R: Programming

140.776 Statistical Computing

September 8, 2011
Why programming?
Why programming
Why programming

140.776 Statistical Computing  R: Programming
Programming is more than just putting commands you’ve learnt so far into a *.R file. A key element of programming (which is also true for other languages) is that you can use control structures to control the flow of execution of the program.

For example, “for()” is a control structure in R to repeatedly execute a series of similar commands.
Control structures commonly used in R include:

- **if, else**: testing a condition
- **for**: execute a loop for a fixed number of times
- **while**: execute a loop while a condition is true
- **repeat**: execute a loop until seeing a break
- **break**: break the execution of a loop
- **next**: skip an iteration of a loop
- **return**: exit a function
if(<condition>){
    ## do something
}
else {
    ## do something else
}

if(<condition1>){
    ## do something
} else if (<condition2>){
    ## do something different
} else {
    ## do something else
}
Example: compute the absolute value of x and assign it to y.

```r
if(x<0) {
  y<-(-x)
} else {
  y<-x
}
```
The else clause is not necessary:

```r
if(<condition1>) {
    ## do something
}
```

is equivalent to

```r
if(<condition1>) {
    ## do something
} else {
    ## do nothing
}
```
Conditions often use && (AND) and || (OR).

```r
if(x>0 && x<1) {
  y<-x^2
} else {
  y<-x^4
}
```
&& (AND) and || (OR) in conditions

```r
> x<-c(1>2,2<3,3==4)
> x
[1] FALSE TRUE FALSE
> y<-c(1<2,2<3,3!=4)
> y
[1] TRUE TRUE TRUE
> x&&y
```

R: Programming
&& (AND) and || (OR) in conditions

```r
> x
[1] FALSE TRUE FALSE
> y
[1] TRUE TRUE TRUE

> x&&y
[1] FALSE

> x&y
```
&& (AND) and || (OR) in conditions

```r
> x
[1] FALSE  TRUE FALSE

> y
[1] TRUE  TRUE  TRUE

> x && y
[1] FALSE

> x & y
[1] FALSE  TRUE FALSE
```
&& and || are different from & and |:

- The shorter form (& and |) performs elementwise comparisons in much the same way as arithmetic operators.
- The longer form (&& and ||) evaluates left to right, examining only the first element of each vector. Evaluation proceeds only until the result is determined.
&& (AND) and || (OR) in conditions

1<2 || 2>3 && 1>2
&& (AND) and || (OR) in conditions

Compare the following three expressions:

1. \(1<2 \text{ || } 2>3 \text{ && } 1>2\)
   
   [1] TRUE

2. \((1<2 \text{ || } 2>3) \text{ && } 1>2\)
   
   [1] FALSE

3. \(1<2 \text{ || } (2>3 \text{ && } 1>2)\)
   
   [1] TRUE

Why do you obtain different results?
In R, operators belong to different precedence groups. && has higher precedence than ||, therefore && is evaluated first.

About precedence of operators:

- Use help(Syntax) to learn precedence of operators.
- Within an expression, operators of equal precedence are evaluated from left to right.
- If you are not sure about which operator is evaluated first, I recommend you to explicitly specify the priority by using (()).
- There are substantial precedence differences between R and S. For example, in S, &, &&, | and || have equal precedence.
Repetitive execution: for loops

for(var in seq) {
    expr
}

For loops are commonly used for iterating over the element of an object (list, vector, etc.). For example:

for(i in 1:10) {
    print(i)
}
These loops have the same behavior:

```r
x <- c("a", "b", "c", "d")

for(i in 1:4) {
    print(x[i])
}

for(i in seq_along(x)) {
    print(x[i])
}

for(letter in x) {
    print(letter)
}

for(i in 1:4) print(x[i])
```
> load("apple-banana-array.rda")
Loops can be nested:

```r
x<-matrix(1:60,6,10)

for(i in seq_len(nrow(x))) {
    for(j in seq_len(ncol(x))) {
        print(x[i,j])
    }
}
```
while(cond) {
    expr
}

While loops evaluate a condition repetitively. If the condition is true, then the expression in the loop body is executed. Otherwise, the loop will be ended. For example:

count<-0
while(count<10) {
    print(count)
    count<-count+1
}
Another example:

```r
## simulate a random walk
z<-5
while(z>=3 && z<=10) {
  print(z)
  coin<-rbinom(1,1,0.5)
  if(coin == 1) {
    z<-z+1
  } else {
    z<-z-1
  }
}
```
repeat { 
    expr 
} 

This statement executes the expression in the loop repeatedly until it sees a \textit{break}. For example:

\begin{verbatim}
x0<-1
tol<-1e-8
repeat {
x1<-computeEstimate()

    if(abs(x1-x0)<tol) {
        break
    } else {
        x0<-x1
    }
}
\end{verbatim}
The `break` statement can be used to terminate any loop. It is the only way to terminate repeat loops. For example:

```r
x0<-1
tol<-1e-8
err<-10
iter<-0
while (err>tol) {
    x1<-computeEstimate()
    err<-abs(x1-x0)
    x0<-x1
    iter<-iter+1
    if(iter == 100) {
        break
    }
}
```
next is used to skip an iteration of a loop.

```r
for(i in 1:5) {
    if(i<=3) {
        next
    }
    print(i)
}
```

[1] 4
[1] 5

return signals that a function should exit and return a given value.
> load("apple-banana-list.rda")

(name="apple", nextnode)
    |
    V
(name = "apple", nextnode)
    |
    . . .
    V
(name = "banana", nextnode)
    |
    . . .
    V
NA
There are 4 main functions for signalling or handling conditions (i.e. unusual situations) in R.

- **message**: print a message to the console (not necessarily a bad thing)
- **warning**: non-fatal problem; print a message to the console
- **stop**: problem is fatal, execution of the program is halted
- **try, tryCatch**: testing for conditions and executing alternate code (exception handling)
for(i in seq_along(x)) {
    if(<minor condition>) {
        message("a minor condition occurred")
    }
    if(<more serious condition>) {
        warning("something unusual is going on")
    }
    if(<fatal condition>) {
        stop("cannot continue, aborting")
    }
}
Good habits

- **Correct grammar**
  R code: immediately source-able; C code: can be compiled without errors

- **Correct results**
  Produce logically correct answer

- **Code readability**
  Use monospace font; <80 characters/line; indent your code; comment your codes

- **Code efficiency**
  Organize into functional modules; keep the code short if possible

- **Computational efficiency**
  Whoever runs fastest wins
Grading criteria

Example:

correctness 60%
+ computational efficiency 20%
+ readability 10%
+ code efficiency 10%
=
100%