Lab 2 Solutions

Stats 32: Introduction to R for Undergraduates

Harrison Li

April 4, 2024

Your solutions to this lab should be uploaded to Gradescope as a knitted .pdf file before 1:20 pm today.

Please type text responses to each question in the space below. You may also need to write code, which must go in a code chunk.

Note: The content of this lab is borrowed heavily from Kenneth Tay's course materials in the Autumn 2019 iteration of this course.

Functions

R has many built-in functions (and many more functions in packages) that perform certain operations on one or more arguments (inputs), and returns a particular value (or output).

The help page for a function, accessed via the syntax `?functionName` where `functionName` is a placeholder for the name of the function, will show you a description of all of its arguments, followed by some text about what the function does, followed by information about its value.

1. What does the function `colSums()` (in the built-in `base` package) do? How many arguments does it have? What is the minimum number of arguments you need to specify when calling it?

Answer: From the help page (type `?colSums`), we see `colSums()` computes the sum of each column in a 2-dimensional (or higher) array (e.g. a matrix or data frame). It takes 3 arguments: `x`, `na.rm`, and `dims`, though only `x` does not have a default value, so the minimum number of arguments that need to be specified is 1.

We can also write our own functions! Let's write one called `multiply_two` that takes in a single input, `x`, converts it to numeric, and multiplies it by 2:

```
multiply_two <- function(x){
  x <- as.numeric(x)
  return(2*x)
}
```

2. Write your own function called `reciprocal` that computes the reciprocal (multiplicative inverse) of a single input `x`. Verify the function works with some different numeric values of `x`, such as 2, 0.25, and "-4". What happens if you try calling `reciprocal(0)`? How about `reciprocal("Statistics")`?

Answer:

```
reciprocal <- function(x) {
  y <- 1 / x
  return(y)
}
```

```
reciprocal(2)
```

```
## [1] 0.5
```
```
reciprocal(0.25)
```
```
## [1] 4
```
```
reciprocal(-4)
```
```
## [1] -0.25
```

If we call `reciprocal(0)` we get infinity:

```
reciprocal(0)
```
```
## [1] Inf
```

With `reciprocal("Statistics")` we get an error (commented out so that the document can knit)

```
# reciprocal("Statistics")
```

Packages

To install a package `pkgName`, simply type `install.packages("pkgName")` into the console.

To use functions in the package, we first have to load the package: `library(pkgName)`. If the package has not been installed yet, we will get an error. Otherwise, functions in the package are available for use once you have loaded the package.

Data frames

Packages not only give us access to user-created functions, but also user-created datasets, which are typically stored as data frames (or tibbles).

Let's load the `fueleconomy` package. If you haven't installed this package yet, run this command in the console first: `install.packages("fueleconomy")`.

```
library(fueleconomy)
```

Loading the package automatically adds a data frame called `vehicles` to the environment. You can see it by clicking the `Global Environment` dropdown menu, and then selecting `package:fueleconomy`.

Seeing parts of the data

33,000 observations is a lot of observations to look through. Instead of looking through all of it, we can use various functions to give us a feel for the data.

You can use the `head()` and `tail()` functions to display the first few or last few rows of the dataset.

3. Display the first 6 rows of the `vehicles` data frame. Then display the last 2 rows. (Hint: Look at the help pages for `head()` and `tail()`)

