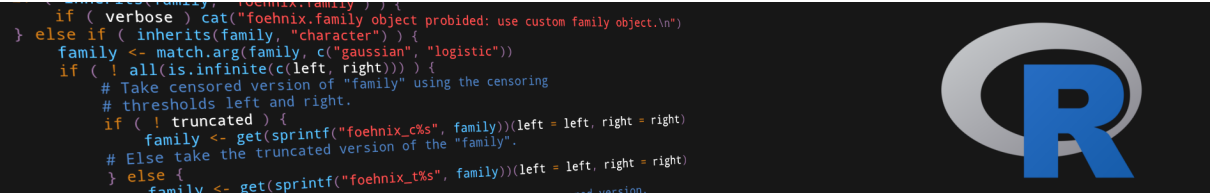
universität innsbruck



Functions

Contents

Last session:

- Basics

This session:

- Exit a function
- Arguments matching
- Functions as arguments
- The ‘...‘ argument

Appendix:

- Lexical scoping
- Recursive function calls
- Lazy evaluation

Functions: Basics

```
01  roll <- function(pips = 1:6) {
02      dice <- sample(pips, size = 2, replace = TRUE)
03      return(sum(dice))
04  }
```

- **Name**: `roll`. For calling the function.
- **Arguments**: `pips`. For providing values to the function.
- **Default values**: `= 1:6`. The value of the argument, if not specified differently.
- **Body**: Line `02` and `03`. List of commands inside the function.
- **Last line of body**: The value of the last line of code is returned by the function. Use `return()` for explicit returning.

Functions: Basics

```
R> roll()
```

```
[1] 9
```

```
R> x <- numeric(10000)
R> for (i in seq_along(x)) {
+    x[i] <- roll()
+  }
R> head(x, 20)
```

```
 [1]  7  4  8  6  6  7 10  3  4  8  2  4  8 10  6  8 12 10 11  5
```

```
R> round(prop.table(table(x)) * 36)
```

```
x
 2  3  4  5  6  7  8  9 10 11 12
 1  2  3  4  5  6  5  4  3  2  1
```

Example: Seven eleven

We want to write a function `seven_eleven` that implements the rules of *Seven Eleven* and executes one round of the game:

- Roll two dice a first time:
  - You win given 7 or 11 points.
  - You lose given 2, 3 or 12 points.
  - If you roll something else the points are called **point**.
- Keep rolling the dice until
  - you roll again the **point**, then you win,
  - or a 7, then you lose.

The function needs no input arguments and should return a numeric `1` if you win or a `0` if you loose.

Collect the functions `roll()` and `seven_eleven()` in an R script called `04_<familyname>.R`, so that you can `source()` it.

Implementation

```
01  seven_eleven <- function() {
02      point <- roll()
03      if (any(point == c(7, 11))) {
04          rval <- 1
05      } else if (any(point == c(2, 3, 12))) {
06          rval <- 0
07      } else {
08          rval <- -1
09          while (rval == -1) {
10              points <- roll()
11              if (points == point) {
12                  rval <- 1
13              } else if (points == 7) {
14                  rval <- 0
15              }
16          }
17      }
18      return(rval)
19  }
```

Enter the casino

```
R> seven_eleven()

[1] 1

R> system.time( x <- replicate(10000, seven_eleven()) )

   user  system elapsed
  0.176   0.000   0.175

R> head(x, 20)

 [1] 1 0 1 0 0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 0

R> prop.table(table(x))

x
     0      1
0.5087 0.4913

R> wiki_value <- 244/495
R> wiki_value

[1] 0.4929
```

Exiting a function

You exit a function either by getting a return ...

```
R> x <- rnorm(100)
R> mean(x, trim = .05)
```

```
[1] -0.111
```

... or by producing an error:

```
R> mean(x, trim = TRUE)
```

```
Error in mean.default(x, trim = TRUE) :
  'trim' must be numeric of length one
```

Exiting a function: Returns

There are three different way to return a value:

- explicit,
- implicit,
- invisible.

