Chapter 6: Data Cleaning, Loops, apply functions

M Affouf

1/8/2018

part1

Data

- We will be using multiple data sets in this lecture:
  - Salary, Monument, Circulator, and Restaurant from OpenBaltimore: `https://data.baltimorecity.gov/browse?limitTo=datasets`
  - Gap Minder - very interesting way of viewing longitudinal data
    - Data is here - `http://www.gapminder.org/data/`
  - `http://spreadsheets.google.com/pub?key=rMsQHawTObBb6_U2ESjKXYw&output=xls`

Data Cleaning

In general, data cleaning is a process of investigating your data for inaccuracies, or recoding it in a way that makes it more manageable.

MOST IMPORTANT RULE - LOOK AT YOUR DATA!

Again - `table`, `summarize`, `is.na`, `any`, `all` are useful.

Dealing with Missing Data

Missing data types

One of the most important aspects of data cleaning is missing values.

Types of "missing" data:

- `NA` - general missing data
- `NaN` - stands for "**N**ot **a** **N**umber", happens when you do 0/0.
- `Inf` and `-Inf` - **I**nfinity, happens when you take a positive number (or negative number) by 0.

Finding Missing data

Each missing data type has a function that returns `TRUE` if the data is missing:

- `NA` - `is.na`
- `NaN` - `is.nan`
- `Inf` and `-Inf` - `is.infinite`
- `is.finite` returns `FALSE` for all missing data and `TRUE` for non-missing
- `complete.cases` on a `data.frame`/`matrix` returns `TRUE` if all values in that row of the object are not missing.

Missing Data with Logicals

One important aspect (esp with subsetting) is that logical operations return `NA` for `NA` values. Think about it, the data could be > 2 or not we don't know, so `R` says there is no `TRUE` or `FALSE`, so that is missing:

```
x = c(0, NA, 2, 3, 4)
x > 2
```

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

Missing Data with Logicals

What to do? What if we want if `x > 2` and `x` isn't `NA`?
Don't do `x != NA`, do `x > 2` and `x` is NOT `NA`:

```
x != NA
```

```
## [1] NA NA NA NA NA
```

```
x > 2 & !is.na(x)
```

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

Missing Data with Logicals

What about seeing if a value is equal to multiple values? You can do `(x == 1 | x == 2) & !is.na(x)`, but that is not efficient. Introduce the `%in%` operator:

```
(x == 0 | x == 2) # has NA
```

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

```
(x == 0 | x == 2) & !is.na(x) # No NA
```

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

```
x %in% c(0, 2) # NEVER has NA and returns logical
```

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

Missing Data with Operations

Similarly with logicals, operations/arithmetic with `NA` will result in `NA`s:

```
x + 2
```

```
## [1]  2 NA  4  5  6
```

```
x * 2
```

```
## [1]  0 NA  4  6  8
```

Tables and Tabulations

Creating One-way Tables

Here we will use `table` to make tabulations of the data. Look at `?table` to see options for missing data.

```r
table(x)
```

```
## x
## 0 2 3 4
## 1 1 1 1
```

```r
table(x, useNA = "ifany")
```

```
## x
##    0    2    3    4 <NA>
##    1    1    1    1    1
```

Creating One-way Tables

You can set `useNA = "always"` to have it always have a column for `NA`

```
table(c(0, 1, 2, 3, 2, 3, 3, 2,2, 3),
        useNA = "always")
```

```
##
##    0    1    2    3 <NA>
##    1    1    4    4    0
```

Creating Two-way Tables

A two-way table. If you pass in 2 vectors, `table` creates a 2-dimensional table.

```
tab <- table(c(0, 1, 2, 3, 2, 3, 3, 2,2, 3),
             c(0, 1, 2, 3, 2, 3, 3, 4, 4, 3),
              useNA = "always")
```

Finding Row or Column Totals

`margin.table` finds the marginal sums of the table. `margin` is 1 for rows, 2 for columns in general in `R`. Here is the column sums of the table:

```
margin.table(tab, 2)
```

```
##
##    0    1    2    3    4 <NA>
##    1    1    2    4    2    0
```

