Statistical Computing
Master Data Science
Winter Semester 2017/18

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Prof. Tim Downie

Workshop 8 — Programming and Functions in R

Introduction

The main purpose of R is to analyse data and perform probability calculations and simulations, but it also comes with a programming environment, which is one aspect that makes R so powerful compared to other statistical software. The two main areas of programming that we will consider are *control structures* and *functions*.

1 Control Structures

Control structures are the commands which make decisions or execute loops. Such structures are fundamental building blocks when writing R programs. We will look at

- Conditional execution of statements;
- Loops.

1.1 Conditional Execution of Statements

Conditional execution can be done using the `if` statement. If statements are usually used inside a loop or a function (see late sections). They allow commands to be optionally executed, depending on whether a condition is satisfied. The general structure is

```
if(logical.condition) statement1 else statement2
```

- First, R evaluates `logical.condition`, a condition that evaluates to `TRUE` or `FALSE`. It must be a single value, not a vector of logical values.
- `statement1` is either one command, or a group of commands that are delimited by `{ }`. It is only executed if `logical.condition` is `TRUE`.
- The `else` statement is optional, but when used `statement2` is evaluated when `logical.condition` is `FALSE`.

The `if` statement can be extended over several lines, and `statement1` (or `statement2` or both) may be a compound of simple statements, separated by semi-colons and enclosed within braces `{ }` .

Examples

```
> x<-3
> if(x>0)
+ sqrt(x)
[1] 1.732051
> x<- -4
> if(x>0)
+  sqrt(x)
>
> x <- 2
> y <- 1
> if ( x >= y )
+ { abval <- x - y ;
+ cat("\n","Absolute value is ",
+ abval,"\n")} else
+ { abval <- y - x ;
+ cat("\n","Absolute value is ",abval,"\n")}

Absolute value is  1
>
```

Points to note:

- The R command `cat()` prints out its arguments.
- The term `"\n"` causes R to insert a carriage return (i.e. to continue output at the beginning of a new line).
- Note that the continuation prompt `+` appears when in the middle of the `if` command. If you type the commands in directly into the console the statement is not executed until the command is complete. Note that `if(logical.condition) statement1` is a complete command so if you want an `else statement 2` the linebreak should not come immediately before `else`.

1.2 `if` statements: Details

- Usually the `logical.condition` contains an equality or inequality: `==`, `<`, `>`, `<=`, or `>=`. Note the logical equality symbol uses a "double equals" symbol, `==`. This is to differentiate it from assignment and from argument names in functions. There is also a "not equals" symbol `!=`. For any other *negation* use `!logical.condition` or `!(logical.condition)`, there is no direct symbol for *not greater than*; `if(x!>y)` will cause an error, use `if(!(x>y))` or better `if(x<=y)` instead.

- The if statement checks one logical condition. If you have more than one condition to test, then you need to consider which of the following you really want.
  - All of the multiple conditions are `TRUE`
    E.g. `if(all(y==z))` will execute the following statement only if all elements of `y` are equal to the corresponding element of `z` (elementwise comparison).
  - At least one of the multiple conditions is `TRUE`
    E.g. `if(any(y==z))` will execute the following statement when at least one element of `y` is equal to the corresponding element of `z` (elementwise comparison).
  - You have a vector and want to test a condition on each element: use the function `ifelse`

```
> x<-1:10
> ch<-ifelse(x>=5,"big","small")
> print(ch)
[1] "small" "small" "small" "small" "big" "big" "big" "big" "big" "big"
```

- If the `logical.condition` is a numerical value then it will be *coerced* to a logical value: 0 evaluates to `FALSE` and any other number evaluates to `TRUE`

```
> x<-3
> if(x) cat("x is non zero\n")
x is non zero
```

- You can use `T` and `F` as the logical values `TRUE` and `FALSE`, but you should be aware of a potential problem. You are not allowed to define objects with the name `TRUE` or `FALSE`, in the same way that you cannot define an object called `2`. You **are** however allowed to call an object `T` or `F`. Doing this can lead to bugs that are very difficult to find. Here is an example. **Example**

```
> x<-15
> if(x>10) y<-TRUE else y<-FALSE
> if(y==T) cat(x,"is greater than 10\n") else
+ cat(x,"is not greater than 10\n")
15 is not greater than 10
> T<-"tea"
> x<-25
> if(y==T) cat(x,"is greater than 10\n") else
+ cat(x,"is not greater than 10\n")
25 is not greater than 10
```

  Although this example looks artificial, a student in another course had exactly this kind of problem earlier this semester. Now please remove the object `T` to avoid future problems.

```
> rm("T")
```

Another `if` example with comments below

```
> x <- c(1,3,-2)
> if (is.numeric(x) && min(x) > 0 )
+ sx <- sqrt(x) else
+ stop("x must be numeric and positive")
Error: x must be numeric and positive
>
```

- The logical operator `&&` is simply the standard "AND" for logical expressions. Similarly `||` is the standard logical "OR". Use `eor(y,x)` if you want to use the "exclusive or" "EOR" expression.
- The command `stop()` ends the execution of whatever R is doing, reports an error, and prints message supplied as an argument.
- The function `is.numeric()` returns `TRUE` only if all elements of its argument are numeric (as opposed to logical, character *etc.*).