Exiting a function: Returns

For explicit returns use `return()`:

```
R> foo1 <- function(x) {
+      if (x >= 0) {
+          return("positive")
+      } else {
+          return("negative")
+      }
+  }
R> foo1(5)
```

```
[1] "positive"
```

```
R> foo1(-5)
```

```
[1] "negative"
```

Exiting a function: Returns

If no explicit return is defined a function in R returns the return value of the last line of the body's code:

```
R> foo2 <- function(x) {
+      if (x >= 0) {
+          rval <- "positive"
+      } else {
+          rval <- "negative"
+      }
+      rval
+  }
R> foo2(5)
[1] "positive"
R> foo2(-5)
[1] "negative"
```

Exiting a function: Returns

For invisible returns use `invisible()`:

```
R> foo3 <- function(x) {
+      if (x >= 0) {
+          rval <- "positive"
+      } else {
+          rval <- "negative"
+      }
+      invisible(rval)
+  }
R> foo3(5)
R> x <- foo3(5)
R> x
```

```
[1] "positive"
```

This is commonly used for `print` and `plot` methods.

Exiting a function: Errors

Of course, the execution of code might fail, leading to an error which also exits the called function.
You can incudle checks or tests in your own functions using `stop()` in combination with an `if` statement.

```
R> foo4 <- function(x) {
+      if (!is.numeric(x) || length(x) != 1) {
+          stop("x should be numeric and of length 1.")
+      }
+      if (x >= 0) "positive" else "negative"    # unstylish one-liner
+  }
```

```
R> foo4("hallo")
```

```
Error in foo4("hallo") : x should be numeric and of length 1.
```

```
R> foo4(c(-5, 5))
```

```
Error in foo4(c(-5, 5)) : x should be numeric and of length 1.
```

Exiting a function: Errors

`stopifnot()` provides a short-cut:

```
R> foo5 <- function(x) {
+      stopifnot(is.numeric(x), length(x) == 1)
+      if (x >= 0) "positive" else "negative"    # unstylish one-liner
+  }
```

```
R> foo5("hallo")
```

```
Error in foo5("hallo") : is.numeric(x) is not TRUE
```

```
R> foo5(c(-5, 5))
```

```
Error in foo5(c(-5, 5)) : length(x) == 1 is not TRUE
```

But:

```
R> foo5(10)
```

```
[1] "positive"
```

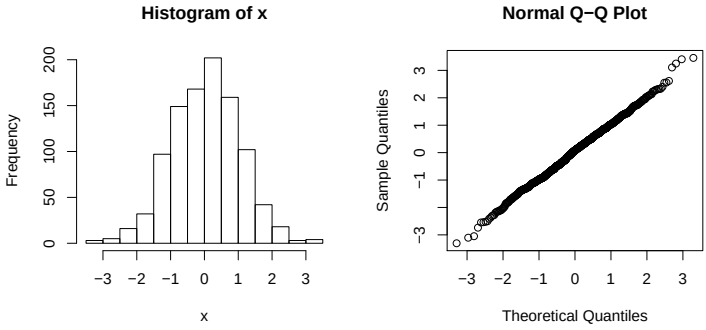
Exiting a function: Exit handlers

Sometimes functions change global parameters, e.g., graphical parameters, options, working directory, . . . that should be reset when exiting a function. To specify the commands that redo these changes use `on.exit()`.

```
R> myplot <- function(x) {
+    # exit handlers
+    old_par <- par(mfrow = c(1, 2))
+    on.exit(par(old_par), add = TRUE)
+
+    # plotting
+    hist(x)
+    qqnorm(x)
+
+    # return
+    invisible(x)
+  }
```

Exiting a function: Exit handlers

```
R> x <- rnorm(1000)
R> myplot(x)
```




```
R> current_par <- par()
R> current_par$mfrow
[1] 1 1
```

Exiting a function: Exit handlers

```
R> myplot("hallo")
Error in hist.default(x) : 'x' must be numeric
R> par()$mfrow
[1] 1 1
```

Argument matching

The function `match.arg()` can be used inside a function to match arguments against a table of candidate values.

The candidate values are specified within the argument list, when the function is defined.

On the one hand this enables auto-completion of arguments, on the other hand it automatically checks if the argument is valid.

Back to the casino example: We decide it would be fancy to specify the type of die—one type of Platonic solid—rather that specifying the pips of a die.

