MT 3607: Computing in Statistics

Lecture 4: Modular programming and R Functions

Len Thomas

September 27 2013

1 Modular programming

1.1 Introduction

What is "modular programming"?

Splitting a program into discrete blocks of code so that each block does a small amount.

Why program in modules?

- For small pieces of programming, it's fine to write them as a list of commands
- However, as the complexity of the task increases, the length of the code also increases...
- The code becomes hard to follow, hard to debug, repetitive...
- It becomes "Spagetti code"!

Modules in R

- In R modular programming is implemented through the use of **functions**.
- Syntax:

```
function name <- function (arguments) {
  body
}
```

Example R function

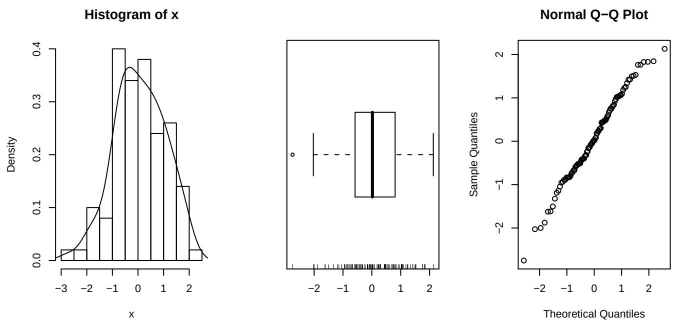
- Exploratory data analysis function (adapted from page 392 of "Using R for Introductory Statistics").
- Shows 3 plots and returns a summary of vector x

```
eda <- function (x) {
  par(mfrow=c(1,3))
  hist(x, probability=TRUE)
  lines(density(x))
  boxplot(x,horizontal=TRUE)
```

```
   rug(x)
   qqnorm(x)
   summary(x)
 }
```

- Running the function:

```
> x<-rnorm(100)
> eda(x)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
-2.74900 -0.57480  0.02036  0.07203  0.80080  2.13000
```



1.2 Advantages (and disadvantages) of modular programming

Advantages

- Easier to program
  - Divide and conquer! – divide a large problem up into small pieces and attack each separately
  - Helps us to think logically about the problem before sitting down at the keyboard (see next lecture)
- Easier to test
  - Each small piece has a defined input and output, and does only one small thing, so can easily test it works.
- Easier to debug
  - Each module is only a few lines of code – easy to see what it does and find problems
- Easier to maintain
  - Avoids having lots of lines of very similar code, so when you want to slightly change something, you just change it once.
  - E.g., From our example, imagine code to do EDA on 3 different vectors.
- Facilitates code re-use
  - You can build up a library of useful modules
- Easier to evolve
  - You can easily add functionality to modules, and then make this available to all the code that calls the module.

Disadvantages

- More thinking required
  - Modular programming works best when you sit down and plan the code in advance of starting to type
- Can be slower to run
  - There is an overhead to function calls ...
  - ... but if it is a really big deal, there are tricks to minimize the overhead
  - ... but it's not a big deal for R code, otherwise you wouldn't be using R!

1.3 Good practice in modular programming

How big should the modules be?

- A module should only do one thing
  - Why? Makes it easier to write, test, debug
  - If your module is doing lots of things, break it up into a series of modules with a "master" module that calls each in turn
  - But don't go overboard with this – e.g., the EDA example.
- Test: Give the module a name that says what it does
  - If you can't think of a name, maybe it's doing too many things!
- If you have code that does pretty-much the same thing twice, turn it into a module
- If you have many lines of code, turn them into separate modules.
  - Research has shown that far fewer bugs occur if you can fit all the code onto one screen
  - However, sometimes this isn't feasible – especially if you have lots of comments. I regularly write 2-3 screen long modules.
  - Short modules are easy to maintain.
- Start with comments that describe:
  - what the module does, and what its limitations are
  - what inputs it requires
  - what outputs it returns
- Group lines into blocks and put comments anywhere they will be useful
- Use indenting to make structure clear (a good editor will help with this)
- For more on this, see next 2 lectures

Example R function (revisited)

```
eda <- function (x) {
#Purpose: Exploratory data analysis function.  Draws 3 plots of the data,
# and returns a data summary (which is printed by default).
#Inputs:
# x - a numeric vector
#Outputs:
# a summary object

  # set the output device so it shows three plots on one page
  par(mfrow=c(1,3))

  # draw a data histogram and a density line
  hist(x, probability=TRUE)
  lines(density(x))

  # do a boxplot and add a rug to the bottom
  boxplot(x,horizontal=TRUE)
  rug(x)

  # normal probability plot
  qqnorm(x)

  # return a summary of the data
  return(summary(x))
}
```

How should I pass stuff into and out of the module?

- Use "encapsulation":
  - Pass in everything needed as parameters (function arguments in R)
  - Return all outputs explicitly at the end (in R)
  - **Do not make any global changes!** (principle of least privilege)
- Very rarely it may be necessary/useful to violate this principle, but if you do so then document why you did it.

Example of a global side-effect

```
(see previous code example)
```

- Suggestions how to fix it?
- (See also Practical 2)

2 Module interfaces

2.1 Parameters

Passing parameters

- In general programmer-speak, the things you pass into and out of modules are called “parameters”
- In R, they are called “arguments”
- e.g., in

  ```
  print.and.multiply <- function (x,y) {
    cat('Inside function x=',x,'y=',y,'\n')
    return(x*y)
  }
  ```

  `x` and `y` are function arguments.
- Arguments are the main way to pass information into functions in R. e.g.:

  ```
  > var.1<-10
  > var.2<-20
  > print.and.multiply(var.1, var.2)
  Inside function x= 10 y= 20
  [1] 200
  ```
- In general, parameters can be passed either *by value* or *by reference*.
- In R you can only pass parameters by value.

Passing by value

- A copy is made of the value of each variable passed in to a function
- These copies are stored in a separate location in memory from the original variables
- So, changes to the variable inside a function have no affect on its’ value outside the function

-
  ```
  >print.and.multiply <- function (x,y) {
  >  cat('Inside function x=',x,'y=',y,'\n')
  >  x<-x*y
  >  return(x)
  >}
  > var.1<-10
  > var.2<-20
  > print.and.multiply(var.1, var.2)
  Inside function x= 10 y= 20
  [1] 200
  > var.1
  [1] 10
  ```
- Aside - what would happen if you now typed

  ```
  > x
  ```

Passing by reference

- The memory location of the variable passed in is given to the function
- Therefore any changes to the variable within the function affect it's value after the function has completed
- The following code will not work in R as it does not allow passing by reference

```
>print.and.multiply <- function (ByReference x,y) {
>   cat('Inside function x=',x,'y=',y,'\n')
>   x<-x*y
>   return(x)
>}
> var.1<-10
> var.2<-20
> print.and.multiply(var.1, var.2)
Inside function x= 10 y= 20
[1] 200
> var.1
[1] 200
```

Pros and cons

- Passing by value:
  - + Safer – you can do what you like with the variable inside the function, and you won't affect what goes on outside the function
  - − Inefficient – the computer has to make a copy of all variables - takes time and memory
- Passing by reference:
  - + Efficient – no copying of data
  - − Dangerous – anything you do to the variables inside the function affects their value outside

Passing parameters – Conclusion

- Most 3GLs let you choose which of the above is appropriate to the circumstance
  - Some (e.g., C) allow you to specify whether variables passed by value can be changed within the function – best of both worlds!
- R only supports passing by value
- So how do you get information *out* of a function in R?
  - Use `return` - See next section!

Default argument values in R

- You can give function arguments default values.
- These arguments do not then need to be given values when the function is called.
-
```
>eda <- function (x,breaks='Scott',colour='purple') {
># Comments go here!
>  par(mfrow=c(1,3))
>  hist(x, probability=TRUE,breaks=breaks,col=colour)
>  lines(density(x))
>  boxplot(x,horizontal=TRUE)
>  rug(x)
>  qqnorm(x)
>  return(summary(x))
>}
>x<-rnorm(100)
>eda(x) #use the defaults
>eda(x,seq(-5,5,by=1)) #over-ride the default for breaks
```

Named arguments in R

- Instead of passing arguments in to functions in the order they are defined, you can use named arguments to pass them in any order.
- Useful if there are many arguments with defaults
- `>eda(x,colour='red') #over-ride the default for colour, using a named argum`
- The `...` argument can be used to pass on an unlimited number of arguments to functions within your function.
-
```
>eda <- function (x,breaks='Scott',colour='purple',...) {
># Comments go here!
>  par(mfrow=c(1,3))
>  hist(x, probability=TRUE,breaks=breaks,col=colour,...)
>  lines(density(x))
>  boxplot(x,horizontal=TRUE)
>  rug(x)
>  qqnorm(x)
>  return(summary(x))
>}
>x<-rnorm(100)
>eda(x,xlab='histogram of x') #change the x label
```

2.2 Returning values from functions in R

The `return` statement

- The main way to return values from functions is via the `return` statement

```
>multiply <- function (x,y) {
>  return(x*y)
>}
>multiply(10,20)
[1] 200
```

- If there isn't a return statement then the value of the last line is returned [1]

```
>multiply <- function (x,y) {
>  x*y
>}
>multiply(10,20)
[1] 200
```

- You can have more than one return statement (see the factorial function in lecture 2)
- If you want to return more than one variable, return a named list
- e.g.:

```
>double.value <- function (x,y) {
>  return(list(x2=x*2,y2=y*2))
>}
>double.value(10,20)
$x2
[1] 20

$y2
[1] 40
```

3 Scoping in R

3.1 Introduction to scoping rules

Scoping rules

- Scope: the section of code within which a variable[2] has meaning
- e.g., Will this work?

```
>multiply.y <- function (x) {
>  x*y
>}
>val.1<-10
>y<-20
>multiply.y(val.1)
```

- All programming languages have rules that describe when variables are in scope
- The rules in R are a bit strange, and can easily land you in trouble, which is why I'll giving you a brief outline of them now

[1]But you should have one in any case for clarity
[2]or other object – e.g., function name

Scoping rules in R

- R works with a hierarchy of "environments"
- E.g., when you're typing at the console:

```
workspace
database 1
database 2
⋮
```

- When you type a name into R, it looks for it first in the current environment (the workspace in this case), then it searches down the tree until it either finds it or bottoms out (in which case it returns an error).
- You can see the search path by typing `search()`, e.g.:

```
> search()
 [1] ".GlobalEnv"        "package:methods"   "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"     "package:datasets"  "package:svIO"
 [9] "package:R2HTML"    "package:svMisc"    "package:svSocket"  "package:svIDE"
[13] "package:tcltk"     "Autoloads"         "package:base"
```

- You can add to the search path using `attach()` (for a data frame) or `library()` (for a package). E.g.:

```
> data<-data.frame(col1=1:10,col2=1:10)
> attach(data)
> search()
 [1] ".GlobalEnv"        "data"              "package:methods"   "package:stats"
 [5] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"
 [9] "package:svIO"      "package:R2HTML"    "package:svMisc"    "package:svSocket"
[13] "package:svIDE"     "package:tcltk"     "Autoloads"         "package:base"
```

- To remove the items use `detach()` (for packages use `detach(package:packagename)`)

3.2 Scoping in functions

Scoping rules for functions

- Functions are evaluated in a new environment just above the current one (usually the workspace):

```
frame 1
workspace
database 1
database 2
⋮
```

- This means that if you use the name of a variable that is not defined in the function, the next place R will look is the workspace.

Avoiding bugs due to scoping rules

- This makes it easy to make silly typing mistakes or related bugs. E.g.:

  ```
  > multiply <- function (x,y) {
  >   x1*y
  > }
  > x<-10
  > y<-20
  > multiply(x,y)
  ```

  will not cause an error if you have a variable `x1` lying around in your workspace

- To avoid this kind of bug, make a habit of starting R sessions by removing all objects – at least if you're trying to debug some code

  ```
  > rm(list=objects())
  ```

Scoping in R - Conclusion

- Scoping is an advanced topic that you usually don't have to worry much about – unless it's causing that hard-to-identify bug.
- If you want to become a black-belt R (and SPlus) programmer, you can find out more in section 3.4 of "S Programming" (see references list from Lecture 1).