R Basics: Datasets, Variables, Subsets, If-Else Statements, Lists, and For-Loops

Zach Branson

January 26, 2022

Contents

This document outlines how to do basic tasks in `R` that will be useful for 36-318, including the first homework. Those already very familiar with `R` (e.g., those who have taken 36-350) may not need this document at all, but it nonetheless serves as a review and resource for you.

Loading Data into R

Throughout 36-318, we will use the `read.csv()` function to read/load datasets into `R`. For example, here is how we would load a basic dataset that was used as an example in class:

```
smokingData = read.csv("https://raw.githubusercontent.com/zjbranson/318Spring2022/main/smokingDataExampl
```

As you can see above, we inserted a URL into `read.csv()`; datasets will always be made available via URL in 36-318. There are many other ways to load datasets in `R`, such as downloading a .csv file on your computer and referencing that file in `read.csv()`.

Classes and Working with `data.frames`

When a variable is loaded/defined in `R`, it will always have a "class" that determines how that variable will behave and be used. Notice that we defined the variable `smoking` above; it is a dataset, as we can tell by seeing that it has class `data.frame`:

```
class(smokingData)
```

```
## [1] "data.frame"
```

Datasets (i.e. `data.frames`) can be envisioned as tables/matrices, where subjects are indexed by rows and variables measured for each subject are indexed by columns. Here are some functions that are useful for initially exploring a dataset:

- `dim()`: Gives you the number of rows and columns in the dataset. For example, we can see that the `smoking` dataset has 60 subjects and 3 variables:

```
dim(smokingData)
```

```
## [1] 60  3
```

- `head()` and `tail()`: Gives you the first or last (respectively) rows of a dataset (6 by default, but you can change the argument `n` within these functions to give you more/fewer rows).

```
head(smokingData)
```

```
##   smoking health.c health.t
## 1     yes 56.18480 58.18480
## 2     yes 55.37205 57.37205
## 3     yes 57.09584 59.09584
## 4     yes 56.76091 58.76091
## 5     yes 56.06631 58.06631
## 6     yes 56.83162 58.83162
```

```
tail(smokingData)
```

```
##    smoking health.c health.t
## 55      no 60.30665 62.30665
## 56      no 60.16323 62.16323
## 57      no 60.05195 62.05195
## 58      no 59.05387 61.05387
## 59      no 60.60397 62.60397
## 60      no 61.56781 63.56781
```

- `names()`: Gives you the variable names in the dataset. We can already see above that the variable names are "smoking", "health.c", and "health.t"; this can be validated with `names()`:

```
names(smokingData)
```

```
## [1] "smoking"  "health.c" "health.t"
```

Indexing Datasets

Sometimes it's useful to "grab" certain rows or columns of a dataset. For example, the following code grabs the first row and/or second column of the dataset:

```
#this is the first subject
smokingData[1,]
```

```
##   smoking health.c health.t
## 1     yes  56.1848  58.1848
```

```
#this is the second variable for every subject
smokingData[,2]
```

```
##  [1] 56.18480 55.37205 57.09584 56.76091 56.06631 56.83162 53.46733 55.55429
##  [9] 58.22430 58.05051 58.13734 56.56691 57.78944 57.63470 56.66121 56.19328
## [17] 56.10521 55.72089 55.97153 57.17113 56.68300 57.05455 57.27918 57.46492
## [25] 57.38487 56.99501 57.29257 54.86657 56.68733 57.13599 60.39472 59.93358
## [33] 59.54058 61.00571 59.16430 60.55080 59.89579 58.62292 58.10977 61.14661
## [41] 59.94557 59.21682 58.76224 59.03517 61.53693 60.52921 58.95508 57.96988
## [49] 58.86009 59.86850 59.37188 60.41342 58.87585 59.10812 60.30665 60.16323
## [57] 60.05195 59.05387 60.60397 61.56781
```

```
#this is the first subject's second variable
smokingData[1,2]
```

```
## [1] 56.1848
```

In general, brackets `[]` are used to index objects in `R`. For datasets and tables, the first number denotes the rows, and the second number denotes the columns. Note that we need to put a comma between the row number and column number, even if one of them isn't specified.

