Introduction to R programming and data structures
Week 1, Lecture 02

Richard E.W. Berl

Spring 2019

Contents

Basic concepts in R (cont.)

Objects

Variables

```
a = 1
b = 2
c = 42
```

Reassignment

Just like we assign values to variables, we can reassign them and overwrite the old value.

```
a = a + b
a
```

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

```
a = a + b
a
```

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

```
a = a + b
a
```

```
## [1] 7
```

What is the value of `b`?

```
b
```

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

Why didn't `b` change?

We never reassigned `b` (`b = ...`), we only reassigned the value of `a`.

Scripts

Always work in scripts!

Open a new R script:

- Ctrl+Shift+N (Windows)
- Command+Shift+N (Mac)

Source: RStudio Keyboard Shortcuts

Commenting

```
# "This line is commented out."
"This line is not commented out."
```

```
## [1] "This line is not commented out."
```

Comment/uncomment line:

- Ctrl+Shift+C (Windows)
- Command+Shift+C (Mac)

An example of well-commented code:

```
# Load iris data
data(iris)

# View structure of iris data
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
# Subset iris data to Species versicolor
irisVe = subset(x=iris, subset=Species == "versicolor")

# Find correlation between sepal length and petal length in versicolor
cor(x=irisVe$Sepal.Length, y=irisVe$Petal.Length)  # Result: 0.754049
```

```
## [1] 0.754049
```

You can read about the iris data set by entering `?iris` in the console. Notice where R. A. Fisher's paper was published. We'll address this, and the dark side of the origins of statistics, in Week 4.

Packages

Packages are collections of functions and data sets useful for conducting different types of analyses.

You can get a list of packages that come installed with R using the command (more detail available by removing the `rownames()` wrapper):

```
rownames(installed.packages(priority="high"))
```

```
##  [1] "nlme"       "base"       "boot"       "class"      "cluster"
##  [6] "codetools"  "compiler"   "datasets"   "foreign"    "graphics"
## [11] "grDevices"  "grid"       "KernSmooth" "lattice"    "MASS"
## [16] "Matrix"     "methods"    "mgcv"       "nlme"       "nnet"
## [21] "parallel"   "rpart"      "spatial"    "splines"    "stats"
## [26] "stats4"     "survival"   "tcltk"      "tools"      "utils"
```

Additional packages can be installed using the `install.packages()` function, for example:

```
install.packages("tidyverse")
install.packages(c("reshape2","Hmisc"))
```

Notice that if you're installing multiple packages at once, the list of packages **must** be given as a vector (i.e. wrapped inside the `c()` function).

Load packages using the `library()` function:

```
library(ggplot2)
```

You can only load packages one at a time. Sometimes load order matters, because if two packages have functions with the same name, the last one loaded is the one R will assume you are referring to. To specify which one you want, you can put the package name before your function call, like so: `dplyr::arrange()`.

`install.packages()` has to have the package name in quotes. For `library()`, quotes are optional.

Lists of packages useful for specific tasks, such as machine learning or Bayesian inference, are available on the Task Views page of CRAN ("Comprehensive R Archive Network"), which is an official site run by the developers of R.

Data types and classes

Nominal (or Categorical)

"A nominal scale variable (also referred to as a categorical variable) is one in which there is no particular relationship between the different possibilities: for these kinds of variables it doesn't make any sense to say that one of them is"bigger" or "better" than any other one, and it absolutely doesn't make any sense to average them... In short, nominal scale variables are those for which the only thing you can say about the different possibilities is that they are different. That's it."

Source: Navarro, Section 2.2

Corresponding class in R:

- **Factor**

We'll come back to these later, when we get to data frames.

Ordinal (or Ranked)

"Ordinal scale variables have a bit more structure than nominal scale variables, but not by a lot. An ordinal scale variable is one in which there is a natural, meaningful way to order the different possibilities, but you can't do anything else."

Source: Navarro, Section 2.2

Corresponding class in R:

- **Ordered factor**

We'll have a whole section later on how to deal with ordinal data. Likert-type scales are ordinal.

Measurement

Continuous

"A continuous variable is one in which, for any two values that you can think of, it's always logically possible to have another value in between."

