Fundamental object types in R II: Vectors

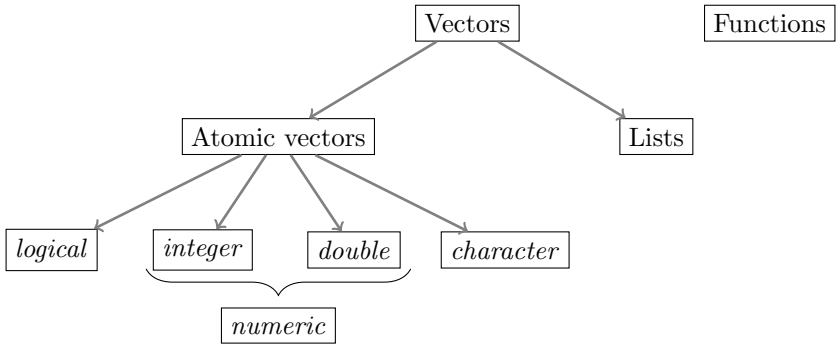
Claudius Gräbner-Radkowitsch

2/12/2022

Overview

We already learned that everything in R that exists is an *object*. You most likely already noted that there are different types of objects: `2`, for instance, was a number, but `assign` was a function.[1] As you might have guessed, there are many more types of objects. To understand the fundamental object types in R is a very essential prerequisite to master more complicated programming challenges than those we have encountered so far. Thus, this post is among those that will introduce you to the most important object types that you will encounter in R.

These data types are summarized in the following figure:



This post will be about the most common types of vectors. See the previous post for a treatment of functions, and the upcoming one for more advanced types of vectors, such as `factor`, `matrix`, and `data.frame`.

Vectors

Vectors are the most important object type in R - almost all data that we will work with in R are vectors of some sort. Within the class of vectors, the most important distinction is that between **atomic vectors** and **lists**, which are sometimes also called *generic vectors*.[2] Both atomic vectors and lists consist of one or more other objects. What distinguishes the two is that while *atomic vectors are composed only of objects of the same type*, lists can comprise objects of different types.

This makes it easy to classify atomic vectors in more detail: we usually say that the type of atomic vector is the type of the object it encompasses. Four major types of atomic vectors in this sense exist:

- `logical` (logical values): es gibt zwei logische Werte, `TRUE` und `FALSE`, welche auch mit `T` oder `F` abgekürzt werden können

[1] In fact, we will learn below that `2` is not really a number, but a vector or length 1. Only in a next step, `2` counts as a 'number', or, more precisely as a 'double'.

[2] The only object type that is of relevance to use aside these two is `NULL`. We will learn about it during the end of this post.

- `integer` (whole numbers): das sollte im Prinzip selbsterklärend sein, allerding muss den ganzen Zahlen in R immer der Buchstabe `L` folgen, damit die Zahl tatsächlich als ganze Zahl interpretiert wird.[3] Beispiele sind `1L`, `400L` oder `10L`.

- `double` (decimal numbers): auch das sollte selbsterklärend sein; Beispiele wären `1.5`, `0.0`, oder `-500.32`.
- Ganze Zahlen und Dezimalzahlen werden häufig unter der Kategorie `numeric` zusammengefasst. Dies ist in der Praxis aber quasi nie hilfreich und man sollte diese Kategorie möglichst nie verwenden.
- `character` (words): sie sind dadurch gekennzeichnet, dass sie auch Buchstaben enthalten können und am Anfang und Ende ein `"` haben. Beispiele hier wären `"Hallo"`, `"500"` oder `"1_2_Drei"`.

As indicated above, an atomic vector only comprises
ELEMENTS SAME TYPE BUT NA While it is not an atomic vector in the strict sense, another data type that we will encounter in the context of atomic vectors is `NA`. * Es gibt noch zwei weitere besondere 'Typen', die strikt gesehen keine atomaren Vektoren darstellen, allerdings in diesem Kontext schon häufig auftauchen: `NULL`, was strikt genommen ein eigener Datentyp ist und immer die Länge 0 hat, sowie `NA`, das einen fehlenden Wert darstellt.