Argument matching

Example:

```
01  roll2 <- function(type = c("cube", "tetrahedron", "octahedron", "dodecahedron",
02                     "icosahedron"), n_dice = 2) {
03      type <- match.arg(type)
04
05      look_up_table <- list(
06          cube         = 1:6,
07          tetrahedron  = 1:4,
08          octahedron   = 1:8,
09          dodecahedron = 1:12,
10          icosahedron  = 1:20
11      )
12      pips <- look_up_table[[type]]
13
14      all_dice <- sample(pips, size = n_dice, replace = TRUE)
15      sum(all_dice)
16  }
```

Argument matching

Example:

```
01  roll2 <- function(type = c("cube", "tetrahedron", "octahedron", "dodecahedron",
02                     "icosahedron"), n_dice = 2) {
03      type <- match.arg(type)
04
05      pips <- switch(type,
06          cube         = 1:6,
07          tetrahedron  = 1:4,
08          octahedron   = 1:8,
09          dodecahedron = 1:12,
10          icosahedron  = 1:20
11      )
12
13      all_dice <- sample(pips, size = n_dice, replace = TRUE)
14      sum(all_dice)
15  }
```

`switch()` is a convenient short-cut for creating the look-up table and looking up the requested element.

Argument matching

```
R> replicate(20, roll2())
 [1]  5 10  6 11  7 11  8  7 10  7  7  3  4  7  3  7 11  6  5  3
R> replicate(20, roll2("icosahedron", n_dice = 1))
 [1]  4 10 12 12  9  6  9 12  7 17  1  9  2  4  8 11  4 10 18  8
R> replicate(20, roll2("octa", n_dice = 1))
 [1] 8 4 4 3 7 6 6 2 2 4 6 8 4 6 8 2 2 4 6 2
R> replicate(20, roll2("t", n_dice = 1))
 [1] 1 1 2 1 3 2 2 3 2 3 1 2 4 2 3 2 3 3 4 3
R> roll2(type = "hallo")
Error in match.arg(type) :
 'arg' should be one of "cube", "tetrahedron", "octahedron", "dodecahedron", "icosahedron"
```

Functions as arguments

One programming paradigm of R is that *everything* is an object, even functions. Thus, functions can be passed to other functions via arguments:

```
01  roll3 <- function(pips = 1:6, n_dice = 2, fn = sum) {
02      all_dice <- sample(pips, size = n_dice, replace = TRUE)
03      fn(all_dice)
04  }
```

```
R> replicate(20, roll3() )
 [1]  3  3  8  9  6  6  9  2  5  9  8  9  7  3 12  8  7  8  3 10
R> replicate(20, roll3(fn = max) )
 [1] 6 1 6 6 6 6 5 4 5 5 5 3 6 3 6 6 4 4 2 5
R> replicate(20, roll3(n_dice = 1, fn = function(x) {x^2}) )
 [1]  1  1 25 25  4  1 25 36  4  1  4 36  4 36 36 25  1 25 36 16
```

Functions as arguments

```
R> roll3(n_dice = 5, fn = c)
[1] 4 6 2 4 3
R> foo <- function(x) {if (x == 1) "You win!" else "You lose!"}
R> roll3(n_dice = 1, fn = foo)
[1] "You win!"
```

CAUTION!!!: This could potentially modify the type of the return value!

Functions as arguments

Examples for functions (in base R) that take a funciton as argument:

- `optim()` R's general purpose optimizer find the set of parameter that minimize a function `fn` provided as argument.
- `integrate()` numerically integrates a function `f`.
- `uniroot()` finds a root (zero) of function `f` in an interval.
- `lapply()` calls a function `FUN` for each element of a list `X`.

The ‘...‘ argument

If you allow functions as an argument, you sometimes want to modify the arguments of the passed functions.
Or there is a key function within your function, of which you want to modify the arguments, e.g. graphical parameters.
This can be done using ‘`...`‘ (*dot-dot-dot*).

```
01  roll4 <- function(pips = 1:6, n_dice = 2, fn = sum, ...) {
02      all_dice <- sample(pips, size = n_dice, replace = TRUE)
03      fn(all_dice, ...)
04  }
```

```
R> replicate(20, roll4())
```

```
 [1]  5  8  9  8  8 10 11  4  2  6  5  7  9  8  6  3  3 10  5 12
```

The '...' argument

```
R> roll4(n_dice = 20, fn = matrix, ncol = 5)
```

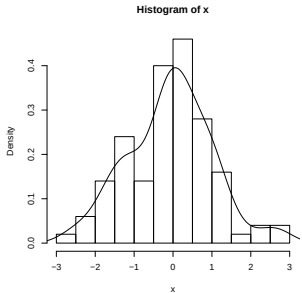
```
     [,1] [,2] [,3] [,4] [,5]
[1,]    4    2    5    5    6
[2,]    4    6    1    1    2
[3,]    4    2    4    5    2
[4,]    6    2    2    2    5
```

```
R> roll4(1:12, n_dice = 8, fn = sort, decreasing = TRUE)
```

```
[1] 11  8  7  6  5  5  3  3
```

