Elements of a programming language – 4

Marcin Kierczak

23 October 2016

Contents of the lecture

- variables and their types
- operators
- vectors
- numbers as vectors
- strings as vectors
- matrices
- lists
- data frames
- objects
- **repeating actions: iteration and recursion**
- **decision taking: control structures**
- **functions in general**
- **variable scope**
- **base functions**

Repeating actions

In several algorithms, the point is to repeat certain action several times. In a mathematical formulas language, we have for instance the following signs for repeating an action:

$$\Sigma_{i=1}^{n}(expression)$$

which denotes addition over elements with indices $1 \ldots n$ or

$$\Pi_{i=1}^{n}(expression)$$

which denotes multiplication.

It is important to learn how to translate these (and similar) formulas into the R language.

Repeating actions – for loop

One way to repeat an action is to use the **for-loop**

```
for (i in 1:5) {
  cat(paste('Performing operation no.', i), '\n')
}
```

```
## Performing operation no. 1
## Performing operation no. 2
## Performing operation no. 3
## Performing operation no. 4
## Performing operation no. 5
```

Repeating actions – for loop cted.

A slight modification of the above example will skip odd indices.

```
for (i in c(2,4,6,8,10)) {
  cat(paste('Performing operation no.', i), '\n')
}
```

```
## Performing operation no. 2
## Performing operation no. 4
## Performing operation no. 6
## Performing operation no. 8
## Performing operation no. 10
```

Repeating actions – for loop external counter

Sometimes, we also want an external counter:

```
cnt <- 1
for (i in c(2,4,6,8,10)) {
  cat(paste('Performing operation no.', cnt,
            'on element', i), '\n')
  cnt <- cnt + 1
}
```

```
## Performing operation no. 1 on element 2
## Performing operation no. 2 on element 4
## Performing operation no. 3 on element 6
## Performing operation no. 4 on element 8
## Performing operation no. 5 on element 10
```

Repeating actions – for loop an example

Say, we want to add 1 to every element of a vector:

```
vec <- c(1:5)
vec
```

```
## [1] 1 2 3 4 5
```

```
for (i in vec) {
  vec[i] <- vec[i] + 1
}
vec
```

```
## [1] 2 3 4 5 6
```

Repeating actions – avoid loops and vectorize!

The above can be achieved in R by means of vectorization:

```
vec <- c(1:5)
vec + 1
```

```
## [1] 2 3 4 5 6
```

Let us compare the time of execution of the vectorized version (vector with 10,000 elements):

```
##    user  system elapsed
##   0.038   0.003   0.041
```

to the loop version:

```
##    user  system elapsed
##   1.089   0.022   1.124
```

Repeating actions – the while loop

There is also another type of loop inR, the **while loop** which is executed until some condition is true.

```
x <- 1
while (x < 5) {
  cat(x, " ... ")
  x <- x + 1
}
```

```
## 1  ... 2  ... 3  ... 4  ...
```

Recursion

When we explicitely repeat an action using a loop, we talk about **iteration**. We can also repeat actions by means of **recursion**, i.e. when a function calls itself. Let us implement a factorial !:

```
factorial.rec <- function(x) {
  if (x == 0 || x == 1)
    return(1)
  else
    return(x * factorial(x - 1)) # Recursive call!
}
factorial.rec(5)
```

```
## [1] 120
```

Recursion = iteration?

Yes, every iteration can be converted to recursion (Church-Turing conjecture) and vice-versa. It is not always obvious, but theoretically it is doable. Let's see how to implement *factorial* in iterative manner:

```
factorial.iter <- function(x) {
  if (x == 0 || x == 1)
    return(1)
  else {
    tmp <- 1
    for (i in 2:x) {
      tmp <- tmp * i
    }
    return(tmp)
  }
}
factorial.iter(5)
```

Recursion == iteration, really?

