Lecture 8: Data Management Techniques, Regular Expressions, and Dates and Times

STAT 385 - James Balamuta

June 27, 2016

On the Agenda

1. Data Management Techniques
   - Versioning
   - Reading and Writing
   - Overall tips
2. Regular Expressions
   - How to extract information from strings?
3. Dates and Times
   - `POSIXct` object

Data Management Techniques

- When talking about data management techniques, we're going to aim to make things reproducible as always.
- Thus, everything should be done within a script.
- The script should also be within the `data-raw` directory.

Ideal Data Management Setup

Consider the following directory structure:

- Anything related to cleaning data is within the `data-raw` directory.

OS Independent Load

Often times, we collaborate with a colleague through remote storage options such as Dropbox and BoxSync. However, if your colleague is running a Mac and you are running Windows, how can you keep the same source files?

```
# Example of an independent OS environment
os_name = Sys.info()[['sysname']]
if(os_name =="Windows"){
  fp = file.path("F:/BoxSync/stat385/lectures")
}else if(os_name =="Darwin"){ #OS X
  fp = file.path("~/BoxSync/stat385/lectures")
}else{ #Linux
  stop("I'm a penguin.")
}

script_path = file.path(fp, "lec8")
```

Package Dependencies

Similarly, when sharing code, how can you make sure that the receiving party is able to run it, without having to look through the code to figure out dependencies (i.e. `library()` or `require()`)?

```r
# Any package that is required by the script
# below is given here
inst_pkgs = load_pkgs =  c("MASS","faraway","Hmsic",
                           "randomForest","rpart")

# Check to see if the packages are already installed
inst_pkgs = inst_pkgs[!(inst_pkgs %in%
                          installed.packages()[,"Package"])

# Installs any missing package
if(length(inst_pkgs)) install.packages(inst_pkgs)

# Dynamically load required packages
pkgs_loaded = lapply(load_pkgs, require, character.only=T)
```

Reading Data - R Core

There are many different data reading packages now available. The defaults are:

- **base R:** Text Files
  - `read.table()`: Read file in table format.
  - `read.csv()` (US) /`read.csv2()` (Euro): Reads csv file depending on decimal (. or ,) wrapper to `read.table()`
  - `read.delim()` (US) / `read.delim2()` (Euro): Reads deliminated file depending on decimal (. or ,) wrapper to `read.table()`
- **foreign:** Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, Octave

Reading Data - Third Party

Last year, Hadley took it upon himself to write a lot of different file readers. In turn, these readers should be preferred as they are: 1. Faster and 2. Reliable

- **readr:** Text (.csv)
  - `read.file()`: Handles zipped files and `.txt`
- **readxl:** Excel
  - `read_excel()`: Reads in the first sheet (can be specified to others by name)
- **haven:** SAS, SPSS, and Stata
  - `read_sas()`: Handles SAS `.b7dat` and `.b7cat`
  - `read_spss()`: Handles SPSS `.por` and `.sva`
  - `read_stata()`/`read_dta()`: Handles Stata `.dta`

Sample Data Clean

- For the next section, we will be focusing on the cleaning and formatting of data.
- Many of the ideas presented next were discussed by my good friend **Michael Quinn**.
- Michael obtained a Masters in Statistics from UIUC while working as a MAGNET Intern at the State Farm RDC and now works on Google's Ad Team!

Load Multiple Data Sets

- Rarely is data ever in one source. Sometimes, data is found in a combination of **.csv** files with different filenames.

```
# Obtain a list of all files within active directory
# with extension .csv
filenames = list.files(pattern="*.csv")

dsAll = data.frame()
for (i in 1:length(filenames)) {
  # Assign each file to its filename
  assign(filenames[i],
         read.csv(filenames[i], stringsAsFactors=FALSE))
  # Note: The stringsAsFactors = FALSE condition
  # prevents r from building levels into each variable.

  # Quick bind (bad implementation)
  dsAll = rbind(dsAll, filenames[i])
}
```

