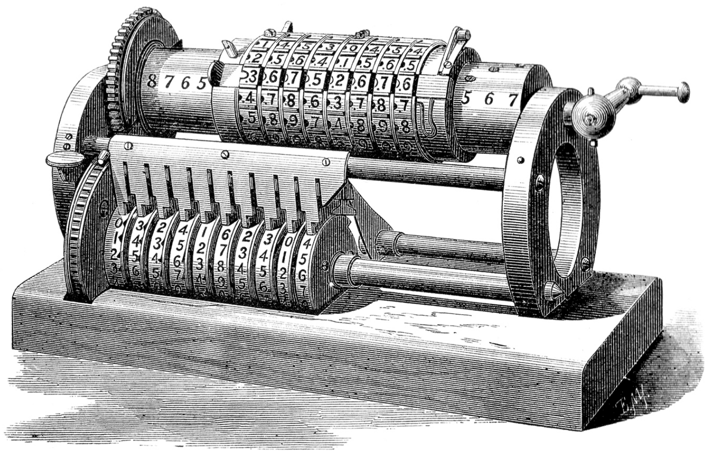
Programming in R



Unit 4: Structured Programming in R (II)

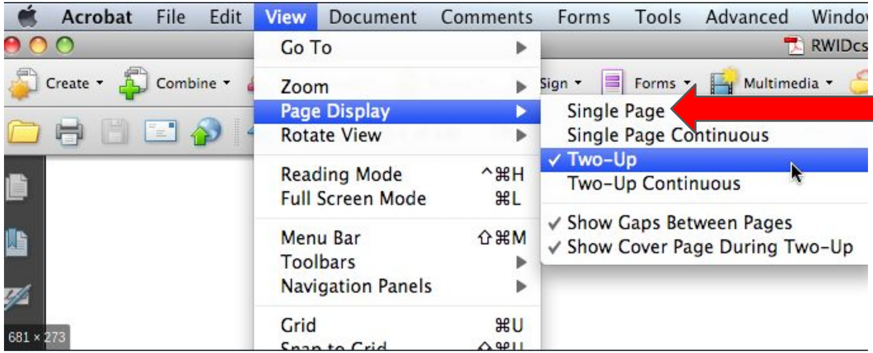
About these Slides

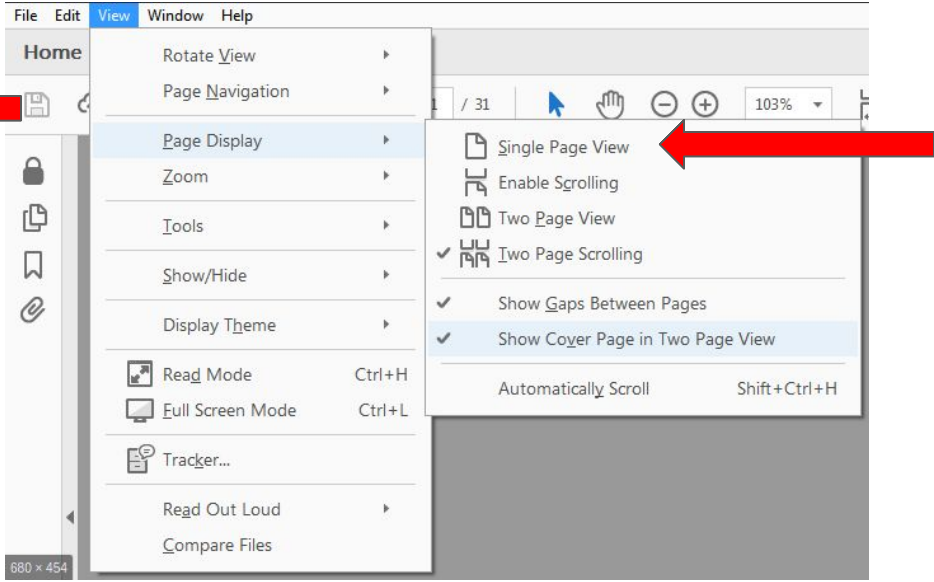
These slides are a bit more verbose than the last set, but the large page number is mostly a consequence because I built in a lot of "animations".

- I have put red circle marks on the "important" slides, mostly at the end of each animation.
- If you just want to read the final slides you can skip over quickly over the non-red-circle pages. I would still recommend looking at the other slides because I think the animations add to the explanation, and because they sometimes contain small comments that may be interesting.
- There are around 40 slides with red dots in this deck

About these Slides

The best way to view these slides (and to get the most out of the animations) is to view it in "presentation mode" or "single page view". See the images below where to set up single page view in your system.





Structured Programming in R

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "R" Track: Functions and `lapply`
- "Tools" Track: The `checkmate` Package

R Track

Functions

Functions

In the following:

- Control flow: what happens in a function?
- Example of recursion
- Default argument values
- Argument matching
- Missing arguments
- "`...`" (dots)-arguments: for functions with dynamic number of arguments
- Variable scope: what variables can be accessed in a function?
- Copy semantics: why functions don't change their arguments from the caller's perspective
- Functions as objects and higher-order functions
- `do.call`

Functions

Argument List

Name

```r
length3D <- function(vector = c(0, 0, 0)) {
  sum.of.squares <- vector[[1]]^2 +
    vector[[2]]^2 +
    vector[[3]]^2
  sqrt(sum.of.squares)
}
```

Function Body

Argument

Default Value of Argument

Last statement of the function is returned as "value" of the function call

Functions

Control Flow:

```
1   > X <- c(1, 2, -2)

2   > Xlen <- length3D(X)

    > print(Xlen)
    [1] 3
```

```
length3D <- function(vector = c(0, 0, 0)) {
  sum.of.squares <- vector[[1]]^2 +
    vector[[2]]^2 +
    vector[[3]]^2
  sqrt(sum.of.squares)
}
```

Functions

Control Flow:

1
```
> X <- c(1, 2, -2)
```
2
```
> Xlen <- length3D(X)

> print(Xlen)
[1] 3
```

```
length3D <- function(vector = c(0, 0, 0)) {
  sum.of.squares <- vector[[1]]^2 +
    vector[[2]]^2 +
    vector[[3]]^2
  sqrt(sum.of.squares)
}
```

Functions

Control Flow:

```
1  > X <- c(1, 2, -2)

2  > Xlen <- length3D(X)

   > print(Xlen)
   [1] 3
```

```
   length3D <- function(vector = c(0, 0, 0)) {
3    sum.of.squares <- vector[[1]]^2 +
       vector[[2]]^2 +
       vector[[3]]^2
4    sqrt(sum.of.squares)
   }
```

Functions

Control Flow:

```
1 > X <- c(1, 2, -2)

2 > Xlen <- length3D(X)

  > print(Xlen)
  [1] 3
```

```
  length3D <- function(vector = c(0, 0, 0)) {
3  sum.of.squares <- vector[[1]]^2 +
     vector[[2]]^2 +
     vector[[3]]^2
4  sqrt(sum.of.squares)
  }
```

Functions

Control Flow:

1

```
> X <- c(1, 2, -2)
```

2

```
> Xlen <- length3D(X)
```

5

```
> print(Xlen)
[1] 3
```

```
length3D <- function(vector = c(0, 0, 0)) {
```

3

```
  sum.of.squares <- vector[[1]]^2 +
    vector[[2]]^2 +
    vector[[3]]^2
```

4

```
  sqrt(sum.of.squares)
}
```

Functions

Control Flow:

```
1   > X <- c(1, 2, -2)

2   > Xlen <- length3D(X)

5   > print(Xlen)
    [1] 3
```

```
    length3D <- function(vector = c(0, 0, 0)) {
3     sum.of.squares <- vector[[1]]^2 +
        vector[[2]]^2 +
        vector[[3]]^2
4     sqrt(sum.of.squares)
    }
```

The above can of course be written much better as

```
sqrt(sum(vector^2))
```

or as

```
norm(vector, type = "2")
```

Functions

`return` statement: return early

```
getSchedule <- function(weekday) {
  if (weekday %in% c("Saturday", "Sunday")) {
    return("Weekend")
  }
  DBI::dbGetQuery(DB,
    DBI::sqlInterpolate(DB,
#  ......
```

Functions

`return` statement: return early

```r
getSchedule <- function(weekday) {
  if (weekday %in% c("Saturday", "Sunday")) {
    return("Weekend")
  }
  DBI::dbGetQuery(DB,
    DBI::sqlInterpolate(DB,
#   ......
```

(As nicer alternative to:)

```r
getSchedule <- function(weekday) {
  if (weekday %in% c("Saturday", "Sunday")) {
    retval <- "Weekend"
  } else {
    DBI::dbGetQuery(DB,
      DBI::sqlInterpolate(DB,
  #   ......
    ))
  }
  retval
}
```

Functions

Recursion

"a method of solving a problem where the solution depends on solutions to smaller instances of the same problem"

Functions

Recursion

"a method of solving a problem where the solution depends on solutions to smaller instances of the same problem"

You will not see recursion that often in *practice* in R but it is vital to understand it since it showcases the behaviour of functions in R very well.

