



Programmierung und Deskriptivstatistik

Dirk Ostwald

Wintersemester 2026/2027

Inhaltsverzeichnis

Vorbemerkungen	3
Zitation	3
Lizenz	3
1 Imperative Programmierung	4
2 Informatische Grundbegriffe	6
2.1 Rechnerarchitektur	7
2.2 Algorithmen und Programme	8
2.3 Programmiersprachen	9
2.4 R	12
2.5 Visual Studio Code	13
3 Arithmetik und Variablen	14
3.1 Arithmetik	14
3.2 Präzedenz	17
3.3 Variablen	21
4 Datenstrukturen	28
4.1 Datenstrukturen in R	29
5 Vektoren	33
5.1 Erzeugung	33
5.2 Charakterisierung	38
5.3 Indizierung	40
5.4 Arithmetik numerischer Vektoren	43
5.5 Attribute	46
6 Matrizen und Arrays	48
6.1 Matrizen	48
6.2 Arrays	54
7 Listen, Dataframes und Tibbles	58
7.1 Listen	58
7.2 Dataframes	64
7.3 Tibbles	69
8 Datenmanagement	73
8.1 Das FAIR-Datenideal	73
8.2 Dateiformate	75
8.3 Verzeichnismanagement	77
8.4 Datenimport und Datenexport	80

9	Kontrollstrukturen	82
9.1	Konditionale Kontrollstrukturen	82
9.2	Iterative Kontrollstrukturen	86
10	Deskriptivstatistiken	89
10.1	Häufigkeitsverteilungen	91
10.2	Histogramme	94
10.3	Empirische Verteilungsfunktionen	98
10.4	Quantile und Boxplots	103
10.5	Maße der zentralen Tendenz	106
10.6	Maße der Variabilität	114
	Referenzen	124

Vorbemerkungen

Dieses Vorlesungsskript begleitet die Veranstaltung *Programmierung und Deskriptivstatistik* am Institut für Psychologie der Otto-von-Guericke-Universität Magdeburg im Wintersemester 2026/2027. Die Kapitel dieses Skripts sind unverändert dem Lehrbuch *Probabilistische Datenwissenschaft für die Psychologie (PDWP)* entnommen.

Zitation

Ostwald, D. (2024) Probabilistische Datenwissenschaft für die Psychologie. [10.5281/zenodo.10730199](https://doi.org/10.5281/zenodo.10730199)

Lizenz

Dieses Werk ist lizenziert unter einer Creative Commons Namensnennung 4.0 International Lizenz (CC BY 4.0).

Verweise auf Kapitel außerhalb dieses Skripts führen zu den entsprechenden Abschnitten des PDWP-Onlinelehrbuchs.

1 Imperative Programmierung

Zur Bedeutung der imperativen Programmierung in der Psychologie

Heutzutage liegen wissenschaftliche Daten in der Regel in digitaler Form vor. Um diese digitalen Daten auszuwerten, werden sie mithilfe eines Computers analysiert. Dazu schreibt man spezielle Computerprogramme, die genau auf die jeweilige Fragestellung zugeschnitten sind. Solche Programme bezeichnet man als Datenanalyseskripte.

Die computergestützte Datenanalyse folgt dabei einer typischen Struktur. Zunächst wird ein digitaler Datensatz eingelesen und bereinigt, um fehlerhafte oder unvollständige Daten zu korrigieren und für die Analyse vorzubereiten. Anschließend werden meist deskriptive Statistiken berechnet und visualisiert, um einen ersten Überblick über die Daten zu gewinnen. Darauf folgt häufig ein Schritt der probabilistischen Modellierung und Inferenz, bei der auf der Wahrscheinlichkeitstheorie basierende Modelle verwendet werden, um Hypothesen zu testen oder Vorhersagen zu treffen. Abschließend werden die Ergebnisse, wiederum mithilfe des Computers, dokumentiert und anschaulich präsentiert.

Zentrales Element der computergestützten Datenanalyse ist das *Datenanalyseskript*, eine Textdatei, die alle Schritte vom Rohdatum bis hin zur finalen Datenvisualisierung dokumentiert und so Datenanalyseprozesse transparent und nachvollziehbar macht. So ermöglicht ein Datenanalyseskript Dritten die Reproduktion wissenschaftlicher Ergebnisse, was einen Grundpfeiler guter wissenschaftlicher Praxis darstellt. Datenanalyseskripte sind damit unverzichtbare Werkzeuge der täglichen wissenschaftlichen Arbeit und essenzielle Bestandteile wissenschaftlicher Publikationen. In den folgenden Abschnitten wollen wir ausgehend von der Diskussion einiger informatischer Grundbegriffe und strukturbildender Elemente der imperativen Programmierung die Entwicklung von Datenanalyseskripten in der Programmierumgebung [R](#) kennenlernen.

Zur Analyse psychologischer Daten kommen verschiedene digitale Werkzeuge zum Einsatz. Besonders verbreitet sind frei verfügbare Programmiersprachen und Plattformen wie [R](#), das häufig in der Datenwissenschaft, Statistik und Psychologie genutzt wird, und [Python](#) mit seinem breiten Einsatzspektrum in der Datenwissenschaft, dem maschinellen Lernen und der künstlichen Intelligenz. Eher noch ein Nischendasein führt die eigentlich datenanalytisch optimierte freie Programmiersprache [Julia](#), die momentan insbesondere in industriellen Anwendungen noch keine weite Verbreitung gefunden hat. Freie Programmiersprachen haben den Vorteil, dass sie häufig von einer breiten Community von Anwendern weiterentwickelt werden und auf diese Weise eine vielseitige Infrastruktur entwickeln können.

Daneben kommen aber auch im Bereich der Psychologie weiterhin kommerzielle Programmierumgebungen, deren Quellcode nicht einsehbar ist, zum Einsatz. Im Bereich der Neurobildgebung sowie in ingenieurtechnischen Anwendungen ist zum Beispiel auch heute noch die kommerzielle Programmierumgebung [MATLAB](#) beliebt. Weiterhin werden einige aus heutiger Sicht eher altmodische Programme verwendet, darunter [SPSS](#), das zur Jahrtausendwende vor allem in den Sozialwissenschaften und in der Psychologie sehr

verbreitet war, [JMP](#) mit Einsatzschwerpunkten in Biologie und Psychologie und [STATA](#), das unter anderem in der Medizin und den Wirtschaftswissenschaften populär war.

Die Popularität digitaler Werkzeuge unterliegt einem stetigen Wandel. Eine Möglichkeit, die aktuelle Popularität verschiedener Programmiersprachen und -umgebungen zu messen, ist die Quantifizierung von Google-Suchanfragen zu Programmiersprachentutorials. Einen Überblick über aktuelle Trends bietet der [PYPL Index](#), der auf der Analyse solcher Suchanfragen basiert.

Zusammenfassend lässt sich festhalten, dass die Digitalisierung insbesondere auch die Wissenschaft nachhaltig prägt. Ein effektives Forschungsdatenmanagement stellt dabei eine akute Herausforderung dar, der sich Forschende stellen müssen. Programmierung entwickelt sich zunehmend zu einem zentralen Handwerkszeug wissenschaftlicher Arbeit, und grundlegende Informatikkenntnisse sind in der modernen Arbeitswelt unverzichtbar geworden. Dies gilt nicht zuletzt auch für Psychotherapeut:innen, etwa im Kontext von Online-Interventionen, digital gestützten Behandlungsangeboten oder der Psychotherapieforschung.

2 Informatische Grundbegriffe

Nach allgemeinem Sprachgebrauch (vgl. [Wikipedia](#)) handelt es sich bei der Informatik um die Wissenschaft der systematischen Darstellung, Speicherung, Verarbeitung und Übertragung von Informationen, wobei besonders die automatische Informationsverarbeitung mit Computern betrachtet wird. Die Informatik ist dabei sowohl Grundlagen- und Formalwissenschaft als auch Ingenieurdisziplin.

Zentrale Komponenten der Informatik sind zum einen die *Computer*, zum anderen *Algorithmen und Programme*. *Computer* sind Maschinen, die Daten speichern und einfache Datenoperationen ausführen können. Diese einfachen Operationen führen sie mit extrem hoher Geschwindigkeit aus. Die Universalität von Computern beruht darauf, dass sie sowohl Daten als auch Programme speichern können.

Programme sind in einer Programmiersprache formulierte *Algorithmen*. *Algorithmen* wiederum sind Folgen von *Anweisungen*, die bestimmte *Operationen* vorschreiben. Bei Algorithmen unterscheidet man verschiedene Ebenen: die *Beschreibungsebene*, wie man sie beispielsweise in Form eines Kochrezepts, einer Bauanleitung oder eines Datenanalyseskripts findet, die *Anweisungen* eines Algorithmus, wie “Mehl und Wasser vermengen”, die bildhafte Darstellung des Zusammenschraubens zweier Teile oder die Anweisung $x = c(1, 2, 3)$ in $\mathbf{R}$, und schließlich die *Durchführungsebene*, also der eigentliche Kochvorgang, der Zusammenbau eines Möbelstücks oder das Ausführen des Datenanalyseskripts.

Die Informatik gliedert sich in mehrere Teilgebiete, die auch für die Psychologie von Bedeutung sein können. Die *Theoretische Informatik* beschäftigt sich mit den Grundlagen der Berechnung, wie der Automatentheorie, der Berechenbarkeitstheorie und der Komplexitätstheorie. Die *Angewandte Informatik* befasst sich mit der Entwicklung und Nutzung von Anwendungssoftware, mit Fragen der Mensch-Computer-Interaktion sowie mit den gesellschaftlichen Auswirkungen der Informatik. Die *Technische Informatik* untersucht die Hardwareseite der Informatik, darunter die Mikroprozessortechnik, die Rechnerarchitektur und die Netzwerktechnik. Schließlich behandelt die *Praktische Informatik* die konkrete Umsetzung in Form von Programmierung, der Entwicklung und Analyse von Algorithmen sowie den Aufbau und die Nutzung von Datenbanken. Anwendung in der psychologischen Grundlagenforschung findet vor allem die praktische Informatik. Allerdings sind auch Fragen der Komplexitätstheorie und damit der theoretischen Informatik in der Entwicklung quantitativer psychologischer Theorien von Interesse (vgl. Van Rooij & Wareham (2012)) und die Mensch-Computer-Interaktion bildet eines der Themengebiete der modernen Arbeitspsychologie (vgl. Dahm (2005)).

Darüber hinaus gibt es zahlreiche Spezialgebiete der Informatik, die eng mit der Psychologie verbunden sind. Dazu gehört insbesondere natürlich der Bereich *Maschinelles Lernen* und *Künstliche Intelligenz*, der seine Ursprünge in der quantitativen Kognitionspsychologie hat und es heutzutage ermöglicht, große Datenmengen automatisiert zu analysieren und Vorhersagen zu treffen. Ein weiteres wichtiges Feld ist die *Computervisualistik*, die sich mit

Bilderkennung und Bildsynthese sowie mit der Gestaltung von Virtueller Realität (VR) und Augmented Reality (AR) beschäftigt. Diese Technologien nutzen einerseits Erkenntnisse der Wahrnehmungspsychologie und eröffnen andererseits neue Möglichkeiten für Experimente, Therapien und Trainings in der Psychologie. Die *Computerlinguistik* trägt mit Verfahren der Spracherkennung und Sprachsynthese dazu bei, menschliche Kommunikation zu analysieren, zu modellieren und in Anwendungen wie Chatbots oder Assistenzsystemen nutzbar zu machen. Schließlich spielt für die Psychologie auch die *Bioinformatik* eine große Rolle, indem sie etwa Methoden zur Auswertung bildgebender Verfahren in der Medizin, die auch in der neuropsychologischen Forschung und Diagnostik genutzt werden, weiterentwickelt.

2.1 Rechnerarchitektur

Auch wenn wir die technischen Aspekte der Informatik hier nicht vertiefen wollen, ist es für die imperative Programmierung hilfreich, zumindest ein rudimentäres Wissen um den typischen physischen Aufbau, also die Hardware, eines Computers zu besitzen. Ein Computer besteht dabei aus mehreren wesentlichen Hardwarekomponenten, die zusammenarbeiten, um alle Rechen- und Speicheraufgaben zu bewältigen. Im Zentrum steht die Hauptplatine, auch Motherboard oder Mainboard genannt. Sie ist die zentrale Leiterplatte, auf der wichtige Bauteile wie Prozessor, Speicher und Anschlüsse miteinander verbunden sind.

Das Herzstück eines Computers ist die CPU (Central Processing Unit, auch Mikroprozessor), die das Rechenwerk, das Steuerwerk und Register des Systems vereint. Sie übernimmt alle grundlegenden Berechnungen, steuert den Datenfluss und koordiniert die Abläufe. Zusätzlich verfügt sie über einen kleinen, aber sehr schnellen *Cache*, einen flüchtigen Speicher, der häufig benötigte Daten zwischenspeichert, um die Verarbeitung zu beschleunigen. Typische Beispiele aktueller CPUs sind Intel-Core-i-Prozessoren, AMD-Ryzen-Prozessoren oder Apple-Silicon-M-Prozessoren.

Ergänzt wird die CPU durch den RAM (Random Access Memory), den temporären, flüchtigen Arbeitsspeicher des Systems. Hier werden während der Laufzeit des Computers die aktuell benötigten Programme und Daten abgelegt, um schnellen Zugriff zu ermöglichen. Der RAM ist in seiner Kapazität begrenzt, in vielen modernen Systemen zum Beispiel auf 32 bis 64 GB.

Für die dauerhafte Speicherung von Daten und Programmen dient der Massenspeicher, der stationäre Speicher des Computers. Heute werden dafür in der Regel SSDs (Solid State Drives) verwendet, die wesentlich schneller und robuster als herkömmliche magnetbasierte Festplatten sind. Auch Cloudspeicher wird zunehmend als Ergänzung oder Ersatz für lokale Massenspeicher genutzt.

Eine weitere wichtige Komponente ist die GPU (Graphics Processing Unit), ein leistungsstarker, speziell auf Visualisierungsaufgaben optimierter Prozessor. Die GPU entlastet die CPU bei grafikintensiven Anwendungen wie 3D-Rendering oder Videobearbeitung und spielt zudem eine wachsende Rolle bei rechenintensiven Aufgaben wie dem Training neuronaler Netze, bei denen sie die CPU effektiv unterstützen kann.

Die grundlegende funktionale Architektur moderner Computer geht zurück auf Von Neumann (1945) und wird entsprechend als *Von-Neumann-Architektur* bezeichnet. Das entscheidende Charakteristikum der Von-Neumann-Architektur besteht darin, dass

sowohl die Befehle (Programmanweisungen) als auch die Daten im selben Speicher abgelegt werden, und zwar in Form von Binärzahlen. Die zentrale Verarbeitungseinheit (CPU) liest sowohl die Befehle als auch die Daten aus diesem gemeinsamen Speicher. Dadurch können Programme während der Ausführung dynamisch verändert oder sogar neu erzeugt werden, da sie im Speicher lediglich als Daten vorliegen.

Speziell beschreibt Von Neumann (1945) folgenden Rechneraufbau: Der Rechner setzt sich aus den grundlegenden Funktionseinheiten Steuerwerk, Rechenwerk, Speicher, Eingabewerk und Ausgabewerk zusammen. Programme und Daten werden gemeinsam in den Speicher eingegeben, sodass Daten, Programme sowie Zwischen- und Endergebnisse im gleichen Speicherbereich abgelegt sind. Der Speicher selbst ist in gleich große, nummerierte (adressierte) Zellen unterteilt, deren Inhalt über die jeweilige Adresse gezielt abgerufen oder verändert werden kann. Die Befehle eines Programms liegen typischerweise in aufeinanderfolgenden Speicherzellen, sodass das Steuerwerk den jeweils nächsten Befehl durch einfaches Erhöhen der Befehlsadresse um eins aufrufen kann. Dieses Prinzip der sequentiellen Abarbeitung der Befehle stellt dabei das Grundprinzip der imperativen Programmierung dar und ist somit aufs Engste mit der Von-Neumann-Architektur verknüpft. Durch spezielle Sprungbefehle ist es weiterhin möglich, von dieser festen Reihenfolge abzuweichen und an eine andere Stelle im Programm zu springen. Zu den grundlegenden Befehlen gehören dabei arithmetische Operationen wie Addition und Multiplikation, logische Vergleiche wie das logische UND oder ODER sowie Transportbefehle, mit denen Daten beispielsweise vom Eingabewerk in den Speicher oder vom Speicher ins Rechenwerk übertragen werden. Sämtliche Daten, also sowohl Befehle als auch Adressen, werden dabei binär codiert. Die binäre Encodierung und Dekodierung der Informationen übernimmt dabei eine geeignete Schaltungstechnik im Rechner.

2.2 Algorithmen und Programme

Ein *Realweltproblem* bezeichnet das Problem, das mithilfe eines Computers gelöst werden soll. Ein typisches Beispiel hierfür ist die Auswertung von Fragebogendaten einer psychologischen Studie. Zur Bearbeitung eines solchen Problems erfolgt zunächst eine *Problem-spezifikation*, also die genaue sprachliche Fassung des Realweltproblems, wie man sie etwa im Methodenteil einer wissenschaftlichen Publikation findet. Auf dieser Grundlage wird ein *Algorithmus* entworfen, also eine Folge von Anweisungen zur Lösung des Problems, beispielsweise das Einlesen der Daten, die Berechnung von deskriptiven Statistiken und die Durchführung eines statistischen Tests. Schließlich wird der Algorithmus in Form eines *Programms* implementiert, das heißt als in einer Programmiersprache verfasste Textdatei, die von einem Computer ausgeführt werden kann.

Definition 2.1 (Algorithmus). Ein Algorithmus ist eine Folge von Anweisungen, um aus gewissen Eingabedaten bestimmte Ausgabedaten herzuleiten, wobei folgende Bedingungen erfüllt sein müssen:

- **Finitheit.** Die Anweisungsfolge ist in einem endlichen Text vollständig beschrieben.
- **Effektivität.** Jede Anweisung muss tatsächlich ausführbar sein.
- **Terminierung.** Der Algorithmus endet nach endlich vielen Anweisungen.
- **Determiniertheit.** Der Ablauf des Algorithmus ist zu jeder Zeit vorgeschrieben.

Wenn E die Menge der zulässigen Eingabedaten und A die Menge der zulässigen Ausgabedaten bezeichnet, dann ist ein Algorithmus also eine Funktion:

$$f : E \rightarrow A, \quad e \mapsto f(e). \tag{2.1}$$

Umgekehrt heißen Funktionen, die durch einen Algorithmus beschrieben werden können, *berechenbare Funktionen*.

•

2.3 Programmiersprachen

Programmiersprachen legen die Regeln fest, denen ein Programm gehorchen muss. Sie definieren eine *Syntax*, also das Vokabular und den Aufbau eines Programms, sowie die *Semantik*, also die Bedeutung der erlaubten Anweisungen. Im Folgenden wollen wir einige Begriffsklärungen zur Kategorisierung verschiedener Formen von Programmiersprachen angeben. Dabei unterscheidet man *Maschinensprache* und sogenannte *höhere Programmiersprachen*, verschiedene *Generationen von Programmiersprachen*, die hier im Vordergrund stehende *Imperative Programmierung* von der *Deklarativen Programmierung* sowie *kompilierte* von *interpretierten* Programmiersprachen.

Maschinensprache und höhere Programmiersprachen

Maschinensprache besteht aus elementaren Operationsbefehlen wie Speichern, Vergleichen oder Addieren, die als Binärzahlen kodiert werden. Tabelle 2.1 zeigt einige Beispiele solcher Befehle:

Tabelle 2.1 Beispiele für Maschinensprache

Befehl	Binärcode
Addiere Inhalt R1 zu Inhalt R2	1001 0010
Erhöhe Inhalt R um 1	1001 0110
Übertrage Inhalt R1 nach R3	0010 0011

Programme, die in Maschinensprache geschrieben sind, nennt man Maschinenprogramme. De facto führt ein Computer ausschließlich Maschinenprogramme aus, da diese direkt von der Hardware verstanden werden. Für Menschen ist die Programmierung in Maschinensprache jedoch sehr mühselig, weshalb in der Praxis überwiegend höhere Programmiersprachen verwendet werden.

Höhere Programmiersprachen verwenden an die menschliche Sprache angelehnte Wörter und Sätze, um die Programmierung verständlicher und effizienter zu machen. Programme in höheren Programmiersprachen werden anschließend von einem *Compiler* oder einem *Interpreter* in Maschinensprache übersetzt, damit sie vom Computer ausgeführt werden können. Bekannte Beispiele für höhere Programmiersprachen sind FORTRAN, COBOL, C++, Python oder auch **R**.

Generationen von Programmiersprachen

Die *Programmiersprachen der 1. Generation (1GL)* sind die Maschinensprachen. Programme werden dabei direkt in Form von Binärzahlen geschrieben, etwa 10110000

01100001, was in hexadezimaler Darstellung B0 61 entspricht. Solche Maschinensprachen wurden vor allem auf den ersten Röhren- und Relaiscomputern wie dem ENIAC oder frühen IBM-Großrechnern der 1940er- und frühen 1950er-Jahre eingesetzt.

Mit der *2. Generation (2GL)* entstanden ab den 1950er-Jahren die Assemblersprachen als erste Form der symbolischen Programmierung. Hierbei werden menschenlesbare Kürzel verwendet, die jedoch noch prozessorspezifisch sind. Ein Beispiel für eine solche prozessor-spezifische Intel-Anweisung ist `MOV AL, 61H`. Diese Generation von Programmiersprachen wurde vor allem auf Großrechnern wie der IBM 704 oder frühen Mainframes der 1950er- und 1960er-Jahre genutzt, die bereits Transistoren statt Röhren verwendeten.

Ab den 1970er-Jahren setzten sich die *Programmiersprachen der 3. Generation (3GL)* durch. Dazu zählen höhere Programmiersprachen wie FORTRAN, C, C++ und Java. Sie sind wesentlich programmierfreundlicher und vor allem prozessorunabhängig, was die Portabilität von Programmen deutlich verbessert. Solche Sprachen wurden zunächst auf Minicomputern wie DEC PDP-11 und später auf Workstations, PCs und Servern weit verbreitet verwendet.

Schließlich entwickelten sich ab den 1980er-Jahren die *Programmiersprachen der 4. Generation (4GL)*, zu denen Sprachen wie Python, MATLAB und R gehören. Diese zeichnen sich durch die Minimierung von Code-Overhead, hohe Flexibilität, Unterstützung multiparadigmatischer Ansätze und einen hohen Grad an Automatisierung aus. Sie finden vor allem auf modernen PCs, High-Performance-Computing-Clustern (HPC-Clustern) und cloudbasierten Systemen Anwendung, die seit den 1990er-Jahren zunehmend verfügbar wurden.

Imperative und deklarative Programmierung

Bei der *Imperativen Programmierung* (von *imperare*, lat. “befehlen”) wird der Lösungsweg als Folge von Anweisungen beziehungsweise Befehlen vorgegeben. Diese Befehle verarbeiten Daten, die über Variablen adressiert werden. Innerhalb der imperativen Programmierung unterscheidet man mit der *prozeduralen* und der *objektorientierten* imperativen Programmierung zwei verschiedene Ansätze.

In der *prozeduralen imperativen Programmierung* werden Daten und die sie manipulierenden Befehle getrennt behandelt. Das zentrale Strukturkonzept bilden hier Prozeduren oder Funktionen, die wiederverwendbare Einheiten von Anweisungen kapseln. Demgegenüber werden in der *objektorientierten imperativen Programmierung* Daten und die auf sie wirkenden Befehle als Objekte zusammengefasst. Diese Objekte sind zugleich Datenträger und Funktionsanbieter und bilden das zentrale Strukturkonzept dieses Ansatzes.

