Week 4 R tutorial

Statistics and statistical programming
Northwestern University
MTS 525

Aaron Shaw

October 6, 2020

Contents

Importing (yet more) data **1**
Importing proprietary data formats 3

Tidy and manage data **3**
Conditional means with nested `*apply` functions 4
Conditional means in Tidyverse code 4
Merging data frames 5
Cleaning and organizing data 6
Working with dates 7

Creating (repeated) sequences, distributions, and samples **8**
Managing randomness 9

Creating functions **10**
Example: Creating my own `mean()` function: 11
Two tips for writing and debugging your functions 11

Welcome back! This week we'll get into some more advanced fundamentals to help you import and tidy/manage data, define and run your own functions, generate distributions and samples, and manage dates. Some of these topics are more useful for future problem sets (e.g., the stuff about distributions and dates), but I've included them here to start introducing the ideas.

Importing (yet more) data

So far, we have imported datasets that come installed with R packages with the `data()` function as well as the `load()` function to read R data files (.RData, .rda, etc.). For better and worse, you should be able to import data in other formats too.

Before I get any further, I want to note that my approach here is to use R commands directly that you can type in RMarkdown scripts or run at the console yourself. RStudio also provides a number of handy data import tools through the graphical interface and drop-down menus. This how-to article introduces some of these resources.

Tabular (rows and columns) data files formatted as plain text with "comma-separated values" (".csv's") are quite common, so we'll look at those. R comes with a handy `read.csv()` command that does exactly what you'd expect. Here's an example using a csv file I created from one of R's built-in datasets called `mtcars`, which has old data about fuel consumption cars. Run `help(mtcars)` to learn more about where it comes from and to read the variable descriptions. Since it's built-in, you can import it using `data(mtcars)`, but I also posted it to the course data repository so we can use the `url()` command to point `read.csv()` to download it:

```
data.url <- url("https://communitydata.science/~ads/teaching/2020/stats/data/week_04/mtcars.csv")

my.mtcars <- read.csv(data.url)

head(my.mtcars)
```

```
##                   X  mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1         Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2     Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3        Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## 4    Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 5 Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## 6           Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

As always, take a look at the help documentation for `read.csv()` to learn more about some of the arguments that you can use. Because data comes in so many (weird) formats, there are many possible arguments!

You might notice that the documentation for `read.csv()` is actually part of the documentation for another command called `read.delim()`. Turns out `read.delim()` is just a more general-purpose way to read in tabular data and that `read.csv()` is short-hand for `read.delim()` with some default values that make sense for csv files. Here is a command that produces identical output to the previous one

```
more.cars <- read.delim(url("https://communitydata.cc/~ads/teaching/2019/stats/data/week_03/mtcars.csv"

head(more.cars)
```

```
##                   X  mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1         Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2     Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3        Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## 4    Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 5 Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## 6           Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
table(more.cars == my.mtcars)
```

```
##
## TRUE
##  384
```

When find yourself trying to load a tabular data file that consists of plain text, but has some idiosyncratic difference from a csv (e.g., it is tab-separated instead of comma-separated), you should use `read.delim()`.

How will you know what to use? Get to know your data first! Seriously, try opening it the file (or at least opening up part of it) using a text editor and/or spreadsheet software. Looking at the "raw" plain text can help you figure out what arguments you need to use to make the data load up exactly the way you want it.

For example, you might notice that my import of the mtcars.csv file introduces an important difference from the original `mtcars` dataset. In the original `mtcars`, the car model names are `row.names` attributes of the dataframe instead of a variable.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
row.names(mtcars)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"         
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"            
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"           
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"         
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood" 
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"           
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"      
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"         
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"      
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"       
## [31] "Maserati Bora"       "Volvo 142E"
```

Since it's the first column of the raw data, you can fix this with an additional argument to `read.csv`:

```
my.mtcars <- read.csv(url("https://communitydata.science/~ads/teaching/2020/stats/data/week_04/mtcars.cs
                      row.names=1)
head(my.mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
table(my.mtcars == mtcars)
```

```
## 
## TRUE 
##  352
```

This illustrates a common issue that relates back to variable types (classes). Most of the commands in R that import data try to "guess" what class is appropriate for each column of your dataset. Surprise, surprise, these guesses are sometimes not so great and often quite different from what you might guess. As a result, it's a great idea to inspect the classes of every column of a dataset after you import it (review last week's R lecture materials for more on this).

