Flow Control and making functions

Programming in Psychological Science

Michael Nunez & Hannes Rosenbusch

Basic Flow Control

Basic Flow Control

Flow control can be understood to be a set of built-in R functions that control and route the flow of information during the execution of an R function or script.

Fifa 2021 pseudo code for penalties

If player scores

then: increase goals by one

if not

then: show happy k & sad p

If-Statements

If-statements are of the form:

```
if(conditon){
   somecode
}
```

where *condition* is a logical test that evaluates to a logical variable of length and dimension 1 (i.e., TRUE or FALSE), and *somecode* is a set of R commands

If-Statements example

```
a <- 1
if(a == 1){
  print(a)
}
```

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

Example case

```
shot <- “left”
jumped <- “right”
goals <- 0
if(???){
      goals <- goals + 1
}
print(goals)
```

- {} groups commands to be executed
- Use indentation to keep your code readable (CMD/Ctrl + i)

Some More Examples

```
if(TRUE)
{
  print('Executed')
}
```

```
## [1] "Executed"
```

```
if(FALSE==FALSE)
{
  print('Executed')
}
```

?

Combining Logical Statements

- Combine different logical statements using logical operators:
  - Logical and: &
  - Logical or: | (inclusive or)
  - Logical negation: !

And

A	B	A & B
T	T	T
T	F	F
F	T	F
F	F	F

Or

A	B	A ∨ B
T	T	T
T	F	T
F	T	T
F	F	F

Combining Logical Statements

Some examples:

```
shot <- "missed"
jumped <- "left"
goals <- 0
if(??? | ???){
   goals <- goals + 0
}
```

Combining Logical Statements

Some examples:

```
shot <- "left"
jumped <- "left"
goals <- 0
if(????? | ?????){
goals <- goals + 1
}
```

Things That Can Go Wrong

```
x <- 1:5

if(x > 0)
{
  print('Yes')
}
```

```
## Warning in if (x > 0) {: the condition has length > 1
## and only the first element will be used
```

```
## [1] "Yes"
```

Things That Can Go Wrong

```
foo <- 'a'
if(foo)
{
 print('Correct')
}
```

```
## Error in if (foo) {:
argument is not interpretable as logical
```

More About If-Statements

```
if(1)
{
 print('Correct')
}
```

```
## [1] "Correct"
```

Things That Can Go Wrong

Very frequent error!

```
a <- 3
if(a = 3)
{
 print('Correct')
}
```

If-Else-Statements

Nested if-statements allow you to provide alternative commands if a logical test does not come out TRUE:

```
if (shot == jumped)
{
 print("no goal")
}
```

If-Else-Statements

Nested if-statements allow you to provide alternative commands if a logical test does not come out TRUE:

```
if (shot == jumped)
{
 print("no goal")
} else {
 print("goal")
}
```

If-Else-Statements

Nested if-statements can also be used to test conditions sequentially:

```
if(shot == "right"){
  jump <- "right"
} else if(shot == "left"){
  jump <- “left"
} else if(shot == "middle"){
  jump <- "middle"
} else {
  "haha you missed"
}
```

Vectorised If-Else-Statements

You can simultaneously carry out several logical tests and execute several commands depending on the outcome of each test:

```
ifelse(conditon, if true, if false)
```

where *condition* evaluates to a logical vector of length >= 1.

Vectorised If-Else-Statements

```
x <- 1:7
ifelse(x %% 2 == 0, 'even', 'odd')
```

```
## [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"
```

Vectorised If-Else-Statements

```
x <- 1:7
ifelse(x %% 2 == 0, x, 'odd')
```

```
##  [1] "odd" "2"   "odd" "4"   "odd" "6"   "odd"
```

Vectorised If-Else-Statements

So while

```
x <- 1:5
if(x > 0)
{
 print('Correct')
}
```

```
## Warning in if (x > 0) {: the condition has length > 1
## and only the first element will be used
```

```
## [1] "Correct"
```

doesn't work for vectors...

Vectorised If-Else-Statements

```
x <- 1:5
ifelse(x > 0, 'correct')
```

```
## [1] "correct" "correct" "correct" "correct" "correct"
```

... does work work for vectors.

RStudio Demo: A Game of Dice

Consider the following game of dice: A player rolls 3 dice. If all dice come up different the player wins 1 euro, otherwise they lose 3 euro.

