Programming in R

John Verzani

CUNY/the College of Staten Island

October 28, 2008

1 Introduction

2 Functions

3 Repeating tasks

4 Scoping issues

5 Generic functions
   - S3 Programming
   - S4 programming

6 Bibliography
What this talk is about

R is open source software environment for statistical computing and graphics.

**John Chambers’s continuum**

use\textit{R} $\leftarrow \cdots \rightarrow$ programme\textit{R} $\leftarrow \cdots \rightarrow$ \textit{wizaRd}

The use\textit{R} would be an \textit{R} user familiar with \textit{R}’s command line: e.g.,

```
2 + 2
[1] 4
```

A programmer\textit{R} would know how to extend \textit{R}’s basic facilities by writing functions and perhaps packages.

A \textit{wizaRd} is a person who knows so much, we still feel like we don’t know anything.
What this talk is about

R is open source software environment for statistical computing and graphics.

John Chambers’s continuum

use\textsc{R} \leftarrow \cdots \rightarrow \textit{programme}\textsc{R} \leftarrow \cdots \rightarrow \textsc{wiza}\textsc{Rd}

The use\textsc{R} would be an \textsc{R} user familiar with \textsc{R}’s command line: e.g.,

\begin{verbatim}
  2 + 2
\end{verbatim}

\[ [1] \ 4 \]

A programmer\textsc{R} would know how to extend \textsc{R}’s basic facilities by writing functions and perhaps packages. A \textsc{wiza}\textsc{Rd} is a person who knows so much, we still feel like we don’t know anything.
What is Programming with R

In this text there are several concepts covered:

- starting with R
- commands to function transition
- packages
- objects and data
- classes
- methods and generic functions

The prime directive: Trustworthy software
What is Programming in R?

http://www.mango-solutions.com/ [3] has training in R sessions with about half the topics covering:

- R data objects
- Using R Functions
- The “apply” family of functions
- Writing R functions

The cost of this course is 900 pounds for commercial attendees and 500 pounds for academic attendees.

This presentation gives my 2 cents worth.
Introduction

What is Programming in R?

http://www.mango-solutions.com/ [3] has training in R sessions with about half the topics covering:

- R data objects
- Using R Functions
- The “apply” family of functions
- Writing R functions

The cost of this course is 900 pounds for commercial attendees and 500 pounds for academic attendees

This presentation gives my 2 cents worth.
One person’s learning 'curve'

learning 'curve' --- function viewpoint

computing on the language
scope
S4 methods
S3 methods
generic functions
default arguments
Apply functions
functions
vectorization

useR <-> programmerR <-> wizaRd
An analogy

Why do we do this in mathematics?

\[ (a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \cdots + \binom{n}{n} a^0 b^n \]
Because we have to.

- Builds algebraic facilities
- More importantly we learn to factor:
  \[ ax + bcx + bdx = x(a + b(c + d)) \]

- Factoring helps us solve equations
  - Factoring teaches us to organize our work by breaking it up into smaller tasks (add \( c + d \) ...)
  - Allows us to see the steps that are involved
  - It puts common tasks into one place (multiply \( x \) just once – not three times)
Because we have to.

- Builds algebraic facilities
- More importantly we learn to factor:

\[ ax + bcx + bdx = x(a + b(c + d)) \]

- Factoring helps us solve equations
- Factoring teaches us to organize our work by breaking it up into smaller tasks (add \( c + d \) ...)
- Allows us to see the steps that are involved
- It puts common tasks into one place (multiply \( x \) just once – not three times)
I’m writing a review of this book. I think it has a lot of interesting material and is well written, but as an R user something really bugs me.

**What bugs me?**

There are too few functions defined
MATLAB has a means of defining functions in m-files (one per file). This is just cumbersome enough that much of what is presented in the text is in terms of scripts. Which means in particular

- longer code segments can be hard to read
- some common code parts are repeated

### Why this bothers me

Our “aesthetic” from algebra is to factor out common code!
qqplots. Example 5.6 from Martinez and Martinez (in R)

```
x <- rnorm(100)
prob <- ((1:100) - 0.5)/100
qp <- qnorm(prob)
plot(sort(x), qp, pch=1, xlab="sorted data",
     ylab="Standard normal quantiles")
```

This is then repeated with just the change `x <- runif(100)`.

```
x <- runif(100)
prob <- ((1:100) - 0.5)/100
qp <- qnorm(prob)
plot(sort(x), qp, pch=1, xlab="sorted data",
     ylab="Standard normal quantiles")
```
ourqqplot <- function(x, xlab="sorted data", ylab = "standard normal quantiles", ...) {
    n <- length(x)
    prob <- seq(0,1,length=(n+2))[c(-1,-(n+2))]
    qp <- qnorm(prob)
    plot(sort(x), qp, pch=1, xlab=xlab, ylab=ylab,...)
}

ourqqplot(rnorm(100)) # call once
ourqqplot(runif(100)) # call twice
The function keyword
Assignment to a variable
The body
The last command evaluated
The arguments
Arguments with defaults
The special \ldots \text{ argument}
ourqqplot <- function(x, xlab="sorted data", ylab = "standard normal quantiles",...)
{
  n <- length(x)
  prob <- seq(0,1,length=(n+2))[c(-1,-(n+2))]
  qp <- qnorm(prob)
  plot(sortx, qp, pch=1, xlab=xlab, ylab=ylab,...)
}