Proportion Tables

prop.table finds the marginal proportions of the table. Think of it dividing the table by it's respective marginal totals. If margin not set, divides by overall total.

```
prop.table(tab)
```

```
##
##           0   1   2   3   4 <NA>
##   0     0.1 0.0 0.0 0.0 0.0  0.0
##   1     0.0 0.1 0.0 0.0 0.0  0.0
##   2     0.0 0.0 0.2 0.0 0.2  0.0
##   3     0.0 0.0 0.0 0.4 0.0  0.0
##   <NA> 0.0 0.0 0.0 0.0 0.0  0.0
```

```
prop.table(tab,1)
```

```
##
##           0   1   2   3   4 <NA>
```

Download Salary FY2014 Data

From https://data.baltimorecity.gov/City-Government/Baltimore-City-Employee-Salaries-FY2014/2j28-xzd7

Read the CSV into R `Sal`:

```r
Sal = read.csv("Baltimore_City_Employee_Salaries_FY2014.csv
               as.is = TRUE)
```

Checking for logical conditions

- `any()` - checks if there are any `TRUE`s
- `all()` - checks if ALL are true

```r
head(Sal,2)
```

```
##                     Name                           JobTitle AgencyI
## 1   Aaron,Keontae E                          AIDE BLUE CHIP    W022(
## 2  Aaron,Patricia G Facilities/Office Services II    A030:
##                      Agency   HireDate AnnualSalary   GrossPay
## 1          Youth Summer   06/10/2013    $11310.00    $873.63
## 2 OED-Employment Dev   10/24/1979    $53428.00 $52868.38
```

```r
any(is.na(Sal$Name)) # are there any NAs?
```

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

Recoding Variables

Example of Recoding: base R

For example, let's say gender was coded as Male, M, m, Female, F, f. Using Excel to find all of these would be a matter of filtering and changing all by hand or using if statements.

In R, you can simply do something like:

```
# data$gender[data$gender %in%
# c("Male", "M", "m")] <- "Male"
```

Example of Recoding with `recode`: car package

You can also `recode` a vector:

```
library(car, quietly = TRUE)
x = rep(c("Male", "M", "m", "f", "Female", "female" ),
        each = 3)
car::recode(x, "c('m', 'M', 'male') = 'Male';
          c('f', 'F', 'female') = 'Female';")
```

```
##  [1] "Male"   "Male"   "Male"   "Male"   "Male"   "Male'
##  [8] "Male"   "Male"   "Female" "Female" "Female" "Femal
## [15] "Female" "Female" "Female" "Female"
```

Example of Recoding with revalue: plyr

You can also revalue a vector with the revalue command

```
library(plyr)
plyr::revalue(x, c("M" = "Male", "m" = "Male",
                   "f" = "Female", "female" = "Female"))
```

```
##  [1] "Male"   "Male"   "Male"   "Male"   "Male"   "Male'
##  [8] "Male"   "Male"   "Female" "Female" "Female" "Femal
## [15] "Female" "Female" "Female" "Female"
```

Example of Cleaning: more complicated

Sometimes though, it's not so simple. That's where functions that find patterns come in very useful.

```
table(gender)
```

```
## gender
##      F FeMAle FEMALE     Fm      M     Ma   mAle   Male
##     75     82     74     89     89     79     87     89
##    Man  Woman
##     73     80
```

String functions

Pasting strings with paste and paste0

Paste can be very useful for joining vectors together:

```r
paste("Visit", 1:5, sep = "_")
```

```
## [1] "Visit_1" "Visit_2" "Visit_3" "Visit_4" "Visit_5"
```

```r
paste("Visit", 1:5, sep = "_", collapse = " ")
```

```
## [1] "Visit_1 Visit_2 Visit_3 Visit_4 Visit_5"
```

```r
paste("To", "is going be the ", "we go to the store!", sep
```

```
## [1] "Today is going be the day we go to the store!"
```

```r
# and paste0 can be even simpler see ?paste0
paste0("Visit",1:5)
```

```
## [1] "Visit1" "Visit2" "Visit3" "Visit4" "Visit5"
```

Paste Depicting How Collapse Works

```r
paste(1:5)
```

```
## [1] "1" "2" "3" "4" "5"
```

```r
paste(1:5, collapse = " ")
```

```
## [1] "1 2 3 4 5"
```

Useful String Functions

Useful String functions

- `toupper()`, `tolower()` - uppercase or lowercase your data:
- `str_trim()` (in the `stringr` package) or `trimws` in base
  - will trim whitespace
- `nchar` - get the number of characters in a string
- `paste()` - paste strings together with a space
- `paste0` - paste strings together with no space as default