Importing proprietary data formats

R has libraries that can read (and write) many proprietary data file formats, including files from Stata, SAS, MS Excel, and SPSS (among others).

That same helpful Rstudio how-to data import article includes several examples of these. You can find other suggestions and examples online.

Tidy and manage data

Last tutorial introduced the "*apply" family of functions (e.g., `sapply`, `lapply`, and `tapply`). This time around, I want to introduce an example of how to use them repeatedly over a set of objects.

Conditional means with nested *apply functions

I used mtcars for this last time around, so let's start off with it again. Before I used tapply() to calculate conditional means of miles per gallon for each number of cylinders. What if I wanted to do calculate conditional means for a bunch of the of the other variables in the mtcars dataset? You *could* just write the same line of tapply code over and over again until you had one line for each variable. However, there's a much more efficient way:

```
variables <- c("mpg", "disp", "hp", "wt")

sapply(mtcars[variables], function(v){
  tapply(v, mtcars$cyl, mean)
  }
  )
```

```
##        mpg     disp        hp       wt
## 4 26.66364 105.1364  82.63636 2.285727
## 6 19.74286 183.3143 122.28571 3.117143
## 8 15.10000 353.1000 209.21429 3.999214
```

Do you follow the example? Try reading it "inside-out" to understand out what it does. The innermost part is the call to tapply() and that should look pretty familiar from the previous tutorial example. The difference now is that it contains this weird call to some variable v. Working backwards, the previous line of code defines v as an argument to a function. The function is, in turn, being passed as an argument to sapply(), which it is applying to the columns in mtcars indexed by my vector of variable names.

Put another way: I have used sapply to call tapply to calculate conditional means (by cylinder number) for each item in the variables vector. My example only included four variables, but you could do the same thing for an arbitrarily large set of variables and it would work just fine. Indeed, it would be a very fast, efficient way of solving a seemingly complicated data processing task with very little code.

Conditional means in Tidyverse code

I should reiterate here that the *apply functions are part of Base R. Other functions to do similar things exist and you may find these other functions more intelligible or useful for you. In particular, this is the sort of task that the Tidyverse handles in an arguably more intuitive fashion than Base R because it allows you to organize functions as a sequence of actions with function names that are verbs and this generally leads to more readable code. Here's an example snippet that replicates the same output. :

```
library(tidyverse)
```

```
## -- Attaching packages ----------------------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.1     v dplyr   1.0.2
## v tidyr   1.1.1     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
```

```
## -- Conflicts -------------------------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
mtcars %>%
  group_by(cyl) %>%
  summarize( across(
    .cols=c(mpg, disp, hp, wt),
    .fns=list(
      avg=mean) ## writing this in this way allows the summarize function to create a useful variable n
    ))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 5
##     cyl mpg_avg disp_avg hp_avg wt_avg
##   <dbl>   <dbl>    <dbl>  <dbl>  <dbl>
## 1     4    26.7     105.   82.6   2.29
## 2     6    19.7     183.  122.    3.12
## 3     8    15.1     353.  209.    4.00
```

A little bit more verbose, but still quite concise![1] Let's unpack it. The first thing you'll notice is the `%>%` symbol on the first line. In Tidyverse-speak `%>%` is referred to as a "pipe." It tells R to take the thing to the left and use it to perform the action that follows on the right (or below in this example). The other thing to notice is that when I'm working inside of these "piped" commands, the Tidyverse doesn't require me to use quotations to indicate variable/column names.

Once you know those Tidyverse fundamentals, the rest is nearly readable: you take the `mtcars` data set, group it by the `cyl` variable, and then summarize across a series of columns (`.cols`) with some function(s) (`.fns`).