Hieraus ergibt sich die in Abbildung @ref(fig:vektoren) aufgezeigte Aufteilung von Vektoren.

Wir werden nun die einzelnen Typen genauer betrachten. Vorher wollen wir jedoch noch die Funktion `typeof` einführen. Sie hilft uns in der Praxis den Typ eines Objekts herauszufinden. Dafür rufen wir einfach die Funktion `typeof` mit dem zu untersuchenden Objekt oder dessen Namen auf:

```
typeof(2L)
```

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

```
x <- 22.0
typeof(x)
```

```
## [1] "double"
```

Wir können auch explizit testen ob ein Objekt tatsächlich ein Objekt eines bestimmten Typs ist. Die generelle Syntax hierfür ist: `is.*()`, also z.B.:

```
x <- 1.0
is.integer(x)
```

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

```
is.double(x)
```

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

Diese Funktion gibt als Output also immer einen logischen Wert aus, je nachdem ob die Inputs des entsprechenden Typs sind oder nicht.

Bestimmte Objekte können auch in einen anderen Typ transformiert werden. Hier spricht man von `coercion` und die generelle Syntax hierfür ist: `as.*()`, also z.B.:

```
x <- "2"
print(
  typeof(x)
)
```

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

[3] Diese auf den ersten Blick merkwürdige Syntax hat historische Gründe: als der integer Typ in die R Programmiersprache eingeführt wurde war er sehr stark an den Typ `long integer` in der Programmiersprache 'C' angelehnt. In C wurde ein solcher 'long integer' mit dem Suffix 'l' oder 'L' definiert, diese Regel wurde aus Kompatibilitätsgründen auch für R übernommen, jedoch nur mit 'L', da man Angst hatte, dass 'l' mit 'i' verwechselt wird, was in R für die imaginäre Komponente komplexer Zahlen verwendet wird.

```
x <- as.double(x)
print(
  typeof(x)
)
```

```
## [1] "double"
```

Allerdings ist eine Transformation nicht immer möglich:

```
as.double("Hallo")
```

```
## Warning: NAs introduced by coercion
```

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

Da R nicht weiß wie man aus dem Wort 'Hallo' eine Dezimalzahl machen soll, transformiert R das Wort in einen 'Fehlenden Wert', der in R als `NA` bekannt ist und unten noch genauer diskutiert wird.

Für die Grundtypen ergibt sich folgende logische Hierachie an trivialen Transformationen: `logical` → `integer` → `double` → `character`, d.h. man kann eine Dezimalzahl ohne Probleme in ein Wort transformieren, aber nicht umgekehrt:

> **Exkurs: Warum überhaupt transformieren?** Für eine Programmiersprache sind Datentypen extrem wichtig, weil sonst unklar bliebe wie mathematische Operationen auf unterschiedliche Objekte wie Zahlen oder Wörter anzuwenden wären. Selbst transformieren werden Sie Objekte vor allem wenn Sie eine bestimmte, nur für eine bestimmte Objektart definierte Operation verwenden wollen und das Objekt bislang als ein anderer Typ gespeichert ist. Das kann zum Beispiel passieren wenn Sie Daten einlesen oder Wörter selbst in Zahlenwerte übersetzen. Wenn in Ihrem Code unerwartete Fehler mit kryptischen Fehlermeldungen auftauchen ist es immer eine gute Idee, erst einmal die Typen der verwendeten Objekte zu checken und die Objekte ggf. zu transformieren.

```
x <- 2
y <- as.character(x)
print(y)
```

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

```
z <- as.double(y) # Das funktioniert
print(z)
```

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

```
k <- as.double("Hallo") # Das nicht
```

```
## Warning: NAs introduced by coercion
```

```
print(k)
```

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

Bei der Transformation logischer Werte wird `TRUE` übrigens zu `1` und `FALSE` zu `0`, eine Tatsache, die wir uns später noch zunutze machen werden:

```
x <- TRUE
as.integer(x)
```

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

```
y <- FALSE
as.integer(y)
```

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

Da nicht immer ganz klar ist wann R bei Transformationen entgegen der gerade eingeführten Hierachie eine Warnung ausgibt und wann nicht sollte man hier immer besondere Vorsicht walten lassen!

