Data Science for Economists

Lecture 6: Misc Base R

Drew Van Kuiken
University of North Carolina | ECON 370

Table of contents

Introduction

Agenda

Today we will cover a bunch of miscellaneous topics that don't quite fit anywhere else.

I will also mention some functions and tips and tricks that I find useful.

Dates and Times

Dates and Time

In order to work with dates and times correctly, they need to be specified as a date or time.

You will likely work with dates more often than times, so that is where more attention will be placed.

Understanding exactly how dates work requires understanding a bit more about the advanced aspects of R; however, to get the basics does not require this.

The default format for dates in R is `YYYY-MM-DD.`

```
as.Date("2021-09-14")
```

```
## [1] "2021-09-14"
```

If you have a date not in the format listed above, you have to give the `as.Date()` function the "format" argument. See this link for different formats.

```
as.Date("09/14/2021",format="%m/%d/%Y")
```

```
## [1] "2021-09-14"
```

Lubridate

While we have been working in base `R` almost exclusively so far, I would highly recommend using the `lubridate` package.

Working with dates and times is much easier than in base `R`.

```r
library(lubridate)
```

```r
ymd("2021-09-14")
```

```
## [1] "2021-09-14"
```

```r
mdy("09/14/2021")
```

```
## [1] "2021-09-14"
```

```r
today()
```

```
## [1] "2024-09-15"
```

Lubridate (Cont.)

```r
year(today())
```

```
## [1] 2024
```

```r
month(today())
```

```
## [1] 9
```

```r
week(today())
```

```
## [1] 37
```

```r
ymd_hms("2017-01-31 20:11:59")
```

```
## [1] "2017-01-31 20:11:59 UTC"
```

```r
mdy_hm("01/31/2017 08:01")
```

```
## [1] "2017-01-31 08:01:00 UTC"
```

More About Dates and Times

There are a lot of little quirks when it comes to dates and times that usually you will not have to know. God bless you if you're ever working with time zones, spring forward, etc.

This is where the skill of Googling really comes in handy.

I would recommend also reading Chapter 16 in *R for Data Science* that is listed on the syllabus.

Reading in Files

Reading in Files

Because we've been working with simulated data or preloaded data, we have yet to have to read in data.

The most common function you will use is `read.csv()`.

- While you might want to read in Excel spreadsheets, I would highly recommend minimizing how much you use Excel.

To read in a file, you need to know where the file is located. You will need to set your "working directory."

```
getwd()
```

```
## [1] "/Users/drewvankuiken/Dropbox/econ370/local/lec6"
```

```
setwd("~/Dropbox/Econ370/local/lec6/")
getwd()
```

```
## [1] "/Users/drewvankuiken/Dropbox/econ370/local/lec6"
```

See Files in Directory

To see the names of the files in your current working directory, use the `list.files()` functions.

```
list.files()
```

```
## [1] "06-miscR.html" "06-miscR.pdf"  "06-miscR.Rmd"  "blp_data.csv"
## [5] "Lecture 7"     "libs"
```

Read in CSV

To read in the CSV listed above (about market share data and prices), use the function `read.csv()`:

```
list.files()
```

```
## [1] "06-miscR.html" "06-miscR.pdf"  "06-miscR.Rmd"  "blp_data.csv"
## [5] "Lecture 7"     "libs"
```

```
OTC_data = read.csv("blp_data.csv")
head(OTC_data)
```

```
##   mkt time firm prod    share  price      x       w
## 1   1    1    1    1 0.202769 1.5955 1.2640 -0.2258
## 2   1    1    2    2 0.032901 0.9492 0.8685 -0.8978
## 3   1    1    3    3 0.139623 1.3500 1.0181 -0.2054
## 4   1    2    1    1 0.229222 1.4541 1.2292 -0.5836
## 5   1    2    2    2 0.038348 1.4270 1.0411  0.1160
## 6   1    2    3    3 0.076592 1.8699 1.0695  0.6043
```

A Few Things on Reading in Data

CSV stands for "comma-separated values" i.e. the data values are separated by commas.