The function keyword
Assignment to a variable
The body
The last command evaluated
The arguments
Arguments with defaults
The special ... argument
ourqqplot <- function(x, xlab="sorted data", ylab = "standard normal quantiles", ...) {
    n <- length(x)
    prob <- seq(0, 1, length=(n+2))[c(-1,-(n+2))]
    qp <- qnorm(prob)
    plot(sortx, qp, pch=1, xlab=xlab, ylab=ylab,...)
}

The function keyword

Assignment to a variable

The body

The last command evaluated

The arguments

Arguments with defaults

The special ... argument
Functions

Parts of an R function

ourqqplot <- function(x, xlab="sorted data",
                        ylab = "standard normal quantiles",...)

  n <- length(x)
  prob <- seq(0,1,length=(n+2))[c(-1,-(n+2))]
  qp <- qnorm(prob)
  plot(sortx, qp, pch=1, xlab=xlab, ylab=ylab,...)
}

The function keyword
Assignment to a variable
The body
The last command evaluated
The arguments
Arguments with defaults
The special ... argument
ourqqplot <- function(x, xlab="sorted data", ylab = "standard normal quantiles",...)
{
  n <- length(x)
  prob <- seq(0,1,length=(n+2))[c(-1,-(n+2))]
  qp <- qnorm(prob)
  plot(sortx, qp, pch=1, xlab=xlab, ylab=ylab,...)
}

The function keyword
Assignment to a variable
The body
The last command evaluated
The arguments
Arguments with defaults
The special ... argument
parts of an R function

ourqqplot <- function(x, xlab="sorted data",
                      ylab = "standard normal quantiles",...)
  
  n <- length(x)
  prob <- seq(0,1,length=(n+2))[c(-1,-(n+2))]
  qp <- qnorm(prob)
  plot(sortx, qp, pch=1, xlab=xlab, ylab=ylab,...)
}

The function keyword
Assignment to a variable
The body
The last command evaluated
The arguments
Arguments with defaults
The special ... argument
ourqqplot <- function(x, xlab="sorted data",
                       ylab = "standard normal quantiles", ...)
    
    n <- length(x)
    prob <- seq(0,1,length=(n+2))[c(-1,-(n+2))]
    qp <- qnorm(prob)
    plot(sortx, qp, pch=1, xlab=xlab, ylab=ylab, ...)
}

The function keyword
Assignment to a variable
The body
The last command evaluated
The arguments
Arguments with defaults
The special ... argument
Defining functions

One can define functions several different ways. Such as:

Functions can be defined at the command line:

```r
f <- function(x) x^2
f
```

Or stored in files and included in R via `source`:

```r
## f.R source
f <- function(x) x^2

rm(f) ## clear out "f"
source("f.R") ## read in
f

function(x) x^2
```

Or from within a package.

John Verzani (CSI) Programming in R October 28, 2008 19 / 166
The function body

Mathematically a function can be viewed as a black box that takes its inputs and returns an output.

- The output is the last command evaluated in the function body.

In R, the “box” is the function body.

- The body may be a single line or a block of commands. The latter must be enclosed in braces (the former can be).
- Some commands may need to have an explicit `print` command specified.
**Body examples**

**no braces**

\[ f <- \text{function}(x) \ x^2 \]

**last expression evaluated (na.rm alternative)**

\[ f <- \text{function}(x) \{x <- x[!\text{is.na}(x)]; \text{mean}(x)\} \]

\[ f(c(1,1,\text{NA},3,4)) \]

\[ [1] \ 2.25 \]

**need for print**

\[ \text{require(lattice)} \]

\[ f <- \text{function}(x) \ \text{print}(\text{bwplot}(x)) \quad \# \text{lattice issue} \]

\[ f(\text{rt}(100, df=3)) \]
Functions

Arguments in R functions

The arguments to a function allow the user to parameterize a function call – a good thing of course – but many arguments can be confusing and/or tedious. R has a few conveniences when processing arguments:

- First, argument values can be referred to by name, partial name, position, or inferred.
- R looks first for exact matches on the tags (argument names)
- Then R looks for partial matches on the tags
- Then R looks at the position
Example of argument processing

The `rnorm` function has arguments

```
  args(rnorm)
  function (n, mean = 0, sd = 1)
  NULL
```

Match on tags (missing tag `mean` uses default of 0)

```
rnorm(n = 2, sd = 10)
[1] 18.53553 -1.73848
```

Partial match on tags (missing tag `sd` uses default value of 1)

```
rnorm(n = 2, me = 2)
[1] 0.3355884 2.9974525
```

Position (missing third argument uses default for `sd` of 1)

```
rnorm(2,2)
[1] 0.3247537 3.3981810
```
Partial match on values of argument: Match.arg

As a shortcut, R tries to resolve argument tags when not fully specified (partial matching).
It can be tedious for a user to have to type exactly the value of an argument. An example is the `alternative` for a significance test:

```r
args(stats:::t.test.default)
```

```r
function (x, y = NULL, alternative = c("two.sided", "less", "greater"),
    mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95,
    ...)
NULL
```