Answer:

```
head(vehicles)
```
```
##      id  make        model year           class         trans
## 1 13309 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
## 2 13310 Acura 2.2CL/3.0CL 1997 Subcompact Cars    Manual 5-spd
## 3 13311 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
## 4 14038 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
## 5 14039 Acura 2.3CL/3.0CL 1998 Subcompact Cars    Manual 5-spd
## 6 14040 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
```

```
##                 drive cyl displ    fuel hwy cty
## 1 Front-Wheel Drive   4    2.2 Regular  26   20
## 2 Front-Wheel Drive   4    2.2 Regular  28   22
## 3 Front-Wheel Drive   6    3.0 Regular  26   18
## 4 Front-Wheel Drive   4    2.3 Regular  27   19
## 5 Front-Wheel Drive   4    2.3 Regular  29   21
## 6 Front-Wheel Drive   6    3.0 Regular  26   17
```

```r
tail(vehicles, 2)
```

```
##          id make       model year            class       trans
## 33441 5498 Yugo       GV/GVX 1989  Subcompact Cars Manual 5-spd
## 33442 1745 Yugo Gy/yugo GVX 1986 Minicompact Cars Manual 4-spd
##                 drive cyl displ    fuel hwy cty
## 33441 Front-Wheel Drive   4    1.3 Regular  28   23
## 33442 Front-Wheel Drive   4    1.1 Regular  29   22
```

Under the hood, a data frame is implemented as a named list of vectors. Each column (a vector) is a single element in the list. Hence, whatever we can do with named lists, we can do with data frames. For example, we can get the data frame's column names using `names()`:

```r
names(vehicles)
```

```
##  [1] "id"    "make"  "model" "year"  "class" "trans" "drive" "cyl"   "displ"
## [10] "fuel"  "hwy"   "cty"
```

To access the values in a particular column of a data frame, we can use either the `[[` or `$` notation, just like with named lists:

```r
head(vehicles[["class"]])
```

```
## [1] "Subcompact Cars" "Subcompact Cars" "Subcompact Cars" "Subcompact Cars"
## [5] "Subcompact Cars" "Subcompact Cars"
```

```r
head(vehicles$class)
```

```
## [1] "Subcompact Cars" "Subcompact Cars" "Subcompact Cars" "Subcompact Cars"
## [5] "Subcompact Cars" "Subcompact Cars"
```

4. What type of data structure is `vehicles$class`?

Answer: It is a vector (as we are indexing into a single element of the data frame `vehicles`, viewed as a named list of vectors).

Since the number of columns in a data frame is just the number of elements in a list, we can get the number of columns using `length()`:

```r
length(vehicles)
```

```
## [1] 12
```

Interestingly, data frames can act like matrices too. For example, we can use `dim()` to figure out the number of rows and columns in the data frame:

```r
dim(vehicles)
```

```
## [1] 33442    12
```

We can also index into the data frame by position as if it were a matrix, though this is not recommended for clarity's sake:

```r
head(vehicles[,5]) # not recommended since you have to manually keep track of what the fifth column is
```

```
## [1] "Subcompact Cars" "Subcompact Cars" "Subcompact Cars" "Subcompact Cars"
## [5] "Subcompact Cars" "Subcompact Cars"
```

5. What is the `make` of the 30th vehicle in the `vehicles` data frame?

Answer:

```
vehicles$make[30]
```

```
## [1] "Acura"
```

Another useful way to get a quick snapshot of the data in a data frame is to use the `summary()` function:

```
summary(vehicles)
```

```
##        id             make               model               year
##  Min.   :    1   Length:33442       Length:33442       Min.   :1984
##  1st Qu.: 8361   Class :character   Class :character   1st Qu.:1991
##  Median :16724   Mode  :character   Mode  :character   Median :1999
##  Mean   :17038                                         Mean   :1999
##  3rd Qu.:25265                                         3rd Qu.:2008
##  Max.   :34932                                         Max.   :2015
##
##     class              trans              drive                cyl
##  Length:33442       Length:33442       Length:33442       Min.   : 2.000
##  Class :character   Class :character   Class :character   1st Qu.: 4.000
##  Mode  :character   Mode  :character   Mode  :character   Median : 6.000
##                                                           Mean   : 5.772
##                                                           3rd Qu.: 6.000
##                                                           Max.   :16.000
##                                                           NA's   :58
##     displ           fuel               hwy              cty
##  Min.   :0.000   Length:33442       Min.   :  9.00   Min.   :  6.00
##  1st Qu.:2.300   Class :character   1st Qu.: 19.00   1st Qu.: 15.00
##  Median :3.000   Mode  :character   Median : 23.00   Median : 17.00
##  Mean   :3.353                      Mean   : 23.55   Mean   : 17.49
##  3rd Qu.:4.300                      3rd Qu.: 27.00   3rd Qu.: 20.00
##  Max.   :8.400                      Max.   :109.00   Max.   :138.00
##  NA's   :57
```

We note that we don't get much useful information for the non-numeric variables - we are simply told they are of class "character". More on this later.

