

## Editors, Arrays and loops in R

These notes are for someone who can use a computer generally but has not done any programming or done other serious computer work. Everything below is here because I've helped a student do it. Just move quickly through the things you already know.

### Scripts and editors

Typing at the command line can be frustrating. A *script* is a sequence of commands prepared in an editor and saved in a file. All you do from the command line is run the script. You use the editor to create and modify the script. The work cycle of programming, what programmers do as they're programming, is: edit, save, run, look at the output, decide what needs changing and do it again.

An *editor* is a program, like Microsoft Word, that allows you to create a file that contains text. But Word is inappropriate for programming (creating and modifying scripts). If you don't believe me, try it. Instead, you should find an editor designed for creating code (a script). I use `xcode`, which is an app that runs on my imac. Everyone who codes has a personal preference, and they're all different. If you do a web search on "what is the best editor for coding", you will start to get a feel for geeks arguing over features of code editors. It will help for you to see some of the issues they discuss. Some of the fancy ones, `emacs` being an extreme example, take a while to learn and may slow down your work in the beginning.

This handout will explain how to use the editor that comes with the R app on my imac. I believe/hope that this editor is available to anyone with the R app on any platform. This editor lacks many of the features the geeks like, such as syntax highlighting. This should not slow you down much if you're a beginner. If you're not a beginner, you probably already have a favorite editor – feel free to use that.

Before you start editing, you have to be aware of where in the computer *file system* the editor and the R command line will look for your files. The command `getwd()` [enter] (for "get working directory") returns the *path* to the *directory* (also called *folder*) the command line and the editor will use. For me, it went like this:

```
> getwd()
[1] "/Users/jg/Desktop/notes/MathFin2019/codes"
>
```

The path above says that there is a *top level* directory (folder) named `Users` on my computer. That directory has files in it. One of those files is the directory `jg` (my initials). The directory `jg` has in it a directory called `Desktop`.

Continuing, `Desktop` has `notes` (files where I prepare for classes), `notes` has `MathFin2019` (for this class), and `MathFin2019` has `codes`, where the R scripts (codes) for this class will go. The *path* is the sequence of directories `Users -> jg -> Desktop -> notes -> MathFin2019 -> codes`. It describes the location, in my computer file system, of the files I will create for this class. You should decide where you will put your R code files for this class and create a directory (folder) in the appropriate place.

The R app allows you to get around, to *navigate*, your file system. When I launch my R app, I get

```
> getwd()
[1] "/Users/jg"
>
```

This directory is too *high level* in my file system. Putting all my files in this directory would be chaos. It would be hard to find anything because there would be too many files. I need to go to a *subdirectory*, which is a file in `jg` that is a directory. One of these subdirectories is `Desktop`. The basic way to do this in R is illustrated here

```
> path = "/Users/jg/Desktop"
> setwd(path)
> getwd()
[1] "/Users/jg/Desktop"
>
```

The first line creates a character string (something in quotes) that is the path to the desired directory. (It won't work without the quotes – try it.) The second line uses `setwd()` (for “set working directory”) to add `Desktop` to the path. I created the top line `"/Users/jg/Desktop"` by pasting `Desktop` on the end of `/Users/jg`. I repeat this to go *down* to `notes`, which is a subdirectory of `Desktop`. With copy-paste I create `path = "/Users/jg/Desktop/notes"`, then the command `setwd(path)` [enter] puts me in the `notes` subdirectory. Eventually, I get to the desired directory `/Users/jg/Desktop/notes/MathFin2019/codes`.

This is a clumsy way to create a character string `path` that is the *path* to the desired directory. It may be quicker to type the whole path at once. This will work if you can type accurately. There probably are other ways that I don't know about. Everyone does things a little differently – and coding geeks argue about the best way to do things like this.

