Tutorial 4 - Basic cleaning, loops, apply functions

Contents

- A common data cleaning task
- If-else statements
- For/while loops to iterate over data
- `apply()`, `lapply()`, `sapply()`, `tapply()`
- `with()` to specify scope

A common problem

- One of the most common problems you'll encounter when importing manually-entered data is inconsistent data types within columns
- For a simple example, let's look at `TVhours` column in a messy version of the survey data from Tutorial 2

```
survey.messy <- read.csv("http://www.andrew.cmu.edu/user/achoulde/94842/data/survey_messy.csv",
header=TRUE)
survey.messy$TVhours
```

```
##  [1] none          3.5           30            6             20
##  [6] 15            approx 6      8             10            0
## [11] 4             0             0             2             24
## [16] 10  w/Netflix 25            0             0             10
## [21] 5ish          35            2.5           2             5
## [26] adfkj         0             0             10ish         5
## [31] 40            10            0             20            0
## [36] 6             2             10            10            7
## 23 Levels: 0 10 10  w/Netflix 10ish 15 2 2.5 20 24 25 3.5 30 35 4 40 ... none
```

What's happening?

```
str(survey.messy)
```

```
## 'data.frame':    40 obs. of  6 variables:
##  $ Program        : Factor w/ 3 levels "MISM","Other",..: 1 3 3 2 1 3 2 2 3 1 ...
##  $ PriorExp       : Factor w/ 3 levels "Extensive experience",..: 1 3 3 3 3 2 3 3 1 1 ...
##  $ Rexperience    : Factor w/ 3 levels "Basic competence",..: 1 3 1 1 1 3 2 1 2 3 ...
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windows": 2 2 1 1 1 2 2 1 2 2 ...
##  $ TVhours        : Factor w/ 23 levels "0","10","10  w/Netflix",..: 23 11 12 18 8 5 22 20 2
1 ...
##  $ Editor         : Factor w/ 3 levels "LaTeX","Microsoft Excel",..: 3 3 3 2 1 3 3 3 3 3 ...
```

- Several of the entries have non-numeric values in them (they contain strings)

- As a result, `TVhours` is being imported as `factor`

Attempt at a fix

- What if we just try to cast it back to numeric?

```
tv.hours.messy <- survey.messy$TVhours
tv.hours.messy
```

```
##  [1] none          3.5           30            6             20
##  [6] 15            approx 6      8             10            0
## [11] 4             0             0             2             24
## [16] 10  w/Netflix 25            0             0             10
## [21] 5ish          35            2.5           2             5
## [26] adfkj         0             0             10ish         5
## [31] 40            10            0             20            0
## [36] 6             2             10            10            7
## 23 Levels: 0 10 10  w/Netflix 10ish 15 2 2.5 20 24 25 3.5 30 35 4 40 ... none
```

```
as.numeric(tv.hours.messy)
```

```
##  [1] 23 11 12 18  8  5 22 20  2  1 14  1  1  6  9  3 10  1  1  2 17 13  7
## [24]  6 16 21  1  1  4 16 15  2  1  8  1 18  6  2  2 19
```

That didn't work…

```
tv.hours.messy
as.numeric(tv.hours.messy)
```

```
##  [1] none          3.5           30            6             20
##  [6] 15            approx 6      8             10            0
## [11] 4             0             0             2             24
## [16] 10  w/Netflix 25            0             0             10
## [21] 5ish          35            2.5           2             5
## [26] adfkj         0             0             10ish         5
## [31] 40            10            0             20            0
## [36] 6             2             10            10            7
## 23 Levels: 0 10 10  w/Netflix 10ish 15 2 2.5 20 24 25 3.5 30 35 4 40 ... none
```

```
##  [1] 23 11 12 18  8  5 22 20  2  1 14  1  1  6  9  3 10  1  1  2 17 13  7
## [24]  6 16 21  1  1  4 16 15  2  1  8  1 18  6  2  2 19
```

- This just converted all the values into the integer-coded levels of the factor
- Not what we wanted!