RStudio Demo: A Game of Dice

Consider the following game of dice: A player rolls 3 dice. If all dice come up different the player wins 1 euro, otherwise he loses 3 euro.

```
wallet <- 10
dice <- sample(1:6, 3, replace=TRUE)
if(...)
{
  ...
} else {
  ...
}
wallet
```

RStudio Demo: A Game of Dice

Consider the following game of dice: A player rolls 3 dice. If all dice come up different the player wins 1 euro, otherwise he loses 3 euro.

```
wallet <- 10
dice <- sample(1:6, 3, replace=TRUE)
if(length(unique(dice)) == 3)
{
  wallet <- wallet + 1
} else {
  wallet <- wallet - 3
}
wallet
```

Loops

Explicit Loops

Loops are a form of flow control that allows you to carry out repeated computations that have the same underlying structure. The basic form of loops are while-loops and all other loops can be reduced to while-loops.

While-Loops

While-loops are of the form:

```
while(conditon){
   body
}
```

where *condition* is a logical test that evaluates to a logical variable of length and dimension 1 (i.e., TRUE or FALSE), and *body* is a set of R commands. The commands in the *body* are executed until *condition* is FALSE.

While-Loops

```
age <- 18
while(age <  25){
  print(“young”)
  age <- age + 1
}
print(“old”)
```

RStudio Demo: A Game of Dice

Imagine a player wants to keep playing the game of dice until he either loses all his money or has accumulated 20 euros. A single round looks like this:

```
wallet <- 10
dice <- sample(1:6, 3, replace=TRUE)
if(length(unique(dice)) == 3)
{
  wallet <- wallet + 1
} else {
  wallet <- wallet - 3
}
wallet
```

```
## [1] 7
```

RStudio Demo: A Game of Dice

Imagine a player wants to keep playing the game of dice until he either loses all his money or has accumulated 20 euros. What should the condition be?

```
wallet <- 10
while(...)
{
  dice <- sample(1:6, 3, replace=TRUE)
  if(length(unique(dice)) == 3)
  {
    wallet <- wallet + 1
  } else {
    wallet <- wallet - 3
  }
}
wallet
```

RStudio Demo: A Game of Dice

Imagine a player wants to keep playing the game of dice until he either loses all his money or has accumulated 20 euros. What should the condition be?

```
wallet <- 10
while((wallet > 0) & (wallet < 20))
{
  dice <- sample(1:6, 3, replace=TRUE)
  if(length(unique(dice)) == 3)
  {
    wallet <- wallet + 1
  } else {
    wallet <- wallet - 3
  }
}
wallet
```

RStudio Demo: A Game of Dice

Our player might also want to stop after a maximum of 10 rounds. How can we keep track of the rounds?

```
counter <- 0
wallet <- 10
while((wallet > 0) & (wallet < 20) & counter <= 10)
{
  counter <- counter + 1
  dice <- sample(1:6, 3, replace=TRUE)
  if(length(unique(dice)) == 3)
  {
    wallet <- wallet + 1
  } else
  {
    wallet <- wallet - 3
  }
}
wallet
```

For-Loops

For-loops are while-loops that run for a fixed number of iterations:

```
for(counter in 1:n){
  body
}
```

where *counter* cycles through the integers from 1 to *n* and *body* is thus evaluated *n* times.

For-Loops

```
for(i in 1:2)
{
  print(i)
}
```

```
## [1] 1
## [1] 2
```

For-Loops

is equivalent to

```
i <- 0
while(i < 2)
{
  i <- i + 1
  print(i)
}
```

```
## [1] 1
## [1] 2
```

For-Loops

You can loop over arbitrary vectors:

```
v <- letters[1:5]
v
## [1] "a" "b" "c" "d" "e"
for(someIndex in v)
{
  print(someIndex)
}
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

For-Loops

But you cannot change the vector within the loop:

```
random_number <- 3
for(some_index in 1:random_number)
{
  random_number <- 1
  print(paste("some_index:",some_index))
  print(paste(" random_number :", random_number ))
}
```

```
## [1] "some_index: 1"
## [1] " random_number : 1"
## [1] "some_index: 2"
## [1] " random_number : 1"
## [1] "some_index: 3"
## [1] " random_number : 1"
```

Potential end of Monday lecture

Potential start of Wednesday lecture

Advanced flow control
&
Making functions

Three Different Ways to Do One Thing

Change all even numbers in a vector to zero (3 ways):

Three Different Ways to Do One Thing

Change all even numbers in a vector to zero (3 ways):

```
set.seed(123)
x <- sample(1:10, size = 50, replace = TRUE)

x[x %% 2 == 0] <- 0 # indexing

ifelse(x %% 2 == 0, 0, x) # vectorised if

for(i in 1:length(x))
{
  if(x[i] %% 2 == 0)
  {
    x[i] <- 0
  }#end of if block
} # end of loop
```

Demo: Sum of a Vector

For a given numeric vector v, compute the sum of v using a for-loop.

Loops All The Way Down

You can nest loops within loops within loops...

