Quantitative Methods in Linguistics – Lecture 1

Adrian Brasoveanu*

March 30, 2014

Contents

We begin with a basic introduction to R (R Core Team 2013). This set of notes is primarily based on Gries (2009).

1 R as calculator

Here are some basic examples of using R as a calculator. Note that # is for comments: lines that begin with # are not executed.

```
> 2 + 3

[1] 5

> 2/3

[1] 0.6667

> 2^3

[1] 8

> # 2^3
```

*These notes have been generated with the 'knitr' package (Xie 2013) and are based on many sources, including but not limited to: Abelson (1995), Miles and Shevlin (2001), Faraway (2004), De Veaux et al. (2005), Braun and Murdoch (2007), Gelman and Hill (2007), Baayen (2008), Johnson (2008), Wright and London (2009), Gries (2009), Kruschke (2011), Diez et al. (2013), Gries (2013).

Note how the cursor changes from $>$ to $+$ when the command is not completely listed on one line.

- complete command on one line:

```
> 2 * 3

[1] 6
```

- command split on two lines:

```
> 2 *
+ 3

[1] 6
```

You can conjoin commands with ";":

```
> 2 + 3; 2/3; 2^3
```

The output for that is a 'batch' output:

```
[1] 5
[1] 0.6667
[1] 8
```

A couple more examples:

```
> sqrt(9)

[1] 3

> sqrt(16)

[1] 4
```

Whitespace generally doesn't matter:

```
> 2+3

[1] 5

> 2 + 3

[1] 5

> 2  +  3

[1] 5
```

But **always** separate terms by one whitespace so that your code is readable – except for very few cases, as listed in the code included in the lecture note. For example:

```
> 2 + 3

[1] 5
```

In general, when you type up your hw assignments, follow the layout and formatting of these lecture notes **very strictly**. Please budget enough time for making the hw assignment look properly! If your code does not follow the formatting in these lecture notes, it will be hard to read and we will not grade the hw assignment at all, so you will get no points for that hw assignment. **Follow the layout and formatting in the lecture notes VERY STRICTLY**.

2 Variables

The assignment operator "<-" allows you to save something in a variable. You cannot separate "<" and "-" by whitespace, they have to be adjacent.

```
> aa <- sqrt(5)
> aa

[1] 2.236
```

R is case-sensitive, so you need to keep in mind exactly what a vector is called, but what you call it is up to you.

```
> Aa

Error:  object 'Aa' not found

> AA

Error:  object 'AA' not found

> aA

Error:  object 'aA' not found
```

Here's a list of all the currently active variables:

```
> ls()

[1] "aa"
```

Use parens around the assignment operation to print out the value of the variable:

```
> (aa <- sqrt(5))

[1] 2.236
```

The above is equivalent to:

```
> aa <- sqrt(5)
> aa

[1] 2.236
```

You can assign values to new variables that depend on the values of other/old variables:

```
> (ab <- aa + 2)

[1] 4.236

> aa

[1] 2.236

> ab

[1] 4.236
```

You can remove one variable from the 'active list' as follows:

```
> rm(aa)
> ls()

[1] "ab"

> aa

Error:  object 'aa' not found
```

Here's one more example:

```
> a <- 1
> b <- 2
> c <- 3
> d <- a + b + c
> a

[1] 1

> b

[1] 2

> c

[1] 3

> d

[1] 6
```

List all the current variables:

```
> ls()

[1] "a"  "ab" "b"  "c"  "d"
```

You can get info/help for a command by prefixing it with a "?", as follows:

```
> ?ls
```

Force yourself to use the wonderful and easily available R help as frequently as possible! This will be one of your main sources of information as you get more familiar with R and you need to recall specific details on the fly for more complex projects.

