Introduction to R*

Lecture 3: Control-flow and functions

Wim R.M. Cardoen

Last updated: 10/14/2022 @ 21:54:35

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] 13 86 46 64 89 94 17 52 62  3
```

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

```
  13 is NOT a multiple of 3
  86 is NOT a multiple of 3
  46 is NOT a multiple of 3
  64 is NOT a multiple of 3
  89 is NOT a multiple of 3
  94 is NOT a multiple of 3
  17 is NOT a multiple of 3
  52 is NOT a multiple of 3
  62 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
  }
}
```

```
 618 is NOT divisible by 7
 787 is NOT divisible by 7
 612 is NOT divisible by 7
 169 is NOT divisible by 7
 154 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
  }
```

```
}
```

```
618 is NOT divisible by 7
787 is NOT divisible by 7
612 is NOT divisible by 7
169 is NOT divisible by 7
154 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
}
```

  An **anonymous** function does **not** bear a name. It can be useful for short expressions.

- A function in `R` has 3 important components:
  - arguments of the function: **formals()**
    `R` allows default arguments. The default values are assigned using the **=** sign.
  - body of the function: **body()**
  - environment in which the function runs: **environment()**

  Primitive functions (e.g. **prod()**) call directly `C` code with `.Primitive()` (no `R` code involved).
  These primitive functions can be found in `package:base`.
  If the functions **formals()**, **body()** and **environment()** are applied to primitive functions **NULL** will be returned.

- 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 <- 2.0
b <- 8.0
myprod1 <- function(x,y){
  x*y
}
cat(sprintf(" Prod of %f and %f is %f\n", a, b, myprod1(a,b)))
```

```
 Prod of 2.000000 and 8.000000 is 16.000000
```

- Use of an explicit return statement

```
myprod2 <- function(x,y){
  return(x*y)
}
cat(sprintf(" Prod of %f and %f is %f\n", a, b, myprod2(a,b)))
```

```
 Prod of 2.000000 and 8.000000 is 16.000000
```

- Default argument

```
myshift <- function(x, shift=1){
  return(x+shift)
}
x <- seq(from=2, to=100, length=5)
x
```

```
[1]   2.0  26.5  51.0  75.5 100.0
```

```
y1 <- myshift(x)
y1
```

```
[1]   3.0  27.5  52.0  76.5 101.0
```

```
y3 <- myshift(x,3)
y3
```

```
[1]   5.0  29.5  54.0  78.5 103.0
```

- Anonymous functions (i.e. function without a name)

```
# sapply: apply a function on a vector.
inp <- 1:10
inp
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
out <- sapply(inp, function(x){return(x%%2==0)})
out
```

```
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

Numerical integration of $\int_0^1 \frac{4}{1+x^2}\,dx$

```
res <- integrate(function(x){4.0/(1.0+x^2)}, 0.0, 1.0)
res
```

```
3.141593 with absolute error < 3.5e-14
```

- Retrieve the formal arguments of a function

```
formals(myprod2)
```

```
$x


$y
```

- Retrieve the body of a function

```
body(myprod2)
```

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

- Retrieve the environment of a function

```
environment(myprod2)
```

```
<environment: R_GlobalEnv>
```

2.2 Lazy evaluation of functions

Still to be worked out.

2.3 Prefix vs. infix functions:

- most functions are prefix: the name of the functions **precedes** the arguments.

  ```
  res <- sum(1,2)
  res
  ```

  ```
  [1] 3
  ```

- infix: the function name/operator is found between the arguments

  ```
  res <- 1 + 2
  res
  ```

  ```
  [1] 3
  ```

  In `R` you can create your infix operator by defining a function as follows:

  ```
  '%op%' <- function(x,y){
      # Body of the function
  }
  ```

  The infix function is then invoked as follows :

  ```
  x %op% y
  ```

- `R` contains some **predefined** infix operators:
  - `%%` : modulo operator

– `%/%` : integer division

– `%*%` : matrix multiplication

– `%o%` : outer product

– `%x%` : Kronecker product

– `%in%` : Matching operator

2.3.1 Example

- Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. The angle θ between $\mathbf{x}$ and $\mathbf{y}$ is given by:

$$\theta = \arccos\left(\frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \, \|\mathbf{y}\|}\right)$$

```
'%theta%' <- function(x,y){
    nom <- sum(x*y)
    denom <- sqrt(sum(x^2)) *sqrt(sum(y^2))
    return(acos(nom/denom))
}
```

```
x <- c(1,0,3)
x
```

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

```
y <- c(3,2,-1)
y
```

```
[1]  3  2 -1
```

```
cat(sprintf("The angle (radians) between x and y is:%8.4f\n", x %theta% y))
```

```
The angle (radians) between x and y is:  1.5708
```

2.4 Function composition/piping

Often the output of a function is used as the input for another function.
There are several options beyond the explicit creation of intermediate variables:

- Function composition (e.g. `h(g(f(x)))`).

```
x <- runif(1000)
xav <- mean(x)
res <- sqrt(sum((x-xav)^2))
res
```

```
[1] 8.914154
```

- Use of piping (cfr. the following `Linux` code: `cat *.r | grep function`).
  The magrittr library was developed to provide this feature[1] among others.

[1] The infix function `%>%` is known as the "*and then*" operator.

```
library(magrittr)
(x-xav)^2 %>%
    sum() %>%
    sqrt()
```

```
[1] 8.914154
```

2.5 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 $i \in \{1, \ldots, n\}$, where:
    $$\begin{aligned} x_0 &= 0.0 \\ \varphi &= 0.7 \\ \varepsilon_i &\sim N(0,1)\ \ , \forall\, i \in \{1, \ldots, n\} \end{aligned}$$
  - Write the sample autocorrelation function (`myacf(x)`) (`ACF`) which calculates a vector of $\rho(h)$'s where the `lag` $h \in \{0, 1, 2, \ldots, n-1\}$.

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

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

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

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

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