Source: Navarro, Section 2.2

Corresponding class in R:

- **Numeric**

This is what we've been working with already (see the variables `a`, `b`, and `c` above). You can perform mathematical operations on a numeric variable (sometimes called a "float," for "floating point number," the "point" part referring to the decimal point), and most statistical procedures are based on the assumption of working with a continuous (numeric) variable.

Discrete

"A discrete variable is, in effect, a variable that isn't continuous. For a discrete variable, it's sometimes the case that there's nothing in the middle."

Source: Navarro, Section 2.2

Corresponding classes in R:

- **Integer**

```
m = 5L
n = 1L
m + n
```

```
## [1] 6
```

Don't ask what the "L" stands for. Nobody knows. It's just how you specify you want a number as an integer rather than as a numeric variable. Most times it won't matter, and you'll be fine sticking with numeric variables.

- **Logical** (*sometimes*)

```
r = 300 > 1
r
```

```
## [1] TRUE
```

```
s = 5 == 4
s
```

```
## [1] FALSE
```

```
r + s
```

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

This behavior is useful for summing how many of something are present or fit a certain condition, using the `sum()` function.

Other types

See the following for a list of all basic classes included in R:

```
?methods::`character-class`
```

Character (or String)

```
x = "apple"
y = "orange"
z = "2.5"
x
```

```
## [1] "apple"
```

```
y
```

```
## [1] "orange"
```

```
z
```

```
## [1] "2.5"
```

Very common, used to represent plain text. If you want them to be meaningful as data (e.g. Condition A vs. Condition B), you should coerce the variable to a factor (see below).

What happens if we try to put them together?

```
x + y
```

```
## Error in x + y: non-numeric argument to binary operator
```

```
z + b
```

```
## Error in z + b: non-numeric argument to binary operator
```

These don't work, because they're apples and oranges.

Really, it's because you can't perform mathematical operations on characters (but see coercion, below). You *can* concatenate them using the `paste()` function if you want to combine them with something else, like this:

```
xy = paste(x, y)
```

```
zb = paste(z, b)
```

You can then split them apart again later, if needed. One way to do this is `strsplit()`:

```
strsplit(x=xy, split=" ", fixed=TRUE)
```

```
## [[1]]
## [1] "apple"  "orange"
```

Date and Time

In R these are:

- Date
- POSIXct
- POSIXlt

For example, see:

```
Sys.Date()
```

```
## [1] "2019-04-02"
```

```
str(Sys.Date())
```

```
##  Date[1:1], format: "2019-04-02"
```

```
Sys.time()
```

```
## [1] "2019-04-02 11:22:29 MDT"
```

```
str(Sys.time())
```

```
##  POSIXct[1:1], format: "2019-04-02 11:22:29"
```

These are a **nightmare** to deal with in R. Avoid doing analyses on dates and times whenever possible. If you must, look into the `lubridate` package to make things a bit easier.

Coercion

Remember `z + b` above? What if, instead of concatenating the strings, you wanted to treat `z` as a number and add it to `b`?

```
z
```

```
## [1] "2.5"
```

```
b
```

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

```
z + b
```

```
## Error in z + b: non-numeric argument to binary operator
```

This doesn't work because R is still treating `z` as a character that reads "2.5" rather than the number 2.5. But we can change a variable's class by "coercing" it to a different class. When R does this, it makes its best guess at how the values should be represented in the other class. With some coercions, such as this one, it is relatively straightforward:

```
z = as.numeric(z)
z
```

```
## [1] 2.5
```

```
z + b
```

```
## [1] 4.5
```

(Coercion functions are usually `as.` and the name of the class.)

Some coercions may not be so intuitive, and R will throw a warning when it can't figure out what to do:

```
steve = "Steve"
as.numeric(steve)
```

```
## Warning: NAs introduced by coercion
```

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

What do you mean you want to change "Steve" to a number? I don't know. R doesn't know either. Maybe you want to sum the numbers of each letter of the alphabet that spells "Steve"? Maybe you want to know what number "Steve" appears in a vector or data frame? Whatever it is, you'll need to be more precise in telling R what it is you want.