1.3 Loops with `for` and `while`

A loop repeats a statement (or group of statements) a number of times. Each iteration (repetition) will execute the same syntax, but will do something different as one or more variable will be different.

There two main types of loop. "`for` loops" and "`while` loops"

1.3.1 `for` statements

A `for` loop allows a statement to be iterated as a counting variable proceeds through a specified sequence. The statement is of the form

```
for (variable in vecttor) statement
```

Example

```
> for(i in 1:5) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

The variable `i` takes the values 1 to 5 sequentially, and `for` each value of `i` the command `print(i)` is executed. This is different from

```
> print(1:5)
[1] 1 2 3 4 5
```

where the `print` command is only called once.

In the above Examples, only one command is executed within the loop, that is `print(i)`. If the `statement` consists of more than one command, then you must enclose the commands in curly braces `{ }`

Example:

```
> sum1<-0
> sum2<-0
> for(i in 1:10){
+ sum1<-sum1+i
+ sum2<-sum2+i^2
+ cat("i=",i," sum =",sum1," sum of squares = ",sum2,"\n")
+ }
i= 1  sum = 1  sum of squares =  1
i= 2  sum = 3  sum of squares =  5
i= 3  sum = 6  sum of squares =  14
i= 4  sum = 10  sum of squares =  30
i= 5  sum = 15  sum of squares =  55
i= 6  sum = 21  sum of squares =  91
i= 7  sum = 28  sum of squares =  140
i= 8  sum = 36  sum of squares =  204
i= 9  sum = 45  sum of squares =  285
i= 10  sum = 55  sum of squares =  385
```

Syntax of a `for` loop

- In the above Examples the *looping variable* is called `i` but it could be called anything:

  ```
  > for(this.would.be.a.tedious.variable.to.have.to.type.everytime in 1:10)
  + print(this.would.be.a.tedious.variable.to.have.to.type.everytime)
  ```

  gives exactly the same result as the first example.

- The word `in` is required verbatim.

- Aa vector follows the word `in`, this is different from most other programming languages. This vector contains the sequence of values that we want our looping variable to take. This vector can be numeric, character or logical.

For example:

```
> for(theta in c(0,pi,2*pi)) print(sin(theta))
[1] 0
[1] 1.224606e-16
[1] -2.449213e-16
> for(let in letters[c(20,9,13)]) cat(let); cat("\n")
tim
```

Note in the last example there are no braces, so only the command `cat(let)` is evaluated in each iteration. Only once the `for` loop has finished is the command `cat("\n")` evaluated. The semicolon allows two commands to be specified on one line.

1.3.2 `while` statements

The `while` loop is very similar to the `for` loop. Instead of a statement like `for(i in 1:10)`, there is a logical condition in the brackets: `while(x<y)`. The general form is:

```
while (logical.condition) statement
```

Example

```
> x <- 1
> while ( x < 11 ) {
+ cat(3*x,"\n")
+ x <- x+2
+ }
3
9
15
21
27
>
```

Again the statement that is to be repeated can either be one command, or several statements bracketed by `{ }`. The difference between a for loop and a while loop is that a `while` loop tests a logical statement *before* each iteration. It executes that iteration if the logical statement is `TRUE` and exits the loop if it is `FALSE`.

Example: what is the first factorial number that is greater than 10,000?

```
> n<-0
> prod.sofar<-1
> while(prod.sofar<10000){
+ n<-n+1
+ prod.sofar<-prod.sofar*n
```

```
+ }
> prod.sofar
[1] 40320
```

Each time the loop is iterated, n increases by one, and `prod.sofar` is multiplied by n. Therefore `prod.sofar` is equal to $n!$ (n factorial). At the end of each iteration, the condition `prod.sofar<10000` is checked. Once it reaches 40320 the `while` loop terminates.

Note: it is up to you to make sure the `while` loop is specified sensibly. **Don't type in the example below**, but consider what would happen if you did.

```
> x<-2.1
> y<-2.5
> while(x<y) print(x)
```

1.3.3 Use of control structures

Loops in R are computationally inefficient There are many functions in R which allow you to perform the same task without needing a loop. For example: Suppose you want to calculate $\sum_{r=1}^{20} r^4$. You could use a for loop

```
> total <- 0
> for (r in 1:20) { total <- total + (r^4) }
> total
[1] 722666
```

But the function `sum` allows you to to do this directly and more efficiently.

