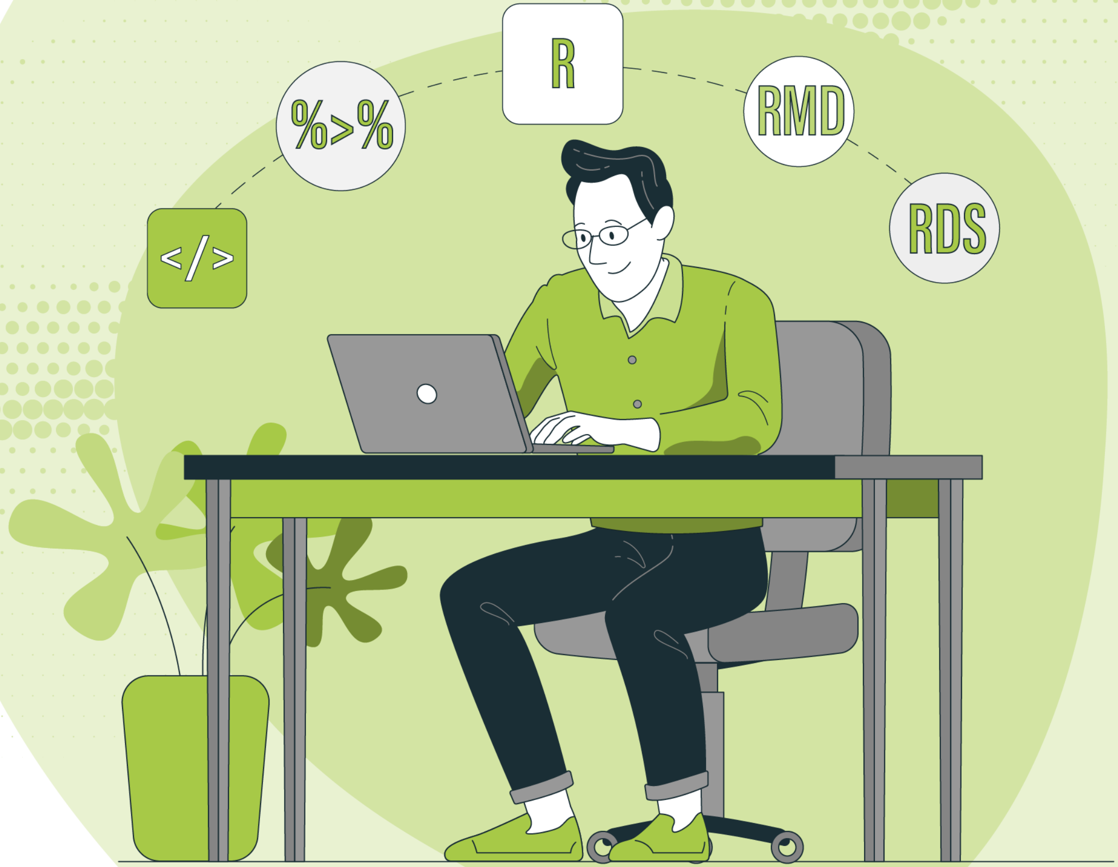


Replication, Control Structures & Functions

Elements of the R language

Marcin Kierczak

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**
- **base functions**

Repeating actions

In several algorithms, the point is to repeat certain action several times. In the language of mathematical formulas, we have for instance the following signs for repeating an action:

$$\Sigma_{i=1}^{n}(expression)$$

which denotes addition over elements $1 \ldots n$ or

$$\Pi_{i=1}^{n}(expression)$$

which denotes multiplication of elements $1 \ldots n$.

It is important to learn how to translate these (and similar) formulas into the R language.

