8.1.1 Come analizzare i dati: R Language

Insegnamento di Informatica

Elisabetta Ronchieri

Corso di Laurea di Economia, Universitá di Ferrara

I semestre, anno 2014-2015

Argomenti

Argomenti

Control Structures

Control structures, supporting the control of flow of execution of the program, are:

- if, else: testing a condition;
- for: execute a loop a fixed number of times;
- while: execute a loop while a condition is true;
- repeat: execute an infinite loop;
- break: break the execution of a loop;
- next: skip an interation of a loop;
- return: exit a function;

Most control structures are used when writing functions or longer expresisons.

if

```
if(<condition>) {
  ## do something
} else {
  ## do something else
}

if(<condition1>) {
  ## do something
} else if(<condition2>) {
  ## do something different
} else {
  ## do something different
}
```

The else clause is not necessary:

```
if(<condition3>) {
  ## do something
}
```

if

Example:

```
> if(x > 3) {
+ y <- 10
+ } else {
+ y <- 0
+ }

> y <- if(x > 3) {
+ 10
+ } else {
+ 0
+ }
```

for

For loops take an iterator variable and assign it successive values from a sequence or vector.
For loops are most commonly used for iterating over the elements of an object such as list and vector.
This loop takes the *i* variable and in each iteration of the loop gives it values $1, 2, 3, ..., 10$, and then exits.

```
> for(i in 1:10) {
+ print(i)
+ }
```

for

Example: These loops have the same behavior.

```
> x <- c("a", "b", "c", "d")
> for(i in 1:4) {
+ print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
> for(i in 1:4) print(x[i])
```

for

The seq_along() function returns an integer vector.
Example:

```
> x <- c("a", "b", "c", "d")
> for(i in seq_along(x)) {
+ print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

for

For loops can be nested, but be careful. Nesting beyond 2-3 levels is often very difficult to read/understand.
The seq_len() function returns an integer vector.

```
> x <- matrix(1:6, 2, 3)
> for(i in seq_len(nrow(x))) {
+ for(j in seq_len(ncol(x))) {
+ print(x[i, j])
+ }
+ }
[1] 1
[1] 3
[1] 5
[1] 2
[1] 4
[1] 6
```

while

While loops begin by testing a condition: if it is true, then they execute the loop body.
Once the loop body is executed, the condition is tested again, and so forth.

```
> count <- 0
> while(count < 10) {
+ print(count)
+ count <- count + 1
+ }
```

While loops can potentially result in infinite loops if not written properly.

while

Sometimes there will be more than one condition in the test.
The runif function returns a random number between 5.0 and 7.5.

```
> z <- 5
> while (z >= 3 && z <= 10) {
+ print(z)
+ coin <- runif(1, 5.0, 7.5)
+ if (coin == 1) {
+ z <- z + 1
+ } else {
+ z <- z - 1
+ }
+ }
```

Conditions are always evaluated from left to right.

repeat

Repeat initiates an infinite loop: the only way to exit a repeat loop is to call break.

```
> x0 <- 1
> tol <- 1e-8
> repeat {
+ x1 <- runif(1, 5.0, 7.5)
+ if(abs(x1 - x0) < tol) {
+ break
+ } else {
+ x0 <- x1
+ }
+ }
```

next

next is used to skip an iteration of a loop.

```
> for(i in 1:100) {
+ if(i <= 20) {
+ ## Skip the first 20 iterations
+ next
+ }
+ ## Do something here
+ }
```

Functions

Functions are created using the function() directive and are stored as R objects just like anything else.
They are R objects of class function.

```
> f <- function(<arguments>) {
+ ## Do something interesting
+ }
```

Functions can be passed as arguments to other functions.
Functions can be nested, so that you can define a function inside of another function
The return value of a function is the last expression in the function body to be evaluated.

Arguments

- Functions have named arguments which potentially have default values.
- Named arguments:
  - are useful when you want to use the defaults.
  - also help if you can remember the name of the argument and not its position on the argument list.
- Function arguments can be:
  - missing or might have default values;
  - matched positionally or by name.
- The ... argument indicates a variable number of arguments that are usually passed on to other functions.
- ... is often used when extending another function and you don't want to copy the entire argument list of the original function

Defining a function

Example: *b* , *c* and *d* are named arguments.

```
> f <- function(a, b = 1, c = 2, d = NULL) {
+ print(a)
+ }
```

In addition to not specifying a default value, you can also set an argument value to NULL.

Evaluation

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

```
> f <- function(a, b) {
+ a^2
+ }
> f(2)
4
```

This function never actually uses the argument *b*.

Evaluation

```
> f <- function(a, b) {
+ print(a)
+ print(b)
+ }
> f(40)
[1] 40
Error in print(b) : argument "b" is missing, with no
default
>
```

Notice that 40 got printed first before the error was triggered. This is because b did not have to be evaluated until after $print(a)$. Once the function tried to evaluate $print(b)$ it had to throw an error.

Debugging

Debugging indicates a problem through one of the following ways:

- message: a generic notification/diagnostic message produced by the message function.
- warning: an indication that something is wrong but not necessarily fatal.
- error: an indication that a fatal problem has occurred during execution stops or produced by the stop function.
- condition: a generic concept for indicating that something unexpected can occur.

Per ulteriori letture

- W. N. Venables, D. M. Smith and the R Core Team, *An Introduction to R*, July 2014, `http://cran.r-project.org/doc/manuals/R-intro.pdf`
- Vito M. R. Muggeo, Giancarlo Ferrara, *Il Linguaggio R: concetti introduttivi ed esempi*, 2005, `http://cran.r-project.org/doc/contrib/nozioniR.pdf`
- Josef Eschgfaller, *Programmare in R*, 2005, `http://cran.r-project.org/doc/contrib/Fondamenti-0405.pdf`