Something that does work

- Consider the following simple example

```
num.vec <- c(3.1, 2.5)
as.factor(num.vec)
```

```
## [1] 3.1 2.5
## Levels: 2.5 3.1
```

```
as.numeric(as.factor(num.vec))
```

```
## [1] 2 1
```

```
as.numeric(as.character(as.factor(num.vec)))
```

```
## [1] 3.1 2.5
```

> If we take a number that's being coded as a factor and *first* turn it into a `character` string, *then* converting the string to a numeric gets back the number

Back to the corrupted TVhours column

```
as.character(tv.hours.messy)
```

```
##  [1] "none"          "3.5"           "30"            "6"
##  [5] "20"            "15"            "approx 6"      "8"
##  [9] "10"            "0"             "4"             "0"
## [13] "0"             "2"             "24"            "10  w/Netflix"
## [17] "25"            "0"             "0"             "10"
## [21] "5ish"          "35"            "2.5"           "2"
## [25] "5"             "adfkj"         "0"             "0"
## [29] "10ish"         "5"             "40"            "10"
## [33] "0"             "20"            "0"             "6"
## [37] "2"             "10"            "10"            "7"
```

```
as.numeric(as.character(tv.hours.messy))
```

```
## Warning: NAs introduced by coercion
```

```
##  [1]   NA  3.5 30.0  6.0 20.0 15.0   NA  8.0 10.0  0.0  4.0  0.0  0.0  2.0
## [15] 24.0   NA 25.0  0.0  0.0 10.0   NA 35.0  2.5  2.0  5.0   NA  0.0  0.0
## [29]   NA  5.0 40.0 10.0  0.0 20.0  0.0  6.0  2.0 10.0 10.0  7.0
```

```
typeof(as.numeric(as.character(tv.hours.messy)))  # Success!! (Almost...)
```

```
## Warning in typeof(as.numeric(as.character(tv.hours.messy))): NAs introduced
## by coercion
```

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

A small improvement

- All the corrupted cells now appear as `NA`, which is R's missing indicator
- We can do a little better by cleaning up the vector once we get it to `character` form

```
tv.hours.strings <- as.character(tv.hours.messy)
tv.hours.strings
```

```
##  [1] "none"          "3.5"           "30"            "6"
##  [5] "20"            "15"            "approx 6"      "8"
##  [9] "10"            "0"             "4"             "0"
## [13] "0"             "2"             "24"            "10  w/Netflix"
## [17] "25"            "0"             "0"             "10"
## [21] "5ish"          "35"            "2.5"           "2"
## [25] "5"             "adfkj"         "0"             "0"
## [29] "10ish"         "5"             "40"            "10"
## [33] "0"             "20"            "0"             "6"
## [37] "2"             "10"            "10"            "7"
```

Deleting non-numeric (or .) characters

```
tv.hours.strings
```

```
##  [1] "none"          "3.5"           "30"            "6"
##  [5] "20"            "15"            "approx 6"      "8"
##  [9] "10"            "0"             "4"             "0"
## [13] "0"             "2"             "24"            "10  w/Netflix"
## [17] "25"            "0"             "0"             "10"
## [21] "5ish"          "35"            "2.5"           "2"
## [25] "5"             "adfkj"         "0"             "0"
## [29] "10ish"         "5"             "40"            "10"
## [33] "0"             "20"            "0"             "6"
## [37] "2"             "10"            "10"            "7"
```

```
# Use gsub() to replace everything except digits and '.' with a blank ""
gsub("[^0-9.]", "", tv.hours.strings)
```

```
##  [1] ""    "3.5" "30"  "6"   "20"  "15"  "6"   "8"   "10"  "0"   "4"
## [12] "0"   "0"   "2"   "24"  "10"  "25"  "0"   "0"   "10"  "5"   "35"
## [23] "2.5" "2"   "5"   ""    "0"   "0"   "10"  "5"   "40"  "10"  "0"
## [34] "20"  "0"   "6"   "2"   "10"  "10"  "7"
```

The final product

```
tv.hours.messy[1:30]
```

```
##  [1] none          3.5           30            6             20
##  [6] 15            approx 6      8             10            0
## [11] 4             0             0             2             24
## [16] 10  w/Netflix 25            0             0             10
## [21] 5ish          35            2.5           2             5
## [26] adfkj         0             0             10ish         5
## 23 Levels: 0 10 10  w/Netflix 10ish 15 2 2.5 20 24 25 3.5 30 35 4 40 ... none
```

```
tv.hours.clean <- as.numeric(gsub("[^0-9.]", "", tv.hours.strings))
tv.hours.clean
```

```
##  [1]   NA  3.5 30.0  6.0 20.0 15.0  6.0  8.0 10.0  0.0  4.0  0.0  0.0  2.0
## [15] 24.0 10.0 25.0  0.0  0.0 10.0  5.0 35.0  2.5  2.0  5.0   NA  0.0  0.0
## [29] 10.0  5.0 40.0 10.0  0.0 20.0  0.0  6.0  2.0 10.0 10.0  7.0
```

- As a last step, we should go through and figure out if any of the `NA` values should really be `0`.
- This step is not shown here.

