Lecture 4

Second R Tutorial: For-loops and Functions

Dr. Ido Filin
ifilin@univ.haifa.ac.il

12 November 2012

Outline

R - scripts

18. Create a new script and save it with the name FirstPopModel.r
19. Write the following command lines into the script file:
    1. `Ninitial <- 10`
    2. `lambda <- 2`
    3. `N <- numeric(10)`
    4. `Time <- 0:9`
    5. `N[1] <- Ninitial`
    6. `N[2] <- lambda * N[1]`
    7. `N[3] <- lambda * N[2]`
    8. `N[4] <- lambda * N[3]`
    9. . . . (i.e., repeat the previous style of commands, each time incrementing the indexes)
    10. `N[10] <- lambda * N[9]`
    11. `print(N)`
    12. `plot( Time, N )`
20. Save the new commands that you added to the script file by pressing Ctrl + S

R - scripts

18. Create a new script and save it with the name FirstPopModel.r
19. Write the following command lines into the script file:
    1. `Ninitial <- 10`
    2. `lambda <- 2`
    3. `N <- numeric(10)`
    4. `Time <- 0:9`
    5. `N[1] <- Ninitial`
    6. `N[2] <- lambda * N[1]`
    7. `N[3] <- lambda * N[2]`
    8. `N[4] <- lambda * N[3]`
    9. . . . (i.e., repeat the previous style of commands, each time incrementing the indexes)
    10. `N[10] <- lambda * N[9]`
    11. `print(N)`
    12. `plot( Time, N )`
20. Save the new commands that you added to the script file by pressing Ctrl + S

R - scripts

18. Create a new script and save it with the name FirstPopModel.r
19. Write the following command lines into the script file:
    1. `Ninitial <- 10`
    2. `lambda <- 2`
    3. `N <- numeric(10)`
    4. `Time <- 0:9`
    5. `N[1] <- Ninitial`
    6. ```
       for ( index in 2:10 )
         { N[index] <- lambda * N[index-1] }
       ```
    7. `print(N)`
    8. `plot( Time, N )`
20. Save the new commands that you added to the script file by pressing Ctrl + S
       ```

For-loops in R

- A for-loop has the following structure

for (*<an index variable>* in *<vector of values>*)

{

<list of R commands>

...

}

- Example: sum values and print their square.

```
sumVar <- 0
for ( val in c(0.1, -1, 2.5, pi, 0.53e+3) )
{
   sumVar <- sumVar + val
   print(val^2)
}
```

For-loops in R

- We use for-loops because we are lazy and don't want to write the same operation repeatedly.

- Especially, if we need to repeat a large number of times.

  e.g., calculate population size for 1000 generations.

- Less typing $\longrightarrow$ Less room for mistakes and errors.

For-loops in R

1. Create a new script and save it with the name `SecondPopModel.r`
2. Write the following command lines into the script file:
   1. `genNum <- 32`
   2. `Ninitial <- 1`
   3. `lambda <- 2`
   4. `N <- numeric(genNum)`
   5. `Time <- 24 * ( 0 :  ( genNum - 1 ) )`
   6. `N[1] <- Ninitial`
   7. ```
      for ( index in 2:genNum )
          { N[index] <- lambda * N[index-1] }
      ```
   8. `print(N)`
   9. `plot( Time, N , xlab = "Time[hours]" )`
3. Save and run the script.
      ```

Outline

Functions in R

- Functions provide another way to easily and safely repeat the same set of operations.

- For example: consider the command
  `w <- x^2 + y - z^3`

- Later in our program, we want to do the same calculation again, but with a different set of variables:
  `w <- alpha^2 + beta - gamma^3`

- Or change order among `x`, `y`, `z`:
  `w <- y^2 + z - x^3`

Functions in R

- **Instead** of writing the same formula each time we can define a **function**

```
fooFunc <- function(x,y,z)
    { return( x^2 + y - z^3 ) }
```

- Then we can substitute the previous calculations with the following **function call** commands:
    - `w <- fooFunc( x, y, z )`
    - `w <- fooFunc( alpha, beta, gamma )`
    - `w <- fooFunc( y, z, x )`

- Saves the trouble of typing the same formula repeatedly, and consequently, reduces risk of errors.

- Clearly, functions become even more valuable when the repeated task includes several commands, all of which would have had to be rewritten again and again.