Zudem ist bei jeder Transformation Vorsicht geboten, da sie häufig Eigenschaften der Objekte implizit verändert. So führt eine Transformation von einer Dezimalzahl hin zu einer ganzen Zahl teils zu unerwartetem Rundungsverhalten:

```
x <- 1.99
as.integer(x)
```

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

Auch führen Transformationen, die der eben genannten Hierachie zuwiderlaufen, nicht zwangsweise zu Fehlern, sondern 'lediglich' zu unerwarteten Änderungen, die in jedem Fall vermieden werden sollten:

```
z <- as.logical(99)
print(z)
```

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

Häufig transformieren Funktionen ihre Argumente automatisch, was meistens hilfreich ist, manchmal aber auch gefährlich sein kann:

```
x <- 1L # Integer
y <- 2.0 # Double
z <- x + y
typeof(z)
```

```
## [1] "double"
```

Bei einer Addition werden logische Werte ebenfalls automatisch transformiert:

```
x <- TRUE
y <- FALSE
z <- x + y # TRUE wird zu 1, FALSE zu 0
print(z)
```

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

Daher sollte man immer den Überblick behalten, mit welchen Objekttypen man gerade arbeitet.

Einen Überblick zu den Test- und Transformationsbefehlen finden Sie in Tabelle @ref(tab:artentests).

Table 1: (#tab:artentests) Ein Überblick zu Test- und Transformationsbefehlen in R.

Typ	Test	Transformation
logical	`is.logical`	`as.logical`
double	`is.double`	`as.double`
integer	`is.integer`	`as.integer`
character	`is.character`	`as.character`
function	`is.function`	`as.function`
NA	`is.na`	NA
NULL	`is.null`	`as.null`

Ein letzter Hinweis zu **Skalaren**. Unter Skalaren verstehen wir in der Regel 'einzelne Zahlen', z.B. `2`. Dieses Konzept gibt es in R nicht. `2` ist ein Vektor der Länge 1. Wir unterscheiden also vom Typ her nicht zwischen einem Vektor, der nur ein oder mehrere Elemente hat.

Hinweis: Um längere Vektoren zu erstellen, verwenden wir die Funktion `c()`:

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

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

Dabei können Vektoren auch miteinander verbunden werden:

```r
x <- 1:3 # Shortcut für: x <- c(1, 2, 3)
y <- 4:6
z <- c(x, y)
z
```

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

Da atomare Vektoren immer nur Objekte des gleichen Typs enthalten, könnte man erwarten, dass es zu einem Fehler kommt, wenn wir Objete unterschiedlichen Type kombinieren wollen:

```r
x <- c(1, "Hallo")
```

Tatsächlich transformiert R die Objekte allerdings nach der oben beschriebenen Hierachie `logical` → `integer` → `double` → `character`. Da hier keine Warnung oder kein Fehler ausgegeben wird, sind derlei Transformationen eine gefährliche Fehlerquelle!

