Introduction to R Functionality with Illustrations

Zhiguo (Alex) Zhao

Division of Cancer Biostatistics
Department of Biostatistics
Vanderbilt University School of Medicine

January 20, 2011
Outline

1. Why R?
2. Why programming in R?
3. Help in R
4. Data Type
   - Numbers
   - Strings
5. Objects in R
6. Programming Structure
7. Write R Functions
8. S3 Method
9. Debug R Functions
10. Optimize R Functions
11. Other Interesting Topics
12. Questions & Comments
13. References
R strengths

- Technical advantages: free, open-source, available for all OSs
- Complete statistical package
- Powerful graphics
- Access to fast growing number of analysis packages
- Is standard for biostatistical analysis
- Most widely used language in bioinformatics
- Much more ...
Programming in R

- Programming language
- Efficient data structures make programming very easy
- Ease of implementing custom functions
- Object-oriented language. Many functions in R return objects, and these objects contain information that can be operated on by other functions.
Find help in R

- `help(fn)` or `? fn` for help on `fn`
- `help.search("topic")` for help pages related to "topic"
- `apropos("tab")` for functions whose names contain "tab"
- `library(help=libraryname)` will list the functions which are available in the package
- `getAnywhere(fn)` can read invisible functions
R data type

- Integers
- Numbers (floating-point)
- Complex
- Character Strings
- Objects
Numbers: Be careful with floating point

> ### Example 1:
> .1 == .3/3
[1] FALSE

> ### Example 2:
> seq(0, 1, by=.1) == .3
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> ### Example 3:
> unique(c(.3, .4 - .1, .5 - .2, .6 - .3, .7 - .4))
[1] 0.3 0.3 0.3
Manipulating character strings: Examples

> ### Example 1:
> str1="This-is-an-example"
> gsub("-"," ",str1)
[1] "This is an example"

> ### Example 2:
> str1="This$is$an$example"
> gsub("$"," ",str1)
[1] "This$is$an$example "

> ### Example 3:
> str1=c("ACGT.C","ACGTAC","ACGTCC","ACGTGC","ACGTTC")
> grep("ACGT.C",str1)
[1] 1 2 3 4 5
Manipulating character strings

- String matching is **NOT** `===`
- Functions for searching and modifying strings are based on regular expressions, which are like simple programs for representing strings.
- R interprets certain functions arguments as regular expressions.
  - `grep(regexp, vector)` finds all the strings in the vector that contain a substring matching the regular expression
  - `sub(regexp, replacement, vector)` replaces the first substring matching the regular expression with the replacement (for each element of the vector). `gsub` does the same thing but can make more than one replacement per string
  - `regexpr(regexp, vector)` returns the position of the first match within each string, `gregexpr` is the same except that it returns all matches.
  - `strsplit()` splits a string at each match to a regular expression
Manipulating character strings: Metacharacters

Includes: $ * + . ? [ ] ^ { } | ( ) \n
- “$” : Match the empty string at the end of a line
- “^” : Match the empty string at the beginning of a line
- “*” : The preceding item will be matched zero or more times
- “+” : The preceding item will be matched one or more times
- “.” : Matches any single character
- “?” : The preceding item is optional and will be matched at most once
- “[ , ]” : Character class brackets
- “{n }” : Matches the preceding character, or character range, n times exactly
- “{n,m}” : Matches the preceding character at least n times but not more than m times
- “|” : OR
- “(, )” : Brackets for grouping
- “\” : Opening \, for example \t, \n
Manipulating character strings: Classes of characters

A character class is a list of characters enclosed between [ and ] which matches any single character in that list; unless the first character of the list is the caret, when it matches any character not in the list.

Their interpretation depends on the locale. The interpretation below is that of the POSIX locale.

```
[:alnum:]
```
: Alpha-numeric characters: “[:alpha:]” and “[:digit:]”

```
[:alpha:]
```
: Alphabetic characters: “[:lower:]” and “[:upper:]”

```
[:blank:]
```
: Blank characters: space and tab. (This is an extension to the POSIX standard)

```
[:digit:]
```
: Digits 0-9

```
[:space:]
```
: Space characters tab, newline, vertical tab, form feed, carriage return, and space
Manipulating character strings: More examples

### Example 1:
```r
str1 = c("abc","ABC","babc","9abc")
grep("^[a-z]+$",str1, value=TRUE)
[1] "abc" "babc"

grep("[[:lower:]]+$",str1, value=TRUE)
[1] "abc" "babc" "9abc"
```

### Example 2:
```r
str1 = c("16__record_1", "106__record_21","107__record_1","107__record_2")
gsub("^[0-9]+__record_[0-9]+$","\\1",str1)
[1] "16" "106" "107" "107"
```
Almost everything in R is an object. R objects generally hold a collection of items rather than just a single value. The basic objects can be imagined to be linear (like a single column in a spreadsheet) or rectangular (like a selection of rows and columns in a spreadsheet), or with even higher dimensions (array).