```
for(first in c("Hans", "Tim", "Hannes"))
{
  for(last in c("Smith", "Rosenbusch"))
  {
    if(paste(first, last) == "Hannes Rosenbusch")
    {
      print(paste(first, last, " ...That is me!"))
    }else
    {
      print(paste(first, last, " ...is not me."))
    }
  }
}
```

Demo: Number Tables

Use a for-loop to generate the following matrix:

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

Demo: Number Tables

Use a for-loop to generate the following matrix:

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

```
my_matrix <- matrix(nrow = 3, ncol = 3)
for(row in 1:3)
{
  for(column in 1:3)
  {
  my_matrix[row,column] <- row * column
  }
}
```

Infinite Loops

You can get infinite loops:

```
i <- 0
while(i != 5)
{
  i <- i + 2
  print(i)
}
```

If this happens, click the red stopsign in the console.

Marginal Sums

Imagine you want to compute the sum of each row of a matrix. You can do that using a for-loop:

```
m <- matrix(1:4, 2, 2)
m
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

for(i in 1:2)
{
  print(sum(m[i,]))
}
## [1] 4
## [1] 6
```

Implicit Loops

A more elegant way to carry out operations over dimensions of matrices (or arrays or data frames) are implicit loops. These are provided by the `apply` family of functions. Implicit loops are the preferred way of looping in R!

Apply

The basic function is `apply`:

```
apply(x, margin, function, function_arguments)
```

where *x* is a data structure,

margin indicates the margin over which the function is applied,

function gives the name of the function,

and *function_arguments* is used to pass arguments to the function.

Apply

Using `apply` to compute the sum of each column:

```
m <- matrix(c(1:5, NA), 2, 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4   NA
```

```
apply(m, 2, sum, na.rm=TRUE)
```

```
## [1] 3 7 5
```

Apply

Using `apply` to compute the sum of each column:

```
m <- matrix(c(1:5, NA), 2, 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4   NA
```

```
apply(m, 2, function(x){sum(x > 2)})
```

Apply

Avoid unnecessary (explicit) loops!

```
m <- matrix(1:4, 2, 2); m
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
for(i in 1:2)
{
  print(sum(m[,i]))
}
```

```
## [1] 3
## [1] 7
```

Apply

Much shorter:

```
apply(m, 2, sum)
```

```
## [1] 3 7
```

Advanced Flow Control

Advanced Flow-Control

Nested loops are easy but often inefficient. Repeated if-else statements can often be replaced by switch-statements. These take the form:

```
switch(expr,
{body 1},
{body 2},
...)
```

Integer Switch

If *expr* evaluates to an integer, the corresponding statement is evaluated:

```
num <- sample(c(1, 2, 3), 1)
result <- switch(num,
       'HI',
       -999,
       my_vector <- c(3,4,5)
)
result
```

```
## [1]
“HI”
```

Integer Switch

If *expr* does not evaluate to an integer, nothing is returned:

```
num <- sample(c(0.1, 0.2, 0.3), 1)
result <- switch(num,
       'HI',
       -999,
       my_vector <- c(3,4,5)
)
result
```

```
## [1] NULL
```

String Switch

If *expr* evaluates to a string, the first statement that is preceded by a matching string is evaluated:

```
ff <- LETTERS[1:3]
switch(ff[1],
       B = {pi},
       A = {1},
       C = {exp(1)}
)
```

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

Extended control over while-loops is provided by `next` and `break`. `next` skips to the next evaluation of the loop whereas `break` completely stops the evaluation of the loop:

Advanced Flow-Control

Extended control over while-loops is provided by `next` and `break`. `next` skips to the remaining evaluation of the iteration whereas `break` completely stops the evaluation of the loop:

```
i <- 0
while(i < 5)
{
   i <- i+1
   if(i==2) next
   print(i)
}
```

```
## [1] 1
## [1] 3
## [1] 4
## [1] 5
```

Advanced Flow-Control

Extended control over while-loops is provided by `next` and `break`. `next` skips to the next evaluation of the loop whereas `break` completely stops the evaluation of the loop:

```
i <- 0
while(i < 5)
{
   i <- i+1
   if(i==3) break
   print(i)
}
```

```
## [1] 1
## [1] 2
```

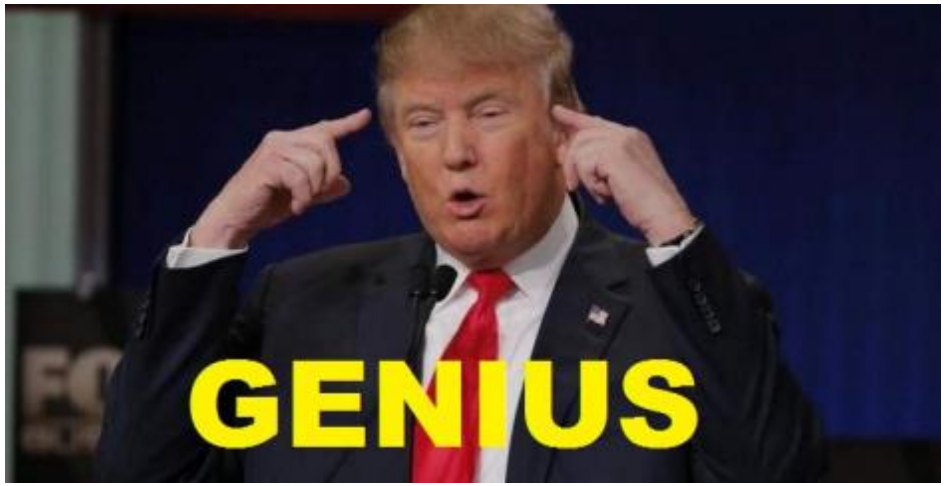
Flow control summary

- If, else if, else, for, while, and switch are important!
- They exist in virtually every programming language!
- Practicing flow control has large pay-offs!

Making your own functions

How to make a function in R:

```
name <- function(arg1, arg2){
      value <- arg1 + arg2
      return(value)
}
```



Different ways

Preferred way:

```
say_hello <- function() {
    return("hello")
}
```

Other ways:

```
say_hello <- function()
    return("hello")
```

Using functions

```
say_hello
```

```
## function() {
##  return("hello")
## }
```

```
say_hello()
```

```
## [1] "hello"
```

Multiple arguments

```
grade_student <- function(assignments, exam){

  avg_assignments <- mean(assignments)
  grade <- 0.6 * avg_assignments + 0.4 * exam
  return(grade)

}

grade_student(assignments = c(5,6,7,8), exam=9)
```

Default values

```
congratulations <- function(x = "everyone") {
  print(paste("Congratulations you passed,", x))
}
congratulations()
```

```
## [1] "Congratulations you passed, everyone"
```

Global vs Local Environment

Functions operate on their local environment

Functions cannot change objects in global environment

```
test_var <- 8
sqr <- function(x) {
     test_var <- x
     return(test_var *  test_var)
}
```

Global vs Local Environment

Functions operate on their local environment

Functions cannot change objects in global environment

```
test_var <- 8
foo <- function(x) {
      test_var <- x
      return(test_var *  test_var)
}
foo(5)
```

```
## [1] 25
```

```
test_var
```

```
## [1] 8
```

Global vs Local Environment

However, functions can use objects from global workspace

```
bar <- function(x) {
    return(x + test_var)
}
```

Global vs Local Environment

However, functions can use objects from global workspace

```
bar <- function(x) {
    return(x + test_var)
}
test_var <- 1:6
bar(3)
```

```
## [1] 4 5 6 7 8 9
```

Returning output

To use output in global environment explicitly return output using return() function

```
f <- function(x) {
      y <- x + 1
}
f(2)      # does this return output? yes
```

```
f <- function(x) {
      y <- x + 1
     return(y)
}
f(2)      # does return output
```

```
## [1] 3
```

Making a simple addition function

Add function

```
add <- function(x, na.rm = TRUE) {

      if(!is.numeric(x)) {
            stop("only numeric values allowed")
      }
      if(sum(is.na(x)) > 0) {
            warning("NA values detected")
      }
      if(na.rm) {
            x <- x[!is.na(x)]
      }
      summation <- 0
      for(i in  x) {
            summation <- summation +  i
      }
      return(summation)
}
```

Using your own functions

Source

If you want to load a function into the workspace:

```
source("my_functions.R")
```

Apply

Using apply to compute the sum of each column:

```
m <- matrix(c(1:5, NA), 2, 3)
```

```
apply(m, 2, add, na.rm=FALSE)
```

Functions summary

- Functions are fun to make
- Functions are easy to reuse
- They make code readable
- There is a function for everything

It’s been quite a journey