```
> total <- sum(1:20)
> total
[1] 722666
```

Another way to avoid loops is using the apply family of functions `apply(), tapply(), lapply()` or `sapply()`.

Example: in most compiled languages to calculate the row sum of an array, you need a double for loop.

```
> M<-matrix(1:25,nrow=5,byrow=TRUE)
> rowsum<-rep(0,5)
> for(i in 1:5)
+ for(j in 1:5)
+ rowsum[i]<-rowsum[i]+M[i,j]
> rowsum
[1]  15  40  65  90 115
```

You can eliminate one loop using `sum` as above.

```
> rowsum<-rep(0,5)
> for(i in 1:5) rowsum[i]<-sum(M[i,])
> rowsum
```

```
[1]  15  40  65  90 115
```

You can eliminate both loops using `apply` and `sum` together

```
> rowsum<-apply(M,1,sum)
> rowsum
[1]  15  40  65  90 115
```

1.4 Exercise

1. Write a for loop to compute
$$\sum_{i=1}^{10} \left(\frac{1}{2}\right)^i.$$

2. Suppose S_n is defined as
$$S_n = \sum_{i=1}^{n} \left(\frac{1}{2}\right)^i.$$
Use a `while` loop to compute S_k, where k is the smallest integer such that
$$|S_k - S_{k-1}| < 10^{-6}.$$

2 Functions

Writing functions is important in R and makes it a very flexible programming and data analysis environment.

Functions enable you to:

- do similar things repeatedly without having to type them in each time,
- do multiple things by typing in just one command,
- make understanding of the code easier by grouping together a set of commands and referring to them by a sensible name,
- customise existing R functions,
- develop programs and algorithms.

Most of the R commands you have used so far are actually functions, which have been supplied with R. You can also define your own, and use them in a similar way.

2.1 Single line functions

Example: we wish to write a simple function that computes $\log_a(x)$ given x and a.

```
> log.base.a<-function(x,a=10) log(x)/log(a)
```

This creates a function called `log.base.a`. It takes two arguments `x` and `a`. If no value for `a` is given then its default value will be 10 (this is specified by the argument specification `a=10`). The value $\log(x)/\log(a)$ is computed and returns the appropriate logarithm given an x (and optionally an a)

```
> log.base.a(3.5,5)
[1] 0.7783854
```

We can check the answer because the R function `log()` takes an argument called `base`!

```
> log(3.5,base=5)
[1] 0.7783854
```

To see that the default value for a will be 10:

```
> log.base.a(3.5)
[1] 0.544068
> log.base.a(3.5,a=10)
[1] 0.544068
```

2.2 Multiline functions

Usually functions are longer than one command, and then they must be enclosed in braces like loops and if statements.

Example: write a function that returns `TRUE` if a number is a factorial number and `FALSE` otherwise

```
is.factorial<-function(n)
{
   i<-0
   prod.sofar<-1
   while(prod.sofar<n){
    i<-i+1
    prod.sofar<-prod.sofar*i
   }

   prod.sofar==n
}
```

Notes:

This is a function called `is.factorial`, and takes one argument `n`. Inside the brackets there is a `while` loop that is very similar to before, except now we stop once the factorial number `prod.sofar` is greater than or equal to `n`.

The variables within the function are *local* – `prod.sofar` will not be in the object list when you type `objects()`, or in the global environment window. This is useful because perhaps you already have an object called `prod.sofar` in your global environment, an calling `is.factorial` will not overwrite it. There are thousands of functions in R. It would be very tedious if calling an R function trashed on of your objects just because an R programmer in 2002 happend to use the same object name!

The last line of the function looks a little odd. It is a logical variable which is `TRUE` if `n` equals `prod.sofar` and `FALSE` otherwise. Thus if `n` (the value supplied by the user) is a factorial number, this line is `TRUE` otherwise it is `FALSE`. **Because this is the last statement in the function**, the value that is returned is the value of `prod.sofar==n`, either `TRUE` or `FALSE`.
So typing

```
>is.factorial(6)
[1] TRUE
```

prints the return value to the screen and

```
> a<-is.factorial(12)
```

assigns the return value (`FALSE`) to object `a`.

Comment: Some students find the general concept of writing functions is confusing at first, and don't appreciate the difference between a sequence of commands in a script file and putting those commands in a function. A useful way to think about writing functions is to pretend that you have been asked to write the function by a client. They do not want to know what commands you use in the function anymore than you want to know what commands are used in the linear modelling function `lm()`. All they need to know are the name of the function and the arguments.

2.3 Some useful points and hints about functions

2.3.1 Breaking From a Function

Functions can call other functions, both the examples above use R supplied functions.

The return value is normally the last line in the function. If you want to return a value before the last line of the function, you can use `return(x)` to return the object `x`, typically in an `if` statement. The following function returns a *complex* number if the user asks for the square root of a negative number.