Referencing and Working with Variables in Datasets

To reference or "grab" individual variables within a dataset, you use the dollar sign notation `datasetName$variableName`:

```
smokingData$smoking
```

```
##  [1] "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes"
## [13] "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes"
## [25] "yes" "yes" "yes" "yes" "yes" "yes" "no"  "no"  "no"  "no"  "no"  "no"
## [37] "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"
## [49] "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"  "no"
```

As we can see above, if we reference an individual variable in a dataset, `R` will by default print out every observation for that variable; this isn't too bad for this small dataset, which only has 60 observations, but you won't want to do this for a big dataset.

Again we can use `class()` to check the "class" of each variable:

```
class(smokingData$smoking)
```

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

```
class(smokingData$health.c)
```

```
## [1] "numeric"
```

```
class(smokingData$health.t)
```

```
## [1] "numeric"
```

Here are some useful functions for working with individual variables, based on their class:

- `table()`: Displays a contingency table for the variable; i.e., displays the number of observations that have each unique value of the variable. This is particularly useful for categorical variables like `smoking`;

we can see that there are 30 subjects who smoke and 30 who don't:

```
table(smokingData$smoking)
```

```
##
##  no yes
##  30  30
```

- `mean()` and `var()`: Gives the sample mean and variance of quantitative variables like `health.c`. Sample means and variances will come up all the time in 36-318, so these will be very useful.

```
mean(smokingData$health.c)
```

```
## [1] 58.21608
```

```
var(smokingData$health.c)
```

```
## [1] 3.357314
```

- `summary()`: Gives several useful summary statistics for individual variables; this is typically used for quantitative variables:

```
summary(smokingData$health.c)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   53.47   56.81   58.12   58.22   59.88   61.57
```

Indexing Variables, and Details about Vectors

Similar to datasets, sometimes it's useful to "grab" certain observations for a variable. First, it's worth noting that individual variables are considered vectors, in that they can be envisioned as one long stream of values, rather than as an object with rows/columns. Thus, unlike datasets, we only need one number to index individual vectors/variables:

```
#this is the second value of health.c;
#i.e., health.c for the second subject
smokingData$health.c[2]
```

```
## [1] 55.37205
```

It's useful to know that we can also use the colon notation `x:y` to produce a vector of numbers ranging from `x` to `y`:

```
1:5
```

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

This is useful for grabbing a sequence of observations:

```
#this is the first through fifth values of health.c
smokingData$health.c[1:5]
```

```
## [1] 56.18480 55.37205 57.09584 56.76091 56.06631
```

Finally, it's worth noting that you can create a new vector using `c()` notation. For example, the following code grabs the first, second, and fourth observations of `health.c`:

```
smokingData$health.c[c(1, 2, 4)]
```

```
## [1] 56.18480 55.37205 56.76091
```

Defining New Variables

So far we have been referring to variables within a dataset, thereby requiring the dollar sign notation. We can also define new variables outside of a dataset. In this example dataset, `health.c` refers to the control potential outcomes $Y(0)$, and `health.t` refers to the treatment potential outcomes $Y(1)$. Thus, we may be interested in the difference between these potential outcomes:

```
tau = smokingData$health.t - smokingData$health.c
```

The above code defines a new variable, `tau`, as the difference between the treatment and control potential outcomes. Then, we can use functions like `mean()` and `var()` on this new variable:

```
mean(tau); var(tau)
```

```
## [1] 2

## [1] 0
```

Notice that I used a semi-colon to place two different bits of code on the same line. We can see that the mean of `tau` is 2 and the variance is 0. This must mean that `tau` is the same for every subject! Indeed, we can validate this with the `table()` function:

```
table(tau)
```

```
## tau
##  2
## 60
```

This shows why it's useful to explore a variable with several functions; if we had only used `mean()`, we could have concluded that the average treatment effect is 2, but we couldn't have concluded that the treatment effect was the same for every subject.

You can also define new variables within a dataset using the dollar sign notation. For example, if I had written `smokingData$tau =` instead of just `tau =`, the new variable `tau` would be "attached" as a new variable in the dataset.