- **Vector**: The most common one.
- **Matrix**: Has a single type of entry.
- **Data frame**: Looks like a matrix, but may have different types in different columns.
- **List**: Contains components which can be any sort of object including another list.
Objects need to be named.

- Names are combinations of letters, digits, the period and underscores.
- Case-sensitive
- Must start with letters
- Some names are reserved. For example: return, TRUE, if.
Subscripting

Everything in R is a vector (but some have only one element). Use [ ] to extract subsets.

- Positive number (can be a sequence, such as 5:10)
- Negative number
- Characters such as names can be used to subscript. For example:
  ```r
  vector1[c("first","third")]
  ```
- Logicals, select the location that you want
  - You need == to test equality, not just =
- You many need indices for more than 1 dimension. For data frames, for example, you need 2 indices. mydata[1:5, c("id","temp","wbc")]

Zhiguo (Alex) Zhao (VU)
Control Structure

- Loops
  - for
  - while
  - repeat, break

- Conditional statements
  - if(statement) { action } else { action}
  - ifelse()
Vectorized Functions

Suppose we have a function $f()$ that we wish to apply to all elements of a vector $x$. In many cases, we can accomplish this by simply calling $f()$ on $x$ itself.
Goal: Never perform any repetitive computer tasks manually again

A key reason that R is a good thing is because it is a language. The power of language is abstraction. The way to make abstraction in R is to write functions. R is a highly functional language; virtually everything in R is done through functions.
Example: ROC curve

Plotting the sensitivity and specificity of a continuous variable as a predictor of a binary variable in an ROC curve.

```r
ROC = function(test, disease) {
  cutpoints <- c(-Inf, sort(unique(test)), Inf)
  sensitivity <- sapply(cutpoints, function(result) mean(test > result & disease) / mean(disease))
  specificity <- sapply(cutpoints, function(result) mean(test <= result & !disease) / mean(!disease))
  plot(sensitivity, 1 - specificity, type="l")
  abline(0, 1, lty=2)
  return(list(sens=sensitivity, spec=specificity))
}
```
Write R Functions

Example: ROC curve

```r
> set.seed(1)
> x<-rnorm(100,mean=0)
> y<-rnorm(100, mean=1)
> isx<-rep(c(TRUE,FALSE),each=100)
> ROC(c(x,y), isx)

$sens
[1] 1.00 0.99 0.98 0.97 0.96 0.95 0.94 0.93 ... 
[17] 0.85 0.84 0.83 0.82 0.82 0.82 0.81 0.80 ... 

$spec
[1] 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 ... 
[17] 0.01 0.01 0.01 0.01 0.02 0.03 0.03 0.03 ...
```
Generics and Methods

Many functions in R are generic. This means that the function itself (eg plot, summary, mean) does not do anything. The work is done by methods that know how to plot, summarize or average particular types of information. When the modeling procedures produce an object which contains an attribute known as the class of the object, certain functions will do the “right” thing when they are called with such an object as their argument. For example:

If you call summary on a data.frame, R works out that the correct function to do the work is summary.data.frame and calls that instead. If there is no specialized method to summarize the information, R will call summary.default.

The class and method system makes it easy to add new types of information and have them work just like the built-in ones.
Create new classes

Creating a new class is easy. For example:

> class(x) = "VUBiostat"

Later, when you type print(x), R will automatically look for the print. VUBiostat method.
Now, we redefine the ROC function. Instead of plotting the curve we return the data needed for the plot.
S3 Methods: ROC Example

```r
> print.ROC <- function(x, ...)
+ {
+  cat("ROC curve: ")
+  print(x$call)
+}

> plot.ROC <- function(x, xlab="1-Specificity", ylab="Sensitivity", type="l", ...){
+  plot(x$fpr, x$tpr, xlab=xlab, ylab=ylab, type=type, ...)
+}

> lines.ROC <- function(x, ...){
+  lines(x$fpr, x$tpr, ...)
+}

> identify.ROC <- function(x, labels=NULL, ..., digits=1)
+ {
+  if (is.null(labels))
+    labels<-round(x$cutpoints,digits)
+  identify(x$fpr, x$tpr, labels=labels,...)
+}
```

Now, we redefine the ROC function. Instead of plotting the curve we return the data needed for the plot.
S3 Methods: ROC Example

