Elements of a programming language 1

Marcin Kierczak with Thomas Källman (labs)

21 September 2016

Introduction

Today, we will talk about various elements of a programming language and see how they are realized in R.

Contents of the lecture

- **variables and their types**
- **operators**
- **vectors**
- **numbers as vectors**
- **strings as vectors**
- matrices
- lists
- data frames
- objects
- repeating actions: iteration and recursion
- decision taking: control structures
- functions in general
- variable scope
- core functions

Variables

Creating a variable is nothing more than assigning a name to data...

```
7 + 9
```

```
## [1] 16
```

```
a <- 7
a
```

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

```
b <- 9
b
```

```
## [1] 9
```

```
c <- a + b
c
```

Variables cted.

We are not constrained to numbers. . .

```
text1 <- 'a'
text2 <- 'qwerty'
text1
```

```
## [1] "a"
```

```
text2
```

```
## [1] "qwerty"
```

Variables – naming conventions

How to write variable names?

- What is legal/valid?
- What is a good style?

A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number.

Names such as ".2way" are not valid, and neither are the so-called *reserved words*.

Reserved words

Reserved words, are:

`if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_`

and you also **cannot** use: `c`, `q`, `t`, `C`, `D`, `I`

and you **should not** use: `T`, `F`

Variables – good style

- make them informative, e.g. `genotypes` instead of `fsjht45jkhsdf4`,
- use consistent notation across your code – the same *naming convention*,
- camelNotation vs. dot.notation vs. dash_notation
- I used to use the camelNotation and the dot.notation and I'm still hesitating :-),
- do not `give.them.too.long.names`,
- in the dot notation avoid `my.variable.2`, use `my.variable2` instead,
- there are certain customary names: `tmp` - for temporary variables; `cnt` for counters; `i,j,k` within loops, `pwd` - for password. . .

Variables have types

We have already discussed the system of types in general. Now, time to look at the types system in R.

A numeric that stores numbers of different *types*:

```
x = 41.99 # assign 41.99 to x
class(x)
```

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

```
mode(x) # representation
```

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

```
typeof(x)
```

```
## [1] "double"
```

Class, type, representation and soorage mode

1. *class* is the point of view of object-oriented programming in R.

```
x <- 1:3
class(x)
```

```
## [1] "integer"
```

any generic function that has an “integer” method can be used.

2. `typeof()` gives the “type” of object from R’s point of view.
3. `mode()` gives the “type” of object from the point of view of the S language.
4. `storage.mode()` is useful when passing R objects to compiled code, e.g. C.

Variables have types cted.

```r
y = 12 # now assign an integer value to y
class(y) # still numeric
```

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

```r
typeof(y) # an integer, but still a double!
```

```
## [1] "double"
```

Even integers are stored as double by default.
Numeric == double == real.

Variables have types cted.

```r
x <- as.integer(x) # type conversion, casting
typeof(x)
```

```
## [1] "integer"
```

```r
class(x)
```

```
## [1] "integer"
```

```r
is.integer(x)
```

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

One rarely works explicitly with integers though. . .

Be careful when casting

```
pi <- 3.1415926536 # assign approximation of pi to pi
pi
```

```
## [1] 3.141593
```

```
pi <- as.integer(pi) # not-so-careful casting
pi
```

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

```
pi <- as.double(pi) # trying to rescue the situation
pi
```

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

Casting is not rounding

```r
as.integer(3.14)
```

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

```r
as.integer(3.51)
```

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

Ceiling, floor and a round corner

```
floor(3.51) # floor of 3.51
```

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

```
ceiling(3.51) # ceiling of 3.51
```

```
## [1] 4
```

```
round(3.51, digits = 1) # round to one decimal
```

```
## [1] 3.5
```

What happens if we cast a string to a number

```r
as.numeric('4.5678')
```

```
## [1] 4.5678
```

```r
as.double('4.5678')
```

```
## [1] 4.5678
```

```r
as.numeric('R course is cool!')
```

```
## Warning: NAs introduced by coercion
```

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

Special values

```
-1/0   # Minus infinity
```

```
## [1] -Inf
```

```
1/0 # Infinity
```

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

Special values cted.

```r
112345^67890  # Also infinity for R
```

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

```r
1/2e78996543  # Zero for R
```

```
## [1] 0
```

```r
Inf - Inf # Not a Number
```

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

Complex number type

Core R supports complex numbers.

```
z = 7 + 4i # create a complex number
z
```

```
## [1] 7+4i
```

```
class(z)
```

```
## [1] "complex"
```

```
typeof(z)
```

```
## [1] "complex"
```

```
is.complex(z)
```

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

Complex number type cted.

```r
sqrt(-1) # not treated as cplx number
```

```
## Warning in sqrt(-1): NaNs produced

## [1] NaN
```

```r
sqrt(-1 + 0i) # now a proper cplx number
```

```
## [1] 0+1i
```

```r
sqrt(as.complex(-1)) # an alternative way
```

```
## [1] 0+1i
```

Logical type

```r
a <- 7 > 2
b <- 2 >= 7
a
```

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

```r
b
```

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

```r
class(a)
```

```
## [1] "logical"
```

```r
typeof(a)
```

```
## [1] "logical"
```

R has three logical values: TRUE, FALSE and NA.

```
x <- c(NA, FALSE, TRUE)
names(x) <- as.character(x)
outer(x, x, "&") # AND table
```

```
##        <NA> FALSE  TRUE
## <NA>     NA FALSE    NA
## FALSE FALSE FALSE FALSE
## TRUE     NA FALSE  TRUE
```

Logical type cted.

```r
x <- TRUE
x
```

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

```r
x <- T # also valid
x
```

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

```r
is.logical(x)
```

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

```r
typeof(x)
```

```
## [1] "logical"
```

Logical as number

It is **very important** to remember that **logical type is also a numeric**!

```
x <- TRUE
y <- FALSE
x + y
```

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

```
2 * x
```

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

```
x * y
```

```
## [1] 0
```

A trap set for you

Never ever use variable names as T or F. Why?

```
F <- T
T
```

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

```
F
```

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

Maybe applicable in politics, but not really in science. . .

Character type

It is easy to work with characters and strings:

```r
character <- 'c'
text <- 'This is my first sentence in R.'
text
```

```
## [1] "This is my first sentence in R."
```

```r
character
```

```
## [1] "c"
```

```r
class(character)
```

```
## [1] "character"
```

```r
typeof(text) # also of 'character' type
```

Character type

```r
number <- 3.14
number.text <- as.character(number) # cast to char
number.text
```

```
## [1] "3.14"
```

```r
class(number.text)
```

```
## [1] "character"
```

```r
as.numeric(number.text) # and the other way round
```

```
## [1] 3.14
```

Basic string operations

```r
text1 <- "John had a yellow "
text2 <- "submarine"
result <- paste(text1, text2, ".", sep='')
result
```

```
## [1] "John had a yellow submarine."
```

```r
sub("submarine", "cab", result)
```

```
## [1] "John had a yellow cab."
```

```r
substr(result, start = 1, stop = 5)
```

```
## [1] "John "
```

Basic printing

```r
txt <- "blue"
val <- 345.78
sprintf("The weight of a %s ball is  %g g", txt, val)
```

```
## [1] "The weight of a blue ball is  345.78 g"
```

Complex data structures

Using the previously discussed basic data types (numeric, integer, logical and character) one can construct more complex data structures:

- vectors
- matrices
- arrays
- factors
- lists

Atomic vectors

An *atomic vector*, or simply a *vector* is a one dimensional data structure (a sequence) of elements of the same data type. Elements of a vector are oficiallly called *components*, but we will just call them *elements*.

We construct vectors using core function c() (construct).

```
vec <- c(1,2,5,7,9,27,45.5)
vec
```

```
## [1]  1.0  2.0  5.0  7.0  9.0 27.0 45.5
```

In R, even a single number is a one-element vector. You have to get used to think in terms of vectors...

Atomic vectors cted.

You can also create empty/zero vectors of a given type and length:

```r
vector('integer', 5) # a vector of 5 integers
```

```
## [1] 0 0 0 0 0
```

```r
vector('character', 5)
```

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

```r
character(5) # does the same
```

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

```r
logical(5) # same as vector('logical', 5)
```

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

Combining two or more vectors

Vectors can easily be combined:

```
v1 <- c(1,3,5,7.56)
v2 <- c('a','b','c')
v3 <- c(0.1, 0.2, 3.1415)
c(v1, v2, v3)
```

```
##  [1] "1"      "3"      "5"      "7.56"   "a"      "b"
##  [8] "0.1"    "0.2"    "3.1415"
```

Please note that after combining vectors, all elements became character. It is called a *coercion*.

Basic vector arithmetics

```
v1 <- c(1, 2, 3, 4)
v2 <- c(7, -9, 15.2, 4)
v1 + v2 # addition
```

```
## [1]  8.0 -7.0 18.2  8.0
```

```
v1 - v2 # subtraction
```

```
## [1]  -6.0  11.0 -12.2   0.0
```

```
v1 * v2 # scalar multiplication
```

```
## [1]   7.0 -18.0  45.6  16.0
```

```
v1 / v2 # division
```

```
## [1]  0.1428571 -0.2222222  0.1973684  1.0000000
```

Vectors – recycling rule

```r
v1 <- c(1, 2, 3, 4, 5)
v2 <- c(1, 2)
v1 + v2
```

```
## Warning in v1 + v2: longer object length is not a multi
## object length
```

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

Values in the shorter vector will be **recycled** to match the length of the longer one: v2 <- c(1, 2, 1, 2, 1)

Vectors – indexing

We can access or retrieve particular elements of a vector by using the [] notation:

```
vec <- c('a', 'b', 'c', 'd', 'e')
vec[1] # the first element
```

```
## [1] "a"
```

```
vec[5] # the fifth element
```

```
## [1] "e"
```

```
vec[-1] # take the last element
```

```
## [1] "b" "c" "d" "e"
```

Vectors – indexing cted.

And what happens if we want to retrieve elements outside the vector?

```
vec[0] # R counts elements from 1
```

```
## character(0)
```

```
vec[78] # Index past the length of the vector
```

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

Note, if you ask for an element with index lower than the index of the first element, you will het an empty vector of the sme type as the original vector. If you ask for an element beyond the vector's length, you get an NA value.

Vectors – indexing cted.

You can also retrieve elements of a vector using a vector of indices:

```
vec <- c('a', 'b', 'c', 'd', 'e')
vec.ind <- c(1,3,5)
vec[vec.ind]
```

```
## [1] "a" "c" "e"
```

Or even a logical vector:

```
vec <- c('a', 'b', 'c', 'd', 'e')
vec.ind <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
vec[vec.ind]
```

```
## [1] "a" "c" "e"
```

Vectors – indexing using names

You can name elements of your vector:

```
vec <- c(23.7, 54.5, 22.7)
names(vec) # by default there are no names
```

```
## NULL
```

```
names(vec) <- c('sample1', 'sample2', 'sample3')
vec[c('sample2', 'sample1')]
```

```
## sample2 sample1
##    54.5    23.7
```

Vectors – removing elements

You can return a vector without certain elements:

```r
vec <- c(1, 2, 3, 4, 5)
vec[-5] # without the 5-th element
```

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

```r
vec[-(c(1,3,5))] # withoutelements 1, 3, 5
```

```
## [1] 2 4
```

Vectors indexing – conditions

Also logical expressions are allowed in indexing:

```
vec <- c(1, 2, 3, 4, 5)
vec < 3 # we can use the value of this logical comparison
```

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

```
vec[vec < 3]# Et voila!
```

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

Vectors – more operations

You can easily reverse a vector:

```
vec <- c(1, 2, 3, 4, 5)
rev(vec)
```

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

You can generate vectors of subsequent numbers using ':', e.g.:

```
v <- c(5:7)
v
```

```
## [1] 5 6 7
```

```
v2 <- c(3:-4)
v2
```

```
## [1]  3  2  1  0 -1 -2 -3 -4
```

Vectors – size

To get the size of a vector, use *length()*:

```
vec <- c(1:78)
length(vec)
```

```
## [1] 78
```

Vectors – substitute element

To substitute an element in a vector simply:

```
vec <- c(1:5)
vec
```

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

```
vec[3] <- 'a' # Note the coercion!
vec
```

```
## [1] "1" "2" "a" "4" "5"
```

To insert ‘a’ at, say, the 2nd position:

```
c(vec[1], 'a', vec[2:length(vec)])
```

```
## [1] "1" "a" "2" "a" "4" "5"
```

Vectors – changing the length

What if we write past the vectors last element?

```
vec <- c(1:5)
vec
```

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

```
vec[9] <- 9
vec
```

```
## [1]  1  2  3  4  5 NA NA NA  9
```

Vectors – counting values

One may be interested in the count of particular values:

```
vec <- c(1:5, 1:4, 1:3) # a vector with repeating values
table(vec) # table of counts
```

```
## vec
## 1 2 3 4 5
## 3 3 3 2 1
```

```
tab <- table(vec)/length(vec) # table of freqs.
round(tab, digits=3) # and let's round it
```

```
## vec
##     1     2     3     4     5
## 0.250 0.250 0.250 0.167 0.083
```

Vectors – sorting

To sort values of a vector:

```
vec <- c(1:5, NA, NA, 1:3)
sort(vec) # oops, NAs got lost
```

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

```
sort(vec, na.last = TRUE)
```

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

```
sort(vec, decreasing = TRUE) # in a decreasing order
```

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

Sequences of numbers

R provides also a few handy functions to generate sequences of numbers:

```
c(1:5, 7:10) # the ':' operator
```

```
## [1]  1  2  3  4  5  7  8  9 10
```

```
(seq1 <- seq(from=1, to=10, by=2))
```

```
## [1] 1 3 5 7 9
```

```
(seq2 <- seq(from=11, along.with = seq1))
```

```
## [1] 11 12 13 14 15
```

```
seq(from=10, to=1, by=-2)
```

```
## [1] 10  8  6  4  2
```

A detour – printing with ()

Note what we did here, if you enclose the expression in (), the result of assignment will be also printed:

```
seq1 <- seq(from=1, to=5)
seq1 # has to be printed explicitely
```

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

```
(seq2 <- seq(from=5, to=1)) # will print automatically
```

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

Back to sequences

One may also wish to repeat certain value or a vector n times:

```r
rep('a', times=5)
```

```
## [1] "a" "a" "a" "a" "a"
```

```r
rep(1:5, times=3)
```

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

```r
rep(seq(from=1, to=3, by=2), times=2)
```

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

Sequences of random numbers

There is also a really useful function **sample** that helps with generating sequences of random numbers:

```
# simulate casting a fair dice 10x
sample(x = c(1:6), size=10, replace = T)
```

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

```
# make it unfair, it is loaded on '3'
myprobs = rep(0.15, times=6)
myprobs[3] <- 0.25 # a bit higher probability for '3'
sample(x = c(1:6), size=10, replace = T, prob=myprobs)
```

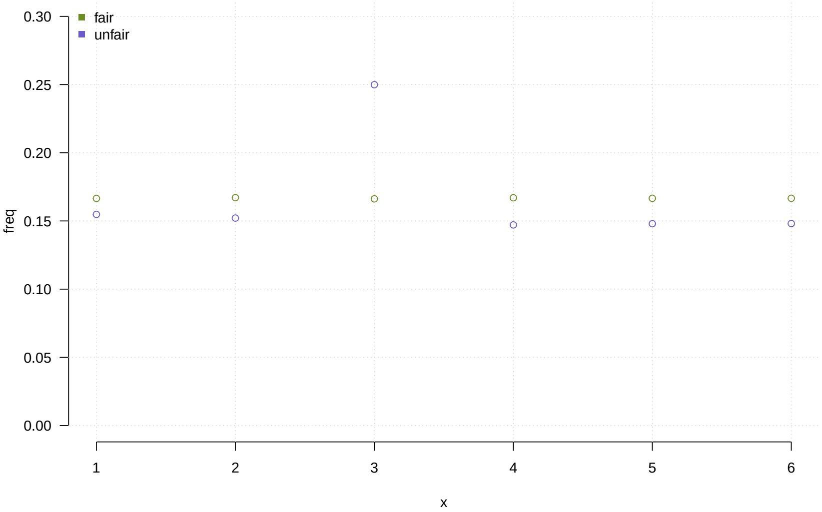
```
##  [1] 3 4 4 3 4 5 2 3 5 6
```

Fair vs. loaded dice

Now, let us see how this can be useful. We need more than 10 results. Let's cast our dices 10,000 times and plot the freq. distribution.

```
# simulate casting a fair dice 10x
fair <- sample(x = c(1:6), size=10e3, replace = T)
unfair <- sample(x = c(1:6), size=10e3, replace = T,
                    prob=myprobs)
```

Fair vs. loaded dice – the result



Sample – one more use

The sample function has one more interesting feature, it can be used to randomize order of already created vectors:

```
mychars <- c('a', 'b', 'c', 'd', 'e', 'f')
mychars
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
sample(mychars)
```

```
## [1] "d" "b" "a" "c" "e" "f"
```

```
sample(mychars)
```

```
## [1] "b" "a" "d" "f" "e" "c"
```

Vectors/sequences – more advanced operations

```r
v1 <- sample(1:5, size = 4)
v1
```

```
## [1] 2 3 5 1
```

```r
max(v1) # max value of the vector
```

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

```r
min(v1) # min value
```

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

```r
sum(v1) # sum all the elements
```

```
## [1] 11
```

Vectors/sequences – more advanced operations 2

```r
v1
```

```
## [1] 2 3 5 1
```

```r
diff(v1) # diff. of element pairs
```

```
## [1]  1  2 -4
```

```r
cumsum(v1) # cumulative sum
```

```
## [1]  2  5 10 11
```

```r
prod(v1) # product of all elements
```

```
## [1] 30
```

Vectors/sequences – more advanced operations 3

```r
v1
```

```
## [1] 2 3 5 1
```

```r
cumprod(v1) # cumulative product
```

```
## [1]  2  6 30 30
```

```r
cummin(v1) # minimum so far (up to i-th el.)
```

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

```r
cummax(v1) # maximum up to i-th element
```

```
## [1] 2 3 5 5
```

Vectors/sequences – pairwise comparisons

```r
v1
```

```
## [1] 2 3 5 1
```

```r
v2
```

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

```r
v1 <= v2 # direct comparison
```

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

```r
pmin(v1, v2) # pairwise min
```

```
## [1] 2 1 3 1
```

```r
pmax(v1, v2) # pairwise max
```

Vectors/sequences – rank() and order()

rank() and order() are a pair of inverse functions.

```
v1 <- c(1, 3, 4, 5, 3, 2)
rank(v1) # show rank of each value (min has rank 1)
```

```
## [1] 1.0 3.5 5.0 6.0 3.5 2.0
```

```
order(v1) # order of indices for a sorted vector
```

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

```
v1[order(v1)]
```

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

```
sort(v1)
```

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

Factors

To work with **nominal** values, R offers a special data type, a *factor*:

```
vec <- c('giraffe', 'donkey', 'liger',
         'liger', 'giraffe', 'liger')
vec.f <- factor(vec)
summary(vec.f)
```

```
##   donkey giraffe   liger
##        1       2       3
```

So donkey is coded as 1, giraffe as 2 and liger as 3. Coding is alphabetical.

```
as.numeric(vec.f)
```

```
## [1] 2 1 3 3 2 3
```

Factors

You can also control the coding/mapping:

```
vec <- c('giraffe', 'donkey', 'liger',
         'liger', 'giraffe', 'liger')
vec.f <- factor(vec, levels=c('donkey', 'giraffe',
                              'liger'),
                labels=c('zonkey','Sophie','tigon'))
summary(vec.f)
```

```
## zonkey Sophie  tigon
##      1      2      3
```

A bit confusing, factors. . .

Ordered

To work with ordinal scale (ordered) variables, one can also use factors:

```
vec <- c('tiny', 'small', 'medium', 'large')
factor(vec) # rearranged alphabetically
```

```
## [1] tiny   small  medium large
## Levels: large medium small tiny
```

```
factor(vec, ordered=T) # order as provided
```

```
## [1] tiny   small  medium large
## Levels: large < medium < small < tiny
```

Matrices

We will talk about matrices in the next lecture. See you!