Week 3 R tutorial supplement

Statistics and statistical programming
Northwestern University
MTS 525

Aaron Shaw and Nick Vincent

September 29, 2020

Contents

This supplementary tutorial goes through some additional examples and concepts that came up in our class session today (September 29, 2020). Nick and Aaron have tried to elaborate some examples that illustrate our responses.

Note that we don't provide any new examples related to the `apply()` family of functions (`lapply()`, `sapply()`, `tapply()`) as the upcoming R tutorials will develop some additional material on those.

If any of these examples are unhelpful and/or more confusing, we strongly encourage you to reach out to us for additional clarification and/or just disregard for the time being. As they say, "your mileage may vary."

Using code chunks to break down a complicated function call

One topic that came up today related into the use of chunks and breaking up functions to create more readable code.

Let's use openintro county data for this example.

```
library(openintro)
```

```
## Loading required package: airports
```

```
## Loading required package: cherryblossom
```

```
## Loading required package: usdata
```

```
data(county)
```

Let's look at three columns: `pop2000`, `pop2010`, and `pop2017`.

For some reason, let's imagine that a homework assignment has asked us to compute the mean population in the years 2000, 2010, and 2017, take the median of the three means, and then take the log of that median. Why not?

Let's start with the mean of one column...

```
mean(county$pop2000)
```

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

Whoops, okay, let's use that `na.rm` argument again.

```
mean(county$pop2000, na.rm=TRUE)
```

```
## [1] 89649.99
```

Ok, now we can grab just the three columns we want. Let's select a subset using brackets and variable names.

```
three_columns <- c('pop2000', 'pop2010', 'pop2017')
three_columns
```

```
## [1] "pop2000" "pop2010" "pop2017"
```

Here's a very concise one-line command that solves our hypothetical homework problem:

```
log(median(sapply(county[three_columns],mean, na.rm=TRUE)))
```

```
## [1] 11.49539
```

That works fine, but it can be quite confusing to read and I can't see what's happening under the hood. One approach to making things more readable and transparent might involve chunks breaking things down and creating a variable for each link in the chain of functions.

First I'll get the 3 means.

```
three_means <- sapply(county[three_columns], mean, na.rm=TRUE)
three_means
```

```
##   pop2000   pop2010   pop2017
##  89649.99  98262.04 103763.41
```

Then I get the median of those 3 means.

```
the_median <- median(three_means)
the_median
```

```
## [1] 98262.04
```

Then I take the log to get a final answer.

```
the_answer <- log(the_median)
the_answer
```

```
## [1] 11.49539
```

It's the same result with nice breaks and space to insert comments that explain what's going on.

Formatting prettier code

Another way to make complex commands more readable is to use prettier formatting. Here's what that might look like:

```
log(
  median(
    sapply(
      county[three_columns],mean, na.rm=TRUE
      )
```

```
    )
  )
```

```
## [1] 11.49539
```

Again, exactly the same output, exactly the same operations. Much easier to see how the functions are layered on top of each other.

Automatic code "prettification"

We also talked about prettifying code using some built-in functions in RStudio within the `Code` dropdown menu. Specifically, you might look at the help documentation for the `Reflow comment`, `Reindent lines`, and `Reformat code` menu items or just try them out. We haven't developed examples yet that can actually benefit from these commands, but here's a chunk of code from the upcoming R tutorial that I've organized "concisely." For now, try not to worry about what the code would do, and instead focus on how it's formatted. It's very short and it works perfectly, but it's pretty hard to figure out what's going on:

```
## messy chunk
my.mean <- function(z){z<-z[!is.na(z)];sigma<-sum(z);n<-length(z);out.value<-sigma/n;return(out.value)}
```

When I highlight that block of code and click the `Reformat Code` command from the `Code` dropdown menu here's what it looks like:

```
## messy chunk after reformatting
my.mean <-
  function(z) {
    z <-
      z[!is.na(z)]
    sigma <- sum(z)
    n <- length(z)
    out.value <- sigma / n
    return(out.value)
  }
```

It's exactly the same code. The only difference is how the text is organized. The result is far more readable.

You can also use an option in RMarkdown's code chunks to call `tidy=TRUE` *inside* the beginning of the chunk within the curly brackets. The following chunk of code looks terrible in the raw .rmd file—it's exactly the same "messy chunk" a few lines up. By including the `tidy=TRUE` option, it looks much better when it's knitted.

```
## messy chunk with `tidy=TRUE` chunk option:
my.mean <- function(z) {
    z <- z[!is.na(z)]
    sigma <- sum(z)
    n <- length(z)
    out.value <- sigma/n
    return(out.value)
}
```

Specifying variable classes with data import

Aaron C. asked a question about whether/how you might specify variable classes when you're importing data. Aaron S. punted at the time, so here's a slightly more specific reply.

The short answer is, "yes, R can do this." The details depend on exactly which function you use to import the data in question (and that depends partly on the file format... etc.).

The most helpful place to look for more information is the help documentation for whatever import function you might be working with. For example, the `read.csv()` function that gets introduced in the next R tutorial takes an optional argument for colClasses that allows you to specify a vector of classes (e.g., `c("character", "factor", "integer", "character")`) corresponding to the classes you want R to assume for each incoming column of the data.

Try reading `help(read.csv)` and look at the documentation for the `colClasses` argument to learn more.

R guesses the classes of variables when you import them

Aaron and Nick both made comments about R guessing the classes of variables when you import data. The nature and quality of these guesses can depend on the import function there too.

Most Base R import stuff makes guesses you might think of as somewhat brittle (assumptions (e.g., looking at just the first five values to inform the guess. In contrast, the Tidyverse data import commands usually use a larger and more random sample of values from each column to make guesses (which are therefore much better).