Make a Rejection List

- When cleaning data, you will want to exclude an observation given a variable's specific value.
- To handle such values, we create a '**Rejection List**.' The list will specify observations that should be removed from the data.
  - Here is an example of a rejection list called `reject_list.txt`:

```
1002 w springfield ave
315 s state st
508 e soughton st
....
```

Updating an Observation

- To update an observation, we look for a unique key to the observation.
  - In this case, it would be the house address. In other cases, it may be the Subject ID that is assigned.
- For most of the cases, we will want to only update a part of an observation.

```
#Find the row value
k = which(houses$St_Address == tolower("3909 Aberdeen Dr"))
#Update row value traits
houses$Bedrooms[k] = "3 beds"
houses$Bathrooms[k] = "2 baths"
houses$lot[k] = "3040 sqft"
houses$lastsoldifavailable[k] = "May 2011 for $135,000"
```

Efficient Cleaning

- To clean efficiently means to vectorize.
- Here are a few vectorizations of cleaning approaches:

```
# Removes any of the St_Addresses in the rejection list
houses = houses[!(houses$St_Address %in% reject),]
# Removes any duplicates
houses = unique(houses)
# Drop column range starting at heattype to floorcover
# heattype, zillowdays, cooling, parking, basement,
# fireplace, floorcover.
start_loc = match("heattype",names(houses))
end_loc = match("floorcover",names(houses))
houses = houses[,-(start_loc:end_loc)]
# Removes variable headers in dataset
houses = houses[-which(houses$lastsoldifavailable
                       =="lastsoldifavailable"),]
```

Concept of Tidy Data

> *Tidy datasets are all alike but every messy dataset is messy in its own way*
> — *Hadley Wickham (JSS Tidy data)*

In tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Data Shape

Consider the following data set:

```
experiment = read.table(header=TRUE, text='
 subject sex control a b
       S1   F     4.2  4.1  2.2
       S2   M     5.9  7.2  6.8
       S3   M     9.1  9.8  10.2
       S5   F     2.1  23.5  5.2
')

experiment
```

```
##   subject sex control    a    b
## 1      S1   F     4.2  4.1  2.2
## 2      S2   M     5.9  7.2  6.8
## 3      S3   M     9.1  9.8 10.2
## 4      S5   F     2.1 23.5  5.2
```

Wide Data

```r
experiment
```

```
##   subject sex control    a    b
## 1      S1   F     4.2  4.1  2.2
## 2      S2   M     5.9  7.2  6.8
## 3      S3   M     9.1  9.8 10.2
## 4      S5   F     2.1 23.5  5.2
```

- In its current form, the data is considered to be **wide**.
- **Wide Data** has repeated responses or treatments of a subject in a single row with each response in its own column along with its properties.

Long Data

```
##    subject sex condition measurement
## 1       S1   F   control         4.2
## 2       S2   M   control         5.9
## 3       S3   M   control         9.1
## 4       S5   F   control         2.1
## 5       S1   F         a         4.1
## 6       S2   M         a         7.2
## 7       S3   M         a         9.8
## 8       S5   F         a        23.5
## 9       S1   F         b         2.2
## 10      S2   M         b         6.8
## 11      S3   M         b        10.2
## 12      S5   F         b         5.2
```

- With a little modification, the data is considered to be **long**.
- **Long Data** has each row as one response per subject and any variables for the subject that do not change over time or treatment will have the same value in all the rows.

Long Data to Wide

- Use spread to move to a wide format

```
library(tidyr)
(data_wide =  spread(data_long, condition, measurement))
```

```
##   subject sex    a    b control
## 1      S1   F  4.1  2.2     4.2
## 2      S2   M  7.2  6.8     5.9
## 3      S3   M  9.8 10.2     9.1
## 4      S5   F 23.5  5.2     2.1
```

Wide Data to Long

- Use gather to move to a long format

```
library(tidyr)
(data_long = gather(experiment,
                    condition, measurement, control:b))
```

```
##    subject sex condition measurement
## 1       S1   F   control         4.2
## 2       S2   M   control         5.9
## 3       S3   M   control         9.1
## 4       S5   F   control         2.1
## 5       S1   F         a         4.1
## 6       S2   M         a         7.2
## 7       S3   M         a         9.8
## 8       S5   F         a        23.5
## 9       S1   F         b         2.2
## 10      S2   M         b         6.8
## 11      S3   M         b        10.2
## 12      S5   F         b         5.2
```

Moving along..