Since the Tidyverse code takes a grammatical structure that nearly maps to English, adding another action (like calculating an additional summary statistic such as the conditional variance) can be done in a fairly straightforward way:

```
mtcars %>%
  group_by(cyl) %>%
  summarize( across(
    .cols=c(mpg, disp, hp, wt),
    .fns=list(avg = mean,
              sigma.sq = var)
    ))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 9
##     cyl mpg_avg mpg_sigma.sq disp_avg disp_sigma.sq hp_avg hp_sigma.sq wt_avg
##   <dbl>   <dbl>        <dbl>    <dbl>         <dbl>  <dbl>       <dbl>  <dbl>
## 1     4    26.7        20.3      105.          722.   82.6        438.   2.29
## 2     6    19.7         2.11     183.         1727.  122.         589.   3.12
## 3     8    15.1         6.55     353.         4593.  209.        2599.   4.00
## # ... with 1 more variable: wt_sigma.sq <dbl>
```

Don't worry if you didn't follow that yet. The Tidyverse documentation is extensive. Indeed, my example looks a lot like some of the examples from the documentation for the `summarize()` function and the `across()` function (Hadley Wickham, the author of most of the Tidyverse uses `mtcars` a **lot** in his example code). For now, I mostly wanted to introduce a Tidyverse data cleanup/management example because it's an area where the Tidyverse really shines.

Merging data frames

Combining various chunks of data together poses another family of common data management and processing challenges.

One way to do this when you have multiple dataframes is to use the `merge()` function. A merge might involve dataframes with mutually exclusive rows/columns or it might involve dataframes that already have

[1] Indeed, this command could be yet more concise, but I've intentionally left it a little "wordy" to facilitate readability and allow the example to develop further.

some things in common. I'll present an example here using two example dataframes that I create by hand and that share a column:

```
spiders <- c("black widow", "brown recluse", "daddy longlegs")
poisonous <- c(TRUE, TRUE, FALSE)
region <- c("south", "central", "all")
genus <- c("Latrodectus", "Lococsceles", "many")
number <- c(2, 1, 250)


# Notice that the datasets here share one column in common:
s1 <- data.frame(spiders, poisonous, region)
s2 <- data.frame(spiders, genus, number)

s.combined <- merge(s1, s2, by="spiders")
s.combined
```

```
##          spiders poisonous  region       genus number
## 1    black widow      TRUE   south Latrodectus      2
## 2  brown recluse      TRUE central  Lococsceles      1
## 3 daddy longlegs     FALSE     all        many    250
```

The by="spiders" part of the call to merge() tells R to use the shared column to produce a combined dataframe.

Take a look at the documentation for merge() and search for additional examples on StackOverflow and beyond if/when you want to pursue more complicated tasks along these lines. The arguments you can pass via by and all can be especially useful.

Cleaning and organizing data

You already saw several ways to cleanup data in my initial presentation of dataframes last week. Let's return to that and build out some additional examples here. A lot of what follows is about taking the results of various data management steps and making sure they are arranged so as to make subsequent steps more intelligible. While it may seem tedious to spend time writing or revising your code to do this kind of thing, the payoff is code and data that are more human-readable. When your code and data are more human-readable, it's easier to develop intuitions and find mistakes.

Take a look at the class of the object returned by a previous example:

```
class(
  sapply(mtcars[variables], function(v){
    tapply(v, mtcars$cyl, mean)
}) )
```

```
## [1] "matrix" "array"
```

Hmm, what if you wanted a data frame? You already know how to do that! I'll call data.frame() and assign the output to a new object:

```
cyl.conditional.means <- data.frame(
  lapply(mtcars[variables], function(v){
    tapply(v, mtcars$cyl, mean)
}) )

head(cyl.conditional.means)
```

```
##        mpg     disp        hp       wt
## 4 26.66364 105.1364  82.63636 2.285727
## 6 19.74286 183.3143 122.28571 3.117143
## 8 15.10000 353.1000 209.21429 3.999214
```

We can even do some more cleanup to give the cylinder count its own variable and rename the other columns to reflect that they consist of averages:

```
names(cyl.conditional.means) <- c("mean.mpg", "mean.disp", "mean.hp", "mean.wt")

# Notice that I'm converting the values back to numeric
cyl.conditional.means$cyl.number <- as.numeric(row.names(cyl.conditional.means))

# Maybe look at the row names before you run this next line so that you can see what it does.
row.names(cyl.conditional.means) <- NULL

head(cyl.conditional.means)
```

```
##   mean.mpg mean.disp   mean.hp  mean.wt cyl.number
## 1 26.66364  105.1364  82.63636 2.285727          4
## 2 19.74286  183.3143 122.28571 3.117143          6
## 3 15.10000  353.1000 209.21429 3.999214          8
```

This data should be clearer and easier to work with now.