Repeating actions — for loop

One way to repeat an action is to use the **for-loop**

```
for (i in 1:5) {
  cat(paste('Performing operation no.', i), '\n')
}
```

```
## Performing operation no. 1
## Performing operation no. 2
## Performing operation no. 3
## Performing operation no. 4
## Performing operation no. 5
```

A slight modification of the above example will skip odd indices.

```
for (i in c(2,4,6,8,10)) {
  cat(paste('Performing operation no.', i), '\n')
}
```

```
## Performing operation no. 2
## Performing operation no. 4
## Performing operation no. 6
## Performing operation no. 8
## Performing operation no. 10
```

Repeating actions — for loop with a counter

Sometimes, we also want an external counter:

```r
cnt <- 1
for (i in c(2,4,6,8,10)) {
  cat(paste('Performing operation no.', cnt,
            'on element', i), '\n')
  cnt <- cnt + 1
}
```

```
## Performing operation no. 1 on element 2
## Performing operation no. 2 on element 4
## Performing operation no. 3 on element 6
## Performing operation no. 4 on element 8
## Performing operation no. 5 on element 10
```

Repeating actions — for loop, an example

Say, we want to add 1 to every element of a vector:

```
vec <- c(1:5)
vec
for (i in vec) {
  vec[i] <- vec[i] + 1
}
vec
```

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

The above can be achieved in R by means of **vectorization**:

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

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

Repeating actions — vectorization

Let us compare the time of execution of the vectorized version (vector with 10,000 elements):

```
vec <- c(1:1e6)
ptm <- proc.time()
vec <- vec + 1
proc.time() - ptm # vectorized
```

```
##    user  system elapsed
##   0.004   0.000   0.004
```

to the loop version:

```
vec <- c(1:1e6)
ptm <- proc.time()
for (i in vec) {
  vec[i] <- vec[i] + 1
}
proc.time() - ptm # for-loop
```

```
##    user  system elapsed
##   0.070   0.000   0.069
```

Repeating actions — the while loop

There is also another type of loop in R, the **while loop** which is executed as long as some condition is true.

```
x <- 1
while (x < 5) {
  cat(x, " ... ")
  x <- x + 1
}
```

```
## 1  ... 2  ... 3  ... 4  ...
```

Recursion

When we explicitely repeat an action using a loop, we talk about **iteration**. We can also repeat actions by means of **recursion**, i.e. when a function calls itself. Let us implement a factorial $!$:

```
factorial.rec <- function(x) {
  if (x == 0 || x == 1)
    return(1)
  else
    return(x * factorial.rec(x - 1)) # Recursive call!
}
factorial.rec(5)
```

```
## [1] 120
```

Loops — avoid growing data

Avoid changing dimensions of an object inside the loop:

```
v <- c() # Initialize
for (i in 1:100) {
  v <- c(v, i)
}
```

It is much better to do it like this:

```
v <- rep(NA, 100) # Initialize with length
for (i in 1:100) {
  v[i] <- i
}
```

Always try to know the size of the object you are going to create!

To R or not to Python? — taking decisions, an if-clause

Often, one has to take a different course of action depending on a flow of the algorithm. You have already seen the **if-else** block. Let's print only odd numbers $[1, 10]$:

```
v <- 1:10
for (i in v) {
  if (i %% 2 != 0) { # if clause
    cat(i, ' ')
  }
}
```

```
## 1  3  5  7  9
```

Decisions — if-else

If we want to print 'o' for an odd number and 'e' for an even, we could write either of:

```
v <- 1:10
for (i in v) {
  if (i %% 2 != 0) { # if clause
    cat('o ')
  }
  if (i %% 2 == 0) { # another if-clause
    cat('e ')
  }
}
```

```
## o e o e o e o e o e
```

```
v <- 1:10
for (i in v) {
  if (i %% 2 != 0) { # if clause
    cat('o ')
  } else { # another if-clause
    cat('e ')
  }
}
```

```
## o e o e o e o e o e
```

```
v <- 1:10
for (i in v) {
  tmp <- 'e ' # set default to even
  if (i %% 2 != 0) { # if clause
    tmp <- 'o ' # change default for odd num
  }
  cat(tmp)
}
```

```
## o e o e o e o e o e
```

Each of these three ways are equally good and are mainly the matter of style...