Now, when you call print(), plot(), lines(), or identify() on object with class ROC, R will use the corresponding functions that we just defined. The original print(), plot() remain unchange.

```r
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
```
S3 Method: ROC Example

```r
> plot
function (x, y, ...)
{
  if (is.function(x) && is.null(attr(x, "class"))) {
    if (missing(y))
      y <- NULL
    hasylab <- function(...) !all(is.na(pmatch(names(list(...)), "ylab")))
    if (hasylab(...))
      plot.function(x, y, ...)
    else plot.function(x, y, ylab = paste(deparse(substitute(x)), 
                      "(x)", ...)
  }
  else UseMethod("plot")
}
<environment: namespace:graphics>
>
Tools that are available

Premature optimization is the root of all evil — Donald Knuth

- `traceback()` shows where R was at the last error: what function it was in, where this was called from, and so on back to your top-level command
- `options(error=dump.frames)` saves the entire state of your program when an error occurs. `debugger()` then lets you start the debugger to inspect any function that was being run. `options(error=recover)` starts the debugger as soon as an error occurs.
- `browser()` starts the debugger at this point in your code
- `options(warn=2)` turns warnings into errors
- in R `debug(fname)` starts the debugger when function `fname()` is called.
Debug R Functions: ROC Example

```r
> ROC = function(test, disease)
+ {
+  cutpoints <- c(-Inf, sort(unique(test)), Inf)
+  sensitivity <- sapply(cutpoints,
+                        function(result) mean(test>result & disease)/mean(disease))
+  specificity <- sapply(cutpoints,
+                        function(result) mean(test<=result & !disease)/mean(!disease))
+  browser()
+  plot(sensitivity, 1-specificity, type="l")
+  abline(0,1,lty=2)
+  return(list(sens=sensitivity, spec=specificity))
+ }
```
> ROC(c(x,y), isx)
Called from: ROC(c(x, y), isx)
Browse[1]> head(sensitivity)
[1] 1.00 0.99 0.98 0.97 0.96 0.95
Browse[1]> head(specificity)
[1] 0 0 0 0 0 0
Browse[1]> n
debug: plot(sensitivity, 1 - specificity, type = "l")
Browse[2]> n
debug: abline(0, 1, lty = 2)
Browse[2]> n
debug: return(list(sens = sensitivity, spec = specificity))
Browse[2]> Q
>
Optimization

Goal: Balance computation speed, memory usage and human effort

Most functions and operators in R will operate on entire vectors, and the most efficient programming techniques in R are ones which utilize this approach. R does provide loops for more traditional programming, but they tend to be inefficient, especially for large problems.

- Operations on whole vectors are fast.
- Matrix operations may be faster even than naive C code.
- Functions that have few options and little error checking are faster: eg `sum(x)/length(x)` is faster than `mean(x)`
- Allocating memory all at once is faster than incremental allocation: `x<-numeric(10000); x[i]<-f(i)` rather than `x<-c(x,f(i))`
Optimization

Goal: Balance computing time, memory usage and human effort

- Data frames are much slower than matrices (especially large ones).
- Running out of memory makes code much slower, especially under Windows.
- If none of this works, coding a small part of the program in C may make it hundreds of times faster.
- Using apply functions has no real impact to make the code faster.
- Adding names to vectors make code slower.
- When vectorization is impossible, put as much outside of loops as possible and make number of iterations as small as possible.
When the code works correctly, the next step is to find out which parts, if any, are too slow, and then speed them up. This requires measurement, rather than guessing.

- `proc.time()`, which returns the current time. Save it before a task and subtract from the value after a task.
- `system.time()` to time the evaluation of an expression.
- In R, `Rprof(filename)` turns on the profiler, and `Rprof(NULL)` turns it off. The profiler writes a list of the current functions being run to filename many times per second. `summaryRprof(filename)` summarizes this to report how much time is spent in each function.
Speed: ROC example

```r
> Rprof();system.time(ROC(c(x,y), isx));Rprof(NULL)
user  system elapsed
0.45  0.03   0.51
> summaryRprof()
$by.self
  self.time self.pct total.time total.pct
&     0.12  21.43   0.12  21.43
mean   0.10  17.86   0.46  82.14
<=     0.10  17.86   0.10  17.86
mean.default   0.06  10.71   0.06  10.71
...
$by.total
  total.time total.pct self.time self.pct
system.time  0.56  100.00      0.00      0.00
ROC           0.52   92.86      0.00      0.00
mean         0.46   82.14      0.10   17.86
FUN          0.46   82.14      0.00      0.00
lapply       0.46   82.14      0.00      0.00
```
Memory

Sorry, you cannot do much about that.

You may have seen this message:

Error: cannot allocate vector of size 79.8 Mb.

It is not saying that you total memory is less than 79.8 Mb. The error message is about how much memory R was going after at the point where it failed. So, please clean your R workspace, especially remove some huge objects that are not going to be used any more.
Other Interesting Topics

Topics not yet covered

- Writing an R package
- Parallel computing
If you have any questions or comments, feel free to send me an email at Alex.Zhao@vanderbilt.edu
If you have any questions or comments, feel free to send me an email at Alex.Zhao@vanderbilt.edu

Thank you!
References

- “An Introduction to R”. Phil Spector, Sep. 24, 2004
- “R for Programmers”, Norman Matloff, Dec. 4 2008