Bonus activity: What would the answer to the first question be? What is the sum of "Steve"?

NA and other constants

Above, you'll notice that R returned the value `NA` when we tried to coerce "Steve" to numeric. `NA` (without quotes or anything) is a special constant defined in R that indicates a missing value. This will be important later when we deal with real data, and it's important to define any missing values as `NA` so that R treats them appropriately.

There are a few other constants defined in R, which you can see listed by running this line:

```
?base::`Constants`
```

Try `letters` (compare with `LETTERS`), `pi`, or `month.name`. They come in handy occasionally (like for our "bonus activity" above).

Data structures

Vectors

A vector is the most basic data structure in R, and most other structures are made up of vectors. A vector is made up of one or more elements, up to (2^31)-1 (or about 2 billion).

(Note for R markdown users, you can have R evaluate a statement within a line of text like this: (2^31)-1 = 2147483647.)

You make a vector using the `c()` function, like so:

```
numVec = c(1,2,3,8,9,10)
numVec
```

```
## [1]  1  2  3  8  9 10
```

```
charVec = c("the","quick","brown","fox")
charVec
```

```
## [1] "the"   "quick" "brown" "fox"
```

In case you don't believe me, let's make sure that they are both vectors:

```
is.vector(numVec)
```

```
## [1] TRUE
```

```
is.vector(charVec)
```

```
## [1] TRUE
```

You can reference individual elements of a vector by using brackets (`[]`) and the number of the element (the first element is `[1]`):

```
numVec[6]
```

```
## [1] 10
```

```
charVec[2]
```

```
## [1] "quick"
```

Note that, if I want the number 8, it's the fourth element in my `numVec` vector, so I have to use `[4]`, not `[8]`:

```
numVec[4]
```

```
## [1] 8
```

```
numVec[8]
```

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

You can reassign individual elements of a vector this way:

```
numVec[1] = 10000
numVec
```

```
## [1] 10000     2     3     8     9    10
```

Or use the referenced value in some math:

```
numVec[4] / 2
```

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

You can combine vectors:

```
numVecB = c(75,100,500)
numVecC = c(numVec, numVecB)
numVecC
```

```
## [1] 10000     2     3     8     9    10    75   100   500
```

```
charVecB = c("jumps","over","the","lazy","dog")
charVecC = c(charVec, charVecB)
charVecC
```

```
## [1] "the"   "quick" "brown" "fox"   "jumps" "over"  "the"   "lazy"  "dog"
```

But a vector can only contain elements of one class. Let's check the classes of our first two vectors and try to combine them:

```
class(numVec)
```

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

```
class(charVec)
```

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

```
mixVec = c(numVec, charVec)
mixVec
```

```
##  [1] "10000" "2"     "3"     "8"     "9"     "10"    "the"   "quick"
##  [9] "brown" "fox"
```

Notice how it converted our numeric vector to character (we established that the other way around makes no sense), so that it could combine them like we asked it to.

You can make an empty vector of a specific class by using the function for that class:

```
emptyNumVec = numeric()
emptyNumVec
```

```
## numeric(0)
```

And then add stuff to it later:

```
emptyNumVec[1] = 2
emptyNumVec[2] = 4
emptyNumVec
```

```
## [1] 2 4
```

You can also perform mathematical operations on numeric vectors, the same way that you can with numeric variables. When you do so, it matches the first element of one vector with the first element of the other, and the second element of one vector with the second element of the other, *et cetera*:

```
numVec
```

```
## [1] 10000     2     3     8     9    10
```

```
numVecB
```

```
## [1]  75 100 500
```

```
numVec + numVecB
```

```
## [1] 10075   102   503    83   109   510
```

What happened here?

There were twice as many elements in `numVec` as in `numVecB`, so R looped through `numVecB` again when it was looking for more elements to add. So, R did `10000 + 75, 2 + 100, 3 + 500, 8 + 75, 9 + 100, 10 + 500`.

This is a cool feature (and it will throw a warning if one vector is not a multiple of the other), but be careful, because you may do it without meaning to, and end up with a mess. It's usually better to make sure both of your vectors are the same length beforehand.