- Any questions on **Data Management Techniques**?
- Moving along to **Regular Expressions**. . .

Regular Expressions

- **Regular Expression** or **regex** is a sequence of characters that defines a search pattern for a collection of strings.
- The idea sprouted from the notion of a **regular language** that was brought into existence by Kleene's theorem written by Stephen Cole Kleene.
- **Regex** is primarily used to:
    1. search for patterns and,
    2. replace patterns
- For it to function, programmers have adopted a set of grammatical statements to build patterns for strings.
- The grammar is available in just about every single programming language.

Regular Expression Usage Cases

- Validate data entry fields
  - dates, e-mail address, credit card numbers
- Filter Text Easily
  - key words or phrases in reviews, web server logs, reading config files
- Restructuring Text
  - mass change variable names, switching line endings
- Counting Occurrences
  - number of words, errors, or warnings

Relevant XKCD Comic

Figure 1: XKCD 208

Words of Wisdom for Regular Expressions

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*
> — *Jamie Zawinski in alt.religion.emacs*

Note: Avoid using *regex* when parsers exist for tree structures (e.g. *html/dom*) to prevent edge cases from not being picked up!

Regular Expressions in R

Function	Description
grep	Returns a vector of the **indices** or **values** that match
grepl	Returns a **logical** vector indicating matches (`TRUE`)
regexpr	Returns the starting position of the **first** match
gregexpr	Returns the starting position of **all** matches
sub	Perform replacement of the **first** match
gsub	Perform replacement of the **all** matches

Regex Example - Finding IL (Concatenation)

- Consider the need to *filter* terms by whether or not they include **IL** (short for Illinois).

```
locs = c('Chicago, IL', 'San Francisco, CA', 'Springfield,
         'Detroit, MI', 'Urbana, IL', 'Tampa, FL')

grep('IL', locs)                 # Obtain the Indices
```

```
## [1] 1 3 5
```

```
grep('IL', locs, value = TRUE) # Obtain the Names
```

```
## [1] "Chicago, IL"     "Springfield, IL" "Urbana, IL"
```

```
grepl('IL', locs)                # Obtain logical response
```

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

Regex Example - Finding IL (Concatenation)

- Find the locations within the string
  - If missing, start and end location return -1

```
regexpr('IL', locs)  # Obtain the first instance in a word
```

```
## [1] 10 -1 14 -1  9 -1
## attr(,"match.length")
## [1]  2 -1  2 -1  2 -1
## attr(,"useBytes")
## [1] TRUE
```

```
gregexpr('IL', locs) # Obtain all instances in a word
```

```
## [[1]]
## [1] 10
## attr(,"match.length")
## [1] 2
## attr(,"useBytes")
## [1] TRUE
```

Regex Example - Removing a (Concatenation)

- Remove instances of the letter a from the text

```
sub('a', "", locs)  # Remove first instance
```

```
## [1] "Chicgo, IL"       "Sn Francisco, CA" "Springfield,
## [4] "Detroit, MI"      "Urbna, IL"        "Tmpa, FL"
```

```
gsub('a', "", locs) # Remove all instances
```

```
## [1] "Chicgo, IL"       "Sn Frncisco, CA" "Springfield, II
## [5] "Urbn, IL"         "Tmp, FL"
```

Regex Lexicon - Round 1

- Regex foundamentals operations:

Operation	Explanation	Symbol
Concatentation	Exact String	word
Wildcard	Any character	.
Union	Either character	|
Closure	Match preceding character 0 or more	*
Parentheses	Matches a pattern group	()
One or More	Match preceding character 1 or more	+

