Erste Schritte mit R

Applied Statistics – A Practical Course

Thomas Petzoldt

2025-04-04

1 Einführung

Der folgende Abschnitt soll einen ersten Eindruck vermitteln, was **R** (R Core Team, 2021) ist und wie es funktioniert. Es wird davon ausgegangen, dass die folgende Software installiert ist:

1. Das R-System für statistische Berechnungen: https://www.r-project.org
2. R Studio, ein Programm, das die Arbeit mit R vereinfacht: https://www.rstudio.org

Anmerkung: Bitte installiere zuerst **R** bevor du **RStudio** installierst.

Ein gutes Beispiel für den Anfang ist das kurze Video „R tutorial - The True Basics of R“ von DataCamp, siehe https://youtu.be/SWxoJqTqo08. Außerdem bietet Datacamp ausgezeichnete **R**-Kurse, die zum Teil sogar kostenlos sind. Ein weiteres nützliches **R**-Tutorial gibt es bei W3Schools.com.

2 Erste Schritte

2.1 Programmstart und Hilfesystem

Der einfachste Weg, **R** zu erlernen, ist das kreative Verstehen und Abwandeln von gegebenen Beispielen, die Verwendung von **R** zur Lösung praktischer Probleme und die Diagnose von Warnungen und Fehlermeldungen. Keine Sorge: Fehlermeldungen sind ein normales Phänomen im wissenschaftlichen Rechnen und kein Hinweis auf eine Funktionsstörung des Computers oder des menschlichen Gehirns. Das Gegenteil ist der Fall: Ein gewisses Maß an Stresshormonen hilft, dauerhafte Lerneffekte zu erzielen. Nach einer gewissen Erfahrung ist dann die offizielle **R**-Dokumentation „An Introduction to R“ (Venables et al., 2021) oder ein gutes **R**-Buch sehr zu empfehlen.

Die ersten Abschnitte dieses „Crashkurses“ sollen einen Überblick über einige der wichtigsten Elemente von **R** und einen Einblick in einen typischen Arbeitsablauf geben, der für die

ersten statistischen Analysen und als Ausgangspunkt für die selbstständige Einarbeitung nützlich sein kann.

Wir beginnen unsere erste Sitzung mit dem Start von **RStudio**, einer plattformunabhängigen Benutzerumgebung, die das Arbeiten mit R erleichtert. RStudio unterteilt den Bildschirm in 3 (bzw. 4) Fenster (sogenannte Panes), von denen einige zusätzliche Tabs haben, um zwischen verschiedenen Ansichten zu wechseln.

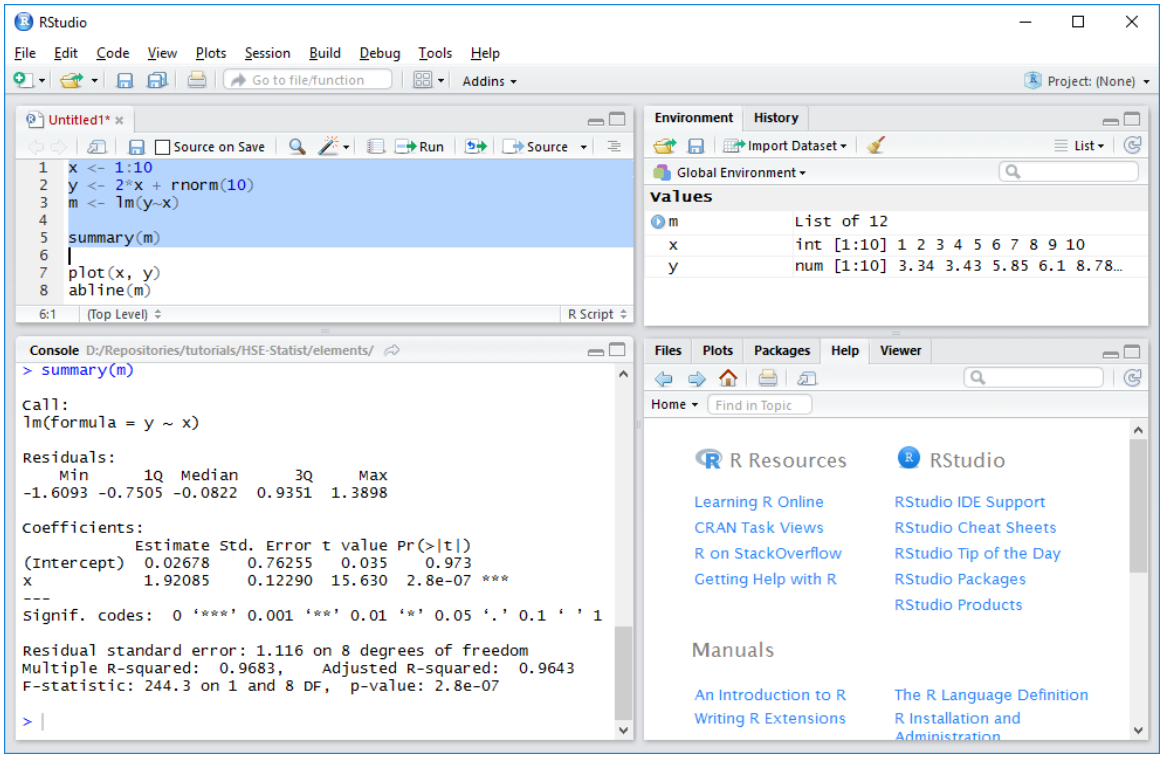


Abbildung 1: R Studio mit 4 Fenstern. Benutze **Datei – Neues R-Skript**, um das Quellcode-Fenster zu öffnen (siehe oben links). Gib dann etwas Code ein und vergiss nicht, im Hilfesystem zu lesen.

In einer neuen RStudio-Sitzung sollte ein „Pane“ die Haupt-Hilfsseite von **R** sein. Es ist eine gute Idee, ein wenig herumzustöbern, um einen Eindruck über den Umfang und den typischen Stil der verfügbaren Hilfsthemen zu bekommen. Die wichtigsten Abschnitte sind „An Introduction to R“, „Search Engine & Keywords“, „Packages“, die „Frequently Asked Questions“ und eventuell „**R** Data Import/Export“.

Wir beginnen nun, das **R**-System selbst zu erkunden.

2.2 R als Taschenrechner

Wenn man einen arithmetischen Ausdruck eingibt:

```
2 + 4
```

sieht man, dass **R** als Taschenrechner verwendet werden kann, der das Ergebnis sofort ausgibt:

```
[1] 6
```

Anstatt das Ergebnis auf dem Bildschirm auszugeben, kann man das Ergebnis mit dem Zuweisungsoperator „<-“ in einer benannten **Variablen** speichern:

```
a <- 2 + 4
```

Es scheint nichts zu passieren, aber das Ergebnis wird nun in der Variablen `a` gespeichert, die jederzeit durch Eingabe des Variablennamens allein aufgerufen werden kann:

```
a
```

Variablennamen in **R** beginnen immer mit einem Buchstaben (oder für besondere Zwecke mit einem Punkt), gefolgt von weiteren Buchstaben, Ziffern, Punkten oder Unterstrichen. Es wird zwischen Klein- und Großbuchstaben unterschieden, d.h. die Variablen `value`, `Value` und `VALUE` können unterschiedliche Daten enthalten. Einige wenige Zeichenkombinationen sind **reservierte Wörter** und können nicht als Variablen verwendet werden:

`break`, `for`, `function`, `if`, `in`, `next`, `repeat`, `while` und “`...`” (drei Punkte).

Andere Bezeichner wie `plot` können umdefiniert werden, doch dies sollte mit Vorsicht geschehen, um unerwünschte Verwechslungen und Nebenwirkungen zu vermeiden.

2.3 Vektoren

Vielleicht hast du bemerkt, dass die Ausgabe des obigen Beispiels ein führendes `[1]` hat. Das bedeutet, dass die Zeile mit dem ersten Element von `a` beginnt. Dies bringt uns zu einer sehr wichtigen Eigenschaft von **R**, dass Variablen mehr als nur einzelne Werte enthalten können: Vektoren, Matrizen, Listen, Dataframes (Tabellen) und so weiter.

