derekfranks on Nov 14, 2014 Additional "advanced" section edits

1 contributor

307 lines (220 sloc) 13.386 kb

Raw Blame History

output
pdf_document

Practice Assignment

The goal of this assignment is to provide a "bridge" between the first two weeks of lectures and assignment 1 for those either new to R or struggling with how to approach the assignment.

This guided example, will **not** provide a solution for programming assignment 1. However, it will guide you through some core concepts and give you some practical experience to hopefully make assignment 1 seems less daunting.

To begin, download this file and unzip it into your R working directory.
http://s3.amazonaws.com/practice_assignment/diet_data.zip

You can do this in R with the following code:

```
dataset_url <- "http://s3.amazonaws.com/practice_assignment/diet_data.zip"
download.file(dataset_url, "diet_data.zip")
unzip("diet_data.zip", exdir = "diet_data")
```

If you're not sure where your working directory is, you can find out with the `getwd()` command. Alternatively, you can view/change it through the Tools > Global Options menu in R Studio.

So assuming you've unzipped the file into your R directory, you should have a folder called diet_data. In that folder there are five files. Let's get a list of those files:

```
list.files("diet_data")
```

Okay, so we have 5 files. Let's take a look at one to see what's inside:

```
andy <- read.csv("diet_data/Andy.csv")
head(andy)
```

It appears that the file has 4 columns, Patient.Name, Age, Weight, and Day. Let's figure out how many rows there are by looking at the length of the 'Day' column:

```
length(andy$Day)
```

Alternatively, you could look at the dimensions of the data.frame:

```
dim(andy)
```

This tells us that we 30 rows of data in 4 columns. There are some other commands we might want to run to get a feel for a new data file, `str()` , `summary()` , and `names()` .

```
str(andy)
summary(andy)
names(andy)
```

So we have 30 days of data. To save you time, all of the other files match this format and length. I've made up 30 days worth of weight data for 5 subjects of an imaginary diet study.

Let's play around with a couple of concepts. First, how would we see Andy's starting weight? We want to subset the data. Specifically, the first row of the 'Weight' column:

```
andy[1, "Weight"]
```

We can do the same thing to find his final weight on Day 30:

```
andy[30, "Weight"]
```

Alternatively, you could create a subset of the 'Weight' column where the data where 'Day' is equal to 30.

```
andy[which(andy$Day == 30), "Weight"]
andy[which(andy[,"Day"] == 30), "Weight"]
```

Or, we could use the `subset()` function to do the same thing:

```
subset(andy$Weight, andy$Day==30)
```

There are lots of ways to get from A to B when using R. However it's important to understand some of the various approaches to subsetting data.

Let's assign Andy's starting and ending weight to vectors:

```
andy_start <- andy[1, "Weight"]
andy_end <- andy[30, "Weight"]
```

We can find out how much weight he lost by subtracting the vectors:

```
andy_loss <- andy_start - andy_end
andy_loss
```

Andy lost 5 pounds over the 30 days. Not bad. What if we want to look at other subjects or maybe even everybody at once?

Let's look back to the `list.files()` command. It returns the contents of a directory in alphabetical order. You can type `?list.files` at the R prompt to learn more about the function.

```
files <- list.files("diet_data")
files
```

Knowing that 'files' is now a list of the contents of 'diet_data' in alphabetical order, we can call a specific file by subsetting it:

```
files[1]
files[2]
files[3:5]
```

Let's take a quick look at John.csv:

```
head(read.csv(files[3]))
```

Woah, what happened? Well, John.csv is sitting inside the diet_data folder. We just tried to run the equivalent of `read.csv("John.csv")` and R correctly told us that there isn't a file called John.csv in our working directory. To fix this, we need to append the directory to the beginning of the file name.

One approach would be to use `paste()` or `sprintf()` . However, if you go back to the help file for `list.files()` , you'll see that there is an argument called `full.names` that will append (technically prepend) the path to the file name for us.

```
files_full <- list.files("diet_data", full.names=TRUE)
files_full
```

Pretty cool. Now let's try taking a look at John.csv again:

```
head(read.csv(files_full[3]))
```

Success! So what if we wanted to create one big data frame with everybody's data in it? We'd do that with rbind and a loop. Let's start with rbind:

```
andy_david <- rbind(andy, read.csv(files_full[2]))
```

This line of code took our existing data frame, Andy, and added the rows from David.csv to the end of it. We can check this with:

```
head(andy_david)
tail(andy_david)
```

One thing to note, rbind needs 2 arguments. The first is an existing data frame and the second is what you want to append to it. This means that there are occassions when you might want to create an empty data frame just so there's *something* to use as the existing data frame in the rbind argument.

Don't worry if you can't imagine when that would be useful because you'll see an example in just a little while.

Now, let's create a subset of the data frame that shows us just the 25th day for Andy and David.

```
day_25 <- andy_david[which(andy_david$Day == 25), ]
day_25
```

Now you could manually go through and append everybody's data to the same data frame using rbind, but that's not a practical solution if you've got lots and lots of files. So let's try using a loop.