R provides a means to specify all the possible values and then match within. The first line in the default t-test function is

```r
alternative <- match.arg(alternative)
```
The `missing` function can test if a formal argument is missing.

```r
f <- function(x, y) {
  if(missing(y))
    boxplot(x)
  else
    plot(x,y)
}
f(rt(100, df = 1))
```
Default values

In specifying the arguments of a function, the use of `tag=value` specifies a default value for the argument `tag`. Default values are very convenient for interactive use, as one need not type every single argument each time a function is called. For instance, to make a scatterplot, the following named arguments are available:

```r
args(plot.default)
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
  log = ",", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
  ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
  panel.last = NULL, asp = NA, ...) NULL
```

As one can see, only the first does not have a default. A careful look shows defaults can depend on other arguments – perhaps other defaults

```r
axes = TRUE, frame.plot = axes
```
Lazy evaluation of default values

**R Language manual:**

R has a form of *lazy evaluation* of function arguments – arguments are not evaluated until needed.

This can have side effects with the defaults. Here is a modified example from the R-language manual [1] (more later):

```r
f <- function(x, label = deparse(x), doLabel = FALSE) {
  if(doLabel) label    # evaluates label
  x <- x + 1
  print(label)
}
f(1)  # does not evaluate label first, so label is 2
[1] "2"

f(1, doLabel = TRUE)
[1] "1"
```
R has a means for functions to have an unspecified number of arguments. The ... argument allows any number of named arguments to be passed to the function (they are not matched by position or by partial tag, as are the formal arguments).

- Functions can have an arbitrary number of arguments
- Functions can easily pass along arguments intended for subsequent function calls.
Examples of...

Passing ... to another function

```r
plotl <- function(x, y, ...) plot(x, y, type="l", ...) 
x <- seq(0, 2*pi, length=100)
plotl(x, sin(x), col="blue") # (issue with type="p", say)
```

Arbitrary number of arguments

```r
args(t.test)
function (x, ...)
NULL

args(stats:::t.test.default)
function (x, y = NULL, alternative = c("two.sided", "less", "greater"),
        mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95,
        ...)
One can turn the arguments passed in via ... into components of a list, so that they can be manipulated:

Using a list

```r
noCol <- function(x,y,...) { ## stop col argument
  a <- list(...)
  a$col <- NULL; a$xlab="x"; a$ylab="y"
  a$x <- x; a$y <- y
  do.call("plot",a)
}
x <- seq(0,2*pi,length=100); noCol(x,sin(x), col="blue")
```
For loops

A common programming task is the need to repeat a task several times. Programming languages have several ways to implement this. For instance, `for` loops. These have a simple structure in R.

**structure of for loop**

```r
for(var in vector_or_list) {
    ... commands where var is bound to different values ...
}
```

**nested for loop example – one way to add**

```r
m <- matrix(rnorm(10*15), nrow=10); ret <- 0
for(i in 1:10) {  # nested for loops
    for(j in 1:15) {
        res <- ret + m[i,j]  # err, sum(m)
    }
}
```
Speed of a for loop

\[
\text{findT} \leftarrow \text{function}(x) \; \text{mean}(x)/\text{sd}(x) \ast \text{sqrt}(\text{length}(x))
\]

\[
f_1 \leftarrow \text{function}(m = 1000, n = 10) \{
\text{res} \leftarrow \text{numeric}(m) \\
\text{for}(i \in 1:m) \{
\quad x \leftarrow \text{rnorm}(n) \; \; \; \# \; \text{data creation} \\
\quad \text{res}[i] \leftarrow \text{findT}(x) \; \; \; \# \; \text{compute summary}
\}
\text{res}
\}
\]

\text{system.time}(f_1()) \; \# \text{system.time}()

\[
\begin{array}{ccc}
\text{user} & \text{system} & \text{elapsed} \\
0.103 & 0.003 & 0.106
\end{array}
\]
One bonus of factoring algebraic expressions is they allow one to see exactly what is going on:

\[(a + b)(c + d) \text{ or } ac + ad + bc + bd?\]

In the example above – applying the t-statistic to a collection of random samples – the `for` loop is mixed up with the task with the bookkeeping necessary to use the `for` loop.

**isolate actions**

It can be more transparent when the two tasks are separated out. (May or may not be faster [5].)
Repeating tasks

Applying a function to different parts of an object

R has many different functions for applying a function to different parts of an object. (Make object then apply summary function.)

apply

The apply function applies a function to parts of a matrix or more generally an array. This example applies a function over the 2nd index – the column.

```r
f2 <- function(m = 1000, n = 10) {
  tmp <- matrix(rnorm(m*n), ncol=m) # data creation
  res <- apply(tmp, 2, FUN = findT) # summarize
  res
}

system.time(f2())
  user  system elapsed
  0.104  0.002  0.107
```
More apply functions

lapply and sapply

The `lapply` function applies a function to each value in a vector, or component in a list (matrices and data frames are treated differently). It returns a list, `sapply` will attempt to simplify the output.

```r
m <- matrix(rnorm(10*100), ncol=100)
res <- sapply(as.data.frame(m), findT)  # apply to each col.
```

or with an anonymous function:

```r
m <- 100; n <- 10
res <- sapply(1:m, function(i) findT(rnorm(n)))
```
tapply

The `tapply` function will apply the function to a variable for each level of a different factor.

