Week 4: Data and functions in R

Stat 201: Statistics I

Programming basics

Though this is not a programming class, there are two programming concepts we will need to understand to be able to do our work in R: variables and functions.

Variables

A variable is a name that points to, or "contains", some piece of data or information. When used in an R expression, the "value" of the variable, the data it points to, replaces the variable before an answer is determined. For example, if the variable `x` is assigned the number `2`, then the expression `5 + x` will give the answer `7`.

Variables, or rather the label or name of variables, can contain letter, numbers, underscores (`_`) and periods, though they can not begin with numbers. R is a case-sensitive language, so `x` and `X` are different variables, as are `avgRain` and `avgrain`.

To assign a value to a variable, the `<-` operator (less than and dash characters) is used. So, to assign the number `2` to the variable `x`, in either the console or an R script or markdown file type, followed by enter,

```
x <- 2
```

Then, the variable can be used instead of the number `2`.

```
5 + x
```

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

To see the value of a variable, simply type the variable name, followed by enter.

```
x
```

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

Functions

Functions are like variables, except instead of pointing to data, they point to a bit of R code. When a function is "called", usually with parameters, the code is executed and often some value is returned. Functions have the same naming rules as variables.

To call a function, type it's name followed by parentheses. Any parameters, information given to a function to work on, are listed within the parentheses separated by commas. Suppose we want the mean of a sample of data stored in the variable `my.data`. We call the `mean()` function with the data as a parameter.

```
mean(my.data)
```

```
## [1] 5.38384
```

If a function is simply called by itself, either in the console or when selected to run from a script file, whatever value is returned by the function is printed on the next line, as in the above example. However, the returned value can be stored in a variable for further calculations or other uses.

```
my.data.mean <- mean(my.data)
my.data.mean * 2
```

```
## [1] 10.76768
```

Function documentation

R has a built in system for access documentation for functions that can be used in your R code. The documentation will tell you what the function does, what paramters are accepted and what options there are for controlling the behavior of the function, what values if any are returned and, in most cases, examples of it's use.

To bring up the documention for a function, in the console type a question mark followed immediately (no space) by the function name. Thus, to access the documentation for the `mean()` function:

```
?mean
```

If you type `example(`*function name*`)` in the console, the example code which can be found at the bottom of the documentation is executed.

Passing parameters

The documentation page for `mean` gives the "usage" as follows:

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

The first line gives the most generic version of the function. You must give it something as a parameter (`x`). The ellipse (`...`) means that you can also pass additional parameters which the function might or might not use. The second version (default S3 method) is more interesting. Parameters listed without an equal sign (`=`) are required parameters. In this case, there is just one (`x`), but many functions will have several required parameters. Parameter with equal signs are option parameters. The value after the equal sign is the default value that will be used by the function if no other value is provided.

When calling a function, you can just pass in values, in order and separated by commas, for each parameter. Alternatively, you can "name" each value you pass to the function. In that case, you can enter them in any order you like. Or you provide the first few values, unnamed and in order, and then add any named parameters you need. This is the most common approach. Call the function with the required parameters unnamed, and then name any of the optional parameters you need. See the examples below. Don't worry about the details of the function calls yet.

```
# Find the mean of the varaible 'my.data',
#   parameters trim and na.rm keep their default values
mean(my.data)

# Find the mean, with trim of 0.1 and na.rm TRUE
mean(my.data, 0.1, TRUE)

# Same as above
mean(trim=0.1, na.rm=TRUE, x=my.data)

# Find the mean of my.data, na.rm TRUE,
#   with trim keeping its default value of 0
mean(my.data, na.rm=TRUE)
```

Data in R

Vectors

The fundamental data type in R is the vector, basically an ordered list of values. To create a vector, you can use the `c()` function (short for "concatenate", I imagine). The following sets a vector of the first four integers to the variable `x`:

```
x <- c(1,2,3,4)
```

