Introduction to R*

Lecture 3: Control-flow and functions

Wim R.M. Cardoen

Last updated: 10/14/2022 @ 13:35:29

Contents

1 R Control-flow

1.1 Conditional constructs

- **if**, **else if**, **else**

  Syntax:

  ```
  if(condition1){
      ...
  }[else if(condition2){
      ...
  }else{
      ...
  }]
  ```

- **ifelse()**

  The `R` language provides a **vectorized** version: **ifelse()** .

  Syntax:

  ```
  ifelse(condition, vecy, vecn)
  ```

  where:

  – *condition* : test condition
  – *vecy* : values in case of **TRUE** values
  – *vecn* : values in case of **FALSE** values

  There exists a more general version of **ifelse()** i.e. dplyr::case_when()

1.1.1 Examples

```
score <- 75.0

if(score>=90.0){
  grade <- 'A'
} else if((score<90.0) && (score>=80.0)){
  grade <- 'B'
} else if((score<80.0) && (score>=70.0)){
  grade <- 'C'
} else{
  grade <- 'D'
}
cat(sprintf("Score:%4.2f -> Grade:%s\n", score, grade))
```

```
Score:75.00 -> Grade:C
```

```
x <- c(-1,2,1,-5,-7)
c
```

```
function (...)  .Primitive("c")
```

```
res <- ifelse(x>=0, x,-x)
res
```

```
[1] 1 2 1 5 7
```

1.2 Loop constructs

There are several loop constructs:

- **while**

  ```
  while(condition){
    body of the loop
  }
  ```

- **for**

  ```
  for(item in sequence){
    body of the loop
  }
  ```

- **repeat**

  ```
  repeat{
    body of the loop
  }
  ```

  The **repeat** loop has no condition to leave the loop: insert a **break**.

The **break** statement allows one to break out of the **while**, **for** and **repeat** constructs.
The **next** statement allows one to go to the next iteration.

1.2.1 Examples

- for loop construct

```
# Loop over all items
fruit <- c("apple", "pear", "banana", "grape")
for(item in fruit){
  cat(sprintf(" Fruit:%s\n", item))
}
```

```
 Fruit:apple
 Fruit:pear
 Fruit:banana
 Fruit:grape
```

```
# Skip all numbers which are multiples of 3
x <- sample(1:100, size=10, replace=FALSE)
x
```

```
 [1] 23 61  5 83 30 44  1 99 10 87
```

```r
for(item in x){
  if(item%%3==0)
     next
  cat(sprintf(" %3d is NOT a multiple of 3\n", item))
}
```

```
  23 is NOT a multiple of 3
  61 is NOT a multiple of 3
   5 is NOT a multiple of 3
  83 is NOT a multiple of 3
  44 is NOT a multiple of 3
   1 is NOT a multiple of 3
  10 is NOT a multiple of 3
```

- while loop

```r
x <- sample(1:1000, size=100, replace= FALSE)
isFound <- FALSE
i <- 1
while(!isFound){
  if(x[i]%%7==0){
      cat(sprintf(" %3d is divisible by 7\n", x[i]))
      isFound <- TRUE
  }
  else{
      cat(sprintf(" %3d is NOT divisible by 7\n", x[i]))
      i <- i + 1
  }
}
```

```
 784 is divisible by 7
```

- repeat loop

```r
i <- 1
repeat{
  # Stop the loop as soon as you find a multiple of 7.
  if(x[i]%%7==0){
      cat(sprintf(" %3d is divisible by 7\n", x[i]))
      break
  }
  else{
      cat(sprintf(" %3d is NOT divisible by 7\n", x[i]))
      i <- i + 1
  }
}
```

```
 784 is divisible by 7
```

1.3 Exercises

- Write code to find the smallest of three numbers, e.g. 21, 12, 17
- The **Fibonacci** sequence is defined by the following recurrence relation:

  $$F_n = F_{n-1} + F_{n-2}$$

  where $F_0 = F_1 = 1$.
  Calculate all Fibonacci numbers up to F_{15}.
- The square root of a number n is equivalent to solving the following equation:

  $$x^2 - n = 0$$

  The solution to this equation can be found iteratively by using e.g. the Newton-Raphson method.

  Iteration $i + 1$ for x is then given by:

  $$x_{i+1} = \frac{1}{2}(x_i + \frac{n}{x_i})$$

  Find the square root of 751 to a precision of at least 8 decimals. You can set x_0 to n itself.

2 R Functions

2.1 General statements:

The **most common way** to create a function is:

- to assign a function name **and**
- use the **function()** statement

Syntax:

```
function_name <- function(arg_list){
   # body of the function
}
```

A function in R has 3 important components:

- arguments of the function: **formals()**
- body of the function: **body()**
- environment in which the function runs: **environment()**

A function can exit in 2 ways:

- by returning a value
  - implicit return: last expression evaluated in the body
  - explicit return: by invoking the **return()** function
- through error e.g. by invoking the **stop()** function

An R function returns **only** 1 object.
If you want to return more than 1 object, put the objects in an R **list()** and return the list.

2.1.1 Examples

- Implicit return

```
a <- 1.0
b <- 2.0
mysum1 <- function(x,y){
  x+y
}
cat(sprintf(" Sum of %f and %f is %f\n", a, b, mysum1(a,b)))
```

```
 Sum of 1.000000 and 2.000000 is 3.000000
```

- Use of an explicit return statement

```
mysum2 <- function(x,y){
  return(x+y)
}
cat(sprintf(" Sum of %f and %f is %f\n", a, b, mysum2(a,b)))
```

```
 Sum of 1.000000 and 2.000000 is 3.000000
```

- Retrieve the formal arguments of a function

```
formals(mysum2)
```

```
$x


$y
```

- Retrieve the body of a function

```
body(mysum2)
```

```
{
    return(x + y)
}
```

- Retrieve the environment of a function

```
environment(mysum2)
```

```
<environment: R_GlobalEnv>
```

2.2 More on arguments

- default arguments
- varargs

2.3 lazy evaluation of functions

Explain what it meansa

2.4 piping

discuss library(magrittr)

2.5 prefix vs. infix functions

2.6 anonymous functions

If a function does **not** bear a name it is called an anonymous function see http://adv-r.had.co.nz/Functional-programming.html#anonymous-functions

2.7 Exercises

- Write your own factorial function named `myfactorial(n)`.
  The factorial function, $n!$ is defined as:
  $$n! = n\,(n-1)!$$
  where $0! := 1$.
- Write your own function named `castdie(n)` which simulates casting a die n times.
  - Assume you have a fair die.
  - Adjust the function `castdie(n)` for the general case i.e. a non-fair die.
  - Hint: you can use `R`'s **sample()** function.
- An auto-regressive time series of type AR(1) is defined as follows:
  $$x_i = \varphi\, x_{i-1} + \varepsilon_i$$
  where $\varepsilon_i \sim N(0, \sigma^2)$.
  - Write the function `genAR1Series(n=1000, x0=0.0, phi=0.7)`
    which returns the AR(1) time series $\{x_i\}$ for $i \in \{1, \dots, n\}$,
    where:
    $$\begin{aligned} x_0 &= 0.0 \\ \varphi &= 0.7 \\ \varepsilon_i &\sim N(0,1) \ , \forall\, i \in \{1, \dots, n\} \end{aligned}$$
  - Write the sample autocorrelation function (`myacf(x)`) (`ACF`) which calculates a vector of $\rho(h)$'s where the `lag` $h \in \{0, 1, 2, \dots, n-1\}$.

    The autocorrelation with lag h, i.e. $\rho(h)$ is defined as follows:
    $$\rho(h) := \frac{\gamma(h)}{\gamma(0)}$$
    where:
    $$\begin{aligned} \gamma(h) &:= \frac{1}{n} \sum_{i=1}^{n-h} \Big(x_{i+h} - \overline{x}\Big)\Big(x_i - \overline{x}\Big) \\ \overline{x} &:= \frac{1}{n} \sum_{i=1}^{n} x_i \end{aligned}$$
  - Calculate the autocorrelation vector for the time series you generated previously.
    You can check your results with `R`'s **stats::acf()** function.
- In the cyclic group $\mathbb{Z}_4$, we only have the (integer) elements: $\{0, 1, 2, 3\}$.
  The addition ($\forall\, x, y \in \mathbb{Z}_4$) is defined as follows:
  $$x + y \equiv x + y \mod 4$$

```
x <- sample(0:3, size=10, replace=TRUE)
x
```

```
 [1] 2 1 3 2 3 1 1 2 0 3
```

```r
y <- sample(0:3, size=12, replace=TRUE)
y
```

```
 [1] 2 2 3 1 1 0 2 2 2 3 1 1
```

Invoking the infix addition `%+%` results into:

```r
res <- x %+% y
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

```r
res
```

```
 [1] 0 3 2 3 0 1 3 0 2 2 3 2
```

Write the infix function (`%+%`) to perform addition in the cyclic group $\mathbb{Z}_4$.