This is how you clear all the variables from the 'active list':

```
> rm(list = ls(all = TRUE))
> ls()

character(0)
```

3 Vectors

"c" (concatenate) makes a sequence/vector.

```
> v <- c(1, 2, 3)
> v

[1] 1 2 3
```

We can now see why "[1]" often appears at the beginning of the R output: it just indicates that the first listed value is at position 1 in the (implicitly assigned) vector. That is, R automatically takes variables like "a" to be vectors, even if we assign only one value to them:

```
> a <- 1
> a

[1] 1
```

We can form vectors out of previously assigned variables and/or vectors:

```
> x <- c(a, v, 17, 19)
> x

[1]  1  1  2  3 17 19
```

We can also have characters or strings as vector values:

```
> v <- c("a", "b", "c", "d", "e", "f", "g", "hello world!")
> v

[1] "a"            "b"            "c"            "d"
[5] "e"            "f"            "g"            "hello world!"
```

To list/extract the value at a particular index/position in the vector, we do this:

```
> v[4]

[1] "d"

> v[7]

[1] "g"

> v[8]

[1] "hello world!"
```

This is called *indexing* the vector. We can also *slice* the vector, i.e., extract values at a range of indices:

```
> v[c(2, 3, 4)]

[1] "b" "c" "d"

> v[2:4]  # this is the same: 2:4 is the sequence of ints from 2 to 4 inclusive

[1] "b" "c" "d"

> 2:4

[1] 2 3 4

> c(2, 3, 4)

[1] 2 3 4

> v[c(2, 5, 8)]  # but explicitly listing the indices is more flexible

[1] "b"            "e"            "hello world!"
```

We can add something to *all* values in a vector:

```
> x

[1]  1  1  2  3 17 19

> x + 10

[1] 11 11 12 13 27 29
```

And there are more vectorial operations:

```
> x * 10

[1]  10  10  20  30 170 190

> x^2

[1]   1   1   4   9 289 361

>
> numbers1 <- c(1, 2, 3)
> numbers2 <- c(4, 5, 6)
> numbers1 + numbers2

[1] 5 7 9

> numbers1 * numbers2

[1]  4 10 18

> numbers1^numbers2

[1]   1  32 729
```

As we already indicated, ":" generates a sequence, which can be either increasing or decreasing:

```
> 1:3

[1] 1 2 3

> 3:7

[1] 3 4 5 6 7

> 1:10

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

> 100:80

 [1] 100  99  98  97  96  95  94  93  92  91  90  89  88  87  86  85  84
[18]  83  82  81  80
```

Adding the concatenation operator on top of a sequence generated with ":" is OK, but redundant:

```
> 3:7

[1] 3 4 5 6 7

> c(3:7)

[1] 3 4 5 6 7

> 7:3

[1] 7 6 5 4 3

> c(7:3)

[1] 7 6 5 4 3
```

We can assign such sequences to variables in the usual way:

```
> y <- 7:3
> y

[1] 7 6 5 4 3
```

We can test whether an object is a vector:

```
> is.vector(y)

[1] TRUE

> is.vector(9:13)

[1] TRUE

> is.vector("hello")

[1] TRUE

> (z <- data.frame(column1 = 1:3, column2 = c("Jim", "Jen", "Joe"),
+     row.names = paste("row", 1:3, sep = "")))
```

```
     column1 column2
row1       1     Jim
row2       2     Jen
row3       3     Joe

> is.vector(z)

[1] FALSE
```

There are a couple more useful commands to get information about the type, structure and contents of (vector) variables:

```
> a

[1] 1

> class(a)

[1] "numeric"

> str(a)

 num 1

> length(a)

[1] 1

> x

[1]  1  1  2  3 17 19

> class(x)

[1] "numeric"

> str(x)

 num [1:6] 1 1 2 3 17 19

> length(x)

[1] 6

> (b <- "h")

[1] "h"

> class(b)

[1] "character"

> str(b)

 chr "h"

> length(b)

[1] 1
```

```
> v

[1] "a"           "b"           "c"           "d"
[5] "e"           "f"           "g"           "hello world!"

> class(v)

[1] "character"

> str(v)

 chr [1:8] "a" "b" "c" "d" "e" "f" "g" "hello world!"

> length(v)

[1] 8
```