My R app opens a window called the **R Console**. This console has the R command window you've been using in the middle. There are some icons around the side, including one that says “Open document in editor” if you *mouseover* (put the cursor in the icon). Click on this icon and the R files window opens. In the beginning, you have no files (probably), so you click on **file** and then **New Document** somewhere. It's the top bar on my imac with the R app is running. I don't know where it is in the Windows version. An editor window should open. It should be blank. Figure 1 has a screen capture of my file window with two files

in it. Figure 2 has a screen capture of a blank editor window. I typed the line `cat("Hello world")` on the first line of the editor window and saved the file as `hello.R` in the directory `/Users/jg/Desktop/notes/MathFin2019/codes`. Basic typing and editing in this code editor is similar to doing it in Word. You may have to figure out the process of navigating the file system to get it to save your file where you want it. Figure 3 shows what the editor window looks like with that line and having been saved. The file name “hello.R” is on the top line of the editor window (instead of “Untitled”).

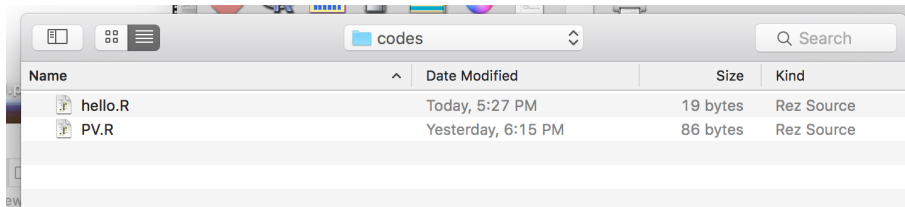


Figure 1: A screen capture of the R files window, on an imac

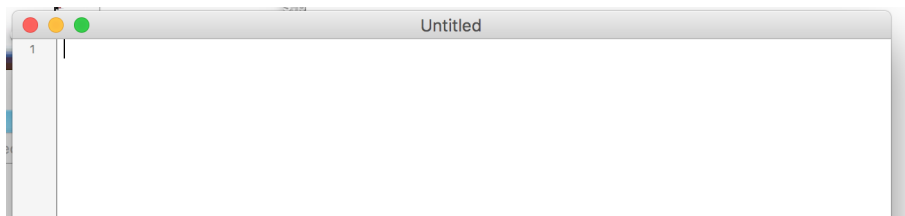


Figure 2: A screen capture of a blank R text editor window

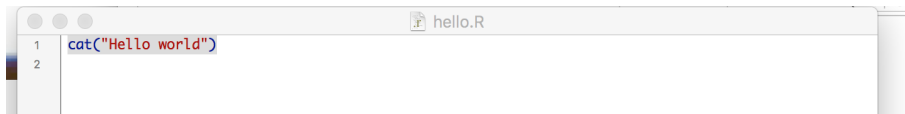
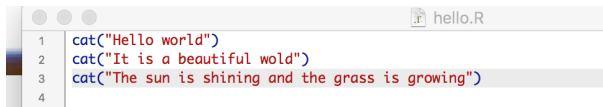


Figure 3: A screen capture of a blank R text editor window

At this point, the file `hello.R` is an R *script*, which is a file containing a sequence of R commands. This file has only one command, but we will soon add more. To execute this command I have to get to the right directory. At the command line, I type `cat("Hello world")` [enter]. The result is `Hello world` as in the first handout. Next I execute the commands (just one command) in the file `hello.R` by typing at the command line `source("hello.R")`. This is the same command and it does the same thing.

To illustrate the value of scripts and editors over the simple command line, I modified `hello.R` to print three lines. The file is in Figure 4 I could have typed these three lines at the command window, but put them in the script instead.



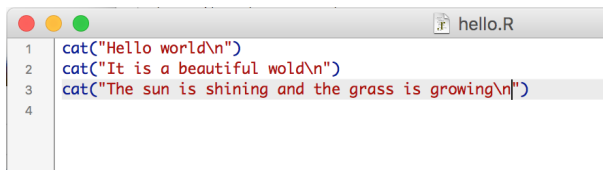
```
1 cat("Hello world")
2 cat("It is a beautiful wold")
3 cat("The sun is shining and the grass is growing")
4
```

Figure 4: The R script to type three lines

Alas, here's what happened when I ran the script:

```
> source("hello.R")
Hello worldIt is a beautiful woldThe sun is shining and the grass is growing
```

