3b - Functions, Functions & Functions

Noud van Giersbergen

University of Amsterdam

2021-04-15

Topics

- PC test 1
- Advanced Functions
- Example: Euler Problem 21
- Kahoot Quiz

PC test 1

- Level of exercises of PC labs is sufficient
- Only R Studio is allowed
  - Closed book
  - No Google
  - No slides
  - No other files
  - Only help function in R Studio
- Only upload one file with `.R` extension
  - No conversion to Word
  - No scanning, no conversion to pdf
  - Be sure to set the working directory to a convenient folder, so you can find the file fast
  - You can set the default working directory in `Tools->Global Options...->Default working directory (when not in a project)`

PC test 1

- Partial credits are given
- 13:00-14:30 ***Amsterdam*** time
- Time: 1.5 hours (+15 min. upload)
- Students with extra time: 1.75 hours (15 additional min.)
- Please, take the ***Practice Exam Proctorio (2020/2021)*** using RStudio and see if you can upload an R file!
- PC test is in Ans Delft, not Canvas!

Practice, Practice & Practice

- Basic Exercises: https://www.r-exercises.com/start-here-to-learn-r/
    - Vectors and sequences: https://www.r-exercises.com/tag/vectors-and-sequences/
    - Arrays and Matrices: https://www.r-exercises.com/tag/arrays-and-matrices/
    - Loops and conditional execution: https://www.r-exercises.com/tag/loops-and-conditional-execution/
    - Functions: https://www.r-exercises.com/tag/functions/
    - Lists and dataframes: https://www.r-exercises.com/tag/lists-and-dataframes/
    - Character strings: https://www.r-exercises.com/tag/character-strings/
- More advanced: https://projecteuler.net/
    - solutions first 25 problems: https://rstudio-pubs-static.s3.amazonaws.com/236332_7a7d3b552016415d9fda1f0fafc31ff9.html
- `Learning_how_to_code_v3.pdf`: https://canvas.uva.nl/courses/21798/files/4495347/download

11.1.1 Function Creation

- *functionname* is placeholder can be any valid R object name
- The number of arguments and whether to include an ellipsis all depend on the particular function
- Simply use `()` for functions without arguments
- The *body code* (also called *function body*) is executed when the function is called
- *arg1,arg2,arg3,* are treated as objects in the *local environment*

> *functionname* <- `function(`*arg1,arg2,arg3,*`...)`{
>   *body code*
>   `return`(*returnobject*)
> }

```
R> mysquare <- function(x){
+   return(x^2)
+ }
R> ls()
```

```
[1] "mysquare"
```

11.1.1 Function Creation: Fibonacci Example

- Note: no values are printed in the function itself!
    - But the returnobject is still displayed

```
R> myfib3 <- function(thresh){
+   fibseq <- c(1,1)
+   counter <- 2
+   repeat{
+     fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
+     counter <- counter+1
+     if(fibseq[counter]>thresh){
+       break
+     }
+   }
+   return(fibseq)
+ }
R> myfib3(1e3)
```

```
 [1]    1    1    2    3    5    8   13   21   34   55   89
[12]  144  233  377  610  987 1597
```

11.1.2 Using `return`

- Default: the last assignment is returned
- Use `return()` to explicitly return a value
  - Make the code more readable

```
dummy1 <- function(){ a <- 1; b <- 2}
dummy1()
c <- dummy1()
cat(c)
```

```
2
```

```
R> dummy3 <- function(){ a <- 1; b <- 2;
+                       return(a)}
R> dummy3()
```

```
[1] 1
```

11.2.2 Settings Defaults

- Default argument values are sensible when arguments have *natural values*

```
R> f <- function(a, b= 10){
+   return(a+b)
+ }
R>
R> f(10)
```

```
[1] 20
```

```
R> f(10,5)
```

```
[1] 15
```

```
R> f(b=5,a=10)
```

```
[1] 15
```

11.2.4 Dealing with Ellipses

- The ellipsis (`...`) allows you to pass extra arguments without having to define them

```
R> mysum <- function(x,y,...) {
+   args=list(...)
+   som <- x+y
+   if(length(args)>0)
+      for(z in args){
+        som <- som+z
+      }
+   return(som)
+ }
R> mysum(1,2)
```

```
[1] 3
```

```
R> mysum(1,2,3,4)
```

```
[1] 10
```

```
R> mysum(1,2,3:4)
```