Regex Lexicon - Round 1 Examples

Operation	Symbol	Example	Match	Failure
Concatentation	word	`james`	james	da yae
Wildcard	.	`j.m.s`	james/jomas	anmes
Union	|	`jb	am`	jjb / jam	toad
Closure	*	`cat*`	catcat / ca	ma / at
Parentheses	()	`(og)`	dog / blog	dgs
One or More	+	`sh+`	shoe / ship	hip / s

Regex Lexicon - Round 2

Operation	Explanation	Symbol
Class Matches	Match within specific classes	`[]`
Range	Match values within a range	`[-]`
Negations	Match any character not within	`[^]`
Once	Matche preceeding element once	`?`
Exact Amount	Match exactly m occurences	`{m}`
At Least	Match at least m occurences	`{m,}`
Between	Match $m \leq x \leq n$ occurences	`{m,n}`
Beginning	Start at the **beginning** of the string	`^`
End	Start at the **end** of the string	`$`

Regex Lexicon - Round 2 Examples

Operation	Symbol	Example	Match	Failure
Class Matches	[]	[abc]	toad / book	room / desk
Range	[-]	[a-zA-Z]	Funky/ word	1234 / #$@
Negations	[^]	[^aeiou]	gst / wd	hi / orange
Once	?	a?	hat / car	no / yes
Exact Amount	{m}	[0-9]{3}	123-423 / 921	12-33 / 9
At Least	{m,}	[m]{2,}	yumm / mommy	mom / yum
Between	{m,n}	[o]{1,2}	zoo / food	dad / phil
Beginning	^	^hi	hiya / hillary	yahi
End	$	s$	james / dogs	sam / sosa

Helpful Regex Tools

- regexplanet.com: Regex recipe database
- regex101.com: Great Regex tester

Extracting and Formatting

- Data may sometimes not come formatted to the necessary requirements.
- One variable might need to be split up among multiple variables.
- This is an ideal case for using regular expressions (`?regex`).

Extracting and Formatting

- In the housing data, the house address is stored as:

"[#### Street Name] , [City], [State] [Zipcode]"

- We would like to split this data into three new variables:
  - **St_Address**, Zipcode, and City.
- Note, we will want to use the lower case alphabet to prevent capitalization mismatches within alphabetical fields. (e.g. "harbor estates ln" $\neq$ "Harbor Estates Ln")

Extracting and Formatting

House_Address: [#### Street Name] , [City], [State] [Zipcode]

```
# Create a temporary variable with strings in lower case
address = tolower(as.character(houses$House_Address))
# Regex: The . means any character
# Regex: The * uses greedy evaluation [repeats]
street = gsub(" ,.*", "", address)
houses$St_Address = street
# Replace city name with 0 or 1.
city = gsub(".*, champaign, .*", "0", address)
city = gsub(".*, urbana, .*", "1", city)
houses$City = as.numeric(city) # Make numeric
# Extract the zip code
zip = gsub(".* , .*, il ", "", address)
houses$ZipCode = as.numeric(zip)
# Delete old House_Address
houses$House_Address = NULL
```

And that's all for regex!

- Coming up next... **Dates and Times**
- Any questions for **Regular Expressions**?

Date and Time Formats

> *"The only reason for time is so that everything doesn't happen at once."*
> — *Albert Einstein*

- *R* has the ability to interface with time information.
- The interface, as we will see, may not be the best but it is highly versatile.
- This is important in a world that is going more and more global.