Functions

Recursion

"a method of solving a problem where the solution depends on solutions to smaller instances of the same problem"

You will not see recursion that often in *practice* in R but it is vital to understand it since it showcases the behaviour of functions in R very well.

Recursion is also very elegant.

Functions

Recursion

"Factorial" function:

3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1

Functions

Recursion

"Factorial" function:

3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1

n! = n * (n - 1) * (n - 2) * ... * 2 * 1

Functions

Recursion

"Factorial" function:

3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1

n! = n * (n - 1) * (n - 2) * ... * 2 * 1

n! = n * (n - 1)!   1! = 1

Functions

Recursion

"Factorial" function:

3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1

n! = n * (n - 1) * (n - 2) * ... * 2 * 1

n! = n * (n - 1)!   1! = 1

```r
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

"Factorial" function:

3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1

n! = n * (n - 1) * (n - 2) * ... * 2 * 1

n! = n * $\boxed{(n - 1)!}$ $\boxed{1! = 1}$

```r
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

"Factorial" function:

3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1

n! = n * (n - 1) * (n - 2) * ... * 2 * 1

n! = n * (n - 1)!   1! = 1

```r
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(1)
```

n is 1 --> TRUE

```
fact              ion(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```r
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
> factorial(1)
[1] 1
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

n is 3 --> FALSE

```
fact                      ion(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
  3 * factorial(3 - 1)
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

n is 2 --> FALSE

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
2 * factorial(2 - 1)
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
2 * factorial(2 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
2 * factorial(2 - 1)
```

n is 1 --> TRUE

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
2 * factorial(2 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
2 * factorial(2 - 1)
2 *           1
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
2 * factorial(2 - 1)
2 *           1
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
  3 * factorial(3 - 1)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
  2 * factorial(2 - 1)
  2 *          1 --> 2
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
3 *            2
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
2 * factorial(2 - 1)
2 *           1 --> 2
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
3 *             2
```

Functions

Recursion

```
> factorial(3)
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
3 *           2 --> 6
```

Functions

Recursion

```
> factorial(3)
[1] 6
```

```
factorial <- function(n) {
  if (n <= 1) {
    return(1)
  }
  n * factorial(n - 1)
}
```

```
3 * factorial(3 - 1)
3 *           2 --> 6
```

Functions

Default argument values

```
distance <- function(point, origin = c(0, 0)) {
  sqrt(sum((point - origin)^2))
}
```

Functions

Default argument values

```r
distance <- function(point, origin = c(0, 0)) {
  sqrt(sum((point - origin)^2))
}
```

```
> distance(c(1, 1))
[1] 1.414214
```

Implied: `origin = c(0, 0)`

Functions

Default argument values

```
distance <- function(point, origin = c(0, 0)) {
  sqrt(sum((point - origin)^2))
}
```

```
> distance(c(1, 1))
[1] 1.414214
> distance(c(1, 1), c(1, 0))
[1] 1
```

Implied: `origin = c(0, 0)`

Functions

Default argument values

```r
distance <- function(point, origin = c(0, 0)) {
  sqrt(sum((point - origin)^2))
}
```

```r
> distance(c(1, 1))
[1] 1.414214
> distance(c(1, 1), c(1, 0))
[1] 1
> distance(c(1, 1), origin = c(1, 0))
[1] 1
```

Implied: `origin = c(0, 0)`

Functions

Default argument values

- Used if value is not explicitly supplied
- Convenience: no need to write the "common case"
- A function will often have one or two arguments without defaults, and may have quite a few with defaults.

```
distance <- function(point, origin = c(0, 0)) {
  sqrt(sum((point - origin)^2))
}
```

```
> distance(c(1, 1))
[1] 1.414214
> distance(c(1, 1), c(1, 0))
[1] 1
> distance(c(1, 1), origin = c(1, 0))
[1] 1
```

Implied: `origin = c(0, 0)`

Functions

Default argument values

- Used if value is not explicitly supplied
- Convenience: no need to write the "common case"
- A function will often have one or two arguments without defaults, and may have quite a few with defaults.
- Also often seen in functions in Base-R

```
distance <- function(point, origin = c(0, 0)) {
  sqrt(sum((point - origin)^2))
}
```

```
> distance(c(1, 1))
[1] 1.414214
> distance(c(1, 1), c(1, 0))
[1] 1
> distance(c(1, 1), origin = c(1, 0))
[1] 1
```

Implied: `origin = c(0, 0)`

```
> sample
function (x, size, replace = FALSE, prob = NULL)
{
    if (length(x) == 1L && is.numeric(x) && is.f
```

Functions

Default argument values

- May refer to other arguments!

```
distance <- function(point, origin = rep(0, length(point))) {
  sqrt(sum((point - origin)^2))
}
```

Functions

Default argument values

- May refer to other arguments!

```r
distance <- function(point, origin = rep(0, length(point))) {
  sqrt(sum((point - origin)^2))
}
```

```r
> distance(3)
[1] 3
```

Implied: `origin = rep(0, length(3))`
`= rep(0, 1)`
`= 0`

Functions

Default argument values

- May refer to other arguments!

```r
distance <- function(point, origin = rep(0, length(point))) {
  sqrt(sum((point - origin)^2))
}
```

```
> distance(3)
[1] 3
> distance(c(1, 1, 1))
[1] 1.732051
```

Implied: `origin = rep(0, length(3))`
`= rep(0, 1)`
`= 0`

`origin = rep(0, length(c(1, 1, 1))`
`= rep(0, 3)`
`= c(0, 0, 0)`

Functions

Argument matching

- Arguments can be called **by position**

```r
count <- function(from, to) {
  from:to
}
> count(2, 4)
[1] 2 3 4
```

`from = 2, to = 4`

Functions

Argument matching

- Arguments can be called **by position**
- ... or **by name**

```
count <- function(from, to) {
  from:to
}
> count(2, 4)   from = 2, to = 4
[1] 2 3 4
> count(from = 2, to = 4)
[1] 2 3 4
```

Functions

Argument matching

- Arguments can be called **by position**
- ... or **by name**
- Calling by name is position-independent.

```
count <- function(from, to) {
  from:to
}
> count(2, 4)  from = 2, to = 4
[1] 2 3 4
> count(from = 2, to = 4)
[1] 2 3 4
> count(to = 4, from = 2)
[1] 2 3 4
```

Functions

Argument matching

- Arguments can be called **by position**
- ... or **by name**
- Calling by name is position-independent.
- if *some* arguments are named, the unnamed ones are assigned in-order

```
count <- function(from, to) {
  from:to
}
> count(2, 4)  from = 2, to = 4
[1] 2 3 4
> count(from = 2, to = 4)
[1] 2 3 4
> count(to = 4, from = 2)
[1] 2 3 4
> count(to = 4, 2)  Implied: from = 2
[1] 2 3 4
```

Functions

Argument matching

- Arguments can be called **by position**
- ... or **by name**
- Calling by name is position-independent.
- if *some* arguments are named, the unnamed ones are assigned in-order

```
count <- function(from, to) {
  from:to
}
> count(2, 4)  from = 2, to = 4
[1] 2 3 4
> count(from = 2, to = 4)
[1] 2 3 4
> count(to = 4, from = 2)
[1] 2 3 4
> count(to = 4, 2)  Implied: from = 2
[1] 2 3 4
> count(4, from = 2)  Implied: to = 4
[1] 2 3 4
```

Functions

Argument matching

- Arguments can be called **by position**
- ... or **by name**
- Calling by name is position-independent.
- if *some* arguments are named, the unnamed ones are assigned in-order
- argument names can match partially. This is **bad**, **don't do it**. But be aware it exists.

```
count <- function(from, to) {
  from:to
}
> count(2, 4)        from = 2, to = 4
[1] 2 3 4
> count(from = 2, to = 4)
[1] 2 3 4
> count(to = 4, from = 2)
[1] 2 3 4
> count(to = 4, 2)   Implied: from = 2
[1] 2 3 4
> count(4, from = 2) Implied: to = 4
[1] 2 3 4
> count(t = 4, 2)    Implied: to = 4, from = 2
[1] 2 3 4
```

Functions

Missing arguments

- Not all arguments *need* to be given
    - default arguments

```r
f <- function(y = 1) {
  y
}
f()
#--> 1 (default value of y)
```

Functions

Missing arguments

- Not all arguments *need* to be given
    - default arguments
    - arguments that are not referenced

```r
f <- function(which, y, z) {
  if (which == "y") {
    return(y)
  } else {
    return(z)
  }
}
f("z", y = 2, z = 3)
#--> 3
```

Functions

Missing arguments

- Not all arguments *need* to be given
  - default arguments
  - arguments that are not referenced

```r
f <- function(which, y, z) {
  if (which == "y") {
    return(y)
  } else {
    return(z)
  }
}
f("z", y = 2, z = 3)
#--> 3
```

When "which" argument is not "y", then the y-variable is not used.

Functions

Missing arguments

- Not all arguments *need* to be given
  - default arguments
  - arguments that are not referenced

```r
f <- function(which, y, z) {
  if (which == "y") {
    return(y)
  } else {
    return(z)
  }
}
f("z", y = 2, z = 3)
#--> 3
```

When "which" argument is not "y", then the y-variable is not used.

R allows us to just leave out the y-variable in that case!

Functions

Missing arguments

- Not all arguments *need* to be given
  - default arguments
  - arguments that are not referenced

```r
f <- function(which, y, z) {
  if (which == "y") {
    return(y)
  } else {
    return(z)
  }
}
f("z", z = 3)
#--> 3
```

When "which" argument is not "y", then the y-variable is not used.

R allows us to just leave out the y-variable in that case!

Functions

Missing arguments

- Not all arguments *need* to be given
  - default arguments
  - arguments that are not referenced
  - Use the "missing()" function to check for arguments

```r
f <- function(y) {
  if (missing(y)) {
    cat("Y was missing\n")
    y <- 3
  }
  y
}
f()
# Prints: "Y was missing"
# Return: 3
```

Functions

Missing arguments

- Not all arguments *need* to be given
  - default arguments
  - arguments that are not referenced
  - Use the "missing()" function to check for arguments
    - But often better to use NULL instead (to differentiate optional from required arguments)

```r
f <- function(y) {
  if (missing(y)) {
    cat("Y was missing\n")
    y <- 3
  }
  y
}
f()
# Prints: "Y was missing"
# Return: 3
```

```r
f <- function(y = NULL) {
  if (is.null(y)) {
    cat("Y was not given\n")
    y <- 3
  }
  y
}
f()
# Prints: "Y was not given"
# Return: 3
```

Functions

Missing arguments

- Not all arguments *need* to be given
  - default arguments
  - arguments that are not referenced
  - Use the "missing()" function to check for arguments
    - But often better to use NULL instead (to differentiate optional from required arguments)
    - If possible / sensible, you should of course use default values for optional args

```r
f <- function(y) {
  if (missing(y)) {
    cat("Y was missing\n")
    y <- 3
  }
  y
}
f()
# Prints: "Y was missing"
# Return: 3
```

```r
f <- function(y = NULL) {
  if (is.null(y)) {
    cat("Y was not given\n")
    y <- 3
  }
  y
}
f()
# Prints: "Y was not given"
# Return: 3
```

Functions

Missing arguments

- Not all arguments *need* to be given
  - default arguments
  - arguments that are not referenced
  - Use the "missing()" function to check for arguments
    - But often better to use NULL instead (to differentiate optional from required arguments)
    - If possible / sensible, you should of course use default values for optional args

```r
f <- function(y) {
  if (missing(y)) {
    cat("Y was missing\n")
    y <- 3
  }
  y
}
f()
# Prints: "Y was missing"
# Return: 3
```

```r
f <- function(y = NULL) {
  if (is.null(y)) {
    cat("Y was not given\n")
    y <- 3
  }
  y
}
f()
# Prints: "Y was not given"
# Return: 3
```

Example: ?`t.test()`

Functions

dots

- special function arguments "`...`" can take variable length arguments

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.

```
                 fmt                  ...
> sprintf("My name is %s", "Erich")
[1] "My name is Erich"
```

Functions

dots

- special function arguments "`...`" can take variable length arguments
    - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.

```
> sprintf("My name is %s", "Erich")
[1] "My name is Erich"
> sprintf("My name is %s, I am %d years old.", "Erich", 23)
[1] "My name is Erich, I am 23 years old."
```

fmt ...

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
  - `...length()` : number of `...`-arguments

```
f <- function(x, ...) {
  ...length()
}
```

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
  - `...length()` : number of `...`-arguments

```
f <- function(x, ...) {
  ...length()
}
                ...
> f("a", "b")
[1] 1
```

Functions

dots

- special function arguments "`...`" can take variable length arguments
    - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
    - `...length()` : number of `...`-arguments

```r
f <- function(x, ...) {
  ...length()
}
```

```r
> f("a", "b")
[1] 1
> f("a", "b", "c")
[1] 2
```

...

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
  - `...length()` : number of `...`-arguments
  - `...elt(n)` : get `...`-element n

```
getArg <- function(i, ...) {
  ...elt(i)
}
```

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
  - `...length()` : number of `...`-arguments
  - `...elt(n)` : get `...`-element n

```r
getArg <- function(i, ...) {
  ...elt(i)
}
```

```
                 ...
> getArg(2, "a", "b", "c")
[1] "b"
```

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)`: depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
  - `...length()`: number of `...`-arguments
  - `...elt(n)`: get `...`-element n
  - `list(...)`: get list of `...`-arguments

```r
getArg <- function(i, ...) {
  list(...)[[i]]
}
```

Does mostly the same as before.

(this version "forces argument evaluation", but that difference goes beyond the scope of this course)

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)` : depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
  - `...length()` : number of `...`-arguments
  - `...elt(n)` : get `...`-element n
  - `list(...)` : get list of `...`-arguments
  - or just call another function with `...` as further arguments!

```
sprintfNicely <- function(fmt, ...) {
  fmt <- paste0(
    "Dearest user, please note the following: ",
    fmt
  )
  sprintf(fmt, ...)
}
```

Functions

dots

- special function arguments "`...`" can take variable length arguments
  - Example: `sprintf(fmt, ...)`: depending on "`fmt`" string, different number of args may be needed.
- Write your own "`...`"-functions! Access the elements:
  - `...length()`: number of `...`-arguments
  - `...elt(n)`: get `...`-element n
  - `list(...)`: get list of `...`-arguments
  - or just call another function with `...` as further arguments!

```r
sprintfNicely <- function(fmt, ...) {
  fmt <- paste0(
    "Dearest user, please note the following: ",
    fmt
  )
  sprintf(fmt, ...)
}
> sprintfNicely("%s + %s is %s", 2, 2, 2 + 2)
[1] "Dearest user, please note the following: 2 + 2 is 4"
```

Functions

Variable Scope

- "Lexical Scoping": Functions "see" variables in the environment where they were created.

```r
y <- 1
f <- function(x) {
  x + y
}
f(10)
#--> 11
```

Value 1 from outside

Functions

Variable Scope

- "Lexical Scoping": Functions "see" variables in the environment where they were created.

```r
y <- 1
f <- function(x) {
  x + y
}
f(10)
#--> 11
y <- 2
f(10)
#--> 12
```

Value 2: Call-time value counts!

Functions

Variable Scope

- "Lexical Scoping": Functions "see" variables in the environment where they were created.

It may even reference variables that are created *after* the function is defined!

```
f <- function(x) {
  x + never.seen.before
}
never.seen.before <- 1
f(10)
#--> 11
```

Functions

Variable Scope

- "Lexical Scoping": Functions "see" variables in the environment where they were created.
- "Variable shadowing"*: The innermost variable definition counts.

```r
x <- 100
y <- 1
f <- function(x) {
  x + y
}
f(10)
#--> 11
y <- 2
f(10)
#--> 12
```

This is the *argument* x, not the outer scope variable, so it is 10!

* Some people seem to call this "name masking", but I haven't found any evidence that this term *wasn't* made up by [someone editing wikipedia a decade ago without giving a reference](). The CS term, as far as I am aware, is "variable shadowing". If you know more about this feel free to write me. [Google Ngrams on that matter.]() **
** Yes that's right, I'm going above and beyond the call of duty here, researching this for a 2nd year R course slide deck.

Functions

Variable Scope

- "Lexical Scoping": Functions "see" variables in the environment where they were created.
- "Variable shadowing": The innermost variable definition counts.
- When variables are written, they are written inside the *function's environment*.

```r
f <- function(x) {
  z <- x + 1
  z
}
f(10)
#--> 11
z
#--> Error: object 'z' not found
```

z only exists within the function, it will not be carried outside

Functions

Variable Scope

- "Lexical Scoping": Functions "see" variables in the environment where they were created.
- "Variable shadowing": The innermost variable definition counts.
- When variables are written, they are written inside the *function's environment*.
- ... even when they exist in an outer scope!

```
z <- 0
f <- function(x) {
  z <- x + 1
  z
}
f(10)
#--> 11
z
#--> 0
```

This z only exists within the function. It "shadows" the z from outside

This is the z from outside, it is not changed.

Functions

Variable Scope

- "Lexical Scoping": Functions "see" variables in the environment where they were created.
- "Variable shadowing": The innermost variable definition counts.
- When variables are written, they are written inside the *function's environment*.
- ... even when they exist in an outer scope!
    - ... unless you use the <<- operator.
      But that one is **bad**, **don't use it**.

```r
z <- 0
f <- function(x) {
  z <- x + 1
  z
}
f(10)
#--> 11
z
#--> 0
```

This z only exists within the function. It "shadows" the z from outside

This is the z from outside, it is not changed.

Functions

Copy Semantics

- Remember: assigning values copies them, and changing these copies leaves originals untouched

```
> x
$a
[1] 1

> y <- x
> y$a <- 2
> y
$a
[1] 2

> x
$a
[1] 1
```

Copy x to y

Change y

x is unchanged!

Functions

Copy Semantics

- Remember: assigning values copies them, and changing these copies leaves originals untouched
- This is the same in functions!

Functions

Copy Semantics

- Remember: assigning values copies them, and changing these copies leaves originals untouched
- This is the same in functions!

```r
f <- function(y) {
  y$a <- 2
  y
}
```

```
> x
$a
[1] 1

> f(x)
$a
[1] 2

> x
$a
[1] 1
```

Here the local variable in f gets a *copy* of x

This is the "return value" of f(x)!

x is unchanged!

Functions

Copy Semantics

- Remember: assigning values copies them, and changing these copies leaves originals untouched
- This is the same in functions!
- If you want to change values through functions, you have to assign them the following way:

```r
f <- function(y) {
  y$a <- 2
  y
}
```

```
> x
$a
[1] 1

> x <- f(x)
> x
$a
[1] 2
```

Assign the return-value back to x
--> x is changed

This only works because f returns the changed value!

Functions

Copy Semantics

- Remember: assigning values copies them, and changing these copies leaves originals untouched
- This is the same in functions!
- If you want to change values through functions, you have to assign them
- Exceptions: `<-`-functions, environments, R6-classes. We will see some of these.

Functions

Functions are Objects

- the `f <- function(x) {`-syntax is *just the assignment of a value to a variable*.

Functions

Functions are Objects

- the `f <- function(x) {`-syntax is *just the assignment of a value to a variable*.
    - Can assign function variables to other variables

```
> fff <- length
> fff(c(3, 3))
[1] 2
```

Just the `length`-function!

Functions

Functions are Objects

- the `f <- function(x) {`-syntax is *just the assignment of a value to a variable*.
  - Can assign function variables to other variables
  - Can have *anonymous functions*!

```
> fff <- length
> f <- function(x) x * 2
> f(10)
[1] 20

> (function(x) x * 2)(10)
[1] 20
```

Functions

Functions are Objects

- the `f <- function(x) {`-syntax is *just the assignment of a value to a variable*.
    - Can assign function variables to other variables
    - Can have *anonymous functions*!

```
> fff <- length
> (function(x) x * 2)(10)
```

- This is a powerful concept. We can have Functions that do things with other functions ("higher order functions")!

```
doFunctionTwice <- function(f, arg) {
  result <- f(arg)
  result <- f(result)
  result
}
```

Functions

Functions are Objects

- the `f <- function(x) {`-syntax is *just the assignment of a value to a variable*.
    - Can assign function variables to other variables
    - Can have *anonymous functions*!

```
> fff <- length
> (function(x) x * 2)(10)
```

- This is a powerful concept. We can have Functions that do things with other functions ("higher order functions")!

```
doFunctionTwice <- function(f, arg) {
  result <- f(arg)
  result <- f(result)
  result
}
```

```
> doFunctionTwice(exp, 2)
[1] 1618.178
--> the same as
> exp(exp(2))
[1] 1618.178
```

Functions

Functions are Objects

- the `f <- function(x) {`-syntax is *just the assignment of a value to a variable*.
  - Can assign function variables to other variables
  - Can have *anonymous functions*!

```
> fff <- length
> (function(x) x * 2)(10)
```

- This is a powerful concept. We can have Functions that do things with other functions ("higher order functions")!

```
doFunctionTwice <- function(f, arg) {
  result <- f(arg)
  result <- f(result)
  result
}
```

Using an **anonymous function**:

```
> doFunctionTwice(function(x) x^2, 3)
[1] 81
```

Functions

Functions are Objects

- the `f <- function(x) {`-syntax is *just the assignment of a value to a variable*.
  - Can assign function variables to other variables
  - Can have *anonymous functions*!

```
> fff <- length
> (function(x) x * 2)(10)
```

- This is a powerful concept. We can have Functions that do things with other functions ("higher order functions")!

```
doFunctionTwice <- function(f, arg) {
  result <- f(arg)
  result <- f(result)
  result
}
```

Using an **anonymous function**:

```
> doFunctionTwice(function(x) x^2, 3)
[1] 81
```

This applies x -> x^2 twice --> (3^2)^2 = 81

Functions

Functions are Objects

- This lets us write the following:
  - The function loops through the rows of `mat`, calls `fun()` on it, and returns a vector with the results

```
doToEveryRowOfMyMatrix <- function(mat, fun) {
  result <- vector(length = nrow(mat))
  for (row in seq_len(nrow(mat))) {
    result[[row]] <- fun(mat[row, ])
  }
  result
}
```

Functions

Functions are Objects

- This lets us write the following:
    - The function loops through the rows of `mat`, calls `fun()` on it, and returns a vector with the results

```r
doToEveryRowOfMyMatrix <- function(mat, fun) {
  result <- vector(length = nrow(mat))
  for (row in seq_len(nrow(mat))) {
    result[[row]] <- fun(mat[row, ])
  }
  result
}
```

```
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> doToEveryRowOfMyMatrix(m, sum)
[1] 12 15 18
```

```r
c(sum(c(1, 4, 7)),
  sum(c(2, 5, 8)),
  sum(c(3, 6, 9)))
```

Functions

Functions are Objects

- This lets us write the following:
  - The function loops through the rows of `mat`, calls `fun()` on it, and returns a vector with the results

```
doToEveryRowOfMyMatrix <- function(mat, fun) {
  result <- vector(length = nrow(mat))
  for (row in seq_len(nrow(mat))) {
    result[[row]] <- fun(mat[row, ])
  }
  result
}
```

```
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> doToEveryRowOfMyMatrix(m, sum)
[1] 12 15 18
> doToEveryRowOfMyMatrix(m, prod)
[1]  28  80 162
```

```
c(prod(c(1, 4, 7)),
  prod(c(2, 5, 8)),
  prod(c(3, 6, 9)))
```

Functions

Functions are Objects

- This lets us write the following:
  - The function loops through the rows of `mat`, calls `fun()` on it, and returns a vector with the results

```
doToEveryRowOfMyMatrix <- function(mat, fun) {
  result <- vector(length = nrow(mat))
  for (row in seq_len(nrow(mat))) {
    result[[row]] <- fun(mat[row, ])
  }
  result
}
```

```
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> doToEveryRowOfMyMatrix(m, sum)
[1] 12 15 18
> doToEveryRowOfMyMatrix(m, prod)
[1]  28  80 162
> doToEveryRowOfMyMatrix(m, mean)
[1] 4 5 6
```

Functions

Functions are Objects

- This lets us write the following:
  - The function loops through the rows of `mat`, calls `fun()` on it, and returns a vector with the results

```r
doToEveryRowOfMyMatrix <- function(mat, fun) {
  result <- vector(length = nrow(mat))
  for (row in seq_len(nrow(mat))) {
    result[[row]] <- fun(mat[row, ])
  }
  result
}
```

```
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> doToEveryRowOfMyMatrix(m, sum)
[1] 12 15 18
> doToEveryRowOfMyMatrix(m, prod)
[1]  28  80 162
> doToEveryRowOfMyMatrix(m, mean)
[1] 4 5 6
```

Think about what is happening here, this concept is used a lot in R!

Functions

Functions are Objects

- This lets us write the following:
  - The function loops through the rows of `mat`, calls `fun()` on it, and returns a vector with the results

```
doToEveryRowOfMyMatrix <- function(mat, fun) {
  result <- vector(length = nrow(mat))
  for (row in seq_len(nrow(mat))) {
    result[[row]] <- fun(mat[row, ])
  }
  result
}
```

```
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> doToEveryRowOfMyMatrix(m, sum)
[1] 12 15 18
> doToEveryRowOfMyMatrix(m, prod)
[1]  28  80 162
> doToEveryRowOfMyMatrix(m, mean)
[1] 4 5 6
```

Think about what is happening here, this concept is used a lot in R!

We have basically implemented "`apply()`" here, see next chapter!

Functions

`do.call` with argument lists

- `do.call(fun, args)` does basically the same as
  `fun(args[[1]], args[[2]], args[[3]], ...)`
  for unnamed list `args`, and
  `fun(name1 = args$name1, name2 = args$name2, ...)`
  for named list `args`.

Functions

`do.call` with argument lists

- `do.call(fun, args)` does basically the same as
  `fun(args[[1]], args[[2]], args[[3]], ...)`
  for unnamed list `args`, and
  `fun(name1 = args$name1, name2 = args$name2, ...)`
  for named list `args`.

```
> do.call(sprintf, list("[[%s]]", "xyz"))
[1] "[[xyz]]"
```

is basically the same as

```
> sprintf("[[%s]]", "xyz")
[1] "[[xyz]]"
```

Functions

`do.call` with argument lists

- `do.call(fun, args)` does basically the same as
  `fun(args[[1]], args[[2]], args[[3]], ...)`
  for unnamed list `args`, and
  `fun(name1 = args$name1, name2 = args$name2, ...)`
  for named list `args`.
- Especially useful for results of `lapply()` (see next section) or when function arguments are created programmatically.

Functions

`do.call` with argument lists

- `do.call(fun, args)` does basically the same as
  `fun(args[[1]], args[[2]], args[[3]], ...)`
  for unnamed list `args`, and
  `fun(name1 = args$name1, name2 = args$name2, ...)`
  for named list `args`.
- Especially useful for results of `lapply()` (see next section) or when function arguments are created programmatically.
- Also a use-case:

```
> rows <- list(
+   data.frame(sex = "M", weight = 102),
+   data.frame(sex = "F", weight = 69)
+ )
> do.call(rbind, rows)
  sex weight
1   M    102
2   F     69
```

Get data.frame from list of smaller data.frames

What We Expect You to Know

Functions

Know the following about functions

- How to define functions
- Control flow, `return` statement
- Functions are objects, and can be defined as anonymous functions
- Arguments, argument default values, missing arguments, and when arguments may be missing
- Function argument matching by position and by name
- How variable scoping works. What variables can a function access, what is variable shadowing, and what variables can a function modify?
- Copy semantics prevents functions from changing their arguments directly
- how the `...`-arguments work, how to get the number and values of `...`-arguments, how to pass them on to other function calls
- `do.call` to call functions with a list of arguments

If You Want to Go Beyond This

What we are not covering here: environments, frames and call stacks, closures, promises and lazy argument evaluation. If you want to learn more about that:

- check out Advanced R's chapters on functions and on environments.
- look at the R Language Definition sections 3.5, 4.3.3 and the R help pages of the functions mentioned there

R Track

Function Application: `lapply` & co.

Function Application: `lapply` & co

The last example solves a very common use-case in R:

- "I have a vector / list / matrix / array of something. Apply my function to each element / row / column of it"

Function Application: `lapply` & co

The last example solves a very common use-case in R:

- "I have a vector / list / matrix / array of something. Apply my function to each element / row / column of it"
- Don't forget: the simplest case is just vectorization!

```
> vec <- c(1, 2, 3)
> vec^2
[1] 1 4 9
```

Function Application: `lapply` & co

The last example solves a very common use-case in R:

- "I have a vector / list / matrix / array of something. Apply my function to each element / row / column of it"
- Don't forget: the simplest case is just vectorization!
- Otherwise: `lapply`(X, FUN): apply function `FUN` to each element of `X` and return a `list` of results.

Function Application: `lapply` & co

`lapply()`

Remember: using an anonymous function here

```
> vec <- c(1, 2, 3)
> lapply(vec, function(x) x^2)
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9
```

Function Application: `lapply` & co

`lapply()`

```
> vec <- c(1, 2, 3)
> lapply(vec, function(x) rep(x, 3))
[[1]]
[1] 1 1 1

[[2]]
[1] 2 2 2

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

Function Application: `lapply` & co

`lapply()`

```
> vec <- c(1, 2, 3)
> lapply(vec, function(x) rep(x, 3))
[[1]]
[1] 1 1 1

[[2]]
[1] 2 2 2

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

Alternative: we can give further arguments to FUN directly to lapply. In this case the '3' is passed on to lapply:

```
> lapply(vec, rep, 3)
[[1]]
[1] 1 1 1

[[2]]
[1] 2 2 2

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

Function Application: `lapply` & co

`lapply()`

```
> vec <- c(1, 2, 3)
> lapply(vec, function(x) rep(9, x))
[[1]]
[1] 9

[[2]]
[1] 9 9

[[3]]
[1] 9 9 9
```

Function Application: `lapply` & co

`lapply()`

```
> vec <- c(1, 2, 3)
> lapply(vec, function(x) rep(9, x))
[[1]]
[1] 9

[[2]]
[1] 9 9

[[3]]
[1] 9 9 9
```

Alternative: we can give further **NAMED** arguments to FUN directly to lapply. In this case the '9' is passed as the named argument x of rep:

```
> lapply(vec, rep, x = 9)
[[1]]
[1] 9

[[2]]
[1] 9 9

[[3]]
[1] 9 9 9
```

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```r
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

- Possibility 1: Loop, build up the result

```
result <- logical(0)
for (dat in dates) {
  dat.as.date <- as.POSIXct(dat)
  is.in.the.future <- (dat.as.date - Sys.time()) > 0
  result <- c(result, is.in.the.future)
}
```

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

- Possibility 1: Loop, build up the result

```
result <- logical(0)
for (dat in dates) {
  dat.as.date <- as.POSIXct(dat)
  is.in.the.future <- (dat.as.date - Sys.time()) > 0
  result <- c(result, is.in.the.future)
```

A problem with frequent `result <- c(result, ...)` is that it builds a new vector in every loop cycle. This may be slow. (However, it is always better to have a slow result than have no result at all, so if this is the only thing you can do, do it!)

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

- Possibility 1: Loop, build up the result
- Possibility 2: Loop, allocate result vector beforehand

```
result <- logical(length(dates))
for (dat.index in seq_along(dates)) {
  dat <- dates[[dat.index]]
  dat.as.date <- as.POSIXct(dat)
  is.in.the.future <- (dat.as.date - Sys.time()) > 0
  result[[dat.index]] <- is.in.the.future
}
```

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```r
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

- Possibility 1: Loop, build up the result
- Possibility 2: Loop, allocate result vector beforehand

> Here we allocate the vector in the result vector in the beginning and just write into the right slot in every cycle.

```r
result <- logical(length(dates))
for (dat.index in seq_along(dates)) {
  dat <- dates[[dat.index]]
  dat.as.date <- as.POSIXct(dat)
  is.in.the.future <- (dat.as.date - Sys.time()) > 0
  result[[dat.index]] <- is.in.the.future
}
```

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

- Possibility 1: Loop, build up the result
- Possibility 2: Loop, allocate result vector beforehand
- Possibility 3: **lapply** (or in this case, *sapply*, see next slides)

```
result <- sapply(dates, function(dat) {
  dat.as.date <- as.POSIXct(dat)
  is.in.the.future <- (dat.as.date - Sys.time()) > 0
  is.in.the.future
})
```

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

- Possibility 1: Loop, build up the result
- Possibility 2: Loop, allocate result vector beforehand
- Possibility 3: **lapply** (or in this case, *sapply*, see next slides)

No need to worry about allocating space, still relatively fast!

```
result <- sapply(dates, function(dat) {
  dat.as.date <- as.POSIXct(dat)
  is.in.the.future <- (dat.as.date - Sys.time()) > 0
  is.in.the.future
})
```

Function Application: `lapply` & co

`lapply()` vs. loops

Suppose we have a vector of dates and want to know the ones that are already in the past.

```
dates <- c("2020-04-01", "2020-05-02", "2020-06-21")
```

- Possibility 1: Loop, build up the result
- Possibility 2: Loop, allocate result vector beforehand
- Possibility 3: **lapply** (or in this case, *sapply*, see next slides)
- Possibility 4, you should *always* consider this: Vectorization.

```
result <- as.POSIXct(dates) - Sys.time() > 0
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- sapply(): simplify to a vector / matrix, if possible

```
> sapply(vec, function(x) x %% 2 == 0)
[1] FALSE  TRUE FALSE
```

Vector (logical)

Function Application: `lapply` & co

`lapply()` and similar Functions:

- sapply(): simplify to a vector / matrix, if possible

```
> sapply(vec, function(x) rep(x, 3))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```

matrix

Function Application: `lapply` & co

`lapply()` and similar Functions:

- sapply(): simplify to a vector / matrix, if possible

```
> sapply(vec, function(x) rep(3, x))
[[1]]
[1] 3

[[2]]
[1] 3 3

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

can not be simplified!

Function Application: `lapply` & co

`lapply()` and similar Functions:

- sapply(): simplify to a vector / matrix, if possible
  - sapply has the `simplify` argument, by which the simplification can actually be turned off. But there is still a difference to lapply: sapply turns character-arguments into names, lapply does not:

```
> sapply(c("a", "ab", "abc"), nchar, simplify = FALSE)
$a
[1] 1

$ab
[1] 2

$abc
[1] 3
```

vs.

```
> lapply(c("a", "ab", "abc"), nchar)
[[1]]
[1] 1

[[2]]
[1] 2

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

Function Application: `lapply` & co

`lapply()` and similar Functions:

- vapply(): sapply, but with *type safety*: ensure the return-type of a function!

```
> sapply(head(letters, 3), nchar)
a b c
1 1 1
> sapply(head(letters, 0), nchar)
named list()
```

We expect a numeric, and we get a numeric: perfect

We expect a numeric, but because our function is never called, the sapply doesn't know what type this should be and we get a list --> Suddenly a problem

Function Application: `lapply` & co

`lapply()` and similar Functions:

- vapply(): sapply, but with *type safety*: ensure the return-type of a function!

```
> sapply(c("a b", "c d"), function(x) strsplit(x, split = " ")[[1]])
     a b c d
[1,] "a" "c"
[2,] "b" "d"
> sapply(c("a b", "c d e"), function(x) strsplit(x, split = " ")[[1]])
$`a b`
[1] "a" "b"

$`c d e`
[1] "c" "d" "e"
```

We expect a matrix, and we get a matrix: perfect

We expect a matrix, but because there are differing numbers of elements, the "simplify" doesn't work and we get a list

Function Application: `lapply` & co

`lapply()` and similar Functions:

- vapply(): sapply, but with *type safety*: ensure the return-type of a function!
  - FUN.VALUE argument: give a value of the type you expect.
    - single value: simplify to vector

```
> vapply(head(letters, 3), nchar, 0)
a b c
1 1 1
> vapply(head(letters, 0), nchar, 0)
named numeric(0)
```

We expect a numeric, and we get a numeric: perfect

Still a numeric: great!

Function Application: `lapply` & co

`lapply()` and similar Functions:

- vapply(): sapply, but with *type safety*: ensure the return-type of a function!
    - FUN.VALUE argument: give a value of the type you expect.
        - single value: simplify to vector
        - vector: simplify to a matrix

```
> vapply(c("a b", "c d"), function(x) strsplit(x, split = " ")[[1]], c("a", "a"))
     a b c d
[1,] "a" "c"
[2,] "b" "d"
> vapply(c("a b", "c d e"), function(x) strsplit(x, split = " ")[[1]], c("a", "a"))
Error in vapply(c("a b", "c d e"), function(x) strsplit(x, split = " ")[[1]],  :
  values must be length 2,
 but FUN(X[[2]]) result is length 3
```

We expect a matrix, and we get a matrix: perfect

We expect a matrix, but the result can not be coerced into a matrix, so we get an error telling us something is wrong. **This is what we want**, we don't want this to go unnoticed!

Function Application: `lapply` & co

`lapply()` and similar Functions:

- mapply(): sapply, but going along more than one input list/vector

```
> mapply(function(a, b) {
+    a ^ b
+ }, c(1, 2, 2, 3), c(3, 2, 1, 2))
[1] 1 4 2 9
```

Calls "a ^ b" for a going along c(1, 2, 2, 3) and b going along c(3, 2, 1, 2):

1^3 -> 1
2^2 -> 4 ---> c(1, 4, 2, 9)
2^1 -> 2
3^2 -> 9

Function Application: `lapply` & co

`lapply()` and similar Functions:

- Map(): like mapply(), but not simplifying (like the difference sapply <--> lapply)

```
> Map(function(a, b) {
+    a ^ b
+ }, c(1, 2, 2, 3), c(3, 2, 1, 2))
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 2

[[4]]
[1] 9
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- apply(): sapply, but to each row / column of a matrix
  - apply(mat, 1, fun) --> apply fun() to each *row*
  - apply(mat, 2, fun) --> apply fun() to each *column*.
  - Memorize:

    mat[x, y]: "x" indicates the row, "y" indicates the column. "x" is position 1, "y" is position 2.

```
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> apply(m, 1, sum)
[1] 12 15 18
> apply(m, 2, sum)
[1]  6 15 24
```

row sums. (rowSums() function also does this)

column sums. (colSums() function also does this)

Function Application: `lapply` & co

`lapply()` and similar Functions:

- tapply(): INDEX argument indicates *group*. apply group-wise:

```
> sex <- c("M", "M", "F", "F", "M")
> weight <- c(78, 95, 72, 68, 85)
> tapply(weight, sex, mean)
 F  M
70 86
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- tapply(): INDEX argument indicates *group*. apply group-wise
  - Also works with data.frames:

```
> df
  weight sex
1     78   M
2     95   M
3     72   F
4     68   F
5     85   M
> tapply(df$weight, df$sex, mean)
 F  M
70 86
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- rapply(): recursive lapply: apply function **`f`** to all instances of class **`classes`** in (possibly nested) list **`object`**. Not used that often in my experience.

Function Application: `lapply` & co

`lapply()` and similar Functions:

- Reduce(): Apply a two-argument successively to elements of a list/vector, *and* to the previous result of the function

```
> vec <- c(1, 4, 2, 9)
> Reduce(function(a, b) {
+   s <- a + b
+   cat("Called with", a, "and", b, "returning", s, "\n")
+   s
+ }, vec)
Called with 1 and 4 returning 5
Called with 5 and 2 returning 7
Called with 7 and 9 returning 16
[1] 16
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- Reduce(): Apply a two-argument successively to elements of a list/vector, *and* to the previous result of the function
  - e.g. "sum" function:

```
> Reduce(`+`, 1:5)
[1] 15
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- Reduce(): Apply a two-argument successively to elements of a list/vector, *and* to the previous result of the function
  - e.g. "sum" function:
    ```
    > Reduce(`+`, 1:5)
    [1] 15
    ```
  - e.g. "cumsum" function:
    ```
    > Reduce(`+`, 1:5, accumulate = TRUE)
    [1]  1  3  6 10 15
    ```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- Filter(): Apply a function ("predicate") to each element of a vector / list; return only members for which the function is (coerced to) TRUE.

```
> Filter(function(x) x %% 2 == 0, 1:10)
[1]  2  4  6  8 10
```

  - Although remember this particular example can be easier:

```
> x <- 1:10
> x[x %% 2 == 0]
[1]  2  4  6  8 10
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- Filter(): Apply a function ("predicate") to each element of a vector / list; return only members for which the function is (coerced to) TRUE.
- Find(): Like "Filter()" but return only first (or last, if argument `right = TRUE`) element for which the predicate is TRUE

```
> Find(function(x) x %% 2 == 0, c(1, 1, 4, 6, 8))
[1] 4
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- Filter(): Apply a function ("predicate") to each element of a vector / list; return only members for which the function is (coerced to) TRUE.
- Find(): Like "Filter()" but return only first (or last, if argument `right = TRUE`) element for which the predicate is TRUE
- Position(): Like "Find()", but instead of returning the element of the first element, return its position

```
> Position(function(x) x %% 2 == 0, c(1, 1, 4, 6, 8))
[1] 3
```

Function Application: `lapply` & co

`lapply()` and similar Functions:

- replicate(n, expr): is the odd one out here. It does not take a function, but an expression, and evaluates that expression n times. Mostly useful for sampling random values, and mostly used interactively for simulations.
- Don't forget simplify = FALSE if you want to get a list instead of a vector/matrix.

```
> replicate(4, sample(letters, 3), simplify = FALSE)
[[1]]
[1] "m" "v" "l"

[[2]]
[1] "p" "a" "m"

[[3]]
[1] "u" "f" "q"

[[4]]
[1] "i" "g" "w"
```

Function Application: `lapply` & co

`lapply()` and similar Functions -- **do we need *all* of these**?

- lapply and similar functions make it much easier to write solutions for common problems in R. However, some of them may be more useful to know about than others. The following list is subjective and your experience may vary.
  - Tier 1: You should definitely know how to use `lapply`, `sapply` and `vapply`
  - Tier 2: `apply` and at least one of `mapply`/`Map` are the next most useful functions. `replicate` is very useful for quick experiments interactively.
  - Tier 3: `Filter`, `Reduce` and `tapply` make for nice tricks sometimes but I don't use them often
  - Tier 4: `rapply`, `Find`, `Position`: I personally haven't come across a situation where they made something significantly easier
- Consistently using `vapply` instead of `sapply` is a bit like brushing your teeth, it is not exactly fun but it is good for you and you should do it. In this course will therefore **force you to use vapply instead of sapply** through the style checks.

What We Expect You to Know

`lapply()` and similar Functions

- sapply(): simplify to a vector / matrix, if possible
- vapply(): sapply, but with *type safety*: ensure the return-type of a function!
- mapply(): sapply, but going along more than one input list/vector
  - Map(): like mapply(), but not simplifying (like the difference sapply <--> lapply)
- apply(): sapply, but to to each row / column of a matrix
- tapply(): INDEX argument indicates *group*. apply group-wise:
- rapply(): recursive lapply (don't need to know this one)
- Reduce(): Apply a two-argument successively to elements and previous result
- Filter(): return only members for which a "predicate" is TRUE
- Find(): Like "Filter()" but return only first element
- Position(): Like "Find()", but return position instead of element

Function Application: `lapply` & co

Advanced topic: The "purrr"-package (and why we don't teach it)

- The "tidyverse" has the "purrr"-package with similar functions and tons of syntactic sugar and convenience functions
- If you use the package you get "code dependencies":
  - *always* carries the danger of "breaking changes": suddenly your code doesn't work any more because someone changed their mind about how their package should behave in some case.
  - This may not be worth it if all you want is to write `vapply()` with fewer letters.
  - The tidyverse-people themselves seem to feel this way, they offer "compatibility functions for purrr [...] in cases where purrr is too heavy a package to depend on". (but be careful when copy-pasting this, you may get a conflict with your software license)
- Like much of "tidyverse": convenient interactively, but think a bit before using this in a program / package you write

Tools Track

Assertions and the `checkmate` package

Assertions

There are often cases where you make implicit assumptions about the state of your program

- The user calls your function with an argument, you assume that the argument is an integer greater than 0
- You call a library function, you assume that the return value is a named list with at least one entry
- You change your code which has grown considerably in the last month. Your function used to be called with a list, but now its input must be a vector. You *think* you adapted all places from which your function is being called to the new change

But can you be sure?

Assertions

There are often cases where you make implicit assumptions about the state of your program

- The user calls your function with an argument, you assume that the argument is an integer greater than 0
- You call a library function, you assume that the return value is a named list with at least one entry
- You change your code which has grown considerably in the last month. Your function used to be called with a list, but now its input must be a vector. You *think* you adapted all places from which your function is being called to the new change

But can you be sure? **You can make sure** by throwing an error if it is not the case!

Assertions

- You always want to make sure you throw *errors with a meaningful error message* when something goes wrong

Assertions

- You always want to make sure you throw *errors with a meaningful error message* when something goes wrong
  - as opposed to: hoping your program just naturally runs into an error by itself and having to decrypt what happened (**bad**)

```
> myfun <- function(x) {
+   x + 1
+ }
> myfun("a")
Error in x + 1 (from #2) : non-numeric argument to binary operator
```

Assertions

- You always want to make sure you throw *errors with a meaningful error message* when something goes wrong
  - as opposed to: hoping your program just naturally runs into an error by itself and having to decrypt what happened (**bad**)
  - as opposed to: running your program with invalid data, possibly giving you incorrect results without you noticing (**much much worse**)

```
> myfun <- function(x) {
+   x + 1
+ }
> myfun(NULL)
numeric(0)
```

Assertions

- You always want to make sure you throw *errors with a meaningful error message* when something goes wrong
  - as opposed to: hoping your program just naturally runs into an error by itself and having to decrypt what happened (**bad**)
  - as opposed to: running your program with invalid data, possibly giving you incorrect results without you noticing (**much much worse**)
- We want something like this:

```r
myfun <- function(x) {
  if (!is.numeric(x) || length(x) != 1) {
    stop("Argument x must be a single number")
  }
  x + 1
}
```

Assertions

- You always want to make sure you throw *errors with a meaningful error message* when something goes wrong
  - as opposed to: hoping your program just naturally runs into an error by itself and having to decrypt what happened (**bad**)
  - as opposed to: running your program with invalid data, possibly giving you incorrect results without you noticing (**much much worse**)
- We want something like this:

```
myfun <- function(x) {
  if (!is.numeric(x) || length(x) != 1) {
    stop("Argument x must be a single number")
  }
  x + 1
}
```

```
> myfun("a")
Error in myfun("a") : Argument x must be a single number
> myfun(NULL)
Error in myfun(NULL) : Argument x must be a single number
```

Assertions

- You always want to make sure you throw *errors with a meaningful error message* when something goes wrong
  - as opposed to: hoping your program just naturally runs into an error by itself and having to decrypt what happened (**bad**)
  - as opposed to: running your program with invalid data, possibly giving you incorrect results without you noticing (**much much worse**)
- We want something like this:

```
myfun <- function(x) {
  if (!is.numeric(x) || length(x) != 1) {
    stop("Argument x must be a single number")
  }
  x + 1
}
```

- Some people call this "defensive programming", or "design by contract", but these terms don't seem to have a settled definition and appear to mean whatever the speaker wants them to mean, so we are not going to bother with them here.

Assertions

```r
myfun <- function(x) {
  if (!is.numeric(x) || length(x) != 1) {
    stop("Argument x must be a single number")
  }
  x + 1
}
```

- This is a lot of code for something we are supposed to do regularly
- ==> The checkmate package

checkmate: Fast Argument Checks for Defensive R Programming

by Michel Lang

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise

```
> testNumeric(1)
[1] TRUE
> testNumeric("a")
[1] FALSE
```

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise
  - `assert`-functions: return the value invisibly when constraint satisfied, throw an **error** otherwise

```
> assertNumeric(1)
> assertNumeric("a")
Error: Assertion on '"a"' failed: Must be of type 'numeric', not 'character'.
```

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise
  - `assert`-functions: return the value invisibly when constraint satisfied, throw an **error** otherwise

```
> assertNumeric(1)
> assertNumeric("a")
Error: Assertion on '"a"' failed: Must be of type 'numeric', not 'character'.
```

(what does "return the value invisibly" mean? It means there is no output when we evaluate it, but the value is still returned:

```
> x <- assertNumeric(1)
> x
[1] 1
```

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise
  - `assert`-functions: return the value invisibly when constraint satisfied, throw an **error** otherwise
  - `check`-functions: return **TRUE** if constraint satisfied, return a **string describing the violation** otherwise

```
> checkNumeric(1)
[1] TRUE
> checkNumeric("a")
[1] "Must be of type 'numeric', not 'character'"
```

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise
  - `assert`-functions: return the value invisibly when constraint satisfied, throw an **error** otherwise
  - `check`-functions: return **TRUE** if constraint satisfied, return a **string describing the violation** otherwise
    - Why is this useful? There is also the `assert()`-function that can check multiple "or"-conditions

```
> myfun <- function(x) {
+   # accept 'numeric' or 'character'
+   assert(
+     checkNumeric(x),
+     checkCharacter(x)
+   )
+ }
> myfun(1)
> myfun("a")
> myfun(TRUE)
Error: Assertion failed. One of the following must apply:
 * checkNumeric(x): Must be of type 'numeric', not 'logical'
 * checkCharacter(x): Must be of type 'character', not 'logical'
```

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise
  - `assert`-functions: return the value invisibly when constraint satisfied, throw an **error** otherwise
  - `check`-functions: return **TRUE** if constraint satisfied, return a **string describing the violation** otherwise