```
require(MASS)
with(Cars93, tapply(MPG.city, Cylinders, mean))
```

```
 3  4  5  6  8
39.33333 24.85714 18.50000 18.41935 17.00000
```

rotary

```
17.00000
```
mapply

mapply for multiple applies

(Here, the function is specified first, the remaining arguments are passed to the function.)

\[
mapply("rnorm", 1, 2^{(1:4)}, 1)\]

\[
[1] -0.03938406 2.99724367 9.32870306 16.66031943
\]

(Above is same as \texttt{rnorm(rep(1,4), 2^{(1:4)}, rep(1,4))} – with arguments repeated in function.)
The *doBy* package generalizes *tapply* by using a formula interface and introducing several interfaces. For instance, *summaryBy*

```r
require(doBy)
summaryBy(MPG.highway ~ Cylinders, data=Cars93,
  FUN = function(x,...) {
    c(xbar=mean(x,...), n=length(x))
  },
  na.rm=TRUE) # not passed to length
```

<table>
<thead>
<tr>
<th>Cylinders</th>
<th>MPG.highway.xbar</th>
<th>MPG.highway.n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43.33333</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>31.44898</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td>24.50000</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>25.32258</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>25.00000</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>25.00000</td>
<td>1</td>
</tr>
<tr>
<td>rotary</td>
<td>25.00000</td>
<td></td>
</tr>
</tbody>
</table>
The new **plyr** package unifies the family of **apply** functions, giving consistent names based on the input data type and output data type.

- **input**
  - array: "aaply" "adply" "alply" "a_ply"
  - data.frame: "daply" "ddply" "dlply" "d_ply"
  - list: "laply" "ldply" "llply" "l_ply"

- **output**
  - array
  - data.frame
  - list
  - nothing
require(plyr)

out <- ddply(baseball,
             variables. = c("year"), # as with tapply
             fun. = function(df) { # df -- extracted df
               ## mean if atleast 1
               mean(subset(df, subset=X3b >= 1, select=X3b))
             })

Whither the triple? Average triples per year for triples hitters.
How to do more baseball variables?

Request during lecture for all the variables – not much harder here with `ddply`

```r
require(plyr)
## note function returns data frame -- with doubles, triples, hr, hits
out <- ddply(baseball,
  variables. = c("year"), # as with tapply
  fun. = function(df) { # df -- extracted df
    ## mean if atleast 1
    hits <- subset(df, select=c("X2b","X3b","hr","h"))
    hits$singles <- with(hits, h - (X2b + X3b - hr))
    mean(hits) # each column
  })
```

How to view?
A plot with matplot – crowded, easier to make

```
d <- dim(out)[2]
matplot(out$year, scale(out[, -1]), type="l", main = "average hits normalized")
legend(1980, 4, names(out)[2:d], col=1:(d-1), lty=1:(d-1))
```
library(reshape) ## for melt
tmp <- sapply(out,scale) # scale to compare, now a matrix
tmp[,1] <- out[,"year"] # unscale year
graph <- xyplot(value ~ year | variable,
data = melt(as.data.frame(tmp), "year"), ## melt from reshape package
    type="l", main="Rise of HR?"
)
print(graph)
Order of operations

For the algebra student, the order of operations determine the proper answer:

\[
\frac{1}{1 + 1}
\]

\[
\frac{1}{1+1}
\]

[1] 2

\[
\frac{1}{(1+1)}
\]

[1] 0.5

In programming, a similar issue is understanding how values for variables are found.
Frames, environments

(From John Fox notes [7])

**Bound** A variable is **bound** to a value when it is assigned

\[ x \leftarrow 2 \]

**Frame** A set of bindings. Within a frame, a variable can have only one binding (but there can be many frames!)

**Environment** An environment is a nesting of frames. A value bound to an earlier frame will take precedence over one bound to the same variable in a later frame. Implemented in R as a frame and a pointer to an *enclosing environment*. (Allowing one to back up a nesting of frames.)

Unbound or free variables. A variable need not be bound in a frame, as hinted at above. Scoping rules specify the sequence of frames consulted to find the value of a variable.
Scoping issues

Environments and functions

Specializing to functions, we have the issue of where a variable’s value is found. Within a function we can have *local variables* defined in the functions; variables from the formal arguments of the function; or *unbound variables* to be evaluated through some means.

```r
unboundArg <- 1
f <- function(formalArg = 2, ...) {
  localArg <- 3
  print(c(localArg, formalArg, unboundArg))
}
f(0)
[1] 3 0 1
```
Scoping issues

The environment of a function call

“The most common example [of an environment] is the frame of variables local to a function call; its enclosure is the environment where the function was defined.” (?environment)

R-Lang: 4.3.3 Argument evaluation

One of the most important things to know about the evaluation of arguments to a function is that supplied arguments and default arguments are treated differently.

- The supplied arguments to a function are evaluated in the evaluation frame of the calling function.
- The default arguments to a function are evaluated in the evaluation frame of the function.
Scoping issues

Environment of a function call