We can also create a list of consecutive integers with a colon (:) between the lower and upper bounds of our desired list. The following command is equivalent to the first:

```
x <- 1:4
```

To see the contents of a variable, simply type its name.

```
x
```

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

Notice the 1 in brackets on the left. That is the index of the first item displayed for a row. If we have a long enough vector to print on a second row, we will see that value change.

```
x <- 1:25
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25
```

Even a single value is stored as a vector. Notice the index still displays on the output.

```
y <- 2+3
y
```

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

Working with vectors

Vectors can be used (almost) anywhere you would normally expect a single value to be used. In most cases operations or functions which expect a single value or pair of values, when given vectors, will do entry-wise calculations. For example, adding two vectors will add the first entries in the vectors, then the second entries, etc.

```
x <- 1:4
y <- 11:14

# We don't have to store results in a variable. An operation
#   without assignment will just output results.
x + y
```

```
## [1] 12 14 16 18
```

If presented with a vector and a single value, R will apply the single value to each entry of the vector.

```
# Square a vector (x still represents the same vector as above)
x ^ 2
```

```
## [1]  1  4  9 16
```

Actually, what is going on here is that when R encounters an operation on two vectors and one vector is shorter than the other, it will repeat copies of the shorter vector until it is as long as the longer vector. This can produce some confusing results if you are not expecting it, but it can be utilized to your advantage in some situations.

```
# Make alternating entries negative
x <- 1:10
x * c(1,-1)
```

```
##  [1]   1  -2   3  -4   5  -6   7  -8   9 -10
```

Of course, vectors can be used in more expected ways, in functions that operate on lists of values.

```
x.mean <- mean(x)
x.mean
```

```
## [1] 5.5
```

If we need to access individual or multiple items in a vector, we can use square brackets (`[]`) with an index number or numbers. To access multiple indexes, use a vector of values inside the brackets.

```
# Get the fifth integer (i.e. 5)
x[5]
```

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

```
# Get the third, fourth and sixth squares
(x^2)[c(3,4,6)]
```

```
## [1]  9 16 36
```

Data frames

We can think of a vector as one factor or variable (in the stats sense), or as one column in a data table. To work with a data table as a whole, we use data frames. We can create a data frame from already defined vectors.

```
x <- 1:4
x.sqr <- x^2

# Create columns by setting column names = vectors of equal length
squares.df <- data.frame(num=x, square=x.sqr)
squares.df
```

```
##   num square
## 1   1      1
## 2   2      4
## 3   3      9
## 4   4     16
```

The left-most numbers in the output are row numbers. Across the top are the column names we chose, with the values of the column below. While just typing the variable name is fine for our little data frame, most data sets will have hundreds, if not thousands, of rows and perhaps dozens of columns. It is impracticable to examine such data frames this way (though R will not print all of thousands of rows, it will stop printing after the first one thousand).

The first command when working with a data set will almost always be `str()` (for structure, not string as in many other languages). This can be used with any variable, but it is particularly useful with data frames.

```
str(squares.df)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ num   : int  1 2 3 4
##  $ square: num  1 4 9 16
```

As you can see `str` gives us the number of rows (`obs.`) and columns (`variables`), as well as the names of the column and the first few values in each column. If we want get a sense of how the data looks, without printing hundreds of rows, the `head{}` function will output the first 6 rows by default, or a specified number of rows with the optional `n=` parameter.

```
head(squares.df, n=2)
```

```
##   num square
## 1   1      1
## 2   2      4
```

Additionally, to view the data in a spreadsheet like format, try the `View(my.dataframe)` command (remember R is case-sensitive).

Accessing data in data frames

Now that we have our data in a data frame, we want to get it out so we can work with it. There are several ways to do this.

