R Programming Practice Assignment

Derek Franks

Twitter: @derek_franks

==

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")
```

```
## [1] "Andy.csv"  "David.csv" "John.csv"  "Mike.csv"  "Steve.csv"
```

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)
```

```
##   Patient.Name Age Weight Day
## 1         Andy  30    140   1
## 2         Andy  30    140   2
## 3         Andy  30    140   3
## 4         Andy  30    139   4
## 5         Andy  30    138   5
## 6         Andy  30    138   6
```

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)
```

```
## [1] 30
```

30 rows. OK.

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

```r
dim(andy)
```

```
## [1] 30  4
```

This tells us that we have 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()`.

```r
str(andy)
```

```
## 'data.frame':    30 obs. of  4 variables:
##  $ Patient.Name: Factor w/ 1 level "Andy": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age         : int  30 30 30 30 30 30 30 30 30 30 ...
##  $ Weight      : int  140 140 140 139 138 138 138 138 138 138 ...
##  $ Day         : int  1 2 3 4 5 6 7 8 9 10 ...
```

```r
summary(andy)
```

```
##  Patient.Name      Age         Weight           Day       
##  Andy:30      Min.   :30   Min.   :135.0   Min.   : 1.00  
##               1st Qu.:30   1st Qu.:137.0   1st Qu.: 8.25  
##               Median :30   Median :137.5   Median :15.50  
##               Mean   :30   Mean   :137.3   Mean   :15.50  
##               3rd Qu.:30   3rd Qu.:138.0   3rd Qu.:22.75  
##               Max.   :30   Max.   :140.0   Max.   :30.00
```

```r
names(andy)
```

```
## [1] "Patient.Name" "Age"          "Weight"       "Day"
```

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:

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

```
## [1] 140
```

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

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

```
## [1] 135
```

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

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

```
## [1] 135
```

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

```
## [1] 135
```

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

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

```
## [1] 135
```

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:

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

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

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

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

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.

Let's take the output of `list.files()` and store it:

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

```
## [1] "Andy.csv"  "David.csv" "John.csv"  "Mike.csv"  "Steve.csv"
```

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:

```r
files[1]
```

```
## [1] "Andy.csv"
```

```
files[2]
```

```
## [1] "David.csv"
```

```
files[3:5]
```

```
## [1] "John.csv"  "Mike.csv"  "Steve.csv"
```

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

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

```
## Warning in file(file, "rt"): cannot open file 'John.csv': No such file or
## directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

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
```

```
## [1] "diet_data/Andy.csv"  "diet_data/David.csv" "diet_data/John.csv"
## [4] "diet_data/Mike.csv"  "diet_data/Steve.csv"
```

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

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

```
##   Patient.Name Age Weight Day
## 1         John  22    175   1
## 2         John  22    175   2
## 3         John  22    175   3
## 4         John  22    175   4
## 5         John  22    175   5
## 6         John  22    175   6
```

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:

```r
head(andy_david)
```

```
##   Patient.Name Age Weight Day
## 1         Andy  30    140   1
## 2         Andy  30    140   2
## 3         Andy  30    140   3
## 4         Andy  30    139   4
## 5         Andy  30    138   5
## 6         Andy  30    138   6
```

```r
tail(andy_david)
```

```
##    Patient.Name Age Weight Day
## 55        David  35    203  25
## 56        David  35    203  26
## 57        David  35    202  27
## 58        David  35    202  28
## 59        David  35    202  29
## 60        David  35    201  30
```

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.

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

```
##    Patient.Name Age Weight Day
## 25         Andy  30    135  25
## 55        David  35    203  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:

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

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

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]))
}
```

```
## Error in rbind(dat, read.csv(files_full[i])): object 'dat' not found
```

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)
```

```
## 'data.frame':    150 obs. of  4 variables:
##  $ Patient.Name: Factor w/ 5 levels "Andy","David",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age         : int  30 30 30 30 30 30 30 30 30 30 ...
##  $ Weight      : int  140 140 140 139 138 138 138 138 138 138 ...
##  $ Day         : int  1 2 3 4 5 6 7 8 9 10 ...
```

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)
```

```
## 'data.frame':    30 obs. of  4 variables:
##  $ Patient.Name: Factor w/ 1 level "Steve": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age         : int  55 55 55 55 55 55 55 55 55 55 ...
##  $ Weight      : int  225 225 225 224 224 224 223 223 223 223 ...
##  $ Day         : int  1 2 3 4 5 6 7 8 9 10 ...
```

```
head(dat2)
```

```
##   Patient.Name Age Weight Day
## 1        Steve  55    225   1
## 2        Steve  55    225   2
## 3        Steve  55    225   3
## 4        Steve  55    224   4
## 5        Steve  55    224   5
## 6        Steve  55    224   6
```

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)
```

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

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)
```

```
## [1] 190
```

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.