You can check the length of a vector with... wait for it... `length()`:

```
length(numVec)
```

```
## [1] 6
```

```
length(numVecB)
```

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

Matrices

A matrix is a way to store data in a tabular (table) form, made up of rows and columns. There are whole fields of math dealing with matrices and matrix operations (matrix algebra and linear algebra), but we're not going to get into their mathematical properties. For you, and for the purposes of analyzing data in R, a matrix is made up of rows and columns, both of which are represented as vectors.

Let's make one:

```
firstM = matrix(data=c(4,81,5,84,
                       70,72,74,69,
                       29,34,27,3),
                nrow=3, ncol=4)
firstM
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    4   84   74   34
## [2,]   81   70   69   27
## [3,]    5   72   29    3
```

Wait, that didn't turn out the way we wanted. Look at how we listed our data how the matrix turned out. Let's look at the help documentation for `matrix()`:

```
help(matrix)
```

Or, as a helpful shortcut, you can also use:

```
?matrix
```

It looks like the `matrix()` function fills the data in by columns by default. But it has an argument (`byrow`) that we can use to make it fill by row, the way we listed the data. Let's try that.

```
firstM = matrix(data=c(4,81,5,84,
                       70,72,74,69,
                       29,34,27,3),
                nrow=3, ncol=4, byrow=TRUE)
firstM
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    4   81    5   84
## [2,]   70   72   74   69
## [3,]   29   34   27    3
```

Perfect. We can make sure the dimensions are correct with these functions:

```
dim(firstM)
```

```
## [1] 3 4
```

```
nrow(firstM)
```

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

```
ncol(firstM)
```

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

Now, we can reference values in a matrix the same way we referenced elements in a vector, but now we have two dimensions. So, we use two numbers inside the brackets. The first always refers to the row, and the second always refers to the column. So, if we want the value in row 2, column 4 (69), we use:

```
firstM[2,4]
```

```
## [1] 69
```

And we can replace it by reassignment, the same as an element in a vector. This time we'll ask it to subtract 1 from its current value:

```
firstM[2,4] = firstM[2,4] - 1
firstM[2,4]
```

```
## [1] 68
```

If we want to reference a whole row or column (vector), we only use the number for that row or column, but make sure to leave the comma so R knows which one you want. So, if we want to get the first row, and then the third column, we would use:

```
firstM[1,]
```

```
## [1]  4 81  5 84
```

```
firstM[,3]
```

```
## [1]  5 74 27
```

If we want multiple rows or columns, we can use vectors to tell R which ones we want.

```
firstM[1:3,c(1,4)]
```

```
##      [,1] [,2]
## [1,]    4   84
## [2,]   70   68
## [3,]   29    3
```

(The `1:3` syntax above is an easy way to create a vector that is a sequence of integers. This one is equvalent to `c(1,2,3)` or `seq(from=1, to=3, by=1)`.)

Data Frames

Data frames are unique to R. They are just like matrices (and most anything you can do to a matrix you can also do to a data frame), but each column has a label and a class assigned to it. This is because, in a data frame, each column is treated as a variable and each row is treated as an observation. You can reference these variable names instead of column numbers, and you can use them in formulas.

For this reason, it's usually much easier to work with data frames than with matrices when doing statistical analyses.

Let's make our matrix `firstM` into a data frame.

```
firstDF = data.frame(firstM)
firstDF
```

```
##   X1 X2 X3 X4
## 1  4 81  5 84
## 2 70 72 74 68
## 3 29 34 27  3
```

Our columns (variables) didn't have names, so R assigned them some. Let's fix that. We can reference the names of the rows or columns of a matrix or data frame with `rownames()` or `colnames()`, respectively.

```
colnames(firstDF)
```

```
## [1] "X1" "X2" "X3" "X4"
```

```
length(colnames(firstDF))
```

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

It's a character vector of length 4, so we need to replace it with something of the same type and length:

```
colnames(firstDF) = c("soil","climate","altitude","rating")
firstDF
```

```
##   soil climate altitude rating
## 1    4      81        5     84
## 2   70      72       74     68
## 3   29      34       27      3
```