If-Else Statements and Boolean Expressions

In 36-318 we'll be working a lot with binary variables, where an event will depend on the value of that binary variable. For example, we'll observe the treatment potential outcome $Y(1)$ if the treatment is $Z = 1$, and otherwise we'll observe the control potential outcome $Y(0)$ if the treatment is $Z = 0$. In other words, *if* $Z = 1$, we'll see $Y(1)$; otherwise (or *else*), we'll see $Y(0)$.

Thus, if-else statements will be very useful in 36-318. In short, if-else statements allow us to define new variables in the form of "if [this], then this value; otherwise, [that] value." For example, the following code defines a new variable, `smoking.num`, which is 1 if someone smokes and 0 otherwise.

```
smoking.num = ifelse(smokingData$smoking == "yes", 1, 0)
```

The `ifelse()` function takes three arguments. The first argument needs to be a TRUE-or-FALSE "test"; the line `smokingData$smoking == "yes"` will yield TRUE if `smoking` is "yes" and will yield FALSE otherwise. The second argument denotes the value that `smoking.num` should take if the "test" produces TRUE, and the third argument denotes the value that `smoking.num` should take if "test" produces FALSE. Thus, now we have a variable of 30 1s and 30 0s:

```
table(smoking.num)
```

```
## smoking.num
##  0  1
## 30 30
```

This is particularly useful within the context of 36-318, because we'll always refer to the binary treatment variable Z in a 0-or-1 fashion. Thus, even though the treatment variable won't always be defined in a 0-or-1 fashion in the original dataset (as was the case with `smoking`), we'll often have to define a new variable like `smoking.num` to get the appropriate 0-or-1 treatment variable Z.

If you want to be clever: Note that you can "nest" multiple `ifelse()` functions within each other by putting another `ifelse()` as the third argument in the first `ifelse()`. This way, you can consider non-binary conditions.

More on Boolean Expressions

I want to give a few more details about the idea of a TRUE-or-FALSE test in `R`, because we'll be using these a lot in 36-318. In the above `ifelse()` code, **note that there are two equal signs!** A common mistake is to use a single equal sign, which *won't* yield TRUE or FALSE. Statements in the form of TRUE or FALSE are known as boolean expressions. There are many boolean expressions in `R`; here are a few useful ones (besides `==`, which is an "is equal to" boolean expression):

- Is Less-Than or Greater-Than: This is expressed as `<` and `>` respectively. You can express "less than or equal to" as `<=` (and analogously for `>`). This is useful if, say, you want to test whether a variable is less than a certain value. For example, earlier we found that the median of `health.c` was 58.12; the following code allows us to pinpoint which subjects have a `health.c` less than 58.12:

```
smokingData$health.c < 58.12
```

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

```
table(smokingData$health.c < 58.12)
```

```
## 
## FALSE  TRUE 
##    30    30
```

- Not Equal To: This is expressed as `!=`. Thus, this acts as the opposite of `==`. In general, you can put the exclamation point `!` in front of a boolean expression to "flip" the expression (i.e. to say "not [this]" instead).

- And: This is expressed as `&`. This allows you to put two different boolean expressions together, such that together they'll only yield TRUE if both the first expression *and* the second expression yield TRUE. For example, maybe we want to know which subjects are smokers and have `health.c` less than 58.12:

```
smokingData$smoking == "yes" & smokingData$health.c < 58.12
```

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

```
table(smokingData$smoking == "yes" & smokingData$health.c < 58.12)
```

```
##
```

```
## FALSE  TRUE
##    32    28
```

The above must mean that there are two non-smokers with `health.c` less than 58.12.

- Or: This is expressed as `|`. This allows you to put two different boolean expressions together, such that together they'll yield TRUE if either the first expression *or* the second expression yield TRUE. For example, maybe we just want to know which subjects are smokers *or* have `health.c` less than 58.12:

```
smokingData$smoking == "yes" | smokingData$health.c < 58.12
```

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

```
table(smokingData$smoking == "yes" | smokingData$health.c < 58.12)
```

```
##
## FALSE  TRUE
##    28    32
```

Subsetting Datasets

One concept that will be extremely useful for 36-318 is subsetting datasets, i.e., dividing a dataset into smaller datasets based on specified criteria. For example, it will be ubiquitous for us to divide data into treatment and control groups. Here is how we would do this for this example dataset using the `subset()` function:

```
smokingData.treatment = subset(smokingData, smoking == "yes")
smokingData.control = subset(smokingData, smoking == "no")
```

The first argument takes in a dataset, and the second argument takes in a boolean expression, such that the function will return the subset of the data for which the boolean expression is TRUE. Note that we don't need the dollar sign notation for the boolean expression, because `subset()` "knows" which dataset you are referring to via the first argument.