```
dat_30 <- dat[which(dat[, "Day"] == 30),]
dat_30
```

```
##     Patient.Name Age Weight Day
## 30          Andy  30    135  30
## 60         David  35    201  30
## 90          John  22    177  30
## 120         Mike  40    192  30
## 150        Steve  55    214  30
```

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

```
## [1] 192
```

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)
```

```
## [1] 197.5
```

```
weightmedian("diet_data", 4)
```

```
## [1] 188
```

```
weightmedian("diet_data", 17)
```

```
## [1] 198
```

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.

FAQs

I thought I would pull together some frequently asked questions regarding Programming Assignment 1 that regularly show up on the forums. If you run into a problem you can't solve while working on assignment 1, take a look through this section.

1) My code runs fine and my answers match the sample output, but whenever I try to submit, I get a message telling me that my answer is incorrect. You're not submitting via the Coursera website are you? You need to re-read the Assignment 1 instructions (all of them).

Instead of submitting via the website, you need to use the `submit()` script. A link and more detailed instructions are included in the "Grading" section of the assignment 1 instructions.

If you're using the `submit()` script and still getting this error, double check to make sure you're not printing the results rather than returning them. In other words, the final line of your code should not contain `print()`. Use `return()` instead (you may not have to do this as R automagically returns the final line of a function).

2) Do I need to round my answers to match the sample output? No. You don't need to do any rounding of your results to pass the submission tests.

3) I only see 3 parts to the assignment. What are these 10 parts listed on the assignment page? You're correct that there are only 3 parts to assignment 1. The 10 parts could probably be more accurately described as tests. The `submit()` script will run your code with a variety of different parameters to test it. If there are issues with your function, it may only pass some of the tests.

4) My `pollutantmean()` passes the first 3 tests, but fails the 4th with the error message: "Error in pollutantmean("specdata","nitrate") : argument"id" is missing, with no default" You didn't assign a default value to `id`. The first line of your function should look exactly like the one in the instructions: `pollutantmean <- function(directory, pollutant, id = 1:332) {`

5) I get an error stating "unexpected '>'" or "unexpected '{'". You probably have an open `(` somewhere in your code. Double check it with a fine tooth comb to make sure you've closed all of your `()`, `{}`and `[]`.

6) My code seems to work but my answers don't match the sample output. Are you calculating the mean value for each file and then taking the mean of those means? That's not the correct approach. You need to combine all of the relevant data into a single data frame or vector and take the mean of *that*.

```
a <- c(1:5)
b <- c(1:10)
c <- c(10:15)

mean(c(a,b,c))
[1] 6.904762

mean(c(mean(a),mean(b),mean(c)))
[1] 7
```

You want the first approach, not the second.

7) My function seems to work when `id` is a single value but I get the following error message when it's something like `70:72`: "In pollutant1$ID == 1:332 : longer object length is not a multiple of shorter object length". You probably have a line of code that looks something like this: `dat_subset <- dat[which(dat[ , "ID"] == id), ]`

Subsetting by `ID` works when `id` is a vector of length 1. However, when `id = 1:10` for example, you have a problem. The issue goes back to the SWIRL example (and maybe lecture?) regarding how R handles vectors of differing lengths.

An example:

```
> x <- 1:10
> y <- 1:5
> x + y
 [1]  2  4  6  8 10  7  9 11 13 15
```

Each value of y gets added to x. But because y is shorter than x, after adding 5+5, R starts over from the beginning of y and adds 6+1, 7+2, etc.

That's what is happening with the subset when id is a vector longer than 1. The first row gets compared to the first value of id. The second row gets compared to the second value of id, etc. It repeats id until it gets to the end of the vector or data frame you're subsetting. The warning is telling you that the length of dat*nitrateordat*sulfate is not divisible by the length of id.

Essentially, there are 2 options to solve this. The first is to not use a subset for `id` at all. You presumably already have a loop in your code, so instead of combining all 332 files together, why not use that loop to combine only the files specified by `id` in the first place?

The other alternative is to replace the `==` with `%in%`. In this case, the %in% operator will check each value of `id` against every value in the `ID` column, which is what you want. The downside to this approach is that it will probably be very, very slow if you've followed the tutorial example to create `pollutantmean()`.

8) How do I subset for either `nitrate` or `sulfate` when I calculate the mean? If you wanted to subset nitrate, you would do that with `dat[, "nitrate"]`. Likewise you would use `dat[, "sulfate"]` for sulfate. When the function gets called you'll have something like: `pollutantmean(directory = "specdata", pollutant = "nitrate", id = 1:332)`.

