PP4RS | R Module

Slot 2

Dora Simon

06.09.2018

Outline of the R-Module

Slot 1: Intro & Data Types

Slot 2: Conditionals and Functions & Loops

Slot 3: Read in Data

Slot 4: Data Manipulation

Slot 5: Regressions

Slot 6: Graphs

Slot 7: knitR

Now: Conditionals and Functions & Loops

Conditionals and Functions

If-Else

An if-else structure is useful if you want to execute different code blocks depending on whether a certain statement is evaluated as `TRUE` or `FALSE`:

```
recession = TRUE

if (recession){
  print("Booh!")
} else {
  print("Yay!")
}
```

```
[1] "Booh!"
```

Syntax:

- statements are in round brackets
- two curly brackets around the code block

"If whathever is in round brackets is correct... ...do whatever is written in the curly brackets"

Switch Function

Instead of if-else, one can also use the `switch` function.

```
x <- 1
y <- 2
operation <- 'plus'
```

Switch Function

Instead of if-else, one can also use the `switch` function.

```r
x <- 1
y <- 2
operation <- 'plus'
```

```r
switch(operation,
       plus = x + y,
       minus = x - y,
       times = x * y,
       divide = x / y,
       stop('You specified an unknown operation!')
       )
```

```
[1] 3
```

Comparisons

Sometimes we want to check if two values are equal. We can do this with ==.

```
A <- c(1, 2, 3, 4)
B <- c(1, 3, 4, 5)
```

Comparisons

Sometimes we want to check if two values are equal. We can do this with ==.

```
A <- c(1, 2, 3, 4)
B <- c(1, 3, 4, 5)
```

```
if (A == B) {
  print('They are equal!')
} else {
  print('They are not equal!')
}
```

```
Warning in if (A == B) {: the condition has length > 1 and only the first
element will be used
```

```
[1] "They are equal!"
```

But: If more than one value is returned, R will check only the first truth-value!

Comparisons

Therefore, you can use the `identical` function:

```
A <- c(1, 2, 3, 4)
B <- c(1, 3, 4, 5)

if (identical(A, B)) {
  print('They are equal!')
} else {
  print('They are not equal!')
}
```

```
[1] "They are not equal!"
```

Comparisons

Therefore, you can use the `identical` function:

```
A <- c(1, 2, 3, 4)
B <- c(1, 3, 4, 5)

if (identical(A, B)) {
  print('They are equal!')
} else {
  print('They are not equal!')
}
```

```
[1] "They are not equal!"
```

```
#Problem: You have to have the same variable type.

identical(0L, 0)
```

```
[1] FALSE
```

Comparisons

Other comparison signs:

- `!=` not identical
- `<` smaller than
- `<=` smaller than or equal
- `>` bigger than
- `>=` bigger than or equal
- `!` not
- `&&` logical 'and'
- `||` logical 'or'
- `is.logical`, etc.

Comparisons

A final note for the use of doubles:

Comparisons

A final note for the use of doubles:

```r
1 - 1/3 - 1/3 - 1/3 == 0
```

```
[1] FALSE
```

Comparisons

A final note for the use of doubles:

```r
1 - 1/3 - 1/3 - 1/3 == 0
```

```
[1] FALSE
```

```r
1 - 1/3 - 1/3 - 1/3
```

```
[1] 1.110223e-16
```

Comparisons

A final note for the use of doubles:

```r
1 - 1/3 - 1/3 - 1/3 == 0
```

```
[1] FALSE
```

```r
1 - 1/3 - 1/3 - 1/3
```

```
[1] 1.110223e-16
```

```r
dplyr::near(1 - 1/3 - 1/3 - 1/3, 0)
```

```
[1] TRUE
```

Functions

We need three things to define a function:

1. a function name, here `calc_percent_missing`
2. function arguments, here `x`,
3. the function body enclosed by `{}`

Functions

We need three things to define a function:

1. a function name, here `calc_percent_missing`
2. function arguments, here `x`,
3. the function body enclosed by `{}`

```
calc_percent_missing <- function(x){
  mean(is.na(x))
  }
```

Functions

We need three things to define a function:

1. a function name, here `calc_percent_missing`
2. function arguments, here `x`,
3. the function body enclosed by `{}`

```
calc_percent_missing <- function(x){
  mean(is.na(x))
  }
```

Now let's use the function:

```
calc_percent_missing(c(1, 2, 6, 3, 7, NA, 9, NA, NA, 1))
```

```
[1] 0.3
```