Decision taking — more alternatives

So far, so good, but we were only dealing with 3 alternatives. Let's say that we want to print '?' for zero, 'e' for even and 'o' for an odd number:

```
v <- c(0:10)
for (i in v) {
  if (i == 0) {
    cat('? ')
  } else if (i %% 2 != 0) { # if clause
    cat('o ')
  } else { # another if-clause
    cat('e ')
  }
}
```

```
## ? o e o e o e o e o e
```

Congratulations! You have just learned the **if-elseif-else** clause.

Switch

If-else clauses operate on logical values. What if we want to take decisions based on non-logical values? Well, if-else will still work by evaluating a number of comparisons, but we can also use **switch**:

```
switch.demo <- function(x) {
  switch(class(x),
         logical = ,
         numeric = cat('Numeric or logical.'),
         factor = cat('Factor.'),
         cat('Undefined')
         )
}
switch.demo(x=TRUE)
switch.demo(x=15)
switch.demo(x=factor('a'))
switch.demo(data.frame())
```

```
## Numeric or logical.Numeric or logical.Factor.Undefined
```

Functions

Often, it is really handy to re-use some code we have written or to pack together the code that is doing some task. Functions are a really good way to do this in R:

```
add.one <- function(arg1) {
  arg1 <- arg1 + 1
  return(arg1)
}
add.one(1)
add.one()
```

```
## Error in add.one(): argument "arg1" is missing, with no default
```

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

Anatomy of a function

A function consists of: *formal arguments*, *function body* and *environment*:

```
formals(ecdf)
body(plot.ecdf)
environment(ecdf)
```

```
## $x
##
##
## {
##     plot.stepfun(x, ..., ylab = ylab, verticals = verticals,
##         pch = pch)
##     abline(h = c(0, 1), col = col.01line, lty = 2)
## }
## <environment: namespace:stats>
```

Functions — default values

Sometimes, it is good to use default values for some arguments:

```
add.a.num <- function(arg, num=1) {
  arg <- arg + num
  return(arg)
}
add.a.num(1, 5)
add.a.num(1) # skip the num argument
add.a.num(num=1) # skip the first argument
```

```
## Error in add.a.num(num = 1): argument "arg" is missing, with no default
```

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

Functions — order of arguments

```
args.demo <- function(x, y, arg3) {
  print(paste('x =', x, 'y =', y, 'arg3 =', arg3))
}
args.demo(1,2,3)
args.demo(x=1, y=2, arg3=3)
args.demo(x=1, 2, 3)
args.demo(a=3, x=1, y=2)
```

```
## [1] "x = 1 y = 2 arg3 = 3"
## [1] "x = 1 y = 2 arg3 = 3"
## [1] "x = 1 y = 2 arg3 = 3"
## [1] "x = 1 y = 2 arg3 = 3"
```

```
args.demo2 <- function(x, arg2, arg3) {
  print(paste('x =', x, 'arg2 =', arg2, 'arg3 =', arg3))
}
args.demo2(x=1, y=2, ar=3)
```

```
## Error in args.demo2(x = 1, y = 2, ar = 3): argument 3 matches multiple formal arguments
```

Functions — variable scope

Functions 'see' not only what has been passed to them as arguments:

```
x <- 7
y <- 3
xyplus <- function(x) {
  x <- x + y
  return(x)
}
xyplus(x)
x
```

```
## [1] 10
## [1] 7
```

Everything outside the function is called **global environment**. There is a special operator `<<-` for working on global environment:

```
x <- 1
xplus <- function(x) {
  x <<- x + 1
}
xplus(x)
x
xplus(x)
x
```

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

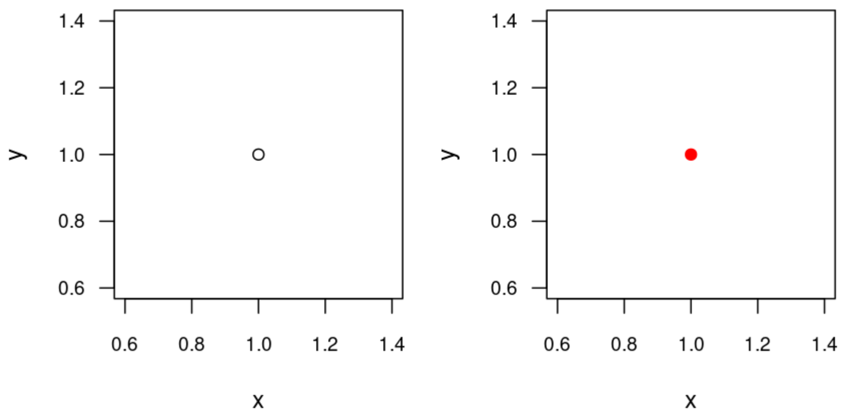
Functions — the `...` argument

There is a special argument **...** (ellipsis) which allows you to give any number of arguments or pass arguments downstream:

```
c # Any number of arguments
my.plot <- function(x, y, ...) { # Passing downstream
  plot(x, y, las=1, cex.axis=.8, ...)
}

{par(mfrow=c(1,2),mar=c(4,4,1,1))
my.plot(1,1)
my.plot(1, 1, col='red', pch=19)}
```

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

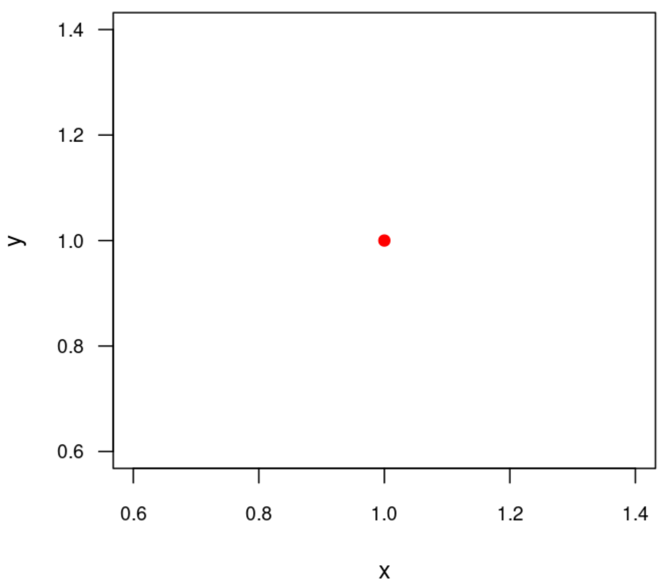


- A function enclosing a function is a **wrapper function**

Functions — the ellipsis argument trick

What if the authors of, e.g. plot.something wrapper forgot about the `...`?

```
my.plot <- function(x, y) { # Passing downstrem
  plot(x, y, las=1, cex.axis=.8, ...)
}
formals(my.plot) <- c(formals(my.plot), alist(... = ))
my.plot(1, 1, col='red', pch=19)
```



R is lazy!

In R, arguments are evaluated as late as possible, i.e. when they are needed. This is **lazy evaluation**:

```
h <- function(a = 1, b = d) {
  d <- (a + 1) ^ 2
  c(a, b)
}
h()
```

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

> The above won't be possible in, e.g. C where values of both arguments have to be known before calling a function **eager evaluation**.

In R everything is a function

Because in R everything is a function

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

we can re-define things like this:

```
`+` <- function(e1, e2) { e1 - e2 }
2 + 2
```

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

and, finally, clean up the mess...

```
rm("+")
2 + 2
```

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

Infix notation

Operators like `+`, `-` or `*` are using the so-called **infix** functions, where the function name is between arguments. We can define our own:

```
`%p%` <- function(x, y) {
  paste(x,y)
}
'a' %p% 'b'
```

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

See you at the next lecture!

Graphics from freepik.com

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