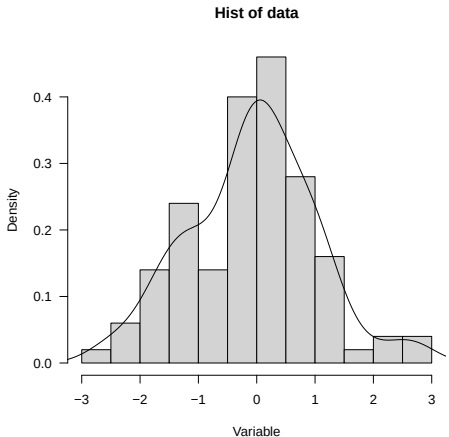
The ‘...‘ argument: Forward plot parameters

```
R> hist2 <- function(x, ...) {
+    h <- hist(x, freq = FALSE, ...)
+    lines(density(x))
+    invisible(h)
+  }
R> x <- rnorm(100)
R> hist2(x)
```



The ‘...‘ argument: Forward plot parameters

```
R> hist2(x, main = "Hist of data", xlab = "Variable", col = "lightgray", las = 1)
```



The ‘...‘ argument: Capture ‘...‘ in list

Write a function that takes only the ‘...‘ argument, and returns the sum of all numeric objects provided within ‘...‘.
One way to do this is

- to capture the dots in a list, and then
- loop of the elements of the list,
- check if the element is numeric, and if yes
- add its value to the return value.

Functions as arguments & the '...' argument

Example: `lapply()` can be used to replace for-loops

Usage: `lapply(X, FUN, ...)`

- `X` an atomic vector or a list.
- `FUN` the function to be applied to each element of `X`.
- `...` optional arguments to `FUN`.

`lapply()` returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.
The other members of the `apply`-family, `sapply` and `vapply`, simplify the return value.

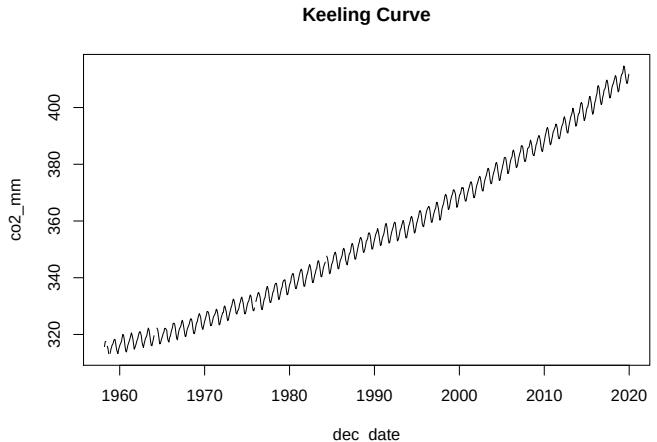
Functions as arguments & the '...' argument

Example: Model selection for the *Keeling curve* (CO_2 record measured at Mauna Loa Observertory, HI)

```
R> # --- load data ---
R> keeling <- read.table("../../data/Keeling/co2_mm_mlo.txt", na.strings = "-99.99")
R> # --- name variables ---
R> names(keeling) <- c("year", "month", "dec_date", "co2_mm", "co2_interpolated",
+                      "co2_trend", "n_days")
R> # --- set missing data for n_days ---
R> keeling$n_days[keeling$n_days == -1] <- NA
```

Example: Keeling Curve

```
R> # --- quicklook ---
R> plot(co2_mm ~ dec_date, data = keeling, type = "l", main = "Keeling Curve")
```



Example: Keeling Curve

Problem: What statistical model fits the CO_2 time-series best?

- A linear trend
- A quadratic trend
- A quadratic trend plus annual cycle

First, we have to derive additional variables:

```
R> keeling <- transform(keeling,
+      dec_date_scaled = (dec_date - min(dec_date)) / diff(range(dec_date)),
+      sin1 = sin(2 * pi * dec_date),
+      cos1 = cos(2 * pi * dec_date)
+  )
```

Example: Keeling Curve

Then, we put all model specifications in a list:

```
R> model_specs <- list(
+      lin   = co2_mm ~ dec_date_scaled,
+      quad  = co2_mm ~ dec_date_scaled + dec_date_scaled^2,
+      cycle = co2_mm ~ dec_date_scaled + dec_date_scaled^2 + sin1 + cos1
+  )
```

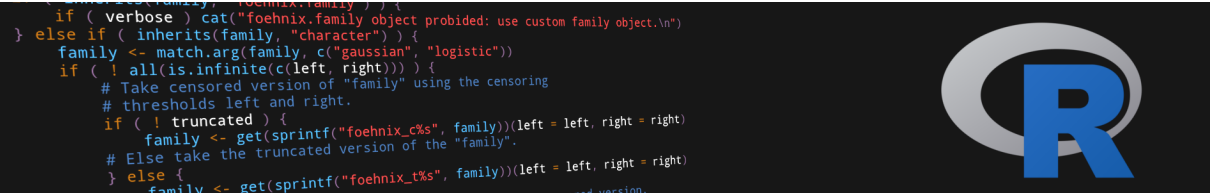
Fit, all models via `lapply()`;

```
R> fitted_models <- lapply(model_specs, lm, data = keeling)
```

And, compute the Bayesian Information Criterion (BIC) for each model:

```
R> sapply(fitted_models, BIC)
```

```
  lin  quad cycle
 4234  4234  4066
```


if (verbose) cat("foehnix.family object probided: use custom family object.\n")
} else if (inherits(family, "character")) {
family <- match.arg(family, c("gaussian", "logistic"))
if (! all(is.infinite(c(left, right)))) {
# Take censored version of "family" using the censoring
# thresholds left and right.
if (! truncated) {
family <- get(sprintf("foehnix_c%s", family))(left = left, right = right)
# Else take the truncated version of the "family".
} else {
family <- get(sprintf("foehnix_t%s", family))(left = left, right = right)

Appendix

Back to the casino

Now after we presented all this functionality for rolling dice, the casino decides to replace it old function `roll()` by this one:

```
01  roll <- function(type = c("tetrahedron", "cube", "octahedron", "dodecahedron",
02                    "icosahedron"), n_dice = 2, fn = sum, ...) {
03      type <- match.arg(type)
04      pips <- switch(type,
05          tetrahedron  = 1:4,
06          cube         = 1:6,
07          octahedron   = 1:8,
08          dodecahedron = 1:12,
09          icosahedron  = 1:20
10      )
11
12      all_dice <- sample(pips, size = n_dice, replace = TRUE)
13      fn(all_dice, ...)
14  }
```

Back to the casino

```
R> x <- replicate(10000, seven_eleven())
R> prop.table(table(x))
```

```
x
     0      1
0.4603 0.5397
```

Oh no, the probability for the player to win is 54%, and thus above 50%, i.e., the casino would lose money.

But how could that happen?

All we did was changing the dice roller `roll()`. In order to understand the relation between `seven_eleven()` and `roll()` we revise the definition of `seven_eleven()`.

Implementation

```
01  seven_eleven <- function() {
02      point <- roll()
03      if (any(point == c(7, 11))) {
04          rval <- 1
05      } else if (any(point == c(2, 3, 12))) {
06          rval <- 0
07      } else {
08          rval <- -1
09          while (rval == -1) {
10              points <- roll()
11              if (points == point) {
12                  rval <- 1
13              } else if (points == 7) {
14                  rval <- 0
15              }
16          }
17      }
18      return(rval)
19  }
```

Lexical scoping

Scoping is the act of finding the value associated with a name.

The function `roll()` was not defined inside the function `seven_eleven()`. Thus, R looks for this function in the *environment* in which the function `seven_eleven()` was defined, here the *global environment*.

If we would have defined a new `roll()` inside `seven_eleven()`, then this new `roll()` would **mask** the `roll()` from the global environment.

After we defined a new `roll()` in the global environment, this `roll()` was found during the execution of `seven_eleven()`. Thus, it doesn't matter **when** but **where** an object is defined.

Lexical scoping

```
R> a <- 1
R> foo <- function() {
+       print(a)
+   }
R> bar <- function() {
+       a <- 10
+       foo()
+   }
```

What is the result when `bar()` is evaulated?

```
R> bar()
```

```
[1] 1
```

Lexical scoping

```
R> a <- 1
R> foo <- function() {
+       print(a)
+   }
R> bar <- function() {
+       a <- 10
+       foo()
+   }
R> a <- 100
```

What is the result when `bar()` is evaulated?

```
R> bar()
```

```
[1] 100
```

Lexical scoping

```
R> a <- 1
R> foo <- function() {
+      print(a)
+  }
R> bar <- function() {
+      a <- 10
+      foo <- function() {
+          print(a)
+      }
+      foo()
+  }
R> a <- 100
```