It's all on one line instead of three. Every script has bugs the first time, including this one. That is fixed (I know but forgot) by adding `\n` to the end of each line. This is carried over from the C programming language. The backslash `\` is the *escape* character. The `n` means **n**ew **l**ine. Together, `\n` says: go to the next line now. The new file `hello.R` is in Figure 5. I made it from the old one by typing `\n` in three places. I did not have to type everything over, which I would have had to do using just the command line.



```
1 cat("Hello world\n")
2 cat("It is a beautiful wold\n")
3 cat("The sun is shining and the grass is growing\n")
4
```

Figure 5: The R script to type three lines, with **n**ewline characters

Here's what happens if you run the script now:

```
> source("hello.R")
Hello world
It is a beautiful wold
The sun is shining and the grass is growing
>
```

It's almost perfect, but there is still one bug: "world" is "worl" on the second line. I edit the `hello.R` file to fix this (one key stroke and a few mouse clicks – remember to save) and it becomes

```
> source("hello.R")
Hello world
It is a beautiful world
The sun is shining and the grass is growing
>
```

If you type "Hello world" at the command line, it types "Hello world" back to you. Why do we use the `cat()` function in a script? I don't know

why, but without `cat()` the output that is typed at the command window goes somewhere else – dunno where. I edited the `hello.R` script to add the line 2 `[enter]` at the end. At the command window, this would have given `[1] 2`, but in the script it gives nothing. Try it yourself.

## Arrays

Computers store and manipulate data. *Arrays* are a way to store and access data. An *array* in R is an object with a name (the array name) that contains many data values. In the simplest R array, you find a specific data value by giving an integer, which is the value *index*. All the data values have the same type. The allowed index values are  $1, 2, \dots, n$ , where  $n$  is the *size* of the array. You get the second data value in an array by giving the array name and the index value 2.

Here is an example, from the command line:

```
> letters = c("alpha", "beta", "gamma", "delta")
> cat(letters)
alpha beta gamma delta
> letter = letters[2]
> cat(letter)
beta
>
```

The first command is an assignment statement that creates an object `letters`, which is an array. The R function `c()` (which also stands for “concatenate”, sorry) returns an array from its arguments. The arguments are the strings “alpha”, etc. The second command uses `cat()` to print the contents of the object `letters`. The four data values are there. The next command *indexes into* the array. The array index is the number in square brackets, 2. Altogether, `letters[2]` says: return the second data value in the array `letters`. It creates (or repurposes) an object called `letter` and assigns it that data value. The next command, `cat(letter)` `[enter]`, shows that the second data value was `beta`.

The array `letters` was created from a list of its data values. There are other ways to create an array. One is by reading the data from a data file or from a data set on the web. There will be a class assignment that does this. We will get stock price data from **Yahoo finance** and use this for quantitative asset allocation. Another is by *allocating* the space for the array (we will see what this means) and using R commands to create the values of the data values. Many mathematical calculations in R create arrays in this way. Here is an example at the command line

```
> numbers = 3:9
> numbers[1]
[1] 3
> numbers[4]
```

```
[1] 6
> numbers
[1] 3 4 5 6 7 8 9
```

The expression `a:b` in R means: the array with all the integers starting at `a` and ending at `b`. Therefore, `3:9` should be an array with the numbers starting at 3 and ending with 9, which is the seven numbers 3, 4, 5, 6, 7, 8, 9. The object `numbers` is created and assigned this array as its value. You can index into this array as before. First data value is `numbers[1]`, which is the first number in the sequence, 3. The fourth value is 6. The last command `numbers [enter]` returns the value of the object `numbers` itself, which is all the data values 3 through 9.

