R flow of control and conditional execution

Alberto Garfagnini

Università di Padova

AA 2023/2024 - R lecture 3

Let's recall what we learned so far ...

- all types are vectors
- there is no "scalar" type in R
- but sometimes you have vectors of length one
- R is a dynamically-typed language
- ➜ variables do not need to be declared

Internal data representation

- floating point numbers are stored as `double` (64-bit)
- numbers and sequences are represented as `integer`

Inspecting data

Function	Description
`class()`	
`typeof()`	show the R type or storage mode of an object
`storage.mode()`	get/set the storage mode of an object
`length()`	get/set the length of a vector
`attributes()`	access object's attributes
`str()`	compactly display the internal structure of an R object

Floating point precision limits

- only operations among `integers` are exact

```
u <- sqrt(2)
u * u == 2
# [1] FALSE

u * u - 2
# [1] 4.440892e-16

print(u * u)
# [1] 2

print(u * u, digits = 18)
# [1] 2.00000000000000044
```

- the `all.equal()` function compares R objects testing 'near equality'
- the `identical()` function tests two R objects for being exactly equal

```
identical(u * u, 2)
# [1] FALSE
all.equal(u * u, 2)
# [1] TRUE
```

➜ check R FAQ 7.31 - Why doesn't R think these numbers are equal?

R flow of control

Overview

- `if / else`
- `switch()`
- `for` loops
- `while`
- `ifelse()`
- `function()`

if statement

- if is the typical conditional execution in many programming languages

```
x <- 35
y <- 45
if (x > y) cat("x is bigger\n")
if (y > x) cat("y is bigger\n")
# y is bigger
```

- with else we can build an alternative branch

```
if (x > y) {
   cat("x is bigger\n")
} else {
   cat("y is bigger\n")
}
# y is bigger
```

always us the {} brackets to delimit the body

- and if statements can be nested

```
if (x > y) {
   cat("x is bigger\n")
} else if (y > x) {
   cat("y is bigger\n")
} else {
   cat("x and y are equal\n")
}
# y is bigger
```


switch (expression, list) conditional statement

- switch() can be used for multiple alternatives, instead of nested if / else
- expression can be a number:

```
switch(1, "red", "green", "blue", "yellow")
# [1] "red"

switch(4, "red", "green", "blue", "yellow")
# [1] "yellow"

switch(0, "red", "green", "blue", "yellow")
# NULL
switch(5, "red", "green", "blue", "yellow")
# NULL
```

expression out of range

or a string ➜ the matching named item's value is returned

```
x <- 35
y <- 45
msg <- switch(as.character(sign(x - y)),
              "1" = "x is bigger",
              "0" = "x and y are equal",
              "-1" = "y is bigger")
cat(msg, "\n")
y is bigger

as.character(sign(x - y))
# [1] "-1"
```

named list

for loops

- Basic usage: `for (j in set_or_sequence)`

```
x <- c(1, 2, 7, 9)

for (i in x) cat(i, "␣|␣")
# 1  | 2  | 7  | 9  |
```

- we can use a sequence `1:5`

```
for (i in 1:5) cat(i, "|␣")
1 | 2 | 3 | 4 | 5 |
```

- but also a decreasing sequence

```
for (i in 1:-1) cat(i, "␣|␣")
# 1  | 0  | -1  |
```

- we can loop over a sequence generated along a vector

```
for (i in seq_along(x)) cat("(", i, ",", x[i],")")
# ( 1 , 1 )( 2 , 2 )( 3 , 7 )( 4 , 9 )
```


while loops

- it's another possible construct for loops
- `while` tests a condition and enters the expression only if the condition is `TRUE`

  syntax:

```
while (some_expression_is_true) { do_something }
```

```
i <- 1
while (i<6) {
   cat(i,"␣|␣")
   i <- i + 1
}
# 1  | 2  | 3  | 4  | 5  |
```

do - while loops in R ?