Date and Time Formats

```r
Sys.Date()                  # Returns only Date
```

```
## [1] "2016-06-28"
```

```r
Sys.time()                  # Returns Date + Time
```

```
## [1] "2016-06-28 14:10:58 CDT"
```

```r
as.numeric(Sys.time()) # Seconds from UNIX Epoch
```

```
## [1] 1467141058
```

Date and Time Formats - Failure of `characters`

- Frequently, dates and times will be given as `characters` within a `data.frame`
- Having dates as `characters` impedes ones ability to be able to use the time information in an analysis
  - For example: How long did it take for the help desk call to be completed?

```
# Bad Time Differencing:

time1 = "2016-06-28 10:25:44 CDT" # UNIX Time Stamp
time2 = "2016-06-28 15:25:44 CDT" # UNIX Time Stamp
time2 - time1
## Error in time2 - time1 : non-numeric argument to binary
```

Date and Time Formats - Time Operations

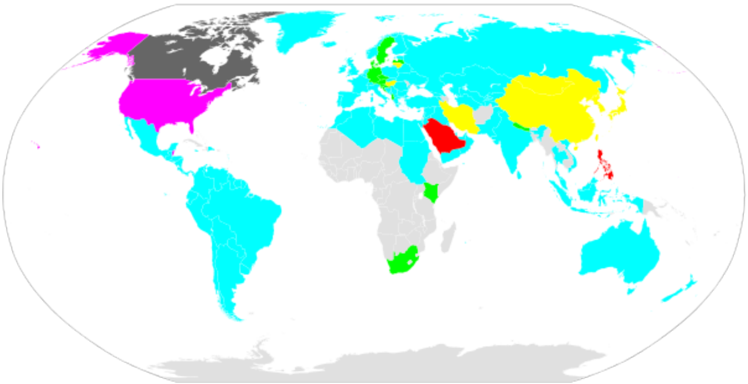
- Performing time operations requires that both dates are given as `POSIXct` object in *R*.

```
time1 = as.POSIXct("2016-06-28 10:25:44")
time2 = as.POSIXct("2016-06-28 15:25:44") # +5 Hours
time2 - time1
```

```
## Time difference of 5 hours
```

Note: Default format for `POSIXct` is `%Y-%m-%d %H:%M:%S`

The Date Format Around the World



Color	Date Format	Main Region	Population (Millions)
Cyan	DD/MM/YYYY	Australia, Russia	3295
Yellow	YYYY/MM/DD	China, Korea, Iran	1660
Magenta	MM/DD/YYYY	United States	320

Formats for Working with Dates

Format	Description	Example
%a	Abbreviated weekday name in the current locale	Tue
%A	Full weekday name in the current locale	Tuesday
%b	Abbreviated month name in the current locale	Jun
%B	Full month name in the current locale	June
%m	Month number (01-12)	06
%d	Day of the month as decimal number (01-31)	28
%e	Day of the month as decimal number (1–31)	28
%y	Year without century (00-99)	16
%Y	Year including century	2016

For more, see `?strptime`

Formating Non-Standard Dates

```r
(yyyy_mm_dd = as.POSIXct("2016-06-28",
                         format = "%Y-%m-%e"))
```

```
## [1] "2016-06-28 CDT"
```

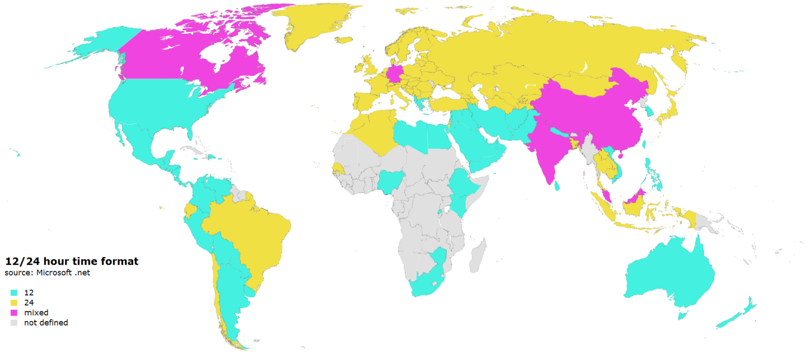
```r
(dd_mm_yy = as.POSIXct("28/06/16",
                       format = "%e/%m/%y"))
```

```
## [1] "2016-06-28 CDT"
```

```r
(mon_dd_yyyy = as.POSIXct("Jun 28, 2016",
                          format = "%b %e, %Y"))
```

```
## [1] "2016-06-28 CDT"
```