In der Praxis kommen häufig Mischformen aus beiden Paradigmen zum Einsatz, da sich die Ansätze je nach Problemstellung gut kombinieren lassen. Ein Beispiel für eine solche Mischform ist die neuere Programmiersprache [Julia](#), die keinen klassischen objektorientierten Ansatz, sondern stattdessen einen *Multiple-Dispatch*-Ansatz bietet: Funktionen können je nach Typ der Argumente unterschiedliche Implementierungen haben und so Aspekte der objektorientierten und der prozeduralen Programmierung verbinden.

Bei der *Deklarativen Programmierung* steht nicht der Lösungsweg, sondern die Beschreibung des Problems selbst im Vordergrund. Der Programmierende formuliert, was erreicht werden soll, und nicht im Detail, wie der Computer dieses Ziel schrittweise erreichen soll. Der Lösungsweg wird dabei anhand vorgegebener Regeln und Mechanismen automatisch vom System ermittelt.

Ein typisches Merkmal deklarativer Ansätze ist, dass der Fokus auf der Definition von Beziehungen, Bedingungen oder Zielen liegt anstatt auf einer expliziten Abfolge von Befehlen. Der Programmierende beschreibt also das gewünschte Ergebnis in einer formalen Sprache, und der Computer sucht eigenständig nach einer Lösung, die diesen Bedingungen genügt.

Bekannte Ansätze der deklarativen Programmierung sind zum Beispiel die logische Programmierung, wie sie in der Sprache [Prolog](#) umgesetzt ist, oder die funktionale Programmierung, wie in [Haskell](#). In der logischen Programmierung werden Fakten und Regeln formuliert, aus denen das System per Inferenz Schlüsse zieht. In der funktionalen Programmierung werden Berechnungen durch die Komposition von Funktionen beschrieben, wobei der Zustand der Daten nicht explizit verändert wird. Deklarative Programmierung eignet sich besonders für Probleme, bei denen der Lösungsweg komplex und vielfältig sein kann, aber die gewünschten Eigenschaften des Ergebnisses klar spezifiziert werden können. Typische Anwendungsgebiete sind zum Beispiel Datenbankabfragen (z. B. SQL), Wissensbasen, Constraint-Lösungen und formale Beweise.

Ein modernes Beispiel für den Einsatz deklarativer Programmierung ist das System [Lean](#), ein sogenannter *Theorem Prover*. Lean wird eingesetzt, um mathematische Sätze formal zu beweisen. Hierbei formuliert der Benutzer in einer Mischung aus deklarativen und interaktiven Befehlen die Axiome, Definitionen und zu beweisenden Aussagen. Das System prüft automatisch, ob die angegebenen Schritte gültig sind, und ergänzt nach festgelegten Regeln die fehlenden Teile des Beweises. Damit verbindet Lean deklarative Programmierung mit automatischer Beweisführung und erlaubt es, große mathematische Theorien formal korrekt zu entwickeln. Für eine Einführung in Lean im Rahmen mathematischer Beweisführung siehe zum Beispiel [The Mechanics of Proof](#).

Kompilierte und interpretierte Programmiersprachen

Kompilierte Programmiersprachen zeichnen sich dadurch aus, dass der gesamte Quellcode vor der Ausführung in Maschinensprache übersetzt wird. Für diese Übersetzung wird ein spezielles Programm, der sogenannte *Compiler*, verwendet. Der vom Compiler erzeugte Maschinencode wird direkt vom Prozessor ausgeführt, was eine sehr schnelle Ausführung des Programms ermöglicht. Da das ausführbare Programm bereits in Maschinensprache vorliegt, muss es zur Laufzeit nicht mehr übersetzt werden. Allerdings ist es notwendig, bei jeder Änderung des Quellcodes das Programm erneut zu kompilieren. Typische Beispiele für kompilierte Programmiersprachen sind C, C++ und Rust.

Im Gegensatz dazu wird bei *interpretierten Programmiersprachen* der Quellcode während der Ausführung in eine maschinennahe Sprache übersetzt. Dies übernimmt ein spezielles Programm, der sogenannte *Interpreter*. Da die Übersetzung und Ausführung gleichzeitig stattfinden, ist die Ausführung im Vergleich zu kompilierten Programmen in der Regel langsamer. Ein Vorteil besteht jedoch darin, dass nach Änderungen kein separater Kompilierungsschritt erforderlich ist. Das Programm kann sofort ausgeführt werden. Typische Beispiele für interpretierte Programmiersprachen sind Python und **R**.

Im Sinne der hier eingeführten Begriffsbildungen ist **R** eine interpretierte multiparadigmatische Programmiersprache der 4. Generation, die objektorientierte Konzepte unterstützt, aber häufig prozedural und funktional genutzt wird und auf die statistische Datenanalyse zugeschnitten ist.

2.4 R

R ist eine Programmiersprache und ein Softwarepaket, das ursprünglich von Ross Ihaka (Ihaka & Gentleman (1996)) entwickelt wurde. Es handelt sich um einen freien Dialekt der proprietären Software S, die von Becker et al. (Becker et al. (1988)) eingeführt wurde. Heute wird **R** vom [R Core Team](#) und der [R Foundation](#) weiterentwickelt und gepflegt. **R** verfügt über eine große und aktive Community, die bislang mehr als 20.000 **R**-Pakete beigetragen hat, die das System um zahlreiche Erweiterungen ergänzen. Dabei entwickelt sich der Kern von **R** bewusst evolutionär und konservativ weiter, während die Vielzahl an [R-Paketen](#) eine konsistente und progressive Weiterentwicklung ermöglicht. **R** lässt sich einfach von [CRAN](#) herunterladen und installieren.

Mit **R** kann man Datensätze laden, manipulieren und in verschiedenen Formaten speichern. Die Sprache ermöglicht es, eine Vielzahl von Berechnungen auf unterschiedlichen Datenstrukturen durchzuführen. Darüber hinaus bietet **R** ein breites Spektrum statistischer Analysemethoden, die sich direkt auf die Daten anwenden lassen. Typisch ist auch das Schreiben von reproduzierbaren Datenanalyseskripten, mit denen sich Analysen automatisieren und Ergebnisse nachvollziehbar dokumentieren lassen. Schließlich eignet sich **R** hervorragend zur Erstellung von Abbildungen und Visualisierungen, um Daten anschaulich zu präsentieren.

Allerdings lassen sich mit **R** einige Dinge, die in der psychologischen Datenwissenschaft von Bedeutung sind, nur eingeschränkt umsetzen. So fehlt **R** per se eine eigene moderne, ansprechende Entwicklungsumgebung. Hier schafft die Einbettung von **R** in VS Code Abhilfe. Im Bereich des Scientific Computing stößt **R** an seine Grenzen. Hier bieten Sprachen wie Python, MATLAB oder Julia oft leistungsfähigere und flexiblere Werkzeuge. Schließlich eignet sich **R** kaum zum Programmieren psychologischer Experimente, wofür ebenfalls eher Python oder MATLAB zum Einsatz kommen.

Um Hilfe zu **R** zu bekommen, gibt es viele Möglichkeiten. Ganz klassisch hilft oft schon eine schnelle Suche bei [Google](#), eine Anfrage bei [ChatGPT](#) oder eine Interaktion mit [Copilot](#) in VS Code. Während der Programmierung und wenn man den Namen einer Funktion bereits kennt, kann man direkt in **R** Hilfe aufrufen:

```
?mean  
help(mean)
```

Für umfassendere, längere Tutorials und Hintergrundinformationen lohnt sich ein Blick in die sogenannten *Vignetten* der installierten Pakete:

```
browseVignettes()
```

Darüber hinaus gibt es zahlreiche hilfreiche Online-Ressourcen, zum Beispiel die spezialisierte Suchmaschine [rseek](#) oder die vielen Artikel und Anleitungen auf [R-Bloggers](#), wobei die meisten Informationen auf diesen Plattformen sicherlich auch Teil der über Chatbots zugänglichen Datenbanken wie [ChatGPT](#), [Gemini](#), [Claude](#) oder [DeepSeek](#) sind.

2.5 Visual Studio Code

[Visual Studio Code \(VS Code\)](#) ist ein kostenloser Quelltext-Editor von Microsoft, der seit 2015 für Windows, macOS und Linux verfügbar ist. Grob gesagt ist VS Code so etwas wie die (vielleicht eher bekannte) RStudio-Umgebung, nur eben für alle Programmiersprachen. VS Code lässt sich über Erweiterungen flexibel erweitern und als integrierte Entwicklungsumgebung (IDE) für nahezu jede Sprache nutzen, zum Beispiel für **R**, Python, Julia, Shell, Quarto und viele mehr. Seit 2018 gilt VS Code als der beliebteste Editor überhaupt, wie jährliche Umfragen zeigen. Bemerkenswerterweise ist damit ausgerechnet ein Microsoft-Produkt der meistgenutzte Editor auch in der Linux-Welt. VS Code ist sehr konfigurierbar, stark von der Community geprägt und ermöglicht über Microsofts GitHub die Synchronisation der Einstellungen und Erweiterungen über mehrere Endgeräte hinweg. Eine ausführliche Dokumentation mit Anleitungen zur Nutzung findet man unter www.code.visualstudio.com/docs. Einen Einstieg zum Arbeiten mit **R** in VS Code liefert die entsprechende Dokumentation [R in Visual Studio Code](#).

3 Arithmetik und Variablen

3.1 Arithmetik

Öffnet man ein **R**-Terminal, so kann man dieses zunächst einmal als Taschenrechner nutzen, wie folgendes Beispiel zeigt.

```
1+1 # Addition
```

```
[1] 2
```

Im Terminal erscheint dabei einerseits das Ergebnis 2, andererseits mit [1] ein Hinweis darauf, dass das Ergebnis der erste Eintrag in dem von **R** erzeugten Ergebnisvektor ist. Vektoren werden wir in Kapitel 5 im Detail behandeln. Weitere Beispiele für den Gebrauch eines **R**-Terminals als Taschenrechner sind folgende.

```
2*3 # Multiplikation
```

```
[1] 6
```

```
sqrt(2) # Quadratwurzel
```

```
[1] 1.414214
```

```
exp(0) # Exponentialfunktion
```

```
[1] 1
```

```
log(1) # Logarithmusfunktion
```

```
[1] 0
```

Tabelle 3.1 gibt einen ersten Überblick über die in Base **R** verfügbaren arithmetischen Operatoren. Neben den vertrauten Operationen der Addition, Subtraktion, Multiplikation, Division und Potenzierung bietet Base **R** auch spezielle Operatoren für weitergehende arithmetische Berechnungen. Dazu gehören etwa die *Matrixmultiplikation*, die *ganzzahlige Division* ($5 \% \% 2 = 2$) sowie die *Modulo-Operation*, die den Rest einer ganzzahligen Division liefert ($5 \% 2 = 1$).

Tabelle 3.1 Beispiele arithmetischer Operatoren in Base **R**.

Operation	Operator
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Potenz	^
Matrixmultiplikation	%*%
Ganzzahlige Division	%/%
Modulo-Operation	%%

Neben den arithmetischen Operatoren zur Verknüpfung zweier Zahlen bietet Base **R** natürlich auch die Möglichkeit, mathematische Standardfunktionen auszuwerten. Einen ersten Überblick dazu gibt Tabelle 3.2.

Tabelle 3.2 Beispiele mathematischer Funktionen in Base **R**.

Funktion	Aufruf
Exponentialfunktion	<code>exp(x)</code>
Logarithmusfunktion	<code>log(x)</code>
Betrag	<code>abs(x)</code>
Wurzel	<code>sqrt(x)</code>
Aufrunden	<code>ceiling(x)</code>
Abrunden	<code>floor(x)</code>
Mathematisches Runden	<code>round(x)</code>

Obwohl die in Tabelle 3.1 aufgeführten Operatoren und die in Tabelle 3.2 aufgeführten Funktionen auf den ersten Blick einen etwas anderen Charakter haben, unterscheidet Base **R** formal nicht zwischen Operatoren und Funktionen. Speziell können Operatoren mithilfe der sogenannten *Infixnotation* auch als Funktionen mehrerer Argumente genutzt und verstanden werden. Untenstehender **R**-Code zeigt, wie die arithmetische Verknüpfung $2 + 3$ als Ausführung einer Funktion der Form $+: \mathbb{R}^2 \rightarrow \mathbb{R}$ mit der Bezeichnung “+” verstanden werden kann.

```
+`(2,3) # Infixnotation für 2 + 3
```

[1] 5

Schließlich bietet Base **R** wie jede Programmiersprache auch die Möglichkeit, Ausdrücke auf ihren logischen Gehalt hin auszuwerten. Dabei ist Logik hier im Sinne der aus [Kapitel „Sprache und Logik“ im PDWP-Lehrbuch](#) bekannten Aussagenlogik zu verstehen, in der Aussagen einen von zwei Werten annehmen können, wahr (TRUE) oder falsch (FALSE). Die in Tabelle 3.3 aufgeführten Relationsoperatoren `<`, `<=`, `>` und `>=` werden zumeist auf numerische Werte angewendet. Die Gleichheitsoperatoren `==` und `!=` können auf beliebige Datenstrukturen angewendet werden. Die Operatoren zur Verknüpfung logischer Werte `&` und `|` schließlich entsprechen den aus [Kapitel „Sprache und Logik“ im PDWP-Lehrbuch](#)

bekannten Begriffen der logischen Konjunktion (“und”) und Disjunktion (“nicht-exklusives oder”).

Tabelle 3.3 Beispiele logischer Operatoren in Base **R**.

Logische Verknüpfung	Operator
Gleich	<code>==</code>
Ungleich	<code>!=</code>
Konjunktion	<code>&</code>
Disjunktion	<code> </code>
Kleiner	<code><</code>
Kleiner-gleich	<code><=</code>

Die in Tabelle 3.1, Tabelle 3.2 und Tabelle 3.3 aufgelisteten Operatoren und Funktionen stellen nur einen kleinen Auszug der von Base **R** bereitgestellten Operatoren und Funktionen dar. Einen vollständigen Überblick bietet der Aufruf `names(methods:::BasicFunsList)`, der die Namen aller von Base **R** bereitgestellten Funktionen auflistet. Viele dieser Funktionen werden wir im weiteren Verlauf kennenlernen.

```
names(methods:::BasicFunsList)
```

```
[1] "$"           "$<-"         "["
[4] "[<-"        "[["          "[[<-"
[7] "crossprod"   "tcrossprod" "xtfrm"
[10] "c"           "all"         "any"
[13] "sum"         "prod"        "max"
[16] "min"         "range"       "cummax"
[19] "rep"         "~"           "cos"
[22] "levels<-"   "cumsum"      "asin"
[25] "anyNA"      "<="          "Conj"
[28] "exp"         "is.matrix"   "log10"
[31] "as.environment" "ceiling"     "asinh"
[34] "abs"         "as.raw"      "is.infinite"
[37] "is.array"    "floor"       "=="
[40] "sign"        "cummin"      "|"
[43] "round"       "Re"          "!"
[46] "is.numeric"  "acosh"       "!="
[49] "names<-"    "&"           "atanh"
[52] "sinh"       ">="          "sin"
[55] "*"          "atan"        "+"
[58] "length"     "length<-"   "sinpi"
[61] "-"          "is.nan"      "/"
[64] "sqrt"       "%/%"         "as.numeric"
[67] "seq.int"    "trunc"       "digamma"
[70] "acos"       "<"          "as.logical"
[73] "cosh"       "Mod"         "tanpi"
[76] "%*%"        "dimnames<-" "log2"
[79] ">"          "tanh"        "is.na"
[82] "dim"        "signif"      "gamma"
[85] "is.finite"  "as.integer"  "dim<-"
[88] "as.double"  "lgamma"      "Arg"
[91] "log1p"      "trigamma"    "%%"
[94] "tan"        "cumprod"     "Im"
```

[97]	"expm1"	"as.call"	"log"
[100]	"as.complex"	"cospi"	"as.character"
[103]	"dimnames"	"names"	"("
[106]	": "	"="	"@"
[109]	"{"	"~"	"&&"
[112]	".C"	"baseenv"	"quote"
[115]	".: "	"<-"	"is.name"
[118]	"if"	" "	"attr<-"
[121]	"untracemem"	".cache_class"	"substitute"
[124]	"interactive"	"is.call"	"switch"
[127]	"function"	"is.single"	"is.null"
[130]	"is.language"	"is.pairlist"	".External.graphics"
[133]	"declare"	"globalenv"	"class<-"
[136]	".Primitive"	"is.logical"	"enc2utf8"
[139]	"UseMethod"	".subset"	"proc.time"
[142]	"enc2native"	"repeat"	":::"
[145]	"<<-"	"@<-"	"missing"
[148]	"nargs"	"isS4"	".isMethodsDispatchOn"
[151]	"forceAndCall"	"Exec"	".primTrace"
[154]	"storage.mode<-"	".Call"	"unclass"
[157]	"gc.time"	".subset2"	"environment<-"
[160]	"emptyenv"	"seq_len"	".External2"
[163]	"is.symbol"	"class"	"on.exit"
[166]	"is.raw"	"for"	"is.complex"
[169]	"list"	"invisible"	"is.character"
[172]	"oldClass<-"	"is.environment"	"attributes"
[175]	"break"	"return"	"attr"
[178]	"tracemem"	"next"	".Call.graphics"
[181]	"standardGeneric"	"is.atomic"	"retracemem"
[184]	"expression"	"is.expression"	"call"
[187]	"is.object"	"pos.to.env"	"attributes<-"
[190]	".primUntrace"	"...length"	"Tailcall"
[193]	".External"	"oldClass"	".Internal"
[196]	".Fortran"	"browser"	"is.double"
[199]	".class2"	"while"	"nzchar"
[202]	"is.list"	"lazyLoadDBfetch"	"unCfillPOSIXlt"
[205]	"...elt"	"...names"	"is.integer"
[208]	"is.function"	"is.recursive"	"seq_along"
[211]	"unlist"	"as.vector"	"lengths"

3.2 Präzedenz

Eine wichtige Eigenschaft bei der Benutzung von Operatoren ist ihre durch die jeweilige Programmiersprache festgelegte *Präzedenz*. Dabei handelt es sich um von den Programmiersprachenentwicklern festgelegte Regeln zur Rangfolge von Operatoren. Aus der Mathematik ist man insbesondere die Präzedenzregel "Punktrechnung geht vor Strichrechnung" gewohnt. Diese besagt, dass in Ausdrücken, in denen sowohl Produkte oder Divisionen als auch Summen oder Differenzen vorkommen, die Produkte oder Divisionen als Erstes ausgeführt werden und ihre Ergebnisse dann als Zweites in die Summen- oder Differenzbildung eingehen. So ergibt sich zum Beispiel

$$2 \cdot 5 - 3 = 10 - 3 = 7 \quad (3.1)$$

und nicht etwa

$$2 \cdot 5 - 3 \neq 2 \cdot 2 = 4. \quad (3.2)$$

Wir setzen als bekannt voraus, dass die Präzedenzregeln durch Klammerbildung betont oder überschrieben werden können. So ist beispielsweise der Ausdruck in Gleichung 3.1 äquivalent zu

$$(2 \cdot 5) - 3 = 10 - 3 = 7 \quad (3.3)$$

und die in Gleichung 3.2 beabsichtigte Rechnung kann durch Klammersetzung als

$$2 \cdot (5 - 3) = 2 \cdot 2 = 4 \quad (3.4)$$

richtiggestellt werden. Diese vertrauten Rechenregeln und ihre Überschreibung durch Klammern finden sich in **R** entsprechend implementiert:

```
2*5 - 3
```

```
[1] 7
```

```
2*(5 - 3)
```

```
[1] 4
```

Eine weitere als bekannt vorausgesetzte Präzedenzregel ist, dass Potenzen eine höhere Präzedenz als Produkte oder Divisionen und Summen oder Differenzen haben. Es ergibt sich also beispielsweise

$$2^3 + 3 = (2 \cdot 2 \cdot 2) + 3 = 8 + 3 = 11 \quad (3.5)$$

und nicht etwa

$$2^3 + 3 \neq 2^{3+3} = 2^6 = 64. \quad (3.6)$$

Entsprechend gelten in **R**

```
2^3 + 3
```

```
[1] 11
```

```
2^(3 + 3)
```

```
[1] 64
```

Eine wichtige Besonderheit hinsichtlich der Präzedenzregeln in **R** ist, dass auch *unäre Vorzeichen*, also Vorzeichen, die eine Zahl als eine negative Zahl identifizieren, in den Bereich der Operatorpräzedenz fallen. Wenn man auch geneigt sein könnte, einen Ausdruck der Form -1^2 als $(-1)^2 = 1$ zu interpretieren, so folgt **R** dennoch der Regel, dass in -1^2 zunächst die Potenz und dann das Vorzeichen ausgewertet werden, dass also $-1^2 = -(1^2) = -1$ gilt:

```
-1^2 # -(1^2) = -1
```

```
[1] -1
```

```
(-1)^2 # (-1)^2 = 1
```

```
[1] 1
```

Weiterhin werden in **R** Potenzen von rechts nach links abgearbeitet, Produkte, Divisionen, Summen und Differenzen dagegen von links nach rechts. So gelten beispielsweise

```
2^2^3 # 2^(2^3) = 2^8 = 256
```

```
[1] 256
```

```
(2^2)^3 # (2^2)^3 = 4^3 = 64
```

```
[1] 64
```

aber

```
2+3/4*5 # 2+(3/4)*5 = 2+(0.75*5) = 2+3.75 = 5.75
```

```
[1] 5.75
```

```
2+3/(4*5) # 2+3/(4*5) = 2+3/20 = 2+0.15 = 2.15
```

```
[1] 2.15
```

Generell gilt, dass man in der Programmierung Präzedenzregeln nicht raten sollte oder versuchen sollte, sie logisch herzuleiten. Ebenso wenig sollte man darauf vertrauen, dass die Programmiersprachenentwickler die Präzedenzregeln vermutlich gemäß dem eigenen Kenntnisstand implementiert haben werden. Stattdessen sollte man Berechnungen immer kritisch überprüfen und zur Sicherheit die beabsichtigte Rechnung lieber einmal zu viel als einmal zu wenig mit Klammern betonen.

Neben den hier betrachteten Präzedenzregeln für die Grundrechenarten gibt es in **R** eine ganze Reihe weiterer Regeln zum Umgang mit vielen weiteren Operatoren, von denen wir bisher nur sehr wenige kennengelernt haben. Wenn man sich beim Erlernen einer Programmiersprache mit einem neuen Operator vertraut macht, sollte man also für sich auch unbedingt die Präzedenz dieses Operators klären. Der in Abbildung 3.1 gezeigte und mit dem Befehl `?Syntax` aufzurufende Text der **R**-Dokumentation diskutiert die in **R** geltenden Präzedenzregeln vollumfänglich und sollte bei der Arbeit mit **R** häufig konsultiert werden.

Operator Syntax and Precedence

Description

Outlines R syntax and gives the precedence of operators.

Details

The following unary and binary operators are defined. They are listed in precedence groups, from highest to lowest.

<code>:: :::</code>	access variables in a namespace
<code>\$ @</code>	component / slot extraction
<code>[[[</code>	indexing
<code>^</code>	exponentiation (right to left)
<code>- +</code>	unary minus and plus
<code>:</code>	sequence operator
<code>%any% ></code>	special operators (including <code>%%</code> and <code>%/%</code>)
<code>* /</code>	multiply, divide
<code>+ -</code>	(binary) add, subtract
<code>< > <= >= == !=</code>	ordering and comparison
<code>!</code>	negation
<code>& &&</code>	and
<code> </code>	or
<code>~</code>	as in formulae
<code>-> ->></code>	rightwards assignment
<code><- <<-</code>	assignment (right to left)
<code>=</code>	assignment (right to left)
<code>?</code>	help (unary and binary)

Within an expression operators of equal precedence are evaluated from left to right except where indicated. (Note that `=` is not necessarily an operator.)

Abbildung 3.1 Präzedenzregeln in R.

3.3 Variablen

In der Programmierung sind *Variablen* abstrakte Behälter für Daten, deren Inhalt im Verlauf einer Programmausführung geändert werden kann. Variablen werden im Programmcode mit Namen bezeichnet und besitzen bei Programmausführung eine Adresse im Arbeitsspeicher. Variablen können verschiedenen *Typs* sein. Das heißt, dass Variablen unterschiedliche Arten von Daten, und nur diese, repräsentieren. In traditionellen Programmiersprachen wird der Typ einer Variablen explizit *deklariert*. Dabei wird zu Beginn der Programmausführung festgelegt, welche Art von Daten eine bestimmte Variable repräsentieren kann. Die Deklaration einer Variablen A, die zum Beispiel ganze Zahlen und eben nur diese repräsentieren kann, hat etwa die Form

```
VAR A : INTEGER                                # Integer-Variable A (ganze Zahl)
```

In der Folge kann der Variablen A dann ein Wert vom Typ ganze Zahl zugewiesen werden, beispielsweise in der Form

```
A := 1                                          # Der Variablen A den Wert 1 zuweisen
```

In einer 4GL-Sprache wie **R** wird der Variablentyp üblicherweise direkt durch eine Initialisierungsanweisung festgelegt. So deklariert folgender **R**-Code die Variable **a** gleichzeitig als vom Typ **double** (Dezimalzahl) mit dem Wert 1:

```
a = 1                                          # double-Variable a mit Wert 1
```

In den allermeisten Programmiersprachen ist der Zuweisungsbefehl **=**. **R** ist dahingehend besonders, dass es mit **<-** und **->** zusätzliche Zuweisungsbefehle erlaubt: **<-** weist von rechts nach links zu, **->** von links nach rechts. Die übliche Zuweisung von rechts nach links wird auch durch **=** erreicht. Offiziell empfohlener Zuweisungsbefehl für **R** ist **<-**. Dies ist jedoch sehr idiosynkratisch und wir benutzen weitestgehend **=** wie in jeder anderen Programmiersprache. Ein erstes Beispiel soll den Umgang mit Variablen verdeutlichen.

Beispiel 3.1 (Variablen in einer Textaufgabe). Wir betrachten folgende Aufgabe:

“Lina geht ins Schreibwarengeschäft und kauft vier Hefte und zwei Stifte. Ein Heft koste 1 Euro und ein Stift koste 3 Euro. Wie viele Dinge kauft Lina insgesamt und wie viel Euro muss Lina insgesamt bezahlen?”

Um die Aufgabe zu lösen, definieren wir zunächst die Variablen **anzahl_hefte** und **anzahl_stifte** und weisen ihnen dabei ihre jeweiligen Werte aus der Aufgabe zu.

```
anzahl_hefte = 4  
anzahl_stifte = 2
```

Nach Zuweisung existieren die Variablen mit den ihnen zugewiesenen Werten nun im Arbeitsspeicher, dem sogenannten *Workspace*. Der Befehl **ls()** zeigt die existierenden benutzbaren Variablen im Arbeitsspeicher an.

```
ls()
```

```
[1] "anzahl_hefte"    "anzahl_stifte"  "pandoc_dir"     "quarto_bin_path"
```

Entscheidend ist, dass die Variablennamen jetzt wie Zahlen in Berechnungen genutzt werden können. `print()` gibt dabei Variablenwerte in einem **R**-Terminal aus:

```
anzahl_dinge = anzahl_hefte + anzahl_stifte
print(anzahl_dinge)
```

```
[1] 6
```

Um den Gesamtpreis zu berechnen, definieren wir als Nächstes die Variablen `preis_heft` und `preis_stift` anhand der Aufgabenspezifikation.

```
preis_heft      = 1
preis_stift     = 3                                # Preis pro Stift in Euro
```

Der Gesamtpreis berechnet sich dann wie folgt.

```
gesamtpreis = anzahl_hefte*preis_heft + anzahl_stifte*preis_stift
print(gesamtpreis)
```

```
[1] 10
```

Variablen im Workspace können auch wieder gelöscht werden. `rm()` (remove) erlaubt das Löschen einzelner Variablen.

```
rm(gesamtpreis)
ls()
```

```
[1] "anzahl_dinge"    "anzahl_hefte"   "anzahl_stifte"  "pandoc_dir"
[5] "preis_heft"      "preis_stift"    "quarto_bin_path"
```

`rm(list=ls())` löscht alle Variablen.

```
rm(list = ls())
ls()
```

```
character(0)
```



Variablennamen

Prinzipiell ist man in der Wahl der Namen für Variablen mit kleineren Einschränkungen frei. Zulässige Variablennamen in **R**

- bestehen aus Buchstaben, Zahlen, Punkten (.) und Unterstrichen (_),
- beginnen mit einem Buchstaben oder einem Punkt, dann allerdings nicht gefolgt von einer Zahl,
- dürfen keine in **R** schon belegten Ausdrücke wie `for`, `if`, `NaN`, ... sein.

Hilfestellung zu den Einschränkungen von Variablennamen gibt `?reserved`. Eine Funktion, um aus beliebigen Vorschlägen einen syntaktisch zulässigen Variablennamen zu generieren, ist `make.names()`. Ihre Hilfe beschreibt auch die hier skizzierten Regeln zur Variablenbenennung in weiteren Details. Generell sind sinnvolle Variablen kurz, um im Code Platz zu sparen, und aussagekräftig, um das menschliche Verständnis des Codes zu erhöhen. Meist bestehen Variablen nur aus Kleinbuchstaben und Unterstrichen.

Variablenrepräsentation

Wir wollen uns noch etwas genauer mit dem Verhältnis von Variablennamen (Bezeichnen) und Variableninhalten (Objekten im Arbeitsspeicher) befassen. Die folgenden Begriffe und Zusammenhänge sind insbesondere dann wichtig, wenn Berechnungen an die Grenzen des verfügbaren Arbeitsspeichers gelangen und optimiert werden müssen.

Wir beginnen mit dem Begriff des *Bindings*. Dazu betrachten wir zunächst die folgende Variableninitialisierung:

```
x = 1
```

Intuitiv wird durch dieses Statement eine Variable namens `x` mit dem Wert 1 erzeugt. Auf technischer Ebene geschehen dabei zwei Dinge (vgl. Abbildung 3.2):

1. **R** erzeugt ein Objekt (Vektor mit Wert 1) mit Arbeitsspeicheradresse `lobstr::obj_addr(x)`.
2. **R** verbindet dieses Objekt mit dem Namen `x` (*binding*), der das Objekt im Arbeitsspeicher referenziert.

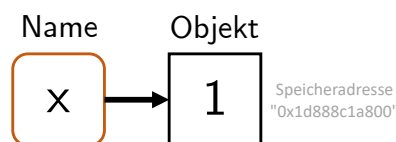


Abbildung 3.2 Binding eines Namens mit einem Objekt im Arbeitsspeicher.

Betrachten wir nun folgendes Statement:

```
y = x
```

Intuitiv wird dabei eine Variable, genannt `y`, mit Wert gleich dem Wert von `x` erzeugt. Auf technischer Ebene wird dabei ein neuer Name `y` erzeugt, der dasselbe Objekt im Arbeitsspeicher referenziert wie `x` (vgl. Abbildung 3.3).

Man kann sich mithilfe von `lobstr::obj_addr(x)` und `lobstr::obj_addr(y)` davon überzeugen, dass die Adressen des Objekts im Arbeitsspeicher identisch sind:

```
print(lobstr::obj_addr(x))
```

```
[1] "0x24020355388"
```

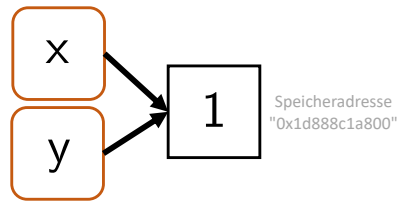


Abbildung 3.3 Binding zweier Namen mit einem Objekt im Arbeitsspeicher.

```
print(lobstr::obj_addr(y))
```

```
[1] "0x24020355388"
```

Das betreffende Objekt wird also bei Zuordnung zu *y* *nicht* kopiert. Dadurch wird Arbeitsspeicher eingespart.

Weiterhin nutzt **R** das sogenannte *Copy-on-modify*-Prinzip. Dieses Prinzip besagt, dass ein Objekt im Arbeitsspeicher kopiert wird und eine neue Adresse erhält, sobald es modifiziert wird. Das ursprüngliche Objekt im Arbeitsspeicher bleibt dabei unverändert. Abbildung 3.4 und folgender **R**-Code verdeutlichen zunächst das *Copy-on-modify*-Prinzip:

```

x = 1          # Neues Objekt; x verweist z. B. auf 0x74b
y = x          # y verweist ebenfalls auf 0x74b
y = 3          # Änderung erzeugt Kopie, z. B. bei 0xcd2
y              # y referenziert jetzt (0xcd2)
  
```

```
[1] 3
```

```
x          # x referenziert weiterhin (0x74b)
```

```
[1] 1
```

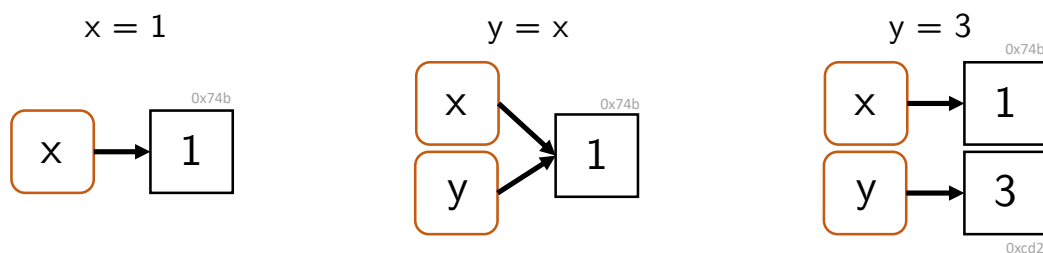


Abbildung 3.4 Copy-on-modify. Erst nach der Änderung des dem Bezeichner *y* zugeordneten Werts wird dieser im Arbeitsspeicher kopiert.

Die Tatsache, dass Objekte bei Veränderung im Arbeitsspeicher kopiert werden, die ursprünglichen Objekte aber im Prinzip unverändert erhalten bleiben, wird auch als die *Immutability* von Objekten in **R** bezeichnet. Allerdings greift die Immutability von Objekten in **R** nicht besonders weit, wie folgendes Beispiel zeigt. Dabei wird zunächst ein Objekt einer 1 im Arbeitsspeicher erzeugt, das mit dem Variablennamen *x* referenziert werden kann. Per *Indexing* (vgl. Kapitel 4) wird das Objekt dann bei einer beabsichtigten Modifikation im Arbeitsspeicher neu adressiert.

```
x = 1                                # Neues Objekt; x verweist auf seine Adresse
print(lobstr::obj_addr(x))           # Speicheradresse des Objekts
```

```
[1] "0x2401d713ee0"
```

```
x[1] = 2                             # Objekt verändert
print(lobstr::obj_addr(x))           # neue Speicheradresse
```

```
[1] "0x2401d76b970"
```

Das Copy-on-modify-Prinzip gilt auch in umgekehrter Reihenfolge. Das heißt, wenn ein Objekt, auf das ein weiterer Variablenname zeigt, verändert wird, so ändert sich das ursprünglich referenzierte Objekt nicht. Folgender **R**-Code mag dies verdeutlichen.

```
x = 1                                # Neues Objekt; x verweist z. B. auf 0x74b
y = x                                # y verweist ebenfalls auf 0x74b
x = 3                                # Neues Objekt; x verweist nun auf 0x2a08
y                                    # y verweist weiter auf 0x74b
```

```
[1] 1
```

Schließlich ist auch ein *Unbinding* von Objekten im Arbeitsspeicher möglich, beispielsweise als Folge folgender **R**-Befehle (vgl. Abbildung 3.5):

```
x = 1                                # x referenziert Objekt (0x74b)
x = "a"                              # x: 0x2a08; 0x74b jetzt ohne Referenz
```

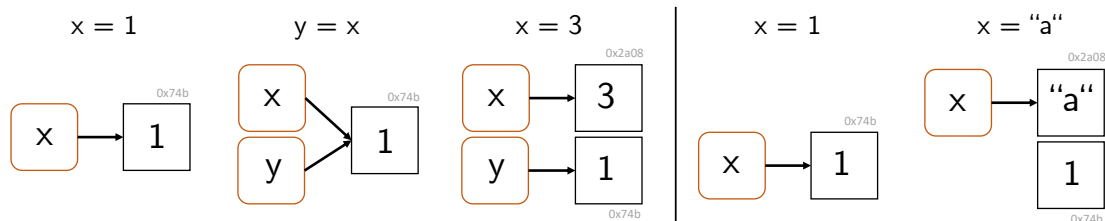


Abbildung 3.5 Unbinding. Es verbleibt belegter Arbeitsspeicher, der durch keinen Bezeichner referenziert wird.

Das Löschen von nicht referenzierten Objekten im Arbeitsspeicher wird als *garbage collection* bezeichnet. **R** löscht nicht referenzierte Objekte im Arbeitsspeicher automatisch, allerdings meist erst dann, wenn der Speicher benötigt wird. Generell funktioniert dies aber gut und es ist in der Regel nicht nötig, aktiv die von **R** bereitgestellte Funktion `gc()` zur Garbage Collection zu benutzen.

Repräsentation fehlender Werte

In der datenanalytischen Programmierung hat man es oft mit fehlenden Werten zu tun, beispielsweise, wenn eine experimentelle Messung bei einer Versuchsperson abgebrochen werden musste. Zur Repräsentation fehlender Werte nutzt **R** `NA` (*not available*). Versucht man mit `NA` zu rechnen, so ergibt dies meist wieder `NA`:

```
3 * NA # Multiplikation eines NA-Wertes ergibt NA
```

```
[1] NA
```

```
NA < 2 # Vergleich mit NA ergibt NA
```

```
[1] NA
```

```
NA^0 # NA hoch 0 ergibt 1
```

```
[1] 1
```

```
NA & FALSE # NA UND FALSE ergibt FALSE
```

```
[1] FALSE
```

Auf NA testet man mit `is.na()`:

```
x = c(NA, 5, NA, 10) # Vektor mit NAs
x == NA # Ungeeigneter NA-Test: 5 == NA ergibt NA
```

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

```
is.na(x) # Logisches Testen auf NA
```

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

Neben NA besitzt **R** mit NaN auch eine Möglichkeit zur Darstellung mathematisch nicht definierter Werte. Im Gegensatz zu NA ist NaN immer vom numerischen Typ. Die Ergebnisse beim Rechnen mit NaN ähneln denen mit NA.

```
3 * NaN # Multiplikation eines NaN-Wertes ergibt NaN
```

```
[1] NaN
```

```
NaN < 2 # Vergleich mit NaN ergibt NA
```

```
[1] NA
```

```
NaN^0 # NaN hoch 0 ergibt 1
```

```
[1] 1
```

```
NaN & FALSE
```

```
# NaN UND FALSE ergibt FALSE
```

```
[1] FALSE
```

Auf NaN testet man mit `is.nan()`:

```
x = c(NaN, 5, NaN, 10)
is.nan(x)
```

```
# Vektor mit NaNs
```

```
# Logisches Testen auf NaN
```

4 Datenstrukturen

In jeder Programmiersprache gibt es abstrakte Behälter, um verschiedene Arten von Daten zu repräsentieren und mit ihnen zu arbeiten. In klassischen 3GL-Programmiersprachen unterscheidet man *fundamentale*, *zusammengesetzte* und *selbstdefinierte* Datenstrukturen.

Fundamentale Datenstrukturen sind in der Programmiersprache vordefiniert. Typische Beispiele sind Behälter für logische Werte (`TRUE`, `FALSE`), ganze Zahlen (`int8`), die z. B. die 256 Werte von -128 bis 127 repräsentieren, Gleitkommazahlen (`single`, `double`) mit unterschiedlicher Genauigkeit oder Schriftzeichen (`a`, `b`), die oft in Anführungszeichen gesetzt werden. Mit diesen grundlegenden Datentypen sind typischerweise passende Operationen verknüpft, wie `AND/OR` für logische Werte, `+`, `-` für Ganzzahlen, `+`, `-`, `*` und `/` für Gleitkommazahlen oder die Konkatenation für Zeichen.

Zusammengesetzte Datenstrukturen sind ebenfalls vordefiniert und dienen dazu, mehrere Variablen gleichen Typs zusammenzufassen. Dazu zählen in **R** zum Beispiel Vektoren, Listen, Arrays, Matrizen und Dataframes. Auch für zusammengesetzte Strukturen existieren spezifische Operationen, wie die Vektorindizierung oder die Matrixmultiplikation.

Schließlich lassen sich in 3GL-Programmiersprachen auch *selbstdefinierte Datenstrukturen* erstellen. Sie ermöglichen die Anpassung der Programmumgebung an spezifische Anwendungsfälle und sind daher besonders praxisrelevant. In der Regel werden sie aus vordefinierten Strukturen zusammengesetzt und können im Rahmen objektorientierter Programmierung auch mit eigenen Operationen assoziiert sein.

Wenn man eine Programmiersprache erlernt, ist ein erster wichtiger Schritt, sich mit ihren Datenstrukturen zumindest rudimentär vertraut zu machen. Feinheiten in der Datenstrukturerepräsentation wird man dann in der täglichen praktischen Arbeit mit der Programmiersprache kennenlernen. Um sich also mit den Datenstrukturen einer Programmiersprache vertraut zu machen, bieten sich folgende Leitfragen an:

Fundamentale Datenstrukturen

- Welche fundamentalen Datenstrukturen bietet die Sprache an?
- Welche Operationen darauf sind bereits definiert?
- Wie lautet die Syntax zur Definition einer Variablen eines fundamentalen Datentyps?
- Wie lautet die Syntax, um vordefinierte Operationen aufzurufen?

Zusammengesetzte Datenstrukturen

- Welche Container und zugehörigen Operationen bietet die Programmiersprache?
- Wie lautet die Syntax zum Umgang mit einem Container?

Selbstdefinierte Datenstrukturen

- Wie erzeugt man selbstdefinierte Datenstrukturen und zugehörige Operationen?
- Wie lautet die Syntax zum Umgang mit einer selbstdefinierten Datenstruktur?

4.1 Datenstrukturen in R

Als 4GL-Sprache geht **R** in seinen Datenstrukturen über die klassischen Datenstrukturen von 3GL-Sprachen hinaus. In **R** ist zunächst einmal alles ein *Objekt*. Egal, ob es sich um einen einzelnen Wert, einen Vektor, eine Funktion oder eine komplexe Datenstruktur handelt – alles wird von **R** als Objekt repräsentiert. Allgemein bezeichnet der Begriff des *Objekts* die Möglichkeit, in einer Sinneinheit sowohl Daten als auch Funktionalität zu repräsentieren. Wir sind hier allerdings primär nur an der Datenrepräsentation interessiert.

Generell unterscheidet man in **R** zwischen *atomaren* Objekten und *rekursiven* Objekten. *Atomare Objekte* bestehen aus Komponenten gleichen Datentyps. Typische Beispiele sind numerische Vektoren, logische Vektoren oder Zeichenketten. *Rekursive Objekte* hingegen können Komponenten unterschiedlichen Typs enthalten. Dazu gehören insbesondere Listen und Dataframes, die es erlauben, heterogene Daten zusammenzufassen.

Jedes **R**-Objekt lässt sich eindeutig beschreiben durch drei zentrale Eigenschaften: seinen *Modus*, seine *Länge* und optional durch weitere *Attribute*. Der *Modus* eines Objekts gibt den grundlegenden Datentyp eines Objekts an und ist eng mit dem *Typ* und der *Klasse* eines Objekts verwandt, wie im Folgenden erläutert. Neben dem Modus besitzt jedes Objekt eine *Länge*, die angibt, wie viele Elemente es enthält. Bei einem Vektor ist die Länge zum Beispiel die Anzahl der enthaltenen Werte, bei einer Liste die Zahl ihrer Komponenten. Darüber hinaus können Objekte in **R** mit zusätzlichen *Attributen* versehen sein. Attribute sind Metadaten, die zusätzliche Informationen über das Objekt bereitstellen, etwa Namen der Elemente (`names`), Dimensionen (`dim`) für Matrizen und Arrays oder Klasseninformationen (`class`), die für das objektorientierte Programmieren in **R** relevant sind.

Modus, Typ und Klasse

Die Klassifizierung von Datenstrukturen in **R** ist etwas komplex, da nebeneinander die Begriffe des *Modus*, des *Typs* und der *Klasse* eines Objekts existieren. In der praktischen Anwendung sind die Feinheiten dieser Klassifizierung meist nicht von Belang, außerdem sind der Modus, Typ und die Klasse eines Objekts auch oft identisch.

Grob gesagt entspricht der *Modus* der grundlegenden Natur eines Objekts und ist insbesondere so gewählt, dass eine hohe Kompatibilität zum Datenstrukturklassifikationssystem von **S** (Becker et al. (1988)) besteht. Beispiele für Modi sind `numeric`, `character`, `logical` oder `list`. Der Modus eines Objektes kann mit der Funktion `mode()` abgerufen werden, wie folgendes Beispiel zeigt.

```
mode(42) # numeric
```

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

```
mode(1L) # numeric
```

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

```
mode("hallo") # character
```

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

```
mode(TRUE) # logical
```

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

```
mode(list(1, 2, 3)) # list
```

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

```
mode(factor(c("a", "b"))) # R-Faktor => numeric
```

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

```
mode(Sys.Date()) # R-Datum => numeric
```

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

```
mode(lm(mpg ~ wt, data=mtcars)) # lineares R-Modell => list
```

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

Etwas näher an der internen Repräsentation eines Objekts im Speicher und damit etwas technischerer Natur ist der *Typ* eines **R**-Objekts. Dies wird insbesondere hinsichtlich Objekten von Modus **numeric** deutlich. Hier gibt es die klassischen 3GL-Typenunterscheidungen **integer** oder **double**. Ein anderes Beispiel ist der Modus *Faktor*. Wie wir an späterer Stelle sehen werden, sehen Faktoren in **R** für den Nutzer aus wie kategorische Daten, oft von Schriftzeichenart. Ihr Modus ist allerdings **numeric** und ihr Typ **integer**. Dies hängt damit zusammen, dass Faktoren in **R** oft dazu genutzt werden, auf der Benutzerebene experimentelle Bedingungen zu spezifizieren, die meist aussagekräftige Namen wie “Treatment” oder “Kontrolle” haben, in der Datenanalyse aber durch mathematische Matrizen mit ganzzahligen Einträgen repräsentiert werden. Der Typ eines Objektes kann mit der Funktion `typeof()` abgerufen werden, wie folgendes Beispiel zeigt. Man beachte die Gemeinsamkeiten und Unterschiede zu den oben aufgerufenen Modi der gleichen Objekte.

```
typeof(42) # double
```

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

```
typeof(1L) # integer
```

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

```
typeof("hallo")           # character
```

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

```
typeof(TRUE)             # logical
```

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

```
typeof(list(1,2,3))       # list
```

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

```
typeof(factor(c("a", "b"))) # R-Faktor => integer
```

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

```
typeof(Sys.Date())        # R-Datum => double
```

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

```
typeof(lm(mpg ~ wt, data=mtcars)) # lineares R-Modell = list
```

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

Die Klasse eines Objektes schließlich spezifiziert die Art eines Objektes aus der Perspektive der objekt-orientierten Programmierung in **R**. Dabei ist es für ein Objekt insbesondere relevant, wie generische Funktionen wie z. B. `print()` oder `summary()` mit dem Objekt umgehen. Im Gegensatz zu Modi und Typen können Programmierende die Klasse von Objekten selbst festlegen und so generische Funktionen entwickeln, die auf viele Objektarten anwendbar sind und trotz ihres gleichen Namens mit unterschiedlichen Objekten unterschiedliche Operationen ausführen. Die Klasse eines Objektes kann mit der Funktion `class()` abgerufen werden, wie folgendes Beispiel zeigt. Man beachte die Gemeinsamkeiten und Unterschiede zu den oben aufgerufenen Modi und Typen der gleichen Objekte.

```
class(42)                 # numeric
```

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

```
class(1L)                 # integer
```

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

```
class("hallo")           # character
```

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

```
class(TRUE) # logical
```

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

```
class(list(1,2,3)) # list
```

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

```
class(factor(c("a", "b"))) # R-Faktor => "factor"
```

```
[1] "factor"
```

```
class(Sys.Date()) # R-Datum => "Date"
```

```
[1] "Date"
```

```
class(lm(mpg ~ wt, data=mtcars)) # lineares R-Modell => "lm"
```

```
[1] "lm"
```

Zusammengefasst lässt sich festhalten: Der Modus beschreibt die allgemeine Art eines Objekts, der Typ beschreibt seine interne, technische Speicherung und die Klasse definiert die Sichtweise von **R** für die funktionale Verarbeitung des Objekts. Für viele Objekte stimmen diese drei Begriffe weitgehend überein, aber bei speziellen Objekten wie Faktoren, Datumswerten oder benutzerdefinierten Objekten unterscheiden sie sich.

5 Vektoren

In diesem Abschnitt wollen wir nun mit den **R**-Vektoren eine erste grundlegende Datenstruktur genauer kennenlernen. Viele Prinzipien, die am Beispiel von Vektoren erläutert werden können, übertragen sich sinngemäß auf komplexere Datenstrukturen in **R**, sodass ein kompetenter Umgang auch mit der hier betrachteten basalen Datenstruktur im Anwendungskontext unverzichtbar ist. Es sei angemerkt, dass sich der Begriff des *Vektors* in diesem Abschnitt streng auf die entsprechende **R**-Datenstruktur, nicht aber auf das gleichnamige mathematische Objekt bezieht. Auch wenn **R**-Vektoren in manchen Eigenschaften ihrem mathematischen Pendant ähneln und bestimmte Operationen zwischen **R**-Vektoren Operationen zwischen den entsprechenden mathematischen Objekten abbilden können, so haben Vektoren in der imperativen Programmierung oft viel mehr und andere Eigenschaften als die entsprechenden mathematischen Objekte.

Allgemein betrachten wir bei der Diskussion einer Datenstruktur immer zunächst, wie sie mithilfe von **R**-Code *erzeugt* wird, durch welche Eigenschaften sie *charakterisiert* ist, wie einzelne Elemente der Datenstruktur *indiziert* und bearbeitet werden können und welche *arithmetischen Operationen* für die Datenstruktur zulässig sind. Schließlich betrachten wir eventuelle weitere Besonderheiten der Datenstruktur. Im Kontext von **R**-Vektoren sind dies beispielsweise die Begriffe der *Datentypangleichung* (*Coercion*), des *Recyclings* oder die Repräsentation fehlender Werte in Datensätzen.

R-Vektoren sind geordnete Folgen von Datenwerten. Die einzelnen Datenwerte eines Vektors heißen *Elemente* des Vektors. Vektoren, deren Elemente alle vom gleichen Datentyp sind, heißen *atomar*. Mit dem Begriff *Vektor* soll hier immer ein atomarer Vektor gemeint sein. Die zentralen Datentypen für Vektoren sind die oben eingeführten *numeric* (*double*, *integer*), *logical* und *character*. Abbildung 5.1 zeigt graphische Darstellungen entsprechender Vektoren.

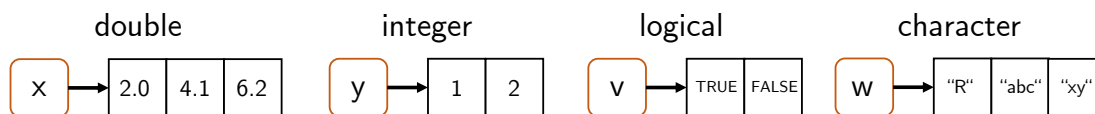


Abbildung 5.1 Vektoren in **R**.

5.1 Erzeugung

Wir betrachten zunächst die Erzeugung einelementiger Vektoren, die auch als *Elementarwerte* bezeichnet werden. Wie alle Datenstrukturen werden auch Elementarwerte durch Zuweisungen der Form

```
Elementarwert = Wert
```

erzeugt. Dabei bestimmt die Spezifikation von **Wert**, welcher Art der erzeugte Elementarwert ist.

Elementarwerte vom Typ **numeric** können durch Zuweisung einer Zahl erzeugt werden. Per Default wird dabei ein Elementarwert vom Typ **double** erzeugt. Die Zahlen können dabei ohne Dezimalstellen, in Dezimalnotation oder in wissenschaftlicher Notation spezifiziert werden. Weitere mögliche Werte zur Erzeugung eines Elementarwerts vom Typ **double** sind **Inf** und **-Inf** für positive und negative Unendlichkeit und **NaN** (Not-a-Number) als Platzhalter für numerische Werte.

```
x = 1                                # Einelementiger Vektor
typeof(x)                           # vom Typ double
```

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

```
y = 2.1e2                            # Einelementiger Vektor
typeof(y)                           # vom Typ double
```

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

```
z = Inf                              # Einelementiger Vektor
typeof(z)                           # vom Typ double
```

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

Numerische Elementarwerte vom Typ **integer** werden wie **double** ohne Dezimalstellen erzeugt, gefolgt von einem **L** für **long integer**.

```
x = 1L                               # Einelementiger Vektor
typeof(x)                           # vom Typ integer
```

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

```
y = 200L                             # Einelementiger Vektor
typeof(y)                           # vom Typ integer
```

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

Elementarwerte vom Typ **logical** können durch Zuweisung der logischen Werte **TRUE** oder **FALSE** bzw. abgekürzt **T** oder **F** erzeugt werden.

```
x = TRUE                             # Einelementiger Vektor
typeof(x)                           # vom Typ logical
```

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

```
y = F                                # Einelementiger Vektor
typeof(y)                            # vom Typ logical
```

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

Elementarwerte vom Typ `character` schließlich werden durch die Zuweisung von Werten in Hochkommata (`'a'`) oder Anführungszeichen (`"a"`) erzeugt.

```
x = "a"                              # Einelementiger Vektor
typeof(x)                            # vom Typ character
```

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

```
y = 'test'                           # Einelementiger Vektor
typeof(y)                            # vom Typ character
```

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

Um nun mehrere Elementarwerte in einem Vektor zusammenzufügen, konkateniert man sie mit der Funktion `c()`. Der Begriff der Konkatenation leitet sich dabei vom Lateinischen *catena* (“die Kette”) her.

```
x = c(1,2,3)                         # numerischer Vektor [1,2,3]
s = c("a", "b", "c")                 # Zeichenvektor ["a", "b", "c"]
l = c(TRUE, FALSE)                   # logischer Vektor [TRUE, FALSE]
```

```
x
```

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

```
s
```

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

```
l
```

```
[1] TRUE FALSE
```

Es können nicht nur Elementarwerte, sondern auch Vektoren mit Elementarwerten oder mit anderen Vektoren konkateniert werden. Mit obigem Vektor `x` erzeugt man beispielsweise folgende Vektoren `y` und `z`:

```
y = c(0,x,4)                         # numerischer Vektor [0,1,2,3,4]
z = c(x,y)                           # numerischer Vektor [1,2,3,0,1,2,3,4]
```

y

[1] 0 1 2 3 4

z

[1] 1 2 3 0 1 2 3 4

Vektoren müssen nicht mit bereits bestehenden Werten erzeugt werden, sondern es können auch “leere” Vektoren von einem gewünschten Datentyp erzeugt werden. **R** stellt dafür die Funktionen `vector()` sowie `double()`, `integer()`, `logical()` und `character()` bereit. Dies ist potenziell hilfreich, um zu Beginn einer längeren Berechnung bereits Arbeitsspeicher des Computers belegt zu haben, sodass während der Programmausführung nicht nach Arbeitsspeicher gesucht werden muss, was unter Umständen die Programmlaufzeit erhöht.

```
# Erzeugen "leerer" Vektoren mit vector()
v = vector("double",3)      # double-Vektor [0,0,0]
w = vector("integer",3)     # integer-Vektor [0,0,0]
l = vector("logical",2)     # logischer Vektor [FALSE, FALSE]
s = vector("character",4)   # Zeichenvektor ["", "", "", ""]

# Alternatives Erzeugen "leerer Vektoren"
v = double(3)               # double-Vektor [0,0,0]
w = integer(3)              # integer-Vektor [0,0,0]
l = logical(2)              # logischer Vektor [FALSE, FALSE]
s = character(4)            # Zeichenvektor ["", "", "", ""]
```

Allerdings sind die so erzeugten “leeren” Vektoren nicht wirklich leer: `v` und `w` bestehen aus Nullen, `l` aus den logischen Werten `FALSE` und `s` aus den Zeichen `""`. Zur Initialisierung von Vektoren als Container und anschließender Befüllung im Rahmen eines Programms sind `vector()`, `double()`, `integer()`, `logical()` und `character()` deshalb nur bedingt geeignet: Übersieht der Code die Befüllung einzelner Einträge, so wird mit den von diesen Funktionen erzeugten Werten gerechnet, was nicht unbedingt beabsichtigt ist. In Kapitel 9 lernen wir eine Methode kennen, um Datenstrukturen zu initialisieren und dabei einigermaßen sicheren Code zu generieren.

Sequenzen

Ein häufiges Hilfsmittel in der Programmierung sind Vektoren, die Folgen von numerischen Werten, z. B. als Repräsentation der Definitionsmengen einer univariaten Funktion, darstellen. **R** stellt zu ihrer Erzeugung den sogenannten *Colonoperator* `:` und die Funktion `seq()` und ihre Varianten zur Verfügung.

Der Colonoperator in der Form `a:b` erzeugt Sequenzen ganzzahliger Schritte zwischen `a` und `b` anhand von

$$(s_i)_{i=0,1,2,\dots} \text{ mit } s_i = a + i \text{ für } i = 0, 1, 2, \dots \text{ und } s_i \leq b. \quad (5.1)$$

Dabei meint “zwischen `a` und `b`” hier also speziell, dass `a` die erste Zahl der generierten Sequenz ist und `b` die maximal mögliche. Folgende Beispiele verdeutlichen dies:

```
x = 0:5          # Sequenz von 0 bis einschließlich 5
y = 1.5:6.1      # Beginn bei 1.5, Ende bei 5.5 < 6.1
```

```
x
```

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

```
y
```

```
[1] 1.5 2.5 3.5 4.5 5.5
```

Die Funktion `seq()` mit der Syntax `seq(from, to, by)`, wobei `from` den Startwert, `to` den maximalen Endwert und `by` die Schrittweite bezeichnen, bietet weitere Möglichkeiten, Sequenzen nicht nur ganzzahliger Schrittweiten zu erzeugen. Die folgenden Beispiele sollen dies verdeutlichen.

```
x_1 = seq(0,5)          # wie 0:5
x_2 = seq(0,1,len = 5)  # 5 Zahlen zwischen 0 (inkl.) und 1 (inkl.)
x_3 = seq(0,2,by = .15)  # Schrittweite 0.15; von 0 bis höchstens 2
x_4 = seq(1,0,by = -.1)  # Schrittweite -0.1; von 1 bis 0
```

```
x_1
```

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

```
x_2
```

```
[1] 0.00 0.25 0.50 0.75 1.00
```

```
x_3
```

```
[1] 0.00 0.15 0.30 0.45 0.60 0.75 0.90 1.05 1.20 1.35 1.50 1.65 1.80 1.95
```

```
x_4
```

```
[1] 1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0
```

Die `seq()`-Variante `seq.int()` kann zur Erzeugung von ganzzahligen Folgen genutzt werden und ist unter Umständen schneller als `seq()`. `seq_len()` erzeugt ganzzahlige Folgen einer vorgegebenen Länge, `seq_along()` generiert ganzzahlige Folgen anhand der Einträge eines Vektors.

```
x_1 = seq.int(0,5)      # wie 0:5, [0,1,2,3,4,5]
x_2 = seq_len(5)        # Natürliche Zahlen bis 5, [1,2,3,4,5]
x_3 = seq_along(c("a","b")) # wie seq_len(length(c("a", "b")))
```

```
x_1
```

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

```
x_2
```

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

```
x_3
```

```
[1] 1 2
```

5.2 Charakterisierung

Zur Charakterisierung von Vektoren stehen mit `length()` und `typeof()` Funktionen bereit, die die Anzahl der Einträge eines Vektors und den Datentyp seiner Einträge ausgeben, wie folgende Beispiele verdeutlichen.

```
x = 0:10          # Vektor  
length(x)         # Anzahl der Elemente des Vektors
```

```
[1] 11
```

```
x = 1:3L          # Vektor  
typeof(x)         # Datentyp des atomaren Vektors
```

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

```
y = c(T,F,T)      # Vektor  
typeof(y)         # Der Datentyp des atomaren Vektors
```

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

`is.logical()`, `is.double()`, `is.integer()`, `is.character()` testen den Datentyp eines Vektors und geben einen logischen Binärwert aus.

```
is.double(x)       # Testen obigen Vektors
```

```
[1] FALSE
```

```
is.logical(y)      # Testen obigen Vektors
```

```
[1] TRUE
```

Datentypangleichung (Coercion)

Versucht man in **R** verschiedene Datentypen in einem Vektor zu konkatenieren, zum Beispiel Elementarwerte vom Typ `numeric`, `character` und `logical`, so erzwingt **R** für den resultierenden Vektor einen einheitlichen Datentyp. Es gilt dabei folgende Rangfolge:

`character > double > integer > logical`

Ist also zum Beispiel einer der Elementarwerte vom Typ `character`, so ist der resultierende Vektor vom Typ `character` und numerische oder logische Werte werden in diesen umgewandelt.

```
x = c(1, "a", TRUE)           # Konkatenation verschiedener Datentypen
typeof(x)                     # erzwungener Datentyp
```

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

```
x                               # Zeichenvektor
```

```
[1] "1"    "a"    "TRUE"
```

Ist dagegen zum Beispiel einer der Elementarwerte vom Typ `integer` und ein anderer vom Typ `logical`, so ist der resultierende Vektor vom Typ `integer` und die logischen Werte werden in ganze Zahlen umgewandelt.

```
x = c(1L, TRUE, FALSE)        # Gemischte Typen: integer vor logical
typeof(x)                     # Erzeugter Vektor ist vom Typ integer
```

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

```
x
```

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

Da **R** diese Datentypangleichung implizit vornimmt, sollte man darauf achten, nur Werte gleichen Datentyps mit der Funktion `c()` zu konkatenieren. Um möglichst flexibel zu sein, nimmt **R** auch noch subtilere Datenangleichungen vor. So ist es zum Beispiel möglich, die Summe logischer Werte zu berechnen, wobei die logischen Werte `TRUE` und `FALSE` implizit als die numerischen Werte 1 und 0 behandelt werden.

```
x = c(TRUE, FALSE, TRUE, TRUE) # logical-Vektor
s = sum(x)                     # Logische Werte als ganze Zahlen summieren
s
```

```
[1] 3
```

Explizite Datentypangleichung erlauben die Funktionen `as.logical()`, `as.integer()`, `as.double()` und `as.character()`. Wendet man diese auf einen Vektor an, so ist der resultierende Vektor vom gewünschten Typ.

```
x = c(0,1,1,0) # double-Vektor
typeof(x)
```

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

```
x
```

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

```
y = as.logical(x) # ... umgewandelt in logical
typeof(y)
```

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

```
y
```

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

```
z = as.integer(x)
typeof(z)
```

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

```
z
```

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

5.3 Indizierung

In **R** und vielen anderen Programmiersprachen werden einzelne oder mehrere Vektoreinträge durch *Indizierung* adressiert. Indizierung wird dabei auch *indexing*, *subsetting* oder *slicing* genannt. In **R** werden zur Indizierung eckige Klammern der Form `[ ]` benutzt. **R** nutzt *one-based indexing*, das heißt, der Index des ersten Elements ist 1 (und nicht etwa 0 wie in Python).

Die Indizierung von Vektoreinträgen kann genutzt werden, um einen bestimmten Eintrag eines Vektors in eine andere Datenstruktur zu kopieren:

```
x = c("a", "b", "c") # Zeichenvektor ["a", "b", "c"]
y = x[2]             # Kopie von "b" in y
y                    # Ausgabe
```

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

Ebenso kann die Indizierung von Vektoreinträgen genutzt werden, um einen bestimmten Eintrag des indizierten Vektors zu manipulieren:

```
x = c("a", "b", "c")      # Zeichenvektor ["a", "b", "c"]
x[3] = "d"                # Änderung von x zu x = ["a", "b", "d"]
x                          # Ausgabe
```

```
[1] "a" "b" "d"
```

Prinzipiell gilt in **R**, dass die Indizierung

- (1) mit einem Vektor positiver Zahlen die entsprechenden Einträge,
- (2) mit einem Vektor negativer Zahlen die entsprechenden komplementären Einträge,
- (3) mit einem logischen Vektor die Einträge mit dem logischen Wert **TRUE** und
- (4) mit einem **character**-Vektor die entsprechend benannten Einträge adressiert.

Wir wollen diese Prinzipien anhand folgender Beispiele verdeutlichen:

(1) Indizierung mit einem Vektor positiver Zahlen

```
x = c(1,4,9,16,25)      # [1,4,9,16,25] = [1^2, 2^2, 3^2, 4^2, 5^2]
y = x[1:3]              # Indizes 1 bis 3: [1,4,9]
z = x[c(1,3,5)]         # Indizes 1, 3 und 5: [1,9,25]
x
```

```
[1] 1 4 9 16 25
```

```
y
```

```
[1] 1 4 9
```

```
z
```

```
[1] 1 9 25
```

(2) Indizierung mit einem Vektor negativer Zahlen

```
x = c(1,4,9,16,25)      # [1,4,9,16,25] = [1^2, 2^2, 3^2, 4^2, 5^2]
y = x[c(-2,-4)]          # Ohne Elemente 2 und 4: [1,9,25]
x
```

```
[1] 1 4 9 16 25
```

```
y
```

```
[1] 1 9 25
```

```
z = x[c(-1,2)]          # Gemischte Vorzeichen ergeben einen Fehler
```

```
Error in `x[c(-1, 2)]`:
! only 0's may be mixed with negative subscripts
```

(3) Indizierung mit einem logischen Vektor

```
x = c(1,4,9,16,25)           # [1,4,9,16,25] = [1^2, 2^2, 3^2, 4^2, 5^2]
y = x[c(T,T,F,F,T)]         # TRUE-Komponenten auswählen: [1,4,25]
z = x[x > 5]                 # x > 5 = [F,F,T,T,T], x[x > 5] = [9,16,25]
x
```

```
[1] 1 4 9 16 25
```

```
y
```

```
[1] 1 4 25
```

```
z
```

```
[1] 9 16 25
```

(4) Indizierung mit einem `character`-Vektor

```
x = c(1,4,9,16,25)           # [1,4,9,16,25] = [1^2, 2^2, 3^2, 4^2, 5^2]
names(x) = c("a","b")        # Namen: [a, b, NA, NA, NA]
y = x["a"]                   # x["a"] = 1
```

Es bleibt anzumerken, dass **R** eine hohe Flexibilität bei Indizierung aufweist, also oft keine Fehler ausgibt, obwohl eine Indizierung per se keinen Sinn ergibt. So verursachen zum Beispiel Out-of-range-Indizes, also Indizes, die die Anzahl der Einträge eines Vektors überschreiten, keine Fehler, sondern geben NA (not available) aus.

```
x = c(1,4,9,16,25)           # [1,4,9,16,25] = [1^2, 2^2, 3^2, 4^2, 5^2]
y = x[10]                    # x[10] = NA (fehlender Wert)
y
```

```
[1] NA
```

Weiterhin verursachen nichtganzzahlige Indizes keine Fehler, sondern werden durch Abschneiden der Nachkommastellen in Richtung null in ganze Zahlen umgewandelt.

```
y = x[4.9]                   # x[4.9] = x[4] = 16
y
```

```
[1] 16
```

```
z = x[-4.9]                   # x[-4.9] = x[-4] = [1,4,9,25]
z
```

```
[1] 1 4 9 25
```

Leere Indizes schließlich indizieren den gesamten Vektor.

```
y = x[]                                # y = x
```

Die Flexibilität von **R** bei nicht sinnhafter Indizierung hat einerseits den Vorteil, dass Programme seltener mit einem Fehler unterbrochen werden. Auf der anderen Seite hat sie den Nachteil, dass unbeabsichtigte Fehlindizierungen unbemerkt bleiben und Datenanalyseprogramme so zu nicht beabsichtigten Ergebnissen kommen können. Im Umgang mit der Indizierung in **R** ist also ein hohes Maß an Aufmerksamkeit und im Idealfall die Validierung von erstellten Datenanalyseprogrammen anhand simulierter Daten mit bekannten Ergebnissen geboten. Ein ähnliches Maß an Vorsicht ist aufgrund von **R**s *Recycling*-Funktionsweise angezeigt, die wir im nächsten Abschnitt kennenlernen werden.

5.4 Arithmetik numerischer Vektoren

Mit **R**-Vektoren kann man arithmetische Operationen ausführen, das heißt, man kann mit ihnen rechnen. Wir unterscheiden dabei *unäre arithmetische Operationen*, also arithmetische Operationen, die auf einen Vektor angewendet werden, und *binäre arithmetische Operationen*, bei denen zwei Vektoren miteinander verknüpft werden. Allgemein gilt, dass arithmetische Vektoroperatoren in **R** elementweise ausgewertet werden, allerdings sind bei binären Operatoren einige Besonderheiten zu beachten.

Betrachten wir zunächst einige unäre arithmetische Operatoren und die Anwendung von Funktionen auf einen Vektor. In folgendem Beispiel werden die unären Operatoren `-` und `^2` sowie die Funktion `log()` auf einen Vektor `a` angewendet. Die Auswertung der Operatoren bzw. der Funktion geschieht dabei *elementweise*, das heißt, der Operator oder die Funktion wird auf jedes Element des Vektors unabhängig von seinen anderen Elementen angewendet.

```
a = seq(0,1,len=11)                    # a = [ 0.0 , 0.1 , ..., 0.9 , 1.0 ]
a
```

```
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
b = -a                                  # b = [-0.0 , -0.1 , ..., -0.9 , -1.0 ]
b
```

```
[1] 0.0 -0.1 -0.2 -0.3 -0.4 -0.5 -0.6 -0.7 -0.8 -0.9 -1.0
```

```
v = a^2                                 # v = [ 0.0^2 , 0.1^2 , ..., 0.9^2 , 1.0^2 ]
v
```

```
[1] 0.00 0.01 0.04 0.09 0.16 0.25 0.36 0.49 0.64 0.81 1.00
```

```
w = log(a)                             # [ln(0), ln(0.1), ..., ln(1)]
w
```

```
[1]      -Inf -2.3025851 -1.6094379 -1.2039728 -0.9162907 -0.6931472
[7] -0.5108256 -0.3566749 -0.2231436 -0.1053605  0.0000000
```

Die Verknüpfung zweier Vektoren durch binäre arithmetische Operatoren geschieht ebenfalls elementweise. Bei der Verknüpfung zweier Vektoren gleicher Länge werden dabei die Vektoreinträge mit den gleichen Indizes anhand des binären arithmetischen Operators verknüpft. Als Resultat ergibt sich ein Vektor mit der gleichen Länge wie die beiden Ausgangsvektoren.

```
a = c(1,2,3)      # a = [1,2,3]
b = c(2,1,4)      # b = [2,1,4]
v = a + b          # [1+2, 2+1, 3+4] = [3, 3, 7]
v
```

```
[1] 3 3 7
```

```
w = a - b          # [1-2, 2-1, 3-4] = [-1, 1, -1]
w
```

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

```
x = a * b          # [1*2, 2*1, 3*4] = [2, 2, 12]
x
```

```
[1] 2 2 12
```

```
y = a / b          # [1/2, 2/1, 3/4] = [0.5, 2, 0.75]
y
```

```
[1] 0.50 2.00 0.75
```

Verknüpft man in **R** einen Vektor und einen *Skalar*, also einen Vektor der Länge 1, so erweitert **R** den Skalar auf einen Vektor der gleichen Länge und führt die elementweise Verknüpfung dieser Vektoren aus. Folgende Beispiele verdeutlichen dies.

```
a = c(1,2,3)      # a = [1,2,3]
b = 2              # b = [2]
v = a + b          # [1+2, 2+2, 3+2] = [3, 4, 5]
v
```

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

```
w = a - b          # [1-2, 2-2, 3-2] = [-1, 0, 1]
w
```

```
[1] -1 0 1
```

```
x = a * b          # [1*2, 2*2, 3*2] = [2, 4, 6]
x
```

```
[1] 2 4 6
```

```
y = a / b                                # [1/2, 2/2, 3/2] = [0.5, 1, 1.5]
y
```

```
[1] 0.5 1.0 1.5
```

Unter dem Begriff des *Recyclings* erlaubt **R** neben der elementweisen Arithmetik für Vektoren gleicher Länge bzw. für Skalare und Vektoren außerdem auch eine Arithmetik mit Vektoren unterschiedlicher Länge. Dazu werden Elemente des kürzeren von zwei Vektoren zum Aufstocken auf die Länge des längeren Vektors genutzt, also in diesem Sinn *recycled*. Wiederum führt diese Eigenschaft der Vektorarithmetik in **R** dazu, dass Programme seltener mit einem Fehler abbrechen und dass das Recycling unter Umständen in informiert geschriebenem Code elegante Lösungen für Berechnungen anbieten kann. Allerdings hat es den klaren Nachteil, dass auch unbeabsichtigte Fehlallokationen von Vektoren, beispielsweise durch das Nichtvorhandensein von Datenpunkten, unentdeckt bleiben können. Auch hier ist also bei der Erstellung von Programmen eine hohe Aufmerksamkeit und insbesondere ausführliches Testen angezeigt. Speziell kann man bei Recycling der Vektorarithmetik in **R** zwei Fälle unterscheiden: Ist die Länge des längeren der beiden Vektoren ein ganzzahliges Vielfaches der Länge des kürzeren Vektors, so wird der kürzere Vektor durch die entsprechend häufige Wiederholung seiner Elemente in ihrer originären Reihenfolge expandiert. Folgendes Beispiel soll dies verdeutlichen.

```
x = 1:2                                # x = [1,2], length(x) = 2
x
```

```
[1] 1 2
```

```
y = 3:6                                # y = [3,4,5,6], length(y) = 4
y
```

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

```
v = x + y                                # v = [1,2,1,2] + [3,4,5,6] = [4,6,6,8]
v
```

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

In diesem Sinn ist dann die Arithmetik von Vektoren und Skalaren, also Vektoren der Länge 1, ein Spezialfall dieses Prinzips. Ist die Länge des längeren der beiden Vektoren kein ganzzahliges Vielfaches der Länge des kürzeren Vektors, so werden Elemente des kürzeren Vektors in ihrer originären Reihenfolge so lange zum Auffüllen des kürzeren Vektors genutzt, bis beide Vektoren die gleiche Länge haben. Folgendes Beispiel soll dies verdeutlichen.

```
x = c(1,3,5)                            # x = [1,3,5], length(x) = 3
x
y = c(2,4,6,8,10)                        # y = [2,4,6,8,10], length(y) = 5
y
v = x + y                                # Wiederholtes x + y: [3,7,11,9,13]
v
```

5.5 Attribute

Attribute in **R** sind Metadaten von Objekten in Form von Schlüssel-Wert-Paaren. Generell ruft die Funktion `attributes()` alle Attribute eines Objektes auf. Vektoren haben per se keine Attribute.

```
a = 1:3                                # ein numerischer Vektor
attributes(a)                          # Aufrufen aller Attribute von a
```

NULL

Die Funktion `attr()` kann zum Aufrufen und Definieren von Attributen genutzt werden. Folgender Code definiert für den Vektor `a` ein Attribut mit dem Schlüssel (Namen) `"S"` und dem Wert `"W"`.

```
attr(a, "S") = "W"                    # Attribut mit Schlüssel S und Wert W setzen
attr(a, "S")                          # Attribut S mit Wert W auslesen
```

```
[1] "W"
```

Allerdings werden Attribute bei Operationen oft entfernt.

```
b = a[1]                              # Erstes Element von a in b kopieren
attributes(b)                          # Aufrufen aller Attribute von b
```

NULL

Die Spezifikation des Attributs `names` gibt den Elementen eines Vektors Namen, die zur Indizierung genutzt werden können und die bei Operationen nicht entfernt werden.

```
v = c(x=1,y=2,z=3)                    # Elementnamengeneration bei Vektorerzeugung
v                                       # Vektorausgabe
```

```
x y z
1 2 3
```

Die Indizierung mithilfe von Namen hat folgende Form:

```
v["y"]                               # Indizierung per Namen
```

```
y
2
```

Im Nachgang der Definition eines Vektors kann die Funktion `names()` genutzt werden, um für die Vektoreinträge Namen zu definieren und diese nachzuschlagen.

```
y = 4:6                                # Erzeugung eines Vektors
names(y) = c("a","b","c")              # Definition von Namen
names(y)                                # Elementnamenaufruf
```

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

Benannte Vektoreinträge können hilfreich sein, wenn der Vektor eine Sinneinheit bildet und zum Beispiel Eigenschaften einer Versuchsperson wie Alter, Größe und Gewicht repräsentiert.

```
p = c(age      = 31,                # Alter (Jahre)
      height   = 198,              # Größe (cm)
      weight    = 75)              # Gewicht (kg)
p                                  # Vektorausgabe
```

```
age height weight
31    198     75
```

Allerdings ist die Arbeit mit benannten Vektoreinträgen eher selten. Das Prinzip, Einträge einer Datenstruktur anstelle eines numerischen Indizes mit einer sinntragenden Bezeichnung zu versehen, ist in **R** allerdings nichtsdestotrotz übliche Praxis, wie wir im Umgang mit der zentralen Datenstruktur der Dataframes in Kapitel 7 sehen werden.

6 Matrizen und Arrays

R-Matrizen und Arrays können als höherdimensionale Generalisierungen von **R**-Vektoren verstanden werden. Wie bei den Vektoren ist es wichtig, Matrizen in der Programmierung von ihren gleichnamigen mathematischen Objekten abzugrenzen. So können Matrizen in der Programmierung einfach der tabellarischen Speicherung numerischer Daten dienen, ohne dass dabei das primäre Ziel die Anwendung der mathematischen Matrizenrechnung sein muss. Andersherum bilden die elementweisen arithmetischen Operationen mit Matrizen in der Programmierung in der Mathematik eher sehr spezielle mathematische Operationen wie das Hadamard-Produkt ab. In diesem Abschnitt wollen wir Matrizen und Arrays primär als Datencontainer in der imperativen Programmierung mit **R** untersuchen. Eine Einführung in die Matrizenrechnung mit den gleichen Datenstrukturen gibt [Kapitel „Matrizen“ im PDWP-Lehrbuch](#).

6.1 Matrizen

Abstrakt gesprochen sind **R**-Matrizen zweidimensionale, rechteckige Datenstrukturen der Form

$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n_c} \\ m_{21} & m_{22} & \cdots & m_{2n_c} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n_r1} & m_{n_r2} & \cdots & m_{n_r n_c} \end{pmatrix} \quad (6.1)$$

Die Elemente m_{ij} , $i = 1, \dots, n_r$, $j = 1, \dots, n_c$ der Matrix sind dabei notwendigerweise vom gleichen Datentyp. In obigem Schema bezeichnet n_r die Anzahl der Zeilen (rows) und n_c die Anzahl der Spalten (columns) der Matrix M . Jedes Element einer Matrix hat damit einen Zeilenindex i und einen Spaltenindex j . Intuitiv sind **R**-Matrizen also numerisch indizierte Tabellen. Formal sind Matrizen in **R** zweidimensional interpretierte atomare Vektoren.

6.1.1 Erzeugung

Zur Erzeugung einer Matrix wird die **R**-Funktion `matrix()` genutzt, die Matrizen mit den Elementen eines Vektors befüllt. Die allgemeine Syntax dieser Funktion ist `M = matrix(v, nrow, ncol, byrow)`, wobei

- `v` einen Vektor der Länge `nrow · ncol`,
- `nrow` die gewünschte Zeilenanzahl von `M`,
- `ncol` die gewünschte Spaltenanzahl von `M`,
- die logische Variable `byrow` die Befüllungsordnung von `M`

bezeichnen. Da sich bei gegebener Länge von `v` und gegebener Zeilen- oder Spaltenanzahl die entsprechende Spalten- oder Zeilenanzahl der Matrix ergibt, ist es lediglich nötig, entweder `nrow` oder `ncol` zu spezifizieren, wie folgende Beispiele zeigen.

```
matrix(c(1:12), nrow = 3)           # 3 x 4-Matrix 1,...,12; spaltenweise
```

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

```
matrix(c(1:12), ncol = 4)           # 3 x 4-Matrix 1,...,12; spaltenweise
```

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

```
matrix(c(1:12), nrow = 3, byrow = TRUE) # 3 x 4-Matrix 1,...,12; zeilenweise
```

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

Bestehende Matrizen gleicher Zeilenanzahl können mit der Funktion `cbind()` spaltenweise konkateniert werden.

```
A = matrix(c(1:4), nrow = 2)        # 2 x 2-Matrix der Zahlen 1,...,4
A
```

```
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
B = matrix(c(5:10), nrow = 2)       # 2 x 3-Matrix der Zahlen 5,...,10
B
```

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

```
C = cbind(A,B)                       # spaltenweise Konkatenierung von A und B
C
```

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

Äquivalent können bestehende Matrizen gleicher Spaltenanzahl mit der Funktion `rbind()` zeilenweise konkateniert werden.

```
A = matrix(c(1:6) , nrow = 2, byrow = TRUE)      # 2 x 3-Matrix der Zahlen 1,...,6
print(A)
```

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

```
B = matrix(c(7:9), nrow = 1)                    # 1 x 3-Matrix der Zahlen 7,...,9
print(B)
```

```
      [,1] [,2] [,3]
[1,]    7    8    9
```

```
C = rbind(A,B)                                  # zeilenweise Konkatenierung von A und B
print(C)
```

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

6.1.2 Charakterisierung

Mit der vertrauten Funktion `typeof()` kann der elementare Datentyp einer Matrix ausgegeben werden:

```
typeof(matrix(c(1,0,0,1) , nrow = 2))          # 2 x 2-Matrix vom Typ double
```

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

```
typeof(matrix(c(1L,0L,0L,1L), nrow = 2))      # 2 x 2-Matrix vom Typ integer
```

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

```
typeof(matrix(c(T,T,F,F) , nrow = 2))        # 2 x 2-Matrix vom Typ logical
```

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

```
typeof(matrix(c("a","b","c"), nrow = 1))     # 1 x 3-Matrix vom Typ character
```

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

Die Zeilen- bzw. Spaltenanzahl einer Matrix kann mit den Funktionen `nrow()` (number of rows) und `ncol()` (number of columns) ausgegeben werden.

```
C = matrix(1:12, nrow = 3)      # 3 x 4-Matrix
nrow(C)                        # Anzahl Zeilen
```

```
[1] 3
```

```
ncol(C)                        # Anzahl Spalten
```

```
[1] 4
```

6.1.3 Indizierung

Generell gilt, dass Matricelemente mit einem Zeilenindex und einem Spaltenindex indiziert werden. Der erste zweier Matrixindizes indiziert dabei immer die Zeile, der zweite immer die Spalte. Die übrigen Prinzipien der Indizierung von Matrizen entsprechen im Wesentlichen den Prinzipien der Indizierung von Vektoren. Eine Besonderheit ist, dass ein fehlender Zeilen- oder Spaltenindex alle Elemente der entsprechenden Dimension indiziert.

```
A = matrix(c(2:7)^2, nrow = 2)  # 2 x 3-Matrix der Zahlen 2^2,...,7^2
A
```

```
      [,1] [,2] [,3]
[1,]    4   16   36
[2,]    9   25   49
```

```
A[1,3]                        # Element in 1. Zeile, 3. Spalte von A [36]
```

```
[1] 36
```

```
A[2,2]                        # Element in 2. Zeile, 2. Spalte von A [25]
```

```
[1] 25
```

```
A[2,]                         # Alle Elemente der 2. Zeile [9,25,49]
```

```
[1]  9 25 49
```

```
A[,3]                         # Alle Elemente der 3. Spalte [36,49]
```

```
[1] 36 49
```

```
A[1:2,1:2]                   # Erste zwei Zeilen und Spalten
```

```
      [,1] [,2]
[1,]    4   16
[2,]    9   25
```

```
A[A>10] # Elemente von A größer 10 [16,25,36,49]
```

```
[1] 16 25 36 49
```

```
A[1,c(F,F,T)] # Element in 1. Zeile, 3. Spalte von A [36]
```

```
[1] 36
```

6.1.4 Arithmetik

Wendet man unäre arithmetische Operatoren und Funktionen auf Matrizen an, so werden diese wie bei den Vektoren elementweise ausgewertet.

```
A = matrix(c(1:4), nrow = 2) # 2 x 2-Matrix der Zahlen 1,2,3,4
A
```

```
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
B = A^2 # B[i,j] = A[i,j]^2, 1 <= i,j <= 2
B
```

```
      [,1] [,2]
[1,]    1    9
[2,]    4   16
```

```
C = sqrt(B) # C[i,j] = sqrt(A[i,j]^2), 1 <= i,j <= 2
C
```

```
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
D = exp(A) # D[i,j] = exp(A[i,j]), 1 <= i,j <= 2
D
```

```
      [,1]      [,2]
[1,] 2.718282 20.08554
[2,] 7.389056 54.59815
```

Wie Vektoren können auch Matrizen mit identischen Zeilen- und Spaltenanzahlen mit den binären arithmetischen Operatoren $+$, $-$, $*$ und $/$ elementweise verknüpft werden. Folgende Beispiele verdeutlichen dies:

```
A = matrix(c(1:4), nrow = 2) # 2 x 2-Matrix der Zahlen 1,2,3,4
A
```

```

      [,1] [,2]
[1,]    1    3
[2,]    2    4

```

```

B = matrix(c(5:8), nrow = 2)           # 2 x 2-Matrix der Zahlen 5,6,7,8
B

```

```

      [,1] [,2]
[1,]    5    7
[2,]    6    8

```

```

C = A + B                               # C[i,j] = A[i,j] + B[i,j], 1 <= i,j <= 2
C

```

```

      [,1] [,2]
[1,]    6   10
[2,]    8   12

```

```

D = A * B                               # D[i,j] = A[i,j] * B[i,j], 1 <= i,j <= 2
D

```

```

      [,1] [,2]
[1,]    5   21
[2,]   12   32

```

Darüber hinaus kann mit **R**-Matrizen im Sinne der Matrizenrechnung der linearen Algebra gerechnet werden. Wir zeigen hier nur einige wenige Beispiele, eine ausführliche Darstellung der mathematischen Prinzipien und ihrer **R**-Implementierung findet sich in [Kapitel „Matrizen“ im PDWP-Lehrbuch](#).

```

C = A %*% B                             # 2 x 2-Matrixprodukt
C

```

```

      [,1] [,2]
[1,]   23   31
[2,]   34   46

```

```

A_T = t(A)                              # Transposition von A
A_T

```

```

      [,1] [,2]
[1,]    1    2
[2,]    3    4

```

```

A_inv = solve(A)                        # Inverse von A
A_inv

```

```

      [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5

```

```
A_det = det(A) # Determinante von A
A_det
```

```
[1] -2
```

6.1.5 Attribute

Formal sind Matrizen atomare Vektoren mit einem `dim`-Attribut.

```
A = matrix(1:12, nrow = 4 ) # 4 x 3-Matrix
attributes(A) # Aufrufen der Attribute von A
```

```
$dim
[1] 4 3
```

Wie bei den Vektoren kann man auch den Matrizenelementen Namen geben, allerdings ist dies auch bei Matrizen eher ungewöhnlich. Speziell definieren die Funktionen `rownames()` und `colnames()` das Matrixattribut `dimnames`.

```
rownames(A) = c("P1", "P2", "P3", "P4") # Benennung der Zeilen von A
colnames(A) = c("Age", "Hgt", "Wgt") # Benennung der Spalten von A
A # A mit Attribut dimnames
```

```
      Age Hgt Wgt
P1     1   5   9
P2     2   6  10
P3     3   7  11
P4     4   8  12
```

```
attr(A, "dimnames") # Aufrufen des Attributs dimnames
```

```
[[1]]
[1] "P1" "P2" "P3" "P4"
```

```
[[2]]
[1] "Age" "Hgt" "Wgt"
```

6.2 Arrays

R-Arrays sind d -dimensionale, hyperrechteckige Datenstrukturen. Intuitiv können Arrays als die Generalisierung von Matrizen auf mehr als zwei Dimensionen verstanden werden. Für $d = 1$ entspricht ein Array einem Vektor, für $d = 2$ entspricht ein Array einer Matrix. Wie bei Vektoren und Matrizen müssen die Elemente eines Arrays vom gleichen Datentyp sein. Arrays sind dadurch charakterisiert, wie viele Elemente $d_i, i = 1, \dots, d$ die i -te Dimension repräsentieren kann. Jedes Element eines Arrays wird dabei durch einen d -dimensionalen Index $i_1, i_2, \dots, i_d$ indiziert.

6.2.1 Erzeugung

In Analogie zur Funktion `matrix()` wird die Funktion `array()` genutzt, um ein Array mit Vektorelementen zu befüllen. Die allgemeine Syntax dieser Funktion ist `A = array(v, dim)`, wobei `v` ein Vektor von Arraydaten und `dim` ein Vektor ganzzahliger Werte ist, der die maximalen Indizes in jeder der `length(dim)` Dimensionen des Arrays repräsentiert. So erzeugt beispielsweise folgender Befehl ein $2 \times 2 \times 3$ -Array.

```
A = array(1:12, dim = c(2,2,3))      # 2 x 2 x 3-Array der Zahlen 1,...,12
A
```

```
, , 1
     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2
     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3
     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

Dreidimensionale Arrays kann man sich intuitiv als im Raum hintereinander aufgestellte Matrizen vorstellen. Die dritte Dimension repräsentiert dann dabei die “Seiten” des Arrays. Allerdings ist es im Umgang mit höherdimensionalen Arrays sinnvoll, zu versuchen, sich von räumlichen Vorstellungen von Arrays zu lösen und Arrays stattdessen als Datencontainer zu verstehen, deren Elemente eindeutig durch ihre Indizes adressiert werden können.

6.2.2 Charakterisierung

Die `length()`-Funktion gibt die Anzahl der Elemente eines Arrays aus.

```
A = array(1:12, dim = c(2,2,3))      # 2 x 2 x 3-Array der Zahlen 1,...,12
length(A)                             # Die Anzahl der Elemente des Arrays
```

```
[1] 12
```

Die Funktion `typeof()` gibt den elementaren Datentyp eines Arrays aus:

```
typeof(A)                             # Der Typ des Arrays
```

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

Die Funktion `dim()` gibt die Dimensionen eines Arrays aus:

```
dim(A)                                     # Dimension des Arrays
```

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

6.2.3 Indizierung

Die Indizierung von Arrays erfolgt analog zu Vektor- und Matrixindizierung, mit dem Unterschied, dass zur Indizierung eines Arrayelements drei oder mehr Indizes benötigt werden.

```
A      = array(1:12, dim = c(2,2,3))      # 2 x 2 x 3-Array der Zahlen 1,...,12
a_223  = A[2,2,3]                        # Arrayelement mit Indexadresse [2,2,3] (12)
```

6.2.4 Arithmetik

Wie bei Vektoren und Matrizen werden unäre arithmetische Operatoren bei Arrays elementweise ausgewertet. Ebenso analog werden binäre arithmetische Operatoren bei Arrays identischer Dimensionalität elementweise ausgewertet.

```
A      = array(1:4, dim = c(1,2,2))      # 1 x 2 x 2-Array der Zahlen 1,...,4
A
```

```
, , 1
```

```
    [,1] [,2]
[1,]    1    2
```

```
, , 2
```

```
    [,1] [,2]
[1,]    3    4
```

```
B      = array(5:8, dim = c(1,2,2))      # 1 x 2 x 2-Array der Zahlen 5,...,8
B
```

```
, , 1
```

```
    [,1] [,2]
[1,]    5    6
```

```
, , 2
```

```
    [,1] [,2]
[1,]    7    8
```

```
C      = A^2                             # elementweise Potenzierung
C
```

```
, , 1
      [,1] [,2]
[1,]    1    4
```

```
, , 2
      [,1] [,2]
[1,]    9   16
```

```
D      = A + B      # elementweise Addition
D
```

```
, , 1
      [,1] [,2]
[1,]    6    8
```

```
, , 2
      [,1] [,2]
[1,]   10   12
```

6.2.5 Attribute

Formal sind Arrays atomare Vektoren mit einem `dim`-Attribut.

```
A = array(1:12, dim = c(2,2,3))      # 2 x 2 x 3-Array der Zahlen 1,...,12
attributes(A)                        # Aufrufen der Attribute von A
```

```
$dim
[1] 2 2 3
```

Wie bei den Matrizen kann die Funktion `dimnames()` zur Benennung der Arraydimensionen benutzt werden, aber auch hier ist die Benennung von Arraydimensionen eher ungewöhnlich.

7 Listen, Dataframes und Tibbles

7.1 Listen

R-Listen sind geordnete Folgen von **R**-Objekten. Listen werden als rekursiver Datentyp bezeichnet, da sie Objekte verschiedener Datentypen, insbesondere auch wiederum Listen, bündeln können. Obwohl sich die intuitive Sichtweise, dass Listen Objekte “enthalten”, anbietet, “enthalten” Listen de facto keine Objekte, sondern lediglich die Referenzen zu den entsprechenden Objekten im Arbeitsspeicher (Abbildung 7.1). Listen sind ein wesentlicher Baustein von **R**-Dataframes.

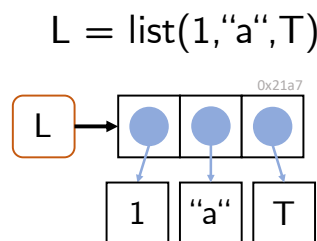


Abbildung 7.1 Listenrepräsentation. Eine Liste in **R** ist eine indizierbare Folge von Referenzen zu **R**-Objekten im Arbeitsspeicher.

Erzeugung

Listen können durch die direkte Konkatenation der Listenelemente mithilfe der Funktion `list()` erzeugt werden.

```
# Erzeugung einer Liste mit einem Vektor-, Matrix- und Funktionselement
L = list(c(1,4,5), matrix(1:8, nrow = 2), exp)
print(L)
```

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

[[2]]
  [,1] [,2] [,3] [,4]
[1,]   1   3   5   7
[2,]   2   4   6   8

[[3]]
function (x) .Primitive("exp")
```

Insbesondere können auch Listen selbst Elemente von Listen sein, was ihre Bezeichnung als *rekursiver* Datentyp motiviert.

```
L = list(list(1))          # Liste mit Element 1 in einer Liste
print(L)                  # Ausgabe
```

Die von den **R**-Vektoren vertraute Funktion `c()` kann auch zum Konkatenieren von mehreren Listen genutzt werden.

```
L = c(list(pi), list("a")) # Konkatenation zweier Listen
print(L)                  # Ausgabe
```

Charakterisierung

Der Datentyp von Listen ist `list`:

```
L = list(1:2, "a", log)    # Erzeugung einer Liste
typeof(L)                  # Typenbestimmung
```

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

`length()` gibt die Anzahl der Listenelemente auf dem obersten Listenlevel aus, die Elementinhalte werden also ignoriert:

```
L = list(1:2, list("a", pi), exp) # Drei Elemente auf oberster Ebene
length(L)                        # length() ignoriert Elementinhalte
```

```
[1] 3
```

Die Dimension, Zeilen- und Spaltenanzahlen von Listen sind `NULL`:

```
L = list(1:2, "a", sin)     # eine Liste
dim(L)                      # Die Dimension von Listen ist NULL
```

```
NULL
```

```
nrow(L)                     # Die Zeilenanzahl von Listen ist NULL
```

```
NULL
```

```
ncol(L)                     # Die Spaltenanzahl von Listen ist NULL
```

```
NULL
```

Indizierung

Bei der Indizierung von Listen ist zu beachten, dass Listen auf zwei unterschiedliche Arten indiziert werden können. Einfache eckige Klammern `[ ]` indizieren Listenelemente als Listen:

```
L = list(1:3, "a", exp)      # eine Liste
l1 = L[1]                  # Indizierung eines Listenelements
print(l1)                  # Ausgabe
```

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

```
typeof(l1)                 # Typbestimmung von l1
```

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

Doppelte eckige Klammern `[[ ]]` indizieren den Inhalt von Listenelementen:

```
L = list(1:3, "a", exp)      # eine Liste
i2 = L[[2]]                  # Indizierung des Listenelementinhalts
print(i2)                    # Ausgabe des Inhalts von Listenelement L[2]
typeof(i2)                   # Typbestimmung von i2
```

Auch beim Ersetzen von Listeninhalten sind diese Konventionen zu beachten. So ist es zum Beispiel nicht möglich, durch den Ausdruck `L[1] = c(1,2,3)` das erste Listenelement zu ändern, da der Vektor `c(1,2,3)` keine Liste ist:

```
L      = list(1:3, "a", exp)      # eine Liste
L[1]    = 4:6                     # keine Typkonversion, Fehlermeldung
```

```
Warning in L[1] = 4:6: number of items to replace is not a multiple of
replacement length
```

```
L[1]    = list(4:6)               # Ersetzung des 1. Listenelementes
L[[3]]   = "c"                    # Ersetzung des 3. Listenelementinhaltes
```

Indizierung

Die Prinzipien der Listenindizierung sind analog zu denen der Vektorindizierung. Vektoren positiver Zahlen adressieren die entsprechenden Elemente:

```
L = list(1:3, "a", pi)      # eine Liste
l = L[c(1,3)]               # 1. und 3. Listenelement
l
```

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

```
[[2]]
[1] 3.141593
```

Vektoren negativer Zahlen adressieren die zu ihren Werten komplementären Elemente:

```
L = list(1:3, "a", pi)           # eine Liste
l = L[-c(1,3)]                  # 2. Listenelement
l
```

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

Logische Vektoren adressieren Elemente mit Index `TRUE`:

```
L = list(1:3, "a", pi)           # eine Liste
l = L[c(T,T,F)]                  # 1. und 2. Listenelement
l
```

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

```
[[2]]
[1] "a"
```

Zur nicht numerischen Adressierung können Listenelementen auch Namen gegeben werden. Dies ist nach der Erzeugung einer Liste mithilfe der `names()`-Funktion möglich:

```
K      = list(1:2, TRUE)           # eine unbenannte Liste
names(K) = c("Frodo", "Sam")       # Namensgebung mit names()
K
```

```
$Frodo
[1] 1 2
```

```
$Sam
[1] TRUE
```

Ebenso ist es möglich, die Namen der Listenelemente direkt bei Erzeugung der Liste durch Zuweisung zu vergeben:

```
L = list(frodo = 1:3, sam = "a", eowyn = "b") # Benannte Liste erzeugen
```

Sowohl Listenelemente als auch Listenelementinhalte können dann mit ihren Namen adressiert werden:

```
L = list(frodo = 1:3, sam = "a", eowyn = "b") # Benannte Liste erzeugen
L["frodo"]                                     # Listenelementausgabe
```

```
$frodo
[1] 1 2 3
```

```
typeof(L["frodo"])                           # Listenelementtyp
```

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

```
L[["frodo"]] # Elementinhalt ausgeben
```

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

```
typeof(L[["sam"]]) # Typ des Elementinhalts
```

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

Insbesondere können Listenelementinhalte auch mit dem `$`-Operator und ihrem Namen indiziert werden. Dies stellt im Rahmen der Dataframes die gebräuchlichste Art der Listenindizierung dar:

```
L = list(frodo = 1:3, sam = "a", eowyn = "b") # Benannte Liste erzeugen  
L$frodo # Elementinhalt ausgeben
```

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

```
typeof(L$frodo) # Typ des Elementinhalts
```

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

```
L$sam # Elementinhalt ausgeben
```

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

```
typeof(L$sam) # Typ des Elementinhalts
```

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

```
L$eowyn # Elementinhalt ausgeben
```

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

```
typeof(L$eowyn) # Typ des Elementinhalts
```

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

Arithmetik

Eine allgemeine Listenarithmetik ist nicht definiert, da Listenelementinhalte ja unterschiedlichen Datentyps sein können, für die unäre oder binäre Operationen nicht definiert sind:

```
L1 = list(1:3, "a" )      # eine Liste
L2 = list(T, exp )       # eine Liste
L1+L2                    # Versuch der Listenaddition
```

```
Error in `L1 + L2`:
! non-numeric argument to binary operator
```

Sind Inhalte von Listenelementen allerdings von einem gemeinsamen Datentyp, für den bestimmte binäre Operationen definiert sind, so können sie auch z. B. arithmetisch verknüpft werden:

```
L1 = list(1:3, pi )      # eine Liste
L2 = list(4:6, exp )     # eine Liste
L1[[1]] + L2[[1]]       # Erste Inhalte addieren: [1+4,2+5,3+6]
```

```
[1] 5 7 9
```

```
L2[[2]](1)              # Zweiten Inhalt anwenden: exp(1)
```

```
[1] 2.718282
```

Copy-on-modify

Wie bei Vektoren gilt bei Listen das *Copy-on-modify*-Prinzip. Speziell gilt bei Listen, dass sogenannte *shallow copies* bei Modifikation einer Liste erzeugt werden. Dabei wird nur das Listenobjekt selbst kopiert und erhält eine neue Adresse im Arbeitsspeicher, nicht aber die durch die Liste gebundenen Objekte, die an den gleichen Adressen im Arbeitsspeicher verbleiben. `lobstr::ref()` erlaubt es, dieses Verhalten nachzuvollziehen.

```
L1 = list(1,2,3)         # Erzeugen einer Liste als Objekt
lobstr::ref(L1)          # Ausgabe der Referenzen
```

```
o [1:0x225a447e6d8] <list>
+--[2:0x225a260a3f8] <dbl>
+--[3:0x225a260a040] <dbl>
\-[4:0x225a261bb28] <dbl>
```

```
L2 = L1                  # L1 und L2 referenzieren dasselbe Objekt
lobstr::ref(L2)          # Ausgabe der Referenzen
```

```
o [1:0x225a447e6d8] <list>
+--[2:0x225a260a3f8] <dbl>
+--[3:0x225a260a040] <dbl>
\-[4:0x225a261bb28] <dbl>
```

```
L2[[3]] = 4              # Flache Kopie bei Änderung
lobstr::ref(L2)          # Ausgabe der Referenzen
```

```
o [1:0x225a4292648] <list>
+-[2:0x225a260a3f8] <dbl>
+-[3:0x225a260a040] <dbl>
\-[4:0x225a2676460] <dbl>
```

Offenbar ändert die Zuweisung $L2 = L1$ zunächst nichts an den Arbeitsspeicheradressen, auf die L1 und L2 sowie ihre Elemente verweisen. Die Modifikation des Inhalts des dritten Listenelements von L2 durch $L2[[3]] = 4$ allerdings ändert dann die Adresse des Listenobjekts im Arbeitsspeicher, da eine Kopie des Listenobjekts erzeugt wird. Gleichzeitig ändern sich gemäß dem Shallow-Copy-Prinzip die Adressen der ersten beiden Inhalte von L2 aber nicht. Abbildung 7.2 visualisiert diese Prinzipien an einem Beispiel.

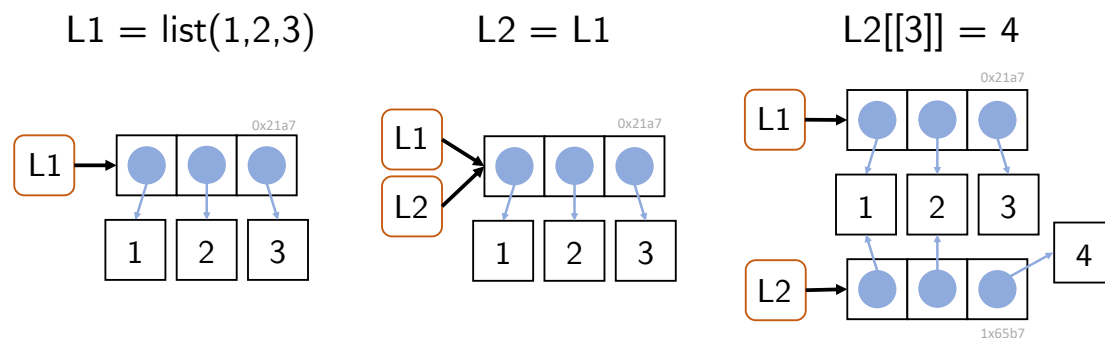


Abbildung 7.2 Effekte der Zuweisung $L2 = L1$ für eine Liste L1 sowie der Modifikation eines Elements von L2.

7.2 Dataframes

R-Dataframes sind die zentrale Datenstruktur in **R**. Dataframes stellt man sich am besten als Tabelle vor, deren Spalten Namen haben. Technisch ist ein Dataframe eine Liste, deren Elemente Vektoren gleicher Länge sind. Die Listenelemente entsprechen dabei den Spalten einer Tabelle und die Vektorelemente an gleicher Position entsprechen den Zeilen einer Tabelle.

Erzeugung

Dataframes werden mit der Funktion `data.frame()` erzeugt. Dabei werden als Argumente der Funktion die Spaltennamen und ihre vektorwertigen Inhalte festgelegt:

```
D = data.frame(                                # Dataframe erzeugen
  x = letters[1:4],                             # 1. Spalte mit dem Namen x
  y = 1:4,                                       # 2. Spalte mit dem Namen y
  z = c(T,T,F,T))                             # 3. Spalte mit dem Namen z
print(D)                                       # Ausgabe des Dataframes als Tabelle
```

```
  x y    z
1 a 1 TRUE
2 b 2 TRUE
3 c 3 FALSE
4 d 4 TRUE
```

```
D = data.frame(c(1,2,3),
               c("a", "b", "c"),
               c(T,F,F))
```

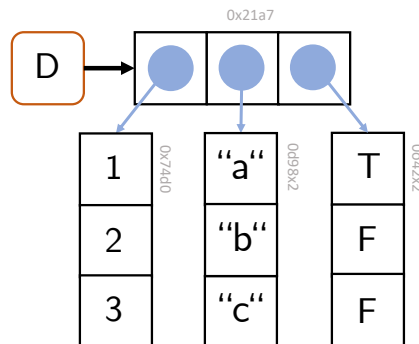


Abbildung 7.3 Dataframerepräsentation. Ein **R**-Dataframe ist eine **R**-Liste, deren Elemente **R**-Vektoren gleicher Länge sind.

Die Spalten des Dataframes müssen die gleiche Länge haben, ansonsten ist ein Dataframe nicht definiert:

```
D = data.frame(                                # Dataframe erzeugen
  x = letters[1:4],                            # 1. Spalte mit dem Namen x
  y = 1:4,                                     # 2. Spalte mit dem Namen y
  z = c(T,T,F))                               # 3. Spalte mit dem Namen z
```

```
Error in `data.frame()` :
! arguments imply differing number of rows: 4, 3
```

Die Spalten eines Dataframes können offenbar unterschiedlichen Datentyps sein, dies ist natürlich mit dem Charakter von **R**-Listen kongruent.

Charakterisierung

Mit den Funktionen `names()` und `colnames()` können die Namen der Spalten eines Dataframes ausgegeben werden, mit der Funktion `rownames()` seine Zeilennamen. Per Default sind die Zeilennamen eines Dataframes die ganzen Zahlen 1,2,...,n:

```
D = data.frame(                                # Dataframe erzeugen
  age   = c(30,35,40,45),                      # 1. Spalte
  height = c(178,189,165,171),                  # 2. Spalte
  weight = c(67, 76, 81, 92))                  # 3. Spalte
names(D)                                       # names gibt die Spaltennamen aus
```

```
[1] "age"      "height"  "weight"
```

```
colnames(D)                                # colnames entspricht names
```

```
[1] "age"      "height" "weight"
```

```
rownames(D)                                # Standardmäßige Zeilenamen: 1, 2, ...
```

```
[1] "1" "2" "3" "4"
```

Des Weiteren ist ein Dataframe durch die Anzahl seiner Zeilen und Spalten charakterisiert. Diese können mit den Funktionen `nrow()` bzw. `ncol()` ausgegeben werden. Die Funktion `length()` gibt für Dataframes wie für Listen die Anzahl der Listeneinträge, bei Dataframes also entsprechend die Spaltenanzahl, aus.

```
nrow(D)                                     # Zeilenanzahl
```

```
[1] 4
```

```
ncol(D)                                     # Spaltenanzahl
```

```
[1] 3
```

```
length(D)                                  # Länge ist die Spaltenanzahl
```

```
[1] 3
```

Eine hilfreiche Funktion zur Charakterisierung eines Dataframes ist die Funktion `str()`, die in kompakter Form wesentliche Aspekte eines Dataframes anzeigt. `str()` steht dabei für *structure* und bezeichnet die Spalten des Dataframes als *variables* und Zeileneinträge als *observations*:

```
str(D)
```

```
'data.frame':  4 obs. of  3 variables:
 $ age   : num  30 35 40 45
 $ height: num  178 189 165 171
 $ weight: num   67  76  81  92
```

Attribute

Technisch sind Dataframes Listen mit Attributen für (column) `names` und `row.names`. Dataframes sind von der class “data.frame”:

```
typeof(D)
```

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

```
attributes(D)
```

```
$names
[1] "age"      "height" "weight"

$class
[1] "data.frame"

$row.names
[1] 1 2 3 4
```

Indizierung

Dataframes kann man sowohl wie Matrizen als auch wie Listen indizieren. Die Prinzipien der Indizierung entsprechen dabei denen von Vektoren. Folgendes Beispiel zeigt zunächst, wie ein Dataframe im Sinne einer Liste mit einfachen Klammern indiziert wird. Das entsprechende Listenelement ist bei Dataframes dann selbst ein Dataframe:

```
D = data.frame(                                # Dataframe erzeugen
  x = letters[1:4],                            # 1. Spalte mit dem Namen x
  y = 1:4,                                     # 2. Spalte mit dem Namen y
  z = c(T,T,F,T))                             # 3. Spalte mit dem Namen z
class(D)                                       # D ist ein Dataframe
```

```
[1] "data.frame"
```

```
v = D[1]                                     # 1. Listenelement als Dataframe
v
```

```
  x
1 a
2 b
3 c
4 d
```

```
class(v)                                     # v ist ein Dataframe
```

```
[1] "data.frame"
```

Des Weiteren können Dataframes wie Listen mit doppelten Klammern indiziert werden. Dadurch werden entsprechend die Inhalte eines Listenelements adressiert:

```
w = D[[1]]                                  # Inhalt des 1. Listenelements
w
```

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

```
class(w) # w ist ein Zeichenvektor
```

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

Die typische Indizierung eines Dataframes erfolgt mit der von den Listen bekannten `$`-Indizierung, die den Inhalt der entsprechenden Dataframe-Spalte adressiert:

```
y = D$y # $ zur Indizierung der y-Spalte
y
```

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

```
class(y) # y ist ein Vektor vom Typ "integer" (!)
```

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

Ist man an speziellen Inhalten des Dataframes in spezifischen Zeilen und Spalten interessiert, so bieten sich die Prinzipien der Matrixindizierung an, wie folgendes Beispiel zeigt:

```
D[2:3,-2] # 1. Index: Zeilen; 2. Index: Spalten
```

```
  x      z
2 b  TRUE
3 c FALSE
```

```
D[c(T,F,T,F),] # 1. Index: Zeilen; 2. Index: Spalten
```

```
  x y      z
1 a 1  TRUE
3 c 3 FALSE
```

```
D[,c("x", "z")] # 1. Index: Zeilen; 2. Index: Spalten
```

```
  x      z
1 a  TRUE
2 b  TRUE
3 c FALSE
4 d  TRUE
```

Copy-on-modify

Zur optimalen Nutzung des Arbeitsspeichers gelten die Copy-on-modify-Prinzipien für Listen auch für Dataframes. Weiterhin gilt dabei, dass die Modifikation einer Spalte zur Kopie der entsprechenden Spalte führt, wohingegen die Modifikation einer Zeile in der Kopie des gesamten Dataframes resultiert.

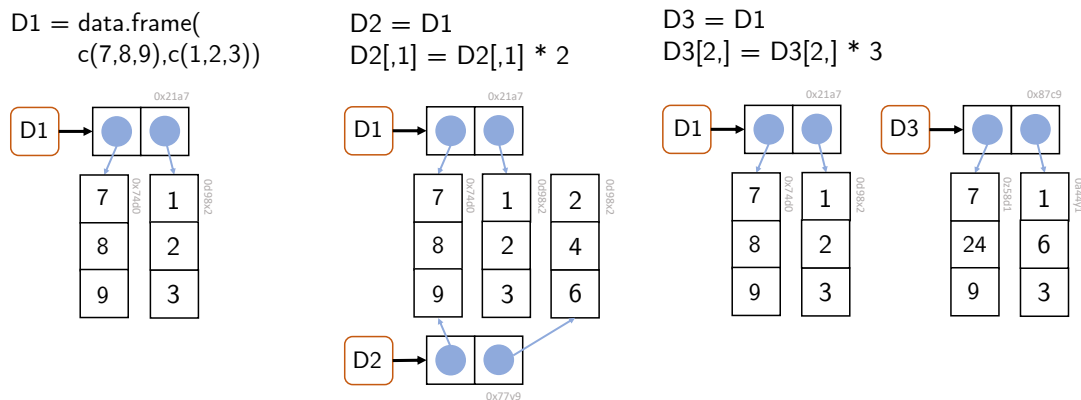


Abbildung 7.4 Effekte von Spalten- und Zeilenmodifikationen bei Dataframes. Die Modifikation einer Spalte führt zur Kopie der entsprechenden Spalte, die Modifikation einer Zeile zur Kopie des gesamten Dataframes.

7.3 Tibbles

In Ergänzung zu den **R**-Dataframes haben in den letzten zehn Jahren die *Tibbles* (Diminutiv für *tables*) als Teil des **tidyverse**-Pakets Verbreitung gefunden. Tibbles sind eine modernere, technisch verbesserte Form von Dataframes und zeigen, wie **R** für datenwissenschaftliche Aufgaben über die Anwendung von klassischen linearen Modellen hinaus erweitert wurde. Zur Nutzung von Tibbles muss das **tidyverse**-Paket installiert und der Tibble-Code geladen sein:

```
install.packages("tidyverse") # Paket einmalig installieren
library(tibble)               # Paket für diese Sitzung laden
```

Im Großen und Ganzen entsprechen Tibbles den Dataframes. Wie bei Dataframes handelt es sich bei Tibbles um Listen von Vektorreferenzen, wobei auch hier die Vektoren die gleiche Länge haben und die benannten Spalten einer Tabelle darstellen. Tibbles können mit der Funktion `tibble()` erzeugt werden. Die Anzeige im **R**-Terminal ist im Vergleich zu Dataframes optimiert, wie folgendes Beispiel zeigt:

```
library(tibble)
T = tibble(x = c("a", "c", "d"), y = c(1,2,3), z = c(TRUE,FALSE,FALSE))
print(T) # Paket
# Erzeugung
# Ausgabe
```

```
# A tibble: 3 x 3
  x         y z
<chr> <dbl> <lgl>
1 a         1 TRUE
2 c         2 FALSE
3 d         3 FALSE
```

```
attributes(T) # Attribute
```

```
$names
[1] "x" "y" "z"
```

```
$row.names
[1] 1 2 3

$class
[1] "tbl_df"      "tbl"        "data.frame"
```

7.3.1 Tibbles vs. Dataframes

Tibbles wurden 2016 im Rahmen des **tidyverse**-Pakets eingeführt, um wahrgenommene Schwächen der **R**-Dataframes auszugleichen. Dazu gehörte unter anderem, dass die Funktion `data.frame()` character-Vektoren automatisiert in den **R**-Datentyp **factor** umwandelte. Dies ist seit **R**-Version 4.0.0 von 2020 allerdings nicht mehr der Fall, wie folgendes Beispiel zeigt:

```
D = data.frame(x = 1:3, y = c("a", "b", "c")) # Zeichenvektor y
str(D)                                         # ... bleibt Zeichenvektor
```

```
'data.frame':  3 obs. of  2 variables:
 $ x: int  1 2 3
 $ y: chr  "a" "b" "c"
```

Per Default wandelt `data.frame()` weiterhin nicht zulässige Spaltennamen wie eine Zahl (1) in legale Spaltennamen (X1) um. Möchte man dies nicht, so kann man dieses Verhalten allerdings durch Setzen des Arguments `check.names` abstellen:

```
names(data.frame(`1` = 1)) # Ungültige Namen anpassen
```

```
[1] "X1"
```

```
names(data.frame(`1` = 1, check.names = FALSE)) # Namen unverändert lassen
```

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

Wie in Kapitel 5 gesehen, hat **R** die Tendenz, bei Vektoren mit unpassenden Längen selbst Daten durch Recycling zu erzeugen, was leicht zu Fehlern führen kann. So recyclet die Funktion `data.frame()` bei der Erzeugung eines Dataframes zum Beispiel Vektoren, wenn einer davon ein ganzzahliges Vielfaches des anderen ist:

```
D = data.frame(x = 1:2, y = 3:6) # length(y) = 2 * length(x)
print(D)                         # Ausgabe
```

```
  x y
1 1 3
2 1 4
3 2 5
4 2 6
```

`tibble()` dagegen ist hier genauer und gibt im obigen Fall einen Fehler aus:

```
T = tibble(x = 1:2, y = 3:6) # tibble gibt hier einen Fehler aus
```

```
Error in `tibble()`:  
! Tibble columns must have compatible sizes.  
* Size 2: Existing data.  
* Size 4: Column `y`.  
i Only values of size one are recycled.
```

Vektoren der Länge 1 werden allerdings auch von `tibble()` recyclet:

```
T = tibble(x = 1:3, y = 3) # Vektoren der Länge 1  
print(T) # Ausgabe
```

```
# A tibble: 3 x 2  
      x     y  
  <int> <dbl>  
1     1     3  
2     2     3  
3     3     3
```

Dataframes sind manchmal inkonsistent, was die von ihnen ausgegebenen Datentypen angeht. So ergibt, wie das untere Beispiel zeigt, die Indizierung einer Dataframespalte mit ihrem Namen etwa einen Dataframe, die Indizierung der gleichen Spalte mit ihrem Namen in Matrixindizierung allerdings einen Vektor:

```
D = data.frame(x = 1:3, y = 4:6) # Dataframe  
str(D["x"]) # Namensindizierung ergibt einen Dataframe
```

```
'data.frame': 3 obs. of 1 variable:  
 $ x: int 1 2 3
```

```
str(D[, "x"]) # Matrixindizierung ergibt einen Vektor
```

```
int [1:3] 1 2 3
```

Tibbles sind in dieser Hinsicht konsistenter:

```
T = tibble(x = 1:3, y = 4:6) # Tibble  
str(T["x"]) # Indizierung mit Namen ergibt tibble
```

```
tibble [3 x 1] (S3: tbl_df/tbl/data.frame)  
 $ x: int [1:3] 1 2 3
```

```
str(T[, "x"]) # Matrixindizierung mit Namen ergibt tibble
```

```
tibble [3 x 1] (S3: tbl_df/tbl/data.frame)  
 $ x: int [1:3] 1 2 3
```

Ein zusätzliches Feature von Tibbles gegenüber Dataframes ist, dass die Funktion `tibble()` es erlaubt, Spaltennamen schon bei Erzeugung eines Tibbles zu referenzieren, was bei Dataframes nicht möglich ist.

```
T = tibble(x = 1, y = 2*x)      # die Spalte x wird definiert und benutzt
D = data.frame(x = 1, y = 2*x)  # data.frame erlaubt dies nicht
```

```
Error:
! object 'x' not found
```

Zusammenfassend lässt sich festhalten, dass Tibbles in mancher Hinsicht weniger flexibel und damit präziser als Dataframes funktionieren. Generell verhalten sich Tibbles in **tidyverse**-Kontexten konsistenter als Dataframes. Allerdings sind die Unterschiede zwischen Dataframes und Tibbles wie oben gesehen eher subtil, sodass ein effizienter Umgang mit Dataframes auch einen effizienten Umgang mit Tibbles ermöglicht. Da nicht alle **R**-Projekte die Pakete des **tidyverse** nutzen, werden wohl auch in Zukunft Dataframes die Standarddatenstruktur für tabellarische Daten in **R** bleiben.

8 Datenmanagement

In diesem Abschnitt wollen wir einige allgemeine Begrifflichkeiten zum Umgang mit Forschungsdaten einführen, gängige Formate zur längerfristigen Speicherung von Daten diskutieren und uns schließlich mit dem praktischen Einlesen und Speichern von Daten im Rahmen des **R**-Verzeichnismanagements beschäftigen.

8.1 Das FAIR-Datenideal

Das FAIR-Datenideal ist ein Prinzip zum Management von Forschungsdaten, das der weiten Verbreitung, Nützlichkeit und Transparenz von Forschungsdaten im Digitalzeitalter Rechnung trägt. Unter *Forschungsdaten* werden dabei im Allgemeinen “elektronisch repräsentierte analoge oder digitale Daten, die im Zuge wissenschaftlicher Vorhaben entstehen oder genutzt werden, z. B. durch Beobachtungen, Experimente, Simulationsrechnungen, Erhebungen, Befragungen, Quellenforschungen, Aufzeichnungen von Audio- und Videosequenzen, Digitalisierung von Objekten, Auswertungen” (Rat für Informationsinfrastrukturen (2017)) verstanden. Konkret handelt es sich dabei etwa um Zahlenarrays oder Characterarrays, aber auch um Computersoftware wie **R**-Skripte, digitale Werkzeuge, Workflows und Analysepipelines.

Neben diesen manchmal auch als *Primärdaten* bezeichneten Datenformen sind für das FAIR-Datenideal sogenannte *Metadaten* von Bedeutung. Dabei handelt es sich um Daten, die Informationen über andere Daten repräsentieren. Man unterscheidet dabei beispielsweise *deskriptive*, *strukturelle* und *administrative Metadaten* (Riley (2017), Ulrich et al. (2022)).

- *Deskriptive Metadaten* dienen dem Auffinden und der Identifikation einer Datenquelle. Beispiele für deskriptive Metadaten sind etwa Titel einer wissenschaftlichen Publikation oder ihr Digital Object Identifier (DOI, Rosenblatt (1997)). DOIs sind alphanumerische Kennungen, die digitale Objekte eindeutig identifizieren und als persistente Links zu Objekten im Internet funktionieren.
- *Strukturelle Metadaten* sind Metadaten über Datencontainer und repräsentieren den strukturellen Aufbau einer Datenquelle. Beispiele sind die Ordnung der Seiten eines Buches oder die For-Schleifen-Enkodierung dreidimensionaler Datenobjekte.
- *Administrative Metadaten* schließlich sind Daten, die das Management einer Datenquelle erleichtern. Beispiele für administrative Metadaten sind Provenienz, das Dateiformat, die Zugangsrechte oder andere technische Informationen zu einer Datenquelle.

Das FAIR-Ideal zum Forschungsdatenmanagement hat seinen Ursprung in Diskussionen im Rahmen der FORCE11-Konferenzserie und wurde 2016 durch Wilkinson et al. (2016) akademisch kommuniziert. Es besagt im Kern, dass Forschungsdaten *Findable*, *Accessible*, *Interoperable* und *Reusable* für Menschen und Maschinen aufbereitet und vorgehalten

werden sollen. Speziell formuliert das FAIR-Datenideal folgende Ansprüche an das Management von Forschungs(meta)data:

Findability (Auffindbarkeit)

- F1. (Meta)Daten haben einen persistenten global einzigartigen Identifikator.
- F2. Daten werden mit Metadaten angereichert.
- F3. Metadaten sind zweifelsfrei einem Datensatz zuzuordnen.
- F4. (Meta)Daten sind in einer durchsuchbaren Ressource indexiert.

Accessibility (Zugänglichkeit)

- A1. (Meta)Daten sind mit standardisierten Protokollen abrufbar.
- A1.1. Das genutzte Protokoll ist offen, kostenlos und nutzbar.
- A1.2. Das Protokoll ermöglicht Authentifizierung und Rechtevergabe.
- A2. Metadaten bleiben zugänglich, auch wenn Daten nicht mehr vorliegen.

Interoperability (Interoperabilität)

- I1. (Meta)Daten nutzen eine formale, zugängliche, gemeinsam genutzte und breit anwendbare Sprache zur Wissensrepräsentation.
- I2. (Meta)Daten nutzen Vokabularen, die den FAIR-Prinzipien folgen.
- I3. (Meta)Daten enthalten qualifizierte Referenzen auf andere (Meta)Daten.

Reusability (Wiederverwendbarkeit)

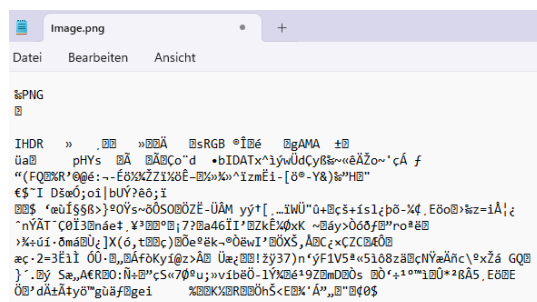
- R1. (Meta)Daten haben eine Vielzahl genauer und relevanter Attribute.
- R1.1. (Meta)Daten enthalten eine eindeutige Nutzungslizenz.
- R1.2. (Meta)Daten enthalten detaillierte Provenienz-Informationen.
- R1.3. (Meta)Daten genügen den Standards der jeweiligen Fachcommunity.

Der Anspruch an ein faires Forschungsdatenmanagement ist mittlerweile recht verbreitet, wobei die FAIR-Prinzipien mit mehr oder weniger bürokratischem Aufwand umgesetzt werden können. Es bleibt aber festzuhalten, dass zurzeit die FAIR-Prinzipien im Wesentlichen ein anzustrebendes Datenmanagementideal sind und längst nicht alle Forschungsdaten FAIR verfügbar sind. Tatsächlich ist der Umgang mit digitalen Forschungsdaten auch weiterhin noch sehr unstrukturiert und gerade das deutsche Universitätssystem begreift das digitale Datenmanagement nur sehr langsam als eine Kernaufgabe. In größerem, wenn auch projektbasiertem und damit nicht nachhaltigem Rahmen versucht die Initiative zur [Nationalen Forschungsdateninfrastruktur \(NFDI\)](#), das deutsche digitale Forschungsdatenmanagement zu verbessern. Leider bleibt weiterhin festzuhalten, dass viele durch öffentliche Mittel finanzierte Wissenschaftler:innen von ihnen erhobene Forschungsdaten weder systematisch aufbereiten noch der Öffentlichkeit zur Verfügung stellen wollen. Hier ist also weiterhin auf einen Kulturwandel zu mehr wissenschaftlicher Transparenz zu hoffen und offene, öffentlich finanzierte Forschung (Open Research, vgl. z. B. Toelch & Ostwald (2018), Miedema (2022), Bertram et al. (2023)) bleibt ein anzustrebendes Ideal.

8.2 Dateiformate

Allgemein definieren Dateiformate Syntax und Semantik von Daten innerhalb einer Datei. Dateiformate sind dabei bijektive Abbildungen von Information auf den binären Langzeitspeicher. Allgemein unterscheidet man *Datendateiformate* gegenüber *Softwaredateiformaten*, *textuelle* gegenüber *binären Dateiformaten* und *offene* gegenüber *proprietären*, d. h. urheberrechtlich geschützten, Dateiformaten. Bei binären Datendateiformaten ist das Einlesen von Daten, ihre Inspektion und ihre Manipulation generell nur mit spezieller Software möglich. Beispiele für binäre Dateiformate sind *.docx*, *.pdf*, *.xlsx*, *.jpg* und *.mp4*. Binäre Dateiformate sind dabei oft proprietär. Früher wurden binäre Dateiformate aufgrund ihres geringeren Speicherbedarfs auch für Forschungsdaten bevorzugt eingesetzt. Heutzutage werden dagegen textuelle Datendateiformate wie *.txt*, *.csv*, *.tsv* und *.json* favorisiert. Bei diesen Formaten ist das Einlesen der gespeicherten Daten, ihre Inspektion und ihre Manipulation mit einfachen allgemeinen Editoren wie beispielsweise VS Code problemlos möglich und die Dateiformate sind generell nicht proprietär. Abbildung 8.1 zeigt die Darstellung eines binären Datenformats und eines textuellen Datenformats im Windows-Editor. Generell bieten sich im Forschungskontext und im Sinne der FAIR-Prinzipien also offene, textuelle Datendateiformate an. Exemplarisch wollen wir hier als textuelle Dateiformate das *.csv*-, das *.tsv*- sowie das *.json*-Dateiformat genauer betrachten.

Binäres Datenformat



Textuelles Datenformat

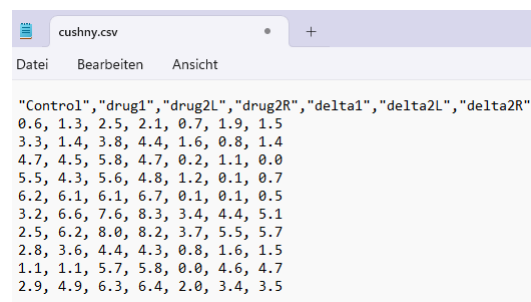


Abbildung 8.1 Binäre und textuelle Dateiformate in Editordarstellung.

8.2.1 .csv- und .tsv-Dateien

.csv- (comma-separated values) und .tsv-Dateien (tab-separated values) sind wichtige Dateiformate zur Speicherung einfacher, tabellarisch strukturierter Daten. Allgemein repräsentieren .csv und .tsv zeilenweise miteinander verknüpfte Datensätze. Dabei werden Datenfelder innerhalb einer Zeile durch Kommata (.csv) oder Tabulatorzeichen (.tsv) und einzelne Datensätze durch Zeilenumbrüche voneinander abgegrenzt. Der Datensatz in der ersten Zeile einer .csv- oder .tsv-Datei enthält typischerweise die Definition der Spaltennamen und wird als Kopfdatensatz (Header) bezeichnet. Ein typisches Beispiel einer .csv-Datei ist in Abbildung 8.1 (rechts) dargestellt. Dabei repräsentieren die Zeilen experimentelle Einheiten (hier Versuchspersonen) und die Spalten verschiedene Variablen, für die jeweils ein Wert bei der Versuchsperson bestimmt wurde.

Im Kontext der Datenanalyse unterscheidet man bei .csv- und .tsv-Dateien eine *Wide-Format*- und eine *Long-Format*-Datenrepräsentation. Bei der Wide-Format- Datenrepräsentation werden alle Variablen einer experimentellen Einheit wie in obigem Beispiel in einer Zeile repräsentiert, vgl. Tabelle 8.1. Allerdings ist es für manche Datenanalysen hilfreicher, die Daten einer experimentellen Einheit über Zeilen zu verteilen. Dies führt auf das sogenannte Long Format, wie in Tabelle 8.2 gezeigt. Generell ist das Wide Format für ein erstes intuitives Verständnis eines Datensatzes oft hilfreicher, wohingegen das Long Format Daten im Rahmen modellbasierter Analysen meist konsistenter mit der Modellarchitektur repräsentiert.

Tabelle 8.1 Wide Format (EE: Experimentelle Einheit, V1,V2,V3: Variablen)

EE	V1	V2	V3
1	10.1	67.5	4
2	12.9	51.2	2
3	20.4	70.8	5
4	17.5	60.3	7

Tabelle 8.2 Long Format (EE: Experimentelle Einheit, V1,V2,V3: Variablen)

Einheit	Variable	Messwert
1	V1	10.1
1	V2	67.5
1	V3	4
2	V1	12.9
2	V2	51.2
2	V3	2
3	V1	20.4
3	V2	70.8
3	V3	5
4	V1	17.5
4	V2	60.3
4	V3	7

8.2.2 .json-Dateien

.json (JavaScript Object Notation) ist ein textuelles Datenformat, das zum Speichern strukturierter Daten in Key-Value-Form verwendet wird. Es weist Ähnlichkeiten mit R-Listen auf, insbesondere mit benannten Listenelementen, und eignet sich daher gut als Format für die Speicherung von Metadaten.

Eine .json-Datei besteht aus Objekten, die in geschweiften Klammern { } notiert werden und eine durch Kommata getrennte Liste von Eigenschaften enthalten. Jede Eigenschaft ist ein Key-Value-Paar, wobei der Key immer als String in Anführungszeichen (" ") geschrieben wird. Der Value kann wiederum ein Objekt, ein Array, ein String, ein Boolean oder eine Zahl sein. Abbildung 8.2 zeigt den Inhalt einer .json-Datei zur Speicherung von Studierendeninformationen.

```
{
  "Vorname" : "Maxi",
  "Nachname" : "Musterfrau",
  "Matrikelnummer" : 12345,
  "Fachsemester" : 2,
  "Studiengang" : "BSc Psychologie",
  "Module" :
  {
    "Deskriptive Statistik" : { "Abgeschlossen" : TRUE, "Note" : 1.0 },
    "Inferenzstatistik" : { "Abgeschlossen" : FALSE, "Note" : NA }
  }
}
```

Abbildung 8.2 .json-Dateiformat.

8.3 Verzeichnismangement

Da Verzeichnissysteme von Computern üblicherweise in Worten (z. B. `C:\Users`) statt numerisch dargestellt sind, ist eine Grundlage für das automatisierte Einlesen von langfristig gespeicherten Daten in den Arbeitsspeicher und umgekehrt das Speichern von Daten aus dem Arbeitsspeicher im langfristigen Speicher das Arbeiten mit sogenannten *Strings* (Zeichenketten). Strings werden in **R** mit Anführungszeichen oder Hochkommata erzeugt:

```
# Anführungszeichen begrenzen Zeichenketten
c("Dies ist ein character vector")
```

```
[1] "Dies ist ein character vector"
```

```
# Hochkommata erlauben Anführungszeichen in der Zeichenkette
c('Dies ist ein "string"')
```

```
[1] "Dies ist ein \"string\""
```

Die Funktion `paste()` konvertiert Vektoren in character und fügt sie elementweise zusammen.

```
# Konvertierung und Konkatenation einelementiger double-Vektoren
paste(1,2)
```

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

```
# Einelementige Zeichenvektoren zusammenfügen
paste("Dies ist", "ein String")
```

```
[1] "Dies ist ein String"
```

Die Funktion `paste()` hat darüber hinaus eine Reihe weiterer Funktionalitäten, die manchmal hilfreich sein können:

```
# Wiederholung von Vektorelementen, elementweise Verknüpfung
paste(c("Rote", "Gelbe"), "Blume")
```

```
[1] "Rote Blume" "Gelbe Blume"
```

```
# Trennzeichen festlegen
paste(c("Rote", "Gelbe"), "Blume", sep = "-")
```

```
[1] "Rote-Blume" "Gelbe-Blume"
```

```
# Mit dem gewählten Trennzeichen zusammenfügen
paste(c("Rote", "Gelbe"), "Blume", collapse = ", ")
```

```
[1] "Rote Blume, Gelbe Blume"
```

Die Funktion `toString()` ist eine `paste()`-Variante für numerische Vektoren:

```
# double-Vektor in eine formatierte Zeichenkette umwandeln
toString(1:10)
```

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

```
# Ausgabe auf width Zeichen begrenzen
toString(1:10, width = 10)
```

```
[1] "1, 2, ...."
```

Um nun die Daten einer Datei im Langzeitspeicher in den Arbeitsspeicher zu laden, benötigt man ihre Adresse in der Verzeichnisstruktur des Computers. Abbildung 8.3 zeigt einen Ausschnitt der Verzeichnisstruktur eines typischen Windows-Rechners.

Die Adressen von Dateien in der Verzeichnisstruktur heißen *Dateipfade* (*paths*). Ein Pfad besteht aus einer durch Schrägstriche getrennten Liste von Verzeichnisnamen. Dabei werden unter Windows traditionell linksgeneigte Schrägstriche (`\`, *backslash*) und unter Mac und Linux rechtsgeneigte Schrägstriche (`/`, *slash*) genutzt. Unter Windows findet sich zusätzlich in der Regel an oberster Stelle der Verzeichnisstruktur der Festplattenname (z. B. `C:\` oder `D:\`), unter Mac der Benutzername (`/Users/`) und unter Linux das Heimverzeichnis (`/home/`). Die folgenden Beispiele sind Windows-basiert.

```
# Pfad, der auf einem Verzeichnisnamen endet
C:\Users\dirko\Nextcloud\Home\pdwp-online\dirk-ostwald.github.io\_data

# Pfad, der auf einem Dateinamen endet
C:\Users\dirko\Nextcloud\Home\pdwp-online\dirk-ostwald.github.io\_data\cushny.csv
```

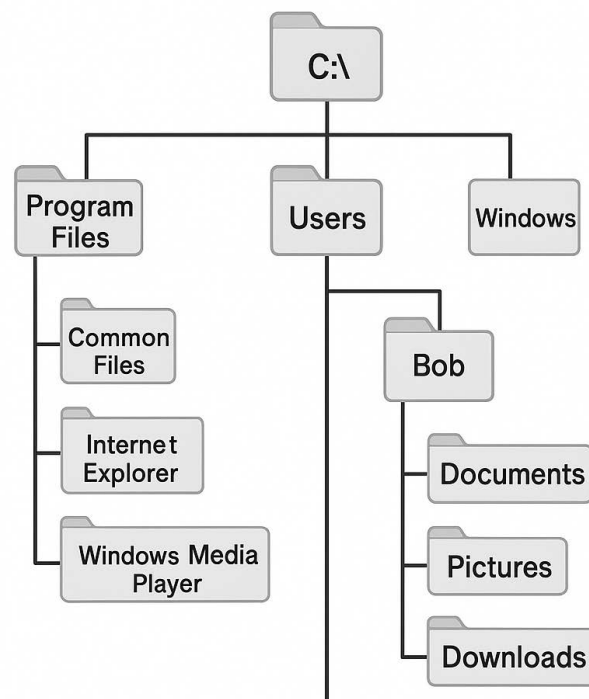


Abbildung 8.3 Windows-Verzeichnisstruktur

Absolute Dateipfade geben die Adresse eines Verzeichnisses oder einer Datei in der Gesamtverzeichnisstruktur der Festplatte und sind wenig anfällig für Dateiverwechslungen. *Relative Dateipfade* beziehen sich dagegen auf einen Speicherort in Relation zum aktuellen Verzeichnis, wobei `.` und `..` das aktuelle und das übergeordnete Verzeichnis bezeichnen. Bei relativen Dateipfaden kann es durchaus zu Dateiverwechslungen kommen. Generell wird also die Verwendung adaptiv generierter absoluter Pfade empfohlen.

```
# Dateiname in absoluter Pfadform
fname = "D:\Lehre\Daten\cushny.csv"
```

R besitzt ein sogenanntes *working directory*, aus dem per Default Dateien gelesen werden. Bei geöffnetem **R**-Terminal gibt `getwd()` das working directory an.

```
getwd()
```

```
[1] "C:/Home/Home/pdwp-online/dirk-ostwald.github.io/_part_2"
```

`setwd()` ändert das working directory. Dabei ist zu beachten, dass **R** nur mit forward slashes `/` arbeitet. Die manuelle Spezifikation von Windows-Pfaden benötigt deshalb doppelte backward slashes `\\`:

```
setwd("C:\\Users\\dirko\\Nextcloud\\Home\\pdwp-online\\dirk-ostwald.github.io\\_data")
getwd()
```

Mithilfe der Funktion `file.path()` können betriebssystemkonforme Verzeichnis- und Dateipfade erzeugt werden:

```
file.path("D:", "Nextcloud", "Home", "Teaching")
```

Die Funktion `dirname()` gibt das Verzeichnis an, das ein Verzeichnis oder eine Datei enthält:

```
getwd()
```

```
[1] "C:/Home/Home/pdwp-online/dirk-ostwald.github.io/_part_2"
```

```
dirname(getwd())
```

```
[1] "C:/Home/Home/pdwp-online/dirk-ostwald.github.io"
```

Die Funktion `basename()` gibt die unterste Ebene eines Datei- oder Verzeichnispfades an:

```
getwd()
```

```
[1] "C:/Home/Home/pdwp-online/dirk-ostwald.github.io/_part_2"
```

```
basename(getwd())
```

```
[1] "_part_2"
```

8.4 Datenimport und Datenexport

Zum Einlesen von einfach strukturierten .csv-Dateien bietet sich die **R**-Funktion `read.csv()` an. Diese Funktion liest eine Datei ein und speichert ihre Inhalte in einem Dataframe.

```
D = read.csv("_data/cushny.csv")
print(D)
```

	Control	drug1	drug2L	drug2R	delta1	delta2L	delta2R
1	0.6	1.3	2.5	2.1	0.7	1.9	1.5
2	3.3	1.4	3.8	4.4	1.6	0.8	1.4
3	4.7	4.5	5.8	4.7	0.2	1.1	0.0
4	5.5	4.3	5.6	4.8	1.2	0.1	0.7
5	6.2	6.1	6.1	6.7	0.1	0.1	0.5
6	3.2	6.6	7.6	8.3	3.4	4.4	5.1
7	2.5	6.2	8.0	8.2	3.7	5.5	5.7
8	2.8	3.6	4.4	4.3	0.8	1.6	1.5
9	1.1	1.1	5.7	5.8	0.0	4.6	4.7
10	2.9	4.9	6.3	6.4	2.0	3.4	3.5

Folgender **R**-Code definiert den absoluten Pfad der Datei adaptiv in der Verzeichnisstruktur des Computers mithilfe des **R**-Arbeitsverzeichnisses.

```
wdir = getwd() # Arbeitsverzeichnis
ddir = file.path(wdir, "_part_2/_data") # Datenverzeichnispfad
fname = "cushny.csv" # Dateiname
fpath = file.path(ddir, fname) # Dateipfad
D = read.csv(fpath) # Einlesen der Datei
```

Darüber hinaus bietet **R** eine Vielzahl von Möglichkeiten des Datenimports mithilfe spezialisierter Funktionen:

Für .csv-Dateien und andere Textdateien:

- `read.csv()`, `read.csv2()`, `read.delim()`, `read.delim2()` als `read.table()`-Varianten.
- `readLines()` für Low-Level-Textdateiimport.
- `fromJSON()` aus dem Paket `rjson` für .json-Dateien.