More writing for the iterative version, right? What about the time efficiency?
The recursive version:

```
## [1] 2.432902e+18
```

```
##    user  system elapsed
##   0.003   0.001   0.003
```

And the iterative one:

```
## [1] 2.432902e+18
```

```
##    user  system elapsed
##   0.003   0.000   0.003
```

Loops – void growing data

Avoid changing dimensions of an object inside the loop:

```
v <- c() # Initialize
for (i in 1:100) {
  v <- c(v, i)
}
```

It is much better to do it like this:

```
v <- rep(NA, 100) # Initialize with length
for (i in 1:100) {
  v[i] <- i
}
```

Always try to know the size of the object you are going to create!

Decision taking – an if clause

Often, one has to take a different course of action depending on a flow of the algorithm. You have already seen the **if-else** block. Let's print only odd numbers $[1, 10]$:

```
v <- 1:10
for (i in v) {
  if (i %% 2 != 0) { # if clause
    cat(i, ' ')
  }
}
```

```
## 1  3  5  7  9
```

Decision taking – if-else

If we want to print 'o' for an odd number and 'e' for an even, we could write either:

```
v <- 1:10
for (i in v) {
  if (i %% 2 != 0) { # if clause
    cat('o ')
  }
  if (i %% 2 == 0) { # another if-clause
    cat('e ')
  }
}
```

```
## o e o e o e o e o e
```

Decision taking – if-else

or

```
v <- 1:10
for (i in v) {
  if (i %% 2 != 0) { # if clause
    cat('o ')
  } else { # another if-clause
    cat('e ')
  }
}
```

```
## o e o e o e o e o e
```

Decision taking – if-else

or else

```
v <- 1:10
for (i in v) {
  tmp <- 'e ' # set default to even
  if (i %% 2 != 0) { # if clause
    tmp <- 'o ' # change default for odd numbers
  }
  cat(tmp)
}
```

```
## o e o e o e o e o e
```

Each three are ways are good and are mainly the matter of style...

Decision taking – more alternatives

So far, so good, but we were only dealing with 3 alternatives. Let's say that we want to print '?' for zero, 'e' for even and 'o' for an odd number:

```
v <- 0:10
for (i in v) {
  if (i == 0) {
    cat('? ')
  } else if (i %% 2 != 0) { # if clause
    cat('o ')
  } else { # another if-clause
    cat('e ')
  }
}
```

```
## ? o e o e o e o e o e
```

Congratulations! You have just learned the **if-else if-else** clause.

Switch

If-else clauses operate on logical values. What if we want to take decisions based on non-logical values? Well, if-else will still work by evaluating a number of comparisons, but we can also use **switch**:

```
switch.demo <- function(x) {
  switch(class(x),
         logical = ,
         numeric = cat('Numeric or logical.'),
         factor = cat('Factor.'),
         cat('Undefined')
         )
}
switch.demo(x=TRUE)
```

```
## Numeric or logical.
```

```
switch.demo(x=15)
```

Functions 1

Often, it is really handy to re-use some code we have written or to pack together the code that is doing some task. Functions are a really good way to do this in R:

```
add.one <- function(arg1) {
  arg1 <- arg1 + 1
  return(arg1)
}
add.one(1)
```

```
## [1] 2
```

```
add.one()
```

```
## Error in add.one(): argument "arg1" is missing, with no
```

Anatomy of a function

A function consists of: formal arguments, function body and environment:

```
formals(ecdf)
```

```
## $x
```

```
body(plot.ecdf)
```

```
## {
##     plot.stepfun(x, ..., ylab = ylab, verticals = vertic
##         pch = pch)
##     abline(h = c(0, 1), col = col.01line, lty = 2)
## }
```

```
environment(ecdf)
```

```
## <environment: namespace:stats>
```