Hinweis: Die Länge eines Vektors kann mit der Funktion `length` bestimmt werden:

```r
x =  c(1, 2, 3)
len_x <- length(x)
len_x
```

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

> **Exkurs: Wie groß kann eine ganze Zahl sein?** In R werden `integer` als 32-Bit Daten gespeichert. Das bedeutet, dass für einen einzelnen `integer` 32 Bit Speicherplatz zur Verfügung steht. Das bedeutet, dass sehr große ganze Zahlen nicht als `integer` gespeichert werden können, einfach weil die 32 Bit nicht ausreichen.

```r
x <- 2147483647L
typeof(x)
```

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

```r
y <- 2147483648L
typeof(y)
```

```
## [1] "double"
```

> Mit 32-Bit Integern kann man also maximal die Zahl 2147483647 speichern. Größere Zahlen können nur als `double` gespeichert werden. Das geht allerdings möglicherweise mit einem Verlust an Präzision einher. Wenn man letzteres vermeiden möchte kann man auch 64-Bit Integer verwenden. Diese wurden nachträglich in R eingeführt um auch sehr große Zahlen als Integer speichern zu können. Das geht über das Paket `bit64` (mehr zu Paketen unten in Abschnit @ref(es:pakete)):

```r
z <- bit64::as.integer64(2147483648)
bit64::is.integer64(z)
```

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

> Da aber dieser Datentyp später hinzugefügt wurde funktionieren einige Funktionen damit nicht, wenn das Paket `bit64` nicht installiert ist und einige Standard- Funktionen geben irreführende Ergebnisse, z.B.:

```
typeof(z)
```

```
## [1] "double"
```

> Deswegen, und weil `bit64` nicht Teil der Standard-Installation von R ist, sollten wir sehr große ganze Zahlen und die Verwendung vom Datentyp `integer64` dringend vermeiden, es sei denn wir haben gute Gründe dazu. Sehr große ganze Zahlen sollten also entweder als Dezimalzahlen gespeichert werden wenn kleine Verluste an Präzision keine Rolle spielen, oder aber sie sollten angemessen skaliert werden.

Logische Werte (logical)

Die logischen Werte `TRUE` und `FALSE` sind häufig das Ergebnis von logischen Abfragen, z.B. 'Ist 2 größer als 1?'. Solche Abfragen kommen in der Forschungspraxis häufig vor und es macht Sinn, sich mit den häufigsten logischen Operatoren vertraut zu machen. Einen Überblick finden Sie in Tabelle @ref(tab:logicaloperators).

Table 2: (#tab:logicaloperators) Zentrale logische Abfragen in R.

Operator	Funktion in R	Beispiel
größer	`>`	`2>1`
kleiner	`<`	`2<4`
gleich	`==`	`4==3`
größer gleich	`>=`	`8>=8`
kleiner gleich	`<=`	`5<=9`
nicht gleich	`!=`	`4!=5`
und	`&`	`x<90 & x>55`
oder	`|`	`x<90 | x>55`
entweder oder	`xor()`	`xor(2<1, 2>1)`
nicht	`!`	`!(x==2)`
ist wahr	`isTRUE()`	`isTRUE(1>2)`

Das Ergebnis eines solches Tests ist immer ein logischer Wert:

```
x <- 4
y <- x == 8
typeof(y)
```

```
## [1] "logical"
```

Es können auch längere Vektoren getestet werden:

```
x <- 1:3
x<2
```

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

Tests können beliebig miteinander verknüpft werden:

```
x <- 1L
x>2 | x<2 & (is.double(x) & x!=0)
```

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

Da für viele mathematischen Operationen `TRUE` als die Zahl `1` interpretiert wird, ist es einfach zu testen wie häufig eine bestimmte Bedingung erfüllt ist:

```
x <- 1:50
smaller_20 <- x<20
```

```
print(
  sum(smaller_20) # Wie viele Elemente sind kleiner als 20?
  )
```

```
## [1] 19
```

```
print(
  sum(smaller_20/length(x)) # Wie hoch ist der Anteil von diesen Elementen?
)
```

```
## [1] 0.38
```

Wörter (character)

Wörter werden in R dadurch gebildet, dass an ihrem Anfang und Ende das Symbol ' oder " steht:

```
x <- "Hallo"
typeof(x)
```

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

```
y <- 'Auf Wiedersehen'
typeof(y)
```