To understand what's happening in a loop, let's try something:

```
for (i in 1:5) {print(i)}
```

As you can see, for each pass through the loop, i increases by 1 from 1 through 5. Let's apply that concept to our list of files.

```
for (i in 1:5) {
        dat <- rbind(dat, read.csv(files_full[i]))
}
```

Whoops. Object 'dat' not found. This is because you can't rbind something into a file that doesn't exist yet. So let's create an empty data frame called 'dat' before running the loop.

```
dat <- data.frame()
for (i in 1:5) {
        dat <- rbind(dat, read.csv(files_full[i]))
}
str(dat)
```

Cool. We now have a data frame called 'dat' with all of our data in it. Out of curiousity, what would happen if we had put `dat <- data.frame()` inside of the loop? Let's see:

```
for (i in 1:5) {
        dat2 <- data.frame()
        dat2 <- rbind(dat2, read.csv(files_full[i]))
}
str(dat2)
head(dat2)
```

Because we put `dat2 <- data.frame()` inside of the loop, `dat2` is being rewritten with each pass of the loop. So we only end up with the data from the last file in our list.

Back to `dat` ... So what if we wanted to know the median weight for all the data? Let's use the `median()` function.

```
median(dat$Weight)
```

NA? Why did that happen? Type 'dat' into the console and you'll see a print out of all 150 obversations. Scroll back up to row 77, and you'll see that we have some missing data from John, which is recorded as NA by R.

We need to get rid of those NA's for the purposes of calculating the median. There are several approaches. For instance, we could subset the data using `complete.cases()` or `is.na()` . But if you look at `?median` , you'll see there is an argument called `na.rm` that will strip the NA values out for us.

```
median(dat$Weight, na.rm=TRUE)
```

So 190 is the median weight. We can find the median weight of day 30 by taking the median of a subset of the data where Day=30.

```
median(dat_30$Weight)
```

We've done a lot of manual data manipulation so far. Let's build a function that will return the median weight of a given day.

Let's start out by defining what the arguments of the function should be. These are the parameters that the user will define. The first parameter the user will need to define is the directory that is holding the data. The second parameter they need to define is the day for which they want to calculate the median.

So our function is going to start out something like this:

```
weightmedian <- function(directory, day) { # content of the function }
```

So what goes in the content? Let's think through it logically. We need a data frame with all of the data from the CSV's. We'll then subset that data frame using the argument `day` and take the median of that subset.

In order to get all of the data into a single data frame, we can use the method we worked through earlier using `list.files()` and `rbind()`.

Essentially, these are all things that we've done in this example. Now we just need to combine them into a single function.

So what does the function look like?

```
weightmedian <- function(directory, day)  {
        files_list <- list.files(directory, full.names=TRUE)    #creates a list of files
        dat <- data.frame()                                     #creates an empty data frame
        for (i in 1:5) {
                #loops through the files, rbinding them together
                dat <- rbind(dat, read.csv(files_list[i]))
        }
        dat_subset <- dat[which(dat[, "Day"] == day),]  #subsets the rows that match the 'day' argument
        median(dat_subset[, "Weight"], na.rm=TRUE)      #identifies the median weight
                                                        #while stripping out the NAs
}
```

You can test this function by running it a few different times:

```
weightmedian(directory = "diet_data", day = 20)
weightmedian("diet_data", 4)
weightmedian("diet_data", 17)
```

Hopefully, this has given you some practice applying the basic concepts from weeks 1 and 2. If you can work your way through this example, you should have all of the tools needed to complete part 1 of assignment 1. Parts 2 and 3 are really just expanding on the same basic concepts, potentially adding in some ideas like cbinds and if-else.

One last quick note: The approach I'm showing above for building the data frame is submoptimal. It works, but generally speaking, you don't want to build data frames or vectors by copying and re-copying them inside of a loop. If you've got a lot of data it can become very, very slow. However, this tutorial is meant to provide an introduction to these concepts, and you can use this approach successfully for programming assignments 1 and 3.

If you're interested in learning the better approach, check out Hadley Wickam's excellent material on functionals within R: http://adv-r.had.co.nz/Functionals.html. But if you're new to both programming and R, I would skip it for now as it will just confuse you. Come back and revisit it (and the rest of this section) once you are able to write working functions using the approach above.

The main issue with the approach above is growing an object inside of loop by copying and recopying it. It works, but it's slow and if you've got a lot of data, it will probably cause issues. The better approach is to create an output object of an appropriate size and then fill it up.

So the first thing we do is create an empty list that's the length of our expected output. In this case, our input object is going to be `files_full` and our empty list is going to be `tmp`.

```
summary(files_full)
tmp <- vector(mode = "list", length = length(files_full))
summary(tmp)
```

Now we need to read in those csv files and drop them into `tmp`.

```
for (i in seq_along(files_full)) {
        tmp[[i]] <- read.csv(files_full[[i]])
}
str(tmp)
```

What we just did was read in each of the csv files and place them inside of our list. Now we have a list of 5 elements called `tmp`, where each element of the list is a data frame containing one of the csv files. It just so happens that what we just did is functionally identical to using `lapply`.

```
str(lapply(files_full, read.csv))
```

This is part of the power of the apply family of functions. You don't have to worry about the "housekeeping" of looping, and instead you can focus on the function you're using. When you or somebody else comes back weeks later and reads through your code, it's easier to understand what you were doing and why. If somebody says that the apply functions are more "expressive", this is what they mean.

Now we still need to go from a list to a single data frame, although you *can* manipulate the data within this structure:

```
str(tmp[[1]])
head(tmp[[1]][,"Day"])
```

We can use a function called `do.call()` to combine `tmp` into a single data frame. `do.call` lets you specify a function and then passes a list as if each element of the list were an argument to the function. The syntax is `do.call(function_you_want_to_use, list_of_arguments)`. In our case, we want to `rbind()` our list of data frames, `tmp`.

```
output <- do.call(rbind, tmp)
str(output)
```

This approach avoids all of the messy copying and recopying of the data to build the final data frame. It's much more "R-like" and works quite a bit faster than our other approach.