Rebuilding our data

```
survey <- transform(survey.messy, TVhours = tv.hours.clean)
str(survey)
```

```
## 'data.frame':    40 obs. of  6 variables:
##  $ Program        : Factor w/ 3 levels "MISM","Other",..: 1 3 3 2 1 3 2 2 3 1 ...
##  $ PriorExp       : Factor w/ 3 levels "Extensive experience",..: 1 3 3 3 3 2 3 3 1 1 ...
##  $ Rexperience    : Factor w/ 3 levels "Basic competence",..: 1 3 1 1 1 3 2 1 2 3 ...
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windows": 2 2 1 1 1 2 2 1 2 2 ...
##  $ TVhours        : num  NA 3.5 30 6 20 15 6 8 10 0 ...
##  $ Editor         : Factor w/ 3 levels "LaTeX","Microsoft Excel",..: 3 3 3 2 1 3 3 3 3 3 ...
```

- **Success!**

A different approach

- We can also handle this problem by setting `stringsAsFactors = FALSE` when importing our data.

```
survey.messy <- read.csv("http://www.andrew.cmu.edu/user/achoulde/94842/data/survey_messy.csv",
header=TRUE, stringsAsFactors=FALSE)
str(survey.messy)
```

```
## 'data.frame':    40 obs. of  6 variables:
##  $ Program        : chr  "MISM" "PPM" "PPM" "Other" ...
##  $ PriorExp       : chr  "Extensive experience" "Some experience" "Some experience" "Some exp
erience" ...
##  $ Rexperience    : chr  "Basic competence" "Never used" "Basic competence" "Basic competenc
e" ...
##  $ OperatingSystem: chr  "Windows" "Windows" "Mac OS X" "Mac OS X" ...
##  $ TVhours        : chr  "none" "3.5" "30" "6" ...
##  $ Editor         : chr  "Microsoft Word" "Microsoft Word" "Microsoft Word" "Microsoft Excel"
...
```

- Now everything is a `character` instead of a `factor`

One-line cleanup

- Let's clean up the `TVhours` column and cast it to numeric all in one command

```
survey <- transform(survey.messy,
                    TVhours = as.numeric(gsub("[^0-9.]", "", TVhours)))
str(survey)
```

```
## 'data.frame':    40 obs. of  6 variables:
##  $ Program        : chr  "MISM" "PPM" "PPM" "Other" ...
##  $ PriorExp       : chr  "Extensive experience" "Some experience" "Some experience" "Some exp
erience" ...
##  $ Rexperience    : chr  "Basic competence" "Never used" "Basic competence" "Basic competenc
e" ...
##  $ OperatingSystem: chr  "Windows" "Windows" "Mac OS X" "Mac OS X" ...
##  $ TVhours        : num  NA 3.5 30 6 20 15 6 8 10 0 ...
##  $ Editor         : chr  "Microsoft Word" "Microsoft Word" "Microsoft Word" "Microsoft Excel"
...
```

What about all those other `character` variables?

```
table(survey[["Program"]])
```

```
##
##  MISM Other   PPM
##    22     9     9
```

```
table(as.factor(survey[["Program"]]))
```

```
##
##  MISM Other   PPM
##    22     9     9
```

- Having factors coded as characters may be OK for many parts of our analysis

To be safe, let's fix things

```
# Figure out which columns are coded as characters
chr.indexes <- sapply(survey, FUN = is.character)
chr.indexes
```

```
##         Program        PriorExp     Rexperience OperatingSystem
##            TRUE            TRUE            TRUE            TRUE
##         TVhours          Editor
##           FALSE            TRUE
```

```
# Re-code all of the character columns to factors
survey[chr.indexes] <- lapply(survey[chr.indexes], FUN = as.factor)
```

Here's the outcome

```
str(survey)
```

```
## 'data.frame':    40 obs. of  6 variables:
##  $ Program        : Factor w/ 3 levels "MISM","Other",..: 1 3 3 2 1 3 2 2 3 1 ...
##  $ PriorExp       : Factor w/ 3 levels "Extensive experience",..: 1 3 3 3 3 2 3 3 1 1 ...
##  $ Rexperience    : Factor w/ 3 levels "Basic competence",..: 1 3 1 1 1 3 2 1 2 3 ...
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windows": 2 2 1 1 1 2 2 1 2 2 ...
##  $ TVhours        : num  NA 3.5 30 6 20 15 6 8 10 0 ...
##  $ Editor         : Factor w/ 3 levels "LaTeX","Microsoft Excel",..: 3 3 3 2 1 3 3 3 3 3 ...
```