We can get information about (some) functions in the same way:

```
> class(length)

[1] "function"

> str(length)

function (x)
```

Importantly, vectors can only store objects of the same type. For example, let's create an empty vector of length 3:

```
> empty <- vector(length = 3)
> length(empty)

[1] 3
```

When an empty vector is created, it is automatically initialized as a boolean / logical vector and all values are set to FALSE:

```
> empty

[1] FALSE FALSE FALSE

> class(empty)

[1] "logical"

> str(empty)

 logi [1:3] FALSE FALSE FALSE
```

We can then store numbers in this vector but to do that, R automatically coerces it to numeric. In the process, all FALSE values are converted to 0:

```
> empty[2] <- 7
> empty

[1] 0 7 0
```

```
> class(empty)

[1] "numeric"

> str(empty)

 num [1:3] 0 7 0
```

We can further coerce this vector to a character vector by storing a string in one of its coordinates. All values are automatically converted to character values:

```
> empty[3] <- "blah"
> empty

[1] "0"    "7"    "blah"

> class(empty)

[1] "character"

> str(empty)

 chr [1:3] "0" "7" "blah"
```

This works because any number can be converted to a string in the obvious way. But trying to coerce the vector in the reverse way (from character to numeric) is not possible. In fact, the new values get automatically converted to the most 'general' type, namely character:

```
> empty[1] <- 23
> empty

[1] "23"   "7"    "blah"

> empty[2] <- TRUE
> empty

[1] "23"   "TRUE" "blah"

> class(empty)

[1] "character"

> str(empty)

 chr [1:3] "23" "TRUE" "blah"
```

Explicit conversion to logical / numeric / character works as follows. Note how NAs (conversion not available) are automatically introduced:

```
> as.logical(empty)

[1]   NA TRUE   NA

> as.numeric(empty)

Warning:  NAs introduced by coercion

[1] 23 NA NA
```

```
> as.character(x)

[1] "1"  "1"  "2"  "3"  "17" "19"
```

Finally, we can take random samples – with and without replacement – from a vector as follows:

```
> sample(x, size = 5, replace = T, prob = NULL)

[1] 17 17  1 17  3

> sample(x, size = 5, replace = F, prob = NULL)

[1] 19  2 17  3  1

> sample(v, size = 8, replace = T, prob = NULL)

[1] "c" "g" "e" "a" "g" "e" "e" "c"

> sample(v, size = 8, replace = F, prob = NULL)

[1] "hello world!" "f"            "c"            "g"
[5] "b"            "e"            "a"            "d"
```

See the help for more details about the "sample" function:

```
> ?sample
```

Some more examples:

```
> sample(x, 3)

[1] 17  1  1

> sample(v, 3)

[1] "c" "a" "f"

> sample(30)

 [1] 18 20  7 11 19  2 21 16  5 10 29 24  8 17 27 12 23  4 30  9  1 13 26
[24] 22 25 14 15 28  6  3

> sample(letters)

 [1] "d" "s" "b" "q" "h" "i" "e" "f" "j" "w" "l" "r" "x" "n" "y" "m" "g"
[18] "z" "t" "v" "a" "c" "u" "k" "p" "o"
```

4 Lists

Lists are heterogeneous containers. We can use them as homogeneous containers, but we don't have to:

```
> list1 <- as.list(17:20)
> list1
```

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

[[2]]
[1] 18

[[3]]
[1] 19

[[4]]
[1] 20

> length(list1)

[1] 4

> class(list1)

[1] "list"

> str(list1)

List of 4
 $ : int 17
 $ : int 18
 $ : int 19
 $ : int 20

> list2 <- as.list(letters[17:20])
> list2

[[1]]
[1] "q"

[[2]]
[1] "r"

[[3]]
[1] "s"

[[4]]
[1] "t"

> length(list2)

[1] 4

> class(list2)

[1] "list"

> str(list2)