What is the result when `bar()` is evaulated?

```
R> bar()
```

```
[1] 10
```

Lexical scoping: Function factories

```
R> make_scalefun <- function(xin) {
+      rval <- function(x) {
+          (x - min(xin, na.rm = TRUE)) / diff(range(xin))
+      }
+      rval
+  }
R> scale_time <- make_scalefun(keeling$dec_date)
R> scale_time
function(x) {
        (x - min(xin, na.rm = TRUE)) / diff(range(xin))
    }
<environment: 0x5615604d6cf0>
R> scale_time(1958)
[1] -0.003368
R> scale_time(2020)
[1] 1.001
R> scale_time(2050)
[1] 1.487
```

Lexical scoping: Function factories

```
R> fm <- lm(co2_mm ~ I(scale_time(dec_date)) + I(scale_time(dec_date)^2), data = keeling)
R> nd <- data.frame(dec_date = seq(2020, 2100, by = 1/12))
R> nd$proj <- predict(fm, nd)
R> plot(proj ~ dec_date, nd, type = "l")
```

Recursive function calls

R allows a function to call itself, which is helpful for the implementation of *recursive* algorithms. E.g., recursive partitioning for tree-structured regression models.

Example: Finding the greatest common divisor using the Euclidean algorithm:

```
R> gcd <- function(x, y) {
+      stopifnot(is.integer(x), is.integer(y))
+      if (y == 0L) {
+          rval <- x
+      } else {
+          rval <- gcd(y, as.integer(x %% y))
+      }
+      return(rval)
+  }
R> gcd(105L, 252L)
[1] 21
R> gcd(111L, 259L)
[1] 37
```

Recursive function calls

We could use recursive function calls to re-write the seven-eleven function, by splitting the *first round* from all *follow up rounds*. The first round could look like this:

```
01  seven_eleven2 <- function() {
02      point <- roll()
03      if (any(point == c(7, 11))) {
04          rval <- 1
05      } else if (any(point == c(2, 3, 12))) {
06          rval <- 0
07      } else {
08          rval <- follow_up_round(point = point)
09      }
10      rval
11  }
```

Exercise: Code the `follow_up_round` function using recursive function calls.

Recursive function calls

```
13  follow_up_round <- function(point) {
14      new_point <- roll()
15      if (points == 7) {
16          rval <- 0
17      } else if (new_point == point) {
18          rval <- 1
19      } else {
20          rval <- follow_up_round(point = point)
21      }
22      rval
23  }
```

Recursive function calls

```
R> set.seed(111)
R> x <- replicate(10000, seven_eleven())
R> set.seed(111)
R> x2 <- replicate(10000, seven_eleven2())
R> identical(x, x2)
```

```
[1] TRUE
```

Lazy evaluation

In R, function arguments are lazily evaluated: They are only evaluated if accessed. We already seen function calls that are enabled by lazy evaluation (at least three times).
Do you know where? (In what functions or operators?)

- `&&` and `||`
- `subset`
- `replicate`

Lazy evaluation

In

```
R> is.character(x) && (x == "hallo")
```

the second condition `x == "hallo"` is only considered, if the first condition is `TRUE`.

```
R> x == 5 || y == 10
```

Similar, for the whole condition to be TRUE, it is sufficient that either the first or second condition is TRUE. Thus, if the first condition `x == 5` is `TRUE` there is no need to look at the second condition.

Lazy evaluation

```
R> d <- data.frame(x = c(-1, 1), y = c(1, 2))
R> subset(d, subset = x > 0, select = y)

  y
2 2
```

`x > 0` is evaluated inside the data.frame `d`. Only if there is not a variable `x` is `d`, `x` is taken from the environment, from where `subset` has been called.

Similar the `y` is not evaluated directly, but later within `subset`. Therefore, `subset` is not disturbed by a variable `y` defined in the global environment.

```
R> x <- c(2, 2)
R> y <- "x"
R> subset(d, subset = x > 0, select = y)

  y
2 2
```

Lazy evaluation

```
R> set.seed(111)
R> replicate(10, seven_eleven())
```

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

The *expression* `seven_eleven()` is not evaluated directly. If it was, it would return a nuermic vales (either `0` or `1`), which would be repeated 10 times.

See `rep()`, which replicates elements of a vector.

```
R> set.seed(111)
R> rep(seven_eleven(), 10)
```

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