```r
f1 <- function() {
    cat("env 1:");print(environment())
    cat("parent 1:");print(parent.env(environment()))
    f2()
}

f2 <- function() {
    cat("env 2:");print(environment())
    cat("parent 2:");print(parent.env(environment()))
}

f1() ## note both parents are global env.

env 1:<environment: 0x33d636c>
parent 1:<environment: R_GlobalEnv>
env 2:<environment: 0x33d44f4>
parent 2:<environment: R_GlobalEnv>
```
Lexical Scope

R adheres to a set of rules following *lexical scope* whereby variable bindings in the environment at the time the expression was created are used to provide values for any unbound symbols in the expression. (R Language manual, but cf. example of lazy evaluation)

Why “lexical scope” (Wikipedia)

Because matching a variable to its binding only requires analysis of the program text, this type of scoping is sometimes also called lexical scoping.
Lexical scoping can cause confusion

This came from a thread on R Help: http://www.mail-archive.com/r-help@stat.math.ethz.ch/msg26678.html

```r
x <- 5
sq <- function() {
  y <- x^2
  y
}
myfunc <- function() {
  x <- 10
  sq()
}
```

Where does `myfunc` find `x`? The poster wanted 100 \(10^2\) but got:

```r
myfunc()
```

```
[1] 25
```
Local functions

The poster was told to pass in the value of \( x \). One could also use **local functions** instead. (A local function is defined within another function body.)

```r
x <- 5
myfunc <- function() {
  x <- 10
  ## sq is now local to myfunc
  ## its parent environment is myfunc's -- x is bound to 10
  sq <- function() {
    y <- x^2
    y
  }
  sq()
}
myfunc() ## 100 = 10^2
[1] 100
```
But specified arguments are evaluated differently than default

With lazy evaluation, we can see the default value(s) for arguments are found in the environment of the function:

```r
x <- "global"
f <- function(x = "default", lazy.y = x) {
  print(x)
  x <- "local"
  print(lazy.y) ## changed binding in environment of function
}
f()
```

```r
[1] "default"
[1] "local"
```
Where are the arguments?

The semantics of invoking a function in R argument are *call-by-value*. In general, supplied arguments behave as if they are local variables initialized with the value supplied and the name of the corresponding formal argument.

### Important consideration

Changing the value of a supplied argument within a function will not affect the value of the variable in the calling frame.

```r
a <- 10
f <- function(a) {
  a <- 15
}
c(f(a), a) # assignment did not change a
[1] 15 10
```
How to set variables within functions?

Assignment via <- binds the value in the current environment. To use a different environment, the assign function can be used:

```r
a <- 10
f <- function(a)
    assign("a", 15, envir=.GlobalEnv)
c(f(a),a)
```

```
[1] 15 15
```

A shortcut is the double arrow assignment which marches up the nesting frames until it finds the bound variable or gets to the global environment.

```r
a <- 10
f <- function(a) a <<- 15
c(f(a),a)
```

```
[1] 15 15
```
Function closures

(This example comes from some online lecture notes of Roger Peng. [6].)

When a function needs optimizing, there are variables which need estimating, variables which are held fixed, and data that makes each problem different.

For `optim` The first two arguments are

- `par` Initial values for the parameters to be optimized over.
- `fn` A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.

Where to put the data? It can be passed into the function via . . . , or found through scoping.

Sometimes it is desirable to keep the data with the function (so it can be altered easily, say). This can be done using a function closure – a function with its environment, by including the data in the environment.
make.negLL <- function(data, fixed = c(FALSE, FALSE)) {
  op <- fixed; force(data) # avoid change of data via lazy load
  function(p) {
    op[!fixed] <- p # adjust fixed parameters
    mu <- op[1]; sigma <- op[2]
    a <- -(1/2) * length(data) * log(2*pi*sigma^2)
    b <- -(1/2) * sum((data-mu)^2)/ sigma^2
    -(a + b)
  }
} # return a closure = fn + env.

This function returns a function whose parent environment contains the
\texttt{data} argument so whatever is passed to the \texttt{make.negLL} for \texttt{data} will be
bound to the \texttt{data} value in the evaluation environment of the function
that is returned.
Using `make.negLL`

```r
normals <- rnorm(100, mean=1, sd=2)  # artificial data
c(xbar=mean(normals), s = sd(normals))  # summaries
  xbar   s
1.217775 1.796399

##
nLL <- make.negLL(normals)  # nll is a function closure
ls(environment(nLL))  # notice data is there
[1] "data" "fixed" "op"

We see that the environment of the closure contains the data passed into the constructor call.

normals <- rnorm(100, mean = 100, sd=2)  # modify normals in GlobalEnv
```

John Verzani (CSI)  Programming in R  October 28, 2008  90 / 166
Estimating the parameters

We can use the data to estimate the parameters using `optim`, assuming we didn’t know them

```r
out <- optim(par=c(mu=0, sigma=1), fn = nLL, method="BFGS")
out[['par']]

  mu        sigma
1.217758  1.787531
```

There is flexibility now, as we can do this with a fixed mean as well:

```r
nLL1 <- make.negLL(normals, fixed=c(mu=100,FALSE))
out <- optim(c(sigma = 1), nLL1, method="BFGS")
out[['par']]

  sigma
1.907655
```
Too many functions

R has over 1700 packages – that’s a lot of different possible functions. Too many to keep track of!

Many functions provide similar functionality:

- Make a diagnostic plot: `plot`
- Compute residuals: `resids`
- Provide a numeric summary: `summary`
- ....

These actions are in many ways “generic” in the sense of describing generally a wide range of actions.
Reduce, Reuse, Recycle

R reduces the number of functions by recycling the same function name for similar tasks, Functions which can be reused this way are called generic functions:

**Define** How to distinguish that when looking for the value bound to a function name the first one found may not be the one desired. One registers a function as “generic” so that a different search can be performed.

**Use** The actual function dispatched should be aware of the type of problem being considered (plot time series, plot factor, plot ???). The class of the argument(s) to the function are consulted to determine this.
Two types of generics

R has two different implementations of generic functions: S3 and S4.

**S3**
Easier to use, used by many of the R’s everyday functions, older more established, more informal – up to the programmer to verify the user is doing what you want.

**S4**
R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class(es) of the first argument to the generic function or ...
S3 Classes

Compared to S4 the term “simple” is apt, but S3 programming is still more work than just defining a function. The `class` function returns the class of a S3 object. All objects have a class either as an attribute, or implicit. This class is used for dispatch:

```r
data(CO2) # C02 data
class(CO2$Type) # factor
[1] "factor"

class(CO2$conc) # numeric
[1] "numeric"

class(lm(uptake ~ conc, data=CO2)) # lm
[1] "lm"

class(CO2$Plant) # length two!
[1] "ordered" "factor"
par(mfrow=c(2,4))
plot(CO2$Type)  # barplot
plot(CO2$conc)  # numeric
plot(lm(uptake ~ conc, data=CO2))  # lm (4 plots)
plot(CO2$Plant)  # barplot
Changing the class

Making an object have a new class is very informal in S3 programming. The `class` function can set it directly.

```r
x <- "some string"
class(x)
[1] "character"
attr(x, "class") # class is implicit
NULL
class(x) <- c("String", class(x))
attr(x, "class") # explicit attribute
[1] "String"    "character"
```

John Verzani (CSI) Programming in R October 28, 2008 102 / 166
Methods, dispatch

When a generic function is called, it dispatches a function to execute based on the class of the first argument.

Which function gets used?

When a function calling `UseMethod("fun")` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `fun.first` and, if it finds it, applies it to the object. If no such function is found a function called `fun.second` is tried. If no class name produces a suitable function, the function `fun.default` is used, if it exists, or an error results. (See `?S3Methods` for more detail.)
A String class with some methods

A constructor for a String object

```r
## constructor of objects of class "String" - same name
String <- function(x) {
  x <- paste(unlist(x), collapse="")
  class(x) <- c("String",class(x))
  return(x)
}
String("bart")
[1] "bart"
attr(."class")
[1] "String"   "character"
```
The **print** method

The print generic is called whenever an expression is evaluated at the command line. For the String class, there is currently no `print` method for either the String or character class, so `print.default` is dispatched. To define a print method for the String class is straightforward using the special naming convention for methods, as the generic is already defined.

```r
print.String <- function(x) print(unclass(x))
String("Bart")

[1] "Bart"
```
\(+ = \text{Paste}\)

A syntactic alternative for paste using Ruby conventions for strings

```
"+.String" <- function(e1,e2) {
  # Not Abelian! (x + y neq y + x)
  out <- paste(e1,e2, sep=" ")
  String(out)
}
x <- "Bart"; y <- "Simpson"
(try(x+y))
```

```
[1] "Error in x + y : non-numeric argument to binary operator"
attr(,"class")
[1] "try-error"
```

```
class(x) <- "String"
x + y
```

```
[1] "Bart Simpson"
```
"[.String" <- function(x,i,j, ..., drop=TRUE) {
  tmp <- unlist(strsplit(x,""))
  if(missing(i))
    out <- tmp[]
  else
    out <- tmp[i]
  if(drop) String(out) else out
}

x[2:3]

[1] "ar"

x[2:3, drop = FALSE]

[1] "a" "r"
A new generic

New generics may be defined. First they must be registered. We create a function to squeeze a string down to a certain size

```r
squeeze <- function(x, size=20, ...) UseMethod("squeeze")
```

Then methods can be defined. A default method

```r
squeeze.default <- function(x, size=20, ...) x # nothing
```
Generic functions

S3 Programming

A method for strings

```
squeeze.String <- function(x, size=20, ...) {
  if(nchar(x) <= size) NextMethod(x) # example of next method
  ind <- which(" " == x[, , drop=FALSE])
  if(length(ind) == 0) {
    if(nchar(x) <= size) return(x) else return(x[1:size] + "...
  } else {
    i <- max(ind[which(ind <= size)])
    return(x[1:i] + "...")
  }
}
```

```
x <- String("The quick brown fox jumped over the lazy dog")
squeeze(x)
```

[1] "The quick brown fox  ..."
Defining an S4 class

- A new class in S4 requires a name and a representation. The latter specifies the information to be contained in the slots of the class.
- The R function to define a new class is `setClass`.
- Besides the representation, there are other characteristics of the class that can be set. In the example below we specify a prototype to assign default values to the slots when a new object is initialized.
A polygon class