If you notice in the help documentation for read.csv, it assumes `sep=","`.

If this is not true, you might have to change the sep argument.

- e.g. Tab separated values are also common.

However, don't worry too much about this. While you need to understand it a little, we will be using `read_csv()` from the `tidyverse` and/or `fread()` from the `data.table` package, which are superior to read.csv.

Regular Expressions

Parsing Text

Parsing text to find patterns or extract data is hard.

There are a group of functions that can be used for this: the `grep()` functions.

`grep()` takes a pattern and character vector and returns a vector of indexes for which elements contained the pattern.

`grepl()` takes a pattern and character vector and returns a logical vector for if each element contains the pattern.

`sub()` takes a pattern, a replacement, and a character vector and returns the character vector with the first occurrence of the pattern replaced with the replacement.

`gsub()` is exactly like `sub()` except that it replaces all occurrences.

Parsing Text: Examples

```r
statecountry_names = c('Florida','Germany','Georgia','Geniva',
                'Istanbul','NewZealand','Australia')
grep("G",statecountry_names)
```

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

```r
grepl("G",statecountry_names)
```

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

```r
grepl("A",statecountry_names,ignore.case = F)
```

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

```r
grepl("A",statecountry_names,ignore.case = T)
```

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

Parsing Text: Examples

```
sub("G","A",statecountry_names)
```

```
## [1] "Florida"     "Aermany"     "Aeorgia"     "Aeniva"      "Istanbul"
## [6] "NewZealand" "Australia"
```

```
sub("G","A",statecountry_names,ignore.case = T)
```

```
## [1] "Florida"     "Aermany"     "Aeorgia"     "Aeniva"      "Istanbul"
## [6] "NewZealand" "Australia"
```

```
gsub("G","A",statecountry_names,ignore.case = T)
```

```
## [1] "Florida"     "Aermany"     "AeorAia"     "Aeniva"      "Istanbul"
## [6] "NewZealand" "Australia"
```

Regular Expressions

Sometimes there are patterns that you would like to detect in text that are more complicated than just a short character expression that you can program.

Or you'd like to do all combinations of something.

This is where regular expressions can be used.

Truthfully, we could spend an entire class just learning regular expressions.

As such, I am only going to briefly touch them and encourage you to look into them more on your own.

The plus side is they are pretty universal across languages. That is, whether you're using `R` or `Python`, regular expressions still work the same way!

Regular Expressions: Examples

Let's say we want to flag character strings that start with "The". The regular expression would be `^The`

```
test_char = c("The beginning.","The end.","The middle.","Other.")
grepl("^The",test_char)
```

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

Now, let's say we want to flag character strings that end with "end." The regular expression would be `end$`

```
grepl("end.$",test_char)
```

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

Find all vowels

```
grepl("[aeiou]",letters)
```

```
##  [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [13] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [25] FALSE FALSE
```

Regular Expressions: More Examples

Let's say we have a couple strings to match:

```r
test_char = c("The beginning.","The end.","The middle.","Other.")
grepl("beginning|end",test_char)
```

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

We can also grab numbers from a string:

```r
grepl("[0-9]",c(1:5,"I had 2 coffees today","Hello World!"))
```

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

Regular Expressions: More

Validating that a user supplied an email to a website form is famously a difficult problem. Here's one regex that a user on StackOverflow put forth:

```
pattern <- (?:[a-z0-9!#$%&'+/=?^_`{|}~-]+(?:.[a-z0-9!#$%&'+/=?^_`{|}~-]+)|'(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\[\x01-\x09\x0b\x0c\x0e-\x7f])')@(?:(?:a-z0-9?.)+a-z0-9?|[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])).){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\[\x01-\x09\x0b\x0c\x0e-\x7f])+)])
```

Regular Expressions: More

Regular expressions are very powerful for processing text data. Processing text data is a skill set in-and-of itself.

At one point, I knew regular expressions (regex) decently well. Now, I have LLMs write my regexs if they're at all complicated.

Lots of learning when it comes to programming is simply being aware of something so that you can look into it more when you actually need to use it.

If you want to learn more about regular expressions, see the following cheat-sheet.

