         Windows       2
## 6        1 Never programmed before       Never used         Windows       5
## 7        1         Some experience Basic competence        Mac OS X       0
## 8        2         Some experience Basic competence         Windows       4
## 9        1         Some experience Basic competence         Windows       0
## 10       3 Never programmed before       Never used         Windows       0
##            Editor
## 1  Microsoft Word
## 2  Microsoft Word
## 3  Microsoft Word
## 4           LaTeX
## 5  Microsoft Word
## 6  Microsoft Word
## 7  Microsoft Word
## 8  Microsoft Word
## 9  Microsoft Word
## 10 Microsoft Word
```

We can get back the program codings using the `transform()`, `as.factor()` and `mapvalues()` functions

```
survey <- transform(survey,
                Program = as.factor(mapvalues(Program,
                                              c(1, 2, 3),
                                              c("CS", "DS", "Other")))
                )
head(survey)
```

```
##   Program                PriorExp      Rexperience OperatingSystem TVhours
## 1      CS         Some experience       Never used        Mac OS X       2
## 2      CS Never programmed before       Never used         Windows      15
## 3      CS         Some experience Basic competence        Mac OS X      16
## 4   Other         Some experience Basic competence        Mac OS X       0
## 5      CS Never programmed before       Never used         Windows       2
## 6      CS Never programmed before       Never used         Windows       5
##           Editor
## 1 Microsoft Word
## 2 Microsoft Word
## 3 Microsoft Word
## 4          LaTeX
## 5 Microsoft Word
## 6 Microsoft Word
```

Lists in R

Creating Lists

Recall: A vector is a data structure for storing *similar kinds of data*. Thus vectors expect elements to be all of the same type (e.g., `Boolean`, `numeric`, `character`). When data of different types are put into a vector, `R` converts everything to a common type. Consider the following examples.

```
# Example - 1
my.vector.1 <- c("Abbas", 165, TRUE) # (name, weight, is.male)
my.vector.1
```

```
## [1] "Abbas" "165"   "TRUE"
```

```
typeof(my.vector.1)  # All the elements are now character strings!
```

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

```
# Example - 2
my.vector.2 <- c(FALSE, TRUE, 27) # (is.male, is.citizen, age)
typeof(my.vector.2)
```

```
## [1] "double"
```

A list is a **data structure** that can be used to store **different kinds** of data.

In `R` the simple way to build lists is by `list()` function

```
my.list <- list("Abbas", 165, TRUE)
my.list
```

```
## [[1]]
## [1] "Abbas"
##
## [[2]]
## [1] 165
##
## [[3]]
## [1] TRUE
```

```
sapply(my.list, typeof) # apply `typeof` to each element of `my.list`
```

```
## [1] "character" "double"    "logical"
```

Names and Referencing

- **Named Elements:** We can assign names to the elements of a list e.g.

  ```
  patient.1 <- list(name="Abbas", weight=165, is.male=TRUE)
  patient.1$name
  ```

  ```
  ## [1] "Abbas"
  ```

- **Referencing Elements:** The list elements can be referenced using the assigned names:

  ```
  patient.1$name # Get "name" element (returns a string)
  ```

  ```
  ## [1] "Abbas"
  ```

```r
patient.1[["name"]] # Get "name" element (returns a string)
```

```
## [1] "Abbas"
```

```r
patient.1["name"] # Get "name" slice (returns a sub-list)
```

```
## $name
## [1] "Abbas"
```

```r
c(typeof(patient.1$name), typeof(patient.1["name"]))
```

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

Functions in R

> A function is a machine that turns **input objects** (arguments) into an **output object** (return value) according to a definite rule.

We have already used a number of built-in R functions: `mean()`, `subset()`, `plot()`, `read.table()`...

An important part of programming and data analysis is to write custom functions. Functions help to make the code **modular**, facilitate debugging and moreover as data analyst we want to learn as how to apply *functions* for *data analysis*:

Creating functions

Let's look at a really simple function

```r
addOne <- function(x) {
  x + 1
}
```

`x` is the **argument** or **input**, the function **output** is the input `x` incremented by 1.

```r
addOne(12)
```

```
## [1] 13
```

A more interesting example is a function that returns a % given a numerator, denominator, and desired number of decimal values.

```r
calculatePercentage <- function(x, y, d) {
  decimal <- x / y  # Calculate decimal value
  round(100 * decimal, d)  # Convert to % and round to d digits
}

calculatePercentage(27, 80, 1)
```

```
## [1] 33.8
```

If you're calculating several %'s for your report, you should use this kind of function instead of repeatedly copying and pasting code

Returning a list

Here's a function that takes a person's full name (FirstName LastName), weight in lb and height in inches and converts it into a list with the person's first name, person's last name, weight in kg, height in m, and BMI.

```
createPatientRecord <- function(full.name, weight, height) {
  name.list <- strsplit(full.name, split=" ")[[1]]
  first.name <- name.list[1]
  last.name <- name.list[2]
  weight.in.kg <- weight / 2.2
  height.in.m <- height * 0.0254
  bmi <- weight.in.kg / (height.in.m ^ 2)
  list(first.name=first.name, last.name=last.name, weight=weight.in.kg, height=height.in.m,
       bmi=bmi)
}
```

```
createPatientRecord("Rashid Minhas", 185, 12 * 6 + 1)
```

```
## $first.name
## [1] "Rashid"
##
## $last.name
## [1] "Minhas"
##
## $weight
## [1] 84.09091
##
## $height
## [1] 1.8542
##
## $bmi
## [1] 24.45884
```

We can define a function to generate 3 number summary by calculating mean, median and standard deviation

```
threeNumberSummary <- function(x) {
  c(mean=mean(x), median=median(x), sd=sd(x))
}
x <- rnorm(100, mean=5, sd=2) # Vector of 100 normals with mean 5 and sd 2
threeNumberSummary(x)
```

```
##     mean   median       sd
## 5.034311 5.038198 1.680162
```

If-else statements

Oftentimes we want our code to have different effects depending on the features of the input e.g. Calculating a student's letter grade: - If grade >= 90, assign A - Otherwise, if grade >= 80, assign B - Otherwise, if grade >= 70, assign C - In all other cases, assign F

To code this up, we use if-else statements

Example: Letter grades

```
calculateLetterGrade <- function(x) {
  if(x >= 90) {
    grade <- "A"
  } else if(x >= 80) {
    grade <- "B"
  } else if(x >= 70) {
    grade <- "C"
  } else {
    grade <- "F"
  }
  grade
}

course.grades <- c(92, 78, 87, 91, 62)
sapply(course.grades, FUN=calculateLetterGrade)
```

```
## [1] "A" "C" "B" "A" "F"
```

The `return()` function

In the previous examples we specified the output simply by writing the output variable as the last line of the function. More explicitly, we can use the `return()` function

```
addOne <- function(x) {
  return(x + 1)
}

addOne(12)
```

```
## [1] 13
```

We will generally avoid the `return()` function, but you can use it if necessary or if it makes writing a particular function easier.

Reminders

- Homework 1 **due 0900 hrs on Wednesday, February 19 2020**
- Lab 2 is uploaded and **due on Wednesday, 12 Februray 2020**
- If you have questions, feel free to post on the course group