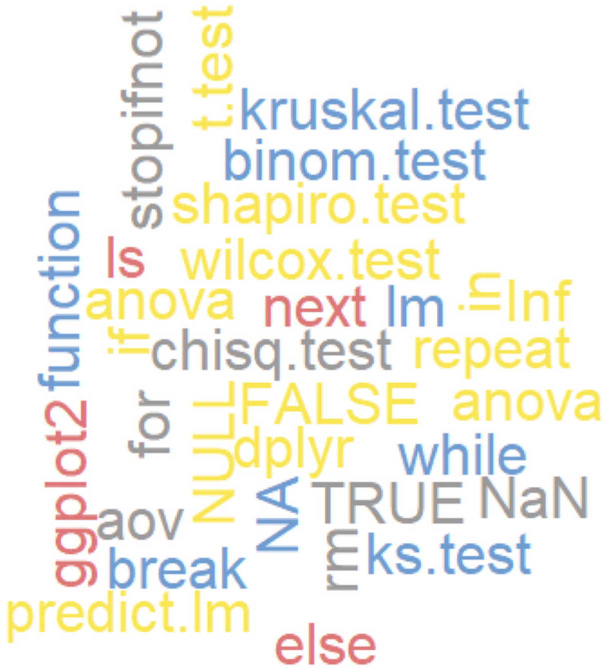
# R - Part I/II




Alexander Hirner
Aleksandra Tyjan

**Version 6 (Juni 2024)**

# R - Part I

**Version 6.0 (June 2024)**

Alexander Hirner

## Table of contents


**1 Getting Started** 2

**2 The Most Important R Objects (Overview)** 4

**3 Data Managment (`data.frame` Objects in Detail)** 9

**4 Selected Functions** 14

**5 Groupwise Descriptive Statistics** 19

**6 Graphs** 21

**7 Functions and Control Flow by Example** 27

**8 Recommended Reading** 30

# 1 Getting Started

## 1.1 Installation

1. Download R from **https://www.r-project.org**.
2. Install the `tidyverse` package by typing in `install.packages("tidyverse")`.
3. Install an editor. Recommendations:
   - **Visual Studio Code** from **https://code.visualstudio.com/** or
   - **RStudio** from **https://www.rstudio.com/**.

> ⚠ Order of Installation
>
> If you are using RStudio, the order of installation (first R, then the editor) matters (otherwise you will get an error during installation).

## 1.2 Recommended Packages

- `tidyverse` - **A collection of packages for data science**.
- `ggplot2` - Elegant data visualisations using *The Grammar of Graphics*.
- `XLConnect` - Excel connector for R.
- `foreign` - Read data stored by Minitab, SPSS, SAS, etc.
- `dplyr` - A grammar of data manipulation.
- `tidyr` - Tidy messy data.
- `data.table` - Extension of `data.frame` (fast!)
- `doBy` - Groupwise Statistics, LSmeans, Linear Estimates, Utilities.
- `cowplot` - Various features to create publication-quality figures (e.g. combining plots).

> Installing Packages
>
> The command to install additional packages is `install.packages("...")`. However, it is possible that the editor which you are using, supports you, i.e. in RStudio you can click on `Tools` and `Install packages` (and then simply follow the instructions).

Functions from previously installed packages can be invoked by

1. using the syntax `packagename::functionname(...)` or
2. "loading" the package using `library(packagename)` or `require(packagename)` (preferably at the beginning of a script) which makes the prefix `packagename::` unnecessary then (to "remove" the package again use `detach`).

Note that each way has its advantages and disadvantages.

## 1.3 Basic R Commands

Let us start with the "basics" first, e.g. creating simple objects, inspecting and deleting them, managing file paths and getting help:

| Command | Description |
|---|---|
| `getwd()` | Gets working directory. |
| `setwd("path")` | Sets working directory (R uses forward slashes in file paths!). |
| `variable <- value` | Assigning a value to a variable. |
| `variable <- c(val1, val2, ...)` | Assigning values to a variable (c is for *combine*). |
| `ls()` | Lists all objects in the workspace. |
| `rm(object)` | Removes an object. |
| `help()` | Getting help. |
| `?object` | An alias for `help()`. |
| `help.search("searchstring")` | Searches the help system for a given string. |
| `??searchstring` | An alias for `help.search()`. |
| `#` | The hash sign introduces a comment. |
| `data()` | Lists built-in sample data. |

Functions can also be nested (i.e. "a function in a function"), e.g. the command `rm(list=ls())` deletes everything from memory.

> ⚠ Using `rm(list=ls())`
>
> The command `rm(list=ls())` is helpful to clean up the memory before starting a new script or project. However, use it with caution since there is no "undo" for this command.

# 2 The Most Important R Objects (Overview)

## 2.1 Vectors

### Usage and Important Operations

Vectors can be used as "containers" to store univariate data.

| Command | Description |
|---|---|
| `x <- c(val1, val2, ...)` | Creates a vector. Use quotes for string literals. |
| `x[i]` | Value on position **i** of vector **x**. The first element is on position **1** in R. |
| `x[-i]` | All values except for the one on position **i**. |
| `x[j:k]` | Observations from position **j** to position **k**. |
| `x[c(j,m)]` | Observations on position **j** and **m**. |
| `length(x)` | Number of Observations in **x**. |
| `rev(x)` | Reverses elements of **x**. |
| `x <- seq(from, to , by)` | Generates an arithmetic progression. Use `1:n` as an alias for `seq(1,n,1)`. |
| `x <- rep(pattern, n)` | Repeats a pattern **n** times |

> ⚠ About R Objects
>
> Commands like `x[-1]` or `rev(x)` never change the object *in situ* - these commands (when used in interactive mode) simply print out the values to the console. If you want to change the vector permanently, the assignment operator (`<-`) must be used.

### Example

```
# a vector:
height <- c(180, 167, 198.5, 156, 170, 172, 169, 155)
height <- height[-3] # drop the third entry
summary(height) # summary statistics
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  155.0   161.5   169.0   167.0   171.0   180.0 
```

## 2.2 Objects of the `data.frame`-Class

### Usage and Important Operations

A `data.frame` is the appropriate structure to store multivariate data, i.e. data consisting of **n** rows (observations) and **k** columns (variables or data fields).

| Command | Description |
|---|---|
| `X <- data.frame(v1, v2, ...)` | Creates a `data.frame`. Here, **v1** and **v2** are **vectors of the same length**. |
| `X[i, ]` | Row (observation) **i**. |
| `X[ ,k]` | Column (variable) **k**. |
| `X[i:j, ]` | Observations from row **i** to **j** (and all variables). |
| `X[c(i,k), ]` | Observations in row **i** and **k** (and all variables). |
| `X[-c(i,k), ]` | All observations except those in row **i** and **k** (and all variables). |
| `X[, c(i,k)]` | All rows, only column **i** and **k**. |
| `X[, -c(i,k)]` | All rows, drop values in column **i** and **k**. |
| `X$varname` | Addresses a column by its name. |
| `names(X)` | Gets (or assigns) variable names. |

### Example

In the following example we will create a small `data.frame` with n=5 observations (cats) and k=3 variables from scratch:

```
id <- c(1,2,3,4,5) # or 1:5
sex <- c("male", "female", "female", "female", "male")
weight <- c(3, 4, 3.7, 2.5, 0.9)
cats <- data.frame(id, sex, weight)
rm(id, sex, weight) # drop original vectors now!
```

This is our data:

```
print(cats)
```

```
  id    sex weight
1  1   male    3.0
2  2 female    4.0
3  3 female    3.7
4  4 female    2.5
5  5   male    0.9
```

Now let us address one of the columns, i.e. "weight" (weight of the cat in kilos) to calculate their average:

```
mean(cats$weight)
```

```
[1] 2.82
```

Now let us drop the column "id" because we do not need it and display the `data.frame`:

```
cats <- cats[ , -1]
cats
```

```
     sex weight
1   male    3.0
2 female    4.0
3 female    3.7
4 female    2.5
5   male    0.9
```

## 2.3 Lists

### Usage and Important Operations

- Lists in R are capable of storing different data types (strings, numbers, vectors, data.frames or even lists).
- If you want to create a list "from scratch", use the function `list` and simply pass the objects that you want to store in the list. With `names()` it is possible to assign names to the elements stored in the list.
- Whenever you run a statistical procedure (e.g. ANOVA, linear regression, cluster analysis, …) the output **should always be stored in a variable**. This variable is often a list (or at least "behaves" like a list). List elements can be addressed by using the `$`-symbol or double square brackets and an index.
- Note that a `data.frame` is a special kind of a `list` - it is a `list` where all elements are vectors of the same length.

### Example

```
v1 <- c(1, 2, 3)
v2 <- c("a","b","c")
v3 <- data.frame(v1, v2)
v4 <- 7
v5 <- "hello world"
mylist <- list(v1,v2,v3,v4,v5)
names(mylist) <- c("v1","v2", "DF", "a_number", "a_string")
```

We can now address single elements from the list:

```
print(mylist$v1)
```

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

```
print(mylist$a_string)
```

```
[1] "hello world"
```

## 2.4 Matrices

### Usage

You might need the `matrix` data type if you do linear algebra with R.

### Example

```
x <- seq(1,12,1) # or simply  x <- 1:12
y <- matrix(x, nrow=3, byrow=TRUE)
print(y)
```

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

In the example above, a matrix with 12 elements (numbers from 1 to 12) and three rows (`nrow`) was created. Alternatively, you can specify `ncol` (the number of columns). The flag `byrow` indicates whether the matrix should be filled up "row-wise" (if `TRUE`) or "column-wise" (if `FALSE`).

With `t(X)` you can transpose a matrix X. This command also works for objects of the `data.frame`-class.

## 2.5 Type Conversions

Sometimes the following type conversion functions can be helpful:

- `as.numeric`
- `as.character`
- `as.vector`
- `as.matrix`
- `as.data.frame`

## 2.6 More Data Types

There are more data types (i.e. there are objects of the `tibble`-class and some more). These data types require additional packages but their behaviour and purpose (storing multivariate data) is similar to a `data.frame`.

# 3 Data Managment (`data.frame` Objects in Detail)

## 3.1 Logical Operators

For some operations (e.g. filtering data or creating new variables) we need comparison operators:

| Command | Description |
|---|---|
| `<` | less than |
| `<=` | less than or equal to |
| `>` | greater than |
| `>=` | greater than or equal to |
| `==` | equality |
| `!=` | inequality |
| `%in%` | match with one element of a vector? |

These operators can be combined using `&` (logical AND) or `|` (logical OR).

## 3.2 Working with Files

| Command | Description |
|---|---|
| `read.table` | Reads a file and creates a `data.frame` from it. |
| `read.csv` | Reads a file and creates a `data.frame` from it. |
| `read.csv2` | Reads a file and creates a `data.frame` from it. |
| `write.table` | Prints a `data.frame` to a file. |
| `write.csv` | Prints a `data.frame` to a file. |
| `write.csv2` | Prints a `data.frame` to a file. |

If you want to import data from an Excel-file directly (i.e. without saving it as a csv-file), you can use the function `readWorksheetFromFile` from the library `XLConnect`.

Possible pitfalls when dealing with files

- Make sure you are in the right directory. You might need the `setwd`-command.
- The functions listed above differ with respect to their default settings (i.e. for the field separator string and the decimal symbol). Use the `help` function to find out details.
- Commands such as `write.table` (or related functions) overwrite existing files **without warning (!)**.

### 3.3 Data Managment Operations in Detail

| Command | Description |
|---|---|
| `dim(X)` | Displays number of rows and columns. |
| `head(X, n)` | Returns the first **n** rows of **X**. |
| `tail(X, n)` | Returns the last **n** rows of **X**. |
| `str(X)` | Compactly displays structure of **X**. |
| `Z <- rbind(X, Y)` | Adding observations from **X** and **Y**. |
| `Z <- cbind(X, Y)` | Adding a column to X. |
| `X$new <- formula` | Creates a **new** column in **X** with a **formula**. |
| `X <- transform(X, new1=f1, new2=f2,...)` | Calculates new columns **new1** and **new2** with the formula **f1** or **f2**, respectively. |
| `names(X)` | Displays (or changes) variable names. |
| `ifelse(cond, ifpart, elsepart)` | Conditional execution (i.e. for recoding). |
| `Y <- subset(X, conditions)` | Creates a subset **Y** based on logical conditions. |
| `X <- X[order(X$var), ]` | Sorts your `data.frame` by one or more variables. Use `decreasing=TRUE` for descending order. |
| `Y <- split(X, X$splitvar)` | Creates a list of `data.frame`-objects (for each value of **splitvar**). |
| `Y <- merge(x, y, by.x, by.y)` | Merges (i.e. joins) `data.frame`-objects by a common column. |

### 3.4 Example

At first let us create some data, e.g. 10 observations and three variables:

```
id <- 1:10
sex <- rep(c("m", "f"), 5)
set.seed(123)
height <- round(rnorm(10, mean=172, sd=10), 1)
X <- data.frame(id, sex, height)
head(X)
```

```
  id sex height
1  1   m  166.4
2  2   f  169.7
3  3   m  187.6
4  4   f  172.7
5  5   m  173.3
6  6   f  189.2
```

Now let us sort the `data.frame` in descending order by the variable height:

```
Y <- X
Y <- Y[order(Y$height, decreasing=TRUE), ]
head(Y, 3)
```

```
  id sex height
6  6   f  189.2
3  3   m  187.6
7  7   m  176.6
```

Now we can create another column, the height in meters:

```
Y <- X
Y <- transform(Y, heightm = height/100)
head(Y, 2)
```

```
  id sex height heightm
1  1   m  166.4   1.664
2  2   f  169.7   1.697
```

We can also classify the subjects as follows (lte… *less than or equal*, gt… *greater than*):

```
Y <- X
Y$class <- ifelse(Y$height > median(Y$height),
"height gt median", "height lte median")
Y[6:8,]
```

```
  id sex height             class
6  6   f  189.2  height gt median
7  7   m  176.6  height gt median
8  8   f  159.3 height lte median
```

We want to create a new `data.frame` using a query and the `subset`-command to copy only females taller than 172 cm:

```
Y <- X
Y1 <- subset(Y, (height > 172) & (sex=="f"))
Y1
```

```
  id sex height
4  4   f  172.7
6  6   f  189.2
```

The following example illustrates the usage of the `split`-function. It returns a list of `data.frame`-objects:

```
Y <- X
by_sex <- split(Y, Y$sex)
by_sex
```

```
$f
   id sex height
2   2   f  169.7
4   4   f  172.7
6   6   f  189.2
8   8   f  159.3
10 10   f  167.5

$m
  id sex height
1  1   m  166.4
3  3   m  187.6
5  5   m  173.3
7  7   m  176.6
9  9   m  165.1
```

Recoding a variable into more than two categories:

```
Y <- X
Y$category[Y$height <= 170] <- "category 1"
Y$category[Y$height >= 170 & Y$height <= 176] <- "category 2"
Y$category[Y$height > 176] <- "category 3"
head(Y, 4)
```

```
  id sex height   category
1  1   m  166.4 category 1
2  2   f  169.7 category 1
3  3   m  187.6 category 3
4  4   f  172.7 category 2
```

# 4 Selected Functions

## 4.1 Preliminary Remarks

### Applying Functions to Columns

> Applying functions to columns
>
> Functions can also be applied to whole vectors of data (columns in your `data.frame`) - in contrast to other programming languages, there is no need to use loops here.

### Possible Pitfalls

> Possible pitfalls
>
> - The function `log()` is the "natural" logarithm.
> - Trigonometric functions (and inverse trigonometric functions) return/expect the angle in radians (not in degrees!)

## 4.2 Common Functions in R

| Function | Description |
|---|---|
| `+`, `-`, `*`, `/` | Basic math. |
| `a^b` or `a**b` | Returns $a^b$. |
| `sqrt(x)` | $\sqrt{x}$ |
| `pi`, `exp(1)` | Mathematical constants. |
| `sin(x)`, `cos(x)`, `tan(x)` | Trigonometric Functions. |
| `asin(x)`,`acos(x)`, `atan(x)` | Inverse trigonmetric functions. |
| `floor(x)`, `ceiling(x)` | Rounding to the next integer. |
| `round(x, digits)` | Rounding to a given precision. |
| `log(x)` | (natural) logarithm. |
| `log10(x)` | logarithm (base 10). |
| `exp(x)` | Returns $e^x$. |
| `sum(x)` | Returns the sum of all elements in vector **x**. |
| `cumsum(x)` | Returns the cumulated sum of all elements in vector **x**. |
| `prod(x)` | Returns the product of all elements in vector **x**. |
| `cumprod(x)` | Returns the cumulated product of all elements in vector **x**. |
| `paste(x, y)` | Concatenates strings. |
| `class(x)` | Object classes. |
| `cat(x, y)` | Concatenates and prints. |

| Function | Description |
|---|---|
| `unique(x)` | Removes duplicates. |
| `set.seed(seed)` | Sets seed for random number generation. |

## 4.3 Using `any`, `all` and `which`

### Functions

| Function | Description |
|---|---|
| `any(...)` | Are some values TRUE? |
| `all(...)` | Are all values TRUE? |
| `which(...)` | Which indices are TRUE? |

### Examples

```r
x1 <- c(1, 3, 5, 9, 7)
any(is.na(x1))
```

```
[1] FALSE
```

```r
all(x1 > 2)
```

```
[1] FALSE
```

```r
which(x1 > 6)
```

```
[1] 4 5
```

## 4.4 Set Functions

| Function | Description |
|---|---|
| `union(A, B)` | $A \cup B$ |
| `intersect(A, B)` | $A \cap B$ |
| `setdiff(A, B)` | $A \backslash B$ |
| `setequal(A, B)` | Is A = B? |
| `is.element(x, A)` | $x \in A$? |

## 4.5 Combinatorics

| Function | Description |
|---|---|
| `factorial(x)` | Returns $x!$ (number of permutations). |
| `choose(n, k)` | Binomial coefficient, i.e. $\binom{n}{k}$. |
| `combn(x, m)` | Generates all combinations with **m** objects of the elements in **x**. |
| `sample(x, size, replace)` | Randomly selects elements. |

## 4.6 Basic Descriptive Statistics

### Functions

| Function | Description |
|---|---|
| `mean(x)` | Arithmetic mean. |
| `median(x)` | Median. |
| `sd(x)` | Standard deviation. |
| `var(x)` | Variance. |
| `fivenum(x)` | Tukey's five number statistics. |
| `min(x)` | Minimum. |
| `max(x)` | Maximum. |
| `quantile(x, probs)` | Quantiles. |
| `cor(x)` | Correlation. |
| `cov(x)` | Covariance. |

### Examples

```
set.seed(789); x <- rnorm(15)
x[1:4]
```

```
[1]  0.52409671 -2.26076788 -0.01967972  0.18313989
```

```
fivenum(x)
```

```
[1] -2.26076788 -0.57710626 -0.36135148  0.08173009  0.92790739
```

```
quantile(x, probs=seq(0, 1, 0.2))
```

```
        0%        20%        40%        60%        80%       100% 
-2.2607679 -0.7336064 -0.4351714 -0.1764192  0.2513313  0.9279074 
```

## 4.7 Probability Distributions

All functions dealing with probability distributions always consist of a prefix (`d`, `p`, `q` or `r`) plus the (abbreviated) name of the distribution (e.g. `norm`, `unif`, `chisq`, etc.).

### Example 1

We want to draw five numbers from a $\chi^2$ distribution with 3 degrees of freedom:

```
set.seed(123); rchisq(n=5, df=3)
```

```
[1] 1.03611518 5.08870916 0.04818784 2.26693313 6.90085393
```

### Example 2

If the height of male students follows a normal distribution with $\mu = 174$ cm and $\sigma = 7$ cm, what percentage is taller than 180 cm? (Solution: $\approx 19.57$ %)

```
pnorm(180, mean=174, sd=7, lower.tail=FALSE)
```

```
[1] 0.195683
```

## 4.8 Notes on Missing Values

### Missing Values

- The function `is.na(x)` returns `TRUE` for missing and `FALSE` for non-missing values in a vector `x`. Therefore you can easily count the missing values by using `is.na` and the function `sum`.
- Other comparison operators (`==`, `!=`) can **not** be used to detect missing values (Comparisons involving missing values always return `NA`).
- The function `length` counts **all** values in a vector (regardless of their "missing status").
- Functions such as `mean` or `sd` fail (return `NA`) whenever there is at least one value missing - unless you specify `na.rm=TRUE` in the function call.
- The `table` command (which creates a frequency table) by default will **ignore** missing values unless you specify `useNA="always"` or `useNA="ifany"` in the function call.
- Sometimes missing values are coded as **999** (or similar) in your data. You can replace them using the `which` command.

#### Example

```r
# weight... weight of students in kilos, 999 means "missing" here.
weight <- c(79, 88, 59, 999, 91, 60)
mean(weight)
```

```
[1] 229.3333
```

Replace **999** by `NA`

```r
weight[which(weight==999)] <- NA
weight
```

```
[1] 79 88 59 NA 91 60
```

```r
mean(weight)
```

```
[1] NA
```

```r
mean(weight, na.rm=TRUE)
```

```
[1] 75.4
```

# 5 Groupwise Descriptive Statistics

## 5.1 Properties of the Function doBy::summaryBy

1. Install the package doBy
2. We will use the function summaryBy to create summary tables that contain exactly the information which we want to see.

| Syntax of doBy::summaryBy |
|---|
| `doBy::summaryBy(var(s) ~ groupvar(s), data=..., FUN=...)` |

Notes:

- If there are more variables in the "formula", use + to separate them.
- It is possible to use built-in functions as well as your own functions.
- The function length applied to **any** column of your data counts the observations.
- If there are more functions to apply groupwise to your data.frame, pack the function names in a list, i.e. FUN=list(fun1, fun2, ...).
- The function doBy::summaryBy returns a handy data.frame!

## 5.2 Examples

### Creating Sample Data

```
set.seed(123)
group <- rep(c("A","B"), 30)
treatment <- rep(c("group1", "group2", "group3"), 10)
values <- rchisq(30, df=5)
mydata <- data.frame(group, treatment, values)
print(head(mydata,5))
```

```
  group treatment     values
1     A    group1  2.5718020
2     B    group2  8.0747086
3     A    group3  0.6485141
4     B    group1  4.3740386
5     A    group2 10.3216603
```

#### Example 1

Average of *values* by *treatment*:

```
summary1 <- doBy::summaryBy(values ~ treatment,
data=mydata, FUN=mean)
print(summary1)
```

```
  treatment values.mean
1    group1    3.207749
2    group2    5.004441
3    group3    4.296330
```

#### Example 2

Minimum, maximum and a frequency count of *values* by *treatment* and *group*:

```
summary2 <- doBy::summaryBy(values ~ treatment + group,
 data=mydata, FUN=list(min, max, length))
# change name of last column in summary table:
names(summary2)[5] <- "N"
print(summary2)
```

```
  treatment group values.min values.max  N
1    group1     A  1.2220565   3.208945 10
2    group1     B  1.3825887   6.184881 10
3    group2     A  1.4285405  10.321660 10
4    group2     B  0.6062728   8.500349 10
5    group3     A  0.6485141   8.211414 10
6    group3     B  3.1459618   5.409890 10
```

# 6 Graphs

## 6.1 Create Sample Data

```
x1 <- rep(c("A","B"), 50)
set.seed(111)
x2 <- sample(c("U","V","W"), 100, replace=TRUE)
set.seed(345)
x3 <- rnorm(100, 100, 15)
set.seed(567)
x4 <- rchisq(100, 3)
X <- data.frame(x1, x2, x3, x4)
head(X,3)
```

```
  x1 x2       x3        x4
1  A  V 88.22638 3.5291558
2  B  W 95.80728 2.7358626
3  A  W 97.57813 0.9237687
```

## 6.2 Colours

Here are some ways to specify colors in your plot:

1. Colour name (use `colours()` to display all available colors)
2. As an RGB or HEX value.
3. Using colour palettes from external libraries.

## 6.3 Saving Graphs to a File

```
pdf("filename.pdf") # or png(...), jpeg(...), ...
# graphics commands here
dev.off()
```

If you want to save output from a statistical procedure, you can use

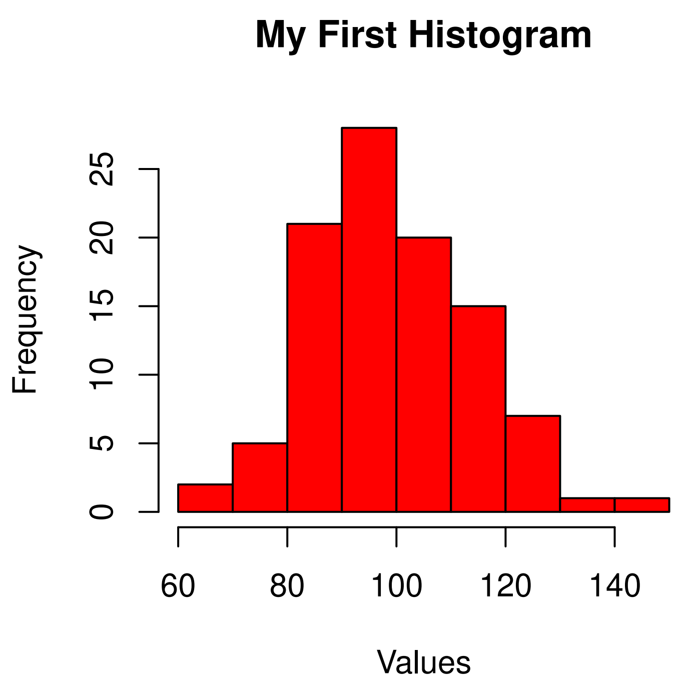
```
sink("filename.txt")
# ...statistics...
sink()
```

### 6.4 Example: Histogram

```
head(X, 3)
```

```
  x1 x2       x3        x4
1  A  V 88.22638 3.5291558
2  B  W 95.80728 2.7358626
3  A  W 97.57813 0.9237687
```

```
hist(X$x3, col=rgb(1,0,0), main="My First Histogram", xlab="Values")
```

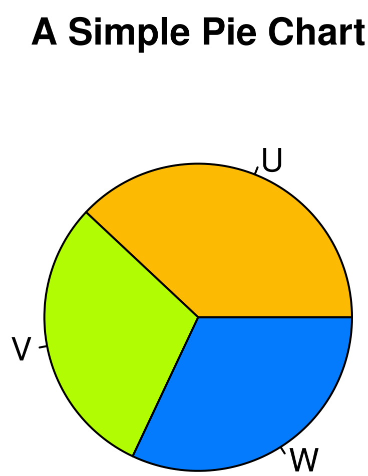


### 6.5 Example: Pie Chart

```
head(X, 3)
```

```
  x1 x2       x3        x4
1  A  V 88.22638 3.5291558
2  B  W 95.80728 2.7358626
3  A  W 97.57813 0.9237687
```

```
mytab <- table(X$x2) # Create frequency table first!
pie(mytab, col=c("#fcba03", "#b1fc03", "#037bfc"),
main="A Simple Pie Chart")
```

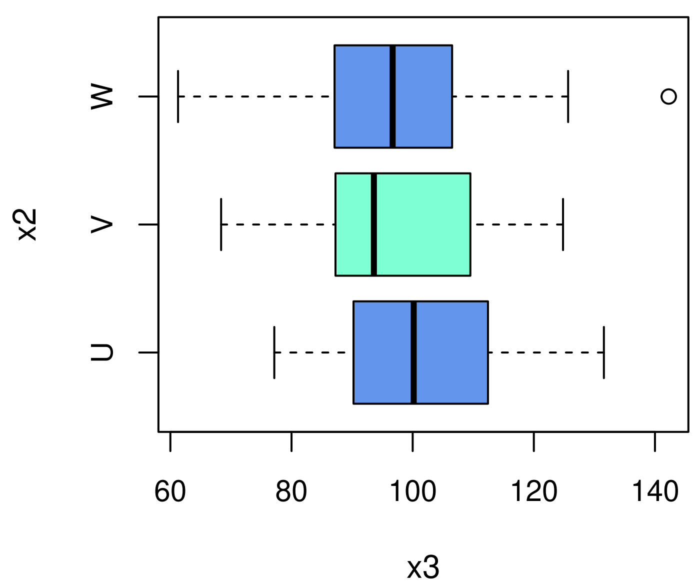


## 6.6 Example: Grouped Boxplot

```r
head(X, 3)
```

```
  x1 x2       x3        x4
1  A  V 88.22638 3.5291558
2  B  W 95.80728 2.7358626
3  A  W 97.57813 0.9237687
```

```r
boxplot(x3 ~ x2, data=X, horizontal=TRUE,
col = c("cornflowerblue", "aquamarine"), cex.axis=0.9)
```



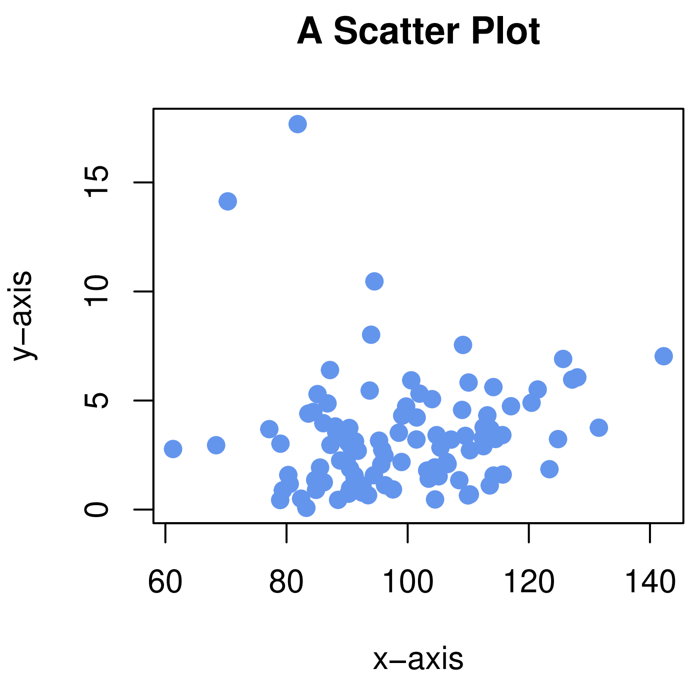

Note that we have three categories but we only passed a vector with two different colours. In this case, the vector is "recycled" here.

### 6.7 Example: Scatter Plot

```
head(X, 3)
```

```
  x1 x2       x3        x4
1  A  V 88.22638 3.5291558
2  B  W 95.80728 2.7358626
3  A  W 97.57813 0.9237687
```

```
plot(X$x3, X$x4, col="cornflowerblue", lwd=2, pch=19, xlab="x-axis",
 ylab="y-axis", main="A Scatter Plot")
```

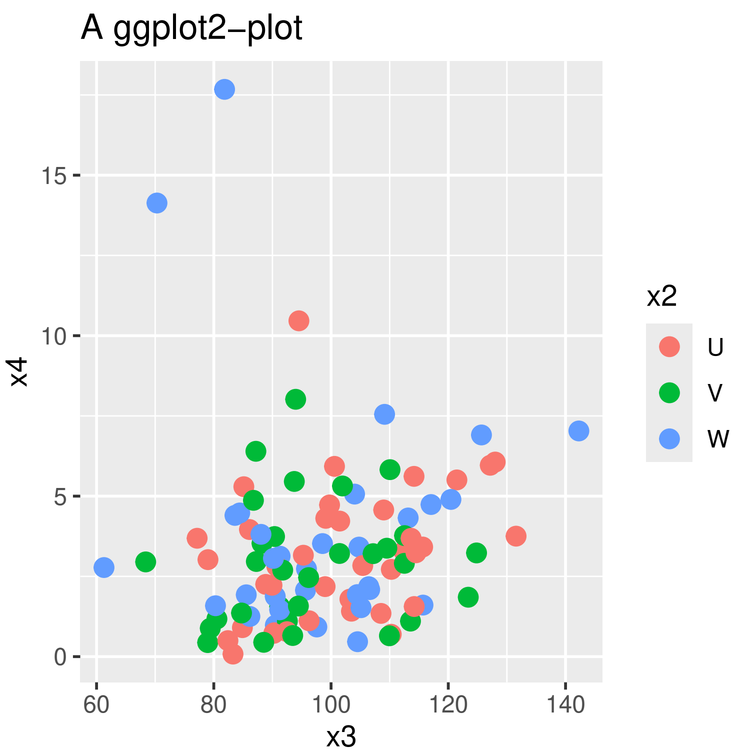


### 6.8 Example: Scatter Plot - `ggplot2`

```
head(X, 3)
```

```
  x1 x2       x3        x4
1  A  V 88.22638 3.5291558
2  B  W 95.80728 2.7358626
3  A  W 97.57813 0.9237687
```

```
library(ggplot2)
plt <-  ggplot(X, aes(x=x3, y=x4, color=x2))
plt <- plt + geom_point(size=3)
plt <- plt + ggtitle("A ggplot2-plot")
plt
```



# 7 Functions and Control Flow by Example

## 7.1 Loops with `for`

```
for (i in 1:3)
        {
                print("Hello")
        }
```

```
[1] "Hello"
[1] "Hello"
[1] "Hello"
```

## 7.2 Loops with `repeat`

```
n <- 1
repeat
{
  print(n)
  if (n >= 5)
    {
    break
    }
n <- n + 1
}
```

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

## 7.3 Loops with `repeat` and `break`

```
n <- 1
repeat {

  if (n == 3) {
    n <- n + 1
    next
  }
  print(n)


  if (n >= 5) {
    break
  }

 n <- n + 1
}
```

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

## 7.4 Loops with `while`

```
i <- 1
while (i <= 3)
        {
                print("Hello")
                i <- i + 1
        }
```

```
[1] "Hello"
[1] "Hello"
[1] "Hello"
```

## 7.5 Conditional Execution - Two Branches

```r
x <- 2
if (x == 1) {
  print("x is one")
} else {
  print("x is something else")
}
```

```
[1] "x is something else"
```

## 7.6 Conditional Execution - More Branches

```r
x <- 2
if (x == 1) {
  print("x is one")
} else if (x == 2) {
  print("x is two")
} else if (x == 3) {
  print("x is three")
} else {
  print("hm...")
}
```

```
[1] "x is two"
```

## 7.7 Conditional Execution - Using `switch`

```r
x <- 2

message <- switch(x,
                  "Value equals 1",
                  "Value equals 2",
                  "Value equals 3",
                  "Value is something else"
                  )

cat(message)
```

```
Value equals 2
```

### 7.8 Functions

```
squareandroundme <- function(x)
{
        result <- x * x
        result <- round(result, 1)
        return(result)
}

test <- squareandroundme(7.123)
test
```

```
[1] 50.7
```

# 8 Recommended Reading


- Kabacoff, R. (2011). *R in Action. Data Analysis and Graphics with R.* Shelter Island: Manning Publications.
- Ligges, U. (2008). *Programmieren mit R.* (3. Auflage). Berlin/Heidelberg: Springer.
- Sauer, S. (2019). *Moderne Datenanalyse mit R: Daten einlesen, aufbereiten, visualisieren, modellieren und kommunizieren.* Wiesbaden: Springer.
- Wickham, H., Grolemund, G. (2016). R for Data Science. Import, tidy, transform, visualize and model data. Sebastopol: O'Reilly Media.