The `stringr` package

Like `dplyr`, the `stringr` package:

- Makes some things more intuitive
- Is different than base R
- Is used on forums for answers
- Has a standard format for most functions
    - the first argument is a string like first argument is a `data.frame` in `dplyr`

Splitting/Find/Replace and Regular Expressions

- R can do much more than find exact matches for a whole string
- Like Perl and other languages, it can use regular expressions.
- What are regular expressions?
  - Ways to search for specific strings
  - Can be very complicated or simple
  - Highly Useful - think “Find” on steroids

A bit on Regular Expressions

- `http://www.regular-expressions.info/reference.html`
- They can use to match a large number of strings in one statement
- `.` matches any single character
- `*` means repeat as many (even if 0) more times the last character
- `?` makes the last thing optional
- `^` matches start of vector `^a` - starts with "a"
- `$` matches end of vector `b$` - ends with "b"

Splitting Strings

Substringing

Very similar:

Base R

- `substr(x, start, stop)` - substrings from position start to position stop
- `strsplit(x, split)` - splits strings up - returns list!

`stringr`

- `str_sub(x, start, end)` - substrings from position start to position end
- `str_split(string, pattern)` - splits strings up - returns list!

Splitting String: base R

In base R, `strsplit` splits a vector on a string into a list

```
x <- c("I really", "like writing", "R code programs")
y <- strsplit(x, split = " ") # returns a list
y
```

```
## [[1]]
## [1] "I"      "really"
##
## [[2]]
## [1] "like"    "writing"
##
## [[3]]
## [1] "R"         "code"     "programs"
```

Splitting String: stringr

stringr::str_split do the same thing:

```r
library(stringr)
y2 <- str_split(x, " ") # returns a list
y2
```

```
## [[1]]
## [1] "I"      "really"
##
## [[2]]
## [1] "like"    "writing"
##
## [[3]]
## [1] "R"        "code"     "programs"
```

Using a fixed expression

One example case is when you want to split on a period ". ". In regular expressions `.` means **ANY** character, so

```
str_split("I.like.strings", ".")
```

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

```
str_split("I.like.strings", fixed("."))
```

```
## [[1]]
## [1] "I"       "like"    "strings"
```

Let's extract from y

```
suppressPackageStartupMessages(library(dplyr)) # must be lo
y[[2]]
```

```
## [1] "like"    "writing"
```

```
sapply(y, dplyr::first) # on the fly
```

```
## [1] "I"    "like" "R"
```

```
sapply(y, nth, 2) # on the fly
```

```
## [1] "really"  "writing" "code"
```

```
sapply(y, last) # on the fly
```

```
## [1] "really"   "writing"  "programs"
```

‘Find’ functions: base R

`grep`: `grep`, `grepl`, `regexpr` and `gregexpr` search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.

`grep(pattern, x, fixed=FALSE)`, where:

- pattern = character string containing a regular expression to be matched in the given character vector.
- x = a character vector where matches are sought, or an object which can be coerced by as.character to a character vector.
- If fixed=TRUE, it will do exact matching for the phrase anywhere in the vector (regular find)

'Find' functions: `stringr`

`str_detect`, `str_subset`, `str_replace`, and `str_replace_all` search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.

- `str_detect` - returns `TRUE` if `pattern` is found
- `str_subset` - returns only the strings which pattern were detected
  - convenient wrapper around `x[str_detect(x, pattern)]`
- `str_extract` - returns only strings which pattern were detected, but ONLY the pattern
- `str_replace` - replaces `pattern` with `replacement` the first time
- `str_replace_all` - replaces `pattern` with `replacement` as many times matched

‘Find’ functions: stringr compared to base R

Base R does not use these functions. Here is a “translator” of the `stringr` function to base R functions

- `str_detect` - similar to `grepl` (return logical)
- `grep(value = FALSE)` is similar to `which(str_detect())`
- `str_subset` - similar to `grep(value = TRUE)` - return value of matched
- `str_replace` - similar to `sub` - replace one time
- `str_replace_all` - similar to `gsub` - replace many times

Let's look at modifier for `stringr`

```
?modifiers
```

- `fixed` - match everything exactly
- `regexp` - default - uses **reg**ular **exp**ressions
- `ignore_case` is an option to not have to use `tolower`

Important Comparisons

Base R:

- Argument order is `(pattern, x)`
- Uses option `(fixed = TRUE)`

`stringr`

- Argument order is `(string, pattern)` aka `(x, pattern)`
- Uses function `fixed(pattern)`

‘Find’ functions: Finding Indices

These are the indices where the pattern match occurs:

```
grep("Rawlings",Sal$Name)
```

```
## [1] 13832 13833 13834 13835
```

```
which(grepl("Rawlings", Sal$Name))
```

```
## [1] 13832 13833 13834 13835
```

```
which(str_detect(Sal$Name, "Rawlings"))
```

```
## [1] 13832 13833 13834 13835
```

‘Find’ functions: Finding Logicals

These are the indices where the pattern match occurs:

```r
head(grepl("Rawlings",Sal$Name))
```

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

```r
head(str_detect(Sal$Name, "Rawlings"))
```

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

'Find' functions: finding values, base R

```
grep("Rawlings",Sal$Name,value=TRUE)
```

```
## [1] "Rawlings,Kellye A"          "Rawlings,MarqWell D"
## [3] "Rawlings,Paula M"           "Rawlings-Blake,Stephar
```

```
Sal[grep("Rawlings",Sal$Name),]
```

```
##                            Name              JobTitle Ag
## 13832         Rawlings,Kellye A EMERGENCY DISPATCHER
## 13833       Rawlings,MarqWell D        AIDE BLUE CHIP
## 13834          Rawlings,Paula M        COMMUNITY AIDE
## 13835 Rawlings-Blake,Stephanie C                 MAYOR
##                      Agency   HireDate AnnualSalary    Gro
## 13832 M-R Info Technology  01/06/2003    $47980.00  $684
## 13833        Youth Summer  06/15/2012    $11310.00     $5
## 13834      R&P-Recreation  12/10/2007    $19802.00    $81
## 13835       Mayors Office  12/07/1995   $163365.00 $1612
```

'Find' functions: finding values, stringr and dplyr

```r
str_subset(Sal$Name, "Rawlings")
```

```
## [1] "Rawlings,Kellye A"          "Rawlings,MarqWell D"
## [3] "Rawlings,Paula M"           "Rawlings-Blake,Stephar
```

```r
Sal %>% filter(str_detect(Name, "Rawlings"))
```

```
##                         Name            JobTitle Agency
## 1          Rawlings,Kellye A EMERGENCY DISPATCHER   A40
## 2        Rawlings,MarqWell D       AIDE BLUE CHIP   W02
## 3           Rawlings,Paula M       COMMUNITY AIDE   A04
## 4 Rawlings-Blake,Stephanie C                MAYOR   A01
##                  Agency   HireDate AnnualSalary   GrossPa
## 1 M-R Info Technology  01/06/2003    $47980.00  $68426.
## 2       Youth Summer   06/15/2012    $11310.00    $507.
## 3      R&P-Recreation  12/10/2007    $19802.00   $8195.
## 4       Mayors Office  12/07/1995   $163365.00 $161219.
```

Showing differnce in str_extract

str_extract extracts just the matched string

```
ss = str_extract(Sal$Name, "Rawling")
head(ss)
```

```
## [1] NA NA NA NA NA NA
```

```
ss[ !is.na(ss)]
```

```
## [1] "Rawling" "Rawling" "Rawling" "Rawling"
```

Showing differnce in str_extract and str_extract_all

`str_extract_all` extracts all the matched strings

```
head(str_extract(Sal$AgencyID, "\\d"))
```

```
## [1] "0" "0" "2" "6" "9" "4"
```

```
head(str_extract_all(Sal$AgencyID, "\\d"), 2)
```

```
## [[1]]
## [1] "0" "2" "2" "0" "0"
##
## [[2]]
## [1] "0" "3" "0" "3" "1"
```

Using Regular Expressions

- Look for any name that starts with:
  - Payne at the beginning,
  - Leonard and then an S
  - Spence then capital C

```
head(grep("^Payne.*", x = Sal$Name, value = TRUE), 3)
```

```
## [1] "Payne El,Jackie"          "Payne Johnson,Nickole A"
## [3] "Payne,Chanel"
```

```
head(grep("Leonard.?S", x = Sal$Name, value = TRUE))
```

```
## [1] "Payne,Leonard S"       "Szumlanski,Leonard S"
```

```
head(grep("Spence.*C.*", x = Sal$Name, value = TRUE))
```

```
## [1] "Greene,Spencer C"    "Spencer,Charles A"   "Spencer
## [4] "Spencer,Clarence W"  "Spencer,Michael C"
```

Using Regular Expressions: stringr

```r
head(str_subset( Sal$Name, "^Payne.*"), 3)
```

```
## [1] "Payne El,Jackie"         "Payne Johnson,Nickole A"
## [3] "Payne,Chanel"
```

```r
head(str_subset( Sal$Name, "Leonard.?S"))
```

```
## [1] "Payne,Leonard S"      "Szumlanski,Leonard S"
```

```r
head(str_subset( Sal$Name, "Spence.*C.*"))
```

```
## [1] "Greene,Spencer C"    "Spencer,Charles A"   "Spencer
## [4] "Spencer,Clarence W"  "Spencer,Michael C"
```

Replace

Let's say we wanted to sort the data set by Annual Salary:

```
class(Sal$AnnualSalary)
```

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

```
sort(c("1", "2", "10")) #  not sort correctly (order simply
```

```
## [1] "1"  "10" "2"
```

```
order(c("1", "2", "10"))
```

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

Replace

So we must change the annual pay into a numeric:

```
head(Sal$AnnualSalary, 4)
```

```
## [1] "$11310.00" "$53428.00" "$68300.00" "$62000.00"
```

```
head(as.numeric(Sal$AnnualSalary), 4)
```

```
## Warning in head(as.numeric(Sal$AnnualSalary), 4): NAs in
## coercion
```

```
## [1] NA NA NA NA
```

R didn't like the $ so it thought turned them all to `NA`.

`sub()` and `gsub()` can do the replacing part in base R.

Replacing and subbing

Now we can replace the $ with nothing (used `fixed=TRUE` because $ means ending):

```
Sal$AnnualSalary <- as.numeric(gsub(pattern = "$", replacen
                              Sal$AnnualSalary, fixed=TRUE)
Sal <- Sal[order(Sal$AnnualSalary, decreasing=TRUE), ]
Sal[1:5, c("Name", "AnnualSalary", "JobTitle")]
```

```
##                    Name AnnualSalary            JobTit
## 1222  Bernstein,Gregg L       238772      STATE'S ATTORN
## 3175   Charles,Ronnie E       200000   EXECUTIVE LEVEL I
## 985     Batts,Anthony W       193800   EXECUTIVE LEVEL I
## 1343      Black,Harry E       190000   EXECUTIVE LEVEL I
## 16352     Swift,Michael       187200 CONTRACT SERV SPEC
```

Replacing and subbing: stringr

We can do the same thing (with 2 piping operations!) in dplyr

```
dplyr_sal = Sal
dplyr_sal = dplyr_sal %>% mutate(
  AnnualSalary = AnnualSalary %>%
    str_replace(
      fixed("$"),
      "") %>%
    as.numeric) %>%
  arrange(desc(AnnualSalary))
check_Sal = Sal
rownames(check_Sal) = NULL
all.equal(check_Sal, dplyr_sal)
```

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

Part 2

Agenda

- 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 Lecture 2

```
survey.messy <- read.csv("survey_messy.csv", header=TRUE)
survey.messy$TVhours
```

```
##  [1] 4                 ~5                2                 5or so
##  [6] 15                3                 2ish              0
## [11] none              7                 3                 6.5
## [16] 0                 gjkhgs            3                 0
## [21] 3                 4                 10                2.5 (a mo
## [26] 6                 zero              0                 2
## [31] 2                 4                 4                 0
## [36] 0                 2
## 18 Levels: ~5 0 10 15 2 2.5 (a movie) 2ish 3 4 5 (netfli
```

What's happening?