To retrieve a column of the data frame (a vector), we can use the `$` operator. The format of the operator is `data.frame$column.name`.

```
squares.df$square
```

```
## [1]  1  4  9 16
```

Like with vectors, we can also use square brackets, but now we are working with two dimensions. Both need to be included within the brackets, rows then columns separated with a comma.

```
# Get the third square. The squares are the second column.
#   So we want the 3rd row and 2nd column.
squares.df[3,2]
```

```
## [1] 9
```

With square brackets, we can use column names instead of numbers if we put the names within quotes (single or double). Note, we didn't use quotes around the column name when we used the dollar sign.

```
squares.df[3,'square']
```

```
## [1] 9
```

We can use multiple indices, like with vectors.

```
# Get the third and fourth squares
squares.df[c(3,4),'square']
```

```
## [1]  9 16
```

If we leave either the row or the column indicator blank, then all rows or all columns, respectively, will be included. We must still use the comma.

```
# The following are all equivalent
squares.df$square         # The square column
```

```
## [1]  1  4  9 16
```

```
squares.df[ , 2]          # all rows of the 2nd column
```

```
## [1]  1  4  9 16
```

```
squares.df[ , 'square']   # all rows of the square column
```

```
## [1]  1  4  9 16
```

Getting data

R has a number of built in data sets for use in testing code and demonstrations. I will be using these data sets in the following guides so you can use the example code without needing an external data file. You can see the list of data sets with the following command (I suppressed the output because the list is quite long):

```
data()
```

To use a data set, say the `chickwts` set:

```
# Make the chickwts data set available
data('chickwts')

# Examine the structure of chickwts
str(chickwts)
```

```
## 'data.frame':    71 obs. of  2 variables:
##  $ weight: num  179 160 136 227 217 168 108 124 143 140 ...
##  $ feed  : Factor w/ 6 levels "casein","horsebean",..: 2 2 2 2 2 2 2 2 2 2 ...
```

```
# Find the mean chick weight
mean(chickwts$weight)
```

```
## [1] 261.3099
```

You can find more information about a built-in data set with the question mark (i.e. `?chickwts`).

Of course, eventually we want to work with our own data. R can import data from may kinds of files (Excel, SPSS, SAS, etc.), but the most common and easiest to work with are csv file (comma separated values). Because I don't want to bother remembering all the different commands for handling different file types, if I have to work with a non-csv file, I will import into Excel and save it as a csv file.

```
my.data <- read.csv('...filepath...', header=TRUE)
```

The value of the first parameter will be the path to the location of the file, including the file name. If you are using R Studio (and this is reason enough to do so, in my opinion), after you type `read.csv` and the first parentheses and the first quote, hit the TAB key. A mini file browser will pop-up that will allow you to select the location of the file and insert the file path in the command. This can be a huge time saver.

The `read.csv` function has many optional parameters (see `?read.csv`) most of which you will never have to bother with. However, the `header` parameter is important. A `TRUE` value for `header` indicates that the first row of the file contains the names of the columns, rather than data. A false value means the file only contains data. If you are not sure which is the case with your file, you can open it in a text editor, or call `read.csv` with `header=FALSE` and then call `head(my.data)`. If the first row of values are names, then read in your data again with `header=T`. (Note: The values `TRUE` and `FALSE` can be abbreviated to `T` and `F`.)

Using code in R markdown

To include R code in R markdown documents, you must designate a section of the document as a code chunk. The start of a code chunk is three accent marks, or backticks (the key in the upper left conner of the keyboard, by the 1), then curly braces with the letter "r" within, all on it's own line. The end of a code chunk is just three accent marks, again on it's own line.

````
```{r}
# Put code here
```
````

Any output from the code chunk, anything that would be printed in the console, will be included in the PDF when the R markdown file is knit. Results from previous code chunks in the document, such as variables that are defined, will be available to use.

You can also use short code chunks within lines of text. So called inline chunks are specified by a single accent mark, an "r", code that will return a value, followed by a final accent mark. So, if you include the following in an R markdown document:

The mean of the data is `` `r mean(my.data)` `` units.

It will render as:

The mean of the data is 5.38384 units.

There are many options that can be included to control how code chunks are treated and displayed, but the stardard use works in many instances, including our homework assignments.