Functions – default values

Sometimes, it is good to use default values for some arguments:

```r
add.a.num <- function(arg, num=1) {
  arg <- arg + num
  return(arg)
}
add.a.num(1, 5)
```

```
## [1] 6
```

```r
add.a.num(1) # skip the num argument?
```

```
## [1] 2
```

```r
add.a.num(num=1) # skip the num argument?
```

```
## Error in add.a.num(num = 1): argument "arg" is missing,
```

Functions – order of arguments

```
args.demo <- function(x, y, arg3) {
  print(paste('x =', x, 'y =', y, 'arg3 =', arg3))
}
args.demo(1,2,3)
```

```
## [1] "x = 1 y = 2 arg3 = 3"
```

```
args.demo(x=1, y=2, arg3=3)
```

```
## [1] "x = 1 y = 2 arg3 = 3"
```

```
args.demo(x=1, 2, 3)
```

```
## [1] "x = 1 y = 2 arg3 = 3"
```

```
args.demo(a=3, x=1, y=2)
```

Functions – order of arguments 2

```
args.demo2 <- function(x, arg2, arg3) {
  print(paste('x =', x, 'arg2 =', arg2, 'arg3 =', arg3))
}
args.demo2(x=1, y=2, ar=3)
```

```
## Error in args.demo2(x = 1, y = 2, ar = 3): argument 3 ma
```

Functions – variables scope

Functions 'see' not only what has been passed to them as arguments:

```r
x <- 7
y <- 3
xyplus <- function(x) {
  x <- x + y
  return(x)
}
y <- xyplus(x)
y
```

```
## [1] 10
```

Functions – variables scope cted.

Everything outside the function is called **global environment**. There is a special operator for working on global environment from within a function:

```
x <- 1
xplus <- function(x) {
  x <<- x + 1
}
xplus(x)
x
```

```
## [1] 2
```

```
xplus(x)
x
```

```
## [1] 3
```

There is a special argument `...` (ellipsis) which allowes you to give any number of arguments or pass arguments downstream:

```
c # Any number of arguments
```

```
## function (..., recursive = FALSE)  .Primitive("c")
```

```
my.plot <- function(x, y, ...) { # Passing downstream
  plot(x, y, las=1, cex.axis=.8, ...)
}
my.plot(1,1)
```

Functions – the dot-dot-dot argument trick

What if the authors of, e.g. plot.something wrapper forgot about the dot-dot-dot?

```
c # Any number of arguments
```

```
## function (..., recursive = FALSE)  .Primitive("c")
```

```
my.plot <- function(x, y) { # Passing downstrem
  plot(x, y, las=1, cex.axis=.8, ...)
}
formals(my.plot) <- c(formals(my.plot), alist(... = ))
my.plot(1, 1, col='red', pch=19)
```

Lazy evaluation

In R, arguments are evaluated as late as possible, i.e. when they are needed. This is **lazy evaluation**:

```
h <- function(a = 1, b = d) {
  d <- (a + 1) ^ 2
  c(a, b)
}
h()
```

```
## [1] 1 4
```

The above won't be possible in, e.g. C where values of both arguments have to be known before calling a function **eager evaluation**.

In R everything is a function

Because in R everything is a function, we cqn redefine things:

```
`(`
```

```
## .Primitive("(")
```

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
```

```
##  [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
rm("(")
```

Infix notation

Operators like '+', '-' or '*' are using the so-called **infix** functions, where the function name is between arguments. We can define our own:

```
`%p%` <- function(x, y) {
  paste(x,y)
}
'a' %p% 'b'
```

```
## [1] "a b"
```

Base functions

When we start R, the following packages are pre-loaded automatically:

```
.libPaths() # get library location
```

```
## [1] "/Library/Frameworks/R.framework/Versions/3.3/Resour
```

```
# library()    # see all packages installed
search()     # see packages currently loaded
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:gra
```

Check what basic functions are offered by packages: *base*, *utils* and we will soon work with package *graphics*. If you want to see what statistical functions are in your arsenal, check out package *stats*.