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

Wie andere Vektoren können sie mit der Funktion `c()` verbunden werden:

```
z <- c(x, "und", y)
z
```

```
## [1] "Hallo"           "und"             "Auf Wiedersehen"
```

Nützlich ist in diesem Zusammenhang die Funktion `paste()`, die Elemente von mehreren Vektoren in Wörter transformiert und verbindet:

```
x <- 1:10
y <- paste("Versuch Nr.", x)
y
```

```
##  [1] "Versuch Nr. 1"  "Versuch Nr. 2"  "Versuch Nr. 3"  "Versuch Nr. 4"
##  [5] "Versuch Nr. 5"  "Versuch Nr. 6"  "Versuch Nr. 7"  "Versuch Nr. 8"
##  [9] "Versuch Nr. 9"  "Versuch Nr. 10"
```

Die Funktion `paste()` akzeptiert ein optionales Argument `sep`, mit dem wir den Wert angeben können, der zwischen die zu verbindenden Elemente gesetzt wird (der Default ist `sep=" "`):

```
tag_nr <- 1:10
x_axis <- paste("Tag", tag_nr, sep = ": ")
x_axis
```

```
##  [1] "Tag: 1"  "Tag: 2"  "Tag: 3"  "Tag: 4"  "Tag: 5"  "Tag: 6"  "Tag: 7"
##  [8] "Tag: 8"  "Tag: 9"  "Tag: 10"
```

> Hinweis: Hier haben wir ein Beispiel für das sogenannte 'Recycling' gesehen: da der Vektor `c("Tag")` kürzer war als der Vektor `tag_nr` wird `c("Tag")` einfach kopiert damit die Operation mit `paste()` Sinn ergibt. Recycling ist oft praktisch, aber manchmal auch schädlich, nämlich dann, wenn man eigentlich davon ausgeht eine Operation mit zwei gleich langen Vektoren durchzuführen, dies aber tatsächlich nicht tut. In einem solchen Fall führt Recycling dazu, dass keine Fehlermeldung ausgegeben wird. Ein Beispiel dafür gibt folgender Code, in dem die Intention klar die Verbindung aller Wochentage zu Zahlen ist und einfach ein Wochentag vergessen wurde:

```r
tage <- paste("Tag ", 1:7, ":", sep="")
tag_namen <- c("Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag")
paste(tage, tag_namen) # default ist sep=" "
```

```
## [1] "Tag 1: Montag"     "Tag 2: Dienstag"   "Tag 3: Mittwoch"
## [4] "Tag 4: Donnerstag" "Tag 5: Freitag"    "Tag 6: Samstag"
## [7] "Tag 7: Montag"
```

Fehlende Werte und NULL

Fehlende Werte werden in R als `NA` kodiert. `NA` erfüllt gerade in statistischen Anwendungen eine wichtige Rolle, da ein bestimmter Platz in einem Vektor aktuell fehlend sein müsste, aber als Platz dennoch existieren muss.

> **Beispiel:** Der Vektor `x` enthält einen logischen Wert, der zeigt ob eine Person die Fragen auf einem Fragebogen richtig beantwortet hat. Wenn die Person die dritte Frage auf dem Fragebogen nicht beantwortet hat, sollte dies durch `NA` kenntlich gemacht werden. Einfach den Wert komplett wegzulassen macht es im Nachhinein unmöglich festzustellen *welche* Frage die Person nicht beantwortet hat.