We define a class to hold a polygon, which we regard as an ordered set of coordinates in the $x$-$y$ plane. Our slots contain separate $x$ and $y$ values to store the points, and a tolerance.

```r
setClass("Polygon",
    representation(x = "numeric", y = "numeric",
                   gefgw = "numeric" # a tolerance
    ),
    prototype(x = numeric(0), y = numeric(0),
              gefgw = 0.0001)
)

[1] "Polygon"
```
Validation of arguments

S4 classes are less trusting than S3 classes. In S3 classes it is expected that the programmer verify the user input. In S4 classes, each value assigned to a slot is checked to ensure it is valid. By default, the representation is specified to check that the values have the specified class. For our Polygon class we need to ensure that both $x$ and $y$ are the same length. The `setValidity` function allows us to specify our own function to verify the lengths are equal. (This is more traditionally added to the `validity` argument of `setClass`.)
Validation for polygon class

Return TRUE if valid else an error message

```r
QT <- setValidity("Polygon", 
    function(object) {
      if(length(object@x) == length(object@y))
        return(TRUE)
      else
        return("unequal length coordinates")
    })
```

(We don’t need to check for numeric values as non-numeric values can’t be put into the slots x and y.)
Classes are templates for objects. To create an instance of a class the `new` function is used. This function creates an initial object from the specified prototype, then initializes it based on further arguments and the `initialize` method for the class (which can be modified). Finally the new object is validated.

Example of `new`. Basic usage

```r
defPoly <- new("Polygon")
```

(Quibbles about this default object not being a “polygon” are noted.)
Constructors

The `new` call is often avoided by the user, as the programmer typically provides a constructor(s). Often they have the same name as the class. For example,

```r
## constructor
Polygon <- function(x,y) {
  new("Polygon",x=x,y=y)
}

##try it out
triangle <- Polygon(x=c(0,2,1), y=c(0,0,1))
(try(Polygon(x = c(0,2,1), y = c(0)))) # different sizes
```

```r
[1] "Error in validObject(.Object) : 
invalid class "Polygon"
attr(,"class")
[1] "try-error"
```
Coercion methods (as functions)

Coercion can be used to turn an object of one class, into an object of another class. The `setAs` function can be used to define how coercion is to be done:

```r
setAs("matrix","Polygon",function(from,to) {  # coerce matrix
  if(nrow(from) > 1)
    new("Polygon",x=from[1,], y=from[2,])
  else
    stop("Need 2 or more rows to define a polygon")
})
setAs("data.frame","Polygon", function(from,to) { # data frame
  if(ncol(from) <= 1)
    stop("need two or more columns")
  if(! (is.numeric(from[,1]) && is.numeric(from[,2])))
    stop("need first two columns to be numeric")
  as(t(as.matrix(from[,1:2])), "Polygon")
})
```
Example of coercion

The package **PBSmapping** has several maps. The 3rd and 4th columns are the coordinates.

```r
data(worldLL, package="PBSmapping")
## continent as polygons, 0 is -- you guess
cont <- worldLL[worldLL[,1] == 0,3:4]
class(cont)

[1] "PolySet" "data.frame"
## no as for PolySet. Could coerce, we promote:
setOldClass("PolySet") # S3 class to S4
setIs("PolySet","data.frame") # force S3 inheritance
p <- as(cont,"Polygon")
```

But what is p?
Generic functions and class methods

- Methods are the functions that get dispatched when a generic is called.
- In S4, the dispatch of methods is controlled by possibly all the arguments of the generic – not just the first, as with S3 methods.
- For a given method, the arguments to match for dispatch are defined by the signature of the method. This is specified when defining the method.
The standard show method

In S3 objects are shown through their `print` method, in S4 the `show` generic is called. This generic has one argument `object`. A `show` method for the Polygon class could be:

```r
setMethod("show",
signature = c(object="Polygon"),
function(object) {
  cat(class(object),"object with\n")
  cat(" Num vertices:", length(object@x),"\n")
  invisible(object)
})
```

[1] "show"

Now call the show method (implicit)

```r
p # calls show(p)
```

Polygon object with

  Num vertices: 8108
Turning S3 methods into S4 methods

R has numerous S3 generics. These are so deeply embedded into the structure of R, that it is an unthinkable task to replace them. Instead of trying, S4 methods were made to play nicely with the S3 ones. The `setMethod` call will work as for `show`

```r
QT <- setGeneric("print")  # S3 generic, no def needed
QT <- setMethod("print",signature(x="Polygon"),
               function(x, ...) {  # match formal args
                  print(rbind(x=x@x, y=x@y))
               })
```

```r
print(triangle)
```

```
[,1] [,2] [,3]
x  0   2   1
y  0   0   1
```
A plot method (local functions)

This plot method is shown to illustrate a use of local functions.

```r
QT <- setGeneric("plot")
QT <- setMethod("plot","Polygon",function(x,y,...) {
  ## use local functions to split up "..."
  localPlot <- function(...,col) plot(...) # pass col to polygon
  localPolygon <- function(...,main,sub,xlab,ylab,xlim,ylim)
    polygon(...)

  localPlot(NA,NA, xlim = range(x@x), ylim = range(x@y),...)
  localPolygon(x@x, x@y,....)
})
```