- the do - while construct does not exist in R
- it can be simulated with a repeat statement

- while tests a condition and executes block only if the condition is TRUE : 0 to MANY
- do - while executes at least once, and tests the condition at the end : 1 TO MANY

syntax:

```
repeat {
    statements
    if ( !condition ) { break }
}
```

condition is a logical expression to remain in the block

- Here is and example

```
i <- 1
repeat {
   cat(i," | ")
   i <- i + 1
   if ( !(i<6) ) { break }
}
# 1  | 2  | 3  | 4  | 5  | >
```


Example: compute the factorial (1)

Problem

- given an integer number n, compute the factorial:
- $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$

Using a for loop

```
n <- 5

fac <- 1
for (j in 2:n) {
    fac <- fac * j
}

cat(paste(n, "! =", fac, "\n"))
# 5 ! = 120
```

Example: compute the factorial (2)

Problem

- given an integer number n, compute the factorial:
- $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$

Using a while loop

```
n <- 5

fac <- 1
tmp <- n
while (tmp > 1) {
    fac <- fac * tmp
    tmp <- tmp - 1
}

cat(paste0(n, "! = ", fac, "\n"))
# 5! = 120
```


Example: compute the factorial (3)

Problem

- given an integer number n, compute the factorial:
- $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$

Using a repeat loop

```
n <- 5

fac <- 1
tmp <- n
repeat {
    fac <- fac * tmp
    tmp <- tmp - 1
    if (tmp < 1) {
        break
    }
}

cat(paste0(n, "! = ", fac, "\n"))
# 5! = 120
```

Example: compute the factorial (4)

Problem

- given an integer number n, compute the factorial:
- $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$
- we could compute the factorial using built-in functions that operate on the entire vector, avoiding the need of loops or repeats

```
cumprod(1:5)
# [1]   1   2   6  24 120

cumprod(0:5) # This does not work, '0' is in the vector
# [1] 0 0 0 0 0 0

# We could get the factorial as the maximum value of the new vector:
max(cumprod(1:5))
[1] 120
```

- R implements a built-in `factorial()` function

```
factorial(5)
# [1] 120
factorial(0)
# [1] 1
factorial(-3)
# [1] NaN
# Warning message:
# In gamma(x + 1) : NaNs produced
```


function

- functions are the most important bulding block in R

```
f <- function(x, y, z) { statement(s) }
```

- a function in R can have (a variable number of) arguments
- that may have names
- and default values
- in a call, arguments are given by position or name

```
fsum <- function(x = 1, y = 2) { x + y }

fsum()
# [1] 3

fsum(2,3)
# [1] 5
fsum(2)
# [1] 4
fsum(y=2)
# [1] 3
fsum(y=2, x=4)
# [1] 6

fsum
# function(x = 1, y = 2) { x + y }
```

R lazy evaluation

What is it?

- it's a programming strategy, used mainly in functional languages
- ➜ only necessary symbols are evaluated ➜ only the needed objects will be loaded in memory and/or looked for

Examples

```
plaz <- function(a, b) { return 10; }

plaz()
# [1] 10
plaz(1)
# [1] 10
plaz(1, 2)
# [1] 10
```

```
play <- function(a, b) { a * 10 }

play()
# Error in play() : argument "a" is missing, with no default
play(1)
# [1] 10
play(1, 2)
# [1] 10
```


R lazy evaluation (2)

More examples

- ➜ and expression is evaluated not at the time the call is made, but only when the expression is used

```
# Here, only TRUE is evaluated
if (TRUE || no_variable) {
  12
}
# [1] 12
```

```
ping <- function(a = Sys.time(), b = Sys.time(), c = Sys.time()){
  print(a)
  Sys.sleep(1)
  print(b)
  Sys.sleep(1)
  print(c)
}

ping()
# [1] "2022-03-16 16:44:07 CET"
# [1] "2022-03-16 16:44:08 CET"
# [1] "2022-03-16 16:44:09 CET"
```

ifelse() : vectorized conditional execution

```
y <- log(rpois(20,1.5))
y
#  [1]      -Inf 1.0986123 0.0000000 0.0000000 0.0000000 0.6931472
#  [7] 0.6931472      -Inf 1.3862944 0.6931472 1.3862944      -Inf
# [13]      -Inf 0.0000000 0.0000000 1.0986123 0.0000000 0.0000000
# [19]      -Inf 1.0986123

mean(y)
# [1] -Inf
```