Functions

Function arguments (whatever is in the round brackets) can usually be broadly divided into two categories:

- data: either a dataframe or a vector, our `x`
- details: parameters which govern the computation, usually with defaults

Functions

Function arguments (whatever is in the round brackets) can usually be broadly divided into two categories:

- data: either a dataframe or a vector, our `x`
- details: parameters which govern the computation, usually with defaults

```
cobb_douglas <- function(x, a = 0.5, b = 0.5){
  u <- x[1]^a * x[2]^b
  }
```

Functions

Function arguments (whatever is in the round brackets) can usually be broadly divided into two categories:

- data: either a dataframe or a vector, our `x`
- details: parameters which govern the computation, usually with defaults

```
cobb_douglas <- function(x, a = 0.5, b = 0.5){
  u <- x[1]^a * x[2]^b
  }
```

Let's use it:

```
x <- c(1, 2)
print(cobb_douglas(x))
```

```
[1] 1.414214
```

Functions

Now let's use different parameters:

```
print(cobb_douglas(x, b=0.4, a=0.6))
```

```
[1] 1.319508
```

```
print(cobb_douglas(x, 0.4, 0.6))
```

```
[1] 1.515717
```

What is the difference?

Exercises

- Write a conditional with the following properties (hint: use `if`, `else if` and `else`):
    - If the vector `color` has the value "red", print "It is a tomatoe!"
    - If the vector `color` has the value "yellow", print "It is a yellow pepper!"
    - If the vector `color` has the value "violet", print "It is an onion!"
    - If none of the above is true, print "No idea what this is!"
- Rewrite your color example with the `switch` function
- Implement a fizzbuzz function (hint: you can use `%%` for the remainder of a division)
    - It takes a single number as input.
    - If the number is divisible by three, it returns “fizz”.
    - If it’s divisible by five it returns “buzz”.
    - If it’s divisible by three and five, it returns “fizzbuzz”.
    - Otherwise, it returns the number.

Solution

- Write a conditional with the following properties (hint: use `if`, `else if` and `else`):
  - If the vector `color` has the value "red", print "It is a tomatoe!"
  - If the vector `color` has the value "yellow", print "It is a yellow pepper!"
  - If the vector `color` has the value "violet", print "It is an onion!"
  - If none of the above is true, print "No idea what this is!"

Solution

- Write a conditional with the following properties (hint: use `if`, `else if` and `else`):
  - If the vector `color` has the value "red", print "It is a tomatoe!"
  - If the vector `color` has the value "yellow", print "It is a yellow pepper!"
  - If the vector `color` has the value "violet", print "It is an onion!"
  - If none of the above is true, print "No idea what this is!"

```r
color = 'violet'
if (color == 'red') {
  print('It is a tomatoe!')
} else if (color == 'yellow') {
  print('It is a yellow pepper!')
} else if (color == 'violet') {
  print('It is an onion!')
} else {
  print('No idea what this is!')
}
```

```
[1] "It is an onion!"
```

Solution

- Rewrite your color example with the `switch` function

Solution

- Rewrite your color example with the `switch` function

```
color = 'violet'
switch(color,
       red = print('It is a tomatoe!'),
       yellow = print('It is a yellow pepper!'),
       violet = print('It is an onion!'),
       stop('You specified an unknown color!'))
```

```
[1] "It is an onion!"
```

Solution

- Implement a fizzbuzz function (hint: you can use `%%` for the remainder of a division)
  - It takes a single number as input.
  - If the number is divisible by three, it returns “fizz”.
  - If it’s divisible by five it returns “buzz”.
  - If it’s divisible by three and five, it returns “fizzbuzz”.
  - Otherwise, it returns the number.

Solution

- Implement a fizzbuzz function (hint: you can use `%%` for the remainder of a division)
  - It takes a single number as input.
  - If the number is divisible by three, it returns “fizz”.
  - If it’s divisible by five it returns “buzz”.
  - If it’s divisible by three and five, it returns “fizzbuzz”.
  - Otherwise, it returns the number.

```
fizzbuzz= function(x) {

if (x%%3==0 && x%%5!=0) {
  print('fizz')
} else if (x%%3!=0 && x%%5==0) {
  print('buzz')
} else if (x%%3==0 && x%%5==0) {
  print('fizzbuzz')
} else {
  x
}}
```

Solution

Let's test it

```
fizzbuzz(9)
```

```
[1] "fizz"
```

```
fizzbuzz(10)
```

```
[1] "buzz"
```

```
fizzbuzz(15)
```

```
[1] "fizzbuzz"
```

```
fizzbuzz(7)
```

```
[1] 7
```

Loops

Types of loops

Loops are useful when you need to do the same operation repeatedly.

