R - 2ª PARTE

Introducción a la Ciencia de Datos

Index

- String Manipulation
- Input/Output
- Functions
- R Programming Structures
- Performance Enhancement: Speed and Memory

String manipulation

- Although R is a statistical language with numeric vectors and matrices playing a central role, character strings are also necessary and R has a number of string-manipulation utilities.

  - `nchar()`: finds the length of a string

```
nchar("Ciencia de Datos")
[1] 16
```

String manipulation

- `paste()`: concatenates several strings, returning the result in one long string.

```
paste("Ciencia", "de", "Datos")
[1] "Ciencia de Datos"
paste("Ciencia", "de", "Datos", sep="_")
[1] "Ciencia_de_Datos"
paste("Ciencia", 5+3, “cierta")
[1] "Ciencia 8 cierta”
paste(1:3,1:5, sep="_", collapse="|")
[1] “1_1|2_2|3_3|1_4|2_5"
```

String manipulation

- `substr()`: The call `substr(x, start, stop)` returns the substring in the given character position range `start:stop` in the given string x.

```
substr("Ciencia de Datos",1,5)
[1] "Cienc"
substr("Ciencia de Datos",12)
[1] "Datos"
```

String manipulation

- `strsplit()`: The call `strsplit(x, split)` splits a string x into an R list of substrings based on another string split in x.

```
strsplit("2-10-2017", split="-")
[[1]]
[1] "2"     "10"    "2017"
```

Regular expressions

- A regular expression is a kind of wild card.
- It's shorthand to specify broad classes of strings.
- In R, you must pay attention to this point when using the string functions `grep()`, `grepl()`, `regexpr()`, `gregexpr()`, `sub()`, `gsub()`, and `strsplit()`.

Regular expressions

- For example, the expression “[ia]” refers to any string that contains either of the letters *i* or *a*.

```
grep("[ia]",c("Ciencia","de","Datos"))
[1] 1 3
```

- A period (.) represents any single character.

```
grep(".e",c("Ciencia","de","Datos"))
[1] 1 2
```

Regular expressions

- Another example using `strsplit()`:

```
strsplit("a.b.c", ".")
[[1]]
[1] "" "" "" "" ""
strsplit("a.b.c", "[.]")
[[1]]
[1] "a" "b" "c"
```

Input/Output: Keyboard and Monitor

- Suppose we have a file (`file.txt`) with this content:

```
12
2 5
641
```

```
scan("file.txt")
Read 4 items
[1]  12   2   5 641
scan("file.txt",what=character())
Read 4 items
[1] "12"  "2"   "5"   “641"
scan(“file.txt",sep="\n")
Read 3 items
[1]  12  25 641
```

Input/Output: Keyboard and Monitor

- You can use `scan()` to read from the keyboard by specifying an empty string for the filename:

```
scan("")
1: 23 4
3: 2
4:
Read 3 items
[1] 23  4  2
```

- Note that we are prompted with the index of the next item to be input, and we signal the end of input with an empty line.

Input/Output: Keyboard and Monitor

- If you want to read in a single line from the keyboard use `readline()`:

```
readline("Input data: ")
Input data: 23 4 2
[1] "23  4  2"
```

- Note that we are prompted with the index of the next item to be input, and we signal the end of input with an empty line.

Input/Output: Print to the screen

- `print()` is a *generic* function, so the actual function called will depend on the class of the object that is printed. If, for example, the argument is of class table, then the `print.table()` function will be called.

```
x <- 1:3
print(x^2)
[1] 1 4 9
```

Input/Output: Print to the screen

- It's a little better to use `cat()` instead of `print()`, as the latter can print only one expression and its output is numbered:

```
x <- 1:3
print(x^2)
[1] 1 4 9
cat(x^2)
1 4 9
cat(x^2, x, "hola")
1 4 9 1 2 3 hola
cat(x^2, x, "hola", sep="_")
1_4_9_1_2_3_hola
```

Input/Output: Reading and Writing files

- We will use of the function `read.table()` to read in a data frame.
- Suppose we have a file `matrix.txt` with the following content:

```
nombre edad
John 25
Mary 28
Jim 19
```

Input/Output: Reading and Writing files

- The first line contains an optional header, specifying column names. We could read the file this way:

```
read.table("matrix.txt",header=TRUE)
  nombre edad
1   John   25
2   Mary   28
3    Jim   19
```

- Note that `scan()` would not work here, because our file has a mixture of numeric and character data (and a header).