- **Success!**

Another common problem

- On Homework 2 you'll learn how to wrangle with another common problem
- When data is entered manually, misspellings and case changes are very common
- E.g., a column showing life support mechanism may look like,

```
life.support <- as.factor(c("dialysis", "Ventilation", "Dialysis", "dialysis", "none", "None",
"nnone", "dyalysis", "dialysis", "ventilation", "none"))
summary(life.support)
```

```
##    dialysis    Dialysis    dyalysis       nnone        none        None
##           3           1           1           1           2           1
## ventilation Ventilation
##           1           1
```

```
summary(life.support)
```

```
##    dialysis    Dialysis    dyalysis       nnone        none        None
##           3           1           1           1           2           1
## ventilation Ventilation
##           1           1
```

- This factor has 8 levels even though it should have 3 (dialysis, ventilation, none)
- We can fix many of the typos by running spellcheck in Excel before importing data, or by changing the values on a case-by-case basis later
- There's a faster way to fix just the capitalization issue (this is an exercise on Homework 2)

What are all these [l/s/t/]apply() functions?

- These are all efficient ways of applying a function to margins of an array or elements of a list
- Before we talk about the details of `apply()` and its relatives, we should first understand loops
- **loops** are ways of iterating over data
- The `apply()` functions can be thought of as good *alternatives* to loops

For loops: a pair of examples

```
for(i in 1:4) {
  print(i)
}
```

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

```
phrase <- "Good Night, "
for(word in c("and", "Good", "Luck")) {
  phrase <- paste(phrase, word)
  print(phrase)
}
```

```
## [1] "Good Night,  and"
## [1] "Good Night,  and Good"
## [1] "Good Night,  and Good Luck"
```

For loops: syntax

> A **for loop** executes a chunk of code for every value of an **index variable** in an **index set**

- The basic syntax takes the form

```
for(index.variable in index.set) {
  code to be repeated at every value of index.variable
}
```

- The index set is often a vector of integers, but can be more general

Example

```
index.set <- list(name="Michael", weight=185, is.male=TRUE) # a list
for(i in index.set) {
  print(c(i, typeof(i)))
}
```

```
## [1] "Michael"   "character"
## [1] "185"    "double"
## [1] "TRUE"    "logical"
```

Example: Calculate sum of each column

```
fake.data <- matrix(rnorm(500), ncol=5) # create fake 100 x 5 data set
head(fake.data,2) # print first two rows
```

```
##            [,1]       [,2]       [,3]        [,4]       [,5]
## [1,] -1.0259650 1.08298791 -0.7780167  0.18089993 -0.8426907
## [2,]  0.5819787 0.04846354 -1.0778208 -0.08477589 -1.6207339
```

```
col.sums <- numeric(ncol(fake.data)) # variable to store running column sums
for(i in 1:nrow(fake.data)) {
  col.sums <- col.sums + fake.data[i,] # add ith observation to the sum
}
col.sums
```

```
## [1] -24.0300771  13.3285305   1.9366008  -3.9194567   0.8717915
```

```
colSums(fake.data) # A better approach (see also colMeans())
```

```
## [1] -24.0300771  13.3285305   1.9366008  -3.9194567   0.8717915
```

while loops

- **while loops** repeat a chunk of code while the specified condition remains true

```
day <- 1
num.days <- 365
while(day <= num.days) {
  day <- day + 1
}
```

- We won't really be using while loops in this class
- Just be aware that they exist, and that they may become useful to you at some point in your analytics career

The various apply() functions

Command	Description
`apply(X, MARGIN, FUN)`	Obtain a vector/array/list by applying `FUN` along the specified `MARGIN` of an array or matrix `X`
`lapply(X, FUN)`	Obtain a list by applying `FUN` to the elements of a list `X`
`sapply(X, FUN)`	Simplified version of `lapply`. Returns a vector/array instead of list.
`tapply(X, INDEX, FUN)`	Obtain a table by applying `FUN` to each combination of the factors given in `INDEX`

- These functions are (good!) alternatives to loops
- They are typically *more efficient* than loops (often run considerably faster on large data sets)
- Take practice to get used to, but make analysis easier to debug and less prone to error when used effectively
- You can always type `example(function)` to get code examples (E.g., `example(apply)`)