You can change a data value in an array by assigning it a new value. Here is an example from the command line. The first command creates an array of *length* 3 (the number of data values). The second command verifies that this was done correctly. The third command, `numbers[2] = 5 [enter]`, assigns the second data value the value 5. The last command shows that `numbers[2]` has changed from 7 to 5.

```
> numbers = 6:8
> numbers
[1] 6 7 8
> numbers[2] = 5
> numbers
[1] 6 5 8
```

All the data values in an array have the same type. The `letters` array has data of type character string. The `numbers` array has data of type `double` (double precision floating point). Here is an experiment from the command line showing what happens if you mess with this. The first command uses the R function `typeof()`, which is supposed to tell you the type of its argument. That's what the documentation says. But if its argument is an array, it tells you the type of the data values instead. It does not tell you that the argument is an array. The data values in `numbers` have type `double`. So, what happens if you try to one of these data values to a character string (the command `numbers[2]="beta" [enter]`)? It turns out that R converts all the data values to string. The remaining numbers 6 and 8 are converted to the one character strings "6" and "8".

```
> typeof(numbers)
[1] "double"
> numbers[2]="beta"
> numbers
[1] "6" "beta" "8"
```

There are at least two lessons to learn from this experiment. One is that you should experiment at the command line or with simple scripts to see for yourself what R does in certain situations. The other is that data values can change

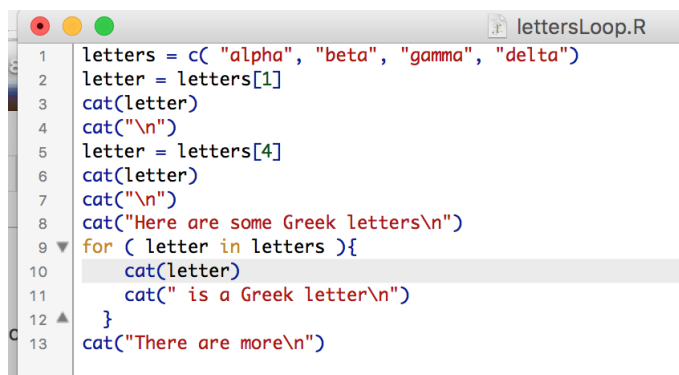
unexpectedly and in ways that can be hard to notice at first. We didn't mean to change the data value `numbers[1]`. When you print the value `numbers[1]`, you might not notice that its value has changed from 6 (type `double`) to "6" (a character string). But if you try to add `1 + 6` using `numbers[1]` for the 6, you get an *error message*. The **binary operator** is `+`. This operator needs two arguments that are numbers. There is no way (in R) to add a number to a character string.

```
> 1 + numbers[1]
Error in 1 + numbers[1] : non-numeric argument to binary operator
```

## Loops

A *loop* is a *block* of R commands (a *code block*) that is executed repeatedly. There are different kinds of loop, the *for loop* illustrated here, and the *while loop* and others. The *for loop* is *controlled* by the `for([variable] in [array])` command at the beginning. The *body* of the loop is a sequence of commands contained inside curly braces. This is like the code block that defines the body of a function, which is described in the first handout. There is an environment for the commands inside the curly braces that contains the object `variable`. The code in the loop body is executed with the value of `variable` being each of the data values in the array.

Figure 6 is a screen capture of an R editor window for editing the script `lettersLoop.R`. The first few commands create and verify the `letters` array as before. The `\n` on line 4 makes the output `delta` appear on the next line after `alpha`. Otherwise it would be `alphadelta`. Line 9 is the *control statement* for the *for loop*. It says that the loop body will be executed with the object `letter` taking all the data values in the array `letters`. Lines 10 and 11, which are enclosed in curly braces, are the body of the loop. The loop body will be executed four times, one time for each data value in the `letters` array.



```
1 letters = c("alpha", "beta", "gamma", "delta")
2 letter = letters[1]
3 cat(letter)
4 cat("\n")
5 letter = letters[4]
6 cat(letter)
7 cat("\n")
8 cat("Here are some Greek letters\n")
9 for ( letter in letters ){
10   cat(letter)
11   cat(" is a Greek letter\n")
12 }
13 cat("There are more\n")
```