Much better. Now we can reference those columns by name:

```
firstDF$soil
```

```
## [1]  4 70 29
```

```
firstDF$rating / firstDF$altitude
```

```
## [1] 16.8000000  0.9189189  0.1111111
```

Remember that these are vectors we're pulling out, so if we want a specific value from a column, we only have one dimension in that column vector to reference.

```
firstDF$altitude[3]
```

```
## [1] 27
```

But we can still use row and column numbers to reference specific values in the data frame, as well:

```
firstDF[1,4] - firstDF[1,1]
```

```
## [1] 80
```

We'll do much, much more work with data frames once we start working with data and running analyses.

Lists

A list is like a vector that can hold other types of objects, or even more lists, inside it. Check it out:

```
myList = list(numVec,
              charVec,
              list(c(1,2,3), c(3,2,1)),
              "Franz Liszt")
myList
```

```
## [[1]]
## [1] 10000     2     3     8     9    10
##
## [[2]]
## [1] "the"   "quick" "brown" "fox"
##
## [[3]]
## [[3]][[1]]
## [1] 1 2 3
##
## [[3]][[2]]
## [1] 3 2 1
##
##
## [[4]]
## [1] "Franz Liszt"
```

Just like a vector, we can use brackets to reference an element of a list:

```
myList[2]
```

```
## [[1]]
## [1] "the"   "quick" "brown" "fox"
```

But this returns another list, containing only the second element. If we want the object itself (a vector), we have to use double brackets.

```
myList[[2]]
```

```
## [1] "the"   "quick" "brown" "fox"
```

We can also reference elements within the referenced object by adding another set of brackets.

```
myList[[3]]
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 3 2 1
```

```
myList[[3]][[2]]
```

```
## [1] 3 2 1
```

Since lists can contain more lists, you could theoretically keep referencing things down many levels. But it wouldn't be good practice to store data like this, because it becomes very cumbersome to access.

Similar to lists are arrays, which are multi-dimensional matrices (matrices within matrices). These can sometimes be useful, and you can learn more about them by bringing up their help page: `?array`.

Functions

Functions are the bread and butter, or meat and potatoes, or whatever, of R. Almost every line of code you write will be used to run a function. Functions take one or more input arguments, and spit out some output which, depending on the function, could be a number, or a vector, or a data frame, or some other class of object.

For instance, let's use the base function `mean()` to calculate the mean of the first three variables we defined.

```
a
```

```
## [1] 7
```

```
b
```

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

```
c
```

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

```
mean(x=c(a, b, c))
```

```
## [1] 17
```

Common functions

Remember that the `help()` function (or `?`) pulls up the documentation for a function, as long as that function is in one of your installed and loaded packages. To get some practice using it, look up the help pages to learn about these functions that you will use all the time:

- `rep()`
- `seq()`
- `sum()`
- `max()` / `min()`
- `mean()`
- `sd()`
- `summary()`
- `str()`
- `class()`
- `paste()` / `paste0()`
- `nchar()`
- `length()`
- `dim()`
- `nrow()` / `ncol()`

- `dimnames()`
- `rownames()` / `colnames()`
- `head()` / `tail()`
- `unique()`
- `is.na()`
- `complete.cases()`
- `rm()`
- `View()` (*note the capital "V"*)

Defining a function

The basic structure of writing a function is `NAME = function(ARGUMENT, ARGUMENT, ...){STATEMENTS}`. To tell the function what to output, you use `return(OUTPUT)` before closing the curly bracket.

The general advice is, for anything you plan on doing more than once, put it in a function so that it's cleaner and more easily repeatable. That said, I have found in my own research that I very rarely use the exact same code twice even when performing the same kinds of analyses, because different types and formats of data always require different ways of dealing with them.

It is useful, however, to make a couple functions on your own to see functions in R work. If you ever want to see the guts of a function from an R library, all you have to do is forget to add the parentheses afterwards, like this one for calculating standard deviation:

```
sd
```

```
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x0000000018e4fcf8>
## <environment: namespace:stats>
```

On the second line, you can see–amongst the mess of checking classes–it takes the square root of the variance of a vector of numbers, just as it should.