Example: apply()

```
colMeans(fake.data)
```

```
## [1] -0.240300771  0.133285305  0.019366008 -0.039194567  0.008717915
```

```
apply(fake.data, MARGIN=2, FUN=mean) # MARGIN = 1 for rows, 2 for columns
```

```
## [1] -0.240300771  0.133285305  0.019366008 -0.039194567  0.008717915
```

```
# Function that calculates proportion of vector indexes that are > 0
propPositive <- function(x) mean(x > 0)
apply(fake.data, MARGIN=2, FUN=propPositive)
```

```
## [1] 0.38 0.60 0.51 0.48 0.51
```

Example: lapply(), sapply()

```
lapply(survey, is.factor) # Returns a list
```

```
## $Program
## [1] TRUE
##
## $PriorExp
## [1] TRUE
##
## $Rexperience
## [1] TRUE
##
## $OperatingSystem
## [1] TRUE
##
## $TVhours
## [1] FALSE
##
## $Editor
## [1] TRUE
```

```
sapply(survey, FUN = is.factor) # Returns a vector with named elements
```

```
##         Program        PriorExp     Rexperience OperatingSystem
##            TRUE            TRUE            TRUE            TRUE
##         TVhours          Editor
##           FALSE            TRUE
```

Example: apply(), lapply(), sapply()

```
apply(cars, 2, FUN=mean) # Data frames are arrays
```

```
## speed  dist
## 15.40 42.98
```

```
lapply(cars, FUN=mean) # Data frames are also lists
```

```
## $speed
## [1] 15.4
##
## $dist
## [1] 42.98
```

```
sapply(cars, FUN=mean) # sapply() is just simplified lapply()
```

```
## speed  dist
## 15.40 42.98
```

Example: tapply()

- Think of `tapply()` as a generalized form of the `table()` function

```
library(MASS)
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     survey
```

```
# Get a count table, data broken down by Origin and DriveTrain
table(Cars93$Origin, Cars93$DriveTrain)
```

```
##
##           4WD Front Rear
##   USA       5    34    9
##   non-USA   5    33    7
```

```
# Calculate average MPG.City, broken down by Origin and Drivetrain
tapply(Cars93$MPG.city, INDEX = Cars93[c("Origin", "DriveTrain")], FUN=mean)
```

```
##          DriveTrain
## Origin     4WD    Front     Rear
##   USA     17.6 22.14706 18.33333
##   non-USA 23.4 24.93939 19.14286
```

Example: tapply()

- Let's get the average horsepower by car `Origin` and `Type`

```
tapply(Cars93[["Horsepower"]], INDEX = Cars93[c("Origin", "Type")], FUN=mean)
```

```
##          Type
## Origin     Compact    Large  Midsize    Small   Sporty    Van
##   USA     117.4286 179.4545 153.5000 89.42857 166.5000 158.40
##   non-USA 141.5556       NA 189.4167 91.78571 151.6667 138.25
```

- What's that `NA` doing there?

```
any(Cars93$Origin == "non-USA" & Cars93$Type == "Large")
```

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

- None of the non-USA manufacturers produced Large cars!

Example: using tapply() to mimic table()

- Here's how one can use `tapply()` to produce the same output as the `table()` function

```
library(MASS)
# Get a count table, data broken down by Origin and DriveTrain
table(Cars93$Origin, Cars93$DriveTrain)
```

```
##
##           4WD Front Rear
##   USA       5    34    9
##   non-USA   5    33    7
```

```
# This one may take a moment to figure out...
tapply(rep(1, nrow(Cars93)), INDEX = Cars93[c("Origin", "DriveTrain")], FUN=sum)
```

```
##          DriveTrain
## Origin    4WD Front Rear
##   USA       5    34    9
##   non-USA   5    33    7
```

with()

- Thus far we've repeatedly typed out the data frame name when referencing its columns
- This is because the data variables don't exist in our working environment
- Using **with** `(data, expr)` lets us specify that the code in `expr` should be evaluated in an environment that contains the elements of `data` as variables

```
with(Cars93, table(Origin, Type))
```

```
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           7    11      10     7      8   5
##   non-USA       9     0      12    14      6   4
```

Example: with()

```
any(Cars93$Origin == "non-USA" & Cars93$Type == "Large")
```

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

```
with(Cars93, any(Origin == "non-USA" & Type == "Large")) # Same effect!
```

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

```
with(Cars93, tapply(Horsepower, INDEX = list(Origin, Type), FUN=mean))
```

```
##          Compact    Large  Midsize    Small   Sporty    Van
## USA     117.4286 179.4545 153.5000 89.42857 166.5000 158.40
## non-USA 141.5556       NA 189.4167 91.78571 151.6667 138.25
```

- Using with() makes code simpler, easier to read, and easier to debug

Next

- Complete **Lab 4** (http://isle.heinz.cmu.edu/94-842/lab04/)