R403: Probabilistic and Statistical Computations with R

Topic 09: Control Structures

Kaloyan Ganev

2022/2023

Lecture Contents

Introduction

Introductory notes

- Control structures allow controlling the programme flow
- In simpler language, they stir program execution in one direction or another depending on whether a condition is fulfilled or not
- The simplest control structures are if-else statements
- They check whether a condition is true and execute a piece of code or another
- Loops are similar in the following respect: they operate also as long as a condition is met
- Still, there is a major difference: in if-else statements code is executed only once; in loops it may be executed many times

If-else statements

If-else statements

- The if-else control structure is among the simplest in the R language
- It is very similar to the analogical structures that other program languages (such as C++, Python, Java, etc.) have
- What is does is to check whether a condition is logically true and then execute a code block
- In its simplest form, it does not contain an `else` clause
- If the condition is true, then the code is executed; otherwise, nothing is done, and R proceeds with the rest of your script (if any)
- Let's take an example

If-else statements (2)

- Generate a random number x by drawing from the standard normal distribution:

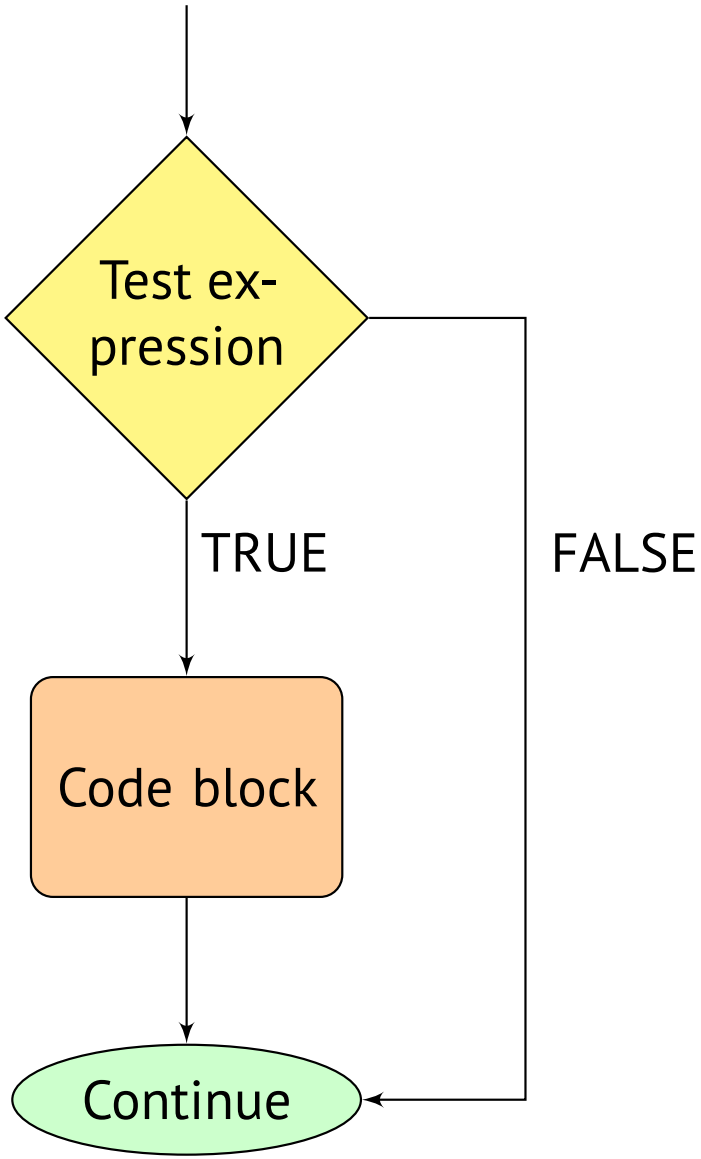
```
x <- rnorm(1)
```

- The simple control statement will be:

```
if (x <= 0){
  y <- 10
}
```

- Interpreted as follows: if x turns out to be non-positive (TRUE condition), then generate y and assign the number 10 to it
- If this condition is FALSE, nothing happens

If-else statements (3)