John Verzani (CSI)
Example of plot method

```
plot(p) # that's a continent?
```
Defining new generics

- New generic functions can be added using the `setGeneric` function, as above.
- As there is nothing to build a template from, a definition must be given.
- This should specify the arguments to the function and then call `standardGeneric` (was `UseMethod` for S3)
## new S4 generic -- use setGeneric with a function template
QT <- setGeneric("rotate", def = function(object, theta) {
  standardGeneric("rotate")
})

## define a method to dispatch when generic is called
QT <- setMethod("rotate",

  signature(object="Polygon",theta="numeric"),
  function(object,theta) {
    A <- rbind(c(cos(theta), -sin(theta)),
               c(sin(theta), cos(theta)))
    m0 = rbind(object@x, object@y)
    m1 = A %*% m0
    return(new("Polygon",x = m1[1,], y = m1[2,])
  })
The Australian view

\texttt{plot(rotate(p,pi), main="Australian view")}
rotate<-?

In object oriented programming the method can usually alter the object. Not so in R, as the object is a local copy inside the method – so <- methods are used.

```r
QT <- setGeneric("rotate<-", def = function(object, value) {
    standardGeneric("rotate<-")
})
## define a method to dispatch when generic is called
QT <- setReplaceMethod("rotate",
    signature(object="Polygon", value="numeric"),
    function(object, value) {
        object <- rotate(object, value)
        return(object)
    })
```
Example of `rotate<-`

```r
par(mfrow=c(1,2))
plot(triangle, main="triangle")
rotate(triangle) <- pi/2
plot(triangle, main="triangle 2")
```
An area method

(to be inherited by sub-classes)

```r
## find the area
QT <- setGeneric("Area", function(object,...)
    standardGeneric("Area"))

QT <- setMethod("Area","Polygon",
    function(object,...) {
        area <- .findArea(object)
        return(area)
    })
```
Subclasses

In algebra we know

\[ a(b + c(d + e)) \]

implies a nesting of the operations:

\[ \text{map of } a(b+c(d+e)) \]
Sub classes

- By factoring, we can end up only needing to multiply one expression by $a$ instead of 3.

- Classes allow something similar, we need only define methods for base classes (superclasses), and any sub class will automatically inherit these methods.

- To define a new class to be a sub class we specify the parent (or super class) using the `contains` argument of `setClass`. The slots of the super class are automatically inherited by the sub class.
What is a trapezoid?

My daughter’s 4th grade math book

trapezoid: A quadrilateral that has exactly one pair of parallel sides. No two sides need be the same length.

Wikipedia

A trapezoid ... is a quadrilateral ... that has at least one pair of parallel lines for sides. Some authors define it as a quadrilateral having exactly one pair of parallel sides, so as to exclude parallelograms, which otherwise would be regarded as a special type of trapezoid, but most mathematicians use the inclusive definition.
Quadrilateral classes

Our example will cover some of the special quadrilaterals.
QT <- setClass("Quadrilateral", contains = "Polygon",
  ## subclass same slots, but stricter defn
  validity = function(object) {
    if(length(object@x) == 4)
      return(TRUE)
    else
      return("quadrilaterals have 4 sides, so need
  })
Parallelograms and special cases

```r
QT <- setClass("Parallelogram", contains = "Quadrilateral", 
              validity = function(object) {
                .parallelSides(object)
              })

pgram <- new("Parallelogram", x=c(0,1,2,1), y = c(0,0,1,1))
Area(pgram)
[1] 1

plot(pgram, col="blue")
```

![Parallelogram Diagram](image)
Rhombus – 4 equal sides

\[
QT \leftarrow \text{setClass("Rhombus", contains = "Parallelogram",}
\]
\[
\quad \text{validity = function(object) }
\]
\[
\quad \quad \quad \quad .\text{equalSideLengths(object)}
\]
\[
\}
\]

\[
rhombus \leftarrow \text{new("Rhombus", x = c(-2,0,2,-0), y = c(0,1,0,-1))}
\]
\[
\text{plot(rhombus, col="red")}
\]
Rectangle – 4 equal angles

QT <- setClass("Rectangle", contains = "Parallelogram",
  validity = function(object) {
    .equalAngles(object)
  })

rect <- new("Rectangle", x=c(0,2,2,0), y = c(0,0,1,1))
plot(rect, col="green")
Square – rectangle with equal side lengths

\[
QT \leftarrow \text{setClass("Square",}
\]
\[
\begin{align*}
&\text{contains} = c("Rectangle"), \\
&\text{validity} = \text{function}\(\text{object}\) \{
&\quad .\text{equalSideLengths}(\text{object})
&\},
\end{align*}
\]

\[
sq \leftarrow \text{new("Square", x = c(0,1,1,0), y = c(0,0,1,1))}
\]

\[
\text{plot}(sq, \text{col="magenta"})
\]
Bibliography


Martin Maechler thoughts, www.ci.tuwien.ac.at/Conferences/useR-2004/Keynotes/Maechler.pdf


John Fox, Appendix on scoping http://socserv.mcmaster.ca/jfox/Books/Companion/appendix-scope.pdf

http://wiki.r-project.org/rwiki/doku.php?id=tips:classes-s4

Christophe Genolini, A (not so) short introduction to S4