Arguments of a function

- Input(s) $\longrightarrow$ function $\longrightarrow$ Output
- Different input $\longrightarrow$ Different output
- In R, the inputs are provided to the function through a comma-separated list of **arguments**.
- We have already seen several functions:
  `plot(x, y, ...)` takes two (or more) arguments.
  `getwd()` has no arguments.
  `print(...)` takes one argument.
- Other built-in functions:
  `sqrt(x)`, `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, `factorial(x)`, `max(v)`, `min(v)`, `mean(v)`, `var(v)`, `sd(v)`, `setwd(...)`, `coplot(...)`, `boxplot(...)`, `lines(...)`, `lm(...)`, `anova(...)` etc.

Return value of a function

- Input(s) ⟶ function ⟶ Output
- Different input ⟶ Different output
- In R, the output of a function is given by its **return value**.
- The return value can then be used in calculations, assignments etc.
- For example:

```
vectorOfSineVals <- sin( c(0, pi/6, pi/3, pi/2) )
y <- 2 * sqrt(3) + factorial(4)
```

- The return value can be a number, a vector, a text string, or any other data type.
- It is important to read the documentation of a function in order to know what is the return value of the function.

Defining our own functions

1. Create a new script: `ThirdPopModel.r`
2. Write the following command lines into the script file:
    1. `genNum <- 32`
    2. `Ninitial <- 1`
    3. `lambda <- 2`
    4. `popGrowth <- function( popSize, growthParam )`
       `{ newVal = growthParam * popSize; return( newVal ) }`
    5. `N <- numeric(genNum)`
    6. `Time <- 24 * ( 0 :  ( genNum - 1 ) )`
    7. `N[1] <- Ninitial`
    8. `for ( index in 2:genNum )`
       `{ N[index] <- popGrowth( N[index-1], lambda ) }`
    9. `print(N)`
    10. `plot( Time, N , xlab = "Time[hours]" )`
3. Save and run the script.

Defining our own functions

- A function declaration has the following structure

<function name> `<-` `function(`*<list of argument names>*`)`

```
{
    <list of R commands>
    ...
    return(...)
}
```

- Note that this is similar to assignment into variables.
- Example: `funfun <- function( numarg, textarg ) { print(textarg); val <- numarg^2 + numarg; return(val) }`
  Test it by typing
  `print( 1.5 * funfun( 4, "Learning R is fun!" ) )`

Default values of arguments

- We can define default values for arguments.
- We do it using the **=** sign within the function declaration.
- Example:

```
funfun <- function( numarg = 5, textarg = "**
Default text **" )
{ print(textarg); val <- numarg^2 + numarg;
return(val) }
```

- Test it by typing
  1. `funfun( 4, "Learning R is fun!" )`
  2. `funfun( 4 )`
  3. `funfun()`
- If we want to change only the second argument
  1. `funfun( , "R is fun!" )`
  2. `funfun( textarg = "Good morning" )`

Default values of arguments

- Or input them in a different order
  1. `funfun( textarg = "fun fun fun!", numarg = 3)`
- We can use the explicit names of the arguments, as defined in the function declaration, when setting values of arguments during a function call.
- In that case, we don't need to observe the original order of the arguments.
- Example: `foo <- function(x, y, z) {...}`
  The function call `foo( z = 3, x = 1, y = 2 )`
  is identical to the call `foo( 1, 2, 3 )`
  but different than `foo( 3, 1, 2 )`.
- We have already seen this syntax with the `plot` function.

```
plot( x, y, xlab = ..., type = ..., ylab = ...)
```

Default values of arguments

1. Change `ThirdPopModel.r` as follows.
   1. `genNum <- 32`
   2. `Ninitial <- 1`
   3. `lambda <- 3`
   4. `popGrowth <- function(popSize = 1, growthParam = 2)`
      `{ newVal = growthParam * popSize; return( newVal ) }`
   5. `N <- numeric(genNum)`
   6. `Time <- 24 * ( 0 :  ( genNum - 1 ) )`
   7. `N[1] <- Ninitial`
   8. `for ( index in 2:genNum )`
      `{ N[index] <- popGrowth( popSize = N[index-1] ) }`
   9. `print(N)`
   10. `plot( Time, N , xlab = "Time[hours]" )`
2. Save and run the script.
3. What is the finite rate of increase of this population?

Outline