There are also `expect`-functions, but we will ignore them for now

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise
  - `assert`-functions: return the value invisibly when constraint satisfied, throw an **error** otherwise
  - `check`-functions: return **TRUE** if constraint satisfied, return a **string describing the violation** otherwise
- A lot of check conditions
  - vector types, scalar types, data.frames, functions, ...

checkmate

```
> install.packages("checkmate")
> library("checkmate")
```

- basically checks constraints on variables: type, length, min / max values, ...
- Four kinds of functions:
  - `test`-functions: return **TRUE** if constraint satisfied, **FALSE** otherwise
  - `assert`-functions: return the value invisibly when constraint satisfied, throw an **error** otherwise
  - `check`-functions: return **TRUE** if constraint satisfied, return a **string describing the violation** otherwise
- A lot of check conditions
  - vector types, scalar types, data.frames, functions, ...
- Offers functions in both snake_case and camelCase to fit with different code styles
  - --> you should use the camelCase functions in this course!

checkmate

- Scalar checks: check / test / assert ...
  - Flag (TRUE/FALSE), Count (nonnegative / positive integer), Number (numeric scalar), Int (integer or close-to-integer*), String (character scalar), ScalarNA (single NA)

checkmate

- Scalar checks: check / test / assert ...
    - Flag (TRUE/FALSE), Count (nonnegative / positive integer), Number (numeric scalar), Int (integer or close-to-integer*), String (character scalar), ScalarNA (single NA)

*A "feature" that may not be what you want is that checkmate will also accept values "close to" an integer here. Future versions of checkmate will let you set the "tol" value to 0, which you should do if you don't have a reason not to.

checkmate

- Scalar checks: check / test / assert ...
  - Flag (TRUE/FALSE), Count (nonnegative / positive integer), Number (numeric scalar), Int (integer or close-to-integer*), String (character scalar), ScalarNA (single NA)
- Vector checks: check / test / assert ...
  - Basically anything with an `is.XXX()` function in R has a checkmate test: Logical, Numeric, Double, Integer (the type integer, not integer value), Character, Complex, Factor, List, POSIXct, Raw, Vector, Atomic, Null
  - Special "vector" checks: Integerish (integer or close-to-integer*), AtomicVector (is.atomic but not NULL and not a matrix / array)

*A "feature" that may not be what you want is that checkmate will also accept values "close to" an integer here. Future versions of checkmate will let you set the "tol" value to 0, which you should do if you don't have a reason not to.

`checkmate`

- Scalar checks: check / test / assert ...
  - Flag (TRUE/FALSE), Count (nonnegative / positive integer), Number (numeric scalar), Int (integer or close-to-integer*), String (character scalar), ScalarNA (single NA)
- Vector checks: check / test / assert ...
  - Basically anything with an `is.XXX()` function in R has a checkmate test: Logical, Numeric, Double, Integer (the type integer, not integer value), Character, Complex, Factor, List, POSIXct, Raw, Vector, Atomic, Null
  - Special "vector" checks: Integerish (integer or close-to-integer*), AtomicVector (is.atomic but not NULL and not a matrix / array)
- Other types: check / test / assert ...
  - Array, DataFrame, Matrix, Date, Function, Formula

`checkmate`

- Scalar checks: check / test / assert ...
  - Flag (TRUE/FALSE), Count (nonnegative / positive integer), Number (numeric scalar), Int (integer or close-to-integer*), String (character scalar), ScalarNA (single NA)
- Vector checks: check / test / assert ...
  - Basically anything with an `is.XXX()` function in R has a checkmate test: Logical, Numeric, Double, Integer (the type integer, not integer value), Character, Complex, Factor, List, POSIXct, Raw, Vector, Atomic, Null
  - Special "vector" checks: Integerish (integer or close-to-integer*), AtomicVector (is.atomic but not NULL and not a matrix / array)
- Other types: check / test / assert ...
  - Array, DataFrame, Matrix, Date, Function, Formula
- Sets: check / test / assert ...
  - Subset, Choice, SetEqual, Disjunct (sic. this checks for ***disjoint*** sets)
  - a "set" here is just an atomic vector with the ordering disregarded.

checkmate

These functions have lots of *common arguments*. Most of them are intuitive:

- null.ok (default FALSE, present for most check functions): accept NULL as value
- na.ok (default FALSE, present for most *scalar* check functions): accept NA as value
- any.missing, all.missing (default TRUE*, present for most *vector* check functions): accept NA for some / for all values.
- len, min.len, max.len (most *vector* check functions): length of acceptable vector
- unique (default FALSE, most *vector* check functions): should values be unique
- names (most *vector* check functions): should the vector / list be named? Interesting possibilities here are "named" and "unique".
- sorted (default FALSE, some *vector* check functions): assume values are sorted ascending
- lower, upper, finite (some *number vector* check functions): lower / upper bounds; should values be non-*Inf*.
- tol (checkInt and checkIntegerish): tolerance up to which to accept a non-integer as "integer".
- min.rows, max.rows, nrows; min.cols, max.cols, ncols (checkMatrix, checkDataFrame)
- ...

*Notice the difference between default acceptance of NAs in "scalar" vs. "vector" functions!

checkmate

Use Cases (a gallery)

```
repeatVector <- function(vector, times) {
  assertVector(vector)
  assertCount(times)
  rep(vector, times)
}
```

checkmate

Use Cases (a gallery)

```r
scheduleDayTime <- function(weekday, hours, minutes) {
  assertChoice(weekday,
    c("Monday", "Tuesday", "Wednesday", "Thursday",
      "Friday", "Saturday", "Sunday")
  )
  assertInt(hours, lower = 0, upper = 23)
  assertInt(minutes, lower = 0, upper = 59)
  ...
}
```

checkmate

Use Cases (a gallery)

```
scheduleDayTime <- function(weekday, hours, minutes) {
  assertChoice(weekday,
    c("Monday", "Tuesday", "Wednesday", "Thursday",
      "Friday", "Saturday", "Sunday")
  )
  hours <- assertInt(hours, lower = 0, upper = 23)
  minutes <- assertInt(minutes, lower = 0, upper = 59)
  ...
}
```

Do the assignment to get conversion to integer (rounding) at the same time, in case you expect values close to but not exactly integers.

checkmate

Use Cases (a gallery)

```
getColumnMeans <- function(df) {
  assertDataFrame(df, min.rows = 1, any.missing = FALSE)
  ...
}
```

checkmate

Use Cases (a gallery)

```
lettersToNumbers <- function(input, use.capitals = FALSE) {
  if (assertFlag(use.capitals)) {
    assertSubset(input, LETTERS)
  } else {
    assertSubset(input, letters)
  }
  ...
}
```

`checkmate`

Interesting helper function: `%??%`:

- Infix-operator: gives the *left* side unless it is NULL, in which case the *right side* is used
- Useful for "default arguments" in some circumstances

checkmate

Interesting helper function: `%??%`:

- Infix-operator: gives the *left* side unless it is NULL, in which case the *right side* is used
- Useful for "default arguments" in some circumstances

```
exponentiate <- function(base, exponent = NULL) {
  base ^ (exponent %??% 2)
}
> exponentiate(3)
[1] 9
> exponentiate(3, 3)
[1] 27
```

checkmate

The *assert*-functions have an additional '.var.name' argument to inform error messages:

```
> assertInteger(x)
Error: Assertion on 'x' failed: Must be of type 'integer', not 'double'.
> assertInteger(x, .var.name = "user input")
Error: Assertion on 'user input' failed: Must be of type 'integer', not 'double'.
```

Use this for more informative errors.

What We Expect You to Know

`checkmate` Functions

- Difference between testXxx, checkXxx, assertXxx functions, and why they are useful
- For a given constraint (type, minimum, length, null / NA ok) you should be able to find the corresponding checkmate assertXxx-function.
  - In particular: vectors, scalars, sets (i.e. atomic vectors with ordering disregarded), data.frame, matrix, function, NULL
- use assert(check(...), check(...), ..) to assert "or"-conditions
- `%??%`-operator to get first non-NULL value

`checkmate` and Homework Tasks

Notice how putting "assertXxx" at the top of your function also serves as implicit documentation: It makes clear at a glance what is expected for a parameter.

To get you used to `checkmate`, we are going to force you to use it in the homework tasks from now on.

E.g. when an exercise tells you to expect a single positive integer, you have to `assertInt(x, lower = 1)` or `assertCount(x, positive = TRUE)` this function argument.