Die meisten Operationen die `NA` als einen Input bekommen geben auch als Output `NA` aus, weil unklar ist wie die Operation mit unterschiedlichen Werten für den fehlenden Wert ausgehen würde:

```r
5 + NA
```

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

Einzige Ausnahmen sind Operationen, die unabhängig vom fehlenden Wert einen bestimmten Wert annehmen:

```r
NA | TRUE # Gibt immer TRUE, unabhängig vom Wert für NA
```

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

Um zu testen ob ein Vektor `x` fehlende Werte enthält sollte die Funktion `is.na` verwendet werden, und nicht etwa der Ausdruck `x==NA`:

```r
x <- c(NA, 5, NA, 10)
print(x == NA) # Unklar, da man nicht weiß, ob alle NA für den gleichen Wert stehen
```

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

```r
print(
  is.na(x)
)
```

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

Wenn eine Operation einen nicht zu definierenden Wert ausgibt, ist das Ergebnis nicht `NA` sondern `NaN` (*not a number*):

```r
0 / 0
```

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

Eine weitere Besonderheit ist `NULL`, welches in der Regel als Vektor der Länge 0 gilt, aber häufig zu besonderen Zwecken verwendet wird:

```r
x <- NULL
length(x)
```

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

`NULL` wird häufig verwendet um zu signalisieren, dass etwas nicht existiert. So ist ein leerer Vektor `NULL`:

```
x <- c()
x
```

```
## NULL
```

```
length(x)
```

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

Damit unterscheidet er sich von einem Vektor mit einem (oder mehreren) fehlenden Werten:

```
y <- NA
length(y)
```

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

Auch im Programmieren von Funktionen wird `NULL` häufig für optionale Argumente verwendet. Solche fortgeschrittene Konzepte werden aber erst an späterer Stelle behandelt. Für jetzt reicht die Idee, `NULL` als einen Vektor der Länge 0 zu verstehen.

Indizierung und Ersetzung

Einzelne Elemente von atomaren Vektoren können mit eckigen Klammern extrahiert werden:

```
x <- c(2,4,6)
x[1]
```

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

Auf diese Weise können auch bestimmte Elemente modifiziert werden:

```
x <- c(2,4,6)
x[2] <- 99
x
```

```
## [1]  2 99  6
```

Es kann auch mehr als ein Element extrahiert werden:

```
x[1:2]
```

```
## [1]  2 99
```

Negative Indizes sind auch möglich, diese eliminieren die entsprechenden Elemente:

```
x[-1]
```

```
## [1] 99  6
```

Um das letzte Element eines Vektors zu bekommen verwendet man einen Umweg über die Funktion `length()`:

```
x[length(x)]
```

```
## [1] 6
```

Nützliche Funktionen für atomare Vektoren

Hier sollen nur einige Funktionen erwähnt werden, die im Kontext von atomaren Vektoren besonders praktisch sind,[4] inbesondere wenn es darum geht solche Vektoren herzustellen, bzw. Rechenoperationen mit ihnen durchzuführen.

[4] Für viele typische Aufgaben gibt es in R bereits eine vordefinierte Funktion. Am einfachsten findet man diese durch googlen.

Herstellung von atomaren Vektoren:

Eine Sequenz ganzer Zahlen wird in der Regel sehr häufig gebraucht. Entsprechend gibt es den hilfreichen Shortcut `:`, den wir bei der Besprechung von Vektoren bereits kennengelernt haben:

```
x <- 1:10
x
```

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

```
y <- 10:1
y
```

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

Häufig möchten wir jedoch eine kompliziertere Sequenz bauen. In dem Fall hilft uns die allgemeinere Funktion `seq()`:

```
x <- seq(1, 10)
print(x)
```

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

In diesem Fall ist `seq()` äquivalent zu `:`. Die Funktion `seq` erlaubt aber mehrere optionale Argumente: so können wir mit `by` die Schrittlänge zwischen den einzelnen Zahlen definieren.

```
y <- seq(1, 10, by = 0.5)
print(y)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
## [16]  8.5  9.0  9.5 10.0
```

Wenn wir die Länge des resultierenden Vektors festlegen wollen und die Schrittlänge von R automatisch festgelegt werden soll, können wir dies mit dem Argument `length.out` machen:

```
z <- seq(2, 8, length.out = 4)
print(z)
```

```
## [1] 2 4 6 8
```

Und wenn wir einen Vektor in der Länge eines anderen Vektors erstellen wollen, bietet sich das Argument `along.with` an. Dies wird häufig für das Erstellen von Indexvektoren verwendet.[5] In einem solchen Fall müssen wir die Indexzahlen nicht direkt angeben:

```
z_index <- seq(along.with = z)
print(z_index)
```

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

Auch häufig möchten wir einen bestimmten Wert wiederholen. Das geht mit der Funktion `rep`:

```
x <- rep(NA, 5)
print(x)
```

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

Rechenoperationen

Es gibt eine Reihe von Operationen, die wir sehr häufig gemeinsam mit Vektoren anwenden. Häufig interessiert und die **Länge** eines Vektors. Dafür können wir die Funktion `length()` verwenden:

[5] Ein Indexvektor `x` zu einem beliebigen Vektor `y` mit `N` Elementen enthält die ganzen Zahlen von 1 bis `N`. Der n-te Wert von x korrespontiert also zum Index des n-ten Wert von `y`.

```r
x <- c(1,2,3,4)
length(x)
```

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

Wenn wir den **größten** oder **kleinsten Wert** eines Vektors erfahren möchten geht das mit den Funktionen `min()` und `max()`:

```r
min(x)
```

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

```r
max(x)
```

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

Beide Funktionen besitzen ein optionales Argument `na.rm`, das entweder `TRUE` oder `FALSE` sein kann. Im Falle von `TRUE` werden alle `NA` Werte für die Rechenoperation entfernt:

```r
y <- c(1,2,3,4,NA)
min(y)
```

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

```r
min(y, na.rm = TRUE)
```

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

Den **Mittelwert** bzw die **Varianz/Standardabweichung** der Elemente bekommen wir mit `mean()`, `var()`, bzw. `sd()`, wobei alle Funktionen auch das optionale Argument `na.rm` akzeptieren:

```r
mean(x)
```

```
## [1] 2.5
```

```r
var(y)
```

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

```r
var(y, na.rm = T)
```

```
## [1] 1.666667
```

Ebenfalls häufig sind wir an der **Summe**, bzw, dem **Produkt** aller Elemente des Vektors interessiert. Die Funktionen `sum()` und `prod()` helfen weiter und auch sie kennen das optionale Argument `na.rm`:

```r
sum(x)
```

```
## [1] 10
```

```r
prod(y, na.rm = T)
```

```
## [1] 24
```

Listen

Im Gegensatz zu atomaren Vektoren können Listen Objekte verschiedenen Typs enthalten. Sie werden mit der Funktion `list()` erstellt:

```r
l_1 <- list(
  "a",
  c(1,2,3),
  FALSE
```

```
)
typeof(l_1)
```

```
## [1] "list"
```

```
l_1
```

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

Wir können Listen mit der Funktion `str()` (kurz für "structure") inspizieren. In diesem Fall erhalten wir unmittelbar Informationen über die Art der Elemente:

```
str(l_1)
```

```
## List of 3
##  $ : chr "a"
##  $ : num [1:3] 1 2 3
##  $ : logi FALSE
```

Die einzelnen Elemente einer Liste können auch benannt werden:

```
l_2 <- list(
  "erstes_element" = "a",
  "zweites_element" = c(1,2,3),
  "drittes_element" = FALSE
)
```

Die Namen aller Elemente in der Liste erhalten wir mit der Funktion `names()`:

```
names(l_2)
```

```
## [1] "erstes_element"  "zweites_element" "drittes_element"
```

Um einzelne Elemente einer Liste auszulesen müssen wir `[[` anstatt `[` verwenden. Wir können dann Elemente entweder nach ihrer Position oder nach ihren Namen auswählen:

```
l_2[[1]]
```

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

```
l_2[["erstes_element"]]
```

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

Im Folgenden wollen wir uns noch mit drei speziellen Typen beschäftigen, die weniger fundamental als die bislang diskutierten Typen sind, jedoch häufig in der alltäglichen Arbeit vorkommen: Faktoren, Matrizen und Data Frames.