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

```
## 'data.frame':    37 obs. of  6 variables:
##  $ Program        : Factor w/ 3 levels "MISM","Other",..
##  $ PriorExp       : Factor w/ 3 levels "Extensive experi
##  $ Rexperience    : Factor w/ 3 levels "Basic competence
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windo
##  $ TVhours        : Factor w/ 18 levels "~5","0","10",..
##  $ Editor         : Factor w/ 5 levels "Google Docs",..:
```

- 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] 4             ~5            2             5or so
##  [6] 15            3             2ish          0
## [11] none          7             3             6.5
## [16] 0             gjkhgs        3             0
## [21] 3             4             10            2.5 (a mo
## [26] 6             zero          0             2
## [31] 2             4             4             0
## [36] 0             2
## 18 Levels: ~5 0 10 15 2 2.5 (a movie) 2ish 3 4 5 (netfl
```

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

```
##  [1]  9  1  5 11  8  4  8  7  2  5 17 14  8 13  2  2 16
```

That didn't work...

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

```
##  [1] 4              ~5             2              5or so
##  [6] 15             3              2ish           0
## [11] none           7              3              6.5
## [16] 0              gjkhgs         3              0
## [21] 3              4              10             2.5 (a mo
## [26] 6              zero           0              2
## [31] 2              4              4              0
## [36] 0              2
## 18 Levels: ~5 0 10 15 2 2.5 (a movie) 2ish 3 4 5 (netfli
```

```
##  [1]  9  1  5 11  8  4  8  7  2  5 17 14  8 13  2  2 16
## [24]  6 15 12 18  2  5  9  5  9  9  2 10  2  5
```

- This just converted all the values into the integer-coded levels of the factor

Something that does work

- Consider the following simple example

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

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

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

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

```r
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

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

```
##  [1] "4"             "~5"            "2"             "5c
##  [5] "3"             "15"            "3"             "2i
##  [9] "0"             "2"             "none"          "7'
## [13] "3"             "6.5"           "0"             "0'
## [17] "gjkhgs"        "3"             "0"             "4'
## [21] "3"             "4"             "10"            "2.
## [25] "8"             "6"             "zero"          "0'
## [29] "2"             "4"             "2"             "4'
## [33] "4"             "0"             "5 (netflix)"   "0'
## [37] "2"
```

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

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

```
##  [1]  4.0   NA  2.0   NA  3.0 15.0  3.0   NA  0.0  2.0
```

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] "4"               "~5"              "2"               "5c
##  [5] "3"               "15"              "3"               "2:
##  [9] "0"               "2"               "none"            "7'
## [13] "3"               "6.5"             "0"               "0'
## [17] "gjkhgs"          "3"               "0"               "4'
## [21] "3"               "4"               "10"              "2.
## [25] "8"               "6"               "zero"            "0'
## [29] "2"               "4"               "2"               "4'
## [33] "4"               "0"               "5 (netflix)"     "0'
## [37] "2"
```

Deleting non-numeric (or .) characters

```r
tv.hours.strings
```

```
##  [1] "4"                "~5"               "2"                "5c
##  [5] "3"                "15"               "3"                "2:
##  [9] "0"                "2"                "none"             "7"
## [13] "3"                "6.5"              "0"                "0"
## [17] "gjkhgs"           "3"                "0"                "4"
## [21] "3"                "4"                "10"               "2.
## [25] "8"                "6"                "zero"             "0"
## [29] "2"                "4"                "2"                "4"
## [33] "4"                "0"                "5 (netflix)"      "0"
## [37] "2"
```

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

```
##  [1] "4"   "5"   "2"   "5"   "3"   "15"  "3"   "2"   "0"
## [12] "7"   "3"   "6.5" "0"   "0"   ""    "3"   "0"   "4"
```

The final product

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

```
##  [1] 4          ~5         2          5or so
##  [6] 15         3          2ish       0
## [11] none       7          3          6.5
## [16] 0          gjkhgs     3          0
## [21] 3          4          10         2.5 (a mo
## [26] 6          zero       0          2
## 18 Levels: ~5 0 10 15 2 2.5 (a movie) 2ish 3 4 5 (netfli
```

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

```
##  [1]  4.0  5.0  2.0  5.0  3.0 15.0  3.0  2.0  0.0  2.0
## [15]  0.0  0.0   NA  3.0  0.0  4.0  3.0  4.0 10.0  2.5
## [29]  2.0  4.0  2.0  4.0  4.0  0.0  5.0  0.0  2.0
```

- As a last step, we should go through and figure out if any of

Rebuilding our data

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

```
## 'data.frame':    37 obs. of  6 variables:
##  $ Program        : Factor w/ 3 levels "MISM","Other",..
##  $ PriorExp       : Factor w/ 3 levels "Extensive experi
##  $ Rexperience    : Factor w/ 3 levels "Basic competence
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windo
##  $ TVhours        : num  4 5 2 5 3 15 3 2 0 2 ...
##  $ Editor         : Factor w/ 5 levels "Google Docs",..:
```

- **Success!**

A different approach

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

```
survey.messy <- read.csv("survey_messy.csv", header=TRUE, s
str(survey.messy)
```

```
## 'data.frame':    37 obs. of  6 variables:
##  $ Program        : chr  "PPM" "PPM" "Other" "PPM" ...
##  $ PriorExp       : chr  "Never programmed before" "Some
##  $ Rexperience    : chr  "Never used" "Basic competence"
##  $ OperatingSystem: chr  "Windows" "Mac OS X" "Mac OS X"
##  $ TVhours        : chr  "4" "~5" "2" "5or so" ...
##  $ Editor         : chr  "Microsoft Word" "Microsoft Wor
```

- 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.]", "'
str(survey)
```

```
## 'data.frame':    37 obs. of  6 variables:
##  $ Program        : chr  "PPM" "PPM" "Other" "PPM" ...
##  $ PriorExp       : chr  "Never programmed before" "Some
##  $ Rexperience    : chr  "Never used" "Basic competence'
##  $ OperatingSystem: chr  "Windows" "Mac OS X" "Mac OS X'
##  $ TVhours        : num  4 5 2 5 3 15 3 2 0 2 ...
##  $ Editor         : chr  "Microsoft Word" "Microsoft Wor
```

What about all those other character variables?

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

```
## 
##  MISM Other   PPM 
##    13     7    17 
```

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

```
## 
##  MISM Other   PPM 
##    13     7    17 
```

- 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 Operatin
##             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
```

Here's the outcome

```r
str(survey)
```

```
## 'data.frame':    37 obs. of  6 variables:
##  $ Program        : Factor w/ 3 levels "MISM","Other",..
##  $ PriorExp       : Factor w/ 3 levels "Extensive experi
##  $ Rexperience    : Factor w/ 3 levels "Basic competence
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windo
##  $ TVhours        : num  4 5 2 5 3 15 3 2 0 2 ...
##  $ Editor         : Factor w/ 5 levels "Google Docs",..:
```

- **Success!**

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", "Dia
summary(life.support)
```

```
##    dialysis    Dialysis    dyalysis        nnone        n
##           3           1           1            1
## ventilation Ventilation
##           1           1
```

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

```
##    dialysis    Dialysis    dyalysis        nnone        n
##           3           1           1            1
```

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)
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 s
head(fake.data,2) # print first two rows
```

```
##            [,1]       [,2]       [,3]       [,4]
## [1,] -1.6446803 -2.1532042  0.5611550 -0.6246019 -0.8565
## [2,] -0.8199759 -0.5727423 -0.4925805  0.9939846  0.8331
```

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

```
## [1] -1.4895621 -4.4912187 -0.8146502 -7.2401811 -5.81867
```

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

```
## [1] -1.4895621 -4.4912187 -0.8146502 -7.2401811 -5.81867
```

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`	Simplified

Example: apply()

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

```
## [1] -0.014895621 -0.044912187 -0.008146502 -0.072401811
```

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

```
## [1] -0.014895621 -0.044912187 -0.008146502 -0.072401811
```

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

```
## [1] 0.50 0.53 0.48 0.46 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
```

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

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

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

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

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

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

```
## 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

## The following object is masked from 'package:dplyr':
##
##     select
```

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

Example: tapply()

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

```
tapply(Cars93[["Horsepower"]], INDEX = Cars93[c("Origin", '
```

```
##            Type
## Origin      Compact    Large  Midsize    Small   Sporty 
##   USA      117.4286 179.4545 153.5000 89.42857 166.5000 1
##   non-USA 141.5556       NA 189.4167 91.78571 151.6667 1
```

- 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 DriveTr
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", "Dr
```

```
##          DriveTrain
## Origin    4WD Front Rear
##   USA       5    34    9
```

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

```r
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")) #
```

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

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

```
##            Compact    Large  Midsize    Small   Sporty
## USA       117.4286 179.4545 153.5000 89.42857 166.5000 15
## non-USA   141.5556       NA 189.4167 91.78571 151.6667 13
```

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