```
general.sqrt<-function(x)
{
# this function returns the square root of
#  any real numeric, whether positive or negative.

# and by the way you can comment your functions
```

```
# by using the hash sign.

  if(x>=0) return(sqrt(x))

# I do not need else here because if x is positive
# then it has left the function already

  complex(real=0,imaginary=sqrt(-x))
}
> general.sqrt(-1)
[1] 0+1i
```

2.3.2 Returning more than one object

An R functions can only return one object, but sometimes you want to return more than one object. The trick is to bundle all those all those objects into a list. For example:

```
{
.....
main body of function
.....
list(temp=tt,lm.res=lm.obj)
}
```

Sometimes you can return several numeric values in one vector. E.g.the function `range` returns the maximum and minimum of the vector supplied as a vector in `c(xmin,xmax)` form.

2.3.3 Functions are objects

When typing `objects()`, you will see that `is.factorial` and `general.sqrt` are in the list of objects. If you type an object name (e.g. `x`) then the value of that object is output to the console. Typing the function name without brackets will output the text definition of the function. Every object has at least one class, function has a class called `function`

```
> class(x)
[1] "numeric"
> class(is.factorial)
[1] "function"
```

You can also output the R supplied functions, but usually what is printed is not so helpful.

2.4 Function Arguments

2.4.1 Supplying Default Values

Sometimes we will wish a particular argument of a function to take a default value unless instructed otherwise. This can be achieved using the following general form,

```
fname <- function(arg1=def1, ...) statement
```

when the value of `arg1` will be `def1` if the user does not supply an alternative.

Example

The following function calculates the sum of the `p`th power of the elements of the vector `x`, with the default value of `p` being 2.

```
> sumpow <- function(x,p=2) {
+  sump <- sum(x^p)
+  cat("\n","Sum of elements to power", p,"is",sump,"\n")
+  invisible(sump)
+ }
> sumpow(x=c(1,-1,2))

 Sum of elements to power 2 is 6
> sp.out<-sumpow(x=c(3,3),p=5)

 Sum of elements to power 5 is 486
> sp.out
[1] 486
```

`invisible()` means that the return value is not output to the console, but is assigned to an object if that is what the user wants. This is useful when the output object is a large object containing many data.

2.4.2 Local variables

A slightly unfortunate feature of R is that it tries to interpret ambiguous instructions in an intelligent way. Why is this unfortunate? Because it may have intelligently *misinterpreted* your instructions without warning you. An example of this occurs in functions, if you forget to initialise local variables. Here's an example:

```
> badfunction <- function() { x+1 }
```

What this function *should* think is: calculate the value of `x+1`; hang on, you haven't said anything about `x` so I don't know what it is; terminate with an error message. What it *actually* thinks is: calculate the value of `x+1`; hang on, you haven't said anything about `x` so I don't know what it is; maybe there's an object called `x` *outside* the function that I'm supposed to be accessing. In this case, if there *is* an object called `x`, the function will use it without telling you. This can lead to unexpected results — beware!

2.4.3 The . . . argument

A special argument is called `...` (exactly 3 points with no gap), which allows the user to specify an argument for a second function nested within the first one. This is very common with plot

parameters. E.g.:

```
convert.temp<-function(fahrenheit,...)
{
  celsius<-(fahrenheit -32)*5/9
  plot(fahrenheit,celsius,...)
  celcius
}
```

The `...` allows the user to specify graphics parameters such as `col`, and `main` to the `plot` function. `convert.temp()` doesn't need to know what those arguments are for it to do the temperature conversion.

```
> convert.temp(c(0,10,20,30,40,50,60,70,80,90,100),
+ main="Celcius against fahrenheit")
```

Exercise: Repeat the above but plotting both points and lines. `type="b"`
You do not need to change the function definition of `convert.temp` in order to do this.

2.5 Editing functions

Small functions can be included directly in your script file. Every time you edit the function, you need to run the function definition again. If you have one large function or use many user defined functions, it is best to put them in a separate text `.R` file. Then to update the functions use the command `source(file="???")` to parse the text in that file.

3 Additional Exercises

Check that each function works properly by calling the function with appropriate arguments.

1. Write a function called `innerp`. This function takes two vectors of the same length as arguments and computes $\sum_{i=1}^{n} x_i \cdot y_i$, where n is the length of the vectors. This function should:

   (a) check that `x` and `y` both have the same length, returning an error message if they are not, and

   (b) Use `x` as the default value for `y`, if `y` is not specified.

   First write the function using a for loop. Then replace the for loop using the function `sum`.

2. Write a function called `is.symmetric`, that takes a square matrix and returns `TRUE` if it symmetric and `FALSE` if it is not. If the matrix is not square then an error should be returned. Hint: make use of the R function `dim`.