(Some won't be helpful in that they just reference other functions or scripts.)

For practice, let's make our own function, one that adds the variable `a` to the variable `b`.

```
a_plus_b = function(a, b){
  aPlusB = a + b
  return(aPlusB)
}

a_plus_b(a=1, b=5)
```

```
## [1] 6
```

Why don't we just use `aPlusB` instead of the whole function?

```
aPlusB(a=1, b=5)
```

```
## Error in aPlusB(a = 1, b = 5): could not find function "aPlusB"
```

Why doesn't this work?

`aPlusB` is a variable, not a function. You can't pass other variables to it. Functions are defined to have input arguments (and always have `()` afterwards, even if there is no input) and output a result. Variables just hold a value or values for you, so sending it an input doesn't do anything.

Let's see what value is stored for the `aPlusB` variable.

```
aPlusB
```

```
## Error in eval(expr, envir, enclos): object 'aPlusB' not found
```

Why doesn't this work?

`aPlusB` is called a "local" variable, because it only exists inside the `a_plus_b` function and is removed as soon as the function is finished running. It doesn't have a value assigned to it in the "global" environment that we're working in. (Look in the "Environment" tab in RStudio.)

Operations

Also known as control-flow statements, you can see a list of the operations below by using `?Control`.

Conditionals

If

`if` statements are common in programming. In R, we use them to evaluate whether a statement is true and, if so, to execute one or more commands. Their structure is `if (CONDITION) {RESULT}`. The conditional statement should evaluate as a logical variable; that is, running it on its own should give a `TRUE` or `FALSE`. The whole condition always needs to be inside the parentheses. The result should be entirely within a set of curly brackets following the condition.

It usually helps visually to break it into multiple lines and have the result commands indented under the `if()` statement (RStudio does this formatting automatically), like so:

```
myValue = 100
critValue = 50
if (myValue > critValue) {
  print("Yes!")
}
```

```
## [1] "Yes!"
```

You can also add an `else{}` after the `if()` result to add an outcome to be executed when the `if()` statement resolves as `FALSE`:

```
myValue = 10
if (myValue > critValue) {
  print("Yes!")
} else {
  print("No!")
}
```

```
## [1] "No!"
```

Sometimes it is also helpful to be able to nest `if()` statements inside one another (but more than a couple levels of this can get confusing fast):

```
myValue = 50
if (myValue > critValue) {
  print("Yes!")
} else {
  if (myValue == critValue) {
    print("Maybe...")
  } else {
    print("No!")
  }
}
```

```
## [1] "Maybe..."
```

Which

`which()` is a function that can be used to find which values of an object meet a certain condition. For instance, let's look at the `rating` variable of `firstDF`:

```
firstDF
```

```
##   soil climate altitude rating
## 1    4      81        5     84
## 2   70      72       74     68
## 3   29      34       27      3
```

```
which(firstDF$rating > 35)
```

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

`which()` doesn't return the values themselves, it returns the a vector of the indices whose values match the condition. If you want to get the actual values, you need to take the vector of indices that `which()` produces and plug it in to the brackets to reference those elements of the `rating` vector:

```
firstDF$rating[which(firstDF$rating > 35)]
```

```
## [1] 84 68
```

This can also be used for subsetting data frames, which we will get into next week.

Iteration

Loops

Loops are another kind of statement common across programming languages. The general idea is to go through each thing in a list of things and execute a command for each one. The most useful one in R is a `for()` statement. The structure is similar to an `if()` statement, in that you have `for (VARIABLE in SEQUENCE) {RESULT}`. The difference is that, instead of a conditional statement inside the parentheses, you have an internal variable (which can be anything you want, but `i`, `j`, `k`... is conventional) that represents how far you are through your loop, and the sequence of things to loop through (represented as the starting value, a colon, and the ending value).

A simple exmaple is:

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

```
## [1] 1
## [1] 2
## [1] 3
```

```
## [1] 4
## [1] 5
```

This loop goes through the sequence of numbers 1 through 5, and for each one prints the value of `i`. In each loop, R changes the value of `i` to match the current number in the sequence, so referencing it in a command is a good way to see where you are (by printing it) or to reference a different value in an object every loop.