Figure 6: A screen capture of the script `lettersLoop.R` in an window

Note that the commands in the loop body, lines 10 and 11, are *indented* by four spaces. This makes the code easier to read by making it visually obvious which commands are in the loop body and which are not. The closing curly could be at the end of the command at line 11, but many coders prefer to put it on its own line, but indented less than four spaces. Things like this are part of *programming style*. You write code so that people (you or others) can easily figure out what it does. Variable names are also part of programming style. The array `letters` could have been called `thing1`. The control statement could have been: `for ( thing2 in thing1 ){`. It would take longer for the person reading the code to figure out that `thing1` is a list of letters, and that `thing2` is one of those letters. *White space* is another part of programming style. This refers to blanks, or blank lines, that help the important things stand out. I used `( letter in letters )`, with blanks before `letter` and after `letters` to make them stand out from the parens. Coders argue about programming style. You can find style guides for R on the web. All good coders have a specific programming style that they stick to. In this class, you will be asked to print and hand in your code. You will lose points if your code has bad style and is hard to read.

Here's what happens when you run this script from the command line. The body of the loop has been executed four times, first with `letter` having the value "alpha", and then "beta", and so on.

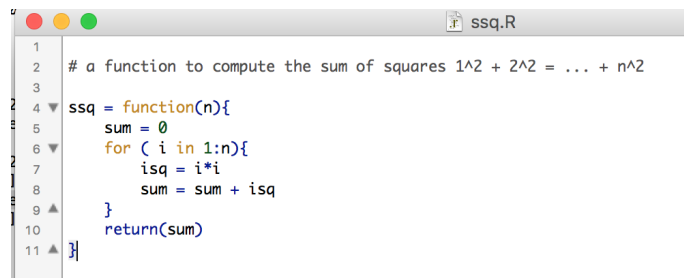
```
> source("lettersLoop.R")
alpha
delta
Here are some Greek letters
alpha is a Greek letter
beta is a Greek letter
gamma is a Greek letter
delta is a Greek letter
There are more
>
```

Figure 7 illustrates a function that does its job using a `for` loop. Lines 1 and 3 are blank lines, which is a form of white space intended to highlight line 2. Blank lines are ignored. Line 2 is a *comment*. The `#` character at the beginning of the line tells the R interpreter to ignore everything that comes after. This makes the interpreter see line 2 also as blank. The comment contains a message to the person reading the code saying what it does. You are required to put comments like this in front of every R function that you write. You will be graded on the clarity of your comments as well as the correctness and clarity of the code itself. After the first three lines of programming style, line 4 defines an object `ssq` as a function of an argument that will be called `n` in the environment of the function. The function computes the *sum of squares*  $ssq = 1 + 4 + \dots + n^2$ . In statistics, a sum of squares is called "ssq". Line 4 defines an object `sum` and assigns it the value 0. We say that the variable `sum` is *defined* (created as an object), and *initialized* (given its starting value) to 0. Line 6 is the control



statement. Inside the body of the loop, the object `i` will get all the values in the array `1:n`. In that way, `i` will have the values 1, then 2, and so on up to `n`. Lines 7 and 8 are the body of the loop. Line 7 computes the square of `i`. The name `isq` stands for “*i* squared”. Line 8 *accumulates* the sum in the variable (object) `sum`. It adds the latest square to the sum of all squares before it.

In the first *trip through* the loop `i` has the value 1 and `sum` gets the value 1. In the second trip through the loop, `i` has the value 2 and `sum` gets the value  $1 + 4 = 5$ . The old value 1 is replaced with the new value 5. This continues until it has added the squares of all the numbers in the array `1:n`.



```
1
2 # a function to compute the sum of squares 1^2 + 2^2 = ... + n^2
3
4 ssq = function(n){
5   sum = 0
6   for ( i in 1:n){
7     isq = i*i
8     sum = sum + isq
9   }
10  return(sum)
11 }
```

Figure 7: A screen capture of the function `ssq.R` in the editor window

Here is one way to use the function `ssq` from the command line. First you have to put it into the environment by executing the script `ssq.R`. Once `ssq()` is in the environment, you can use it as you use any R function. The first value is supposed to be  $1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$ . I did that as a check that it was coded correctly. The last command is something you might want a computer for. If you wanted to know  $1 + 4 + \dots + 100^2$ , it would be quicker to use R in this way than to add the numbers by hand. (This supposed you don't know the formula.)

```
> source("ssq.R")
> ssq(3)
[1] 14
> ssq(100)
[1] 338350
```