# Indexing and iteration

Statistical Computing, STA3005

Jan 15, 2024

## Last chapter: Basic data structures

- We write programs by composing functions to manipulate data
- The basic data types let us represent Booleans, numbers, and characters
- Data structures let us group together related values
- Vectors let us group values of the same type
- Arrays add multi-dimensional structure to vectors
- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Data frames are a special case of lists, allowing each column to have a different data type

## Part I: Indexing

## How R indexes vectors, matrices, lists

Different from C and Python, indexes in R start from 1. There are three ways to index a vector, matrix, data frame, or list in R:

1. Using explicit integer indices (or negative integers)
2. Using a Boolean vector (often created on-the-fly)
3. Using names

Note: in general, we have to set the names by ourselves. Use `names()` for vectors and lists, and `rownames()`, `colnames()` for matrices and data frames

## Indexing with integers

The most transparent way. Can index with an integer, or integer vector (or negative integer, or negative integer vector). Examples for vectors:

```
set.seed(33) # For reproducibility
x.vec = rnorm(6) # Generate a vector of 6 random standard no
x.vec
```

```
## [1] -0.13592452 -0.04079697  1.01053901 -0.15826244 -2.15
```

```
x.vec[3] # Third element
```

```
## [1] 1.010539
```

```
x.vec[c(3,4,5)] # Third through fifth elements
```

```
## [1]  1.0105390 -0.1582624 -2.1566375
```

```
x.vec[3:5] # Same, but written more succintly
```

```
## [1]  1.0105390 -0.1582624 -2.1566375
```

```
x.vec[c(3,5,4)] # Third, fifth, then fourth element
```

```
## [1]  1.0105390 -2.1566375 -0.1582624
```

```
x.vec[-3] # All but third element
```

```
## [1] -0.13592452 -0.04079697 -0.15826244 -2.15663750  0.49
```

```
x.vec[c(-3,-4,-5)] # All but third through fifth element
```

```
## [1] -0.13592452 -0.04079697  0.49864683
```

```
## [1] -0.13592452 -0.04079697  0.49864683
```

```
x.vec[-c(3,4,5)] # Same
```

```
## [1] -0.13592452 -0.04079697  0.49864683
```

```
x.vec[-(3:5)] # Same, more succint (note the parantheses!)
```

```
## [1] -0.13592452 -0.04079697  0.49864683
```

Examples for matrices:

```
x.mat = matrix(x.vec, 3, 2) # Fill a 3 x 2 matrix with those
                            # column major order
x.mat
```

```
##             [,1]       [,2]
## [1,] -0.13592452 -0.1582624
## [2,] -0.04079697 -2.1566375
## [3,]  1.01053901  0.4986468
```

```
x.mat[2,2] # Element in 2nd row, 2nd column
```

```
## [1] -2.156638
```

```
x.mat[5] # Same (note this is using column major order)
```

```
## [1] -2.156638
```

```
x.mat[2,] # Second row
```

```
## [1] -0.04079697 -2.15663750
```

```
x.mat[1:2,] # First and second rows
```

```
##              [,1]       [,2]
## [1,] -0.13592452 -0.1582624
## [2,] -0.04079697 -2.1566375
```

```
x.mat[,1] # First column
```

```
## [1] -0.13592452 -0.04079697  1.01053901
```

```
x.mat[,-1] # All but first column
```

```
## [1] -0.1582624 -2.1566375  0.4986468
```

Examples for lists:

```
x.list = list(x.vec, letters, sample(c(TRUE,FALSE),size=4,re
x.list
```

```
## [[1]]
## [1] -0.13592452 -0.04079697  1.01053901 -0.15826244 -2.15
##
## [[2]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## [[3]]
## [1]  TRUE  TRUE FALSE FALSE
```

```
x.list[[3]] # Third element of list
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

```
x.list[3] # Third element of list, kept as a list
```

```
## [[1]]
## [1]  TRUE  TRUE FALSE FALSE
```

single bracket

```
x.list[1:2] # First and second elements of list (note the si
```

```
## [[1]]
## [1] -0.13592452 -0.04079697  1.01053901 -0.15826244 -2.15
##
## [[2]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
x.list[-1] # All but first element of list
```

```
## [[1]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## [[2]]
## [1]  TRUE  TRUE FALSE FALSE
```

What happens using double brackets `[[ ]]`?

```
x.list[[1]] # First element of list
```

```
## [1] -0.13592452 -0.04079697  1.01053901 -0.15826244 -2.15
```

```
x.list[[c(1,3)]] # Third element of the first element of lis
```

```
## [1] 1.010539
```

```
x.list[[1]][3] # Same
```

```
## [1] 1.010539
```

```
x.list[[-1]] # Error
```

```
## Error in x.list[[-1]]: invalid negative subscript in get1
```

## Indexing with booleans

This might appear a bit more tricky at first but is *very useful*, especially when we define a Boolean vector "on-the-fly". On-the-fly means we instantly create and use the Boolean vector in the brackets. Examples for vectors:

```
x.vec[c(F,F,T,F,F,F)] # Third element
```

```
## [1] 1.010539
```

```
x.vec[c(T,T,F,T,T,T)] # All but third element
```

```
## [1] -0.13592452 -0.04079697 -0.15826244 -2.15663750  0.49
```

```
pos.vec = x.vec > 0 # Boolean vector indicating whether each
pos.vec
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE  TRUE
```

```
x.vec[pos.vec] # Pull out only positive elements
```

```
## [1] 1.0105390 0.4986468
```

```
x.vec[x.vec > 0] # Same, but more succint (this is done "on-
```

```
## [1] 1.0105390 0.4986468
```

Works the same way for lists; similarly, we can apply logical indexing for matrices

## Indexing with names

必须先设置 names

Indexing with names can also be quite useful. We must have names in the first place; with vectors or lists, use `names()` to set the names

```
names(x.list) = c("normals", "letters", "bools")
x.list[["letters"]] # "letters" (second) element
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
x.list$letters # Same, just using different notation
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
x.list[c("normals","bools")]
```

```
## $normals
## [1] -0.13592452 -0.04079697  1.01053901 -0.15826244 -2.15
##
## $bools
## [1]  TRUE  TRUE FALSE FALSE
```

```
rownames(x.mat) <- c("a", "b", "c")
colnames(x.mat) <- c("X", "Y")
x.mat["a",]
```

```
##          X          Y
## -0.1359245 -0.1582624
```

```
x.mat[,"X"]
```

```
##           a           b           c
## -0.13592452 -0.04079697  1.01053901
```

```
x.mat[,"x"]
```

(R 区分大小写)

```
## Error in x.mat[, "x"]: subscript out of bounds
```

```
x.mat$X
```

(matrix 没有 $ 运算符)

```
## Error in x.mat$X: $ operator is invalid for atomic vector
```

- We will see indexing by names being especially useful when we talk more about data frames, shortly
- Names in R are **case sensitive**

# Part II: Control flow (if, else, etc.)

## Control flow

Summary of the control flow tools in R:

- `if()`, `else if()`, `else`: standard conditionals
- `ifelse()`: conditional function that vectorizes nicely
- `switch()`: handy for deciding between several options

## `if()` and `else`

Use `if()` and `else` to decide whether to evaluate one block of code or another, depending on a condition (不能是 vector)

```
x = 0.5

if (x >= 0) {
  x
} else {
```

```
  -x
}
```

```
## [1] 0.5
```

```
if (x >= 0) x else -x # shortening
```

```
## [1] 0.5
```

- Condition in `if()` needs to give one `TRUE` or `FALSE` value
- Note that the `else` statement is optional
- Single line actions don't need braces, i.e., could shorten above to `if (x >= 0) x else -x` (单行才行)

## `else if()`

We can use `else if()` arbitrarily many times following an `if()` statement

```
x = -2

if (x^2 < 1) {
  x^2
} else if (x >= 1) {
  2*x-1
} else {
 -2*x+1
}
```

```
## [1] 5
```

- Each `else if()` only gets considered if the conditions above it were not `TRUE`
- The `else` statement gets evaluated if none of the above conditions were `TRUE`
- Note again that the `else` statement is optional

# Quick decision making

In the `ifelse()` function we specify a condition, then a value if the condition holds, and a value if the condition fails

```
ifelse(x > 0, x, -x)
```

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

```
y.vec = rnorm(6)
y.vec
```

```
## [1]  0.75457795 -1.09954561  0.16734996 -0.02928348  1.87
```

```
ifelse(y.vec > 0, y.vec, 0) # positive part of y.vec
```

```
## [1] 0.7545780 0.0000000 0.1673500 0.0000000 1.8758450 0.2
```

One advantage of `ifelse()` is that it vectorizes nicely

# Deciding between many options (使用 options 的 name 来决定运行哪个)

Instead of an `if()` statement followed by `elseif()` statements (and perhaps a final `else`), we can use `switch()`. We pass a variable to select on, then a value for each option

```
type.of.summary = "mode"

switch(type.of.summary,
       mean=mean(x.vec),
       median=median(x.vec),
       histogram=hist(x.vec),
       "I don't understand")
```

if/elseif (type.of.summary == *) {...}

else

```
## [1] "I don't understand"
```

```
type.of.summary = "mean"

switch(type.of.summary,
       mean=mean(x.vec),
       median=median(x.vec),
       histogram=hist(x.vec),
       "I don't understand")
```

```
## [1] -0.1637393
```

- Here we are expecting `type.of.summary` to be a string, either "mean", "median", or "histogram"; we specify what to do for each
- The last passed argument has no name, and it serves as the `else` clause
- Try changing `type.of.summary` above and see what happens

## Reminder: Boolean operators

Remember our standard Boolean operators, `&` and `|`. These combine terms elementwise

```
u.vec = runif(10, -1, 1)
u.vec
```

```
##  [1]  0.51741668 -0.47789847 -0.01203616  0.58579639 -0.1
##  [7]  0.25153693 -0.72761362  0.09736555  0.84366335
```

```
u.vec[-0.5 <= u.vec & u.vec <= 0.5] = 999
u.vec
```

```
##  [1]   0.5174167 999.0000000 999.0000000   0.5857964 999.
##  [7] 999.0000000  -0.7276136 999.0000000   0.8436634
```

## Lazy Boolean operators (效率更高，建议在 control flow 中使用)

In contrast to the standard Boolean operators, `&&` and `||` give just a single Boolean, "lazily": meaning we terminate evaluating the expression

ASAP → ① 短路原则 ② c(TRUE, FALSE)&& 1 等价于 TRUE & 1；c(TRUE, FALSE)& 1 等价于 c(TRUE, FALSE)& c(1,1)

```
(1 > 0) | (ThisVariableIsNotDefined == 0)
```

not lazy　也会被执行(并报错)

```
## Error in eval(expr, envir, enclos): object 'ThisVariableI
```

```
# Evaluate (1 > 0) and (ThisVariableIsNotDefined == 0) first
(1 > 0) || (ThisVariableIsNotDefined == 0)
```

not lazy　也会被执行(并报错)

lazy　不会被执行(因为(1>0)已经确定了结果)

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

```
# If (1>0) is TRUE, terminate

(1 > 0) && (ThisVariableIsNotDefined == 0)
```

lazy　也会被执行(并报错)(因为(1>0)还不能确定结果)

```
## Error in eval(expr, envir, enclos): object 'ThisVariableI
```

```
# If (1>0) is TRUE, then evaluate (ThisVariableIsNotDefined

(0 > 0) & all(matrix(0,2,2) == matrix(0,3,3))
```

not lazy　也会被执行(并报错)

```
## Error in matrix(0, 2, 2) == matrix(0, 3, 3): non-conforma
```

```
(0 > 0) && all(matrix(0,2,2) == matrix(0,3,3))
```

lazy　不会被执行(因为(0>0)已经确定了结果)

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

```
# Terminate if (0 > 0) is FALSE
```

- In control flow, we typically just want one Boolean
- Rule of thumb: use `&` and `|` for indexing or subsetting, and `&&` and `||` for conditionals

# Part III: Iteration

## Iteration

### Iteration is at the heart of programming

- Computers: good at applying rigid rules over and over again.
- Humans: not so good at this.

Summary of the iteration methods in R:

- `for()`, `while()` loops: standard loop constructs
- Vectorization: use it whenever possible! Often faster and simpler
- The apply family of functions: alternative to `for()` loop, these are base R functions
- The map family of functions: another alternative, very useful, from the `purrr` package

## `for()`

A `for()` loop increments a **counter** variable along a vector. It repeatedly runs a code block, called the **body** of the loop, with the counter set at its current value, until it runs through the vector

```
n = 10
log.vec = rep(0,n)
for (i in 1:n) {
  log.vec[i] = log(i)
}
log.vec
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.
##  [8] 2.0794415 2.1972246 2.3025851
```

Here `i` is the counter and the vector we are iterating over is `1:n`. The body is the code in between the braces

## Breaking from the loop

We can **break** out of a `for()` loop early (before the counter has been iterated over the whole vector), using `break`

```
n = 10
log.vec = rep(NA,n) # NA indicates empty space
for (i in 1:n) {
  if (log(i) > 2) {
    cat("I'm outta here. I don't like numbers bigger than 2\
    break
  }
  log.vec[i] = log(i)
}
```

```
## I'm outta here. I don't like numbers bigger than 2
```

```
log.vec
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.
##  [8]        NA        NA        NA
```

## Variations on standard `for()` loops

Many different variations on standard `for()` are possible. Two common ones:

- Nonnumeric counters: counter variable always gets iterated over a vector, but it doesn't have to be numeric
- Nested loops: body of the `for()` loop can contain another `for()` loop (or several others)

```
for (str in c("A", "B", "C")) {
  cat(paste(str, "declined to comment\n"))
}
```

```
## A declined to comment
## B declined to comment
## C declined to comment
```

```
for (i in 1:4) {
  for (j in 1:i^2) {
    cat(paste(j,""))
  }
  cat("\n")
}
```

```
## 1
## 1 2 3 4
## 1 2 3 4 5 6 7 8 9
## 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

## while()

A `while()` loop repeatedly runs a code block, again called the **body**, until some condition is no longer true

```
i = 1
log.vec = c()
while (log(i) <= 2) {
  log.vec = c(log.vec, log(i))
  i = i+1
}
log.vec
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7
```

## for() versus while()

- `for()` is better when the number of times to repeat (values to iterate over) is clear in advance
- `while()` is better when you can recognize when to stop once you're there, even if you can't guess it to begin with
- `while()` is more general, in that every `for()` could be replaced with a `while()` (but not vice versa)

## while(TRUE) or repeat

`while(TRUE)` and `repeat`: both do the same thing, just repeat the body indefinitely, until something causes the flow to break. Example (try running in your console):

```
repeat {
  ans = readline("What is your favourite statistics course i
  if (ans == "STA3005" || ans == "Statistical Computing") {
    cat("Yes! You get an 'A'.")
    break
  }
  else {
    cat("Wrong answer!\n")
  }
}
```

## Avoiding explicit iteration

Vectorization means to operate on all elements of a vector without needing loops. It makes code more concise, easy to read and less error prone.

```
log.vec <- log(1:n) # equivalent to the for loop
log.vec
log.vec <- log.vec[log.vec < 2] # equivalent to the while lo
log.vec
```

- Warning: some people have a tendency to **overuse** `for()` and `while()` loops in R
- They aren't always needed. Remember **vectorization** should be used whenever possible
- You will find more examples about vectorization in the assignment, and try to hit upon it throughout the course

## Summary

- Three ways to index vectors, matrices, data frames, lists: integers, Booleans, names
- Boolean on-the-fly indexing can be very useful
- Named indexing will be especially useful for data frames

- Indexing lists can be a bit tricky (beware of the difference between `[ ]` and `[[ ]]`)
- `if()`, `elseif()`, `else`: standard conditionals
- `ifelse()`: shortcut for using `if()` and `else` in combination
- `switch()`: shortcut for using `if()`, `elseif()`, and `else` in combination
- `for()`, `while()`, `repeat`: standard loop constructs
- Don't overuse explicit `for()` loops, vectorization is your friend!
- `apply()` and `**ply()`: can also be very useful (we'll see them later)