List of 4
 $ : chr "q"
 $ : chr "r"
 $ : chr "s"
 $ : chr "t"
```

```
> list3 <- list()
> list3

list()

> length(list3)

[1] 0

> class(list3)

[1] "list"

> str(list3)

 list()
```

We index a list in the same way as a vector, except with double square brackets:

```
> list1[[2]]

[1] 18

> list2[[4]]

[1] "t"

> list3[[1]]

Error:  subscript out of bounds
```

List can store vectors of different lengths and different types (they are heterogeneous containers):

```
> list4 <- list(integers = 17:20, numbers = seq(1, 3, length.out = 10),
+     strings = letters, bools = c(T, F, T, F, F, F))
> list4

$integers
[1] 17 18 19 20

$numbers
 [1] 1.000 1.222 1.444 1.667 1.889 2.111 2.333 2.556 2.778 3.000

$strings
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"

$bools
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE
```

Note that we can index this list by position or by the name of the component:

```
> list4[[1]]

[1] 17 18 19 20

> list4$integers
```

```
[1] 17 18 19 20

> str(list4$integers)

 int [1:4] 17 18 19 20

> list4[[2]]

 [1] 1.000 1.222 1.444 1.667 1.889 2.111 2.333 2.556 2.778 3.000

> list4$numbers

 [1] 1.000 1.222 1.444 1.667 1.889 2.111 2.333 2.556 2.778 3.000

> str(list4$numbers)

 num [1:10] 1 1.22 1.44 1.67 1.89 ...

> list4[[3]]

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"

> list4$strings

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"

> str(list4$strings)

 chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" ...

> list4[[4]]

[1]  TRUE FALSE  TRUE FALSE FALSE FALSE

> list4$bools

[1]  TRUE FALSE  TRUE FALSE FALSE FALSE

> str(list4$bools)

 logi [1:6] TRUE FALSE TRUE FALSE FALSE FALSE
```

We can check whether certain objects are vectors or lists and convert between these two types of containers. Note however that the results of these commands might be unexpected:

```
> list1

[[1]]
[1] 17

[[2]]
[1] 18

[[3]]
[1] 19
```

```
[[4]]
[1] 20

> is.list(list1)

[1] TRUE

> is.vector(list1)

[1] TRUE

> as.vector(list1)

[[1]]
[1] 17

[[2]]
[1] 18

[[3]]
[1] 19

[[4]]
[1] 20

> unlist(list1)

[1] 17 18 19 20
```

Let's compare the input and output of "unlist" more closely:

```
> str(list1)

List of 4
 $ : int 17
 $ : int 18
 $ : int 19
 $ : int 20

> str(unlist(list1))

 int [1:4] 17 18 19 20
```

And here's one more example with a truly heterogeneous list:

```
> list4

$integers
[1] 17 18 19 20

$numbers
 [1] 1.000 1.222 1.444 1.667 1.889 2.111 2.333 2.556 2.778 3.000

$strings
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
$bools
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE

> is.vector(list4)

[1] TRUE

> is.list(list4)

[1] TRUE

> as.vector(list4)

$integers
[1] 17 18 19 20

$numbers
 [1] 1.000 1.222 1.444 1.667 1.889 2.111 2.333 2.556 2.778 3.000

$strings
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"

$bools
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE

> unlist(list4)

         integers1          integers2          integers3
              "17"               "18"               "19"
         integers4           numbers1           numbers2
              "20"                "1" "1.22222222222222"
          numbers3           numbers4           numbers5
"1.44444444444444" "1.66666666666667" "1.88888888888889"
          numbers6           numbers7           numbers8
"2.11111111111111" "2.33333333333333" "2.55555555555556"
          numbers9          numbers10           strings1
"2.77777777777778"                "3"                "a"
          strings2           strings3           strings4
               "b"                "c"                "d"
          strings5           strings6           strings7
               "e"                "f"                "g"
          strings8           strings9          strings10
               "h"                "i"                "j"
         strings11          strings12          strings13
               "k"                "l"                "m"
         strings14          strings15          strings16
               "n"                "o"                "p"
         strings17          strings18          strings19
               "q"                "r"                "s"
         strings20          strings21          strings22
               "t"                "u"                "v"
         strings23          strings24          strings25
               "w"                "x"                "y"
         strings26             bools1             bools2
```

