2.1 Datentypen und Datenstrukturen

Nachdem wir uns mit RStudio vertraut gemacht haben, kommen wir nun zur ersten wirklichen inhaltlichen Auseinandersetzung mit R, nämlich mit Datentypen und Datenstrukturen. Wir wollen verstehen, welche es gibt und wie diese funktionieren, um sie in unserer Datenanalyse später benutzen zu können.

Datentypen

Numeric

Der Datentyp numeric beinhaltet alle reellen Zahlen. Zusätzlich werden alle ganzen Zahlen standardmäßig als numeric klassifiziert.

Dies können wir im folgenden Beispiel sehen:

```
ganzzahl <- 1
dezimalzahl <- 1.0

class(ganzzahl)
```

```
[1] "numeric"
```

```
class(dezimalzahl)
```

```
[1] "numeric"
```

```
ganzzahl == dezimalzahl
```

```
[1] TRUE
```

Character

Der Datentyp character beschreibt Wörter, Sätze; alles, was man mit Zahlen und Buchstaben zusammenfügen kann.

```
text <- "Wir lernen R."

class(text)
```

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

```
text_kopie <- "Wir lernen R."

text == text_kopie
```

```
[1] TRUE
```

Logical

Der Datentyp logical beinhaltet die Werte TRUE und FALSE (wahr und falsch). Sie können auch mit T und F abgekürzt werden. Man kann mit TRUE und FALSE auch rechnen, TRUE hat hierbei den Wert 1, FALSE den Wert 0.

```
wahr <- TRUE
falsch <- FALSE

wahr_kurz <- T
falsch_kurz <- F

wahr == wahr_kurz
```

```
[1] TRUE
```

```
falsch == falsch_kurz
```

```
[1] TRUE
```

```
wahr + falsch
```

```
[1] 1
```

Weniger wichtige Datentypen

Integer

Der Datentyp integer enthält alle ganzen Zahlen. Integer werden mit einem `L` hinter der Zahl gekennzeichnet.

```
ganzzahl <- 1L

class(ganzzahl)
```

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

```
andere_ganzzahl <- 2L

summe <- ganzzahl + andere_ganzzahl

class(summe)
```

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

```
# In der Konsole wird das 'L' nicht mit ausgegeben!
ganzzahl
```

```
[1] 1
```

Complex

Dieser Datentyp beinhaltet alle komplexen Zahlen.

```
komplexe_zahl <- 1i + 1

class(komplexe_zahl)
```

```
[1] "complex"
```

```r
andere_komplexe_zahl <- 9i + 3

komplexe_zahl + andere_komplexe_zahl
```

```
[1] 4+10i
```

Datenstrukturen

Wir können mit Variablen aus den oben angegebenen Datentypen sehr gut einfache Rechnungen und Vergleiche durchführen. Um aber effizient zu arbeiten und unsere Daten zu verwalten, braucht es Strukturen, in denen wir Variablen speichern und modifizieren können.

Vektor

Die wohl wichtigste Datenstruktur in R ist der *Vektor*. Vektoren sind Ansammlungen von Variablen eines bestimmten Datentyps. Ein Beispiel ist der Vektor $\vec{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$, der die Zahlen 1, 2, und 3 enthält.

Wir erstellen oder *initialisieren* einen Vektor mit der `c()`-Funktion:

```r
# Vektor initialisieren
x <- c(1, 2, 3)

# Vektor ausgeben
x
```

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

Wenn wir auf Elemente im Vektor zugreifen wollen, dann machen wir das durch eckige Klammern hinter dem Namen des Vektors:

```r
# R beginnt bei der Zählung mit der 1.
x[1]        # 1. Element
```

```
[1] 1
```

```r
x[-1]       # alle außer dem 1. Element
```

```
[1] 2 3
```

```
x[1:2]      # alle Elemente vom 1. bis zum 2.
```

```
[1] 1 2
```

```
x[c(1, 3)]  # alle Elemente, deren Index im Vektor (1,3) enthalten ist
```

```
[1] 1 3
```

Wir können mit Vektoren ganz normal rechnen (Zeile für Zeile):

```
x * 3
```

```
[1] 3 6 9
```

```
# Neuer Vektor
y <- c(3, 4, 5)

# Zeilenweise Addition/Subtraktion
x + y
```

```
[1] 4 6 8
```

```
x - y
```

```
[1] -2 -2 -2
```

```
# Für alle, die schon Vektoren behandelt haben:
# Betrag eines Vektors
# type = "2" steht für die sogenannte "2-Norm" (die euklidische Norm)
norm(x, type = "2")
```

```
[1] 3.741657
```

Wir können verschiedene “Maße” des Vektors nehmen:

```
  length(x)   # Dimension oder "Länge" des Vektors
```

```
[1] 3
```

```
  mean(x)     # Mittelwert des Vektors
```

```
[1] 2
```

```
  sum(x)      # Summe der Einträge des Vektors
```

```
[1] 6
```

Aufgabe 1

Gib den kleinsten und den größten Eintrag sowie den Median von `x` an!

```
min(x)
max(x)
median(x)
```

Dataframe und Tibble

Für diesen Teil und auch darüber hinaus benötigen wir die `tidyverse`-Library. Es bietet zahlreiche nützliche Features, die uns beim Codeschreiben viel Arbeit abnehmen und unseren Code besser lesbar machen werden.

Wenn ihr diese Library noch nicht installiert habt, dann geht das mit dem folgenden Befehl in eurer Konsole:

```
install.packages("tidyverse")
```

Wir laden nun die `tidyverse`-Library in unseren Arbeitsbereich, indem wir folgenden Code benutzen:

```
library(tidyverse)
```

Dataframes bzw. *Tibbles* sind ähnlich wie Tabellen: Sie speichern geordnet Daten. Dabei sind in den Zeilen die verschiedenen Einträge angeordnet, und in den Spalten finden sich Variablen für jeden dieser Einträge.

Wir rufen nun das `mtcars`-Datenset auf, eines der bekanntesten Datensets in R, auf das auch ohne weitere Pakete direkt zugegriffen werden kann. Das `mtcars`-Datenset enthält Daten zu einigen (älteren) Autos. Die Beschreibung dieses Datensets finden wir, indem wir `?mtcars` in die Konsole eingeben.

```
# Um nur die obersten Einträge eines Datensets auszugeben,
# benutzen wir die head()-Funktion
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Wir erhalten eine Übersicht über die ersten Autos und ihre Spezifikationen. Doch wie wir im Einführungskapitel gesehen haben, ist das nicht das Einzige, was wir mit *Dataframes* oder *Tibbles* machen können.

Wir wollen nun alle Autos mit mehr als 30 Meilen pro Gallone Benzin ausgeben. Hierbei können wir mit dem Dollarzeichen auf einzelne Spalten zugreifen, und ähnlich wie bei Vektoren benutzen wir eckige Klammern, um auf einzelne Elemente zuzugreifen. Die Schreibweise hierbei ist wie folgt: `Datenset[Bedingung für Zeilen, Bedingung für Spalten]`. Da wir hier die Zeilen (Autos) mit mehr als 30 Meilen pro Gallone ausgeben wollen, füllen wir diese Bedingung in die *Bedingung für Zeilen* ein und erhalten das gewünschte Ergebnis.

```
mtcars[mtcars$mpg > 30, ]
```

```
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

Das ist aber bei mehreren Bedingungen, oder wenn wir diesen Filter in eine Folge aus Operationen einbinden wollen, ziemlich schwer zu schreiben und vor allem zu lesen. Deswegen überführen wir dieses Datenset mit der `as_tibble()`-Funktion in einen *Tibble*, auf dem wir viel einfacher filtern und später auch Daten entfernen, zusammenfügen usw. können. Für diese Funktion muss nun das `tidyverse`-Paket installiert sein (s.o.).

```
mtcars_tibble <- as_tibble(mtcars)
```

Schauen wir uns diesen Tibble mal an:

```
mtcars_tibble
```

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# i 22 more rows
```

Komisch, die Autonamen sind weg! Das liegt daran, dass Tibbles etwas anders funktionieren als Dataframes. Dataframes haben benannte Zeilen, während Tibbles an sich diese Information nicht speichern. Wir müssen daher Folgendes machen:

```
mtcars_tibble <- as_tibble(rownames_to_column(mtcars))

mtcars_tibble
```

```
# A tibble: 32 x 12
   rowname       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 Mazda RX4    21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2 Mazda RX4 ~  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3 Datsun 710   22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4 Hornet 4 D~  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5 Hornet Spo~  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6 Valiant      18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7 Duster 360   14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8 Merc 240D    24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9 Merc 230     22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
```

```
10 Merc 280     19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# i 22 more rows
```

Viel besser. Mit diesem Tibble können wir nun viel einfacher filtern, nämlich mit der `filter()`-Funktion:

```
filter(mtcars_tibble, mpg > 30)
```

```
# A tibble: 4 x 12
  rowname        mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Fiat 128      32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
2 Honda Civic   30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
3 Toyota Coro~  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1
4 Lotus Europa  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2
```

Oben haben wir schon gesehen, dass wir zum Umformen des Dataframes in einen Tibble mehrere Schritte ineinander bauen müssen. Wollen wir nun noch den ursprünglichen Dataframe vor der Umformung verändern, kann das ziemlich schnell unübersichtlich werden:

```
# Wir wollen z.B. in einem Schritt den Dataframe in einen Tibble umwandeln und dann filter
filter(as_tibble(rownames_to_column(mtcars)), mpg > 30)
```

```
# A tibble: 4 x 12
  rowname        mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Fiat 128      32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
2 Honda Civic   30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
3 Toyota Coro~  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1
4 Lotus Europa  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2
```

```
# Unübersichtlich und fehleranfällig!
```

Stattdessen führen wir den sogenannten `Pipe`-Operator (geschrieben `%>%`) als neue Schreibweise ein. Man kann sich seine Funktion so vorstellen, dass er alles, was auf der linken Seite von ihm steht, nimmt und dem, was auf der rechten Seite steht, übergibt. Wir wollen den vorherigen Codeblock einfacher schreiben:

```r
# Lesen von links nach rechts:
mtcars %>%                      # Wir nehmen das mtcars-Datenset...
  rownames_to_column() %>%      # und lagern die Namen in eine eigene Spalte aus...
  as_tibble() %>%               # und konvertieren das in einen Tibble...
  filter(mpg > 30)              # und filtern nach Autos, die mehr als 30 MPG haben.
```

```
# A tibble: 4 x 12
  rowname        mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Fiat 128      32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
2 Honda Civic   30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
3 Toyota Coro~  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1
4 Lotus Europa  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2
```

Viel einfacher lesbar, oder? Wir werden diese Datenstruktur sehr häufig nutzen, eben weil wir diese Funktionen anwenden können.

Aufgabe 2

Wähle alle *außer* den ersten 15 Autos aus (`?slice` in die Konsole einzugeben könnte hilfreich sein, schaue dort nach einem Weg, Zeilen auszuschließen) und filtere dann nach den Autos mit mindestens 6 Zylindern. Sortiere dann die Autos nach Meilen pro Gallone (absteigend; `?arrange` könnte hilfreich sein, schaue dort unter den Beispielen nach einem Weg, absteigend zu sortieren). Zeige nur die Namen der Autos, die Meilen pro Gallone, die Zylinder und die PS an (`?select` könnte hilfreich sein).

```r
mtcars_tibble %>%                   # Tibble auswählen
  slice(-(1:15)) %>%                # Alle außer den ersten 15 Zeilen auswählen
  filter(cyl >= 6) %>%              # Alle Autos mit mindestens 6 Zylindern
  arrange(desc(mpg)) %>%            # Autos nach Meilen pro Gallone sortieren
  select(rowname, mpg, cyl, hp)     # Nur Name, MPG, Zylinder, PS anzeigen
```

Aufgabe 3

New York gehört mit seinen Flughäfen zu den weltweit bekanntesten Verkehrsdrehkreuzen. Es ist so bekannt, dass es in R eine eigene Library für die Aufzeichnung aller abgehenden Flüge im Jahr 2013 gibt.

Installiere für diese Aufgabe die `nycflights13`-Library und lade sie in den Arbeitsbereich. Uns interessiert vor allem der `flights`-Tibble.

1. Welche Flüge sind verfrüht abgehoben?
2. Welche Flüge sind verspätet abgehoben und haben die Verspätung aufgeholt?
3. Füge eine neue Variable `short_flight` zum Tibble hinzu, die den Wert 1 annimmt, wenn der Flug weniger als 2 Stunden Flugzeit hat, und ansonsten 0 ist (Hinweis: `?mutate` könnte hilfreich sein).
4. Welcher Anteil an verspätet abgehobenen Flügen, die die Verspätung aufgeholt haben, hatten weniger als zwei Stunden Flugzeit?

Tipp

Zu 3.: Nutze nach dem Filtern `mutate(... = if_else(...))`, um für jeden Flug die Variable `short_flight` zu erstellen.
Zu 4.: Verwende die Funktionen `summarise()`, `sum()` und `n()`, um den Anteil zu errechnen.

```r
library(nycflights13)

# 1
flights %>%
  filter(dep_delay < 0)

# 2
flights %>%
  filter(dep_delay > 0 & arr_delay <= 0)

# 3
flights %>%
  filter(dep_delay > 0 & arr_delay <= 0) %>%
  mutate(short_flight = if_else(air_time < 120, 1, 0))

# 4
flights %>%
  filter(dep_delay > 0 & arr_delay <= 0) %>%
  mutate(short_flight = if_else(air_time < 120, 1, 0)) %>%
  summarise(share_short = sum(short_flight)/n())
```

Liste

Info: *Dieses Kapitel folgt grob den Inhalten aus Kapitel 9 in* "Beginning Computer Science with R" *von Homer White.*

Die Datenstrukturen, die wir uns bisher angeschaut haben, waren *atomisch*, das heißt, sie

können pro Einheit (bspw. Spalte) nur eine Art Datentyp beinhalten. Nun gehen wir über zu den *Listen*, die Objekte aller Arten beinhalten können. Zahlen, Wörter, andere Listen, alles ist möglich.

Wir beginnen die Erkundung, indem wir mit `list()` selbst Listen erstellen.

```
liste1 <- list(
  modellname = "Audi A6",
  zulassung = as_date("15.05.2013", format = "%d.%m.%Y"),
  kilometerstand = 115290
)

tibble1 <- tibble(
  x = length(letters),
  y = letters
)

liste2 <- list(
  auto = liste1,
  buchstaben = tibble1
)

liste3 <- list(
  vokale = c("a", "e", "i", "o", "u"),
  listen = list(liste1, liste2)
)
```

Die hier zu sehenden Listen beinhalten Objekte unterschiedlicher Art, und sie können wiederum Listen enthalten. Wir lassen uns nun Liste 1 ausgeben:

```
liste1
```

```
$modellname
[1] "Audi A6"

$zulassung
[1] "2013-05-15"

$kilometerstand
[1] 115290
```

Wir sehen, dass jedes Element dieser Liste einen Namen (z.B. `$modellname`) hat. Man kann die Elemente dementsprechend auch über ihren Namen aufrufen:

```
liste1$modellname
```

```
[1] "Audi A6"
```

Wir schon bei Eingabe des Dollarzeichens Gebrauch von der Autovervollständigung machen: Die Elemente der Liste werden uns sofort angezeigt.

Leere Listen können auch erstellt werden, zum Beispiel, wenn wir noch nicht wissen, wie viele Elemente welcher Art gesammelt werden sollen:

```
leere_liste <- list()
```

Wissen wir hingegen, wie viele Elemente wir speichern wollen, können wir folgende Schreibweise verwenden:

```
liste <- vector(mode = "list", length = 5) # 5 durch gewünschte Länge ersetzen
```

Aufgabe 4

Erstelle eine Liste `uebe_liste` mit den folgenden drei Elementen:

- Alle geraden Zahlen von 2 bis 123 (Name: `gerade_zahlen`)
- Alle großen Buchstaben (Name: `alphabet_gross`)
- Die Billboard Top 100 aus dem Jahr 2000 (Zu finden unter `billboard`, Name: `top100`)

> **Tipp**
>
> Eine Abfolge von Zahlen mit einem bestimmten Abstand kann mit `seq()` erzeugt werden.

```
uebe_liste <- list(
  gerade_zahlen = seq(2, 123, by = 2),
  alphabet_gross = LETTERS,
  top100 = billboard
)
```

Aufgabe 5

Was passiert, wenn wir nach dem Erstellen der Liste aus Aufgabe 4 folgenden Code ausführst?

```r
uebe_liste <- c(uebe_liste, list(zahl = 5))
```

Lösung

Die Liste `uebe_liste` wird erweitert mit dem Element `zahl`.

Operationen auf Listen

Wollen wir auf einen Teil der Liste zugreifen, so können wir das genau wie bei Vektoren mit folgendermaßen machen:

```r
liste1[1:2] # Gibt die ersten beiden Elemente der Liste (modellname, zulassung)
```

```
$modellname
[1] "Audi A6"

$zulassung
[1] "2013-05-15"
```

Wir können auch auf genau ein Element der Liste zugreifen:

```r
vokale <- liste3[1]
```

Nehmen wir nun an, wir wollen den vierten Vokal aus `vokale` ausgeben. Versuchen wir es wie folgt:

```r
vokale[4]
```

```
$<NA>
NULL
```

erhalten wir die Ausgabe `$<NA> NULL`. Das liegt daran, dass beim Zugreifen auf eine Liste mit einer einfachen eckigen Klammer (`[...]`) eine Liste mit genau den entsprechenden Inhalten zurückgegeben wird. In unserem Fall ist das eine Liste mit einem Element:

```r
length(vokale)
```

```
[1] 1
```

Versuchen wir nun, auf das vierte Element (in dem Fall der Liste) zuzugreifen, funktioniert das nicht, weil die zurückgegebene Liste eben nur ein Element hat.

Wenn wir das Element selbst (und nicht eine Liste bestehend aus diesem Element) zurückgeben wollen, müssen wir doppelte eckige Klammern (`[[...]]`) benutzen:

```
vokale <- liste3[[1]]

vokale[4]
```

```
[1] "o"
```