UCLA Department of Statistics
R Bootcamp

More on R

Denise Ferrari
denise@stat.ucla.edu

September 20, 2009
Outline

1. The Workspace
2. Packages
3. Beyond the Command Line
4. Loops and Functions
5. More on Plots

Denise Ferrari  denise@stat.ucla.edu
1. The Workspace

2. Packages

3. Beyond the Command Line

4. Loops and Functions

5. More on Plots
The R workspace

The R workspace consists of any user defined object (vectors, matrices, data frames, etc).

The following are some standard commands useful for managing the workspace:

Working Directory:

1. # print the current working directory
2. getwd()
3. # list the objects in the workspace
4. ls()
5. # change the working directory
6. setwd(new.dir)
The R workspace

Options:

1. # view and set options for the session
2. # list of available options
3. help(options)
4. # view current options settings
5. options()
6. # set number of digits to print on output
7. options(digits=3)
The R workspace

Session history:

1. # display last 25 commands
2. history()
3. # display all previous commands
4. history(max.show=Inf)
5. # save your command history to a file
6. savehistory(file="myfile") # default is ".Rhistory"
7. # recall your command history
8. loadhistory(file="myfile") # default is ".Rhistory"
The R workspace

Saving and loading workspaces:

```r
# save the workspace to the file "RData" (default)
save.image()

# save a specific object to a file
save(object, file="myfile.RData")

# load a saved workspace into the current session
load("myfile.RData")

# quit R
q()

# to avoid having R asking you whether it should save your data
q(save="no")
```
1. The Workspace

2. Packages

3. Beyond the Command Line

4. Loops and Functions

5. More on Plots

Denise Ferrari  denise@stat.ucla.edu
Packages

Packages consist in a set of pre-programmed functions, sometimes developed for performing specific tasks.

There are two types of packages:

- Those that come with the base installation of R.
- Those that are available for download and need to be installed manually.

To check which packages are available in your R installation:

- choose Packages & Data > Package Manager
Packages

Packages included in the base installation

Even if a package came with the base installation, you may need to load it in order to be able to use its functions.

For example, try to load a package:

```r
# load the package "splines"
library(splines)
# or
require(splines)

# to get help with the package
help(splines)
```

Now, you can start using the functions included in the loaded package.
If a package is not included in the base installation, you need to install it. You can either download it from the R website, or you can do it within the R environment:

To obtain and install the package gstat (or any other available package) from within R:

- choose Packages & Data > Package Installer
- type gstat in the package search field and hit 'Get List'
- choose the desired package and click 'Install Selected'.

Note that the process of installing a package needs to be done only once.
1. The Workspace

2. Packages

3. Beyond the Command Line
   - Script Code

4. Loops and Functions

5. More on Plots
Script code

As mentioned earlier, the process of using R requires the user to type commands on the console window and click return.

This can be burdensome, specially if you need to repeat a certain collection of commands from time to time, or if you wish to fix mistakes.

So, instead of typing your commands on the console window, you can use an R script. An R script is just a text file where you can write and save R commands for later use.
The text editor

Using a special text editor to type, document and save code for later use is highly recommended.

There is a great variety of external text editors available. We are going to use the R built-in editor.
To **create** a new R script, you may:

- choose File > New Document,
- or
- hit Command + N

So, now, you’re ready to start creating R code.
To **open** an existing R script, you may:

- choose File > Open Document,
- or
- from Finder, double-click the desired R script file.
Script code

If you’re not interested in editing the script, you can just run it.

To **run** a script, let’s say one with the name:

/Users/deniseferrari/Documents/Rcode/LHS.R

you may use:

- **The R command line:**
  
  ```r
  source("/Users/deniseferrari/Documents/Rcode/LHS.R")
  ```

- **Terminal:**
  
  ```bash
  R CMD BATCH /Users/deniseferrari/Documents/Rcode/LHS.R
  ```
Documenting script code

General guidelines

It is important that you write code in a clear, simple and well documented way. The following is a list with some general guidelines:

- Add comments: start the line with the symbol “#”.
- Use indentations to indicate which pieces of code belong together.
- Add spaces around commands, variables, commas, etc.
Using scripts to change directories

You may wish to keep your R scripts in a different directory from that in which you start the R session. An easy way of doing that is:

```r
# store the current directory
old.dir <- getwd()

# move to the new directory
setwd("/Users/deniseferrari/Documents/Rcode")

# set the output file
sink("2009-09-20.out")

# perform all desired tasks
...

# close the output file
sink()

# go back to the original directory
setwd(old.dir)
```
General Advice

Common errors can be avoided if extra care is taken regarding the following aspects:

- **R is case sensitive**
  So, a variable called `my.var` is **NOT** the same as a variable called `My.var` or `my.Var`, etc.