What if, instead of having that dataframe sorted by the number of cylinders, you wanted to sort it by the value of `mean.mpg`? The `sort.list` function can help:

```
cyl.conditional.means[sort.list(cyl.conditional.means$mean.mpg),]
```

```
##   mean.mpg mean.disp   mean.hp  mean.wt cyl.number
## 3 15.10000  353.1000 209.21429 3.999214          8
## 2 19.74286  183.3143 122.28571 3.117143          6
## 1 26.66364  105.1364  82.63636 2.285727          4
```

Again, you could run `row.names(cyl.conditional.means)` <- NULL‘ to reset the row numbers.

By now you might have noticed that a some of my code in this section replicates parts of the output that the Tidyverse example above created by default. As I said earlier, the Tidyverse really shines when it comes to tidying data (shocking, I know). There are also functions ("verbs") in the Tidyverse libraries to perform all of the additional steps (such as sorting data by the values of a column). If you are interested in doing so, check out more of the Tidyverse documentation and the Wickham and Grolemund *R for Data Science* book listed among the course resources.

Working with dates

Date and time objects are another more advanced data class in R. Managing date and time data can be a headache. This is because dates and times tend to be formatted inconsistently and are usually given to you as character variables, so you will need to transform them into a format that R can "understand" as dates. There are many packages for working with dates and times, but for now I'll introduce you to the Base R way of doing so. This uses a data type formally called "POSIX" (no need to worry about why it's called that).

To build up an example, I'll create some date-time values, add a little noise, and convert them into a character vector:

```
add.an.hour <- seq(0, 3600*24, by=3600)
some.hours <- as.character(Sys.time() + add.an.hour) ## Look up Sys.time() to see what it does.
```

Now, let's take a look at the results of that:

```
class(some.hours)
```

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

```r
head(some.hours)
```

```
## [1] "2020-09-28 12:05:27" "2020-09-28 13:05:27" "2020-09-28 14:05:27"
## [4] "2020-09-28 15:05:27" "2020-09-28 16:05:27" "2020-09-28 17:05:27"
```

These are beautifully formatted timestamps, but R will not understand them as such. This is often how you might receive data in, for example, a dataset you import from Qualtrics, scrape from the web, or elsehwere. You can convert the `some.hours` vector into an object class that R will recognize as a time object using the `as.POSIXct()` function. Notice that it even adds a timezone back in!

```r
as.POSIXct(some.hours)
```

```
##  [1] "2020-09-28 12:05:27 CDT" "2020-09-28 13:05:27 CDT"
##  [3] "2020-09-28 14:05:27 CDT" "2020-09-28 15:05:27 CDT"
##  [5] "2020-09-28 16:05:27 CDT" "2020-09-28 17:05:27 CDT"
##  [7] "2020-09-28 18:05:27 CDT" "2020-09-28 19:05:27 CDT"
##  [9] "2020-09-28 20:05:27 CDT" "2020-09-28 21:05:27 CDT"
## [11] "2020-09-28 22:05:27 CDT" "2020-09-28 23:05:27 CDT"
## [13] "2020-09-29 00:05:27 CDT" "2020-09-29 01:05:27 CDT"
## [15] "2020-09-29 02:05:27 CDT" "2020-09-29 03:05:27 CDT"
## [17] "2020-09-29 04:05:27 CDT" "2020-09-29 05:05:27 CDT"
## [19] "2020-09-29 06:05:27 CDT" "2020-09-29 07:05:27 CDT"
## [21] "2020-09-29 08:05:27 CDT" "2020-09-29 09:05:27 CDT"
## [23] "2020-09-29 10:05:27 CDT" "2020-09-29 11:05:27 CDT"
## [25] "2020-09-29 12:05:27 CDT"
```

If things aren't formatted in quite the way R expects, you can also tell it how to parse a character string as a POSIXct object:

```r
m <- "2019-02-21 04:35:00"
class(m)
```

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

```r
a.good.time <- as.POSIXct(m, format="%Y-%m-%d %H:%M:%S", tz="CDT")
class(a.good.time)
```

```
## [1] "POSIXct" "POSIXt"
```

Once you have a time object, you can even do date arithmetic with `difftime()` (but watch out as this can get complicated):

```r
difftime(Sys.time(), a.good.time, units="weeks")
```

```
## Time difference of 83.64588 weeks
```

This calculated the number of weeks elapsed between the current time and an example date/time I created above.

Creating (repeated) sequences, distributions, and samples

The *OpenIntro* reading this week introduces some core concepts and tools for probability. R has a number of functions that you can use to illustrate these concepts, replicate examples, and generate simulations of your own. This section of the tutorial introduces a few useful starting points. You may also find these snippets and functions valuable for managing real data.

First, let's create two sequences of numbers using the `seq()` function:

```
odds <- seq(from=1, to=100, by=2)
evens <- seq(from=2, to=100, by=2)

head(odds)
```

```
## [1]  1  3  5  7  9 11
```

```
head(evens)
```

```
## [1]  2  4  6  8 10 12
```

Or maybe (for some reason) I want to have a vector that repeats the odd integers between 1 and 100 five times? Try `rep()`:

```
more.odds <- rep(seq(from=1, to=100, by=2), 5)
```

You can use `sample()` to draw samples from an object:

```
sample(x=odds, size=3)
```

```
## [1] 85 95  3
```

```
sample(x=evens, size=3)
```

```
## [1]  36 100  38
```

You can also sample "with replacement." Here I take 100 random draws from the binomial distribution, which is analogous to 100 independent trials of, say, a coin flip:

```
draws <- sample(x=c(0,1), size=100, replace=TRUE)

table(draws)
```

```
## draws
##  0  1
## 52 48
```

What if you wanted to take a random set of 10 observations from a dataframe? You can use `sample` on an index of the rows:

```
odds.n.evens <- data.frame(odds, evens)

odds.n.evens[ sample(row.names(odds.n.evens), 10), ]
```

```
##    odds evens
## 46   91    92
## 30   59    60
## 18   35    36
## 41   81    82
## 11   21    22
## 43   85    86
## 7    13    14
## 34   67    68
## 27   53    54
## 10   19    20
```

Managing randomness

Try running one of the `sample` commands above again. You will (probably!) get a different result because `sample` makes a random draw each time it runs.

Even in statistics, you sometimes want things to be a little *less random*. For example, let's say you want to reproduce a specific random draw. To do this, you use the `set.seed()` command, which takes any integer as its argument. Setting the seed value in this way ensures that the next random draw will be the same each time:

```
set.seed(42)
x <- sample(odds, 10)

y <- sample(odds, 10) # different

set.seed(42)
z <- sample(odds, 10) #

head(x)
```

```
## [1] 97 73  1 49 19 71
```

```
head(y)
```

```
## [1] 49 73 91 39 51  5
```

```
head(z)
```

```
## [1] 97 73  1 49 19 71
```

```
table(x==y) ##  One in ten!
```

```
##
## FALSE  TRUE
##     9     1
```

```
table(x==z)
```

```
##
## TRUE
##   10
```

Note that the first random draw after I set the seed to the same value is the same both times (see the results of `table(x==z)`), but not the same when I create `y`.

Creating functions

You should learn how to define and run your own functions. This is a big part of what makes statistical **programming** a thing in R and what has made it possible for people to develop such a wide range of packages on top of the R Base.

Defining a function in R requires that you use the `function()` command to let R know what your function can do as well as what arguments you will pass to it. Here is an example of a function that I'll call `times.two()`. It takes a number and multiplies it by two:

```
times.two <- function(x){
  twicex <- x * 2
  return(twicex)
}

times.two(21)
```

```
## [1] 42
```

A few things to notice here. First, you can see that I assigned my function to an object/variable (in this case `times.two`) just like we've done with other kinds of objects. (**Important**: You need to execute the code that defines the function in order to be able to use it!)

