Writing functions

Part I

Introduction

- Most tasks are performed by calling a function in R. In fact, everything we have done so far is calling an existing function which then performed a certain task resulting in some kind of output.
- A function can be regarded as a collection of statements and is an object in R of class `function'. One of the strengths of R is the ability to extend R by writing new functions.

Introduction

- The general form of a function is given by:

functionname <- function(arg1, arg2,...) {

Body of function: a collection of valid statements

}

- In the above display arg1 and arg2 in the function header are input arguments of the function.
- Note that a function doesn't need to have any input arguments.
- The body of the function consists of valid R statements. For example, the commands, functions and expressions you type in the R console window.
- Normally, the last statement of the function body will be the return value of the function. This can be a vector, a matrix or any other data structure.

Introduction

- The following short function meank calculates the mean of a vector x by removing the k percent smallest and the k percent largest elements of the vector.

```
meank <- function(x,k){
xt <- quantile(x, c(k,1-k))
mean( x[ x > xt[1] & x < xt[2] ])
}
```

Introduction

- Once the function has been created, it can be ran.

  test <- rnorm(100)

  meank(test,0.1)

  [1] 0.06221995

- The function meank calls two standard functions, quantile and mean. Once meank is created it can be called from any other function.

Introduction

- If you write a short function, a one-liner or two-liner, you can type the function directly in the console window.
- If you write longer functions, it is more convenient to use a script file.
- Type the function definition in a script file and run the script file.
- Note that when you run a script file with a function definition, you will only define the function (you will create a new object). To actually run it, you will need to call the function with the necessary arguments.

Introduction

- Writing large functions in R can be difficult for novice users. You may wonder where and how to begin, how to check input parameters or how to use loop structures.
- Fortunately, the code of many functions can be viewed directly. For example, just type the name of a function without brackets in the console window and you will get the code.
- Don't be intimidated by the (lengthy) code. Learn from it, by trying to read line by line and looking at the help of the functions that you don't know yet.
- Some functions call `internal' functions or pre-compiled code, which can be recognized by calls like: .C, .Internal or .Call.

Arguments and variables
Required and optional arguments

- When calling functions in R, the syntax of the function definition determines whether argument values are required or optional.
- With optional arguments, the specification of the arguments in the function header is:

  argname = defaultvalue

- In the following function, for example, the argument x is required and R will give an error if you don't provide it. The argument k is optional, having the default value 2:

Arguments and variables
Required and optional arguments

```
power <- function(x, k=2){
x^k
}


power(5)
[1] 25


power()
Error in power() : argument "x" is missing, with no default
```

Arguments and variables
The ... argument

- The three dots argument can be used to pass arguments from one function to another.
- For example, graphical parameters that are passed to plotting functions or numerical parameters that are passed to numerical routines.
- Suppose you write a small function to plot the sin function from zero to xup. (See the following page)
- The function plotsin now accepts any argument that can be passed to the plot function(like col, xlab, etc.) without needing to specify those arguments in the header of plotsin.

Arguments and variables
The ... argument

```
plotsin <- function(xup = 2*pi, ...)
{
x <- seq(0, xup, l = 100)
plot(x,sin(x), type = "l", ...)
}



plotsin(col="red")
```

Arguments and variables
Local variables

- Assignments of variables inside a function are local, unless you explicitly use a global assignment (the <<- construction or the assign function).
- This means a normal assignment within a function will not overwrite objects outside the function.
- An object created within a function will be lost when the function has finished.
- Only if the last line of the function definition is an assignment, then the result of that assignment will be returned by the function.

Arguments and variables
Local variables

- In the next example an object x will be defined with value zero. Inside the function functionx, x is defined with value 3. Executing the function functionx will not affect the value of the global variable `x'.

```
x <- 0
functionx <- function(){(x <- 3)}


functionx()
[1] 3
x
[1] 0
```

Arguments and variables
Local variables

- If you want to change the global variable x with the return value of the function functionx, you can assign the function result to x.

```
# overwriting the object x with the result of functionx
x <- functionx()
```

Arguments and variables
Local variables

- The arguments of a function can be objects of any type, even functions! Consider the next example:
- test <- function(n, fun)
- {
- u <- runif(n)
- fun(u)
- }
- test(10,sin)
- The second argument of the function test needs to be a function which will be called inside the function.

Arguments and variables Returning an object

- Often the purpose of a function is to do some calculations on input arguments and return the result.
- By default the last expression of the function will be returned.

```
myf <- function(x,y){
z1 <- sin(x)
z2 <- cos(y)
z1+z2
}
```

Arguments and variables Returning an object

- In the above example z1 + z2 is returned, note that the individual objects z1 and z2 will be lost.
- You can only return one object.
- If you want to return more than one object, you have to return a list where the components of the list are the objects to be returned.

Arguments and variables Returning an object

- For example:

```
myf <- function(x,y){
z1 <- sin(x)
z2 <- cos(y)
list(z1,z2)
}
```

Arguments and variables Returning an object

- To exit a function before it reaches the last line, use the return function.
- Any code after the return statement inside a function will be ignored. For example:

```
  myf <- function(x,y){
  z1 <- sin(x)
  z2 <- cos(y)
  if(z1 < 0){
  return( list(z1,z2))}
  else{return( z1+z2)}
  }
myf(1.2*pi,1.4*pi)
```

Arguments and variables
Lazy Evaluation

- When writing functions in R, a function argument can be defined as an expression like:

```
myf <- function(x, nc = length(x))
{
rest of the function……..
}
```

- When arguments are defined in such a way you must be aware of the lazy evaluation mechanism in R. This means that arguments of a function are not evaluated until needed.

Arguments and variables Lazy Evaluation

- Consider the following examples.

```
myf <- function(x, nc = length(x))
{
x <- c(x, x)
print(nc)
}
xin <- 1:10
myf(xin)
[1] 20
```

- The argument nc is evaluated after x has doubled in length, it is not ten, the initial length of x when it entered the function.

Arguments and variables Lazy Evaluation

```
logplot <- function(y, ylab = deparse(substitute(y))) {
y <- log(y)
plot(y, ylab = ylab)
}
y <- seq(0.1,10,l=200)
logplot(y)
```

- The plot will create a nasty label on the y axis. This is the result of lazy evaluation, ylab is evaluated after y has changed.

Arguments and variables Lazy Evaluation

- One solution is to force an evaluation of ylab first.

```
logplot <- function(y, ylab = deparse(substitute(y))) {
ylab
y <- log(y)
plot(y, ylab = ylab)
}
```