11.3.1 Helper Function - Internally Defined

- A *helper* function is another functions written to facilitate the computations carried out by a function

```
R> roots <- function(a,b,c){
+   discriminant<-function(a,b,c){
+     b^2-4*a*c
+   }
+   D=discriminant(a,b,c)
+   if(D > 0){ # first case D>0
+     x_1 = (-b+sqrt(D))/(2*a)
+     x_2 = (-b-sqrt(D))/(2*a)
+     return(c(x_1,x_2))
+   }
+   else if(delta(a,b,c) == 0){ # second case D=0
+     return(-b/(2*a))
+   }
+   else {return("There are no real roots.")} # third case D<0
+ }
R>
R> roots(1,-6,8)
```

```
[1] 4 2
```

11.3.2 Anonymous Functions

- Anonymous (or disposable) functions allow you to define a function intended for single use
  - no new object is created in your global environment

```r
g <- function(x){
  return(x^2)
}

integrate(g,0,2)
```

```
2.666667 with absolute error < 3e-14
```

```r
integrate(function(x){x^2},0,2)
```

```
2.666667 with absolute error < 3e-14
```

11.3.2 Anonymous Functions

- What if the function you want to integrate becomes more complex

```
f <- function(x,mu,sigma){
  z <- (x - mu)/sigma
  return( 1/sqrt(2*pi*sigma)*exp(-1/2*z^2) )
}
```

```
integrate(f,-Inf,0)
```

```
Error in f(x, ...): argument "mu" is missing, with no default
```

```
integrate(function(x) f(x,0,1),-Inf,0)
```

```
0.5 with absolute error < 4.7e-05
```

11.3.2 Anonymous Functions - Example `apply()`

```
R> df <- data.frame(first=5:7, second=(0:2)^2, third=-1:1)
R> df
```

```
  first second third
1     5      0    -1
2     6      1     0
3     7      4     1
```

```
R> apply(df, 2, function(x) { sqrt(sum(x^2)) })
```

```
    first    second     third
10.488088  4.123106  1.414214
```

11.3.2 Anonymous Functions - 2nd example `apply()`

```
R> set.seed(1)
R> x <- sample(1:6, 12, replace=TRUE)
R> mat <- matrix(x, nrow=3)
R> mat
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    6    3
[2,]    4    5    2    1
[3,]    1    3    3    5
```

```
R> apply(mat, 1, function(x) { seq(min(x), max(x)) })
```

```
[[1]]
[1] 1 2 3 4 5 6

[[2]]
[1] 1 2 3 4 5

[[3]]
[1] 1 2 3 4 5
```

11.3.3 Recursive Functions

- *Recursive function*: function that calls itself

```
R> facn <- function(n) {
+   if(n == 1) {
+     return(1)
+   } else {
+     return(n * facn(n-1))
+   }
+ }
R> facn(3)
```

```
[1] 6
```

Remarks:

- vectorized code usually works faster (why?)
- there must always be a *base case* to end the recursion to prevent an infinite recursion

11.3.3 Recursive Functions - Simple Example

```
R> facn <- function(n) {
+   if(n == 1) {
+     return(1)
+   } else {
+     return(n * facn(n-1))
+   }
+ }
R> facn(3)
```

```
[1] 6
```

Returning More Than One Argument

- If you want to return more than one argument, you can use a *list*

```
R> descr.stats <- function(x){
+   return(list(xbar=mean(x),std=sd(x)))
+ }
R>
R> data <- rnorm(100)
R> stats <- descr.stats(data)
R> stats
```

```
$xbar
[1] 0.1107369

$std
[1] 0.9014408
```

```
R> print(coefficient.of.variation <- stats$std/stats$xbar)
```

```
[1] 8.140381
```

Amicable numbers

Let $d(n)$ be defined as the sum of ***proper*** divisors of n (numbers less than n which divide evenly into n). If $d(a) = b$ and $d(b) = a$, where $a \neq b$, then a and b are an amicable pair and each of a and b are called amicable numbers.

For example, the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110; therefore $d(220) = 284$. The proper divisors of 284 are 1, 2, 4, 71 and 142; so $d(284) = 220$.

Evaluate the sum of all the amicable numbers under 10000.

Amicable numbers - Analysis

For example, the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110; therefore $d(220) = 284$. The proper divisors of 284 are 1, 2, 4, 71 and 142; so $d(284) = 220$.

- Note if a is prime, then $d(a) = 1$
- Let (a, b) be an amicable pair. Since $b \neq a$ we can suppose $a < b$. Therefore, to find (a, b), we scan each number a from 2 to $N - 1$ and verify the following conditions
    - $b = d(a)$
    - $a < b < N$
    - $d(b) = a$
- To eliminate repeated numbers, we should set a mark somehow