I use curly braces (`{ }`) to contain the content of the function. Everything inside the curly braces is the "action" that the function executes when I call it later (in this case, the function takes an arbitrary value, multiplies it by two, and returns the result).

The `return()` syntax at the end of the function tells R exactly what output I want my function to "return" when I execute it. As usual in R, you can leave out the explicit call to something like `return()` and R will try to guess what you want based on the last thing you asked it to do inside the function. That guess will usually be bad, so you're almost always better off calling `return()`.

Any function needs something inside the parentheses, known as an "argument." In this case the argument, `x` is included in the definition and content of my function. The important thing to know is that this `x` will really only have meaning *inside* the context of the function. If another `x` is defined elsewhere in my R environment at the time I run my function, R won't necessarily "see" it within the function.

Finally, in the last line of that code block, you can see that I have "called" my function with the argument "21." Since I execute the part of my code that defines the function, R now knows what `two.times()` means, so when I call it and provide the argument `21`, R goes ahead and tries to execute the code I wrote substituting the argument for `x`.

Example: Creating my own `mean()` function:

Here is a slightly more complicated function that takes a numeric vector, removes any `NA` (missing) values, and calculates the arithmetic mean:

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

You can try running this on the built-in `rivers` dataset (run `help(rivers)` to see what it's about). Here I'll add a missing value and then I'll compare the output of `my.mean()` to the output of the `mean()` function included with R:

```r
data(rivers)
rivers[7] <- NA

my.mean(rivers)
```

```
## [1] 584.9857
```

```r
mean(rivers, na.rm=TRUE)
```

```
## [1] 584.9857
```

A key reminder: in order to access a user-defined function you must execute the code that defines the function first. If you don't do this, R won't know what you're talking about when you ask it to run the new function.

Two tips for writing and debugging your functions

When you're creating a function of you're own it's common to have things not-quite-work at some point along the way (especially while you're just learning the syntax of the R language!). In other words, your code will have bugs. What to do?!

Debugging is a big topic and a reality of programming in any language (just ask someone you know with previous programming experience!). Without going into much depth, I offer two very simple tips for avoiding, identifying, and fixing bugs in your R code:

(1) **Work small.**

You may have big plans for your function, but it's often helpful to build towards those big plans in small chunks. For example, notice that the `my.mean()` function I defined above only does one thing on each line. This is intentional. It is possible to write an elaborate single line of code to do all of the things in that function. However, doing so increases the likelihood that there will be an error somewhere and that the error will be much, much harder to see. Shorter, simpler lines of code (especially while you're learning) are often better.

(2) **Test often.**

If you have a function that involves multiple steps, any one of them may go wrong. A great way to evaluate/prevent this is to test each step as you go to make sure it's producing the output you expect. One way to do this is by building up your function adding one piece at a time and inspecting the output. For example, you might create some test data and run each line of your code individually on the test data to make sure it does what you expect. Here's an example that test the first line of `my.mean()`:

```
y <- c(2,12,31,26,NA,25)

y
```

```
## [1]  2 12 31 26 NA 25
```

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

```
## [1]  2 12 31 26 25
```

This line-by-line approach is good, but eventually you need to also make sure things are working *inside* the function. A basic way to do this is using the `print()` command to print the results of intermediate steps in your function.

Here is an example of that using the `y` vector I just defined and a new version of the `my.mean` function with a debugging (`print()`) line included:

```
debug.my.mean <- function(z){
  z <- z[!is.na(z)]
  sigma <- sum(z)
  print(sigma) ## here's my debugging line.
  n <- length(z)
  out.value <- sigma/n
  return(out.value)
}

debug.my.mean(y)
```

```
## [1] 96
```

```
## [1] 19.2
```

```
sum(y, na.rm = TRUE) ## Checks my code against a built-in function
```

```
## [1] 96
```

Notice that the call to `debug.my.mean()` produces two pieces of output: the thing I asked to be printed in the middle of the function as well as the returned value of the function itself.