Lab 02 Notes

Fernando Miguez

2019-09-06

Lab 2: Basic Programming

Topics

- writing scripts
- sourcing
- if else

Writing scripts

When typing many commands at the R command line it quickly becomes cumbersome to keep track of computations and logic of the code. Thus, it is more efficient to write the code in a separate file which we will call script. The extension normally used for R extensions is `.r`. Since it is possible that previous objects were created in an R session, we can first choose to clean the workspace using the `rm(list=ls())` command. A simple script can be as follows

```
## Converting from Fahrenheit (F) to Celsius (C)
## First eliminate existing objects in the workspace
rm(list=ls())

## Use some input values
x1 <- 40
x2 <- 50
x3 <- 70
x4 <- 80

## Convert these values to Celsius
## The formula is C = (F - 32) * 5/9

x1.C <- (x1 - 32) * 5/9
x2.C <- (x2 - 32) * 5/9
x3.C <- (x3 - 32) * 5/9
x4.C <- (x4 - 32) * 5/9

## Original values
print(c(x1, x2, x3, x4))
```

```
## [1] 40 50 70 80
```

```
## Celsius values
print(c(x1.C, x2.C, x3.C, x4.C))
```

```
## [1]  4.444444 10.000000 21.111111 26.666667
```

If we save this file as `far-cel.r` we can execute it at the R command line

```
source("far-cel.r")
## [1] 40 50 70 80
## [1]  4.44 10.00 21.11 26.67
```

This assumes that the script is in the same directory (or folder) as the current working directory (`getwd()`). At the end of lab you will have an opportunity to improve this script.

Control Flow

Let us say that we are given some temperature data and we want to convert from F to C but we are not sure whether the input is in F or not. For problems like this we can sometimes use programming statements such as `if` and `else`. For example

```
## Convert x from F to C if input is in F

x <- 50

units <- "F"

if(units == "F"){
  x <- (x - 32) * 5/9
}
```

At this point the previous code might not make a lot of sense, but it serves to illustrate the if statement and it shows how we started with one object (`x`) which contained temperature in the F scale and then replaced that value with temperature in the C scale. Notice that although many times we use an `if` together with an `else` the else part is optional. Also the basic construct of an `if` statement is as follows

```
if(logical_expression){
   expression_1
   expression_2
   ...
}else{
   expression_n
   expression_n+1
   ...
}
```

The brackets `{}` are used to combine several statements with the `if, else` clause.

In many programs it is necessary to create a loop. This can be accomplished with the `for` command

```
for(x in vector){
   expression_1
   expression_2
   ...
}
```

As an example we can compute the cumulative sum of the first 10 integers

```
sum_x <- 0
for(i in 1:10){
   sum_x <- sum_x + i
   cat("the current loop element is ",i,"\n")
   cat("the current cumulative sum is ",sum_x,"\n")
}
```

```
## the current loop element is  1
## the current cumulative sum is  1
## the current loop element is  2
## the current cumulative sum is  3
## the current loop element is  3
## the current cumulative sum is  6
## the current loop element is  4
## the current cumulative sum is  10
## the current loop element is  5
## the current cumulative sum is  15
## the current loop element is  6
## the current cumulative sum is  21
## the current loop element is  7
## the current cumulative sum is  28
## the current loop element is  8
## the current cumulative sum is  36
## the current loop element is  9
## the current cumulative sum is  45
## the current loop element is  10
## the current cumulative sum is  55
```

The `for` command is not the only way to create a loop. There is also the `while` command

```
while(logical_expression){
     expression_1
     expression_2
     ...
}
```

The statements inside the brackets will be executed until the condition in the while statement becomes false. This should be done with caution as the condition is never falsified we will end up creating an infinite loop.

As an example we can look at the Fibonacci series. In botany it appears, for example, in the disposition of flowers in a head of sunflower. The Fibonacci sequence is defined as $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. We can find the first `x` Fibonacci numbers

```
## Calculate the first x Fibonacci numbers

## Clear the workspace
rm(list=ls())

## Set x equal to 100
x <- 100

## Initialize variables
F <- c(1, 1)
n <- length(F)

while(F[n] <= 100){
      cat("n = ",n," F[n] = ",F[n], "\n")
      n <- n + 1
      F[n] <- F[n - 1] + F[n - 2]
}
```

```
## n =  2  F[n] =  1
```

```
## n =  3  F[n] =  2
## n =  4  F[n] =  3
## n =  5  F[n] =  5
## n =  6  F[n] =  8
## n =  7  F[n] =  13
## n =  8  F[n] =  21
## n =  9  F[n] =  34
## n =  10  F[n] =  55
## n =  11  F[n] =  89
```

Vector-based programming

In R it is generally more efficient to use what is called vectorized operations. These tend to be shorter and more computationally efficient.