- `while`: while the statement is true, …
- `for`: for some elements of a vector, …
- `repeat`: repeat until a condition is fulfilled

The syntax is similar to the conditionals, you need the normal and the curly brackets.

Types of loops

Loops are useful when you need to do the same operation repeatedly.

- `while`: while the statement is true, …
- `for`: for some elements of a vector, …
- `repeat`: repeat until a condition is fulfilled

The syntax is similar to the conditionals, you need the normal and the curly brackets.

Tips and Tricks

- Try to put as little code as possible within the loop
- Take out as many instructions as possible
- If you need a nested for loop it is probably a sign that things are not implemented the best way

Vectorization[1]

Consider two math problems.

Vectorized:

$$\begin{bmatrix}1\\2\\3\end{bmatrix}+\begin{bmatrix}1\\2\\3\end{bmatrix}=\begin{bmatrix}2\\4\\6\end{bmatrix}$$

Not Vectorized:

$$\begin{aligned}1+1&=2\\2+2&=4\\3+3&=6\end{aligned}$$

In R, the vectorized version is faster than the not vectorized one.

[1] This and the following slides draw heavily from
http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html

R vs. C

R is a high-level, interpreted computer language

- `i<-5.0`

R vs. C

R is a high-level, interpreted computer language

- `i<-5.0`

C is a lower-level language

- `int i`
- `i=5`

R vs. C

R is a high-level, interpreted computer language

- `i<-5.0`

C is a lower-level language

- `int i`
- `i=5`

C is a compiled program:

- code is written
- translated into binary computer language
- code is run

R vs. C

R is a high-level, interpreted computer language

- `i<-5.0`

C is a lower-level language

- `int i`
- `i=5`

C is a compiled program:

- code is written
- translated into binary computer language
- code is run

R is not!

- code is written
- code is run
- no translation needed

But: Many functions in R are written in C, C++ or FORTRAN!

Why to vectorize

Imagine you use a function which was written in a compiled language

Why to vectorize

Imagine you use a function which was written in a compiled language

- R interprets the input of the function (e.g. data types)

Why to vectorize

Imagine you use a function which was written in a compiled language

- R interprets the input of the function (e.g. data types)
- R passes the information on to the compiled code, e.g. in C

Why to vectorize

Imagine you use a function which was written in a compiled language

- R interprets the input of the function (e.g. data types)
- R passes the information on to the compiled code, e.g. in C
- that code runs faster than pure R

Why to vectorize

Imagine you use a function which was written in a compiled language

- R interprets the input of the function (e.g. data types)
- R passes the information on to the compiled code, e.g. in C
- that code runs faster than pure R

If you use a function element by element, R has to interpret the inputs every time!

Why to vectorize

Imagine you use a function which was written in a compiled language

- R interprets the input of the function (e.g. data types)
- R passes the information on to the compiled code, e.g. in C
- that code runs faster than pure R

If you use a function element by element, R has to interpret the inputs every time!

Therefore, it is faster to use a function on a vector.

Why not to loop

Memory allocation is slow.

```r
j <- 1
for (i in 1:10) {
    j[i] <- 10
}
```

Why not to loop

Memory allocation is slow.

```
j <- 1
for (i in 1:10) {
    j[i] <- 10
}
```

This is better:

```
j <- rep(NA, 10)
for (i in 1:10) {
    j[i] = 10
}
```

There are functions in R, which automatically make sure that you do a "proper" loop. These are from the `apply` and `plyr` packages.

To sum up

Try to avoid loops.

- Functions which are written in C will be faster, especially when used for the whole vector
- Functions which are written in R might still include loops, but they are written in a "clean" way
  - They preallocate memory
  - They get rid of the "side effect", the `i` from your loop automatically

To sum up

Try to avoid loops.

- Functions which are written in C will be faster, especially when used for the whole vector
- Functions which are written in R might still include loops, but they are written in a "clean" way
  - They preallocate memory
  - They get rid of the "side effect", the `i` from your loop automatically

When is it ok to loop?

- If a function does not take vector arguments
- If the iteration is dependent on the results of previous iterations

To sum up

Try to avoid loops.

- Functions which are written in C will be faster, especially when used for the whole vector
- Functions which are written in R might still include loops, but they are written in a "clean" way
  - They preallocate memory
  - They get rid of the "side effect", the `i` from your loop automatically

When is it ok to loop?

