Berliner Hochschule für Technik

Data Science: Statistical Computing with R

Wintersemester 2022/23

Prof. Dr. Steffen Wagner
Berliner Hochschule für Technik

Outline

Why R?

That's why:

- Powerful statistical software
- It allows 'Statistical Programming'
- 'Statistical Programming' allows for automation
- R is the easiest language to speak badly[1]
- A vital user community
- Share code for reproducible research
- It's free!

[1] http://www.r-bloggers.com/r-is-the-easiest-language-to-speak-badly/

Why R? (cont'd)

Blog article: **Which programming language should I learn?**
source: `https://www.r-bloggers.com/2022/08/which-programming-language-should-i-learn/`

R ranks on 4th position because . . .

- . . . it focusses on statistical computing and graphical modeling.
- . . . is a top programming languages for tasks requiring **in-depth** data analysis, graphical data modeling, and time-series and spatial analysis across time.
- . . . offers excellent extensibility.
- . . . integrates with other programming languages effectively, including C, C++, Python, Java, and.NET.

Why R? (cont'd)

Blog article: **Which programming language should I learn?**
source: `https://www.r-bloggers.com/2022/08/which-programming-language-should-i-learn/`

R ranks on 4th position because . . .

. . . it focusses on statistical computing and graphical modeling.

. . . is a top programming languages for tasks requiring **in-depth** data analysis, graphical data modeling, and time-series and spatial analysis across time.

. . . offers excellent extensibility.

. . . integrates with other programming languages effectively, including C, C++, Python, Java, and.NET.

Curious about position 1-3?

What you will learn

Write better Code with Fewer Lines.
- Hadley Wickham -

better code:	Reproducibility and performance
fewer lines:	easy to maintain and share
	no ‘copy & paste’

Course Material

Most of what you will see on the slides, you will find at:

- Course: Roger Peng, Computing for Data Analysis, John Hopkins University, available at `https://www.coursera.org/learn/r-programming`

And also in:

- Wickham, Hadley (2017): R for Data Science, `https://r4ds.had.co.nz/`
- Wickham, Hadley (2019): Advanced R, CRC Press, `https://adv-r.hadley.nz/`
- Harrell jr., Frank (2022): R Workflow, `https://hbiostat.org/rflow/`

Additional Material

General information:

- `https://cran.r-project.org/` and Task Views (`https://cran.r-project.org/web/views/`)
- R-bloggers: `https://www.r-bloggers.com/`

Getting help:

- Rseek: `https://www.rseek.org`
- stackoverflow: `https://stackoverflow.com/`

Outline

- Fundamentals:
  - Data types
  - Subsetting
  - Missing Values
  - Statistics in R
- Graphics
- Data Handling
  - Reading and writing data
  - Data Frames
  - Computing on Data Frames
  - String Manipulation
- Programming
  - Functions
  - Scoping
  - Control Structures
  - `*apply`
  - Debugging
  - Object-Orientation (S3)
  - ...

Outline

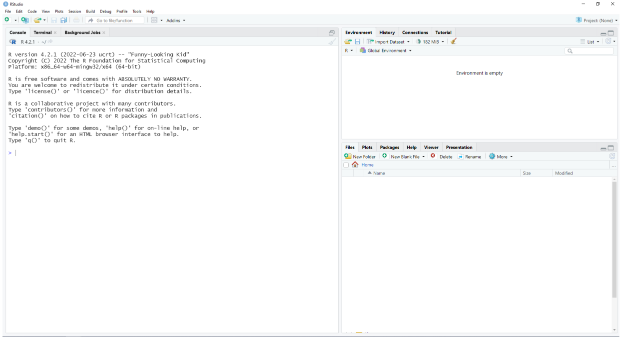
Installation: R

- R can be downloaded from one of the mirror sites in `https://cran.r-project.org/mirrors.html`. You should pick your nearest location.
- " Windows and Mac users most likely want to download the precompiled binaries. Since R is part of many Linux distributions, you should check with your Linux package management system."
- Download R for Linux, Mac or Windows.
- Install R with the standard settings, since later on we will work with RStudio (IDE).

Installation: RStudio

- RStudio can be downloaded from `https://www.rstudio.com/products/rstudio/`.
- Download the *Desktop Version* for Linux, Mac or Windows.
- Install RStudio. (Make sure you already installed R.)
- Start RStudio.

Installation: RStudio



Your screen should look like this.

What is a package?

- An R package includes a set of functions and datasets which is not included in the ‘base’ R System.
- Packages provide additional functionality.
- An exhaustive list of available packages is on CRAN (August 22: about 18.500): `https://cran.r-project.org/web/packages/`

How to Install an R Package

- Install a new package, e. g. ggplot2: `install.packages("ggplot2")`
- Sometimes you need to specify more options. For instance, this is the case if you are not an administrator of the computer.

  `lib` specifies the directory where you want to store the package
  `repos` specifies a list of repositories (CRAN mirrors)
  `dep=T` specifies that all the required packages are also downloaded and installed
- Stay up to date: `update.packages("ggplot2")`
- Load packages so that you are able to use them: `library("ggplot2")`
- Unload packages: `detach("package:ggplot2")`

Outline

Outline

Atomic data types

R has six basic or "atomic" data types:

- character
- numeric (real numbers)
- integer
- complex
- logical (TRUE/FALSE)
- raw

The most basic object is a vector

- a vector can **only** contain objects of the same type
- **Exception:** A *list*, which is represented as a vector, can contain objects of different classes.

Numbers

- Numbers in R are generally treated as numeric objects (i.e. double precision real numbers)
- If you explicitly want an integer, you need to specify the L suffix
- Example: Entering 1 gives you a numeric object;
  entering 1L explicitly gives you an integer.
- There is also a special number `Inf` which represents infinity; e.g.`1/0`; `Inf` can be used in ordinary calculations; e.g. `1/Inf` is `0`
- The value NaN represents an undefined value ("not a number"); e.g. `0/0`; NaN can also be thought of as a missing value (more on that later)

Entering Input

At the R prompt we type *expressions*. The <- symbol is the assignment operator.

```
x <- 1
print(x)
```

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

```
x
```

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

```
msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
# Some R code
x <- ## Incomplete expression
```

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored.

Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
# Some R code
x <- 5 ## nothing printed, but Object x is created
x      ## auto-printing occurs
```

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

```
print(x) ## explicit printing
```

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

The [1] indicates that x is a vector and 5 is the first element.

Printing

```r
x <- 1:50
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

The `:` operator is used to create integer sequences.

```r
length(1:100)
```

```
## [1] 100
```

Creating Vectors

The `c()` function can be used to create vectors of objects.

```
x <- c(0.5, 0.6)       ## numeric
x <- c(TRUE, FALSE)    ## logical
x <- c(T, F)           ## logical, but harder to read
x <- c("a", "b", "c") ## character
x <- 9:29              ## integer
x <- c(1+0i, 2+4i)     ## complex
```

Using the `vector()` function

```
x <- vector("numeric", length = 10)
x
```

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

Mixing Objects

What about the following?

```
c(1.7, "a")          ## character
c(TRUE, 2)           ## numeric
c("a", TRUE)         ## character
c("a", TRUE, 1)      ## character
c("a", c(TRUE, 1))   ## character
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
x <- 0:6
class(x)
```

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

```
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```
as.logical(x)
```

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

```
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```
as.complex(x)
```

```
## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Explicit Coercion

Nonsensical coercion results in NAs.

```
x <- c("a", "b", "c")
as.numeric(x)
```

```
## Warning: NAs durch Umwandlung erzeugt
## [1] NA NA NA
```

```
as.logical(x)
```

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

Factors

Factors are used to represent categorical data. Factors can be *unordered* or *ordered*. Internally factor are stored as "labeled" integer vectors.

- Factors are treated specially by modelling functions like `lm()` and `glm()`
- Using factors with labels is better than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Factors

```r
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```r
table(x)
```

```
## x
##  no yes
##   2   3
```

```r
unclass(x)
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

Factors

The order of the levels can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
x <- factor(c("yes", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

```
table(x)
```

```
## x
## yes  no
##   3   2
```

```
unclass(x)
```

```
## [1] 1 1 2 1 2
## attr(,"levels")
## [1] "yes" "no"
```

Factors

The first levels can be set using `relevel`. Levels can be relabeled and added.

```
## Reorder Levels of Factor
relevel(x, ref="no")
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```
## New and relabeled Levels
exam <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
exam <- factor(exam,
               levels=c(TRUE, FALSE),
               labels=c("success", "failure"))
table(exam)  ## frequency table
```

```
## exam
## success failure
##       3       2
```

Data Frames

Data frames are used to store tabular data

- They are represented as a **special type of list** where every element of the list has to have the same length
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
- data frames can store **different classes of objects** in each column

Data Frames

```
x <- data.frame(eggs = 1:4,
                ham = c(TRUE, TRUE, FALSE, FALSE))
x
```

```
##   eggs   ham
## 1    1  TRUE
## 2    2  TRUE
## 3    3 FALSE
## 4    4 FALSE
```

```
nrow(x)
```

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

```
ncol(x)
```

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

```
dim(x)
```

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

Constructing Data Frames

- `str()` is helpful to check the variable classes and dimensions of a data frame.
- remark[2]: R versions < 4.0.0 coerced character vectors in `data.frames` to factors by default.

```
x <- data.frame(num = 1:4,
                char = "a",
                logic = c(TRUE, FALSE, TRUE, FALSE))
str(x)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ num  : int  1 2 3 4
##  $ char : chr  "a" "a" "a" "a"
##  $ logic: logi  TRUE FALSE TRUE FALSE
```

[2] https://developer.r-project.org/Blog/public/2020/02/16/stringsasfactors/

Constructing Data Frames

Data frames can be constructed by combining data objects. The elements to be combined should have the same number of rows. If not, R will try to repeat elements.

```
a <- 1:4
b <- c("a", "b")
c <- c(TRUE, FALSE, TRUE, FALSE)
x <- data.frame(num = a, "char" = b, c)
str(x)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ num : int  1 2 3 4
##  $ char: chr  "a" "b" "a" "b"
##  $ c   : logi  TRUE FALSE TRUE FALSE
```

Lists

- Lists are a special type of vector that can contain elements of different data types.
- They are different from `data.frames` because the elements can differ in length, so that the structure of the data can not be thought of as being tabular.
- Elements of a list can be lists or data frames.

```
x <- list(1, "a", TRUE)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
```

Named/nested Lists

- Lists can also have names.
- Lists can also be nested and used to represent complex data structures, although this feature should only be used when absolutely necessary:

```
x <- list(a = 1, b = list(1, 2), c = data.frame(x = 1:2))
str(x)
```

```
## List of 3
##  $ a: num 1
##  $ b:List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ c:'data.frame':   2 obs. of  1 variable:
##   ..$ x: int [1:2] 1 2
```

The NULL Object

There is a special object called `NULL`. It

... represents **the** null object in R

... is an object with defined neutral ("null") behavior.

... has no type and no modifiable properties

... should not be confused with a vector or list of zero length

... is a *reserved* word

... is often returned by expressions and functions whose value is undefined

To test for `NULL` use `is.null`.

Outline

Subsetting

There are a number of operators that can be used to extract subsets of R objects.

- [ ] always returns an object of the same class as the original (there is one exception); can be used to select more than one element
- [[ ]] is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
- $ is used to extract elements of a list or data frame by name; semantics are similar to that of [[ ]].

Subsetting

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]
```

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

```
x[2]
```

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

```
x[1:4]
```

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

```
x[x > "a"]
```

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

```
u <- x > "a"
u
```

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

```
x[u]
```

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

Subsetting Data Frames

Subsetting data frames with names:

```
x <- data.frame(eggs = 1:4,
                ham = c(TRUE, TRUE, FALSE, FALSE))
x$eggs
```

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

```
x[["eggs"]]
```

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

```
x[, "eggs"]
```

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

Subsetting Data Frames

Subsetting data frames with positions:

```
x <- data.frame(eggs = 1:4,
                ham = c(TRUE, TRUE, FALSE, FALSE))
x[[1]] #class(x[[1]]) is integer
```

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

```
x[1] #class(x[1]) is data.frame
```

```
##   eggs
## 1    1
## 2    2
## 3    3
## 4    4
```

```
x[c(1,3), 2]
```

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

Subsetting Data Frames

Subsetting data frames with logicals:

```
x <- data.frame(eggs = 1:4,
                ham = c(TRUE, TRUE, FALSE, FALSE))
x[x$eggs > 2, ]
```

```
##   eggs   ham
## 3    3 FALSE
## 4    4 FALSE
```

```
x[x$ham, ]
```

```
##   eggs  ham
## 1    1 TRUE
## 2    2 TRUE
```

```
x[c(TRUE, FALSE)]
```

```
##   eggs
## 1    1
## 2    2
## 3    3
## 4    4
```

Deleting elements of Lists/Data Frames

To delete elements in a list you can simply assign the value `NULL` to them.