For example, we can find the sum of the first 100 squares using a loop

```
n <- 100
S <- 0
for(i in 1:n){
     S <- S + i^2
}
S
```

```
## [1] 338350
```

Using vector operations this is simply

```
S <- sum((1:n)^2)
```

This one line needs to be broken down to understand what is going on. First, a sequence from 1 to 100 was created `1:n`, then the squaring is applied to each element (i.e. it is vectorized). Finally, all the 100 elements squared are added up and stored in an object `S`. The result is the same.

Program flow

It is an useful exercise to study how the variables change through a program and how the different steps are executed. From Jones 2009 we have a nice example of flow control, Figure 1.

Programming with functions

This is similar to chapter 5 of Jones 2009.

"Functions are one of the main building blocks of large programs and they are an essential tool for structuring complex algorithms. In some other languages *procedures* and *subroutines* play the same role as functions in R."

Functions

A function has the form

```
   # program: spuRs/resources/scripts/threexplus1.r
 1 x <- 3
 2 for (i in 1:3) {
 3   show(x)
 4   if (x %% 2 == 0) {
 5     x <- x/2
 6   } else {
 7     x <- 3*x + 1
 8   }
 9 }
10 show(x)
```

Charting the flow through this program, we get the output presented in Table 3.1.

Table 3.1 *Charting the flow for program* `threexplus1.r`

line	x	i	comments
1	3		i not defined yet
2	3	1	i is set to 1
3	3	1	3 written to screen
4	3	1	`(x %% 2 == 0)` is `FALSE` so go to line 7
7	10	1	x is set to 10
8	10	1	end of else part
9	10	1	end of for loop, not finished so back to line 2
2	10	2	i is set to 2
3	10	2	10 written to screen
4	10	2	`(x %% 2 == 0)` is `TRUE` so go to line 5
5	5	2	x is set to 5
6	5	2	end of if part, go to line 9
9	5	2	end of for loop, not finished so back to line 2
2	5	3	i is set to 3
3	5	3	5 written to screen
4	5	3	`(x %% 2 == 0)` is `FALSE` so go to line 7
7	16	3	x is set to 16
8	16	3	end of else part
9	16	3	end of for loop, finished so continue to line 10
10	16	3	16 written to screen

This is exactly what the computer does when it executes a program: it keeps track of its current position in the program and maintains a list of variables and their values. *Whatever line you are currently at, if you know all the variables then you always know which line to go to next.*

Figure 1: Scanned page from Jones 2009 illustrating program flow.

```
fname <- function(argument_1, argument_2, ...){
                 expression_1
                 expression_2
                 ...
                 return(output)
}
```

The function is run by simply

```
fname(x1, x2, ...)
```

We say here that `x1` and `x2` are passed to the arguments of the function. Let us say we write a very simple function to add to values.

```
## Creating functions
add2values <- function(x, y){

  result <- x + y

  return(result)

}
```

This is, of course, a trivial function but it illustrates the basic building blocks of writing a function. The return statement is optional, so the same can be achieved with the following

```
## Creating functions
add2values <- function(x, y){

  result <- x + y

  result

}
```

Going back to the notes on 'introduction to modeling'. We can write this polynomial as an R function.

```
poly3 <- function(x, beta0, beta1, beta2, beta3){
  y <- beta0 + x * beta1 + x^2 * beta2 + x^3 * beta3
  y
}
```

Similarly, we can turn to the logistic equation and build an R function (without the error term). It is important to notice that we need to run these function so that they appear in the R workspace.

```
## A function for a logisitc model
logistic <- function(x, Asym, xmid, scal){

  y <- Asym / (1 + exp((xmid - x)/scal))
  y

}
```

We can run the `ploy3` equation with some arguments

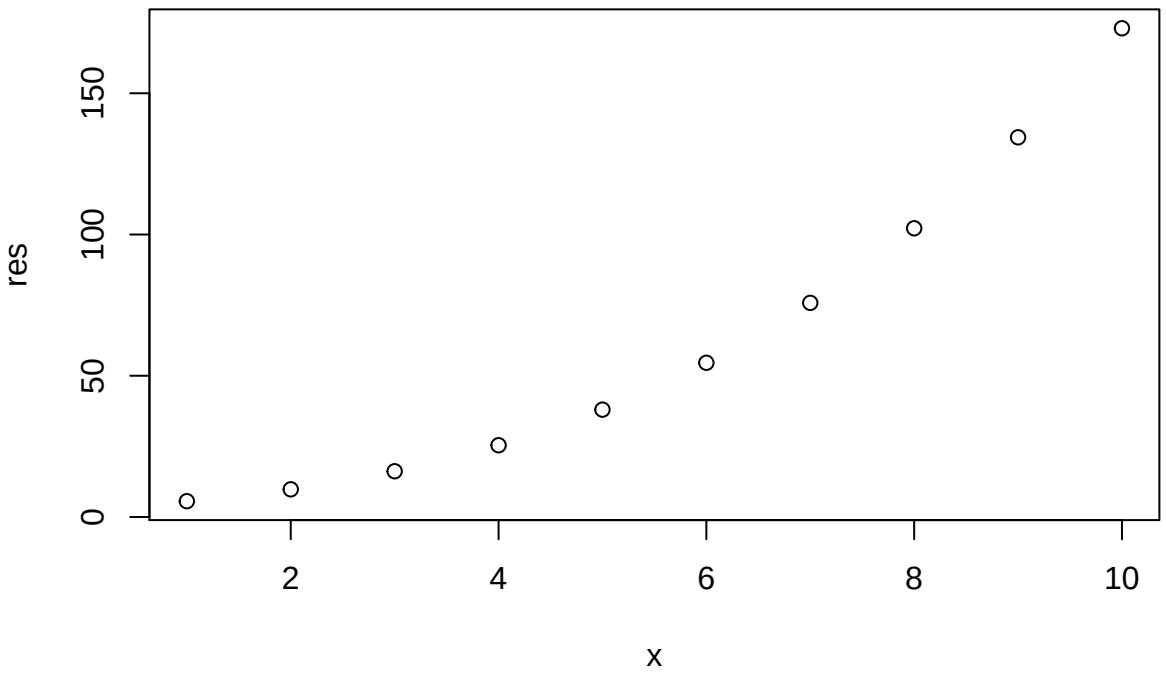
```
x <- 1:10
res <- poly3(x, 3, 2, 0.5)
## Error in x^3 * beta3 : 'beta3' is missing
```