- we need to replace `-Inf` with `NA`

```
(y <- ifelse(y<0, NA, y))
#  [1]        NA 1.0986123 0.0000000 0.0000000 0.0000000 0.6931472
#  [7] 0.6931472        NA 1.3862944 0.6931472 1.3862944        NA
# [13]        NA 0.0000000 0.0000000 1.0986123 0.0000000 0.0000000
# [19]        NA 1.0986123

mean(y, na.rm=TRUE)
# [1] 0.5431911
```

`ifelse()` allows to perform conditional execution on the whole vector

Loops may be slow, if compared to vectorized operations

Problem

- we generate a vector of 10^7 random numbers from a uniform distribution
- we want to serch for the maximum value in the vector, using the R `max()` function and conventional loops

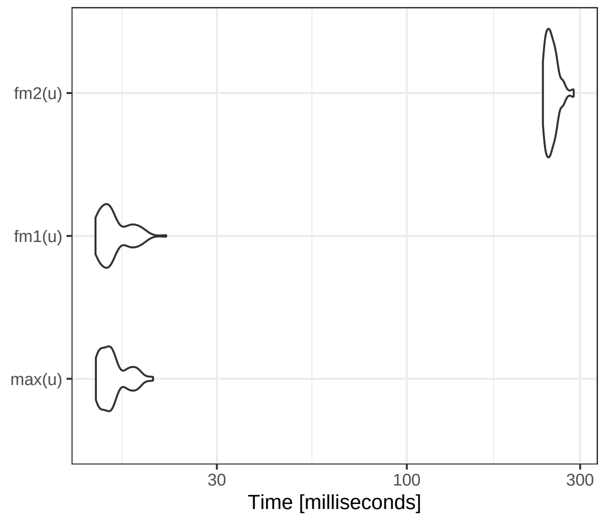
```
library(microbenchmark)

fm1 <- function(x) { max(x) }

fm2 <- function(x) {
  cmax <- x[1]
  for (i in 2:length(x)) {
    if (x[i] > cmax) cmax <- x[i]
  }
}

u <- runif(10^7)

microbenchmark(max(u), fm1(u), fm2(u
```




```
Unit: milliseconds
   expr      min       lq     mean   median      uq      max neval cld
 max(u)  13.8899  14.0255  15.0040  14.3078  15.753  18.1426   100 a
 fm1(u)  13.8918  13.9899  14.8993  14.4838  15.186  21.7245   100 a
 fm2(u) 237.0996 238.9029 243.2218 240.7945 246.097 259.4658   100  b
```

Good/Bad practice in building vectors

Problem

- we want to build a vector containing 10^n elements in the sequence $1{:}10^n$
- three ways are analyzed

```
test1 <- function(n){
    y <- 1:n
}

test2 <- function(n){
    y <- numeric(n)
    for (i in 1:n)
        y[i] <- i
}

test3 <- function(n){
   y <- NULL
   for (i in 1:n)
       y <- c(y,i)
}
```


Good/Bad practice in building vectors (2)

```
test1 <- function(n){
    y <- 1:n
}
```

```
test2 <- function(n){
    y <- numeric(n)
    for (i in 1:n)
        y[i] <- i
}
```

```
test3 <- function(n){
   y <- NULL
   for (i in 1:n)
       y <- c(y,i)
}
```

- the first method (`test1`) is the best
- the loop using a pre-determined vector length is reasonably fast
- the last method (`test3`) is the slowest
- Moral: never grow vectors by repeated concatenation