```
               "z"             "TRUE"            "FALSE"
            bools3             bools4             bools5
            "TRUE"            "FALSE"            "FALSE"
            bools6
           "FALSE"
```

Let's compare the input and output of "unlist" more closely:

```
> str(list4)

List of 4
 $ integers: int [1:4] 17 18 19 20
 $ numbers : num [1:10] 1 1.22 1.44 1.67 1.89 ...
 $ strings : chr [1:26] "a" "b" "c" "d" ...
 $ bools   : logi [1:6] TRUE FALSE TRUE FALSE FALSE FALSE

> str(unlist(list4))

 Named chr [1:46] "17" "18" "19" "20" "1" ...
 - attr(*, "names")= chr [1:46] "integers1" "integers2" "integers3" "integers4" ...
```

5 More about string manipulation

Let's store a string in a variable:

```
> a_string <- "semelfactive"
> a_string

[1] "semelfactive"
```

Note the type of the variable content, and here's some commands to obtain some of its properties:

```
> class(a_string)

[1] "character"

> str(a_string)

 chr "semelfactive"

> length(a_string)

[1] 1

> nchar(a_string)

[1] 12
```

Note the difference between "length" and "nchar" in particular.
Now let's store multiple strings in a variable and try the same commands:

```
> three_strings <- c("semelfactive", "achievement", "accomplishment")
> three_strings

[1] "semelfactive"   "achievement"    "accomplishment"
```

```
> length(three_strings)

[1] 3

> nchar(three_strings)

[1] 12 11 14

> sum(nchar(three_strings))

[1] 37
```

This is how we can extract substrings of a string:

```
> a_string[1]

[1] "semelfactive"

> substr(a_string, 2, 10)

[1] "emelfacti"

> substr(a_string[1], 2, 4)

[1] "eme"

> three_strings

[1] "semelfactive"   "achievement"    "accomplishment"

> substr(three_strings[1], 2, 4)

[1] "eme"

> substr(three_strings, 2, 4)

[1] "eme" "chi" "cco"
```

More substring examples:

```
> substr(a_string, 2, 2)

[1] "e"

> substr(a_string, 1, nchar(a_string))

[1] "semelfactive"
```

This is how we can split a string:

```
> strsplit(a_string, "f")

[[1]]
[1] "semel"  "active"

> unlist(strsplit(a_string, "f"))

[1] "semel"  "active"
```

```
> strsplit(a_string, "")

[[1]]
 [1] "s" "e" "m" "e" "l" "f" "a" "c" "t" "i" "v" "e"

> unlist(strsplit(a_string, ""))

 [1] "s" "e" "m" "e" "l" "f" "a" "c" "t" "i" "v" "e"
```

So this is how we can convert a string into a vector of characters: we string-split on the empty string "".

```
> a_vector_of_char <- unlist(strsplit(a_string, ""))
> length(a_vector_of_char)

[1] 12

> a_vector_of_char[2:4]

[1] "e" "m" "e"

> a_vector_of_char[2]

[1] "e"

> a_vector_of_char[1:length(a_vector_of_char)]

 [1] "s" "e" "m" "e" "l" "f" "a" "c" "t" "i" "v" "e"
```

More examples:

```
> strsplit(three_strings, "e")

[[1]]
[1] "s"       "m"       "lfactiv"

[[2]]
[1] "achi" "v"    "m"    "nt"

[[3]]
[1] "accomplishm" "nt"

> unlist(strsplit(three_strings, "e"))

[1] "s"           "m"           "lfactiv"     "achi"        "v"
[6] "m"           "nt"          "accomplishm" "nt"

> strsplit(three_strings, "")