Getting an overview of the data

For an overview of the entire data set, the `str()` function we introduced last class is very handy. When applied to a data frame, `str()` goes through each column and tells us what type of variable it is, as well as the first couple of values for the column:

```
str(vehicles)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    33442 obs. of  12 variables:
##  $ id   : num  13309 13310 13311 14038 14039 ...
##  $ make : chr  "Acura" "Acura" "Acura" "Acura" ...
##  $ model: chr  "2.2CL/3.0CL" "2.2CL/3.0CL" "2.2CL/3.0CL" "2.3CL/3.0CL" ...
##  $ year : num  1997 1997 1997 1998 1998 ...
##  $ class: chr  "Subcompact Cars" "Subcompact Cars" "Subcompact Cars" "Subcompact Cars" ...
##  $ trans: chr  "Automatic 4-spd" "Manual 5-spd" "Automatic 4-spd" "Automatic 4-spd" ...
##  $ drive: chr  "Front-Wheel Drive" "Front-Wheel Drive" "Front-Wheel Drive" "Front-Wheel Drive" ...
```

```
## $ cyl  : num  4 4 6 4 4 6 4 4 6 5 ...
## $ displ: num  2.2 2.2 3 2.3 2.3 3 2.3 2.3 3 2.5 ...
## $ fuel : chr  "Regular" "Regular" "Regular" "Regular" ...
## $ hwy  : num  26 28 26 27 29 26 27 29 26 23 ...
## $ cty  : num  20 22 18 19 21 17 20 21 17 18 ...
```

Note that the default types for the variables may not be what you want.

6. What is a more sensible type for the `id` variable? Convert `vehicles$id` (the `id` column in the data frame) into this type, using the appropriate `as.x()` function. Reassign this to `vehicles$id` to update the content of `vehicles` appropriately.

Answer: Currently `id` is numeric:

```
typeof(vehicles$id)
```

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

Character is a more sensible type, as we do not think of ID numbers as having a quantitative interpretation, but rather as a raw sequence of digits, serving the same role as if they were letters or other non-numeric characters.

```
vehicles$id <- as.character(vehicles$id)
```

Factors

Recall that in the output of `summary(vehicles)`, we did not get any useful information about the character variables. But if our character variables can only take one of several values (eye color, age group), we might naturally want to know e.g. how many observations in our data have each of these values.

In that case, the `table()` function is helpful:

```
table(vehicles$drive)
```

```
##
##             2-Wheel Drive              4-Wheel Drive
##                       507                        699
## 4-Wheel or All-Wheel Drive            All-Wheel Drive
##                      6647                       1267
##           Front-Wheel Drive   Part-time 4-Wheel Drive
##                     12233                         96
##            Rear-Wheel Drive
##                     11993
```

Another option is to have R treat categorical variables as **factors**. Factors represent **categorical variables**: variables that can take on one of several possible values. Examples include race (in a dataset of personal characteristics) or brand (in a dataset about cereal). Categories can be unordered (e.g. eye color; we call them **nominal variables**), or ordered (e.g. age group; we call them **ordinal variables**).

We can make a character variable into a factor variable by using `as.factor()`. Then `summary()` gives more useful information.

```
vehicles$drive <- as.factor(vehicles$drive)
summary(vehicles$drive)
```

```
##             2-Wheel Drive              4-Wheel Drive
##                       507                        699
## 4-Wheel or All-Wheel Drive            All-Wheel Drive
##                      6647                       1267
##           Front-Wheel Drive   Part-time 4-Wheel Drive
```

```
##                          12233                                    96
##          Rear-Wheel Drive
##                          11993
```

Let's look at the internal structure of the factor variable:

```
str(vehicles$drive)
```

```
##  Factor w/ 7 levels "2-Wheel Drive",..: 5 5 5 5 5 5 5 5 5 5 ...
```

Notice that the words ("2 Wheel Drive", etc.) have been changed into numbers! That's because R assigns each category a number. We can see this assignment somewhat by calling `levels()`, which shows us the "levels", or categories, for this variable:

```
levels(vehicles$drive)
```

```
## [1] "2-Wheel Drive"             "4-Wheel Drive"
## [3] "4-Wheel or All-Wheel Drive" "All-Wheel Drive"
## [5] "Front-Wheel Drive"         "Part-time 4-Wheel Drive"
## [7] "Rear-Wheel Drive"
```

So 2-Wheel Drives are labeled 1, and so on. By default, R assigns this internal labeling by alphabetical order. We can change the order of this labeling using the `fct_relevel()` function in the tidyverse. For example, suppose we want `4-Wheel Drive` and `All-Wheel Drive` to be the first two levels:

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ----------------------- tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
vehicles$drive <- fct_relevel(.f=vehicles$drive, "4-Wheel Drive", "All-Wheel Drive")
levels(vehicles$drive)
```

```
## [1] "4-Wheel Drive"             "All-Wheel Drive"
## [3] "2-Wheel Drive"             "4-Wheel or All-Wheel Drive"
## [5] "Front-Wheel Drive"         "Part-time 4-Wheel Drive"
## [7] "Rear-Wheel Drive"
```

Now suppose we want to make "Front-Wheel Drive" the second level:

```
vehicles$drive <- fct_relevel(.f=vehicles$drive, "Front-Wheel Drive", after=1)
levels(vehicles$drive)
```

```
## [1] "4-Wheel Drive"             "Front-Wheel Drive"
## [3] "All-Wheel Drive"           "2-Wheel Drive"
## [5] "4-Wheel or All-Wheel Drive" "Part-time 4-Wheel Drive"
## [7] "Rear-Wheel Drive"
```

7. Convert the `class` variable in `vehicles` to a factor. How many levels does it have? Is it nominal or ordinal?

Answer:

```r
vehicles$class <- as.factor(vehicles$class)
levels(vehicles$class)
```

```
##  [1] "Compact Cars"                      "Large Cars"
##  [3] "Midsize Cars"                      "Midsize Station Wagons"
##  [5] "Midsize-Large Station Wagons"      "Minicompact Cars"
##  [7] "Minivan - 2WD"                     "Minivan - 4WD"
##  [9] "Small Pickup Trucks"               "Small Pickup Trucks 2WD"
## [11] "Small Pickup Trucks 4WD"           "Small Sport Utility Vehicle 2WD"
## [13] "Small Sport Utility Vehicle 4WD"   "Small Station Wagons"
## [15] "Special Purpose Vehicle"           "Special Purpose Vehicle 2WD"
## [17] "Special Purpose Vehicle 4WD"       "Special Purpose Vehicles"
## [19] "Special Purpose Vehicles/2wd"      "Special Purpose Vehicles/4wd"
## [21] "Sport Utility Vehicle - 2WD"       "Sport Utility Vehicle - 4WD"
## [23] "Standard Pickup Trucks"            "Standard Pickup Trucks 2WD"
## [25] "Standard Pickup Trucks 4WD"        "Standard Pickup Trucks/2wd"
## [27] "Standard Sport Utility Vehicle 2WD" "Standard Sport Utility Vehicle 4WD"
## [29] "Subcompact Cars"                   "Two Seaters"
## [31] "Vans"                              "Vans Passenger"
## [33] "Vans, Cargo Type"                  "Vans, Passenger Type"
```

We see there are 34 different levels, i.e. there are 34 different classes represented in the `vehicles` data frame. There is no inherent ordering in these levels, so the variable is nominal, not ordinal.