- If a function does not take vector arguments
- If the iteration is dependent on the results of previous iterations

Functions that are essentially loops in C

- `cumsum`: cumulative sums
- `rle`: counting number of repeated value
- `ifelse`: vectorized if…else

Exercises

- Create the dataframe `my_df <- data.frame(a = rnorm(10), b = rnorm(10), c = rnorm(10), d = rnorm(10))`. Print the median of column a.
- Print the medians of all columns using a `for` loop.
- Define the vector `firstnames<-c("Dora", "Adam", "Gergely")`. Create a vector `full_names` which combines the first names with the last name `Simon` using a loop. (hint: use `paste`)
- Create a vector `full_names_fast` which combines the first names with the last name `Simon` without a loop. (hint: define the last name and then use `paste` on the vectors)

Solutions

Create the dataframe `my_df <- data.frame(a = rnorm(10), b = rnorm(10), c = rnorm(10), d = rnorm(10))`. Print the median of column a.

```
my_df <- data.frame(a = rnorm(10), b = rnorm(10),
                    c = rnorm(10), d = rnorm(10))
median(my_df$a)
```

```
## [1] 0.5819043
```

Print the medians of all columns using a `for` loop.

```
for (x in my_df){
  print(median(x))
}
```

```
## [1] 0.5819043
## [1] 0.3612418
## [1] 0.2209345
## [1] 0.7046403
```

Aside: Use of square brackets

- `[`: subset of an object
    - usually the subset has the same type as the original object
- selects the container with its contents

Aside: Use of square brackets

- `[`: subset of an object
  - usually the subset has the same type as the original object
- selects the container with its contents

```
my_df["a"]
```

```
##              a
## 1   0.99843184
## 2   1.20375273
## 3   1.45426710
## 4   2.22915014
## 5   0.07335183
## 6   0.62568005
## 7   0.53812857
## 8  -0.96985580
## 9   0.13431042
## 10 -1.17379152
```

```
class(my_df["a"])
```

```
## [1] "data.frame"
```

Aside: Use of square brackets

- `[[`: extract one element from potentially many
  - the result is usually not the same type as the original
  - for vectors: vector with a single value
  - for data frames: column vector
  - for lists: one element
- selects the contents
- `$`: Accessing with that is a special case of `[[]]`

Aside: Use of square brackets

- `[[`: extract one element from potentially many
  - the result is usually not the same type as the original
  - for vectors: vector with a single value
  - for data frames: column vector
  - for lists: one element
- selects the contents
- `$`: Accessing with that is a special case of `[[]]`

```
my_df[["a"]]
```

```
##  [1]  0.99843184  1.20375273  1.45426710  2.22915014  0.07335183
##  [6]  0.62568005  0.53812857 -0.96985580  0.13431042 -1.17379152
```

```
class(my_df[["a"]])
```

```
## [1] "numeric"
```

Aside: Use of square brackets

- `[[`: extract one element from potentially many
  - the result is usually not the same type as the original
  - for vectors: vector with a single value
  - for data frames: column vector
  - for lists: one element
- selects the contents
- `$`: Accessing with that is a special case of `[[]]`

```
my_df[["a"]]
```

```
##  [1]  0.99843184  1.20375273  1.45426710  2.22915014  0.07335183
##  [6]  0.62568005  0.53812857 -0.96985580  0.13431042 -1.17379152
```

```
class(my_df[["a"]])
```

```
## [1] "numeric"
```

More details can be found here.

Solutions

Define the vector `firstnames<-c("Dora", "Adam", "Gergely")`. Create a vector `full_names` which combines the first names with the last name `Simon` using a loop. (hint: use `paste`)

```
firstnames<-c("Dora", "Adam", "Gergely")
full_names<-c()
for (i in 1:3){
  full_names[i]<-paste(firstnames[i], "Simon")
    }
full_names
```

```
## [1] "Dora Simon"    "Adam Simon"    "Gergely Simon"
```

Aside: Use of square brackets

```r
firstnames[1]
```

```
## [1] "Dora"
```

```r
firstnames[[1]]
```

```
## [1] "Dora"
```

Solutions

Create a vector `full_names_fast` which combines the first names with the last name `Simon` without a loop. (hint: define the last name and then use `paste` on the vectors)

```
firstnames<-c("Dora", "Adam", "Gergely")
lastname<-c("Simon")
full_names_fast<-paste(firstnames, lastname)
full_names_fast
```

```
## [1] "Dora Simon"    "Adam Simon"    "Gergely Simon"
```