[[1]]
 [1] "s" "e" "m" "e" "l" "f" "a" "c" "t" "i" "v" "e"

[[2]]
 [1] "a" "c" "h" "i" "e" "v" "e" "m" "e" "n" "t"

[[3]]
 [1] "a" "c" "c" "o" "m" "p" "l" "i" "s" "h" "m" "e" "n" "t"
```

```
> another_vector_of_chars <- unlist(strsplit(three_strings, ""))
> another_vector_of_chars

 [1] "s" "e" "m" "e" "l" "f" "a" "c" "t" "i" "v" "e" "a" "c" "h" "i" "e"
[18] "v" "e" "m" "e" "n" "t" "a" "c" "c" "o" "m" "p" "l" "i" "s" "h" "m"
[35] "e" "n" "t"

> length(another_vector_of_chars)

[1] 37

> str(another_vector_of_chars)

 chr [1:37] "s" "e" "m" "e" "l" "f" "a" "c" "t" ...
```

6 The basics of working with (local) files

Here's how we can list the working directory (wd for short):

```
> getwd()

[1] "/home/ady/Desktop/Dropbox/quant_methods_spring2014/lecture_notes/lecture1"
```

Also, we can set the working directory with the command "setwd("insert here the path to whatever directory you want to work in")". For example, "." is just an abbreviation for the current wd:

```
> setwd(".")
> getwd()

[1] "/home/ady/Desktop/Dropbox/quant_methods_spring2014/lecture_notes/lecture1"
```

Let's load a file that contains text:

```
> file1 <- scan(file = "test1.txt", what = "char", sep = "\n")
> file1

[1] "Hello world!" "Hello hello!" "World world!"
```

And now let's open a file that contains numbers ("double" stands for double-precision floating point number):

```
> file2 <- scan(file = "test2.txt", what = double(), sep = "\n")
> file2

[1] 12.000  3.140  2.780  1.111
```

Here's a bit more info about the R objects that we get when we load the two files:

```
> str(file1)

 chr [1:3] "Hello world!" "Hello hello!" "World world!"

> str(file2)

 num [1:4] 12 3.14 2.78 1.11
```

As usual, type "?scan" if you want to learn more about this command.

Finally, the command to quit R is:

```
> q()
```

References

Abelson, R.P. (1995). *Statistics as Principled Argument*. L. Erlbaum Associates.

Baayen, R. Harald (2008). *Analyzing Linguistic Data: A Practical Introduction to Statistics Using R*. Cambridge University Press.

Braun, J. and D.J. Murdoch (2007). *A First Course in Statistical Programming with R*. Cambridge University Press.

De Veaux, R.D. et al. (2005). *Stats: Data and Models*. Pearson Education, Limited.

Diez, D. et al. (2013). *OpenIntro Statistics: Second Edition*. CreateSpace Independent Publishing Platform. URL: http://www.openintro.org/stat/textbook.php.

Faraway, J.J. (2004). *Linear Models With R*. Chapman & Hall Texts in Statistical Science Series. Chapman & Hall/CRC.

Gelman, A. and J. Hill (2007). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Analytical Methods for Social Research. Cambridge University Press.

Gries, S.T. (2009). *Quantitative Corpus Linguistics with R: A Practical Introduction*. Taylor & Francis.

— (2013). *Statistics for Linguistics with R: A Practical Introduction, 2nd Edition*. Mouton De Gruyter.

Johnson, K. (2008). *Quantitative methods in linguistics*. Blackwell Pub.

Kruschke, John K. (2011). *Doing Bayesian Data Analysis: A Tutorial with R and BUGS*. Academic Press/Elsevier.

Miles, J. and M. Shevlin (2001). *Applying Regression and Correlation: A Guide for Students and Researchers*. SAGE Publications.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: http://www.R-project.org/.

Wright, D.B. and K. London (2009). *Modern regression techniques using R: A practical guide for students and researchers*. SAGE.

Xie, Yihui (2013). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC.