Control

V. Batagelj

Sequences of expressions

Branching

Loops

Jumps

Functions

Control

Summer School on R

Vladimir Batagelj

University of Ljubljana, FMF, Mathematics

Quantitative methods in the Social Sciences 2
University of Bucharest, Romania
September 1-7, 2010

Outline

1. Sequences of expressions
2. Branching
3. Loops
4. Jumps
5. Functions

Sequences of expressions

A *program* in R is a *sequence* of *expressions*. They are separated by either a semi-colon, `;`, or a newline.

The expressions in a sequence are evaluated in the same order as they appear in the sequence. In the interactive execution of a program the value of an expression is printed if it is not an assignment.

The value of the most recently evaluated non-assignment expression is stored in the variable `.Last.value` .

```
1  > a <- 3; a; b <- 5; a+b
2  [1] 3
3  [1] 8
4  > c <- a+b
5  > .Last.value
6  [1] 8
7  > (d <- c+2)
8  [1] 10
```

Blocks

A sequence of expressions can be made an expression, called *compound expression* or *block*, by enclosing it within braces
{ $expression_1$; $expression_2$; . . . ; $expression_k$ }
The value of the compound expression is equal to the value of its last expression.

```
1 > {p <- 3; s <- 2*p; s+5}
2 [1] 11
3 > ({p <- 3; s <- 2*p; q <- s+5})
4 [1] 11
```

Branching

Control

V. Batagelj

Sequences of expressions

Branching

Loops

Jumps

Functions

The control expressions (statements), `help(Control)`, enable us to branch or repeat the evaluation.

A *condition* is an expression with (a single) logical value (`TRUE` or `FALSE`).

Simple conditional

`if(`*condition*`)` *expression*

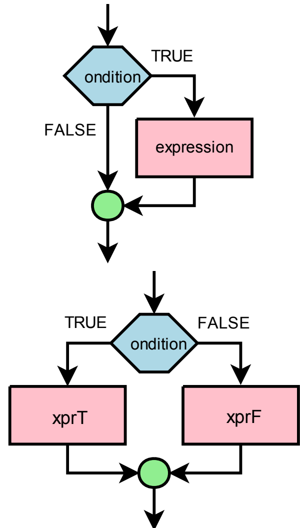
if the *condition* is satisfied it evaluates and returns the value of the *expression*; otherwise it returns the value `NULL`.

Branching conditional

`if(`*condition*`)` *exprT* `else` *exprF*

if the *condition* is satisfied it evaluates and returns the value of the *exprT*; otherwise it evaluates and returns the value of the *exprF*.

See also `switch` and `ifelse`.



Branching – examples


```
1  > a <- 3
2  > (if (a > 2) "OK")
3  [1] "OK"
4  > a <- 0
5  > (if (a > 2) "OK")
6  NULL
7  > a <- 3
8  > (b <- if (a<0) "N" else "P")
9  [1] "P"
10 > a <- -3
11 > (b <- if (a<0) "N" else "P")
12 [1] "N"
```

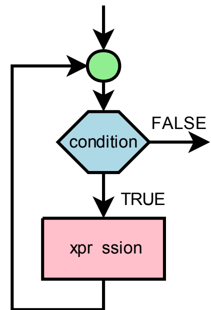
Loops

Loops enable us the repetitive evaluation of an expression.

while loop

`while(`*condition*`)` *expression*

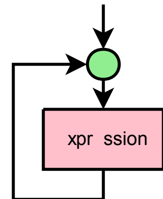
repeats the following: if the *condition* is satisfied it evaluates the *expression*; otherwise it breaks the repetition and continues with the following expression.



repeat loop

`repeat` *expression*

repeats evaluating the *expression* until inside the *expression* a break is requested.



Loops – examples

repeat n times

```
1  > i <- 0; n <- 5
2  > while(i<n){ i <- i+1; print(i) }
3  [1] 1
4  [1] 2
5  [1] 3
6  [1] 4
7  [1] 5
```

sum of elements of vector x

```
1  > x <- c(3,7,8,5,1,3,6)
2  > i <- 0; s <- 0
3  > while(i<length(x)){ i <- i+1; s <- s+x[[i]] }
4  > print(s)
5  [1] 33
```

Loops

for loop

The statement

for(*v* in *V*) *expression*

requires sequential evaluation of the *expression* for all values *v* from the sequence *V*.

```
1  > s <- 0; for(i in 1:10) s <- s+i
2  > cat("i =",i,"  s =",s,"\n")
3  i = 10   s = 55
4  > sum(1:10)
5  [1] 55
6  > for(c in unlist(strsplit("Text",""))) print(c)
7  [1] "T"
8  [1] "e"
9  [1] "x"
10 [1] "t"
```

Timing

Most tasks with vectors can be done using vector operations and functions `*apply` that are much faster than `for` loops.

```
1  > system.time({a <- NULL; for(i in 1:50000) a <- c(a,sin(runif(1)))})
2     user  system elapsed
3    13.44    0.02   12.65
4  > system.time({b <- numeric(50000)
5  +   for(i in 1:50000) b[[i]] <- sin(runif(1))})
6     user  system elapsed
7     0.87    0.00    0.78
8  > system.time({c <- sin(runif(50000))})
9     user  system elapsed
10    0.03    0.00    0.04
```

Jumps

Control

V. Batagelj

Sequences of expressions

Branching

Loops

Jumps

Functions

Exit the loop

```
break
```

requires the exit of the innermost loop that contains it.

Back to start

```
next
```

requires the transition of execution flow to the first expression in the innermost loop that contains it.

The loops return the value `NULL`.

There are some additional control statements: `stop(`*message*`)`, `stopifnot(`$cond_1$`,`$cond_2$`,`$\cdots$ $cond_k$`)`, `return(`*expression*`)`, `tryCatch(`*expression*`,...,finally=`*exitExpression*`)`

Hints

Attention: in control statements the *condition* should be enclosed in braces ( ) .
In conditions use || and &&.
In the branching conditional the part else has to be in the same line as the end of expression *exprT*.

```
if(condition) {
   exprT
} else {
   exprF
}
```

When a control statement contains a compound expression we *indent* its expressions to show the logical structure and increase readability.

Jumps – example

Number of throws of dice until 6 appears

```
1  > k <- 0
2  > repeat {
3  +   k <- k+1
4  +   dice <- 1+trunc(6*runif(1))
5  +   if (dice==6) break
6  + }
7  > print(k)
8  [1] 2
```

Jumps – example

Guess the number

```
1  > m <- 50
2  > a <- 1+trunc(m*runif(1))
3  > s <- 0
4  > repeat{
5  +    s <- s+1
6  +    g <- as.integer(readline(paste(s,". Your guess = ",sep="")))
7  +    if(a < g) {cat("smaller\n"); next}
8  +    if(a > g) {cat("larger\n"); next}
9  +    break
10 + }
11 1. Your guess = 25
12 larger
13 2. Your guess = 37
14 larger
15 3. Your guess = 44
16 larger
17 4. Your guess = 47
18 smaller
19 5. Your guess = 45
20 > c(s,a,g)
21 [1]  5 45 45
```

Functions

Expression

`function(`*arguments*`)` *expression*

creates a *function*. If we assign it to a name we obtain a named function.

arguments is a comma-separated list of (formal) arguments – function's input data.

An argument can have either the form *name* or the form *name* = *expression*. Argument of the form *name* = *expression* defines a default value that is used if the value of this argument is not given in the call of the function.

expression determines how the value of the function is computed.

The execution of the *expression* can be terminated using the expression `return(`*valueExpr*`)` that requires that the value of *valueExpr* is returned as the function's value. If it terminates without `return` its value is the last computed expression value.

Functions

We use (call) the function with expression of the form
fun(*actual_arguments*)
where *fun* is a function or a function name, *actual_arguments* determine the values of function's arguments. To the arguments of the form *name* the values of actual arguments have to be supplied in the same order as they are listed in the function definition.
Arguments of the form *name* = *expression* can follow in any order.
Definition and call of the function should contain ( ) also in the case when the function has no argument.
If a function has ... as a formal argument then any actual arguments that do not match a formal argument are matched with ....
Functions are used to: structure larger programs – levels of abstraction; eliminate repetitions of similar parts of program; increase readability; division of work – black box approach – libraries of functions.

Scope of variables

The *scope* of a variable tells us where a variable is "visible". Variables defined within the function are *local* – visible only inside the function. We can use the variable with the same name in different functions without risk of their clash.
Variables defined in the interactive input are *global* – visible in any user-defined function and functions defined in it.
We can control the scope of variables also by *environments* and *namespaces* in packages.

```
args, body, formals, environment, alist, debug,
invisible
help("function")
```

Functions – examples


```
> (function(x) x^2-x+41)(3)
[1] 47
> ascii <- function(char) {
+ a <- as.integer(charToRaw(char))
+ if (length(a)>1) NA else a[1] }
> ascii("A")
[1] 65
> ascii("a")
[1] 97
> ascii("\u263A")
[1] NA
> gcd <- function(a,b)
+ if (b==0) abs(a) else gcd(b,a%%b)
> gcd(12,21)
[1] 3
> gcd(624,918)
[1] 6
> "%m%" <- function(a,b) min(a,b)
> "%M%" <- function(a,b) max(a,b)
> 4 %m% 3 %M% 5
[1] 5
> set <- function(x) union(x,NULL)
> card <- function(x) length(set(x))
> is.set <- function(x) length(x)==card(x)
> subseteq <- function(x,y){setequal(intersect(x,y),x)}
> charSet <- function(z)
+ union(substring(tolower(z),1:nchar(z),1:nchar(z)),NULL)
```

Functions – examples


```
1   > dif <- function(a,b) return(a-b)
2   > dif(7,3)
3   [1] 4
4   > dif(b=3,a=7)
5   [1] 4
6   > dif(b=3,7)
7   [1] 4
8   > dif(3,a=7)
9   [1] 4
10  > x <- 3
11  > f <- function(x,a=7,u,z="a"){
12  +   x <- x+3
13  +   cat("f: x=",x," a=",a," u=",u," z=",z,"\n",sep="")
14  +   return(x+u)
15  + }
16  > g <- function(p,q,...){
17  +   v <- f(u=p,q,...)
18  +   cat("g: x=",x," p=",p," q=",q," v=",v," ...=",...,"\n",sep="")
19  +   return(v)
20  + }
21  > g(2,11,z="b")
22  f: x=14 a=7 u=2 z=b
23  g: x=3 p=2 q=11 v=16 ...=b
24  [1] 16
```

Functions – examples


```
> counter
Error: object "counter" not found
> count <- function(){
  if (!exists("counter")) counter <<- 0;
  counter <<- counter+1; counter}
> count()
[1] 1
> count()
[1] 2
> count()
[1] 3
> count()
[1] 4
> counter
[1] 4

> eval(parse(text="x <- 5"))
> x
[1] 5
> run <- function(s) eval(parse(text=s))
> run("x <- 6; x")
[1] 6
> source(textConnection("y <- 14; z <- y*(y+1); print(z)"))
[1] 210
```

Write to file and then use it as a source.