Input/Output: Reading and Writing files

- If we want to write a file, we change `read.table()` for `write.table()` function:

```
write.table(matrix(1:6, nrow=2), "output.txt",
row.names=FALSE, col.names=FALSE)
```

```
output.txt:
1 3 5
2 4 6
```

Input/Output: Reading and Writing files

- The function `cat()` can also be used to write to a file, one part at a time:

```
cat("abc\n",file="u.txt")
cat("de\n",file="u.txt",append=TRUE)
```

```
u.txt:
abc
de
```

Functions

- A *function* is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

```
suma <- function(x,y) {
  x + y
}
suma(2,3)
[1] 5
```

Functions

- Default values:

```
suma <- function(x,y=5) {
  x + y
}
suma(2)
[1] 7
```

Functions

- Default values:

```
suma <- function(x=5,y) {
  x + y
}
suma(y=2)
[1] 7
```

Programming Structures

- R is a block-structured language (like C, C++, Python, Perl, and so on).
- Blocks are delineated by braces, while statements are separated by newline characters or, optionally, by semicolons.
- As with many scripting languages, we do not “declare” variables in R, therefore we have to take care of possible variable scoping issues.

Basic R operators

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>x + y</td><td>Addition</td></tr>
<tr><td>x - y</td><td>Subtraction</td></tr>
<tr><td>x * y</td><td>Multiplication</td></tr>
<tr><td>x / y</td><td>Division</td></tr>
<tr><td>x ^ y</td><td>Exponentiation</td></tr>
<tr><td>x %% y</td><td>Modular arithmetic</td></tr>
<tr><td>x %/% y</td><td>Integer division</td></tr>
<tr><td>x == y</td><td>Test for equality</td></tr>
<tr><td>x <= y</td><td>Test for less than or equal to</td></tr>
<tr><td>x >= y</td><td>Test for greater than or equal to</td></tr>
<tr><td>x && y</td><td>Boolean AND for scalars</td></tr>
<tr><td>x || y</td><td>Boolean OR for scalars</td></tr>
<tr><td>x & y</td><td>Boolean AND for vectors (vector x,y,result)</td></tr>
<tr><td>x | y</td><td>Boolean OR for vectors (vector x,y,result)</td></tr>
<tr><td>!x</td><td>Boolean negation</td></tr>
</table>

Control Statements: if-else

- The syntax for if-else looks like:

```
if (r == 4) {
    x <- 1
}
else {
    x <- 3
    y <- 4
}
```

Control Statements: if-else

- An if-else statement works as a function call, and as such, it returns the last value assigned.

```
if(x == 2) y <- x else y <- x+1

y <- if(x == 2) x else x+1
```

Control Statements: if-else

- When working with vectors, use the ifelse() function.
- The form is: `ifelse(b,u,v)` where **`b`** is a Boolean vector, and **`u`** and **`v`** are vectors.
- The return value is itself a vector: element i is u[i] if b[i] is true, or v[i] if b[i] is false.

Control Statements: if-else

```
x <- 1:10
ifelse(x %% 2 == 0,"par","impar")
```

```
[1] "impar" "par" "impar" "par" "impar"
"par" "impar" "par" "impar" "par"
```

```
x <- c(5,2,9,12)
ifelse(x > 6,2*x,3*x)
```

```
[1]15 6 18 24
```

Control Statements: loops

- One of the major themes of R programming is to avoid loops if possible; if not, keep loops simple.
- We have:
  - For loops
  - While loops
  - Repeat loops

Control Statements: loops

- For loops:

```
k <- 0

for(n in x) {
 if (n %% 2 == 1) k <- k+1

}

k

[1] 5
```

Control Statements: loops

- For loops:

```
k <- 0

for (i in 1:length(x)) {
   if (x[i] %% 2 == 1) k <- k+1
}

k

[1] 5
```

Control Statements: loops

- For loops:

```
for (fn in c("file1","file2")) print(scan(fn))
Read 6 items
[1] 1 2 3 4 5 6
Read 3 items
[1] 5 12 13
```

Control Statements: While loops

- While loops:

```
i <- 1
while (i <= 10) i <- i+4
i
```

```
[1] 13
```

Control Statements: Repeat loops

- Repeat loops:

```
i <- 1
repeat {
  i <- i+4
  if (i > 10) break
}
i
[1] 13
```

Looping, the R way…

- In R you have more options when doing repeating calculations:

function	input	output	comment
`apply`	matrix or array	vector or array or list	
`lapply`	list or vector	list	
`sapply`	list or vector	vector or matrix or list	simplify
`vapply`	list or vector	vector or matrix or list	safer simplify
`tapply`	data, categories	array or list	ragged
`mapply`	lists and/or vectors	vector or matrix or list	multiple
`rapply`	list	vector or list	recursive
`eapply`	environment	list	
`dendrapply`	dendogram	dendogram	
`rollapply`	data	similar to input	package `zoo`

The apply function

- This is the general form of apply for matrices:

```
apply(m,dimcode,f,fargs)
```

- where:
  - *m* is the matrix.
  - *dimcode* is the dimension, equal to 1 if the function applies to rows or 2 for columns.
  - *f* is the function to be applied.
  - *fargs* is an optional set of arguments to be supplied to *f*.

The apply function

- Apply over the margins of an array (e.g., the rows or columns of a matrix).

```
z <- matrix(c(1,1,2,2,3,3), nrow=3, byrow=TRUE)
apply(z,1,mean)
[1] 1 2 3
apply(z,2,mean)
[1] 2 2
apply(z,1:2,mean)
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
```

The lapply function

- Returns a list of the same length as the input data `X`, each element of which is the result of applying a function to the corresponding element of `X`.

```
z <- list(vector=c(1,2,3), matriz=matrix(1,nrow=2,ncol=2))
lapply(z,mean)
$vector
[1] 2
$matriz
[1] 1
```

The sapply function

- Apply over an object and return a simplified object (an array) if possible.

```
sapply(z, mean)
vector matriz
     2      1
```

Performance Enhancement: Speed and Memory

- In order to have a fast-running program, you may need to use more memory space.
- On the other hand, in order to conserve memory space, you might need to settle for slower code.
- R is an interpreted language.
  - Many of the commands are written in C and thus do run in fast machine code. But other commands, and your own R code, are pure R and thus interpreted.
- All objects in an R session are stored in memory.
  - More precisely, all objects are stored in R’s memory address space.

Performance Enhancement: Tips

- Optimize your R code through **vectorization**, use of byte-code compilation, and other approaches.
- Write the key, CPU-intensive parts of your code in a compiled language such as C/C++.
- Write your code in some form of parallel R.

Vectorization

- Loops:
  - It’s important to understand that simply rewriting code to avoid loops will not necessarily make the code faster.
  - However, in some cases, dramatic speedup may be attained, usually through vectorization.

Vectorization

```
for (i in 1:length(x)) z[i] <- x[i] + y[i]
```

vs.

```
z <- x + y
```

```
x <- runif(1000000)
y <- runif(1000000)
z <- vector(length=1000000)
system.time(for (i in 1:length(x)) z[i] <- x[i] + y[i])
   user  system elapsed
  1.504   0.102   1.724
system.time(z <- x + y)
   user  system elapsed
  0.002   0.004   0.007
```

Vectorization

- Examples of other vectorized functions that may speed up your code are `ifelse()`, `which()`, `where()`, `any()`, `all()`, `cumsum()`, and `cumprod()`.
- In the matrix case, you can use `rowSums()`, `colSums()`, and so on.
- In “all possible combinations” types of settings, `combin()`, `outer()`, `lower.tri()`, `upper.tri()`, or `expand.grid()` may be just what you need.
- Though `apply()` eliminates an explicit loop, it is actually implemented in R rather than C and thus will usually not speed up your code. However, the other apply functions, such as `lapply()`, can be very helpful in speeding up your code.

Performance Enhancement

- Slow algorithm in R

```
xs <- runif(1000)
res <- c()
for (x in xs) {
  # This is slow!
  res <- c(res, sqrt(x))
}
```

Performance Enhancement

- Faster algorithm in R

```
xs <- runif(1000)
res <- numeric(length(xs))
for (i in seq_along(xs)) {
  res[i] <- sqrt(xs[i])
}
```

Performance Enhancement

- Slower algorithm in R

```
amat <- matrix(1:20, nrow=4)
bmat <- matrix(NA, nrow(amat)/2, ncol(amat))
for(i in 1:nrow(bmat))
  bmat[i,] <- amat[2*i-1,] * amat[2*i,]
```

Performance Enhancement

- Faster algorithm in R

```
amat <- matrix(1:20, nrow=4)

bmat2 <- amat[seq(1, nrow(amat), by=2),] *
amat[seq(2, nrow(amat), by=2),]
```

Over-Vectorizing

- It is a good thing to want to vectorize when there is no effective way to do so. It is a bad thing to attempt it anyway.
- A common reflex is to use a function in the apply family. This is not vectorization, it is loop-hiding.
  - The apply function has a for loop in its definition.
  - Use an explicit for loop when each iteration is a non-trivial task. But a simple loop can be more clearly and compactly expressed using an apply function.

Over-Vectorizing

- Suppose that we want to find all of the sets of three positive integers that sum to 6, where the order matters:

```
the.seq <- 1:4
which(outer(outer(the.seq, the.seq, '+'),
the.seq, '+') == 6, arr.ind=TRUE)
```

Performance Enhancement

- Some things are not possible to vectorize.
- If you need to use a loop, then:
  - Put as much outside of loops as possible.Make the number of iterations as small as possible.

References

- Norman Matloff. 2011. The Art of R Programming: A Tour of Statistical Software Design (1st ed.). No Starch Press, San Francisco, CA, USA.
- Patrick Burns. 2011. The R Inferno.