Der grundlegendste Datentyp ist der Vektor, der mit Hilfe der Funktion „c“ (combine) mit Daten gefüllt werden kann:

```
values <- c(2, 3, 5, 7, 8.3, 10)
values
```

```
[1]  2.0  3.0  5.0  7.0  8.3 10.0
```

Um eine Folge von Werten zu erstellen, kann man den `:` (Doppelpunkt) verwenden:

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

oder, noch flexibler, die Funktion `seq`:

```r
x <- seq(2, 4, 0.25)
x
```

Sequenzen von wiederholten gleichen Werten können mit `rep` erzeugt werden:

```r
x <- rep(2, 4)
x
```

2.4 Übung

Es gibt viele Möglichkeiten, diese Funktionen zu verwenden, versuche zum Beispiel:

```r
seq(0, 10)
seq(0, 10, by = 2)
seq(0, pi, length = 12)
rep(c(0, 1, 2, 4, 9), times = 5)
rep(c(0, 1, 2, 4, 9), each = 2)
rep(c(0, 1, 2, 4, 9), each = 2, times = 5)
```

2.5 Zugriff auf Vektorelemente

Anstatt auf Vektoren als Ganzes zuzugreifen, ist es auch möglich, einzelne Elemente zu extrahieren, wobei der Index der angeforderten Daten ebenfalls ein Vektor sein kann:

```r
values[5]
values[2:4]
values[c(1, 3, 5)]
```

Manchmal haben die Elemente eines Vektors individuelle Namen, was den Zugriff erleichtert:

```r
named <- c(a = 1, b = 2.3, c = 4.5)
named
named["a"]
```

In **R** (und im Gegensatz zu anderen Sprachen wie **C**/**C++**) beginnen Vektorindizes mit 1. Negative Indizes sind ebenfalls möglich, haben aber den speziellen Zweck, ein oder mehrere Elemente zu löschen:

```
values[-3]
```

Es ist auch möglich, einen gegebenen Vektor durch Voranstellen oder Anhängen von Werten mit der Combine-Funktion (`c`) zu erweitern:

```
c(1, 1, values, 0, 0)
```

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

```
length(values)
```

Auch leere Vektoren sind möglich, d. h. Vektoren, die zwar existieren, aber keine Werte enthalten. Hier bedeutet das Schlüsselwort `NULL` „nichts“ im Gegensatz zu „0“ (Null), was die Länge 1 hat:

```
values <- NULL
values
length(values)
```

Solche leeren Vektoren werden manchmal als „Container“ zum schrittweisen Anhängen von Daten verwendet:

```
values <- NULL
values
length(values)
values <- c(values, 1)
values
values <- c(values, 1.34)
values
```

Soll ein Datenelement vollständig entfernt werden, so kann dies mit der Funktion `remove` (kurz `rm`) geschehen:

```
rm(values)
values
Error: Object "values" not found
```

Der gesamte Arbeitsbereich kann über das Menü von **R** oder **RStudio** (Session – Clear workspace) oder über die Kommandozeile mit `rm` (remove) gelöscht werden:

```
rm(list = ls(all = TRUE))
```

Die **R**-Sitzung kann wie gewohnt über das Menü oder durch Eingabe beendet werden:

```
q()
```

Manchmal und abhängig von der Konfiguration fragt **R**, ob der „**R** Workspace“ auf der Festplatte gespeichert werden soll. Dies kann nützlich sein, um die Arbeit zu einem späteren Zeitpunkt fortzusetzen, birgt aber das Risiko, den Arbeitsbereich zu überladen und bei einer späteren Sitzung nicht reproduzierbare Ergebnisse zu erhalten, so dass es empfehlenswert ist, vorerst „Nein“ zu sagen, es sei denn, man weiß genau, warum.

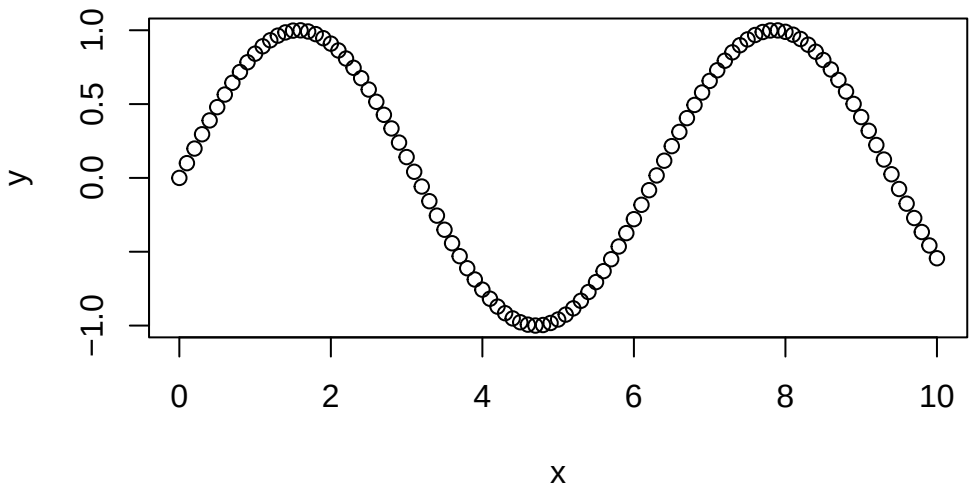
Später werden wir lernen, wie man nur die Daten (und Befehle) speichert, die benötigt werden.

3 Grafiken

3.1 R als Funktionsplotter

Nun wollen wir sehen, wie man **R** als Funktionsplotter verwendet, indem man Sinus- oder Kosinusfunktionen innerhalb eines Intervalls zwischen 0 und 10 zeichnet. Zuerst erstellen wir zwei Vektoren mit `x` und `y`. Um eine glatte Kurve zu erhalten, ist es sinnvoll, eine kleine Schrittweite zu wählen. Als Faustregel empfehle ich immer, etwa 100…400 kleine Schritte zu verwenden. Das ist ein guter Kompromiss zwischen Glättung und Speicherbedarf. Im Folgenden setzen wir die Schrittweite deshalb auf 0.1:

```
x <- seq(0, 10, 0.1)
y <- sin(x)
plot(x, y)
```



Anstatt Punkte zu plotten, können wir auch durchgehende Linien zeichnen. Dies wird durch die Angabe eines optionalen Arguments angegeben `type = "l"`.

Anmerkung: das hier verwendete Symbol für `type` ist der **kleine Buchstabe „L“** für ‘line’ und nicht die – im Druck sehr ähnliche – Zahl “1” (eins)!

Wir sehen auch, dass optionale Argumente wie `type` als „keyword = value“ Paar angegeben werden können. Dies hat den Vorteil, dass die Reihenfolge der Argumente keine Rolle spielt, da die Argumente durch ihren Namen referenziert werden:

```
plot(x, y, type = "l")
```

Jetzt wollen wir eine Kosinusfunktion mit einer anderen Farbe zeichnen. Dies kann mit einer der Funktionen `lines` oder `points` erreicht werden, um Linien oder Punkte zu einem bestehenden Plot hinzuzufügen:

```
y1 <- cos(x)
lines(x, y1, col = "red")
```

Mit Hilfe von `text` ist es auch möglich, beliebigen Text hinzuzufügen, indem man zuerst die x- und y-Koordinaten und dann den Text angibt:

```
x1 <- 1:10
text(x1, sin(x1), x1, col = "green")
```

Es gibt noch viele weitere Optionen, um das Verhalten der Grafikfunktionen zu ändern. Im Folgenden werden benutzerdefinierte Koordinatengrenzen (`xlim`, `ylim`), Achsenbeschriftungen und eine Überschrift (`xlab`, `ylab`, `main`) angegeben:

```
plot(x, y, xlim = c(-10, 10), ylim = c(-2, 2),
    xlab = "x-Values", ylab = "y-Values", main = "Example Graphics")
```

Codeformatierung und Zeilenumbrüche

Das obige Beispiel zeigt einen ziemlich langen Befehl, der möglicherweise nicht auf eine einzelne Zeile passt. In solchen Fällen zeigt **R** ein `+` (Pluszeichen) an, um anzuzeigen, dass ein Befehl fortgesetzt werden muss, z.B. weil noch eine schließende Klammer oder ein schließendes Anführungszeichen fehlt. Ein solches `+` am Anfang einer Zeile ist ein automatischer „Prompt“ ähnlich dem gewöhnlichen `>`-Prompt und muss niemals manuell eingegeben werden. Wenn die „+“-Fortsetzungsaufforderung jedoch versehentlich auftritt, drücke „ESC“, um diesen Modus abzubrechen.

Im Gegensatz zur Fortsetzungsaufforderung ist es auch möglich, mehrere Befehle in eine Zeile zu schreiben, getrennt durch ein Semikolon „`;`“. Das ist in einigen Fällen sinnvoll, aber im Allgemeinen ist es viel besser, den Skript-Editor zu verwenden und dann:

- jeden Befehl in eine eigene Zeile zu schreiben
- lange Zeilen mit mehr als 80 Zeichen zu vermeiden
- eine angemessene Einrückung zu verwenden, z.B. 2 Zeichen pro Einrückungsebene

- Leerzeichen einzufügen, um die Lesbarkeit des Codes zu verbessern, z.B. vor und nach dem Zuweisungsoperator `<-`.

Schließlich bedeutet ein Zahlensymbol (oder eine Raute) `#`, dass eine komplette Zeile oder der Teil der Zeile, der auf `#` folgt, ein Kommentar ist und von **R** ignoriert wird.

3.2 Zusätzliche Plotting-Optionen

Um die Fülle der grafischen Funktionen zu erkunden, lohnt es sich nun, einen Blick in die Online-Hilfe zu werfen, insbesondere in Bezug auf `?plot` oder `?plot.default`, und ein wenig mit verschiedenen Plot-Parametern zu experimentieren, wie `lty`, `pch`, `lwd`, `type`, `log` usw. **R** enthält unzählige Möglichkeiten, um die volle Kontrolle über den Stil und Inhalt von Grafiken zu bekommen, z.B. mit benutzerdefinierten Achsen (`axis`), Legenden (`legend`) oder benutzerdefinierten Linien und Flächen (`abline`, `rect`, `polygon`). Der allgemeine Stil von Abbildungen wie (Schriftgröße, Ränder, Linienbreite) kann mit der Funktion `par` beeinflusst werden.

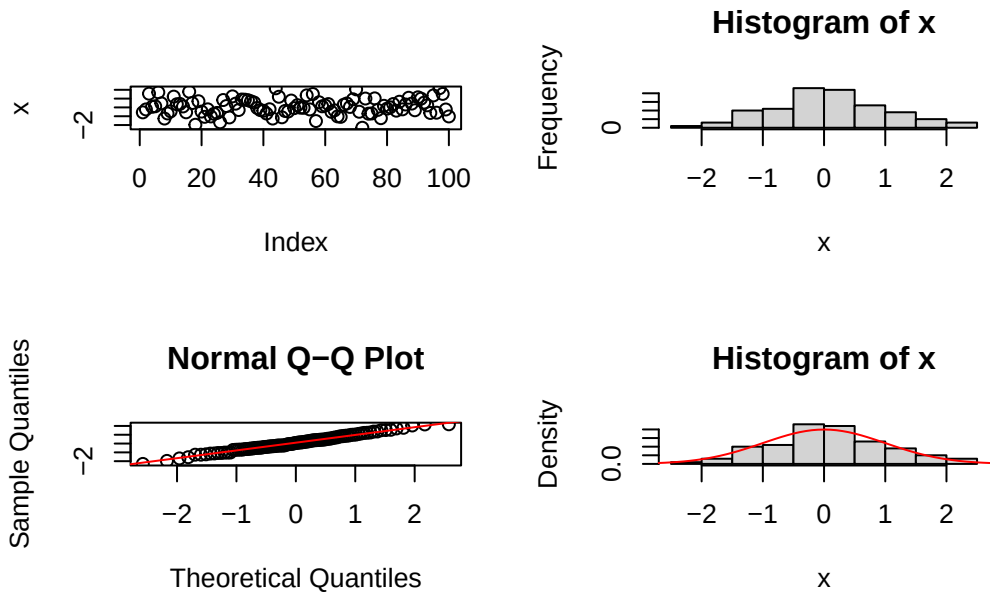
3.3 High-Level Grafiken

Darüber hinaus enthalten **R** und seine Pakete zahlreiche „High-Level"-Grafikfunktionen für bestimmte Zwecke. Um einige davon zu demonstrieren, erzeugen wir zunächst einen Datensatz mit normalverteilten Zufallszahlen (`Mittelwert = 0`, Standardabweichung `sd = 1`), dann plotten wir sie und erstellen ein Histogramm. In diesem Fall teilt die Funktion `par(mfrow = c(2, 2))` den Darstellungsbereich in 2 Zeilen und 2 Spalten, um 4 separate Abbildungen anzuzeigen:

```
par(mfrow = c(2, 2))
x <- rnorm(100)
plot(x)
hist(x)
```

Nun erzeugen wir ein sogenanntes *Normalwahrscheinlichkeitsdiagramm* und ein zweites Histogramm mit relativen Häufigkeiten zusammen mit der glockenförmigen Dichtekurve der Standardnormalverteilung. Das optionale Argument `probability = TRUE` sorgt dafür, dass das Histogramm die gleiche Skalierung wie die Dichtefunktion hat, so dass beide überlagert werden können:

```
qqnorm(x)
qqline(x, col = "red")
hist(x, probability = TRUE)
xx <- seq(-3, 3, 0.1)
lines(xx, dnorm(xx, 0, 1), col = "red")
```



Hier bietet sich eine gute Gelegenheit, verschiedene Funktionen der zusammenfassenden Statistik auszuprobieren: z.B. `mean(x)`, `var(x)`, `sd(x)`, `range(x)`, `summary(x)`, `min(x)`, `max(x)`, …

Alternativ können wir prüfen, ob die generierten Zufallszahlen `x` annähernd normalverteilt sind, indem wir den Shapiro-Wilks-W-Test verwenden:

```
x <- rnorm(100)
shapiro.test(x)
```

Ein p-Wert größer als 0.05 sagt uns, dass der Test keine Einwände gegen die Normalverteilung der Daten hat. Die konkreten Ergebnisse können abweichen, da `x` Zufallszahlen enthält, so dass es sinnvoll ist, den Test mehrmals zu wiederholen. Es kann auch sinnvoll sein, die mit `rnorm` erzeugten normalverteilten Zufallszahlen mit gleichverteilten Zufallszahlen zu vergleichen, die mit `runif` erzeugt wurden:

```
par(mfrow=c(2,2))
y <- runif(100)
plot(y)
hist(y)
qqnorm(y)
qqline(y, col="red")
mean(y)
var(y)
min(y)
max(y)
hist(y, probability=TRUE)
yy <- seq(min(y), max(y), length = 50)
```

```
lines(yy, dnorm(yy, mean(y), sd(y)), col = "red")
shapiro.test(y)
```

Zum Schluss vergleichen wir die Muster beider Datensätze mit Box-and-Whisker-Diagrammen:

```
par(mfrow=c(1, 1))
boxplot(x, y)
```

3.4 Übungen

Wiederhole dieses Beispiel mit neuen Zufallszahlen und variiere die Stichprobengröße (`n`), den Mittelwert (`mean`) und die Standardabweichung (`sd`) für Zufallszahlen, die mit `rnorm` erzeugt wurden, und verwende verschiedene `min` und `max` für `runif`. Konsultiere die Hilfeseiten, um eine Erklärung der Funktionen und ihrer Argumente zu erhalten, und erstelle Boxplots mit verschiedenen Datensätzen.

4 Grundlegende Datenstrukturen von R

Neben Vektoren enthält **R** einige weitere Objektklassen zur Speicherung von Daten, z.B. `matrix`, `array`, `list` oder `data.frame`. Darüber hinaus enthalten sowohl „base **R**“ als auch Zusatzpakete viele weitere Klassen. Es ist auch möglich, eigene benutzerdefinierte Klassen zu definieren.

Unter bestimmten Umständen können einige der Datentypen ineinander konvertiert werden, z.B. durch die Verwendung von Funktionen wie `as.matrix`, `as.vector`, `as.data.frame` und so weiter.

4.1 Numerische und Zeichen-Vektoren

Alle Datenobjekte haben die beiden eingebauten Attribute `mode` (Datentyp) und `length` (Anzahl der Daten im Objekt).

Modi können „numerisch“ (numeric) für Berechnungen oder „Zeichen“ (character) für Textelemente sein.

```
x <- c(1, 3, 4, 5)       # numeric
a <- c("hello", "world") # character
```

Das Folgende ist ebenfalls eine Zeichenvariable, da die Zahlen in Anführungszeichen stehen. Es ist damit nicht möglich, Berechnungen durchzuführen:

```
x <- c("1", "3", "4", "5")  # character
sum(x)

Error in sum(x) : invalid 'type' (character) of argument
```

Hier ist es notwendig, das Zeichen zuerst in eine Zahl umzuwandeln:

```
y <- as.numeric(x)
sum(y)
```

```
[1] 9.040591
```

4.2 Faktoren

Eine besondere Art von Modus ist `factor`. Dies ist, statistisch gesehen, eine nominale Variable die wie eine Zeichenvariable aussieht, z. B. „control“, „treatment A“, „treatment B“ …, aber ihre **Faktorstufen** sind intern als Ganzzahl kodiert.

Hier ein typisches Beispiel mit drei Faktorstufen. In einem ersten Schritt erstellen wir eine Zeichenvariable:

```
text <- rep(c("control", "treatment A", "treatment B"), each = 5)
text
```

```
 [1] "control"     "control"     "control"     "control"     "control"
 [6] "treatment A" "treatment A" "treatment A" "treatment A" "treatment A"
[11] "treatment B" "treatment B" "treatment B" "treatment B" "treatment B"
```

und konvertieren sie dann in einen Faktor:

```
f <- factor(text)
f
```

```
 [1] control     control     control     control     control     treatment A
 [7] treatment A treatment A treatment A treatment A treatment B treatment B
[13] treatment B treatment B treatment B
Levels: control treatment A treatment B
```

Wir sehen, dass die Zeichenvariable mit Anführungszeichen ausgedruckt wird und der Faktor ohne Anführungszeichen, aber mit einer zusätzlichen Information über die Faktorstufen. Der Grund dafür ist, dass der Faktor intern als Ganzzahlwerte mit zugewiesenen Stufen als Übersetzungstabelle kodiert ist:

```
levels(f)
```

```
[1] "control"     "treatment A" "treatment B"
```

Um die Kodierung zu zeigen, können wir den Faktor in eine ganze Zahl umwandeln

```
as.integer(f)
```

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

Die Kodierung wird standardmäßig in alphabetischer Reihenfolge vorgenommen. Sie kann durch ein zusätzliches Argument `levels` geändert werden:

```r
f2 <- factor(text, levels=c("treatment A", "treatment B", "control"))
f2
```

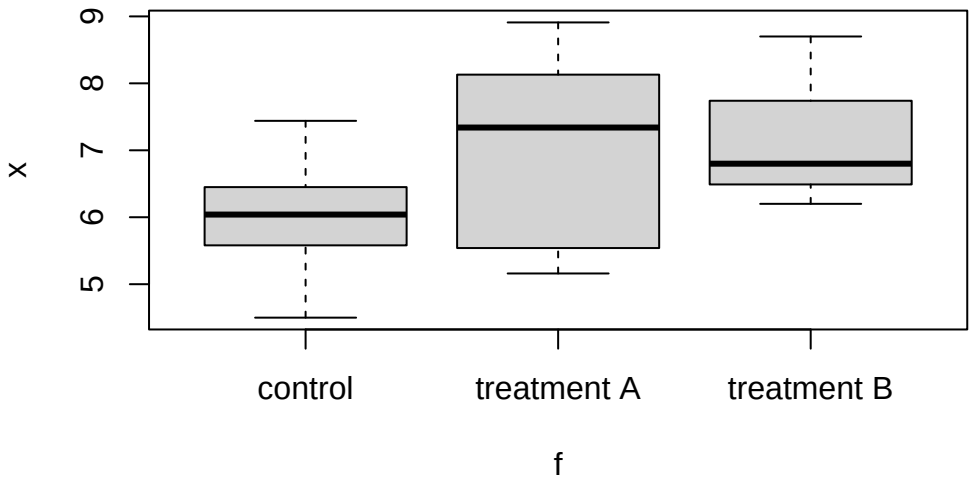
```
 [1] control     control     control     control     control     treatment A
 [7] treatment A treatment A treatment A treatment A treatment B treatment B
[13] treatment B treatment B treatment B
Levels: treatment A treatment B control
```

```r
as.numeric(f2)
```

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

Faktoren sind für statistische Analysen wie ANOVA und statistische Tests nützlich oder auch als Kategorien für die Darstellung:

```r
x <- c(7.44, 6.45, 6.04, 5.58, 4.5, 8.13, 5.54, 7.34, 8.91, 5.16, 8.7, 7.74, 6.8, 6.49, 6.
plot(x ~ f)
```



Wir sehen, dass ein Boxplot erstellt wird und kein x-y-Plot, weil die Erklärungsvariable ein Faktor ist.

Numerische Variablen (insbesondere ordinale) können ebenfalls in Faktoren umgewandelt werden. Dies ist nützlich, wenn wir deutlich machen wollen, dass Zahlen als Namen ohne Reihenfolge zu behandeln sind.

Bei der Rückkonvertierung solcher Faktoren in numerische Variablentypen ist große Vorsicht geboten, da ein Zeichen „123“ mit einem anderen Wert (z. B. 1) und nicht mit 123 kodiert werden kann, siehe die folgende Demonstration einer korrekten und einer falschen Faktorumwandlung:

```
x <- c(2, 4, 6, 5, 8)
f <- as.factor(x)
as.numeric(f)                # falsch !!!
as.numeric(as.character(f)) # richtig
as.numeric(levels(f))[f]    # noch besser
```

Eine derartige Faktorkodierung ist nicht spezifisch für **R** und kommt auch in anderen Statistikpaketen vor. Sie werden dann manchmal als „Dummy-Variablen“ bezeichnet.

4.3 Matrizen und Arrays

Eine Matrix ist eine zweidimensionale Datenstruktur, die für Matrixalgebra verwendet werden kann. Um eine Matrix zu erhalten, können wir zunächst einen eindimensionalen Vektor erstellen und ihn dann als zweidimensionale Matrix mit „nrow“-Zeilen und „ncol“-Spalten umformatieren:

```
x <- 1:20
x

y <- matrix(x, nrow = 5, ncol = 4)
y
```

Wir sehen, dass die Matrix zeilenweise gefüllt aufgefüllt wird. Wir können sie auch wieder in einen Vektor umwandeln:

```
as.vector(y) # reduziert die Matrix auf einen Vektor
```

Ein „Array“ erweitert das Matrixkonzept auf mehr als zwei Dimensionen:

```
x <- array(1:24, dim=c(3, 4, 2))
```

Vektoren, Matrizen und Arrays haben eine wichtige Einschränkung: Sie können nur einen Datentyp (Modus) enthalten, entweder einen numerischen oder einen Zeichentyp. Wenn mindestens ein einzelnes Element vom Typ Zeichen ist, hat die gesamte Matrix den Modus Zeichen und erscheint in Anführungszeichen:

```
x <- c(1, 2, 5, 2, "a")
mode(x)
x
```

4.4 Listen

Der flexibelste Datentyp von **R** ist die Liste. Sie kann beliebige Daten unterschiedlicher Art enthalten. Listen können verschachtelt werden, um eine baumartige Struktur zu bilden:

```
l <- list(x = 1:3, y = c(1,2,3,4), a = "hello", L = list(x = 1, y = 2))
```

Listen sind äußerst leistungsfähig und flexibel und werden später behandelt. Die Ungeduldigen können einen Blick auf das Tutorial von w3schools.com werfen.

4.5 Dataframes

Die typische Datenstruktur für die Datenanalyse in **R** ist der sogenannte `data.frame`. Er kann sowohl Spalten mit numerischen Daten als auch Spalten mit Zeichenvariablen enthalten. Einige Pakete verwenden eine von Dataframes abgeleitete Objektklasse, das so genannte `tibble`.

Ein Dataframe kann von Grund auf direkt im **R**-Code erstellt werden oder aus einer Datei oder dem Internet gelesen werden. Als Beispiel wurden die Studierenden eines Kurses in verschiedenen Jahren nach ihrer Lieblingszahl von eins bis 9 gefragt. Die Ergebnisse können wie folgt in einen Dataframe eingetragen werden:

```
favnum <- data.frame(
  favorite = 1:9,
  obs2019  = c(1, 1, 6, 2, 2,  5,  8, 6, 3),
  obs2020  = c(1, 2, 8, 1, 2,  2, 20, 2, 4),
  obs2021  = c(2, 6, 8, 1, 6,  4, 13, 2, 4),
  obs2022  = c(2, 3, 7, 8, 2, 10, 12, 6, 1)
)
```

5 Dateneingabe

Es gibt verschiedene Methoden zur Eingabe von Daten in **R**. Die wichtigsten sind in einem speziellen Handbuch **R** Datenimport/-export ausführlich erläutert und wir wollen hier nur eine Auswahl zeigen:

1. Direkte Eingabe in den **R**-Code (siehe Beispiel oben)
2. Einlesen aus einer Textdatei
3. Einlesen aus einer Tabellenkalkulationsdatei (LibreOffice oder Excel)
4. Einlesen aus einer Textdatei aus dem Internet
5. Einlesen aus einer Datenbank

Darüber hinaus ist es möglich, Daten aus anderen Statistikpaketen wie SPSS, SAS oder Stata zu importieren (`library(foreign)`), GIS-Shapefiles zu lesen (`library(shapefiles)`), und sogar Sounddateien oder Bilder.

5.1 Direkte Eingabe

Diese Methode haben wir bereits bei der Erstellung von Vektoren mit der `c` (combine)-Funktion verwendet:

```
x <- c(1, 2, 5, 7, 3, 4, 5, 8)
x
```

Auf die gleiche Weise ist es möglich, andere Datentypen wie z.B. Dataframes zu erstellen:

```
dat <- data.frame(f = c("a", "a", "a", "b", "b", "b"),
                  x = c(1,   4,   3,   3,   5,   7)
       )
dat
```

oder Matrizen:

```
A <- matrix(c(1:9), nrow = 3)
A
```

Wie wir sehen, unterscheidet sich eine Matrix nicht wesentlich von einem Vektor, nur dass er in Zeilen und Spalten formatiert ist.

5.2 Daten aus einer Textdatei lesen

R hat sehr flexible Funktionen zum Lesen von Daten aus Textdateien. Nehmen wir zum Beispiel eine Tabelle, die einige Daten aus einem Seegebiet in Nord-Ost-Deutschland enthält (Tabelle 1).

Tabelle 1: Morphometrische und chemische Eigenschaften ausgewählter Seen (S=Stechlinsee, NN=Nehmitzsee Nord, NS=Nehmitzsee Süd, BL=Breiter Luzin, SL = Schmaler Luzin, DA = Dagowsee, HS = Feldberger Haussee; z=mittlere Tiefe (m), t=theoretische Retentionszeit (a), P=Phosphorkonzentration (gL^{-1}), N=Stickstoffkonzentration ($mgL^{-}1$), Chl=Chlorophyllkonzentration (gL^{-1}), PP=jährliche Primärproduktion ($gCm^{-2}a^{-1}$), SD = Secchi-Tiefe (m)). Die Daten sind eine angepasste und vereinfachte „Spielzeugversion“ aus Casper (1985) und Koschel & Scheffler (1985).

Lake	z	t	P	N	Chl	PP	SD
S	23.7	40	2.5	0.20	0.7	95	8.4
NN	5.9	10	2.0	0.20	1.1	140	7.4
NS	7.1	10	2.5	0.10	0.9	145	6.5
BL	25.2	17	50.0	0.10	6.1	210	3.8
SL	7.8	2	30.0	0.10	4.7	200	3.7
DA	5.0	4	100.0	0.50	14.9	250	1.9

Lake	z	t	P	N	Chl	PP	SD
HS	6.3	4	1150.0	0.75	17.5	420	1.6

Die Daten können von https://tpetzoldt.github.io/datasets/data/lakes.csv heruntergeladen werden.

5.2.1 Arbeitsverzeichnis (working directory) festlegen

R muss wissen, wo die Daten auf deinem Computer zu finden sind. Eine Möglichkeit ist, den vollständigen Pfad zum Datensatz anzugeben, z.B. wenn er `c:/users/<Benutzername>/documents` lautet, dann

```
lakes <- read.csv("c:/users/julia/documents/lakes.csv")
```

Dies kann umständlich und fehleranfällig sein, so dass die bevorzugte Methode darin besteht, das **Arbeitsverzeichnis** von **R** auf das Datenverzeichnis zu setzen. Dies kann in **RStudio** wie folgt gemacht werden:

1. Suche nach dem Ordner mit den Daten im Fenster „Files“.
2. Wähle „More“
3. Wähle „Set As Working Directory“

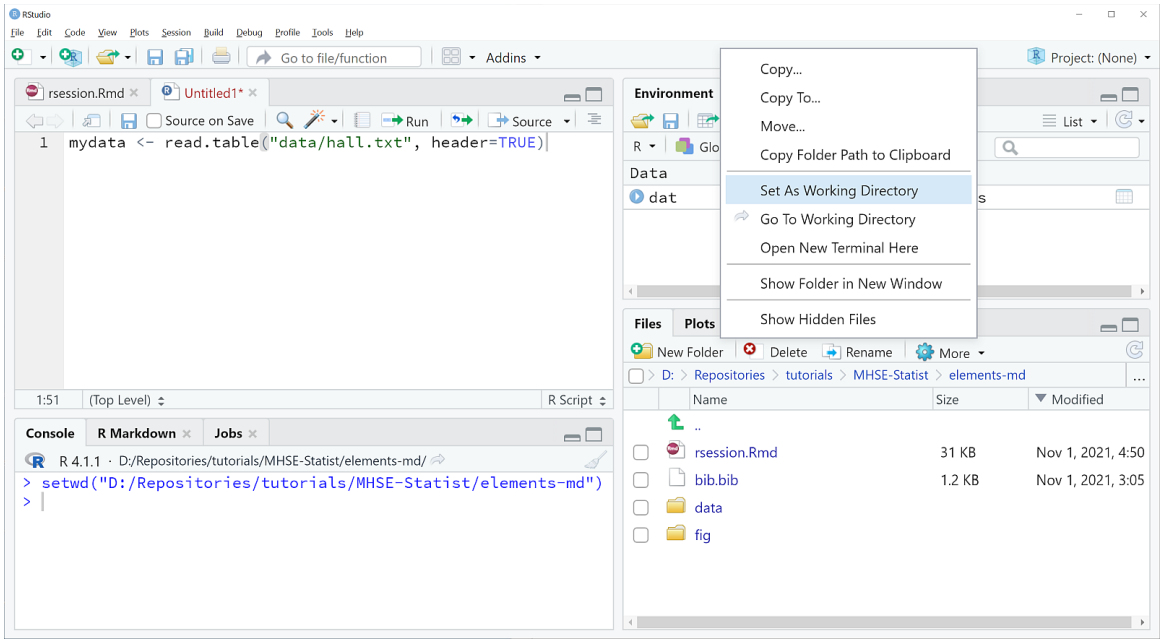


Abbildung 2: Festlegen des Arbeitsverzeichnisses in **RStudio**

Danach können die Daten direkt aus dem **Arbeitsverzeichnis** abgerufen werden:

```
lakes <- read.csv("lakes.csv", header = TRUE)
```

Wir können auch einen Unterordner `data` des Arbeitsverzeichnisses erstellen und die Daten dort ablegen. Dann könnten wir zum Beispiel verwenden:

```
lakes <- read.csv("../data/lakes.csv", header = TRUE)
```

Beachte auch, dass wir immer den gewöhnlichen Schrägstrich „/“ und nicht den Backslash „\“ verwenden, auch unter Windows.

In einigen Ländern, in denen das Komma und nicht der Punkt als Dezimaltrennzeichen und dann z.B. ein Semikolon als Spaltentrennzeichen verwendet wird, können zusätzliche Argumente `dec = ","`, `sep=";"` erforderlich sein. Die Details sind auf der Hilfeseite `read.table` zu finden.

5.3 Daten aus dem Internet lesen

Wenn die Daten auf einem Internet-Server verfügbar sind, können sie direkt von dort gelesen werden:

```
lakes <- read.csv("https://tpetzoldt.github.io/datasets/data/lakes.csv")
```

Da die Daten im Dataframe lakes‘ gespeichert sind, ist es nun möglich, wie gewohnt auf sie zuzugreifen:

```
lakes
summary(lakes)
boxplot(lakes[-1])
```

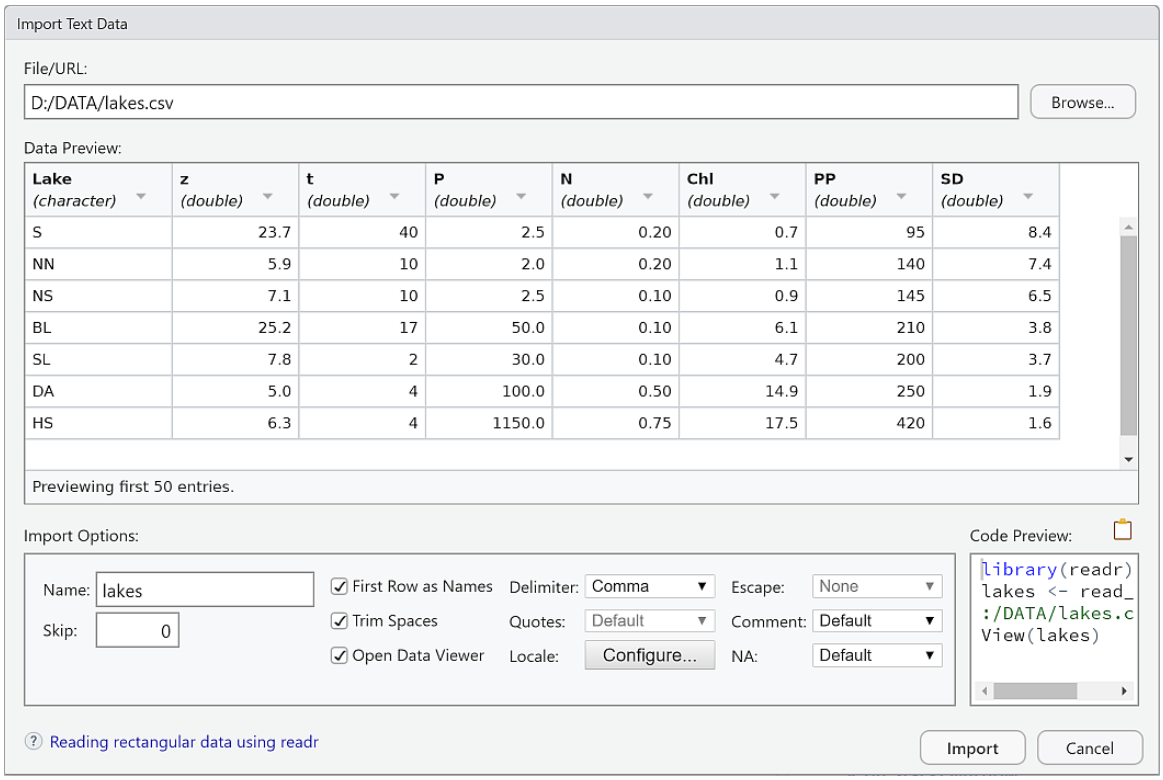
Hier zeigt `summary` einen schnellen Überblick und `boxplot` erstellt ein Boxplot für alle Spalten außer der ersten, die keine Zahlen enthält.

Jetzt können wir den Inhalt der neuen Variable `lakes` untersuchen. Wenn wir **RStudio** verwenden, kann eine Ansicht (“View”) aufgerufen werden, indem man auf `lakes` im Environment-Fenster klickt.

5.4 „Datensatz importieren“ in RStudio

RStudio enthält eine praktische Funktion, die das Importieren von Daten vereinfacht. Im Wesentlichen hilft uns dieser „Import Dataset“-Assistent dabei, interaktiv die richtige `read.table`, `read.csv` oder `read_delim` Funktion zu konstruieren. Es ist möglich, verschiedene Optionen auszuprobieren, bis ein zufriedenstellendes Ergebnis erzielt wird. Aktuelle Versionen von **RStudio** enthalten mehrere verschiedene Möglichkeiten dafür. Hier demonstrieren wir den Assistenten „Import Dataset From Text (readr)“:

1. Wähle aus dem Menü: File – Import DataSet – From CSV.
2. Wähle die gewünschte Datei und gib geeignete Optionen an, wie den Namen der Variablen, der die Daten zugeordnet werden sollen, das Trennzeichen (Komma oder Tabulator) und ob die erste Zeile der Datei Variablennamen enthalten soll.



"Import Dataset From Text (readr)" Assistent von **RStudio**.

Hinweis: Im obigen Beispiel ist der Name des Dataframes identisch mit dem Dateinamen, d.h „lakes". Wenn wir ihn anders benennen wollen (z.B.: `dat`), dürfen wir nicht vergessen, diese Einstellung zu ändern.

Hierbei zeigt die Code-Vorschau die Befehle, die der Assistent erstellt hat. Wenn wir diese Befehle in das Skriptfenster kopieren, können die Daten mehrmals gelesen werden ohne zum Menüsystem zurückkehren zu müssen:

```
library(readr)
lakes <- read_csv("D:/DATA/lakes.csv")
View(lakes)
```

6 Arbeiten mit Dataframes

6.1 Den Inhalt des Dataframes anzeigen

Am einfachsten ist es, den Namen des Dataframes in die **R**-Konsole einzugeben, z.B.:

```
lakes
```

oder man klickt auf den Namen des Dataframes im „Environment“-Explorer von **RStudio**. Dies führt dann `View(lakes)` aus und die Daten werden angezeigt.

Bei großen Tabellen ist es manchmal unbersichtlich, den gesamten Inhalt mit `View` anzuzeigen, so dass es besser sein kann, die Funktion `str` (Struktur) zu verwenden, die einen kompakten Überblick über Typ, Größe und Inhalt einer Variablen gibt:

```
str(mydata)
```

Die Funktion `str` ist universell und auch für komplizierte Objekttypen wie Listen geeignet. Natürlich gibt es noch viele weitere Möglichkeiten, den Inhalt einer Variablen zu untersuchen:

```
names(lakes)
mode(lakes)
length(lakes)
```

und manchmal sogar:

```
plot(lakes)
```

6.2 Zugriff auf einzelne Spalten mit $

Auf einzelne Spalten eines Dataframes kann durch die Indizes (mit `[]`), ähnlich wie bei einem Vektor oder einer Matrix oder mit dem Operator `$` und dem Spaltennamen) zugegriffen werden:

```
mean(lakes[,2])
mean(lakes$z)
mean(lakes[,"z"])
mean(lakes[["z"]])
plot(lakes$z, lakes$t)
```

wobei `z` die mittlere Tiefe der Seen und `t` die sogenannte mittlere Verweilzeit ist.

Wir sollten auch den subtilen Unterschied zwischen der Ausgabe der `[]` und der `[[]]`-Version beachten. Der Unterschied ist folgender: einfache Klammern geben einen Dataframe mit einer Spalte zurück, aber doppelte eckige Klammern ergeben den Inhalt der Spalte als Vektor ohne den Spaltennamen.

Warnung: In einigen älteren Büchern wird der `$`-Stil manchmal mit den Funktionen `attach` und `detach` abgekürzt. Von diesem „prähistorischen Relikt“ wird dringend abgeraten, da

es zu Dateninkonsistenz und seltsamen Fehlern führen kann. Wenn man es irgendwo findet, wo es noch benutzt wird, dann ist es eine gute Idee, `detach` wiederholt zu benutzen, bis eine Fehlermeldung uns bestätigt, dass es nichts mehr gibt, was abgetrennt werden kann. Schließlich: Verwende niemals `attach/detach` in einem Paket.

Stattdessen ist es viel besser, eine andere Funktion `width` zu verwenden, die den Dataframe nur vorübergehend öffnet:

```
with(lakes, plot(z, t))
```

6.3 Teilmengen und logische Indizes

Im Folgenden verwenden wir einen weiteren Datensatz, den wir direkt aus dem Internet eingelesen haben:

```
fruits <- read.csv("https://tpetzoldt.github.io/datasets/data/clementines.csv")
```

Er enthält das Gewicht, die Breite und die Höhe von zwei Marken von Clementinenfrüchten. Nach dem Laden der Daten können wir sie mit `View(fruits)` oder dem Environment Explorer von **Rstudio** betrachten.

Eine sehr leistungsfähige Funktion von **R** ist die Verwendung von logischen Vektoren als Indizes, mit ähnlichen Ergebnissen wie bei Datenbankabfragen. Als Beispiel können wir das Gewicht aller Früchte der Marke „A“ anzeigen mit

```
fruits[fruits$brand == "A", c("brand", "weight")]
```

```
   brand weight
6      A     81
7      A    113
8      A     94
9      A     86
10     A    108
11     A     91
12     A     94
13     A     86
14     A     91
15     A     88
```

Hier gibt der erste Index in den eckigen Klammern die gewünschte Teilmenge der Zeilen an und das zweite Argument die Spalten. Wenn wir alle Spalten sehen wollen, lassen wir das Argument nach dem Komma leer:

```
fruits[fruits$brand == "A", ]
```

```
   id no brand weight width height
6   6  7     A     81    53     54
7   7  8     A    113    61     60
8   8  9     A     94    62     49
9   9  3     A     86    58     53
10 10  6     A    108    64     50
11 11  1     A     91    59     51
12 12 10     A     94    59     51
13 13  4     A     86    55     55
14 14  2     A     91    61     51
15 15  5     A     88    54     55
```

Ein logischer Vergleich erfordert immer ein doppeltes „==“. Logische Operationen wie `&` (und) und `|` (oder) sind ebenfalls möglich. Beachte, dass „und“ immer Vorrang vor „oder“ hat, außer dies wird durch Klammern geändert.

Eine Teilmenge eines Dataframes kann auch mit der Funktion `subset` extrahiert werden:

```
brand_B <- subset(fruits, brand == "B")
brand_B
```

Wie im Beispiel zuvor, erlaubt das Bedingungsargument auch logische Ausdrücke mit `&` (und) und `|` (oder).

Wir können auch auf einzelne Elemente in Matrix-ähnlicher Weise zugreifen, z.B. auf das Element aus der 2. Zeile und der 4. Spalte mit:

```
fruits[2, 4]
```

die komplette 5. Zeile mit:

```
fruits[5, ]
```

und die Zeilen 5:10 der 4. Spalte (`Gewicht`) mit:

```
fruits[5:10, 4]
```

Weitere Methoden für die Arbeit mit Matrizen, Dataframes und Listen sind in **R**-Lehrbüchern oder in der offiziellen **R**-Dokumentation zu finden.

7 Ausblick

7.1 Pipelines und Zusammenfassungen

Die letzten Beispiele sollen zeigen, wie mächtig eine einzelne Zeile in **R** sein kann. Die Art und Weise, wie Datensätze analysiert werden können, hat sich im Laufe der Geschichte von **R** weiterentwickelt, so gibt es z.B. eine Funktion `aggregate`, um Statistiken (z.B. Mittelwerte) in Abhängigkeit von bestimmten Kriterien zu berechnen.

Dies funktioniert gut und wird immer noch verwendet, aber die modernen Methoden aus dem sogenannten „tidyverse“ sind kompakter und einfacher zu verstehen. Hier wollen wir zunächst ein modernes Konzept vorstellen, das „Pipelining“ genannt wird. Die Idee ist, dass das Ergebnis einer Funktion direkt an eine andere Funktion weitergeleitet wird, also anstatt zu schreiben:

```
brand_B   <- subset(fruits, brand == "B")
columns_B <- brand_B[c("weight", "width", "height")]
means_B   <- colMeans(columns_B)
means_B
```

können wir direkt schreiben:

```
library("dplyr")
fruits |> filter(brand == "B") |> select(weight, width, height) |> colMeans()
```

Hier laden wir zunächst ein Zusatzpaket *dplyr*, das viele hilfreiche Funktionen für die Datenverwaltung enthält, z.B. `filter`, das Zeilen auswählt und `select`, das Spalten auswählt. Der Pipeline-Operator `|>` leitet dann die Datenmenge `fruits` zur `filter`- Funktion und anschließend zur nächsten. Der „native Pipeline-Operator“ `|>` wurde mit **R** 4.1 eingeführt. Alternativ können wir auch den `%>%`-Pipelineoperator verwenden, der vom **dplyr**-Paket geladen wird. Die Funktionsweise ist in diesem Fall identisch.

Zwei weitere äußerst nützliche **dplyr**-Funktionen sind `group_by` und `summary`, die es ermöglichen, beliebige zusammenfassende Statistiken in Abhängigkeit von gruppierenden Variablen zu berechnen. Wenn wir die „Marke“ (“brand”) zur Gruppierung verwenden, können wir alle Gruppen gleichzeitig zusammenfassen:

```
fruits |>
  group_by(brand) |>
  summarise(mean(weight), mean(width), mean(height))
```

Die obige Zeile `summarise` kann immer noch verbessert werden, und es gibt viele andere verblüffende Möglichkeiten, daher werden wir später darauf zurückkommen.

7.2 Ausgabe von Ergebnissen

Die einfachste Methode, die Ergebnisse von **R** zu speichern, besteht darin, sie direkt von der **R**-Konsole über die Zwischenablage in ein beliebiges anderes Programm (z. B. LibreOffice, Microsoft Word oder Powerpoint) zu kopieren. Dies ist bequem, kann aber nicht automatisiert werden. Daher ist es besser, einen Programmieransatz zu verwenden.

Nehmen wir das obige Beispiel und speichern wir die Ergebnisse in einen neue Dataframe `results`:

```
results <-
  fruits |>
  group_by(brand) |>
  summarise(mean(weight), mean(width), mean(height))
```

Dataframes können als Textdateien mit `write.table`, `write.csv` oder `write_csv` gespeichert werden. Hier verwenden wir `write_csv` (mit Unterstrich, nicht Punkt) aus dem Paket **readr**:

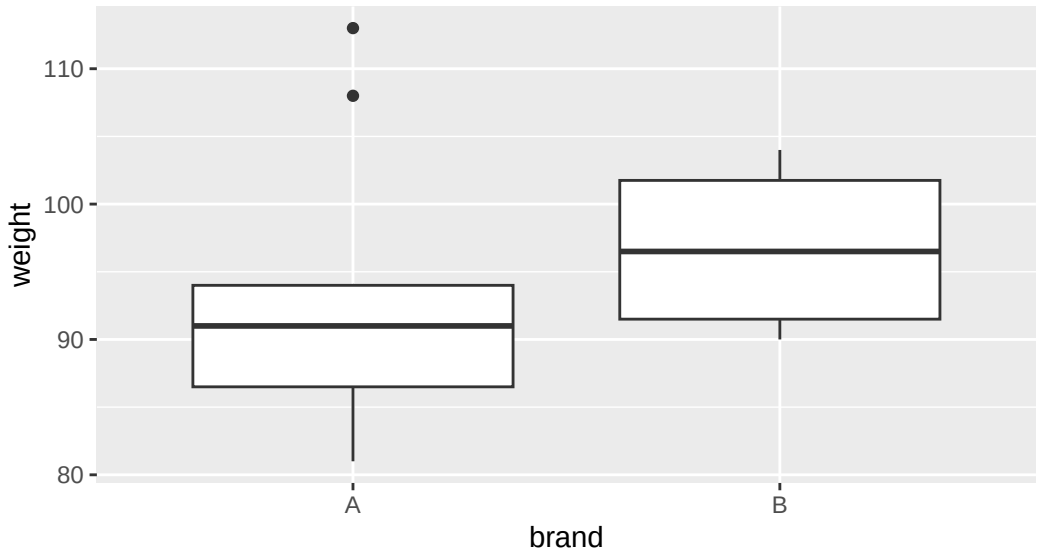
```
library("readr")
write_csv(results, file="output-data.csv")
```

Zusätzlich zu diesen Grundfunktionen verfügt **R** über eine Fülle von Möglichkeiten, Ausgaben und Daten für die spätere Verwendung in Berichten und Präsentationen zu speichern. Alle sind selbstverständlich in der Online-Hilfe dokumentiert, z.B. `print`, `print.table`, `cat` für Textdateien, und `pdf`, `png` für Abbildungen. Das Zusatzpaket **xtable** enthält Funktionen zur Erstellung von **LaTeX**- oder HTML-Tabellen, während die vollständige HTML-Ausgabe von den Paketen `R2HTML` oder `knitr` unterstützt wird.

7.3 Plots mit ggplot2

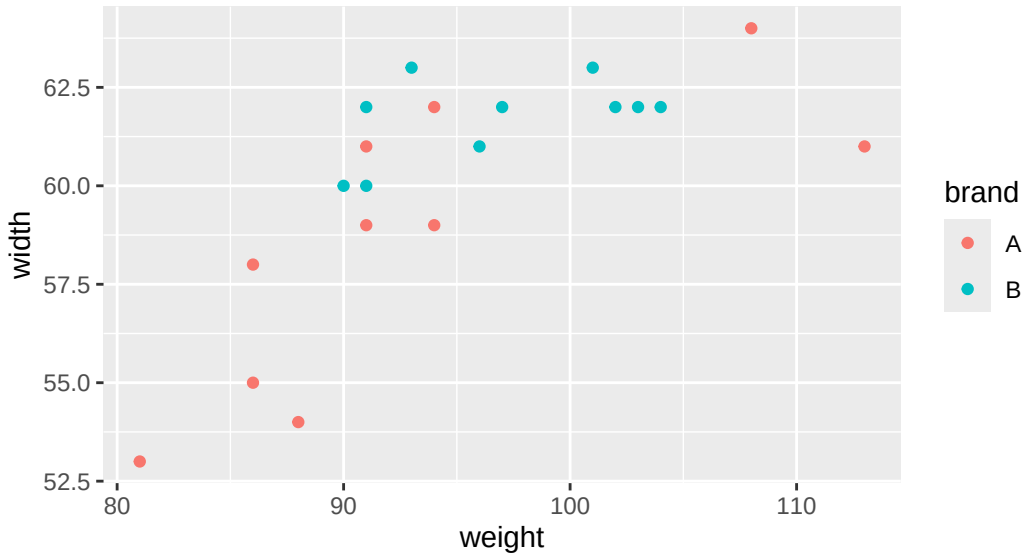
Zusätzlich zu den bisher verwendeten Plot-Funktionen gibt es noch andere Plot-Pakete, zum Beispiel **lattice** oder **ggplot2**. Hier ein paar kleine Beispiele mit dem sehr populären **ggplot2*-Paket:

```
library("ggplot2")
fruits |>
  ggplot(aes(brand, weight)) + geom_boxplot()
```



Oder ein Streudiagramm zum Vergleich von Gewicht ("weight") und Breite ("width") der Früchte

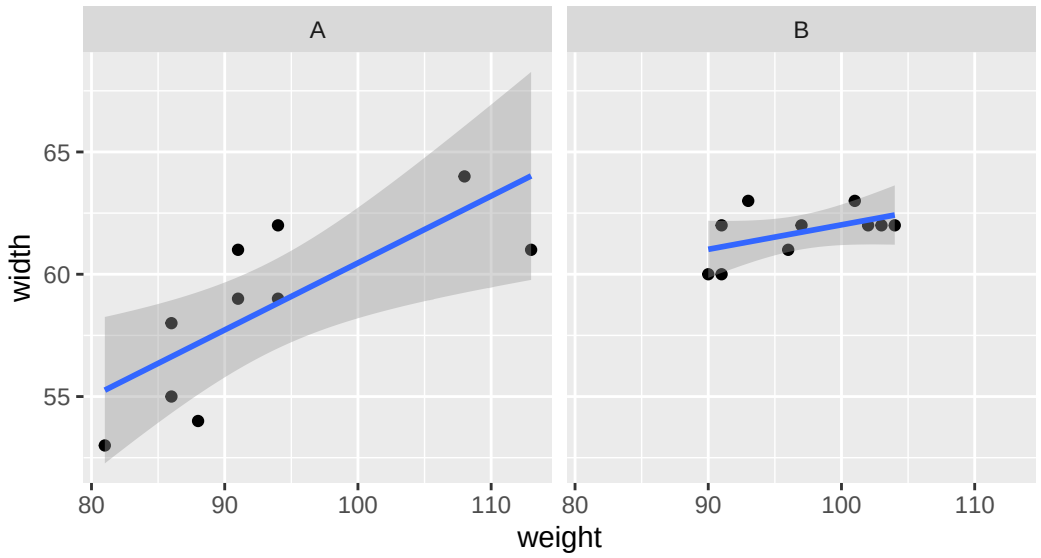
```
fruits |> ggplot(aes(weight, width)) + geom_point(aes(color=brand))
```



wobei die Marke als Farbe angegeben ist. Eine andere Option könnte sein:

```
fruits |> ggplot(aes(weight, width)) +
  geom_point() +
  geom_smooth(method="lm") +
  facet_grid(~brand)
```

```
`geom_smooth()` using formula = 'y ~ x'
```



7.4 Übungen

R enthält viele Datensätze zum Ausprobieren seiner grafischen und statistischen Funktionen, die mit der data-Funktion aktiviert werden können, z.B. `data(iris)` oder `data(cars)`. Eine Beschreibung des Datensatzes findet sich wie üblich in den Hilfedateien, z.B. `?iris`, `?cars`.

Verwende einen dieser Datensätze und probiere aus

1. wie man auf Spalten zugreift, Zeilen auswählt und Teilmengen erstellt
2. wie man Statistiken zusammenfasst und mit **R's** Basisplotfunktionen und optional mit `ggplot` visualisiert.

8 Danksagungen

Dieses Dokument wurde in **RStudio** (RStudio Team, 2021) mit **RMarkdown** und dem Paket **knitr** (Xie, 2015) geschrieben.

Literaturverzeichnis

Casper, S. J. (1985). *Lake Stechlin: A temperate oligotrophic lake*. Dr. W. Junk Publishers.

Koschel, R., & Scheffler, W. (1985). The primary production. In S. J. Casper (Ed.), *Lake Stechlin. A temperate oligotrophic lake* (pp. 287–345). Dr. W. Junk Publishers.

R Core Team. (2021). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. https://www.R-project.org/

RStudio Team. (2021). *RStudio: Integrated development environment for R*. RStudio, PBC. http://www.rstudio.com/

Venables, W. N., Smith, D. M., & Team, R. D. C. (2021). *An introduction to R.*

Xie, Y. (2015). *Dynamic documents with R and knitr* (2nd ed.). Chapman; Hall/CRC. https://yihui.org/knitr/