- **Brackets: `{ }`, `[]`, `( )`**
  Programming makes intense use of brackets. Make sure that an opening bracket `{` is matched with a closing bracket `}` and that it is used in the correct position for the task.
The Workspace

Packages

Beyond the Command Line

Loops and Functions

Loops

Functions

More on Plots

Denise Ferrari  
denise@stat.ucla.edu

More on R  
UCLA SCC
Introduction to Loops

A loop allows the program to repeatedly execute commands. There are three kinds of loops in R:

- ‘for’ loops
- ‘while’ loops
- ‘repeat’ loops
‘for’ Loops

General form:

```r
for (variable in sequence) {
    set_of_expressions
}
```

Example:

```r
for (i in 1:10) {
    print(sqrt(i))
}
```

To run the code, highlight it and hit command + return.
In the console you will see:

```r
[1] 1
[1] 1.414214
...  
[1] 3
[1] 3.162278
```
‘while’ Loops

General form:

```
while (condition) {
    set_of_expressions
}
```

Example:

```r
a <- 0; b <- 1
while (b < 10) {
    print (b)
    temp <- a + b
    a <- b
    b <- temp
}
```

In the console you will get:

```
[1] 1
[1] 1
[1] 2
...  
[1] 8
```
`repeat` Loops

General form:

```
repeat (condition) {
  set_of_expressions
  if (condition) { break }
}
```

Example:

```
a <- 0; b <- 1
repeat {
  print(b)
  temp <- a+b
  a <- b
  b <- temp
  if(b>=10){break}
}
```

In the console you will get:

```
[1] 1
[1] 1
[1] 2
... 
[1] 8
```

Note that the loop is terminated by the `break` command.
Cleaning the mess

In the previous cases, the output and the R commands are both shown in the console.

To have a cleaner version when working with loops, we can do:

```
1 x <- c(0,1)
2 while (length(x) < 10) {
3  position <- length(x)
4  new <- x[position] + x[position-1]
5  x <- c(x, new)
6 }
7 print(x)
```

Now, you'll get:

```
[1] 0 1 1 2 3 5 8 13 21 34
```
Writing Functions

A function is a collection of commands that perform a specific task.

General form:

```
fuction.name <- function (arguments){
    set_of_expressions
    return (answer)
}
```

Example:

```
Fibonacci <- function(n){
    x <- c(0,1)
    while (length(x) < n){
        position <- x[position] + x[position-1]
        x <- c(x, new)
    }
    return(x)
}
```
Writing Functions

Once you run this code, you will have available a new function called `Fibonacci`. To run the function, type on the console:

```r
> Fibonacci (10)
[1] 0 1 1 2 3 5 8 13 21 34
```

Note that for `n==1` the function is returning the first two Fibonacci numbers.

We can easily fix this with an if statement.
The ‘if’ statement

General form:

```r
if (condition) {
    set_of_expressions
}
```

We can also combine the ‘if’ with the ‘else’ statement:

```r
if (condition) {
    set_of_expressions
} else {
    set_of_expressions
}
```
The ‘if’ statement

Example:

```r
Fibonacci <- function(n) {
  if (n==1) {
    x <- 0
  } else {
    x <- c(0,1)
    while (length(x) < n) {
      position <- length(x)
      new <- x[position] + x[position-1]
      x <- c(x, new)
    }
  }
  return(x)
}
```

Now, we get:

```r
> Fibonacci (1)
[1] 0
```
The ‘ifelse’ statement

The ‘if’ statement works fine if you are conditioning on a single value. If you are working with a vector, the ‘ifelse’ statement can come in handy. General form:

ifelse (test, yes, no)

Example:

1 x <- c(6:-4)
2 sqrt(x) # gives a warning
3
4 # fix that with an ifelse statement
5 sqrt(ifelse(x >=0, x, NA))
1 The Workspace

2 Packages

3 Beyond the Command Line

4 Loops and Functions

5 More on Plots
   - Graphical Parameters
   - Text and symbols
   - Lines
   - Going Further
Graphical parameters

We can customize many features of graphs such as:

- font
- color
- characters
- axes
- etc.

One possible way of specifying these options is by using the function `par()`.

General form:

```
par (option.1 = value.1, option.2 = value.2, ... )
```
Graphical parameters

Set graphical parameters using `par()`:

```r
# view the current settings
par()

# save the current settings in the variable old.par
old.par <- par()

# change the color of the main title to red
par(col.main = "red")

# load the airquality dataset
data(airquality)

# create a histogram with the new settings
hist(airquality$Ozone)

# restore the original settings
par(old.par)
```
Graphical parameters

As result, we get the following plots:
Graphical parameters

Another way of getting the same results is by specifying the graphical parameter within the plotting function:

```r
# create a histogram with red title
hist(airquality$Ozone, col.main="red")
```

Note that, in this case, the original parameters were not modified. Thus, there is no need of saving the original configuration. In this case, the new option is valid only for this specific graph.
Changing text size

Some of the parameters responsible for controlling text size in graphs are:

- **cex**: magnification of plotting text and symbols
  - $1 = \text{default}, 1.5 = 50\% \text{ larger}, 0.5 = 50\% \text{ smaller}$
- **cex.lab**: magnification of x and y labels relative to cex
- **cex.main**: magnification of titles relative to cex

```r
# plot with the original text sizes
gplot(Ozone ~ Wind,
data = airquality,
main = "Ozone vs. Wind")

# create a plot with modified text sizes
gplot(Ozone ~ Wind,
data = airquality,
main = "Ozone vs. Wind",
cex.main = 2,
cex.lab = 1.5)
```
Changing text size

The resulting graphs are:

![Graph 1](image1)

![Graph 2](image2)
Changing plotting symbols

To change the plotting symbol use the `pch=` option:

```r
# plot with the original symbol
plot(Ozone ~ Wind,
data=airquality,
main="Ozone vs. Wind")

# create a plot with modified plotting symbols
plot(Ozone ~ Wind,
data=airquality,
main="Ozone vs. Wind",
cex = 1.5,
pch = 16,
col = "red"
)
```
Changing plotting symbols

The resulting graphs are:
Line types

You can change lines using the following options:

- lty: line type
- lwd: line width relative to the default
Let’s add a smoothing line to the plot:

```r
# original plot
plot(Wind ~ Temp,
     data = airquality,
     main = "Wind vs. Temp")

# add the smoothing line
m.loess <- loess(Wind ~ Temp, data = airquality)

# fit the smoothing curve
fit.loess <- fitted(m.loess)

# add the smoothing curve to the plot
lines(airquality$Temp, fit.loess)
```

We need to sort the values to get a proper smoothing curve.
Changing line types

Let’s add a smoothing line to the plot:

```r
# original plot
plot(Wind ~ Temp,
     data = airquality,
     main = "Wind vs. Temp")

# add the smoothing line
m.loess <- loess(Wind ~ Temp, data = airquality)

# fit the smoothing curve
fit.loess <- fitted(m.loess)

# sort the values
ord.1 <- order(airquality$Temp)

# add the smoothing curve to the plot
lines(airquality$Temp[ord.1], fit.loess[ord.1],
      lwd = 3,
      lty = 2,
      col = "red")
```
Changing line types

The resulting graphs are:
Further graphical parameters

It is possible to customize other graphical parameters such as:

- font size and style
- margins and graph size
- etc

To get more information on graphical parameters type `help(par)`.
We can use the function `title()` to add labels to a plot.

```r
# original plot
plot(Wind ~ Temp, 
    data=airquality, 
    main="", xlab="", ylab="" )

# add a red title and a blue subtitle
# make x and y labels 25\% smaller than original and green

title(main = "Wind vs. Temperature",
    col.main = "red",
    sub = "Data from La Guardia Airport",
    col.sub = "blue",
    xlab = "Temperature (F)",
    ylab = "Wind (mph)",
    col.lab = "green",
    cex.lab = 0.75)
```
Titles

The resulting graphs are:

![Wind vs. Temp](image1)

![Wind vs. Temperature](image2)
Annotations

It is possible to add text to graphs using the functions:

- `text(location, "text to add", pos, ...)`
- `mtext("text to add", side, ...)`

To get more information on graphical parameters type `help(par)`.

To get help on adding mathematical formulas to graphs, type `help(plotmath)`.

Legends can be added using the function `legend()`. For help on legends, type `help(legend)`.
A final example

```r
# original plot
boxplot(Ozone~Month, 
       data=airquality, 
       main="", xlab="", ylab="", 
       xaxt="n", 
       col=topo.colors(12))

# add a red title and a blue subtitle
title(main = "Ozone Concentration by Month", 
       col.main = "red", 
       sub = "Data from La Guardia Airport", 
       col.sub = "blue", 
       xlab = "Month", 
       ylab = "Ozone (ppm)")

# add a legend
legend("topright", inset=.05, horiz=FALSE, 
        title="Summer Months", 
        c("May","Jun","Jul", "Aug", "Sep"), 
        fill=topo.colors(12))
```

Denise Ferrari  denise@stat.ucla.edu
A final example

The resulting graph is: