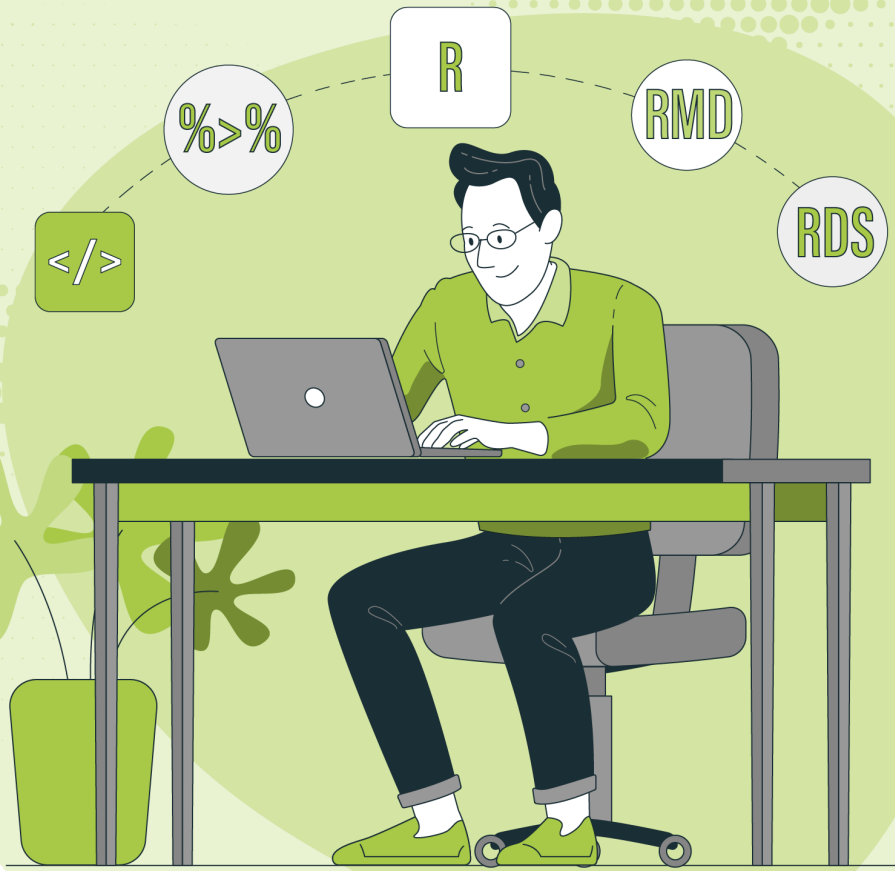


Introduction To Programming in R (2)

R Foundations for Life Scientists

Marcin Kierczak, Sebastian DiLorenzo

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

Complex data structures

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

dim	Homogenous	Heterogenous
0d	n/a	n/a
1d	vectors	list
2d	matrices	data frame
nd	arrays	n/a

- factors – special type

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 oficially 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:

```
vector('integer', 5) # a vector of 5 integers
vector('character', 5)
character(5) # does the same
logical(5) # same as vector('logical', 5)
```

```
## [1] 0 0 0 0 0
## [1] "" "" "" "" ""
## [1] "" "" "" "" ""
## [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"      "c"      "0.1"
##  [9] "0.2"    "3.1415"
```

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

Basic vector arithmetics

```r
v1 <- c(1, 2, 3, 4)
v2 <- c(7, -9, 15.2, 4)
v1 + v2 # addition
v1 - v2 # subtraction
v1 * v2 # scalar multiplication
v1 / v2 # division
```

```
## [1]  8.0 -7.0 18.2  8.0
## [1]  -6.0  11.0 -12.2   0.0
## [1]   7.0 -18.0  45.6  16.0
## [1]  0.1428571 -0.2222222  0.1973684  1.0000000
```

Vectors – recycling rule

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

```
## [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
vec[5] # the fifth element
vec[-1] # remove the first element
```

```
## [1] "a"
## [1] "e"
## [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
vec[78] # Index past the length of the vector
```

```
## character(0)
## [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
names(vec) <- c('sample1', 'sample2', 'sample3')
vec[c('sample2', 'sample1')]
```

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

Vectors – removing elements

You can return a vector without certain elements:

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

```
## [1] 1 2 3 4
## [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
vec[vec < 3]# Et voila!
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE
## [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
v2 <- c(3:-4)
v2
```

```
## [1] 5 6 7
## [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
vec[3] <- 'a' # Note the coercion!
vec
```

```
## [1] 1 2 3 4 5
## [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
vec[9] <- 9
vec
```

```
## [1] 1 2 3 4 5
## [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
tab <- table(vec)/length(vec) # table of freqs.
round(tab, digits=3) # and let's round it
```

```
## vec
## 1 2 3 4 5
## 3 3 3 2 1
## 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
sort(vec, na.last = TRUE)
sort(vec, decreasing = TRUE) # in a decreasing order
```

```
## [1] 1 1 2 2 3 3 4 5
##  [1]  1  1  2  2  3  3  4  5 NA NA
## [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
(seq1 <- seq(from=1, to=10, by=2))
(seq2 <- seq(from=11, along.with = seq1))
seq(from=10, to=1, by=-2)
```

```
## [1]  1  2  3  4  5  7  8  9 10
## [1] 1 3 5 7 9
## [1] 11 12 13 14 15
## [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 explicitly
```

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

while:

```
(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:

```
rep('a', times=5)
rep(1:5, times=3)
rep(seq(from=1, to=3, by=2), times=2)
```

```
## [1] "a" "a" "a" "a" "a"
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
## [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)
# 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] 6 1 3 6 2 1 2 1 3 4
##  [1] 2 4 1 6 2 3 3 4 1 4
```

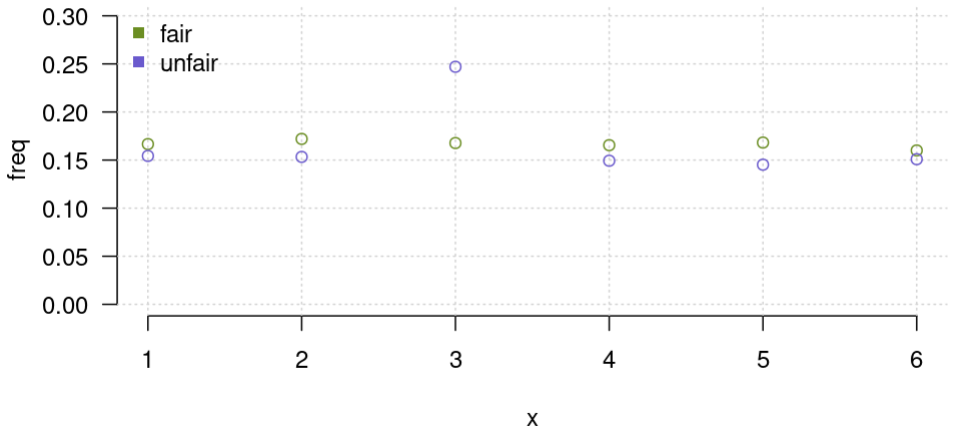
Fair vs. loaded dice

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

```
t1 <- table(fair)/length(fair)
t2 <- table(unfair)/length(unfair)
plot(0,0,type="n",xlim=c(1,6.0),ylim=c(0,.3),xlab="x",ylab="freq",bty='n', las=1
grid()
points(1:6, t1, col="olivedrab")
points(1:6, t2, col="slateblue")
legend('topleft', legend = c('fair','unfair'), col = c('olivedrab', 'slateblue')
```



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
sample(mychars)
sample(mychars)
```

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

Vectors/sequences – more advanced operations

```r
v1 <- sample(1:5, size = 4)
v1
max(v1) # max value of the vector
min(v1) # min value
sum(v1) # sum all the elements
```

```
## [1] 4 3 2 1
## [1] 4
## [1] 1
## [1] 10
```

Vectors/sequences – more advanced operations 2

```
v1
diff(v1) # diff. of element pairs
cumsum(v1) # cumulative sum
prod(v1) # product of all elements
```

```
## [1] 4 3 2 1
## [1] -1 -1 -1
## [1]  4  7  9 10
## [1] 24
```

Vectors/sequences – more advanced operations 3

```
v1
cumprod(v1) # cumulative product
cummin(v1) # minimum so far (up to i-th el.)
cummax(v1) # maximum up to i-th element
```

```
## [1] 4 3 2 1
## [1]  4 12 24 24
## [1] 4 3 2 1
## [1] 4 4 4 4
```

Vectors/sequences – pairwise comparisons

```
v2 <- sample(1:5, size=4)
```

```
v1
v2
v1 <= v2 # direct comparison
pmin(v1, v2) # pairwise min
pmax(v1, v2) # pairwise max
```

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

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

```
## [1] 1.0 3.5 5.0 6.0 3.5 2.0
## [1] 1 6 2 5 3 4
## [1] 1 2 3 3 4 5
## [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...

We will talk about matrices in the next lecture!

Graphics from freepik.com

Created: 09-Aug-2023 • Roy Francis • SciLifeLab • NBIS