Time Format Used Around the World



Formats for Working with Times

Format	Description	Example
%S	Second as integer (00–61)	05
%M	Minute as decimal number (00–59)	41
%H	Hours as decimal number (00–23)	11
%I	Hours as decimal number (01–12)	11
%p	AM/PM indicator in the locale	AM
%z	Signed offset in hours and minutes from UTC	-0500
%Z	Time zone abbreviation as a character string	CDT

For more, see `?strptime`

Formating Non-Standard Times

```
(h_m = as.POSIXct("11:38",
                         format = "%H:%M"))
```

```
## [1] "2016-06-28 11:38:00 CDT"
```

```
(h_am = as.POSIXct("11 AM",
                         format = "%I %p"))
```

```
## [1] "2016-06-28 11:00:00 CDT"
```

```
(h_m_s_z = as.POSIXct("11:38:22", # Chop off the TZ
                         format = "%H:%M:%S",
                         tz = "America/New_York"))
```

```
## [1] "2016-06-28 11:38:22 EDT"
```

Time Zone Notes

- *R* makes use of time zones via `tz` parameter.
- The accepted values of `tz` depend on the location.
  - `CST` is given with `"CST6CDT"` or `"America/Chicago"`
- For supported locations and time zones use:
  - In R: `OlsonNames()`
  - Alternatively, try in R: `system("cat $R_HOME/share/zoneinfo/zone.tab")`
- These locations are given by Internet Assigned Numbers Authority (IANA)
  - List of tz database time zones (Wikipedia)
  - IANA TZ Data (2016e)

Specifics on POSIXct

- POSIXct: Stores time as seconds since UNIX epoch on 1970-01-01 00:00:00
  - Used exclusively in the Hadleyverse and world of UNIX.

```
# POSIXct output
(origin = as.POSIXct("1970-01-01 00:00:00",
                     format ="%Y-%m-%d %H:%M:%S",
                     tz = "UTC"))
```

```
## [1] "1970-01-01 UTC"
```

```
as.numeric(origin)     # At epoch
```

```
## [1] 0
```

```
as.numeric(Sys.time()) # Right now
```

```
## [1] 1467129382
```

The “other” time object: `POSIXlt`

- `POSIXlt`: Stores a list of day, month, year, hour, minute, second, and so on.
  - It is **slower** than `POSIXct` and has **zero support** in the Hadleyverse.
  - **Warning:** `POSIXlt` will be returned if you use `strptime()`
  - Always convert `POSIXlt` to `POSIXct` using `as.POSIXct()`!!!

```r
# POSIXlt output
posixlt = as.POSIXlt(Sys.time(),
                     format ="%Y-%m-%d %H:%M:%S",
                     tz = "America/Chicago")

# Convert to POISXct
posixct = as.POSIXct(posixlt)
posixct
```

```
## [1] "2016-06-28 10:57:54 CDT"
```

POSIXlt - List Values

```r
posixlt$sec    # Seconds 0-61
```

```
## [1] 54.42393
```

```r
posixlt$min    # Minutes 0-59
```

```
## [1] 57
```

```r
posixlt$hour  # Hour 0-23
```

```
## [1] 10
```

```r
posixlt$mday  # Day of the Month 1-31
```

```
## [1] 28
```

```r
posixlt$mon    # Months after the first of the year 0-11
```

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

`lubridate` - Dates Made Easy

- `lubridate` by Garret Grolemund, Hadley Wickham, and Gang is a great way to work with dates.
- Many built in helpers & easy parsers, e.g.

```
library(lubridate)
ymd("20160628")
```

```
## [1] "2016-06-28"
```

```
interval(mdy("06-28-2016"), dmy("29/06/2016"))
```

```
## [1] 2016-06-28 UTC--2016-06-29 UTC
```

- For more, please read the Lubridate vignette on the Lubridate CRAN Page

Summary

- To analyze time, you must have it in a `POSIXct` object
  - Avoid `POSIXlt` like the plague.
- Date and Time Stamps differ greatly around the world.