```
x[1] <- NULL
x
```

```
##      ham
## 1  TRUE
## 2  TRUE
## 3 FALSE
## 4 FALSE
```

For a subset of rows select the rows to keep or use `data.frame[-<rows>, ]`:

```
x <- x[-(1:2), ]
x
```

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

Missing Values

Missing values are denoted by NA or NaN for undefined mathematical operations.

- `is.na()` is used to test objects if they are NA
- `is.nan()` is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

Removing NA values

A common task is to remove missing values (NA).

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
bad
```

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

```
x[!bad]
```

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

Removing NA values

```
airquality[1:6, ]
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
na.omit(airquality)[1:6, ]
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

Subsetting Lists

- Semantics for subsetting a `list` is equivalent to data frames because they are closely related.
- One exception: A `list` is not meant to represent tabular data, hence subsetting rows/columns is not meaningfull: `list[<rows>, <cols>]` won't work!
- Lists can be recursive (elements of a list can be lists). The subsetting for nested elements works *just* like for a normal list:

```
x <- list(a = list(10, 12, 14),
          b = c(3.14, 2.81))
x[[1]][[3]]
```

```
## [1] 14
```

- The `[[ ]]` can also take an integer sequence:

```
x[[c(1, 3)]]
```

```
## [1] 14
```

```
# not to be confused with: x[c(1, 3)] - lists are vectors!
```

Partial Matching

Partial matching of names is allowed (but not recommended) with [[ ]] and $.

```
x <- list(ham = 1:5, cheese = pi)
x$c
```

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

```
x[["c"]]
```

```
## NULL
```

```
x[["c", exact = FALSE]]
```

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

```
x$h
```

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

```
x <- c(x, hohoho="Hohoho")
x$h
```

```
## NULL
```

Outline

Vectorized Operations

Many operations in R are *vectorized* making code more effcient, concise, and easier to read.

```
x <- 1:4; y <- 6:9
x + y
```

```
## [1]  7  9 11 13
```

```
x > 2
```

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

```
x >= 2
```

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

```
y == 8
```

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

```
x * y
```

```
## [1]  6 14 24 36
```

```
x / y
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Operators

R contains a number of operators. They are listed in the table below.

$+, -$	Plus, Minus, unary or binary
$!$	NOT, unary
$:$	Sequence, binary (in model formulae: interaction)
$*, /$	Multiplication, Division, binary
^	Exponentiation, binary
$\%\%$	Modulus, binary
$\%/\%$	Integer divide, binary
$\% * \%$	Matrix product, binary
$\%in\%$	Matching operator, binary (in model formulae: nesting)
$<, <=$	Less than, Less than or equal, binary
$>, >=$	Greater than, Greater than or equal, binary
$==$	Equal to, binary
$\&, \&\&$	AND (vectorized, not vectorized), binary
$\vert, \vert\vert$	OR (vectorized, not vectorized), binary

Further Details: R Language Definition 3.1.4

AND, OR: vectorized vs. non-vectorized evaluation

&, | The shorter form performs elementwise comparisons in much the same way as arithmetic operators

&&, || The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined. The longer form is appropriate for programming control-flow and typically preferred in `if` clauses.

```
c(TRUE, TRUE) & c(TRUE, FALSE)
```

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

```
c(TRUE, TRUE) && c(TRUE, FALSE)
```

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

```
FALSE && NULL
```

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

```
FALSE && NULL
```

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

Outline

R Style Rules

The goal of Style Rules is to make R code easier to read, share, and verify. There is no unique standard.[3]

1. **File names:** end in `.r` (and have a meaningful name)
2. **Identifiers:** should be meaningful; R is case sensitive

   variables: lowerCamelCase (`dataFrame`, `someData`)
   functions: lowerCamelCase (`someFunction`, `functionName`)
   constants: lowerCamelCase (`i`, `j`, `meanOfSomething`)

3. **Line Length:** maximum 80 characters
4. **Spacing:** Place spaces around all binary operators (`=, +, -, <-`, etc.).
5. **Curly Braces:** first on same line, last on own line
6. **Assignment:** use <-, not =
7. **Semicolons:** don't use them
8. **Commenting Guidelines:** First, do use comments. All comments begin with # followed by a space; inline comments need two spaces before the #

[3] `http://journal.r-project.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf`

Outline

Outline

Reading Data

There are a few principal functions reading data into R:

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `load`, for reading in saved workspaces (*.Rdata files)

Writing Data

There are analogous functions for writing data to files:

- `write.table`
- `writeLines`
- `save`

read.table

The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are seperated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset
- `comment.char`, a character string indicating the comment character
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, logical indicating if character variables should be coded as factors

read.table

```
x <- read.table("Data/someData.txt")
str(x)
```

```
## 'data.frame':    100 obs. of  2 variables:
##  $ someNumbers: num  -0.626 0.184 -0.836 1.595 0.33 ...
##  $ aChar      : chr  "a" "a" "a" "a" ...
```

R will automatically

- skip lines that begin with a #
- detect the number of rows
- assign a class to each variable of the table

You can speed up R by defining these things directly - this is very useful for large datasets. `read.csv` is identical to `read.table` except that the default seperator is a comma.

read.table

Define the class for each variable manually:

```
x <- read.table("Data/someData.txt")
classes <- c(someNumbers = "numeric", aChar = "character")
y <- read.table("Data/someData.txt",
                nrows = 100,
                colClasses = classes)
str(y)
```

```
## 'data.frame':    100 obs. of  2 variables:
##  $ someNumbers: num  -0.626 0.184 -0.836 1.595 0.33 ...
##  $ aChar      : chr  "a" "a" "a" "a" ...
```

readLines

The readLines function can be used to simply read lines of a text file and store them in a character vector.

```
x <- readLines("Data/someData.txt")
x[1:4]
```

```
## [1] "\"someNumbers\" \"aChar\""       "\"1\" -0.626453810742332 \"a\""
## [3] "\"2\" 0.183643324222082 \"a\""  "\"3\" -0.835628612410047 \"a\""
```

```
writeLines(x, con="Data/someData.txt")
x <- read.table("Data/someData.txt")
str(x)
```

```
## 'data.frame':    100 obs. of  2 variables:
##  $ someNumbers: num  -0.626 0.184 -0.836 1.595 0.33 ...
##  $ aChar      : chr  "a" "a" "a" "a" ...
```

Reading and Writing “Foreign” Data

R is able to read and write from different data sources although not by default. The following packages can be helpful:

- `readxl`, `openxlsx` read and write data from and to excel files
- `foreign`, Haven read data stored by Minitab, S, SAS, SPSS, Stata, . . .
- `RODBC`, `sqldf` or `RMySQL`, for SQL databases
- `SPARQL` for semantic web
- `RCurl` for more communication with web resources - `twitteR` as an example for communicating with APIs

Outline

How to manipulate data

When working with data you must:

- Figure out what you want to do.
- Precisely describe what you want in the form of a computer program.
- Execute the code.
- The **`dplyr` package** makes each of these steps as fast and easy as (currently) possible.

In the following, we will have a look at the package's (shortened) introductory vignette.

Single table verbs

dplyr aims to provide a function for each basic verb of data manipulating:

- `filter()` (and `slice()`)
- `arrange()`
- `select()` (and `rename()`)
- `distinct()`
- `mutate()` (and `transmute()`)
- `summarise()`
- `sample_n()` and `sample_frac()`

example dataset

```
library(dplyr)
library(nycflights13)
flights
# A tibble: 336,776 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515         2      830            819
2  2013     1     1      533            529         4      850            830
3  2013     1     1      542            540         2      923            850
4  2013     1     1      544            545        -1     1004           1022
# ... with 336,772 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Data description availabe via `?flights`.

remark

```
class(flights)
[1] "tbl_df"     "tbl"        "data.frame"
```

dplyr can work with data frames as is, but if you're dealing with large data, it's worthwhile to convert them to a `tbl_df`: this is a wrapper around a data frame that won't accidentally print a lot of data to the screen.

Filter rows with `filter()`

For example, we can select all flights on July 1st with:

```
filter(flights, month == 7, day == 1)
# A tibble: 966 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     7     1        1           2029       212      236           2359
2  2013     7     1        2           2359         3      344            344
3  2013     7     1       29           2245       104      151              1
4  2013     7     1       43           2130       193      322             14
# ... with 962 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

This is equivalent to the more verbose:

```
flights[flights$month == 7 & flights$day == 1, ]
```

Multiple filtering conditions

For example, we can select all flights that started early and arrived late:

```
filter(flights, dep_delay < 0 & arr_delay > 0 )
# A tibble: 35,648 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      554            558        -4      740            728
2  2013     1     1      555            600        -5      913            854
3  2013     1     1      558            600        -2      753            745
4  2013     1     1      558            600        -2      924            917
# ... with 35,644 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Select rows by position

To select rows by position, use `slice()`:

```
slice(flights, 3:2)
# A tibble: 2 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      542            540         2      923            850
2  2013     1     1      533            529         4      850            830
# ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Arrange (reorder) rows with `arrange()`

```
arrange(flights, year, month, day)
# A tibble: 336,776 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515         2      830            819
2  2013     1     1      533            529         4      850            830
3  2013     1     1      542            540         2      923            850
4  2013     1     1      544            545        -1     1004           1022
# ... with 336,772 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Arrange (reorder) rows with `arrange()`

Use `desc()` to order a column in descending order:

```
arrange(flights, desc(arr_delay))
# A tibble: 336,776 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     9      641            900      1301     1242           1530
2  2013     6    15     1432           1935      1137     1607           2120
3  2013     1    10     1121           1635      1126     1239           1810
4  2013     9    20     1139           1845      1014     1457           2210
# ... with 336,772 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Select columns with select()

```
# Select columns by name
select(flights, year, month, day)
# A tibble: 336,776 x 3
   year month   day
  <int> <int> <int>
1  2013     1     1
2  2013     1     1
3  2013     1     1
4  2013     1     1
# ... with 336,772 more rows
```

Select columns with select()

```
# Select all columns between year and day (inclusive)
select(flights, year:day)
# A tibble: 336,776 x 3
   year month   day
  <int> <int> <int>
1  2013     1     1
2  2013     1     1
3  2013     1     1
4  2013     1     1
# ... with 336,772 more rows
# Select all columns except those from year to arr_time (inclusive)
select(flights, -(year:arr_time))
# A tibble: 336,776 x 12
  sched_arr_time arr_delay carrier flight tailnum origin dest  air_time distance
           <int>     <dbl> <chr>    <int> <chr>   <chr>  <chr>    <dbl>    <dbl>
1            819        11 UA        1545 N14228  EWR    IAH        227     1400
2            830        20 UA        1714 N24211  LGA    IAH        227     1416
3            850        33 AA        1141 N619AA  JFK    MIA        160     1089
4           1022       -18 B6         725 N804JB  JFK    BQN        183     1576
# ... with 336,772 more rows, and 3 more variables: hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

automated column selection

There are a number of helper functions you can use within `select()`, like

- `starts_with()`
- `ends_with()`
- `matches()`
- `contains()`.

These let you quickly match larger blocks of variable that meet some criterion. See `?select` for more details.

renaming of variables

You can rename variables with `select()` by using named arguments:

```
select(flights, depTime = dep_time)
# A tibble: 336,776 x 1
  depTime
    <int>
1     517
2     533
3     542
4     544
# ... with 336,772 more rows
```

But because `select()` drops all the variables not explicitly mentioned, it's not that useful.

renaming of variables

Instead, use rename():

```
rename(flights, depTime = dep_time)
# A tibble: 336,776 x 19
   year month   day depTime sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>   <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1     517            515         2      830            819
2  2013     1     1     533            529         4      850            830
3  2013     1     1     542            540         2      923            850
4  2013     1     1     544            545        -1     1004           1022
# ... with 336,772 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Extract distinct (unique) rows

A common use of `select()` is to find out which values a set of variables takes. In conjunction with `distinct()` only the unique values are returned.

```
distinct(select(flights, tailnum))
# A tibble: 4,044 x 1
  tailnum
  <chr>
1 N14228
2 N24211
3 N619AA
4 N804JB
# ... with 4,040 more rows
distinct(select(flights, origin, dest))
# A tibble: 224 x 2
  origin dest
  <chr>  <chr>
1 EWR    IAH
2 LGA    IAH
3 JFK    MIA
4 JFK    BQN
# ... with 220 more rows
```

This is very similar to `base::unique()` but should be much faster.

Generate new columns with `mutate()`

As well as selecting from the set of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of the `mutate()` verb.

```
mutate(flights,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60)
# A tibble: 336,776 x 21
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515         2      830            819
2  2013     1     1      533            529         4      850            830
3  2013     1     1      542            540         2      923            850
4  2013     1     1      544            545        -1     1004           1022
# ... with 336,772 more rows, and 13 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#   gain <dbl>, speed <dbl>
```

Generate new columns with mutate()

mutate() allows you to refer to columns that you just created:

```
mutate(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
# A tibble: 336,776 x 21
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     1     1      517            515         2      830            819
2  2013     1     1      533            529         4      850            830
3  2013     1     1      542            540         2      923            850
4  2013     1     1      544            545        -1     1004           1022
# ... with 336,772 more rows, and 13 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#   gain <dbl>, gain_per_hour <dbl>
```

Generate new columns with `transmute()`

If you only want to keep the new variables, use `transmute()`:

```
transmute(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
# A tibble: 336,776 x 2
   gain gain_per_hour
  <dbl>         <dbl>
1     9          2.38
2    16          4.23
3    31         11.6
4   -17         -5.57
# ... with 336,772 more rows
```

Summarise values with `summarise()`

The last verb is `summarise()`, which collapses a data frame to a single row. It's not very useful yet:

```
summarise(flights,
  delay = mean(dep_delay, na.rm = TRUE))
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

Randomly sample rows

You can use `sample_n()` and `sample_frac()` to take a random sample of rows

- a fixed number via `sample_n()`
- a fixed fraction via `sample_frac()`.

Randomly sample rows

```
sample_n(flights, 10)
# A tibble: 10 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     9    30     1720           1720         0     1944           2030
2  2013     7    25     1553           1559        -6     1738           1730
3  2013     3    11     2051           2100        -9     2247           2257
4  2013     1     7     1551           1550         1     1803           1820
# ... with 6 more rows, and 11 more variables: arr_delay <dbl>, carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Randomly sample rows

```
sample_frac(flights, 0.01)
# A tibble: 3,368 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     8    19      708            714        -6      942           1015
2  2013    12    28     1317           1300        17     1423           1425
3  2013    11     8      908            915        -7     1222           1235
4  2013     9    18      826            829        -3     1100           1122
# ... with 3,364 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Use `replace = TRUE` to perform a bootstrap sample, and optionally weight the sample with the `weight` argument.

General structure

You may have noticed that all these functions are very similar:

- The first argument is a data frame.
- The subsequent arguments describe what to do with it, and you can refer to columns in the data frame directly without using $.
- The result is a new data frame

Together these properties make it easy to chain together multiple simple steps to achieve a complex result. These five functions provide the basis of a language of data manipulation.

Grouped operations

At the most basic level, you can only alter a data frame in five useful ways:

1. you can reorder the rows (`arrange()`),
2. pick observations (`filter()`) and
3. variables of interest (`select()`),
4. add new variables that are functions of existing variables (`mutate()`) or
5. collapse many values to a summary (`summarise()`).

The remainder of the language comes from applying the five functions to different types of data, like to **grouped data**, as described next.

Grouped operations

- The introduced verbs are useful, but they become really **powerful** when you combine them with the idea of “group by”, repeating the **operation individually on groups** of observations within the dataset.
- In dplyr, you use the `group_by()` function to describe how to break a dataset down into groups of rows. You can then use the resulting object in exactly the same functions as above; they’ll automatically work “by group” when the input is a grouped.

Grouped operations

The verbs are affected by grouping as follows:

- grouped `select()` is the same as ungrouped `select()`, excepted that retains grouping variables are always retained.
- grouped `arrange()` orders first by grouping variables
- `mutate()` and `filter()` are most useful in conjunction with window functions (like `rank()`, or `min(x) == x`), and are described in detail in `vignette("window-function")`.
- `sample_n()` and `sample_frac()` sample the specified number/fraction of rows in each group.
- `slice()` extracts rows within each group.
- `summarise()` is easy to understand and very useful, and is described in more detail below.

Grouped operations

In the following example, we split the complete dataset into individual planes and then summarise each plane by counting the number of flights (`count = n()`) and computing the average distance (`dist = mean(Distance, na.rm = TRUE)`) and delay (`delay = mean(ArrDelay, na.rm = TRUE)`).

```
by_tailnum <- group_by(flights, tailnum)
delay <- summarise(by_tailnum,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE))
delay <- filter(delay, count > 20, dist < 2000)
delay
# A tibble: 2,962 x 4
  tailnum count  dist delay
  <chr>   <int> <dbl> <dbl>
1 N0EGMQ    371  676.  9.98
2 N10156    153  758. 12.7
3 N102UW     48  536.  2.94
4 N103US     46  535. -6.93
# ... with 2,958 more rows
```

Grouped operations

You use `summarise()` with **aggregate functions**, which take a vector of values, and return a single number. There are many useful functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`. dplyr provides a handful of others:

- `n()`: number of observations in the current group
- `n_distinct(x)`: count the number of unique values in x.
- `first(x)`, `last(x)` and `nth(x, n)` - these work similarly to `x[1]`, `x[length(x)]`, and `x[n]` but give you more control of the result if the value isn't present.

Grouped operations

For example, we could use these to find the number of planes and the number of flights that go to each possible destination:

```
destinations <- group_by(flights, dest)
summarise(destinations,
  planes = n_distinct(tailnum),
  flights = n()
)
# A tibble: 105 x 3
  dest  planes flights
  <chr>  <int>   <int>
1 ABQ      108     254
2 ACK       58     265
3 ALB      172     439
4 ANC        6       8
# ... with 101 more rows
```

Chaining

The dplyr API is functional in the sense that function calls don’t have side-effects, and you must always save their results. This doesn’t lead to particularly elegant code if you want to do many operations at once. You either have to do it step-by-step:

```
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
a3 <- summarise(a2,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
```

Chaining

Or if you don’t want to save the intermediate results, you need to wrap the function calls inside each other:

```
filter(
  summarise(
    select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)
# A tibble: 49 x 5
# Groups:   year, month [11]
   year month   day   arr   dep
  <int> <int> <int> <dbl> <dbl>
1  2013     1    16  34.2  24.6
2  2013     1    31  32.6  28.7
3  2013     2    11  36.3  39.1
4  2013     2    27  31.3  37.8
# ... with 45 more rows
```

The pipe operator

This is difficult to read because the order of the operations is from inside to out, and the arguments are a long way away from the function. To get around this problem, dplyr provides the **`%>%` operator**. `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations so you can read from left-to-right, top-to-bottom:

The pipe operator

```
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)
# A tibble: 49 x 5
# Groups:   year, month [11]
   year month   day   arr   dep
  <int> <int> <int> <dbl> <dbl>
1  2013     1    16  34.2  24.6
2  2013     1    31  32.6  28.7
3  2013     2    11  36.3  39.1
4  2013     2    27  31.3  37.8
# ... with 45 more rows
```

Outline

Strings and Characters in Statistical Analysis

Read in data often must be “cleaned” and explicitly coerced to the class of data, that R can deal with.

- Specification of missing values (“99” , “NA”, “.”)
- Different formats in the same column: (“0.5”, “0,5”, “1/2”)
- Dates: `"2021-05-03 10:14:15"`, `"03.05.2021 10:14:15"`
- Contamination of data
- Getting information out of text, for example in text mining, HTML-code and other data represented in textual format

Functions

There are some fundamental functions for manipulating strings and character vectors in R:

- `paste()`, converts all arguments to character and concatenates them to one vector
- `strsplit()`, splits a string into a list of substring
- `grep()`, search for pattern in string
- and many related functions

Additional topic: Regular expressions

paste

paste() converts any arguments to a character and concatenates them:

```
x <- "Some"
paste(x, "String")
```

```
## [1] "Some String"
```

```
y <- paste(x, "String")
y
```

```
## [1] "Some String"
```

```
z <- c("Some", "More")
a <- paste(z, "String", sep = "_")
a
```

```
## [1] "Some_String" "More_String"
```

strsplit

`strsplit` splits a string into several substring:

```
strsplit(y, split = " ")
```

```
## [[1]]
## [1] "Some"   "String"
```

```
b <- strsplit(a, split = "_")
b
```

```
## [[1]]
## [1] "Some"   "String"
##
## [[2]]
## [1] "More"   "String"
```

escape sequences

There are several string constants which can be used with escape sequences. Escape sequences are introduced using a backslash.

- `\"` double quote
- `\n` new line
- `\t` tab
- `\\` backslash
- see more in the R Language Definition

This is why you have to specify `"path/someFile.R"` or `"path\\someFile.R"` and not `"path\someFile.R"`.

escape sequences

Escape sequences can be stored in character vectors, they will be “evaluated’’ at the time when R needs to interprete a character - for example when defining file names or paths.

```
newLine <- "new\nline"
print(newLine)
```

```
## [1] "new\nline"
```

```
cat(newLine)
```

```
## new
## line
```

```
moreNewLines <- paste(rep("new", 3), "line", sep = "\n")
cat(moreNewLines)
```

```
## new
## line new
## line new
## line
```

grep

If we want to find patterns inside of characters or strings the grep-function family supplies various ways to find, extract or replace substrings. See the help page for grep to find out more.

```
a
```

```
## [1] "Some_String" "More_String"
```

```
grep("String", a)
```

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

```
grep("Something", a)
```

```
## integer(0)
```

```
grepl("Some", a)
```

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

```
sub("String", "Something", a)
```

```
## [1] "Some_Something" "More_Something"
```

grep

Select only those columns in dat which start with “x’ ’:

```
dat <- as.data.frame(matrix(1:100, nrow = 10))
names(dat)[1:5] <- paste("x", 1:5, sep = "")
str(dat)
```

```
## 'data.frame':	10 obs. of  10 variables:
##  $ x1 : int  1 2 3 4 5 6 7 8 9 10
##  $ x2 : int  11 12 13 14 15 16 17 18 19 20
##  $ x3 : int  21 22 23 24 25 26 27 28 29 30
##  $ x4 : int  31 32 33 34 35 36 37 38 39 40
##  $ x5 : int  41 42 43 44 45 46 47 48 49 50
##  $ V6 : int  51 52 53 54 55 56 57 58 59 60
##  $ V7 : int  61 62 63 64 65 66 67 68 69 70
##  $ V8 : int  71 72 73 74 75 76 77 78 79 80
##  $ V9 : int  81 82 83 84 85 86 87 88 89 90
##  $ V10: int  91 92 93 94 95 96 97 98 99 100
```

```
dat[1:3, grep("x", names(dat))]
```

```
##   x1 x2 x3 x4 x5
## 1  1 11 21 31 41
## 2  2 12 22 32 42
## 3  3 13 23 33 43
```

grep

```
names(dat)[1:2] <- c("v1x", "x_Something")
dat[1:3, grep("x", names(dat))]  # that's not what we want
```

```
##   v1x x_Something x3 x4 x5
## 1   1             11 21 31 41
## 2   2             12 22 32 42
## 3   3             13 23 33 43
```

```
dat %>% select(starts_with("x")) %>% slice(1:3)  # dplyr
```

```
##   x_Something x3 x4 x5
## 1             11 21 31 41
## 2             12 22 32 42
## 3             13 23 33 43
```

Regular expressions

The previous example showed that just by defining `pattern = "x"`, `grep` will find any string containing the letter ‘x’. Regular expressions can be used to generalize the pattern, hence to abstract it. This is implemented by using special symbols, reserved to match specific patterns. To solve the previous problem, we can modify the pattern:

```
dat[1:3, grep("^x", names(dat))]
```

```
##   x_Something x3 x4 x5
## 1          11 21 31 41
## 2          12 22 32 42
## 3          13 23 33 43
```

‘^’ defines that the pattern to be searched will **begin with** ‘x’.

Regular expressions

For a further extension we only want to use those variables which start with an ‘x’ and are followed by a number between 0 and 9:

```
names(dat)
```

```
##  [1] "v1x"         "x_Something" "x3"          "x4"          "x5"
##  [6] "V6"          "V7"          "V8"          "V9"          "V10"
```

```
grep("^x[0-9]", names(dat))
```

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

Now allow ‘x’ to be upper or lower case:

```
grep("^[Xx][0-9]", names(dat))
```

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

If you should ever need regular expressions, `?regex` offers some information about the implementation and possibilities in R. See Spector (2008) for an introduction.

Final Remarks

There are many more topics to cover in the context of data handling. Some remarks if you want or need more:

- See for a focused overview P. Spector (2008): Data Manipulation with R, Springer and/or P. Teetor (2011): R Cookbook, O'Reilly for many good examples with valuable solutions to many typical problems
- The package `tidyr` for data cleaning and reshaping
- In case that you face difficulties due to really large data sets, the `readr` package might be helpful: similar to `read.table` but *much* faster!

Outline

Plotting

The plotting and graphics engine in R is encapsulated in a few base and recommended packages:

- **graphics**: contains plotting functions for the ‘base’ graphing systems, including `plot`, `hist`, `boxplot` and many others.
- **ggplot2**: introduces a clean syntax for customized graphics using the ’grammar of graphics’. A lot of default settings for legends and colours.
- **lattice:** there´s a nice introductive tutorial: `http://dsarkar.fhcrc.org/lattice-lab/latticeIntro.pdf`
- **ggvis**: interactive graphics based on the ’grammar of graphics’
- **shiny**: interactive applications

Outline

Basic graphics

- Base graphics are usually constructed piecewise, with each aspect of the plot handled separately through a series of function calls.
- This plotting approach mirrors the thought process.
- Plotting commands are divided into two basic groups:
  - **High-level plotting functions**
    create a new plot on the graphics device (more on that later), possibly with axes, labels, titles and so on.
  - **Low-level plotting functions**
    add more information to an existing plot, such as extra points, lines, labels, etc.
- In addition, R maintains a list of graphical parameters `par()` which can be manipulated to customize your plots.

High-level plotting functions

High-level plotting functions

- generate a complete plot of the data passed as arguments to the function. Where appropriate, axes, labels and titles are automatically generated (unless you request otherwise).
- (usually) start a new plot, erasing the current plot if necessary.
- **`plot()` ... *generic function*; output depends**
  `pairs()` ... matrix of scatterplots
  `smoothScatter()` ... color density representation of a 2D-scatterplot
  `boxplot` ... box-and-whisker plot(s) of (grouped) values
  `barplot()` ... dto
  `qqplot()` ... dto
  `hist()` ... histogramm

  ⋮

  Further Functions: An Introduction to R 12.1.3

The `plot()` function is a *generic function*

generic functions: result depends on class or type of the first argument.

- **`plot(x)`:**
  if x is a time series → time-series plot
  if x is a numeric vector → plot of the values in the vector against index
  if x is a complex vector → plot of imaginary vs real parts of the vector elements
  if x is a a factor → barplot
- **`plot(x, y)`:**
  if x and y are numeric vectors → scatter plot
  if x is a factor and y is numeric → grouped boxplot
- **`plot(xy)`:**
  if xy is either a list containing two elements x and y or
  xy is a two-column matrix → scatter plot / box plot
- **`plot(df)`:**
  if `df` is a data frame → matrix of scatterplots
- **`plot(fx)`:**
  if `fx` is a function → a curve corresponding to `fx` ist plotted
- ...

Arguments to high-level plotting functions

`add = TRUE`	superimposes plot on current plot (forces the function to act as a low-level graphics function)
`axes = FALSE`	suppresses the generation of axis (you can customize the axes later with axis)
`log = "x", "y"`	
`log = "xy"`	Causes the x, y or both axes to be logarithmic.
`type=`	The `type=` argument controls the type of plot as follows:

`type="p"`	Plot individual points (default)
`type="l"`	Plot lines
`type="b", "o"`	Plot points connected/overlaid by lines
`type="h"`	Plot vertical lines from points to x axis
`type="s", "S"`	Step function plots
`type="n"`	no plotting, axes are still drawn

`xlab = string`	
`ylab = string`	Axis labels for the x and y axes
`main = string`	Figure title, placed at the top of the plot in a large font.
`sub = string`	Sub-title, placed just below the x-axis in a smaller font.

Low-level plotting functions

Low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot.

`points(x, y)`	
`lines(x, y)`	Adds points or connected lines to the current plot.
`text(x, y, ...)`	Add text to a plot at points given by x, y.
`abline(a, b)`	
`abline(h = y)`	
`abline(v = x)`	
`abline(lm.obj)`	Adds a line of slope b and intercept a to the current plot. h and v add horizontal and vertical lines. The result of a linear modell `lm.obj` with coefficients a and b may be assigned as well.
`legend(...)`	Adds a legend to the current plot.
`title(...)`	Adds a title main to the top of the current plot.
`axis(side, ...)`	Adds an axis to the current plot on specified the side.

Outline

The elements of the grammar (`ggplot2`)

- **Data** and aesthetic **mappings**:
  - Every plot is based on a data frame
  - Aesthetic mappings: data is mapped to aesthetic attributes
- **Geoms** are the graphical representation of a statistic you want to visualize: points, lines, polygons, etc.
- **Stats** are the statistics you want to calculate for the data. A scatter plot simply uses the identity, a bar plot often uses relative frequencies. A statistic is always associated with a geom.
- **Scales** map values in the data space to values in the aesthetic space. They define the shape, size or colour and are used to create legends and axes
- The coordinate system (**coord**), typically the Cartesian coordinate system
- **Facet**ing is used to split the data into subsets and then to reproduce the graphic on each subset (see extra slides on graphics for an example)
- **Theme** defines fonts of all labels, the shape of ticks and background. Use the function `labs` to add labels and a title

The elements of the grammar

- Unlike most functions in R there is a naming convention in the package.
  - `geom_<geom-name>` to add geoms with statistic
  - `stat_<stat-name>` to add a statistic with geom
  - `scale_<aesthetic-name>_<scale-type>` to manipulate scales
  - `coord_<coord-type>` to change the coordinate system
  - `facet_<facet-type>` to add facets
  - `theme` to manipulate text and line elements (which are not geoms)
- This naming conventions mirrors the grammar and is the reason why you need to understand it
- The grammar does not define how a graphic should look like. What is appropriate and inappropriate is up to you
- `ggplot2` only produces static graphics, no movement, no interaction, no 3D
- The function `qplot` mimics the syntactical usage of `plot`

ggplot

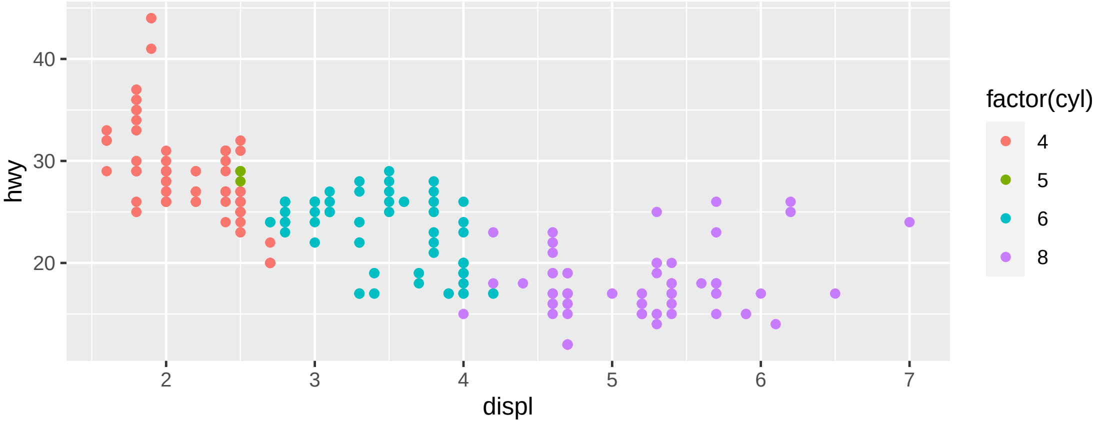
- The fundamental function in `ggplot2` is `ggplot`
- `ggplot` is the fundament of every plot created by `ggplot2`, any value specified in `ggplot` is used as default for any layer. Actually it is simply passed using `...`
- A layer is the representation of the grammar elements which define the plot
- A plot can be created with more than one layer. Actually it is created layer-by-layer
- Every component of a plot - aesthetics, geoms, stats, scales, etc. - can be added using the binary operator '+'

scales

- Scales map values in the data space to values in the aesthetic space
- `aes()` simply connects data values (variables) to aesthetic attributes, like x, y, fill, alpha, etc.
- Scales calculate the positions in the Cartesian coordinate system - represented by numbers between 0 and 1
- Scales will pick colours and automatically generate legends for aesthetics like 'colour'
- Every scale can be manipulated by using `scale_'aesthetic-name'_'scale-type'()`
- Scales are picked by default according to the class of the data: Numeric data will have a continuous scale, a factor usually uses a discrete scale
  - if 'x' is numeric: `scale_x_continuous()`
  - if 'colour' is a factor: `scale_colour_discrete()`

Example: Mapping and scales

```
data(mpg)
ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) +
  geom_point()
```



Example: Mapping and scales

. . . the data:

manufacturer	**model**	**displ**	**hwy**	**cty**
audi	*a4*	1.8	29	4
audi	*a4*	1.8	29	4
audi	*a4*	2.0	31	4
audi	*a4*	2.0	30	4
audi	*a4*	2.8	26	6
audi	*a4*	2.8	26	6
audi	*a4*	3.1	27	6
audi	*a4*	1.8	26	4
audi	*a4*	1.8	25	4
audi	*a4*	2.0	28	4

Example: Mapping and scales

. . . connection between aesthetic attribute and data:

x(*displ*)	**y**(*hwy*)	**colour**(*cty*)
1.8	29	4
1.8	29	4
2.0	31	4
2.0	30	4
2.8	26	6
2.8	26	6
3.1	27	6
1.8	26	4
1.8	25	4
2.0	28	4

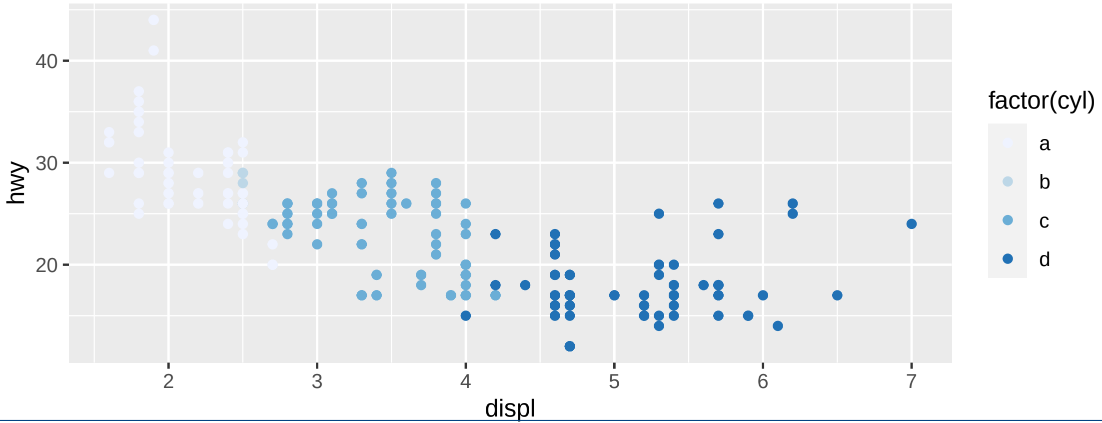
Example: Mapping and scales

. . . map scales:

x	y	colour	size	shape
0.037	0.531	#FF6C91	1	19
0.037	0.531	#FF6C91	1	19
0.074	0.594	#FF6C91	1	19
0.074	0.562	#FF6C91	1	19
0.222	0.438	#00C1A9	1	19
0.222	0.438	#00C1A9	1	19
0.278	0.469	#00C1A9	1	19
0.037	0.438	#FF6C91	1	19
0.037	0.406	#FF6C91	1	19
0.074	0.500	#FF6C91	1	19

Example: Mapping and scales

```
data(mpg)
ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) +
  geom_point() +
  scale_colour_brewer(palette = 1, labels = letters[1:4])
```



Outline

Outline

Functions

> ***Wikipedia:*** *A function is a sequence of code that performs a* ***specific task****, packaged as a* ***unit****. This unit can then be used wherever that particular task should be performed.*

A function in R can be characterized by its *arguments*, the *body* and the *environment* in which they are defined. Functions in R are objects of class *function* and have the following structure:

```
[ someFunction <- ] function( <arguments> ) {
    <expression>
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be returned by other functions
- Functions can be stored in a list
- Functions can be nested, so that functions can be defined inside of another function

Functions

A function in R is an object of class ‘function’ and contains three components:

- the body, can be accessed with `body()`
- the formals, the formal arguments list - can be accessed with `formals()`
- the environment, which determines how variables are found inside the function - can be accessed with `environment()`
- Printing a function to the console shows all three components
- If the environment is not specified the global environment is the environment of the function
- Typically you will add an environment to your functions when creating a *package* or implicitly when creating a *closure*

Functions

```
f <- function(x) x
f
```

```
## function(x) x
```

```
formals(f)
```

```
## $x
```

```
body(f)
```

```
## x
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

Functions

```
nrow
```

```
## function (x)
## dim(x)[1L]
## <bytecode: 0x0000000014785580>
## <environment: namespace:base>
```

```
formals(nrow)
```

```
## $x
```

```
body(nrow)
```

```
## dim(x)[1L]
```

```
environment(nrow)
```

```
## <environment: namespace:base>
```

Function Arguments

Functions have *named arguments* which potentially have *default* values.

- The formal arguments are the arguments included in the function definition
- Exact, positional and partial matching
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have *default* values
- Arguments in terms of other arguments
- Arguments in terms of values inside the function

Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to sd are all equivalent:

```
mydata <- rnorm(100)  ## Sample from N(0,1)
sd(mydata)            ## Compute Standard Deviation
sd(x = mydata)
sd(x = mydata, na.rm = FALSE)
sd(na.rm = FALSE, x = mydata)
sd(na.rm = FALSE, mydata)
```

Even though it is *legal*, it is not recommended messing around with the order of the arguments too much, since it will lead to confusion.

Argument Matching

Function arguments can also be partially matched, which is a potential cause for errors and confusion. It can be useful for *interactive work*, though. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Functions Arguments

Arguments in terms of other arguments:

```
someFunction <- function(x, y = x + 1) x + y
someFunction(2)
```

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

Arguments in terms of values inside the function:

```
someFunction <- function(x, y = z + 1) {
  z <- x + 1
  x + y + z
}
someFunction(2)
```

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

Defining default values in terms of values defined inside the function itself is usually bad practise since it is hard to understand the function call without reading the code of the function itself.

The “...” Argument

The ... argument can be used to pass arguments on to other function calls.

```
myplot <- function(x, y, type = "l", ...) {
    plot(x, y, type = type, ...)
}
```

Generic functions use ... so that extra arguments can be passed to methods (more on this later).

The “...” Argument

The `...` argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```
args(cat)
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

The "..." Argument

One catch with `...` is that any arguments that appear after `...` on the argument list must be named explicitly and cannot be partially matched.

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```
paste("a", "b", sep = ":")
```

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

```
paste("a", "b", se = ":")
```

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

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

```r
f <- function(a, b) {
  a^2
}
f(2)
```

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

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because 2 gets positionally matched to `a`.

Lazy Evaluation

Another example:

```
f <- function(a, b) {
  print(a)
  print(b)
}
f(45)
```

```
## [1] 45
## Error in print(b): Argument "b" fehlt (ohne Standardwert)
```

Notice that 45 got printed first and before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

Lazy evaluation

- The reason that arguments can be defined in terms of other arguments is the concept of *lazy evaluation*
- Expressions in a function are only evaluated when they are needed
- This means that before R tries to evaluate the values of the argument a new environment is created in which the evaluation takes place. At the time when the value for y is needed (when y is evaluated) x is part of the environment

```
someFunction <- function(x, y = x + 1) x + y
someFunction(2)
```

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

In combination with the scoping rules (name masking) it follows that any object outside the function (in the global environment) named x will be masked. Hence R can only find the object named x which is the value of the argument x:

```
x <- 1
someFunction(2)
```

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

Replacement Functions

- We have already seen that R can distinguish functions from other objects. The same happens for replacement functions:

```
names(x) <- newNames
```

- When evaluating the expression R will notice, that the left hand side is not a simple name but a function call
- R will then search for a function called ‘names<-’
- This implies that there are two function definitions for ‘names’, names and names<-:

```
str(names)
```

```
## function (x)
```

```
names
```

```
## function (x)  .Primitive("names")
```

```
str(`names<-`)
```

```
## function (x, value)
```

```
`names<-`
```

```
## function (x, value)  .Primitive("names<-")
```

Replacement Functions

- These functions act (syntactically) as if they modify their argument
- Internally they are ‘ordinary’ functions and are distinguished with a special naming and should return the modified input

```
"functionName<-" <- function(x, value) {
    x[subSet] <- value
    return(x)
  }
```

- Consider the following example:

```
"replacementFunction<-" <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
replacementFunction(x) <- -1
x
```

```
##  [1]  1 -1  3  4  5  6  7  8  9 10
```

Replacement Functions

- Since replacement functions are simply functions the usual function call is also valid
- However, when using special names the function name must be referred to using single quotes: '

```
'replacementFunction<-'(x, -2)
```

```
##  [1]  1 -2  3  4  5  6  7  8  9 10
```

- The above function call will not replace the object 'x'. The following expression is much closer to how R understands replacement functions:

```
x <- 'replacementFunction<-'(x, -2)
```

- Note that R only knows one way to interpret a function call and that is: `functionName(arguments)`. When evaluating replacement functions the initial expression: `replacementFunction(x) <- value` is rewritten as `x <- 'replacementFunction<-'(x, value)`. The purpose of this way to call a function is to add some *sintactic sugar*.

Replacement Functions

Often it is very useful to combine replacement and subsetting. Expressions like the following are also valid:

```
x <- c(a = 1, b = 2, c = 3)
names(x)[2] <- "two"
x
```

```
##    a  two    c
##    1    2    3
```

This is turned internally into:

```
`*tmp*` <- names(x)
`*tmp*`[2] <- "two"
names(x) <- `*tmp*`
x
```

This implies that for evaluating the above expressions two functions need to be available, names and `names<-`. For our `replacementFunction<-` this is not meaningful.

Infix functions

- Most of the time functions are used as ‘prefix’ operators
- Binary operators like `'+'`, `'-'`, `'=='` and even `'<-'` are also functions
- Functions with reserved or illegal names can be referred to using single quotes: ’
- You can create your own infix operators using the symbol ‘%’ - there are already predefined functions in R using this structure: `%*%`, `%/%`, etc.
- Note that you have to use double quotes to be able to assign values (the function definition) to these variable names - see the examples

Example: Infix Functions

Don’t try this at home!!!

```
1 + 2
```

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

```
"+" <- function(x, y) x - y
1 + 2
```

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

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

This is why we have to protect ourselves with package creation and namespaces!

Example: Infix Functions

A more useful example: Create an operator for pasting strings together!

```
"%+%" <- function(x, y) paste(x, y)

"some" %+% "string"
```

```
## [1] "some string"
```

The naming of functions in this context is more flexible: You are allowed to use all symbols except %.

```
"%paste%" <- function(x, y, ...) paste(x, y, ...)
"new" %paste% "string"
```

```
## [1] "new string"
```

Example: Infix Functions

Note the following function calls are equivalent.

```
1 + 2
```

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

```
'+'(1, 2)
```

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

and

```
"new" %paste% "string"
```

```
## [1] "new string"
```

```
'%paste%'("new", "string")
```

```
## [1] "new string"
```

and also

```
x <- 1
'<-'(x, 1)
x
```

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

Homework: Infix Functions

As homework: Explain the following function and result!

```
"%-%" <- function(a, b) paste("(", a, " %-% ", b, ")", sep = "")
"a" %-% "b" %-% "c"
```

```
## [1] "((a %-% b) %-% c)"
```

Write an infix function %and% that behaves like the logical operator &, except that it won't work on vectors. Use only the if-else control structure and the function '=='!

```
TRUE %and% TRUE
```

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

```
FALSE %and% TRUE
```

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

```
TRUE %and% FALSE
```

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

```
FALSE %and% FALSE
```

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

Functions inside other functions

- Functions can be defined anywhere - in any environment
- Function definitions inside of other functions will define where a defined function is available - the scoping rules apply as for any other object
- If function ‘a’ is defined inside function ‘b’ function ‘a’ will only be available during the evaluation of function ‘b’

```
b <- function(x) {
  a <- function(y) y^2
  a(x)
}
b(2)
```

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

```
exists("a")
```

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

Outline

Scoping Rules - Outline

- The scoping rules is the set of rules which define how R finds objects
- **Lexical scoping** and **dynamic lookup** are the two concepts discussed in the following
- **Lexical scoping** summarizes the aspects covered by:
  1. The Search Path
  2. Name Masking
  3. Functions vs. Variables and determines *how* objects are found
- **Dynamic lookup** determines *when* to look for an object

Exkursus: Environments

- An environment binds a set of names to values
- Environments are objects and can have a name and be stored in lists etc.
- Environments are very similar to lists. Three fundamental differences:
  1. Environments have reference semantics - Whenever you modify an environment, you modify every copy
  2. Environments have parent - if an object is not found in an environment, R will continue to search in the parent, which is an environment and thus has a parent . . .
  3. Every object in an environment must have a name and the names must be unique
- Environments can be useful data structures - however reference objects should only be used with great care (in R). Most R users are not familiar with reference semantics or do not expect them (in R)
- Typically you will only work implicitly with environments (all the time!)

Exkursus: Environments

Environments are objects!

```
e <- new.env()
e$x <- 1
e$x
```

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

```
ls(e)
```

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

And similar to lists!

```
a <- list(x = 1, y = 2)
f <- as.environment(a)
f$y
```

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

```
ls(f)
```

```
## [1] "x" "y"
```

Exkursus: Environments - reference vs. copy-on-modify

For most objects in R the ‘copy-on-modify’ semantics are applied. Meaning there is no true modification possible only replacements. Every time an object is modified it is copied first, then modified and then replaces the existing object.

```
a <- list(a = 1)
b <- a
b$b <- 2
a
```

```
## $a
## [1] 1
```

Environments are reference objects. What we use as objects are pointers to the actual values. Copying an environments means copying the pointer, not the value!

```
e <- new.env()
e$x <- 1
f <- e
f$y <- 2
e$y
```

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

Exkursus: Environments - parent

If a parent is not explicitly set, the default is the environment in which it is created.

```
f <- new.env()
f$x <- 1
parent.env(f)
```

```
## <environment: R_GlobalEnv>
```

The parent can also be set explicitly:

```
e <- new.env(parent=f)
e$y <- 2
parent.env(e)
```

```
## <environment: 0x000000001e2bdec8>
```

```
f
```

```
## <environment: 0x000000001e2bdec8>
```

Exkursus: Environments - parent

Every environment has a parent. If an object can not be found in a specific environment the ‘search’ will be continued in the parent:

```
exists("x", envir = f)
```

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

```
exists("y", envir = e)
```

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

```
exists("x")
```

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

```
exists("x", envir = e)
```

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

```
exists("y", envir = f)
```

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

Exkursus: Environments - names

- Every object in an environment must have a name and the names must be unique
- Sub-setting with position and logical vectors is not possible for environments - only with names

```
# Try this on your own
f <- as.environment(list(a = 1, b = 2))
e <- as.environment(list(a = 1, 2))
f$a
f[["a"]]
f[1]
f[[1]]
```

The Search Path

- Every evaluation in R is made in a specific environment
- The workspace is the ‘global environment’. It can be accessed using `globalenv()` or using the build-in object `.GlobalEnv`
- Try: `ls(.GlobalEnv)` to see what is in your workspace. The names should correspond to what you see in RStudio
- The workspace typically contains the objects you initialized in your current session - the question is then how R can find the build-in functions (like `dim()` since it is not in your workspace)
- To omit the explicit reference to a package from which we want to use a function, the search mechanism for environments is applied

The Search Path

- Since the workspace, the global environment, is an environment it has a parent. Typically this is the last package loaded, which itself is again an environment
- The parent of the last package loaded is the package loaded before
- The connection via the parent between the loaded packages is the *search path*
- The search path is used to determine the order in which the loaded packages are searched to find a function or any other object
- The 'end' of the search path is `package:base` (`baseenv()`)
- The parent of `package:base` is the empty environment, `emptyenv()`, which itself has no parent, is empty and can not contain any objects: the search ends here

The Search Path

```
search()
```

```
##  [1] ".GlobalEnv"        "package:ggplot2"   "package:dplyr"
##  [4] "package:knitr"     "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "package:methods"   "Autoloads"         "package:base"
```

```
library(ggplot2)
search()
```

```
##  [1] ".GlobalEnv"        "package:ggplot2"   "package:dplyr"
##  [4] "package:knitr"     "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "package:methods"   "Autoloads"         "package:base"
```

The Search Path

Searching for objects - the function dim

```
funOfInterest <- "dim"
loadedEnv <- search()
objInEnv <- sapply(as.list(loadedEnv),
  function(env, ...) exists(envir = as.environment(env), ...),
  x = funOfInterest, inherits = F)
loadedEnv[objInEnv]
```

```
## [1] "package:base"
```

Make it a function:

```
searchForFun <- function(funOfInterest) {
  loadedEnv <- search()
  objInEnv <- sapply(as.list(loadedEnv),
    function(env, ...) exists(envir = as.environment(env), ...),
    x = funOfInterest, inherits = F)
  loadedEnv[objInEnv]
}
```

The Search Path

Finding functions:

```
searchForFun("dim")
```

```
## [1] "package:base"
```

```
searchForFun("plot")
```

```
## [1] "package:graphics" "package:base"
```

```
searchForFun("qplot")
```

```
## [1] "package:ggplot2"
```

```
x <- 1
searchForFun("x")
```

```
## [1] ".GlobalEnv"
```

Name Masking

Whenever there is more than one object with the same name, the object in the 'closest' environment will be selected. An environment is closer if it appears earlier in the search path.

```
dim
```

```
## function (x)  .Primitive("dim")
```

```
dim <- function(x) x
searchForFun("dim")
```

```
## [1] ".GlobalEnv"   "package:base"
```

```
dim
```

```
## function(x) x
```

Here two functions with the same name can be found. Whenever you call the function `dim`, R will use the version in your workspace. The function `dim` in the base package is not overritten, it is simply masked by the function in the workspace.

Excursus: Environments and functions

- Every call to a function creates a temporal environment in which the function call is evaluated
- This guarantees (potentially) that the evaluation does not affect the global environment - your workspace
- This also determines how names are looked for inside a function (we only consider user written functions) - if not specified differently the environment is the global environment plus what is passed into the function in form of *formal arguments* and objects created inside the function
- Objects created inside of a function and passed to the function by *formal arguments* are called *local* and are only available during the function call

Excursus: Environments and functions

The environment of `nrow` guarantees that `nrow` will allways find the 'correct' function named `dim` which is defined in the base package.

```
x <- data.frame(V1 = 1:2)
str(x)
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ V1: int  1 2
```

```
dim <- function(x) 1
newNrow <- function(x) dim(x)[1L]
nrow(x)
```

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

```
newNrow(x)
```

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

Excursus: Environments and functions

Consider the following example:

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

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

What is the value of x?

Excursus: Environments and functions

Consider the following example:

```
x <- 1
someFunction <- function(x) {
  y <- x + 1
  y
}
someFunction(x)
```

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

What is the value of y?

Excursus: Environments and functions

Consider the following examples:

```
x <- 1
someFunction <- function() x
```

What is the result of a call to `someFunction`?

```
rm(list=ls())
x <- 1
someFunction <- function() x * y
```

What is the result of a call to `someFunction`?

Excursus: Environments and functions

The same rules apply if you define functions inside other functions:

```
x <- 1
someFunction <- function() {
  y <- 2
  otherFunction <- function() {
    z <- 3
    c(x, y, z)
  }
  otherFunction()
}
```

What is the result of a call to `someFunction`?

Excursus: Environments and functions - A fresh start

What is the result of the second call to `someFunction`?

```
rm(list = ls())
someFunction <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  print(a)
}
```

Every time a function is called a new environment is created for the evaluation. The object a will only be created locally.

Excursus: Environments and functions

More name masking as homework: What is the return value of `f(10)`?

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x ^ 2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

Functions vs. variables

If you search for an object from a context from which it is obvious that you are searching a function (a function call like `someFunction(2)` the search is continued until R finds a *function* ignoring other objects with the same name. Thus the following code is valid:

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

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

Be aware that the lookup for the value 'c' and the function 'c' is possible since both objects named 'c' are defined in different environments. This is not possible when working in the global environment (names in an environment must be unique!):

```
f <- 10
f <- function(x) x
f(f)
```

```
## function(x) x
```

Of course this may get very confusing, proper naming is preventive ... If we wanted to rewrite the function `searchForFun` defined previously in order to only search for functions, we would have to use `match.fun` instead of `exists`. Try this at home!

Dynamic lookup

- Lexical scoping determines **where** to look for values, not **when** to look for them
- R will search for objects when a function is run, not when it is created
- This means that a function can have different return values which depend on objects defined in the environment in which it is created *and* run
- This means an important property of *well defined* functions is violated - *self containment*
- The property of self containment is fulfilled if the return value of a function depends only on the input values (formal arguments)
- You want to write self contained functions. If you don't it is very easy to produce unexpected results

Dynamic lookup

R will search for objects when a function is run, not when it is created

```
rm(list=ls())
someFunction <- function(x) x + y
```

The above expression is valid since R will not try to find the value of y until `someFunction` is called. Depending on the value of y the function `someFunction` will have a different return value:

```
y <- 1
someFunction(1)
```

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

```
y <- 2
someFunction(1)
```

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

In this definition `someFunction` is not self containing - such behaviour should be avoided if possible

Outline

Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if, else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop while a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an interation of a loop

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresissons.

Control Structures: if

```
if(<condition>) {
  <do something>
} else {
  <do something else>
}

if(<condition1>) {
  <do something>
} else if(<condition2>) {
  <do something different>
} else {
  <do something even more different>
}
```

if (cont.)

This is a valid `if/else` structure.

```
if(x > 3) {
   y <- 10
} else {
   y <- 0
}
```

So is this one.

```
y <- if(x > 3) {
        10
    } else {
        0
    }
```

`if` (cont.)

Of course, the `else` clause is not necessary.

```
if(<condition1>) {

}
if(<condition2>) {

}
```

Control Structures: `for`

`for` loops take a *loop variable* and assign it successive values from a sequence or a vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
for(i in 1:10) {
    print(i)
}
```

This loop takes the `i` variable and in each iteration of the loop gives it the successive values contained in vector `1:10` and then exits.

for (cont.)

These four loops have the same behavior.

```
x <- c("a", "b", "c", "d")
for(i in 1:4) {
    print(x[i])
}

for(i in seq_along(x)) {
    print(x[i])
}

for(letter in x) {
    print(letter)
}

for(i in x) print(i)
```

Nested `for` loops

`for` loops can be nested

```
x <- matrix(data = 1:6, ncol = 2, ncol = 3, byrow = TRUE))

for(i in seq_len(nrow(x))) {

    for(j in seq_len(ncol(x))) {

        print(x[i, j])
    }
}
```

Be careful with nesting. Nesting beyond 2-3 levels is often very diffcult to read/understand.

Control Structures: `while`

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
number <- 0

while(number < 10) {
    print(number)
    number <- number + 1
}
```

While loops can potentially result in infinite loops if not written properly. Use with care!

while (cont.)

Sometimes there will be more than one condition in the test.

```
z <- 5

while(z >= 3 && z <= 10) {
    print(z)
    coin <- rbinom(n = 1, size = 1, prob = 0.5)

    if(coin == 1) { ## random walk
        z <- z + 1
    } else {
        z <- z - 1
    }
}
```

Conditions are always evaluated from left to right.

Control Structures: `repeat`

`repeat` initiates an infinite loop. The only way to *exit* a repeat loop is to call `break`.

```
x0 <- 1
tol <- 1e-8

repeat {
    x1 <- computeEstimate(x0)

    if(abs(x1 - x0) < tol) {
        break
    } else {
        x0 <- x1
    }
}
```

This loop is a bit dangerous because there is no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

next

`next` is used to skip an iteration of a loop

```
for(i in 1:100) {
    if(i <= 20) {
        ## Skip the first 20 iterations
        next
    }
    ## Do something here
}
```

Control Structures

Summary

- Control structures like `if`, `while` and `for` allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if they are theoretically correct
- Control structures mentiond here are primarily useful for writing programs;
- For command-line interactive work and "advanced" code, the `*apply` functions are more useful and advisable.

Outline

The *apply Family

Writing `for`, `while` loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

- `lapply` Loop over a list and evaluate a function on each element
- `sapply` Same as `lapply` but tries to simplify the result
- `mapply` Multivariate version of `lapply`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

lapply

lapply takes three arguments: a list X, a function (or the name of a function) FUN, and other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x0000000012cbc3e0>
## <environment: namespace:base>
```

The actual looping is done internally in C code which makes it fast.

lapply

lapply always returns a list, regardless of the class of the input

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] -0.2453451
```

lapply

```r
x <- list(a = 1:4,
          b = rnorm(10),
          c = rnorm(20, 1),
          d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] -0.2932513
##
## $c
## [1] 0.4580873
##
## $d
## [1] 5.016555
```

lapply

```
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.9322158
##
## [[2]]
## [1] 0.4196383 0.9854747
##
## [[3]]
## [1] 0.8891253 0.1510683 0.3791930
##
## [[4]]
## [1] 0.2054858 0.6422853 0.5685747 0.3942860
```

lapply

```
x <- 1:4
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 1.765009
##
## [[2]]
## [1] 2.737756 9.296052
##
## [[3]]
## [1] 9.591474 7.188546 7.510993
##
## [[4]]
## [1] 6.2833597 0.6523421 8.3920019 5.3698889
```

lapply

```
x <- 1:4
lapply(x, runif, min = 0, n = 10)
```

```
## [[1]]
##  [1] 0.4572506 0.4401416 0.9986626 0.4487867 0.1699146 0.9577954 0.2149442
##  [8] 0.4349542 0.3240849 0.2701586
##
## [[2]]
##  [1] 1.8448817 1.3844085 1.4017337 1.8072580 1.3658769 1.1505822 1.4819859
##  [8] 1.3712049 0.1364364 1.2625441
##
## [[3]]
##  [1] 1.60577587 0.07595627 2.63406050 2.30868168 0.46173406 2.54037271
##  [7] 2.33542617 1.30265663 0.40360018 0.37334548
##
## [[4]]
##  [1] 2.33340091 0.20760196 0.32999750 0.70647803 3.35921669 1.35389873
##  [7] 3.25587472 0.08428678 3.66237487 3.51845962
```

lapply

lapply and friends make heavy use of *anonymous functions*.

```
x <- list(a = matrix(data = 1:4, nrow = 2, ncol = 2, byrow = FALSE),
          b = matrix(data = 1:6, nrow = 3, ncol = 2, byrow = TRUE))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

lapply

An anonymous function for extracting the first column of each matrix.

```
lapply(x, function(y) y[ ,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 3 5
```

`sapply`

`sapply` will try to simplify the result of `lapply` if possible.

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned.
- Check the manual for the 'sapply' argument 'simplify'.

sapply

```
x <- list(a = 1:4,
          b = rnorm(10),
          c = rnorm(20, 1),
          d = rnorm(100, 5))
sapply(x, mean)
```

```
##          a          b          c          d
## 2.50000000 0.06344584 0.97608739 4.92705213
```

```
mean(x)
```

```
## Warning in mean.default(x): Argument ist weder numerisch noch boolesch: gebe NA
## zurück
## [1] NA
```

split

split takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

- `x` is usually a data frame
- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates if empty factor levels should be dropped
- `...` further potential arguments passed to methods.

Splitting a Data Frame

```
head(airquality, n=3)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
```

```
s <- split(airquality[, c("Ozone", "Solar.R", "Wind")],
         f = airquality$Month)
sapply(s, colMeans)
```

```
##                5         6          7        8        9
## Ozone         NA        NA         NA       NA       NA
## Solar.R       NA 190.16667 216.483871       NA 167.4333
## Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```
sapply(s, colMeans, na.rm=TRUE)
```

```
##                 5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

split

Another common idiom is to split-apply-and-combine data.

```
dataList <- lapply(split(x, f), quantile) #Split-Apply
do.call(cbind, dataList) #Combine the result
```

```
##               1         2          3
## 0%   -1.6313849 0.0482403 -0.1348567
## 25%  -0.5700407 0.4116411  0.3270585
## 50%  -0.3089372 0.5311908  1.1472963
## 75%   0.8533024 0.7680078  1.5822608
## 100%  1.3578907 0.9974186  2.0261817
```

split

This is equivalent to:

```
sapply(split(x, f), quantile)
```

```
##               1         2          3
## 0%   -1.6313849 0.0482403 -0.1348567
## 25%  -0.5700407 0.4116411  0.3270585
## 50%  -0.3089372 0.5311908  1.1472963
## 75%   0.8533024 0.7680078  1.5822608
## 100%  1.3578907 0.9974186  2.0261817
```

However, `sapply` automates the “combine’ ’-step - it may not allways be what you expect!

Use `do.call(FUN, list)` to control the output structure.

supplement strsplit, lapply and do.call

Split a character vector and recombine the results as a data.frame

```
names <- c("Adam Riese", "Albert Einstein", "Oliver Kahn")
namesList <- strsplit(names, split = " ")
str(namesList)
```

```
## List of 3
##  $ : chr [1:2] "Adam" "Riese"
##  $ : chr [1:2] "Albert" "Einstein"
##  $ : chr [1:2] "Oliver" "Kahn"
```

```
namesList <- lapply(namesList,
                    function(x) data.frame(firstName = x[[1]],
                                           familyName = x[[2]]))
do.call(rbind, namesList)
```

```
##   firstName familyName
## 1      Adam      Riese
## 2    Albert   Einstein
## 3    Oliver       Kahn
```

mapply

mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
str(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

FUN	is a function to apply
...	contains arguments to apply over
MoreArgs	is a list of other arguments to FUN
SIMPLIFY	indicates whether the result should be simplified
USE.NAMES	if the first ... argument has names, use that names

mapply

```r
## the hard way
lHard <- list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))

## the smart way
lSmart <- mapply(rep, 1:4, 4:1)
lSmart
```

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

Vectorizing a Function

```r
noise <- function(n, mean, sd) {
  rnorm(n, mean, sd)
}
noise(5, 1, 2)
```

```
## [1]  4.1095555  3.5505077 -2.5743942  0.2007773  1.6858285
```

```r
noise(1:4, 1:4*10, 2)
```

```
## [1]  8.948993 22.395396 30.100440 37.631934
```

Instant Vectorization

```
set.seed(15061969)
mapply(noise, 1:4, 1:4*10, 2)
```

```
## [[1]]
## [1] 7.44371
##
## [[2]]
## [1] 19.65748 18.59855
##
## [[3]]
## [1] 30.27678 32.84333 27.59776
##
## [[4]]
## [1] 39.69425 39.72227 39.95984 40.35376
```

Instant Vectorization

Which is the same as

```
set.seed(15061969)
list(noise(1, 10, 2), noise(2, 20, 2),
     noise(3, 30, 2), noise(4, 40, 2))
```

```
## [[1]]
## [1] 7.44371
##
## [[2]]
## [1] 19.65748 18.59855
##
## [[3]]
## [1] 30.27678 32.84333 27.59776
##
## [[4]]
## [1] 39.69425 39.72227 39.95984 40.35376
```

Vectorizing a Function internally

```r
noise <- function(n, mean, sd) {
  mapply(rnorm, n, mean, sd)
}
set.seed(29081975)
noise(1:4, 1:4*10, 2)
```

```
## [[1]]
## [1] 6.873807
##
## [[2]]
## [1] 24.88267 18.17274
##
## [[3]]
## [1] 29.15017 29.93314 28.98341
##
## [[4]]
## [1] 42.12792 39.37629 37.73670 39.20634
```

Remark

`*apply` vs `for` loops

- `*apply` functions are fast
- there are special packages so that `*apply` functions can be calculated parallel on multicore systems → super-fast
- **but:**
  each element is calculated separately; calculations that depend on the outcome of previous elements are not possible with `*apply` but with `for` loops.

Outline

Motivation

- Reusable code: Why don't we just write scripts?
- Documentation: dedicated and structured documentation vs. comments inline
- Stability: which functions from which packages do your functions need?
- Share your code with others
- Automated checks (and customized tests)

Creating a new package

- To initialize a package, use `package.skeleton()`, `devtools::create()` or the RStudio IDE
- Must have ingredients:
  - Folder “R” for *.R* files
    - File ending: *.R* or *.r*
    - All R objects defined in these files are part of the package
    - Typically only functions (data is stored somewhere else)
  - DESCRIPTION file
    - Contains basic information about your package
    - Package, Version, License, Description, Title, Author, Maintainer
    - Edited by hand
  - NAMESPACE file
    - Which functions does the package provide, which functions does it need?
    - Will be created automatically later on
- Useful: *.Rproj* file with RStudio project

DESCRIPTION

```
Package: myPackage
Title: Examples for package creation
Version: 0.1
License: GPL
Description: This package demonstrates package creation.
    This includes documentation and the namespace.
Author: Mira Klein
Maintainer: <mira.klein@inwt-statistics.de>
```

Writing a new package function

```
greet <- function() {
  "Hello"
}
```

To build the package, click on “Install and Restart”.

Sharing a package

When your package is finished, you can share it in one single file:

- For Windows: “Build” → “More” → “Build **Binary** Package”
- For Linux: “Build” → “More” → “Build **Source** Package”

Documentation

- In a package, all *exported* objects have to be documented.
- Documentation files are *.Rd* files located in the folder “man”.
- Those files are used to build a HTML and PDF documentation (e.g., what you see when you type `?mean`)
- Don’t create the *.Rd* files manually: use the package `roxygen2`
- To use `roxygen2` for documentation, first place a tick mark under “Tools” → “Project options” → “Build Tools” → “Rd Files”)

All lines starting with `#'` are processed by `roxygen2`.

```
#' @title Say hello
#' @description This function says hello in a nice way.
greet <- function() {
  "Hello"
}
```

Documentation

Documentation of function arguments:

```
#' @title Say hello
#' @description This function says hello in a nice way.
#' @param who character: name of person you want to greet
#' @examples greet("Tim")
greet <- function(who) {
  paste("Hello", who)
}
```

Mentioning the class (“character”) is not mandatory, but helpful.

Documentation: Exercise

Save the following function in a file called “graphics.R” in the “R” folder:

```r
histoNorm <- function(sampleSize, title, colour = "black") {
  x <- rnorm(n = sampleSize)
  hist(x, col = colour, main = title)
}
```

Write a documentation for the function.

Documentation: Solution

```r
#' @title Plot histogram from simulation
#' @description Plot a histogram for normally distributed simulated
#' random values
#' @param sampleSize integer: Number of data points
#' @param title character: Plot title
#' @param colour character: Colour of the histogram
histoNorm <- function(sampleSize, title, colour = "black") {
  x <- rnorm(n = sampleSize)
  hist(x, col = colour, main = title)
}
```

Namespace

- The NAMESPACE file determines which functions your package uses and provides.
- Which functions does your package provide (“Exports”)?
  - roxygen2 comment: `#' @export`
- Which functions from other packages does your package need (“Imports”)?
  - roxygen2 comment: `#' @importFrom packagename fun1 fun2 fun3`
- Adjust project options to create namespace with `roxygen2`: “Tools” → “Project options” → “Build Tools” → “Configure” → “Namespace”
- Don’t forget to list all required packages in the DESCRIPTION file under “Imports”

Namespace: Example

This new function uses functions from the ggplot2 package:

```
#' @title Plot histogram from simulation with ggplot2
#' @description Plot a histogram for normally distributed random
#' values
#' @param sampleSize integer: Number of data points
#' @param title character: Title of the plot
#' @param colour character: Colour of the histogram
histoNormGg <- function(sampleSize, title, colour = "black") {

  dat <- data.frame(var1 = rnorm(n = sampleSize))

  ggplot(dat) +
    geom_histogram(aes(var1),
                   fill = colour,
                   col = NA,
                   binwidth = 0.5) +
    labs(title = title)
}
```

Namespace: Example

This code will create a proper namespace:

```
#' @title Plot histogram from simulation
#' @description Plot a histogram for normally distributed random
#' values
#' @param sampleSize integer: Number of rows in the data
#' @param title character: Title of the plot
#' @param colour character: Colour of the histogram
#' @importFrom ggplot2 aes_string geom_histogram ggplot labs
#' @importFrom stats rnorm
#' @export
histoNormGg <- function(sampleSize, title, colour = "black") {

  dat <- data.frame(var1 = rnorm(n = sampleSize))

  ggplot(dat) +
    geom_histogram(aes_string("var1"),
                   fill = colour,
                   col = NA,
                   binwidth = 0.5) +
    labs(title = title)
}
```

Namespace

Why don't we just use `library()`?

- Naming conflicts: Two packages could provide different functions with the same name
- Order of the `library()` statements would be important

Imports in Namespace vs. Description

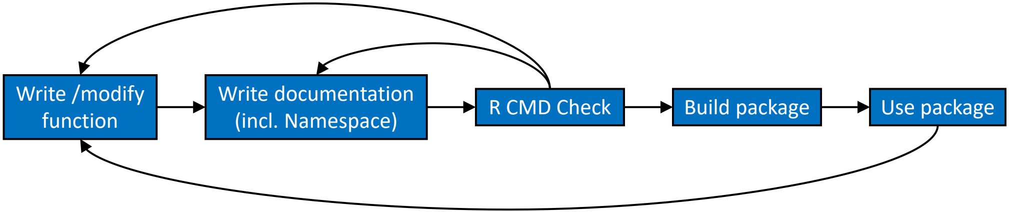
Clarification:

- Roxygen comments (`@importFrom packagename fun1 fun2 ...`) control which functions from which packages are *used*
- DESCRIPTION file
  - Imports in the DESCRIPTION file control which packages need to be *installed* to use this package
  - Depends: Packages listed in “Depends” are attached to the search path when your package is loaded. (This should be avoided because of potential naming conflicts.)

R CMD Check

- Performs many basic checks
- Prerequisite to publish packages on CRAN – but also very useful for private use!
- Examples:
  - Are there objects in the functions that where neither defined nor passed to the function?
  - Are there any problematic characters?
  - Is the documentation complete?
  - Is the namespace properly defined?
  - Are all required packages (Imports, Depends) installed on your PC?
  - Are there any files in the project folder which are not related to the package?
  - Further potential problems in the code, e.g., `library()` statements
  - ...

Recap: Package development workflow



Resources

- Wickham (2017): *R Packages* (online, free, covers all important aspects, not too long)
- Leisch (2009): *Creating R Packages: A Tutorial*
- Chambers (2008): *Trustworthy Software: The Prime Directive* (goes more into detail)

Outline

S3: Motivation

- Let's assume we want to define a mean for the *character* type in R.
- As mean we simply define the mean length of each element in a vector of type *character*. The length of an element can be defined as the number of characters.

```
characterVector <- c("some", "more", "text", "and", "different", "nchar")
meanCharacter <- function(x) mean(nchar(x))
meanCharacter(characterVector)
```

```
## [1] 4.833333
```

- There is one obstacle with the definition of `meanCharacter`, the name. With every function we define we have to memorize one more function name and also we have to find new useful names (which is not easy!).
- How is a potential user (you in 2 weeks) ever going to understand and know about all the functions you defined? Not at all.

S3: Motivation

- If the name is a problem let's define a new mean function.
- One thing to keep in mind, however, is that we should preserve the behaviour of the original mean function.

```
mean <- function(x) {
  if (is.character(x)) {
    base::mean(nchar(x))
  } else {
    base::mean(x)
  }
}
mean(characterVector)
```

```
## [1] 4.833333
```

- Every time we want to define a mean function for a new data type we have to add more if clauses to the definition. In the long run this strategy is going to be a mess.
- Also we can not extend functionality, we have to change it. Extending is good, changing is bad.

S3: Basic idea

- The basic idea of the S3 class system is that it should be possible to extend the functionality of a generic vocabulary without adding new words (function names).
- This is a convenience. You can throw any statistical model and data type into the `summary` function. And somehow (allmost) for all data types (linear models, data frames, etc.) `summary` knows what to do.
- Even more, `summary` knows about data types from different packages, although the original author had no way of anticipating these types.
- To understand this we need to answer the three following questions:
  - What is a S3 class?
  - What is a generic function?
  - What is a method?

What is a S3 class?

- S3 classes serve a simple purpose, to give different or new data types a name.
- This is done by adding an attribute to the data with the name of the class.

```
class(1:10)
```

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

```
class("a")
```

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

```
dat <- list(c(1, 2), c(2, 3), c(2, 4))
class(dat) <- "rational"
str(dat)
```

```
## List of 3
##  $ : num [1:2] 1 2
##  $ : num [1:2] 2 3
##  $ : num [1:2] 2 4
##  - attr(*, "class")= chr "rational"
```

More on S3 classes

- Typically you do not assign the class *interactively* but you return data with a class attribute from a constructor function (e.g. `numeric`, `list`, `lm`).

```
rational <- function(num, denom) {
    rat <- mapply(c, num, denom, SIMPLIFY = FALSE)
    class(rat) <- "rational"
    rat
}
str(rational(c(1, 2, 2), c(2, 3, 4)))
```

```
## List of 3
##  $ : num [1:2] 1 2
##  $ : num [1:2] 2 3
##  $ : num [1:2] 2 4
##  - attr(*, "class")= chr "rational"
```

Even more on S3 classes

- `rational` is a *constructor* function for instances of class *rational*.
- It is named *constructor* because it knows how to construct an object of class *rational*.
- There is no formal definition of the class, you simply say a list is of class *X*.
- In most scenarios the *list* type is used as the basic data structure to compose new data (e.g. `data.frame` and `lm`).
- So what is the great benefit of defining new classes? An example: Printing an object of class *rational* to the console results in verbose information. Now we can fix this:

```
print.rational <- function(x, ...) cat(sapply(x, paste, collapse = "/"))
rational(c(1, 2, 2), c(2, 3, 4))
```

```
## 1/2 2/3 2/4
```

Generic functions and methods

- A generic functions is a function which only purpose is to find the appropriate method given the class of its first argument.
- The overall purpose is, that you have a function name, say `mean`, and this function somehow figures out how the mean is defined for a given data type.
- The function `print` and `mean` are generic functions. `print` will find the correct print method to print things to the console. And `mean` searches for the correct mean method for a given data type.
- Methods are defined by a naming convention: `<generic>.<class>`

```
print.rational <- function(x, ...) cat(sapply(x, paste, collapse = "/"))
```

- We say `print.rational` is the print method for objects of class *rational*. There is a print method for most of the data types in R which define how output is printed to the console.

More on generic functions

- You are not restricted on a given set of generic functions, you can define them.

```
mean <- function(x, ...) UseMethod("mean")
```

- This is it. `UseMethod` is used to start searching for methods. It will search for a function called `mean.<class(x)>` and passes the arguments to this method. This *search* is also called *method dispatch*.
- If no method for a given class can be found a *default* method is called if it exists:

```
mean("a")
```

```
## Warning in mean.default("a"): Argument ist weder numerisch noch boolesch: gebe
## NA zurück
## [1] NA
```

- You define default methods with a naming convention: `<generic>.default`

More on the mean

- We return to our initial example of defining means. We discovered that `if-else` constructs may be problematic when the number of branches can grow. S3 offers a solution:

```
mean.character <- function(x, ...) mean(nchar(x), ...)
mean.rational <- function(x, ...) mean(sapply(x, function(e) e[1] / e[2], ...))
mean(characterVector)
```

```
## [1] 4.833333
```

```
num <- 1:6
denom <- 11:16
mean(rational(num, denom))
```

```
## [1] 0.2470654
```

Conventions

- Defining generics: `generic <- function(<args>) UseMethod("generic")`
- Defining methods: `generic.<class> <- function(<args>) ...`
- Defining default method: `generic.default <- function(<args>) ...`
- The arguments of methods need to include all arguments of the generic even if they are not used. Methods can have more arguments than the generic, though.
- Defining a class:

```
myClass <- function(<args>) {
    ...
    class(out) <- "myClass"
    out
}
```

Final remarks

- The S3 class system was introduced in version 3 of the S software, hence its name.
- Essentially it is a naming convention.
- Methods are associated with generic functions (`<generic>.<class>`).
- S3 classes are defined by adding an attribute to any object in R.
- It is a special form of object-orientation, but not to be confused with other implementations in other languages which can be very different!
- One of the main benefits of the S3 system is that you do not have to remember to many function names but can rely on a generic vocabulary.
- To find out more about which generic functions exist see the help pages for `.S3methods`.

More final remarks

- Like in many other languages there was a need to extend the simple class system of S3.
- Hence a new version of the S software (version 4) introduced a novel system for object orientation, S4.
- Main features in contrast to S3 are that S4 has formal class definitions. So if you see an object of class *lm* you can be sure about its properties. Method dispatch for more than one argument is possible in S4, in S3 only the first is used.
- Also there was a need in the community to support a system similar to languages like Java. This is implemented in the function `setRefClass`.
- The R community absolutely does not agree on how object-orientation should be implemented. An indication for that is the variety of packages on CRAN which implement different class systems (e.g. `methods`, R6, `R.oo` and `proto` are frequently used).
- Should you only plan to do interactive analysis in R then it is save to say that it is sufficient to know about S3.

Outline

Functional Programming - Outline

At it's heart R is a Functional Programming Language. This means that functions are First Class Citizens, i.e. functions can be treated like any other object in R. The following will address the implications of this statement:

- Functions can be defined without a name (anonymous function)
- Functions can be defined anywhere, including inside other functions
- 'Functionals' - Like any other value, they can be passed as parameters to functions
- 'Closure' - Like any other value, they can be returned as results from functions
- Like any other value, they can be stored in lists
- As for other values, there exists a set of operators to compose functions

Function composition, the last bullet, is an important technique in functional programming. However, this is beyond the scope of this course. Understanding all other implications is fundamental to function composition and will already enable you to accomplish any task you'll ever encounter (in fewer lines).

Higher-Order-Functions

1. Functions which take functions as parameters - functional
2. Functions which return functions - closure

Functional:

```
functional <- function(f, ...) f(...)
functional(mean, runif(1e3))
functional(sum, runif(1e3))
```

Closure:

```
closure <- function(x) function(y) x + y
closure(1)(2)
closure(2)(3)
```

Outline

Evaluation - Outline

-

Homework: Evaluation

Rewrite the following expressions as it has been done for line 1 to 2:

```
x <- 1:10
'<-'(x, ':'(1, 10))
```

1. `(1:10)[2]`
2. `sum(1+1:10)`
3. `1:10^2`
4. `x <- 1:10+2^2`

The rewritten expressions can be said to be evaluated from left to right. The functions which you can rewrite are `'('`, `':'`, `'['`, `'+'`, `'^'` and `'<-'`. Check your results in R. What is meant by the phrase: "Everything in R is a call to a function!"?

Outline

Overview

Debugging in general:

- running a program step by step
- pausing a program to examine the current state
- tracking the values of some variables
- modify the program while it is running

Since R is an *interpreter* language, debugging in R means **debugging functions**:

- running a *function* step by step
- pausing a *function* to examine the current state
- tracking the values of some variables inside[4] *functions*
- modify the *function* while it is running

[4] variables that (only) exist inside the environment of the function

Debugging in R

Functions for debugging (`package: base`):

`traceback`	prints the call stack of the last uncaught error
`debug`	flags a function for debugging
`browser`	interrupts the execution of an expression and allows the inspection of the environment
`trace`	allows to insert debugging code into an existing function
`recover`	allows to browse directly on ***any*** of the currently active function calls

Motivation

As with programs written in any other language, functions written in R can contain unforseen problems which lead to failure.

The **purpose of the debugging** tools is to help the programmer find these problems quickly and effciently.[1]

[1] Roger Peng, *An Introduction to the Interactive Debugging Tools in R*, (2002)

Problem Reporting in R

Two kind of problems:

warnings do not halt the execution of a function.

“Something unusual happened during the execution of this function, but the function was nevertheless able to execute to completion.”

errors are problems that are fatal and result in a complete halt in the execution, because the function simply cannot execute to completion due to the problem.

Problem Reporting in R

Example:

```
> message <- function(x) {
+ if(x > 0)
+ print("Hello")
+ else
+ print("Goodbye")
+ }
> x <- log(-1)
Warning message:
In log(-1) : NaNs produced
> message(x)
Error in if (x > 0) print("Hello") else print("Goodbye") :
missing value where TRUE/FALSE needed
```

General remark: use robust code that checks for input errors.

Debugging Tools: `traceback`

The `traceback` function prints the list of functions which were called before the error occurred.

```
> x <- log(-1)
Warning message:
In log(-1) : NaNs produced
> message(x)
Error in if (x > 0) print("Hello") else print("Goodbye") :
missing value where TRUE/FALSE needed
> traceback()
1: message(x)
```

`traceback` shows in which function the error occurred. Since only one function was in fact called, this information is not very useful.

Debugging Tools: traceback - Example 2

```
> f <- function(x) x - g(x)
> g <- function(y) y * h(y)
> h <- function(z) {
+ r <- log(z)
+ if (r < 10)
+ r^2
+ else r^3
+ }
> f(-10)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
```

Where did `f(-10)` fail?

```
> traceback()
3: h(y) at #1
2: g(x) at #1
1: f(-10)
```

`traceback` shows that the error occurred during evaluation of `h(y)`.

Debugging Tools: traceback - Example 2 contd.

```
> set.seed(100)
> xList <- as.list(rpois(1000, lambda = 5) - 1)
> lapply(xList, f)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
```

Which element of `xList` caused the error?

```
> traceback()
4: h(y) at #1
3: g(x) at #1
2: FUN(X[[417L]], ...)
1: lapply(xList, f)
```

`traceback` shows that evaluation of element `xList[[417]]` caused the error. Only 0:8% of the list elements cause an error:

```
> which(unlist(xList)<0)
[1] 417 498 516 559 719 733 903 997
```

Debugging Tools: debug

Where in the function h did the error occur? - Use debug to find out.

`debug(h)` . . .

- allows to step *through* the function h line by line
- alters the way h is executed
- flags the function h for debugging

When a flagged function is called . . .

- the body of the function is printed
- a *browser* command line opens in the console
- each statement in the function is executed one at a time
- the user can control when each statement gets executed
- the user interacts with the environment of the function

Debugging Tools: debug - Example 2 contd.

What happens, if function h is called after flagging?

```
> debug(h)
> f(xList[[417]])
debugging in: h(y)
debug at #1: {
r <- log(z)
if (r < 10)
r^2
else r^3
}
Browse [2]>
```

Now, we can interact via the `browser` with the environment of the function:

```
Browse[2]> ls()
[1] "z"
Browse[2]> summary(z)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-1 -1 -1 -1 -1 -1
Browse[2]> str(z)
num -1
Browse [2]>
```

Debugging Tools: debug

The four basic debugging commands inside the browser:

`c, cont`	exit the browser and continue execution at the next statement
`n`	enter the step-through debugger if the function is interpreted
`where`	print a stack trace of all active function calls
`Q`	exit the browser and the current evaluation and return to the top-level prompt

Besides the four basic debugging commands, all other R commands (including assignments) are allowed.

New objects are created in the local environment of the debugged function and will disappear when the debugger finishes.

If you have objects in your environment with the names `n`, `c`, or `Q`, then you must explicitly use the `print` function to print their values (i.e. `print(n)` or `print(c)`).

Debugging Tools: debug - Example 2 contd.

browsing function h

```
Browse[2]> where
where 1 at #1: h(y)
where 2 at #1: g(x)
where 3: f(xList[[417]])
```

```
Browse[2]> n
debug at #2: r <- log(z)
Browse[2]> n
debug at #3: if (r < 10) r^2 else r^3
Browse[2]> ls()
[1] "r" "z"
Warning message:
    In log(z) : NaNs produced
Browse[2]> r < 10
[1] NA
Browse[2]> n
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
```

Debugging Tools: browser - Example 2 contd.3

Manual debugging: explicit calls to browser

```
> h <- function(z) {
    + r <- log(z)
    + browser()
    + if (r < 10)
        + r^2
    + else r^3
    + }
> f(-10)
Called from: h(y)
Browse[1]> ls()
[1] "r" "z"
Warning message:
    In log(z) : NaNs produced
Browse[1]> r
[1] NaN
```

Caution: do not forget to remove the call to browser() after debugging.

Debugging Tools: trace

Modify code temporarily with trace: - trace makes minor modi
cations to existing functions *on the fly*. - The traced functions are only modi
ed indirectly without re-sourcing them. - Since base functions cannot be edited by the user, trace may be the only option available for making modifications.

```
> str(trace)
function (what, tracer, at, ...)
    ```


\begin{itemize}
\item[what] name of function to be traced
\item[tracer] code to be inserted (name of function or unevaluated expression)
\item[at] the position where the code will be inserted
\item[...] is for a lot more arguments available for \code{trace}, see \code{?trace}
\end{itemize}

Debugging Tools: `trace` - Example 2 contd.

Let's use trace with function `h`:
```

```
```r
> h <- function(z) {
+ r <- log(z)
+ if (r < 10)
    + r^2
+ else r^3
```

Debugging Tools: trace - Example 2 contd.

continued . . .

```
> h
Object with tracing code, class "functionWithTrace"
Original definition:
    function(z) {
        r <- log(z)
        if (r < 10)
            r^2
        else r^3
        }
# (to see the tracing code, look at body(object))
> body(h)
{
    r <- log(z)
    {
        .doTrace(if (is.nan(r))
            browser(), "step 3")
        if (r < 10)
            r^2
        else r^3
        }
    }
```

Debugging Tools: trace - Example 2 contd.

continued . . .

```
> f(1)
[1] 1
> f(-10)
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "r" "z"
Warning message:
    In log(z) : NaNs produced
Browse[1]> Q
```

A call to untrace cancels the tracing:

```
> untrace(f)
> f(-10)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
    In log(z) : NaNs produced
```

Debugging Tools: recover - Example 2 contd.

The recover function helps in situations where you want to browse functions several functions in the stack:

```
> ## We set a conditional break point for the function recover
    > trace(what = h, tracer = quote(if(is.nan(r)) recover()),
            + at = 3, print = FALSE)
[1] "h"
> body(h)
{
    r <- log(z)
    {
        .doTrace(if (is.nan(r))
            recover())
        if (r < 10)
            r^2
        else r^3
        }
    }
```

Debugging Tools: recover - Example 2 contd.

continued . . .

```
> f(-10)
Enter a frame number, or 0 to exit
1: f(-10)
2: #1: g(x)
3: #1: h(y)
Selection: 2 ## Browse the g function
Called from: eval.parent(exprObj)
Browse[1]> ls()
[1] "y"
Warning message:
    In log(z) : NaNs produced
Browse[1]> y
[1] -10
Browse[1]> c
Enter a frame number, or 0 to exit
1: f(-10)
2: #1: g(x)
3: #1: h(y)
Selection: 0 ## Exit the recover function
Error in if (r < 10) rˆ2 else rˆ3 : missing value where TRUE/FALSE needed
```

Debugging Tools: recover - Example 3

Problem: how to browse functions inside other functions?

```
> f1 <- function(x) {
+   g1 <- function(y) {
+       h1 <- function(z) {
+           r <- log(z)
+           if (r < 10) r^2 else r^3
+       }
+       y * h1(y)
+   }
+   x - g1(x)
+ }
> f1(-10)
Error in if (r < 10) r^2 else r^3 ...
> traceback()
3: h1(y) at #7
...
> trace(what = h1, tracer = quote(if(is.nan(r)) recover()),
+ at = 3, print = FALSE)
Error in methods::.TraceWithMethods(what = h1, tracer = quote(if (is.nan(object 'h1' not found
```

Debugging Tools: recover - Example 3

More general approach to use recover:

```
> options()$error
NULL
options(error = recover)
f1(-10)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
Enter a frame number, or 0 to exit
1: f1(-10)
2: #9: g1(x)
3: #7: h1(y)
Selection:
```

Debugging Tools: recover - Example 3

Even more general approach to use recover to treat warnings:

```
> options()$warn
[1] 0
> options(warn = 2)
> f1(-10)
Error in log(z) : (converted from warning) NaNs produced
Enter a frame number, or 0 to exit
1: f1(-10)
2: #9: g1(x)
3: #7: h1(y)
4: #4: .signalSimpleWarning("NaNs produced", quote(log(z)))
5: withRestarts({
.Internal(.signalCondition(simpleWarning(msg, call), msg, call))
.Internal(.dfltWar
6: withOneRestart(expr, restarts[[1]])
7: doWithOneRestart(return(expr), restart)
Selection:
```

Debugging in R

Summary

`traceback`	prints the call stack of the last uncaught error
`debug`	flags a function for debugging
`browser`	interrupts the execution of an expression and allows the inspection of the environment
`trace`	allows to insert debugging code into an existing function
`recover`	allows to browse directly on ***any*** of the currently active function calls; with `options(warn=2, error=recover)` warnings can be debugged as well

Outline

Performance Enhancement - Outline

Goal:

- Fast Code
-

Include R Code

```
#Some R code
some.function <- function(formal arguments) {
    #body
    return(value)
    }
```