Thus, we now have two smaller datasets, one for the treatment group and one for the control group:

```
class(smokingData.treatment); head(smokingData.treatment)
```

```
## [1] "data.frame"
```

```
##   smoking health.c health.t
## 1     yes 56.18480 58.18480
## 2     yes 55.37205 57.37205
## 3     yes 57.09584 59.09584
## 4     yes 56.76091 58.76091
## 5     yes 56.06631 58.06631
## 6     yes 56.83162 58.83162
```

```
class(smokingData.control); head(smokingData.control)
```

```
## [1] "data.frame"
```

```
##    smoking health.c health.t
## 31      no 60.39472 62.39472
## 32      no 59.93358 61.93358
## 33      no 59.54058 61.54058
## 34      no 61.00571 63.00571
## 35      no 59.16430 61.16430
## 36      no 60.55080 62.55080
```

The above process is particularly useful if we want to, say, compute a sample mean within a treatment group specifically and a sample mean within a control group specifically.

You can also use the `subset()` function to "select" certain variables within a dataset; for example, this line of code defines a dataset that only has the potential outcomes:

```r
smokingData.outcomes = subset(smokingData, select = c(health.c, health.t))
dim(smokingData.outcomes); head(smokingData.outcomes)
```

```
## [1] 60  2
```

```
##   health.c health.t
## 1 56.18480 58.18480
## 2 55.37205 57.37205
## 3 57.09584 59.09584
## 4 56.76091 58.76091
## 5 56.06631 58.06631
## 6 56.83162 58.83162
```

Here I've used `select` as the second argument instead, which is specified as a vector of variable names. Note that you *do not* need quotes for the variable names, and you also don't need the dollar sign notation.

Sometimes it's easier to "throw out" variables instead of "select" certain variables; this can also be done with `select`. For example, the following code "throws out" the `smoking` variable, which in this case is equivalent to selecting the potential outcomes:

```r
smokingData.outcomes = subset(smokingData, select = -c(smoking))
dim(smokingData.outcomes); head(smokingData.outcomes)
```

```
## [1] 60  2
```

```
##   health.c health.t
## 1 56.18480 58.18480
## 2 55.37205 57.37205
## 3 57.09584 59.09584
## 4 56.76091 58.76091
## 5 56.06631 58.06631
## 6 56.83162 58.83162
```

The only difference here is that I put a minus sign before the vector of variable names. This tells `subset` to throw out these variables, rather than select them.

Lists

Lists are flexible objects that allow you to store a sequence of objects in an organized way. For example, in the first homework, we will consider many hypothetical datasets that can be generated from an experimental design. If I gave you 1000 different datasets, how would you store them in an easy-to-reference way? The best way to do this is with a *list* of datasets, where the first "item" in the list is the first dataset, the second item the second dataset, and so on.

As an example, let's say we have a second dataset, `smokingData.alt`, which is exactly the same as `smokingData`, but with the rows shuffled:

```
smokingData.alt = smokingData[sample(1:nrow(smokingData)),]
head(smokingData); head(smokingData.alt)
```

```
##   smoking health.c health.t
## 1     yes 56.18480 58.18480
## 2     yes 55.37205 57.37205
## 3     yes 57.09584 59.09584
## 4     yes 56.76091 58.76091
## 5     yes 56.06631 58.06631
## 6     yes 56.83162 58.83162
```

```
##    smoking health.c health.t
## 60      no 61.56781 63.56781
## 1      yes 56.18480 58.18480
## 21     yes 56.68300 58.68300
## 32      no 59.93358 61.93358
## 33      no 59.54058 61.54058
## 38      no 58.62292 60.62292
```

We can store these datasets together as a list with `list()`:

```
datasets = list(smokingData, smokingData.alt)
```

Now, if we want to refer to a particular "item" within the list (in this case a dataset), we use the following notation:

```
#this is the head of the first dataset in the list
head(datasets[[1]])
```

```
##   smoking health.c health.t
## 1     yes 56.18480 58.18480
## 2     yes 55.37205 57.37205
## 3     yes 57.09584 59.09584
## 4     yes 56.76091 58.76091
## 5     yes 56.06631 58.06631
## 6     yes 56.83162 58.83162
```

Notice that this uses the double bracket notation `[[]]`; this notation is specific to indexing lists.

As an alternative to defining a list, you can also define an empty list, which you then add to one-by-one:

```
datasets = list()
datasets[[1]] = smokingData
datasets[[2]] = smokingData.alt
head(datasets[[1]])
```

```
##   smoking health.c health.t
## 1     yes 56.18480 58.18480
## 2     yes 55.37205 57.37205
## 3     yes 57.09584 59.09584
## 4     yes 56.76091 58.76091
## 5     yes 56.06631 58.06631
## 6     yes 56.83162 58.83162
```

The above is seemingly less efficient; it uses three lines of code instead of one to define the same list! However, this process is useful when you define lists using for-loops, which we discuss next.

For-Loops

For-loops are an essential programming technique to do a similar task many times. For example, as mentioned earlier, in the first homework, we will consider many hypothetical datasets that can be generated from an experimental design. Thus, we'll have to repeat an experimental design (e.g. complete randomization) over and over (e.g. 1000 times), and it would be absolutely evil for me to ask you to write 1000 lines of code to do this. Instead, it's much better to do this with a for-loop.

For example, here is code that shuffles the rows of `smokingData` many times, and for each time, stores the new dataset as an item in the list:

```r
#initalize the list
datasets = list()
#for the numbers 1 through 1000...
for(i in 1:1000){
  #define the item in the list
  datasets[[i]] = smokingData[sample(1:nrow(smokingData)),]
}
#look at the first item
head(datasets[[1]])
```

```
##    smoking health.c health.t
## 21     yes 56.68300 58.68300
## 9      yes 58.22430 60.22430
## 42      no 59.21682 61.21682
## 59      no 60.60397 62.60397
## 54      no 59.10812 61.10812
## 11     yes 58.13734 60.13734
```

```r
#look at the last item
head(datasets[[1000]])
```

```
##    smoking health.c health.t
## 55      no 60.30665 62.30665
## 37      no 59.89579 61.89579
## 32      no 59.93358 61.93358
## 9      yes 58.22430 60.22430
## 26     yes 56.99501 58.99501
## 19     yes 55.97153 57.97153
```

The notation `for(i in 1:1000)` essentially denotes "for each number `i` from 1 to 1000, do this." The "this" is contained in braces `{}`. Note that I use `i` within the list index notation `[[i]]`, such that the ith item of the list is defined. It's also very common to define elements of vectors (in which case you would use `[i]`), rows of matrices/datasets (in which case you would use `[i,]`), or columns of matrices/datasets (in which case you would use `[,i]`).

I've heard for-loops are dumb, and Professor Branson is recommending that we use for-loops. Does that mean Professor Branson is dumb?

To answer your question: Maybe. If you Google about for-loops in `R`, you'll get many posts (especially very old posts) suggesting that for-loops are very slow computationally, and thus they shouldn't be used. This is somewhat true; for-loops can be a bit slow, although this isn't necessarily specific to `R`. It was once the case that "more clever" code that uses functions like `apply()`, `lapply()`, etc. are faster than for-loops, but that has not been the case for years, even though that rumor persists; in fact, the family of `apply()` functions are essentially for-loops internally. There are standard ways that you can make for-loops more efficient, such as initializing objects and using lists, as we did above. See here for more details.

Of course, cleaner and more efficient code is best for any project, and I commend you if you want to make your code fast. However, we will not be grading for the efficiency of your code in 36-318, and for-loops will often do the job just fine; I don't expect 36-318 code to take an excruciatingly long time to run on your computer. In my experience, it takes a lot more time (and brain power) to write code that is more efficient than a for-loop, such that (at least for me) it in practice saves time to just write a for-loop, instead of spending a lot of time writing efficient code. Many statistical methods that we'll discuss in class are also, in practice, mathematically written as for-loops, such that programming a method as a for-loop is natural.

Does this mean that I am a dumb programmer? Maybe. But I am at least thoughtfully dumb, I think.