So if you have either `pollutant = "nitrate"` or `pollutant = "sulfate"`, what would you put in place of `"sulfate"` and `"nitrate"` in subsetting examples above so that it would work in either case?

9) I'm subsetting my data frame using `dat$pollutant` but it doesn't seem to be working. Recall from the lectures that $ makes R look for a literal name match. That's not what you want. You want to subset by the value of pollutant (either "sulfate" or "nitrate"), not by "pollutant" since you don't have a column by that name. So, you need to use brackets instead of $.

That could be either something like [[pollutant]] or [, pollutant]

10) My code runs for a really long time (especially on submission #4)

This is due to the way that the loop is being used in this practice assignment. The approach I'm showing for building the data frame is suboptimal. 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.

So for those of you that do want to see a better way to create a dataframe. . . .

As I said before, the main issue with the approach above is growing an object inside of a loop by copying and recopying it. It works, but it's slow and inefficient. 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)
```

```
##    Length     Class      Mode
##         5 character character
```

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

```
##      Length Class  Mode
## [1,] 0      -none- NULL
## [2,] 0      -none- NULL
## [3,] 0      -none- NULL
## [4,] 0      -none- NULL
## [5,] 0      -none- NULL
```

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

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

```
## List of 5
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "Andy": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 30 30 30 30 30 30 30 30 30 30 ...
##   ..$ Weight      : int [1:30] 140 140 140 139 138 138 138 138 138 138 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "David": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 35 35 35 35 35 35 35 35 35 35 ...
##   ..$ Weight      : int [1:30] 210 209 209 209 209 209 209 208 208 208 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "John": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 22 22 22 22 22 22 22 22 22 22 ...
##   ..$ Weight      : int [1:30] 175 175 175 175 175 175 175 175 175 175 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "Mike": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 40 40 40 40 40 40 40 40 40 40 ...
##   ..$ Weight      : int [1:30] 188 188 188 188 189 189 189 189 189 189 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "Steve": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 55 55 55 55 55 55 55 55 55 55 ...
##   ..$ Weight      : int [1:30] 225 225 225 224 224 224 223 223 223 223 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

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 this is functionally identical to using `lapply`.

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

```
## List of 5
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "Andy": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 30 30 30 30 30 30 30 30 30 30 ...
##   ..$ Weight      : int [1:30] 140 140 140 139 138 138 138 138 138 138 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "David": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 35 35 35 35 35 35 35 35 35 35 ...
##   ..$ Weight      : int [1:30] 210 209 209 209 209 209 209 208 208 208 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "John": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 22 22 22 22 22 22 22 22 22 22 ...
##   ..$ Weight      : int [1:30] 175 175 175 175 175 175 175 175 175 175 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "Mike": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 40 40 40 40 40 40 40 40 40 40 ...
##   ..$ Weight      : int [1:30] 188 188 188 188 189 189 189 189 189 189 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame':    30 obs. of  4 variables:
##   ..$ Patient.Name: Factor w/ 1 level "Steve": 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ Age         : int [1:30] 55 55 55 55 55 55 55 55 55 55 ...
##   ..$ Weight      : int [1:30] 225 225 225 224 224 224 223 223 223 223 ...
##   ..$ Day         : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

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 or months 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're talking about.

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

```r
str(tmp[[1]])
```

```
## 'data.frame':    30 obs. of  4 variables:
##  $ Patient.Name: Factor w/ 1 level "Andy": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age         : int  30 30 30 30 30 30 30 30 30 30 ...
##  $ Weight      : int  140 140 140 139 138 138 138 138 138 138 ...
##  $ Day         : int  1 2 3 4 5 6 7 8 9 10 ...
```

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

```
## [1] 1 2 3 4 5 6
```

One way to do this would be to manually `rbind` everything together:

```r
output <- rbind(tmp[[1]], tmp[[2]], tmp[[3]], tmp[[4]], tmp[[5]])
str(output)
```

```
## 'data.frame':    150 obs. of  4 variables:
```

```
##  $ Patient.Name: Factor w/ 5 levels "Andy","David",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age         : int  30 30 30 30 30 30 30 30 30 30 ...
##  $ Weight      : int  140 140 140 139 138 138 138 138 138 138 ...
##  $ Day         : int  1 2 3 4 5 6 7 8 9 10 ...
```

But of course there's a better way. We can use a function called `do.call()` to combine `tmp` into a single data frame with much less typing. `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)
```

```
## 'data.frame':    150 obs. of  4 variables:
##  $ Patient.Name: Factor w/ 5 levels "Andy","David",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age         : int  30 30 30 30 30 30 30 30 30 30 ...
##  $ Weight      : int  140 140 140 139 138 138 138 138 138 138 ...
##  $ Day         : int  1 2 3 4 5 6 7 8 9 10 ...
```

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.