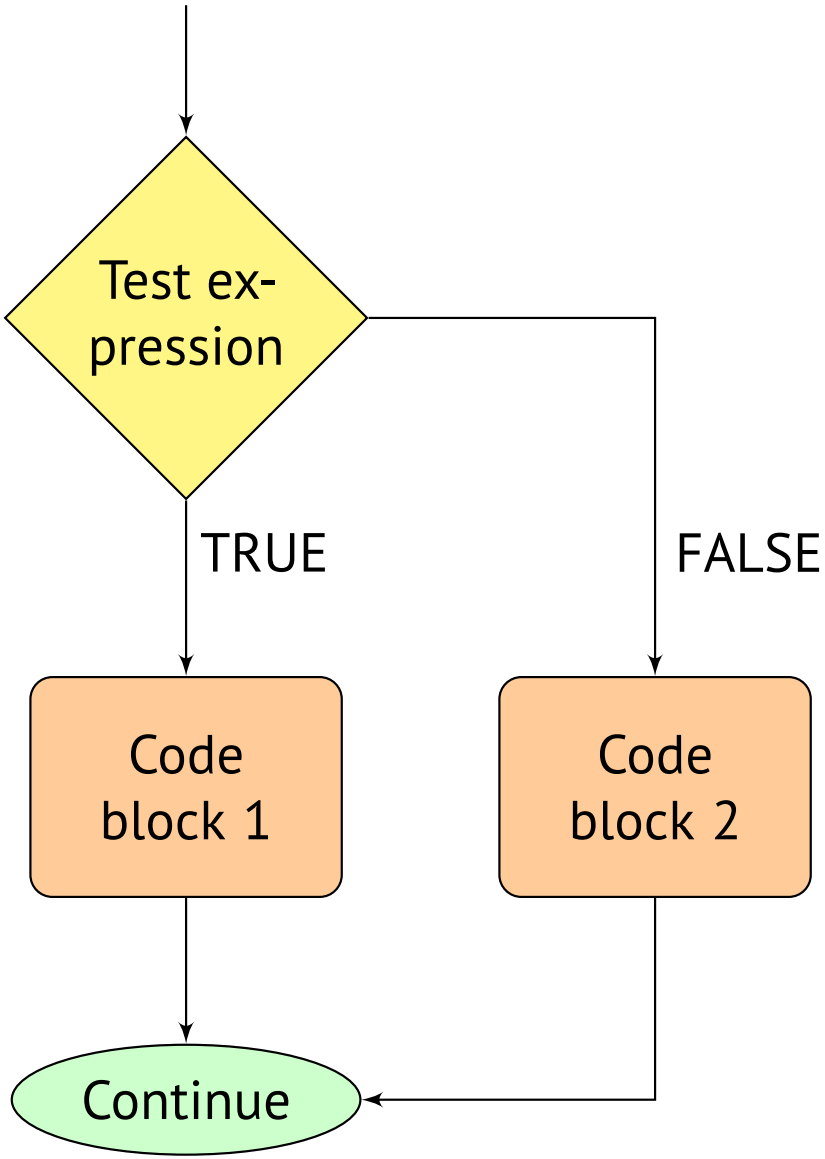
If-else statements (4)

- Now let's make our R script do something also in the case when the condition is FALSE
- We just add the `else` clause
- In our example, if the condition is not met, we assign the value of 20 to y:

```
if (x <= 0){
  y <- 10
  } else {
  y <- 20
  }
```

- Note that `else` should be in the same line as the closing curly bracket of `if`, otherwise R will produce an error

If-else statements (5)

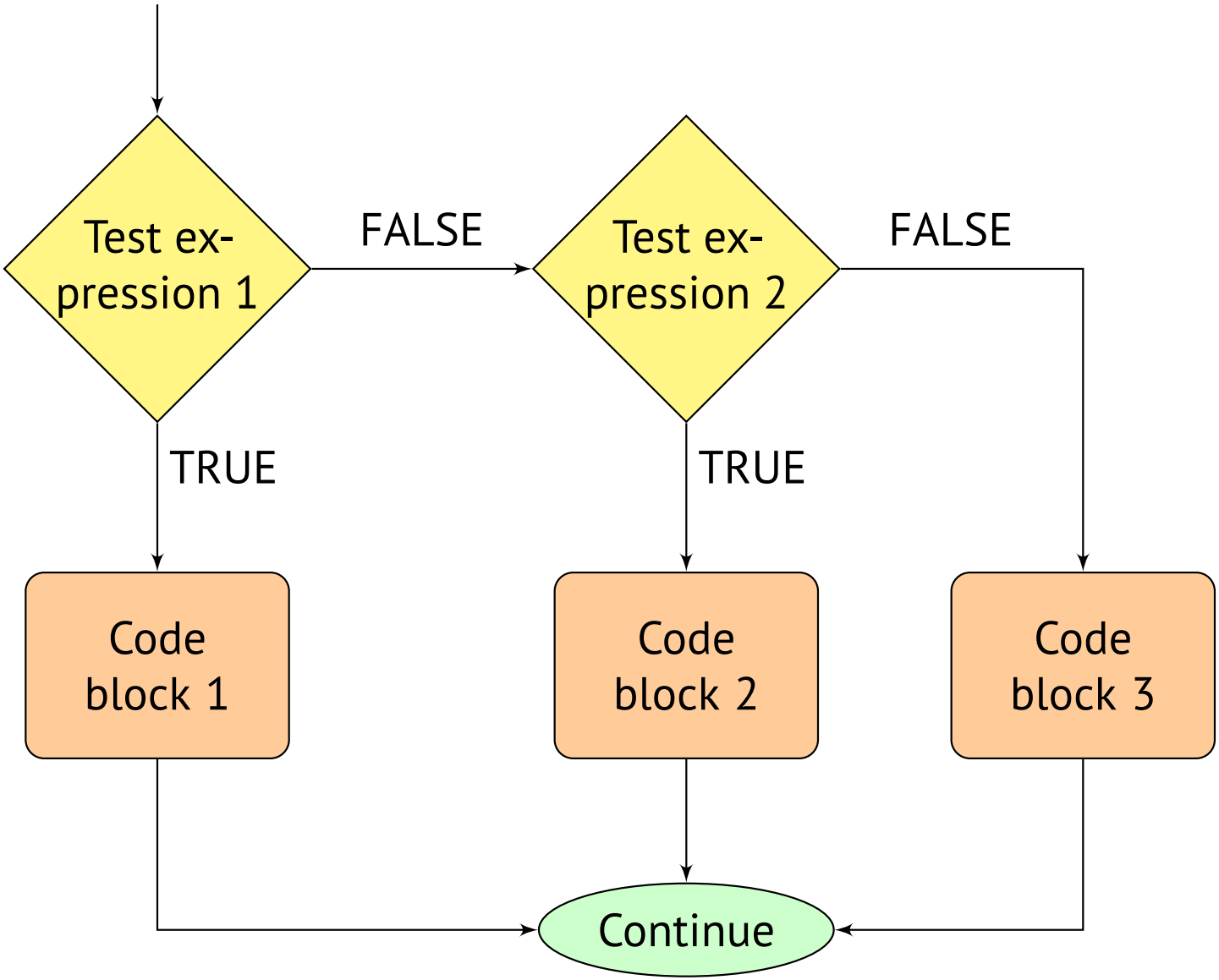


If-else statements (6)

- If-else statements can be nested
- This means that the first check on a condition might lead to another condition to be checked (this can be done many times, actually)
- We will take again an example
- Let us make an additional check if the condition $x \leq 0$ is FALSE, this time whether $0 < x \leq 3$
- If the latter is TRUE, we assign 15 to y, otherwise we assign 20 to y:

```
if (x <= 0){
  y <- 10
  } else if (x > 0 && x <=3){
    y <- 15
    } else {
      y <- 20
    }
```

If-else statements (5)



Logical operations

- We introduced one already (logical AND), or at least a form of it
- Now, let's get through all of them, one by one
- Logical negation (NOT): implemented through '!'; example:

```
if (!(x > 10)){
  z <- 300
  }
```

- Logical OR: '||' or '|'
- Logical AND: '&&' or '&'
- The longer versions of the above start from the left-most elements of the compared objects and proceed until the result is determined; the shorter versions make the comparison for each element (i.e. comparison is element-wise)
- Therefore, the longer versions are usually preferred as they might take less time

Logical operations (2)

- Logical XOR: `xor(x,y)`
- Interpreted as 'exclusive OR', results in TRUE only when x and y differ
- Best illustrated by means of the following truth table:

x	y	$xor(x,y)$
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

- `isTRUE(x)`: if x is a single-value logical variable, the command checks whether x is identical to TRUE

Vectorized `if` operations

- The common `if-else` structure is designed to operate on just one logical value at a time
- What if it is necessary to perform programme flow control on vectors of logical values?
- There comes the R command `ifelse`
- Syntax:

```
ifelse(test, value_for_TRUE_elements, value_for_FALSE_elements)
```

- In the above, `test` is a logical vector, the remaining two are self-explanatory (only note they have the same length as `test`)

Vectorized `if` operations (2)

- Let's look at an example: simulate 10000 tosses of a coin and record the outcomes as "Heads" and "Tails"

```
coin_toss <- ifelse(rbinom(10000, 1, 0.5), "Head", "Tail")
```

- Explanation: we perform here 10000 Bernoulli trials with a fair coin (equal probabilities of heads and tails)
- The example shows that in addition to logical vectors, vectors of values coercible to logical values are also admissible as arguments to the function

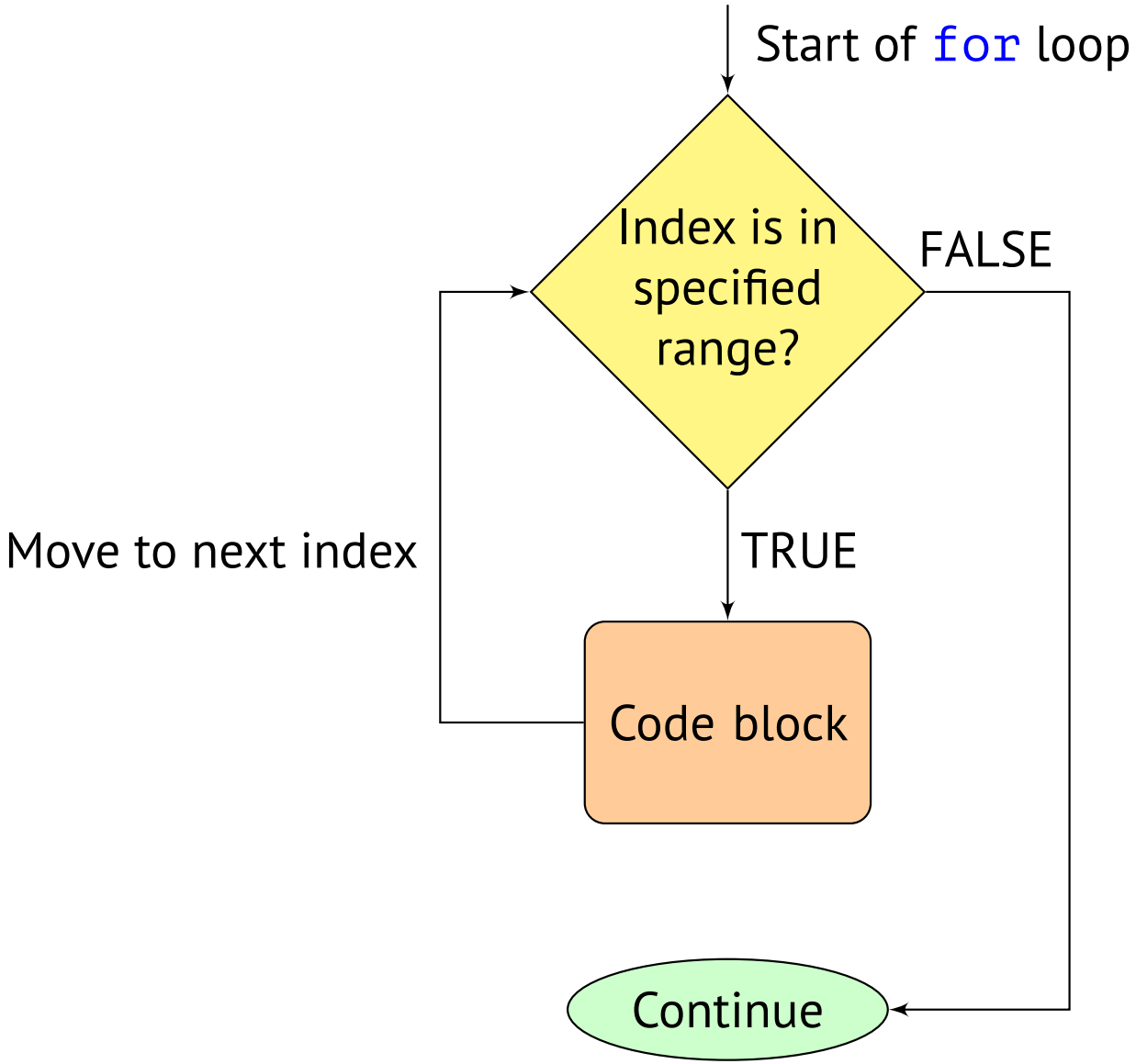
Loops

`for` loops

- Loops are about repeating a certain operation (set of operations) many times
- The `for` loop is designed to perform **a fixed number of iterations**
- It is maybe the most used loop type in R, and in most cases it is sufficient to perform the necessary task
- Usually, `for` loops are used to iterate through the items of an array (matrix, vector) or a list
- Index variables are used in order to go over the respective elements
- Index values are integers contained in a vector
- This leads to the following generalization of syntax concerning `for` loops:

```
for (i in <vector>){
  operation1
  operation2
  ...
  }
```

`for` loops (2)


Start of for loop
Index is in specified range?
FALSE
TRUE
Move to next index
Code block
Continue

Examples of `for` loops

- Example:

```
x <- c(1,3,5,7,9,11)
for (i in x){
  print(i%%3)
  }
```

- Another example: join capital letters and corresponding indices:

```
vec1 <- character(length(LETTERS))
for (i in 1:length(LETTERS)){
  vec1[i] <- paste(LETTERS[i],i, sep = "")
  }
```

Nested `for` loops

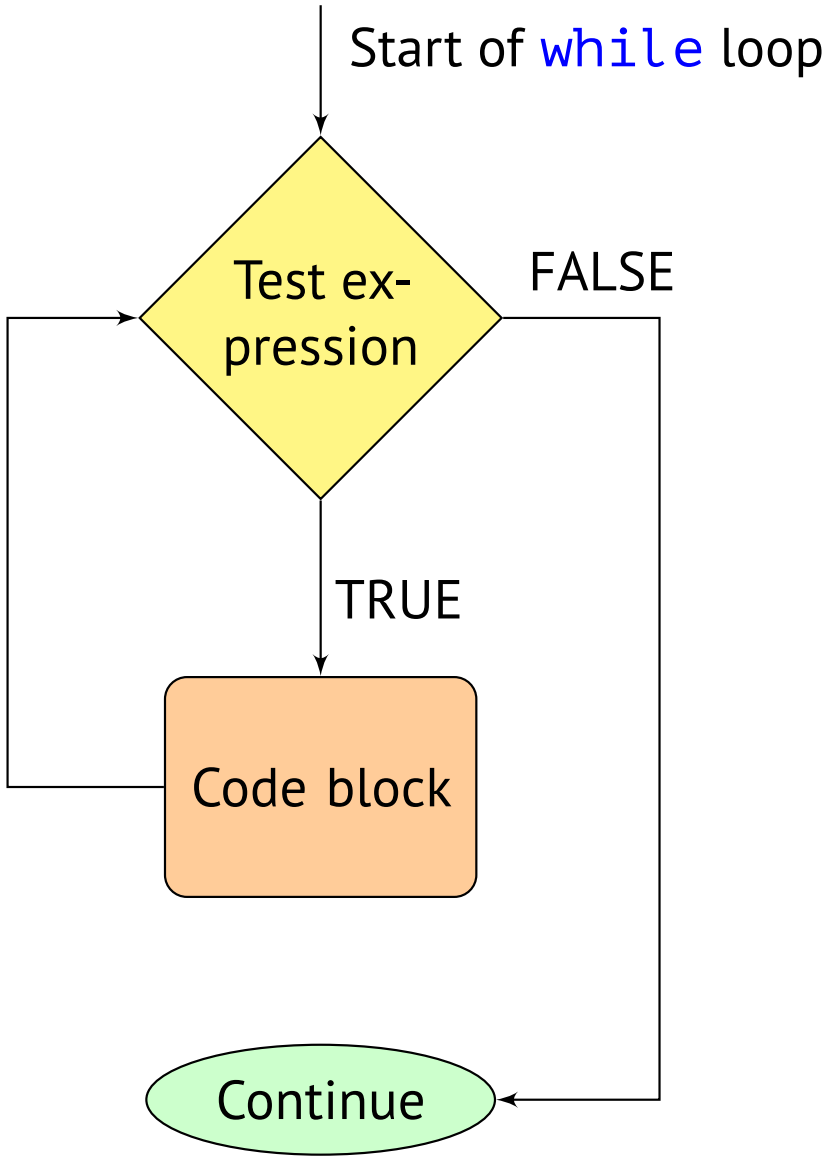
- Suitable when it is necessary to iterate over multiple dimensions
- An example: create a 3D array, $5 \times 10 \times 15$; each element should equal the product of dimension indices

```
array1 <- array(dim = c(5,10,15))
for (i in 1:dim(array1)[1]){
  for (j in 1:dim(array1)[2]){
    for (k in 1:dim(array1)[3]){
      array1[i,j,k] <- i*j*k
      }
    }
  }
```

`while` loops

- Suitable when the number of iterations is not known in advance
- A certain operation is iterated while a condition is TRUE; when it turns to FALSE, the operation is interrupted
- Two extreme possibilities emerge:
  1. The condition is FALSE at the start of the iteration; then, the code in the body of the loop is not executed at all
  2. The FALSE condition is never encountered which makes the loop run indefinitely (infinite number of iterations)
- Therefore, `while` loops should be used very carefully

`while` loops (2)



Examples of `while` loops

- Initialize x to equal 0
- Generate a random number from the standard normal distribution
- Add it to x
- Do so as long as x does not exceed 3

```
x <- 0
while (x <= 3) {
  z <- rnorm(1)
  x <- x + z
  }
```

- A more practical example: keep asking for a user name until the user supplies it:

```
uname <- character()
while (length(uname) == 0){
  cat("Please enter your user name:")
  uname <- scan(what=character(),nmax=1,quiet=TRUE)
  }
```

Some clarifications on `cat()` and `scan()`

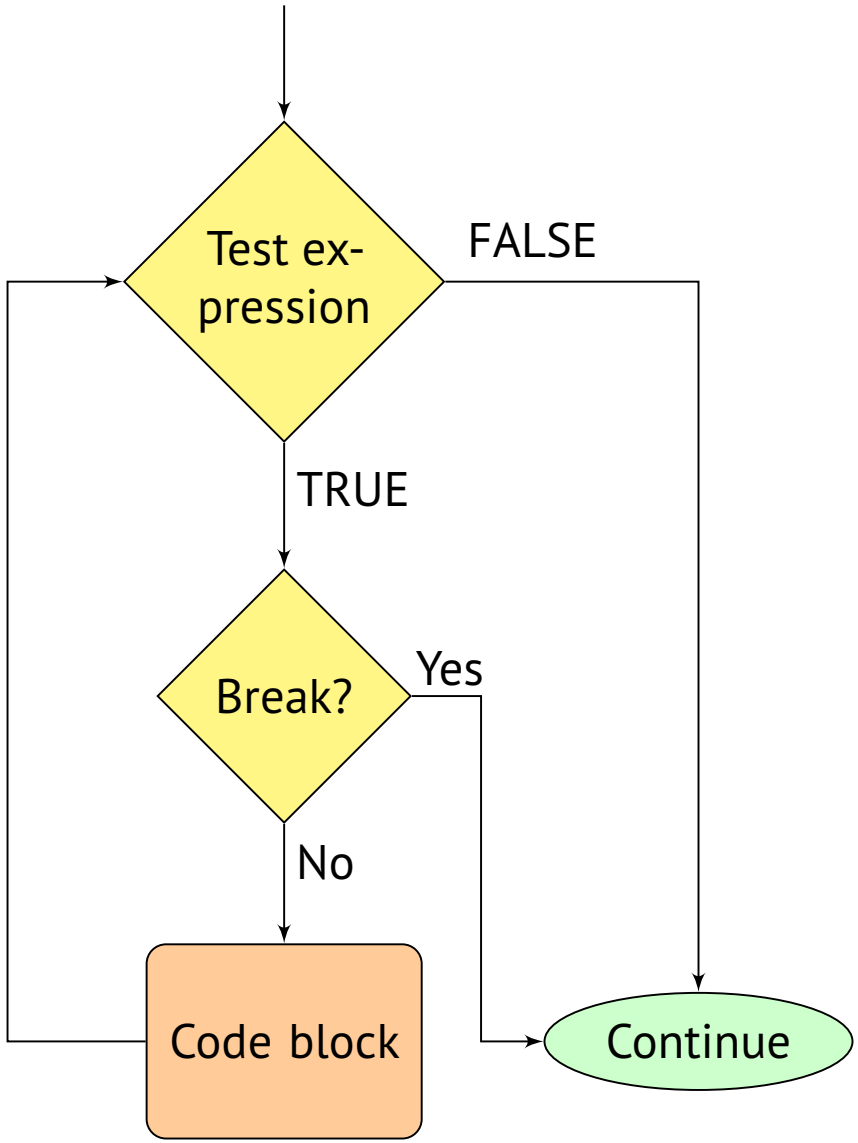
- `cat()`:
  1. Prints output to the screen or to a file
  2. If necessary, coerces to character mode
  3. Valid only for atomic data types (logical, integer, numeric, complex, character) and for names
- `scan()`:
  1. Reads data into a vector or list from the console or file
  2. The `what` option defines the data type to be read
  3. The `nmax` option defines the maximum number of values to be read
  4. The `quiet` option, if FALSE, makes `scan()` print a message saying how many items have been read

`break` statements

- When encountered in a loop, it makes R exit the loop and proceed with the remaining part of your script (if any)
- This means there are two options: if it is in an inner part of a loop, it is exited and R goes to the outer loop; if the loop is not nested, R goes to the code that follows after the loop
- Example: Find the largest integer that divides another integer with no remainder:

```
int1 <- 10000
int2 <- int1
div_by <- 2437
while (int1 > 0)  {
  if (int1%%div_by == 0)  {
    print(paste("The largest integer between 0 and", int2, "
        divisible by", div_by, "is ", int1, "."))
  break
  }
int1 = int1 - 1
}
```

`break` statements (2)

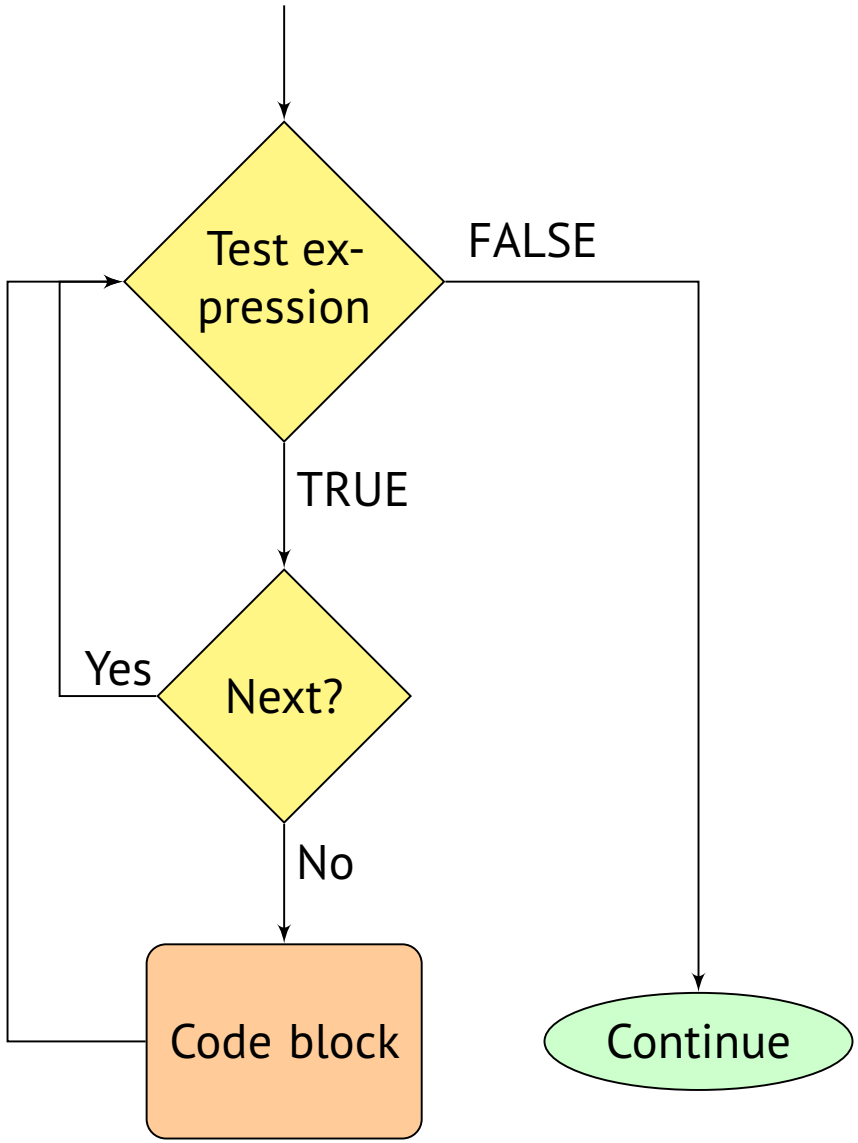


`next` statements

- Useful when the an iteration of a loop has to be skipped without interrupting the loop
- When encountered, it leads to skipping the current iteration and moving to the next one (i.e. it goes to evaluating the condition of the loop and then to execution of the corresponding code)
- Example: Get all odd integers between 1 and 100 in a vector

```
int3 <- 0
vec_odd <- integer()
for (i in c(1:100)){
  if(i%%2 == 0){
    next
    }
vec_odd <- c(vec_odd,i)
}
```

`next` statements (2)

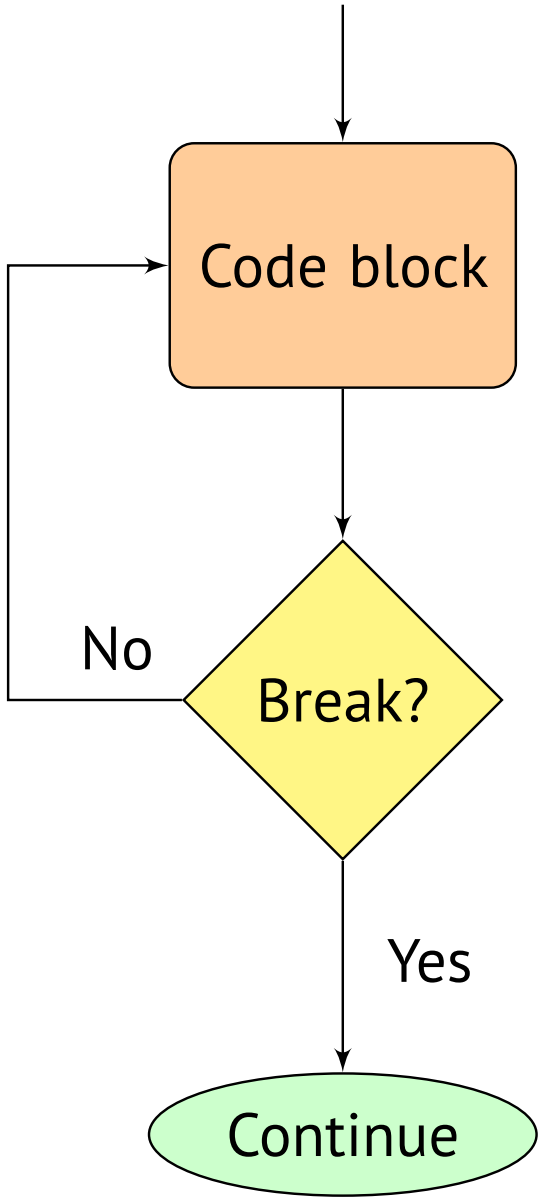


`repeat` loops

- Iterates a code block many times
- Practically infinite loops (Use very cautiously!)
- There is no condition to check in the beginning before the code block is entered
- This means that the `repeat` loop is executed at least once
- The loop needs to be explicitly terminated by a `break` statement
- Example:

```
uname <- character()
repeat{
  cat("Enter your username:")
  uname <- scan(what=character(),nmax=1,quiet=TRUE)
  if(length(uname) != 0){
    break
  }
}
```

`repeat` loops (2)



Loop functions

The `apply` family of functions

- For large and multi-dimensional data objects, loops can be quite tedious and slow
- There are, fortunately, the so-called vectorized functions which perform the loop operations in a much more efficient way
- Thus, resources (time, human, computing) are economized and can be channelled to better alternative uses
- The `apply` family of functions is a widely used set of functions that perform tasks of this kind
- The actual looping is performed by pre-compiled C code which is a lot faster
- We will look in terms at the `apply()`, `lapply()`, `sapply()`, `mapply()`, `rapply()`, `vapply()` and `tapply()` functions
- Then we will consider several more examples of other useful vectorized functions (`split()` and `sweep()`)

The `apply()` function

- Operates on arrays (incl. matrices, but not single-dimension vectors)
- General construct:

```
apply(x, MARGIN, FUN, ...)
```

where `x` is the name of the respective array, `MARGIN` defines the dimension(s) to which to apply the respective function, and `FUN` is the function to apply; the "`...`" stand for possible options that `FUN` might need to use

- Example (on a matrix):

```
m1 <- matrix(1:100, nrow = 10)
apply(m1, MARGIN = 1, mean)
apply(m1, MARGIN = 2, sum)
apply(m1, MARGIN = c(1,2), sqrt)
```

The `apply()` function (2)

- Returns an object which has the type of its argument (in this respect, it depends on the value of `MARGIN`)
- If it receives as an argument an object different from an array, it coerces it to an array if possible
- This function is actually not faster than a regular loop but is much more parsimonious (just one line of code)
- `colSums`, `rowSums`, `colMeans`, and `rowMeans` actually can do some of the jobs that `apply()` does but at a significantly greater speed

The `lapply()` function

- Works on lists (that's where the 'l' in `lapply()` comes from) and returns a list
- If the input object is not a list, if it is possible, it is coerced to a list
- Loops over each element and applies to it a specified function
- Written in C, so it provides much greater speed of execution of tasks
- General construct:

```
lapply(x, FUN, ...)
```

where `x` is a list, `FUN` is the function that is applied to the list's elements, and the dots are optional and stand for the arguments of `FUN` that might be used

The `lapply()` function (2)

- Example of usage:

```
object1 <- list(arr1 = array(rnorm(1000),dim = c(10,10,10)),
    vect1 = rnorm(10), mat1 = matrix(rnorm(100), nrow = 10))
lapply(object1, sum)
```

- The output is a list

The `sapply()` function

- `sapply()` is a variant of `lapply()` and works similarly to it
- The major difference is that it attempts to simplify the result (this is where the 's' comes from)
- Simplification works in the following alternative ways:
  - If the input list is of length zero, a zero-length list is also output
  - If each element of the input list is of length one, `sapply()` returns a vector
  - If list elements' length is equal but greater than one, it outputs a matrix (2D array)
  - In every other instance, a list of the same length as the input list is returned

The `sapply()` function (2)

- Example of usage: we will use the same object that we used in the `lapply()` example:

```
sapply(object1, sum)
```

- The output is a vector

The `vapply()` function

- Also used to simplify output in order to produce an atomic vector
- 'v' comes from 'verbose', i.e. uses more words in the description of the task
- At the same time, if it encounters an error, it provides more information on it (`sapply()` is actually giving no such information and even you don't know there's been an error)
- Unlike `sapply()` which tries to guess the output type, in `vapply()` output type is explicitly specified by means of an additional argument
- By this, execution speed is higher, and results are much more consistent with expectations (e.g. of data type)

The `vapply()` function (2)

- General construct:

```
vapply(x, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

- As output type is pre-specified, `vapply()` returns a vector or an array
- If `length(FUN.VALUE) == 1`, a vector having the same length as `x` is returned; else, an array is returned.
- If `FUN.VALUE` is not an array, R outputs a matrix with dimensions `c(length(FUN.VALUE),length(x))`; else it outputs an array a with dimensions `c(dim(FUN.VALUE), length(x))`
- Example:

```
vapply(-10 + 0i:10 + 0i, sqrt, 1i) # outputs complex numbers
vapply(1:10, log, 1) # outputs floating point numbers
```

The `mapply()` function

- As the 'm' in the name suggests, `mapply()` is a multivariate version of `lapply()` and `sapply()`
- Multivariate in the sense that it can apply the specified function to multiple inputs (i.e. multiple lists)
- General construct:

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES =
    TRUE)
```

- Note that unlike in the previous cases, the function to apply, `FUN`, is written first
- Next follows the list of inputs (the dots)

The `mapply()` function (2)

- `MoreArgs` relates to the additional arguments of `FUN` that are used (if necessary)
- If `USE.NAMES` equals TRUE and if X is character, the values of `x` are used as names for the output (unless names of the elements of `x` have already been explicitly specified)
- Example:

```
mapply(rep,5:1,1:5) # Equivalent to mapply(rep,x = 5:1,times =
    1:5)
```

The `rapply()` function

- 'r' comes from 'recursive', i.e. `rapply()` is a recursive variant of `lapply()`
- Applies recursively a function to all elements of a list
- General construct:

```
rapply(x, FUN, classes = "ANY", deflt = NULL, how = c("unlist", "
    replace", "list"), ...)
```

- The input `x` should be a list (or be coercible to a list)
- `FUN` is the function that is applied
- `classes` selects the classes of non-list elements to which to apply `FUN`

The `rapply()` function (2)

- If either `how = ‘‘list’’` or `how = ‘‘unlist’’`:[1] the input list is copied, all non-list elements are replaced by the result of applying `FUN`; all other elements are replaced by `deflt`
- If `how = ‘‘replace’’`: each non-list element is replaced by the result of applying `FUN`; the elements which not match the class definition are directly copied to the new list (in other words, in such a case the `deflt` option is irrelevant)
- Examples:

```
rapply(object1, mean)
rapply(object1, mean, how = "unlist")
rapply(object1, mean, how = "list")
object2 <- list(a1 = c(1:100), b1 = c(101:200), char1 <- "Text")
rapply(object2, log, classes = "integer", how = "replace")
```

[1] If `how = ‘‘unlist’’`, the resulting list is unlisted by means of the command `unlist()` with the option `recursive = TRUE`.

The `tapply()` function

- Applies a specified function to a subset of a vector
- General construct:

```
tapply(x, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- Selection of elements is carried out on the basis of the values of `INDEX`
- `INDEX` itself has to be of either factor (i.e. a categorical variable) or logical type and has to have the same length as `x`
- It is possible to use multiple indexes; in such a case, a list of indexes is created
- Attempts to simplify the result if `simplify = TRUE`; else it returns a list

The `tapply()` function (2)

- Example:

```
m1 = matrix(1:9, 3, 3)
idx1 = matrix(c(1,2,1,3,2,2,1,2,3), 3, 3)
tapply(m1, idx1, sum)
```

- Now, try to figure this one out:

```
x <- 1:100
idx2 <- rep(c('A1', 'A2', 'A3'), length = 100)
idx3 <- rep(c('B1', 'B2' ,'B3', 'B4'), length = 100)
idx4 <- rep(c('C1','C2','C3','C4', 'C5'), length = 100)
tapply(x, list(idx2, idx3, idx4), mean, na.rm = TRUE)
```

The `split()` function

- Not a loop function itself but used in combination with loop functions
- General construct:

```
split(x, f, drop = FALSE, ...)
```

- Splits the data contained in the vector `x` into the groups defined by the factor variable `f`.
- Example

```
for (i in 1:length(y)){
  if (y[i]%%2 == 0){
    f1[i] <- "A"
  } else {
    f1[i] <- "B"
    }
  }
lapply(split(x, f1), mean)
```

- There is also an `unsplit()` command, check it out

The sweep() function

- Gets an array as an input, returns an array
- The output is obtained by sweeping out a summary statistic
- General construct:

```
sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)
```

- Examples:

```
mat1 <- matrix(1:12, nrow = 3)
sweep(mat1, 2, colMeans(mat1), "-") # remove column means from
    elements
sweep(mat1, 2, colSums(mat1), "/") # divide each element by the
    column sum

arr1 = array(1:18, dim = c(3,3,2))
sweep(arr1, 1, apply(X, 1, mean)) # sweeps out the row means from
     each element
sweep(arr2, 2, apply(X, 2, mean)) # sweeps out the column means
```

References

- Cotton, R. (2013): *Learning R*, O'Reilly, ch. 8
- Peng, R. (2020): *R Programming for Data Science*, LeanPub, ch. 18