Functions That I Use

Misc Functions

- `rep()`: repeats a vector N times
  - Used a lot for preallocating vectors
- `unique()`: takes the unique values of a vector
- `sort()`: sorts a vector
- `order()`: like sort, but gives the IDs rather than a sorted vector
- `sum()`: sums up vectors
- `seq()`: creates sequences
- `cbind()`: combine matrices or data.frames by columns
- `rbind()`: combine matrices or data.frames by rows
- `ncol()`: returns number of columns of data.frame or matrix
- `nrow()`: returns number of rows of data.frame or matrix
- `dim()`: returns dimensions of an array-like object
- `substr()`: returns a string subsetted by the indexes supplied
- `strsplit()`: splits a string by a pattern

Examples

```
rep(0,5)
```

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

```
rep(1:10,2)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10
```

```
rep(1:10,each=2)
```

```
##  [1]  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10
```

```
letter_samp = sample(letters,12,replace=T)
letter_samp
```

```
##  [1] "o" "s" "n" "c" "j" "r" "v" "k" "e" "t" "n" "v"
```

```
unique(letter_samp)
```

```
##  [1] "o" "s" "n" "c" "j" "r" "v" "k" "e" "t"
```

Examples (Cont.)

```
norm_draws = rnorm(10)
norm_draws
```

```
##  [1]  0.1830826  1.2805549 -1.7272706  1.6901844  0.5038124  2.5283366
##  [7]  0.5490967  0.2382129 -1.0488931  1.2947633
```

```
sort(norm_draws)
```

```
##  [1] -1.7272706 -1.0488931  0.1830826  0.2382129  0.5038124  0.5490967
##  [7]  1.2805549  1.2947633  1.6901844  2.5283366
```

```
order(norm_draws)
```

```
##  [1]  3  9  1  8  5  7  2 10  4  6
```

```
sum(norm_draws)
```

```
## [1] 5.49188
```

```
seq(1,2.4,by=0.1)
```

```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4
```

Examples (Cont.)

```r
rand_mat1 = matrix(rnorm(2*3),nrow=3)
rand_mat2 = matrix(rnorm(1*3),nrow=3)
rand_mat1
```

```
##             [,1]       [,2]
## [1,]  0.82553984 -0.7335032
## [2,] -0.05568601 -0.2158654
## [3,] -0.78438222 -0.3349128
```

```r
rand_mat2
```

```
##             [,1]
## [1,] -1.08569914
## [2,] -0.08542326
## [3,]  1.07061054
```

```r
cbind(rand_mat1,rand_mat2)
```

```
##             [,1]       [,2]        [,3]
## [1,]  0.82553984 -0.7335032 -1.08569914
## [2,] -0.05568601 -0.2158654 -0.08542326
## [3,] -0.78438222 -0.3349128  1.07061054
```

Examples (Cont.)

```
rand_mat1 = matrix(rand_mat1,nrow=2)
rand_mat2 = matrix(rand_mat2,nrow=1)
rand_mat1
```

```
##             [,1]       [,2]       [,3]
## [1,]  0.82553984 -0.7843822 -0.2158654
## [2,] -0.05568601 -0.7335032 -0.3349128
```

```
rand_mat2
```

```
##           [,1]        [,2]     [,3]
## [1,] -1.085699 -0.08542326 1.070611
```

```
rbind(rand_mat1,rand_mat2)
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.82553984 -0.78438222 -0.2158654
## [2,] -0.05568601 -0.73350322 -0.3349128
## [3,] -1.08569914 -0.08542326  1.0706105
```

Examples (Cont.)

```r
ncol(rand_mat2)
```

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

```r
nrow(rand_mat2)
```

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

```r
dim(rand_mat2)
```

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

```r
my_string = "Hello, This is Econ 370, and my name is Drew"
substr(my_string,1,10)
```

```
## [1] "Hello, Thi"
```

```r
strsplit(my_string,",")
```

```
## [[1]]
## [1] "Hello"                " This is Econ 370"    " and my name is Drew"
```

Next lecture: Introduction to Regression