For example, let's imagine that `firstDF$altitude` is measured in meters and we want to convert it to feet. We could do this more easily by just multiplying the whole vector by 3.28084, but here we'll do it with a loop.

(There are often many different ways to accomplish the same result in R, and you can always do what works best for you even if it isn't the most straightforward solution.)

We also make a new, empty numeric variable to hold the converted values, called `altitudeFt`.

```
firstDF
```

```
##   soil climate altitude rating
## 1    4      81        5     84
## 2   70      72       74     68
## 3   29      34       27      3
```

```
length(firstDF$altitude)
```

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

```
altitudeFt = numeric()
for (i in 1:length(firstDF$altitude)) {
  altitudeFt[i] = firstDF$altitude[i] * 3.28084
}

altitudeFt
```

```
## [1]  16.40420 242.78216  88.58268
```

Remember that we can use the counting variable `i` in the `for()` loop to reference objects. Here, we set the loop to run for as many times as the length of the `firstDF$altitude` vector, and then used `i` to reference both the old vector and our new vector at the same time, so that their values end up corresponding with one another.

Once we have the `altitudeFt` variable, we can add it to our data frame by referencing it as a new column variable:

```
firstDF$altitudeFt = altitudeFt
firstDF
```

```
##   soil climate altitude rating altitudeFt
## 1    4      81        5     84   16.40420
## 2   70      72       74     68  242.78216
## 3   29      34       27      3   88.58268
```

Like `if()` statements, we can also nest `for()` loops. Here is an example, where `i` represents the row of `firstM` and `j` represents the column, to make the loop run through every value in the two-dimensional matrix. This loop also uses a variable called `tick` and increases it by one at some point in the loop.

```
tick = 1
for (i in 1:nrow(firstM)) {
  for (j in 1:ncol(firstM)) {
    firstM[i,j] = firstM[i,j] - tick
  }
  tick = tick + 1
}
```

```
firstM
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3   80    4   83
## [2,]   68   70   72   66
## [3,]   26   31   24    0
```

So when does `tick` increase and what does this do to our matrix?

The loop starts at row (`i`) 1, then loops through every column (`j`) of row 1, subtracting the value of `tick` from each element. Then, before advancing to row 2, it increases the value of `tick` by 1. Then, it moves to row 2 and loops through every column, subtracting `tick` (which now has a value of 2) from each element, increases `tick` by 1, and so on for the third row.

These kinds of structures are good for doing something to every value of a specific set of rows or columns in a matrix or data frame. Since they have two dimensions, you need two loops and two indicator values (`i` and `j`).

Two other loop structures are available in R, `while()` and `repeat()`. You can check them out. In my experience, they are much more useful in traditional programming and less so in R when you are manipulating and analyzing data. In addition, anything you could do using `while()` or `repeat()` can also be done with a `for()` loop, so at this point it's best to just concentrate on the more flexible `for()` loops.

Apply

`apply()` is actually not one function, but a family of functions each for a different data structure that includes `apply()`, `lapply()`, `sapply()`, `vapply()`, `mapply()`, `rapply()`, and `tapply()`. You can check the help file for each for more information. Essentially, they are a faster and more efficient way of looping through the rows or columns (referred to as "margins" in the help) of an object and applying a function to them.

Here is an example, in which we use the `apply()` function (which works on arrays, and matrices count as two-dimensional arrays, just like variables count as one-dimensional vectors) to loop through the columns of `firstM` and calculate the `sum()` of each column:

```
apply(firstM, 2, sum)
```

```
## [1]  97 181 100 149
```

Works great, and it's fast. But, to be honest, I never learned how to use the `apply()` functions well because `for()` loops have always made better sense to me for complex tasks and I haven't been in many situations where speed mattered enough to change `for()` loops to solutions that used the `apply()` family of functions. So, for the purposes of this course, be aware of them, and if you want or need to use an `apply()` function for a specific task, StackOverflow is a good place to find answers and example code.

(pdf / Rmd)