Notice that we only provided values for `beta0`, `beta1`, `beta2`, but not for `beta3`. R prints an error saying that there is no value assigned to `beta3` and thus cannot complete the operation.

```
## Second try
x <- 1:10
res <- poly3(x, 3, 2, 0.5, 0.1)
```

To visualize the results we can use the `plot` command.

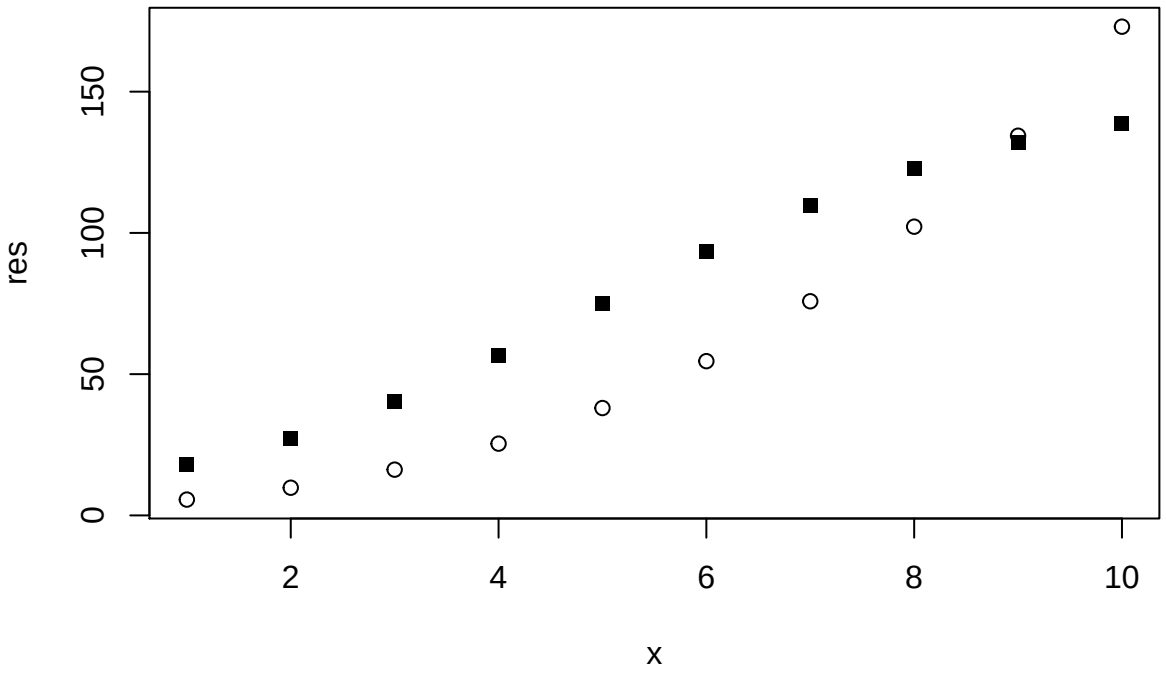
```
## Visualizing results
plot(x, res)
```



We can run the logistic function and display the results together

```
## Running the logisitc function
res2 <- logistic(x, 150, 5, 2)

## Adding the result to the previous plot
plot(x, res)
points(x, res2, pch = 15)
```



Exercises

1. Improve the script `far-cel.r` by creating a function.
2. Write a cumulative sum function for an arbitrary vector using the `for` command. Note that the `cumsum` function does this.
3. Write a function that calculates growing degree days.

References

Introduction to Scientific Programming and Simulation Using R (Chapman & Hall/CRC The R Series) 2nd Edition. by Owen Jones, Robert Maillardet, Andrew Robinson