Für binäre Dateien:

- `read.xlsx()` und `read.xlsx2()` aus dem Paket `xlsx` für Excel-.xlsx-Dateien.
- `read.spss()` aus dem Paket `foreign` für SPSS-.sav-Dateien.
- `readMat()` aus dem Paket `R.matlab` für MATLAB-.mat-Dateien.

Für Datenbanken:

- SQL-Datenbanken können mithilfe der Pakete `DBI` und `RSQLite` abgefragt werden.

9 Kontrollstrukturen

Per se wird imperativer Programmiercode streng sequentiell Befehl für Befehl ausgeführt. Manchmal möchte man von dieser rein sequentiellen Befehlsreihenfolge allerdings abweichen und die Ausführungsreihenfolge von Befehlen flexibler kontrollieren. Prinzipielle Werkzeuge zu diesem Zweck sind *konditionale Kontrollstrukturen* und *iterative Kontrollstrukturen*. *Konditionale Kontrollstrukturen* steuern die Programmausführung, indem sie festlegen, dass bestimmte Befehle nur dann ausgeführt werden, wenn definierte Bedingungen erfüllt sind. **R** bietet zu diesem Zweck `if()`- und `switch()`-Statements. Oft möchte man bestimmte Codeabschnitte auch in Schleifen häufig wiederholen (*iterieren*). Zu diesem Zweck bietet **R** `for()`-, `while()`- und `repeat()`-Schleifen. In diesem Abschnitt stellen wir die grundlegenden konditionalen und iterativen Kontrollstrukturen in **R** vor und folgen dabei der Darstellung in Cotton (2013), Kapitel 8 und Kapitel 9, sowie Wickham et al. (2019), Kapitel 5.

9.1 Konditionale Kontrollstrukturen

Konditionale Kontrollstrukturen erlauben die bedingungsabhängige Ausführung von Befehlen.

if-Statements

Die grundlegende konditionale Kontrollstruktur ist das if-Statement. Die allgemeine Syntax für ein if-Statement in **R** hat folgende Form:

```
if (Bedingung){  
  TrueActions  
}
```

Dabei gilt, dass, wenn die Variable `Bedingung` den Wert `TRUE` hat, also wahr ist, die mit `TrueActions` bezeichneten Befehle ausgeführt werden. Ist der Wert der Variablen `Bedingung` dagegen `FALSE`, so werden die Befehle `TrueActions` nicht ausgeführt und die Programmausführung wird in der nächsten Zeile nach dem if-Statement fortgesetzt. Beispielsweise wird in folgendem Code der Befehl `print("x ist größer als 0")` ausgeführt, da `x` ja nach vorheriger Zuordnung den Wert 1 hat.

```
x = 1  
if(x > 0){  
  print("x ist größer als 0")  
}
```

[1] "x ist größer als 0"

Das if-Statement kann durch eine **else**-Bedingung erweitert werden, um zu spezifizieren, welche Befehle in dem Fall ausgeführt werden sollen, dass die Variable **Bedingung** den Wert **FALSE** hat.

```
if (Bedingung){
  TrueActions
} else {
  FalseActions
}
```

Hier werden die Befehle **TrueActions** ausgeführt, wenn die Variable **Bedingung** den Wert **TRUE** hat und es werden die Befehle **FalseActions** ausgeführt, wenn die Variable **Bedingung** den Wert **FALSE** hat. Beispielsweise wird in folgendem Code der Befehl `print("y ist nicht größer als 0")` ausgeführt, da die Bedingung `y > 0` nicht zutrifft.

```
y = -1
if(y > 0){
  print("y ist größer als 0")
} else{
  print("y ist nicht größer als 0")
}
```

[1] "y ist nicht größer als 0"

R umfasst die in Tabelle 9.1 aufgeführten Relationsoperatoren zur Prüfung binärer logischer Bedingungen.

Tabelle 9.1 Relationsoperatoren in **R**

Relationsoperator	Bedeutung
==	Gleich
!=	Ungleich
<, >	Kleiner, größer
<=, >=	Kleiner oder gleich, größer oder gleich
&	UND
	ODER

Dabei werden die Relationsoperatoren `<`, `<=`, `>`, `>=` zumeist nur auf numerische Werte angewendet. Die Gleichheitsoperatoren `==`, `!=` können auf beliebige Datenstrukturen angewendet werden, um deren Gleichheit zu überprüfen. Die logischen Verknüpfungen `&` und `|` werden zumeist auf logische Werte angewendet. Die Funktion `xor()` schließlich implementiert das exklusive ODER. Es können beispielsweise folgende Tests auf Gleichheit und Relation durchgeführt werden:

```
# Variablendefinition
x = 1
y = 2

# Test auf Gleichheit
```

```
if(x == y){  
  print("x ist gleich y")  
} else {  
  print("x ist ungleich y")  
}
```

```
[1] "x ist ungleich y"
```

```
# Test auf Ungleichheit  
if(x < y){  
  print("x ist kleiner als y")  
} else {  
  print("x ist größer oder gleich y")  
}
```

```
[1] "x ist kleiner als y"
```

Die if-Statement-Tests können durch logische Bedingungen erweitert werden, wie folgendes Beispiel demonstrieren soll:

```
# Variablendefinition  
x = 2  
y = 2  
  
# logisches UND/ODER  
if(x == y | x < y){  
  print("x ist kleiner oder gleich y")  
} else {  
  print("x ist größer als y")  
}
```

```
[1] "x ist kleiner oder gleich y"
```

```
# logisches UND  
if(x > y & x != y){  
  print("x ist größer als y")  
} else {  
  print("x ist kleiner oder gleich y")  
}
```

```
[1] "x ist kleiner oder gleich y"
```

Bei if-Statements ist es entscheidend, dass die Bedingung einem einzigen logischen Wert entspricht. Folgende Bedingungen sind beispielsweise fehlerhaft:

```
if("x"){  
  print(1)  
}
```

```
Error in `if` ("x") ...`:  
! argument is not interpretable as logical
```

```
if(NA){
  print(1)
}
```

```
Error in `if (NA) ...`:
! missing value where TRUE/FALSE needed
```

```
if(c(T,F)){
  print(1)
}
```

```
Error in `if (c(T, F)) ...`:
! the condition has length > 1
```

switch-Statements

Eine weitere konditionale Kontrollstruktur ist das switch-Statement. Switch-Statements sind dadurch motiviert, dass kombinierte if-else-Statements leicht unübersichtlich werden können, wie folgendes Beispiel zeigt:

```
x = 2
if (x == 1){
  print("Aktion 1")
} else if(x == 2){
  print("Aktion 2")
} else if(x == 3){
  print("Aktion 3")
} else if(x == 4){
  print("Aktion 4")
}
```

```
[1] "Aktion 2"
```

Hat man also einen Fall vorliegen, in dem in Abhängigkeit des Wertes einer Variablen eine von vielen Aktionen ausgeführt werden soll, so kann sich ein switch-Statement zur Verbesserung der Codelesbarkeit anbieten. **R** implementiert switch-Statements in der `switch()`-Funktion, die in Abhängigkeit vom Datentyp ihrer Variablen leicht unterschiedlich funktioniert. Ist die switch-Variable vom Typ Integer, so wird die *i*-te Aktion ausgeführt, wie folgendes Beispiel demonstriert:

<pre>x = 2 switch(x, print("Aktion 1"), print("Aktion 2"), print("Aktion 3"), print("Aktion 4"))</pre>	<pre># switch-Variable # x = 2 # 1. Aktion # 2. Aktion # 3. Aktion # 4. Aktion</pre>
--	--

```
[1] "Aktion 2"
```

Ist die switch-Variable dagegen vom Typ character, so wird die Aktion mit dem entsprechenden Namen ausgeführt:

```
x = "a"
switch(x,
a = print("Aktion a"),
b = print("Aktion b"),
c = print("Aktion c"),
d = print("Aktion d"))
# switch-Variable
# x = "a"
# a Aktion
# b Aktion
# c Aktion
# d Aktion
```

```
[1] "Aktion a"
```

Generell lässt sich zu if- und switch-Statements festhalten, dass sich jedes if-Statement äquivalent als switch-Statement formulieren lässt und andersherum. If-Statements sind dabei in der Anwendung sicherlich häufiger anzutreffen als switch-Statements. Generell macht es für das menschliche Codeverständnis Sinn, if-else-Statements nicht zu komplex zu formulieren. Ist dies allerdings nötig, so können komplexe if-else-Statements durch switch-Statements vereinfacht werden.

9.2 Iterative Kontrollstrukturen

Iterative Kontrollstrukturen dienen dazu, bestimmte Befehle wiederholt auszuführen. Die grundlegende iterative Kontrollstruktur in **R** ist die **for**-Schleife. Diese hat die generelle Form:

```
for (item in vector){
  perform_action
}
```

Dabei wird der Befehl `perform_action` für jedes `item` in `vector` einmal ausgeführt und der Wert von `item` jeweils aktualisiert. Folgendes Beispiel soll dies verdeutlichen:

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

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

Der Vektor besteht hier aus `1:3 = [1,2,3]`, der ausgeführte Befehl ist `print(i)` und die `items` sind die Einträge des Vektors. Allerdings muss der Befehl innerhalb der `for`-Schleife nicht zwangsläufig auf die `items` Bezug nehmen, wie folgendes Beispiel zeigt:

```
for (i in 1:3){
  print('a')
}
```

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

Hier wird dreimal der konstante Wert `a` ausgegeben. Typischerweise wird innerhalb einer `for`-Schleife etwas erzeugt und gespeichert. Dabei sollte generell die entsprechende Speicherstruktur vorallokiert werden, da dann der entsprechende Platz im Arbeitsspeicher schon bereitgestellt ist und nicht während der Ausführung der `for`-Schleife bereitgestellt werden muss. Der Effizienzgewinn dieser Strategie mag in den meisten Fällen vernachlässigbar sein, allerdings erlaubt dieses Vorgehen auch insbesondere, Fehler leichter zu detektieren, wenn es beispielsweise Dimensionserwartungsunterschiede oder fehlende Einträge gibt, die dann schon als `NaN` alloziert sind. Folgendes Beispiel demonstriert das iterative Auffüllen eines Vektors mit Mittelwerten.

```
nsim = 3                                # Anzahl an Simulationen
X_bar = rep(NaN, nsim)                 # Speicherstruktur

# Simulationsiterationen
for(i in 1:nsim){                      # Übliche Iterationsindizes: i, j, k
  X = rnorm(12)                        # Realisierung von 12 Z-Variablen
  X_bar[i] = mean(X)                  # Mittelwert der i-ten Realisierung
  print(X_bar)                        # Anzeige zur Demonstration
}
```

```
[1] -0.3023045      NaN      NaN
[1] -0.3023045 -0.1232600      NaN
[1] -0.3023045 -0.1232600 -0.3111732
```

Um linear entlang eines Vektors mit variablen Einträgen zu iterieren, bietet sich die Funktion `seq_along()` an. Diese erzeugt die linearen Indizes 1, 2, 3, ... bis zur Länge des betreffenden Vektors, wie folgendes Beispiel demonstriert:

```
mu = c(0,5,50)                         # Drei Erwartungswertparameter
X_bar = rep(NaN, nsim)                  # Speicherstruktur

# Simulationsiterationen
for(i in seq_along(mu)){                # Übliche Iterationsindizes: i, j, k
  X = rnorm(12, mu[i], 1)              # 12 normalverteilte Werte: N(mu, 1)
  X_bar[i] = mean(X)                   # Mittelwert der i-ten Realisierung
  print(X_bar)                         # Anzeige zur Demonstration
}
```

```
[1] 0.2002776      NaN      NaN
[1] 0.2002776 5.4843517      NaN
[1] 0.2002776 5.4843517 50.0707885
```

`for`-Schleifen können auch geschachtelt werden. Das heißt, innerhalb von `for`-Schleifen können weitere `for`-Schleifen ausgeführt werden. Dies mag etwas gewöhnungsbedürftig sein. Entscheidend ist dabei, dass man sich klarmacht, dass für *jede* Iteration der äußeren Schleife *alle* Iterationen der inneren Schleife durchlaufen werden. Es ist oft hilfreich, sich die generelle Logik einer geschachtelten Schleifenstruktur zunächst mithilfe von `print()`-Statements klarzumachen, wie folgendes Beispiel demonstriert:

```
n_i = 2                                # Iterationsanzahl der äußeren Schleife
n_j = 3                                # Iterationsanzahl der inneren Schleife
for(i in 1:n_i){                       # Äußere Schleife
  for(j in 1:n_j){                     # Innere Schleife
```

```

    print(c(i,j))          # Anzeigen der Iterationsindizes [i,j]
  }
}

```

```

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

```

Folgendes Beispiel etwa erzeugt eine 4 x 5-Matrix, deren Elemente den jeweiligen Spaltenindizes entsprechen. Dabei bedient die innere **for**-Schleife mit der Iteratorvariable **j** die Spaltenindizes und die äußere **for**-Schleife mit der Iteratorvariable **i** bedient die Zeilenindizes:

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	2	3	4	5
[2,]	1	2	3	4	5
[3,]	1	2	3	4	5
[4,]	1	2	3	4	5

Analog erzeugt folgendes Beispiel eine 4 x 5-Matrix, deren Elemente den jeweiligen Zeilenindizes entsprechen. Dabei bedient die innere **for**-Schleife mit der Iteratorvariable **j** wiederum die Spaltenindizes und die äußere **for**-Schleife mit der Iteratorvariable **i** bedient die Zeilenindizes:

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	1	1	1	1
[2,]	2	2	2	2	2
[3,]	3	3	3	3	3
[4,]	4	4	4	4	4

In folgendem Beispiel schließlich wird mithilfe zweier geschachtelter **for**-Schleifen und eines **if**-Statements eine 4 x 5-Matrix, deren Elemente gleich den Spaltenindizes sind, wenn der jeweilige Spaltenindex größer oder gleich dem Zeilenindex ist, und deren Elemente ansonsten null sind, erzeugt:

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	2	3	4	5
[2,]	0	2	3	4	5
[3,]	0	0	3	4	5
[4,]	0	0	0	4	5

10 Deskriptivstatistiken

Ziel der Auswertung von Deskriptivstatistiken ist es, Datensätze möglichst effizient der menschlichen Betrachtung zugänglich zu machen. Sobald die Anzahl der Datenpunkte eines Datensatzes eine gewisse Anzahl überschreitet, sind reine Zahlendarstellungen für die menschliche Kognition meist eher schwer zu erfassen. Hier setzt die Deskriptivstatistik an und bietet neben grafischen Darstellungen größerer Datensätze auch Maßzahlen an, die im Sinne der Datenreduktion bestimmte charakteristische Aspekte eines Datensatzes wie seinen “durchschnittlichen” Wert oder seine Variabilität beschreiben. In diesem Abschnitt wollen wir mit den univariaten Deskriptivstatistiken erste Verfahren und Maße kennenlernen, die es erlauben, Datensätze, in denen pro Studieneinheit eine Zahl vorliegt, zusammenzufassen. Wir betrachten dabei insbesondere *Häufigkeitsverteilungen* und *Histogramme*, *Verteilungsfunktionen* und *Quantile* sowie *Maße der zentralen Tendenz*, die Aussagen über “durchschnittliche” Datenwerte erlauben und *Maße der Datenvariabilität*, die Aussagen über die Streuung der Datenwerte erlauben.

In Abgrenzung zur probabilistischen Modellierung im Rahmen der Frequentistischen und Bayesianischen Inferenz basiert die deskriptive Statistik nicht auf der Wahrscheinlichkeitstheorie, kann also losgelöst von dieser betrachtet werden. Allerdings ergeben viele Aspekte der Deskriptivstatistik letztlich nur vor dem Hintergrund wahrscheinlichkeitstheoretischer Betrachtungen Sinn und umgekehrt sind viele Begriffe der Wahrscheinlichkeitstheorie durch deskriptivstatistische Konzepte motiviert. Wenn also der vorliegende Abschnitt ohne einen wahrscheinlichkeitstheoretischen Hintergrund auskommt, so erschließt sich seine volle Bedeutung sicherlich erst im Kontext der Begriffe der Wahrscheinlichkeitstheorie.

Wir fokussieren in diesem Abschnitt auf Deskriptivstatistiken zur Beschreibung eines Datensatzes der Form

$$y := (y_1, \dots, y_n) \text{ mit } y_i \in \mathbb{R} \text{ für } i = 1, \dots, n. \quad (10.1)$$

Dabei ist es zunächst unerheblich, ob ein solcher Datensatz in Form eines Spaltenvektors $y \in \mathbb{R}^{n \times 1}$ oder eines Zeilenvektors $y \in \mathbb{R}^{1 \times n}$ vorliegt. Weiterhin fokussieren wir auf die praktische Auswertung der betrachteten Deskriptivstatistiken, geben also viele Beispiele für ihre Berechnung mit **R**.

Anwendungsbeispiel

Als Anwendungsbeispiel betrachten wir durchgängig einen Datensatz zur evidenzbasierten Evaluation von Psychotherapie bei Depression. Dazu nehmen wir an, dass $n = 100$ Patient:innen zufällig auf eine Treatment- und eine Kontrollstudienbedingung aufgeteilt wurden und zu Beginn der jeweiligen Intervention mithilfe des BDI-II-Fragebogens zu ihrer Depressionsschwere befragt wurden. Die nach den Interventionen erhobenen BDI-II-Werte wollen wir in diesem Kapitel nicht betrachten. Wir nehmen also an, dass wir einen Datensatz wie durch folgenden **R**-Code in Tabelle 10.1 dargestellt vorliegen haben.

```
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
knitr::kable(D[,1:2], digits = 2, align = "c")        # Tabellenausgabe
```

Tabelle 10.1 BDI-II-Werte (PRE) von $n = 100$ Patient:innen in einer Treatmentgruppe (T) und einer Kontrollstudiengruppe (C) vor Beginn einer psychologischen Intervention

GROUP	PRE
T	17
T	20
T	16
T	18
T	21
T	17
T	17
T	17
T	18
T	18
T	20
T	17
T	16
T	18
T	16
T	18
T	17
T	14
T	18
T	18
T	20
T	20
T	21
T	19
T	19
T	17
T	20
T	21
T	17
T	17
T	19
T	20
T	22
T	20
T	21
T	20
T	16
T	15
T	15
T	18
T	21
T	17
T	18
T	21
T	17
T	19
T	16
T	17
T	17
T	18
C	22
C	19

Tabelle 10.1 BDI-II-Werte (PRE) von $n = 100$ Patient:innen in einer Treatmentgruppe (T) und einer Kontrollstudiengruppe (C) vor Beginn einer psychologischen Intervention

GROUP	PRE
C	21
C	18
C	19
C	17
C	20
C	19
C	19
C	19
C	17
C	20
C	18
C	20
C	18
C	18
C	15
C	18
C	17
C	19
C	19
C	19
C	20
C	19
C	20
C	19
C	21
C	19
C	19
C	18
C	20
C	16
C	21
C	19
C	20
C	18
C	23
C	18
C	19
C	18
C	21
C	17
C	20
C	21
C	21
C	20
C	18
C	18
C	19
C	19

10.1 Häufigkeitsverteilungen

Wir beginnen mit den Begriffen der *absoluten* und der *relativen Häufigkeitsverteilung*. Für Datensätze mit Werten, in denen der gleiche Wert mehrfach auftritt, können

Häufigkeitsverteilungen einen ersten Eindruck von der Beschaffenheit des Datensatzes liefern.

Definition 10.1 (Absolute und relative Häufigkeitsverteilungen). $y := (y_1, \dots, y_n)$ sei ein Datensatz und $W := \{w_1, \dots, w_k\}$ mit $k \leq n$ sei die Menge der im Datensatz vorkommenden Werte. Dann heißt die Funktion

$$h : W \rightarrow \mathbb{N}, w \mapsto h(w) := \text{Anzahl der } y_i \text{ aus } y \text{ mit } y_i = w \quad (10.2)$$

die *absolute Häufigkeitsverteilung* der Werte von y und die Funktion

$$r : W \rightarrow [0, 1], w \mapsto r(w) := \frac{h(w)}{n} \quad (10.3)$$

heißt die *relative Häufigkeitsverteilung* der Werte von y .

•

Häufigkeitsverteilungen können in **R** durch Kombination der `table()`- und `barplot()`-Funktion ausgewertet und dargestellt werden. Folgender **R**-Code erzeugt zunächst die absolute und die relative Häufigkeitsverteilung der Pre-Interventions-BDI-II-Daten des Beispieldatensatzes.

```
D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                           # PRE-Werte als double
n      = length(y)                                       # Anzahl der Werte (100)
H      = as.data.frame(table(y))                       # Absolute Häufigkeiten
names(H) = c("w", "ah")                                # Spaltenbenennung
H$rh   = H$ah/n                                         # Relative Häufigkeiten
print(H, digits = 1)                                   # Ausgabe
```

```
      w ah  rh
1  14  1 0.01
2  15  3 0.03
3  16  6 0.06
4  17 17 0.17
5  18 21 0.21
6  19 20 0.20
7  20 17 0.17
8  21 12 0.12
9  22  2 0.02
10 23  1 0.01
```

An der Ausgabe des Dataframes `H` können wir ablesen, dass im betrachteten Datensatz 10 verschiedene Werte `w` zwischen 14 und 23 vorkommen, wobei zum Beispiel die Werte 18 und 22 jeweils 21-mal bzw. zweimal auftreten. Sowohl die absolute als auch die relative Häufigkeitsverteilung zeigen, dass Werte um 18 im vorliegenden Datensatz gehäuft auftreten, Extremwerte wie 14 oder 23 dagegen eher selten. Man beachte, dass es sich bei der relativen Häufigkeitsverteilung lediglich um eine skalierte Form der absoluten Häufigkeitsverteilung handelt. Qualitativ, also hinsichtlich des gehäuft oder selteneren Vorkommens bestimmter Werte, unterscheiden sich beide Häufigkeitsverteilungen nicht. Dies wird insbesondere auch deutlich, wenn man wie in Abbildung 10.1 beide Häufigkeitsverteilungen visualisiert. Folgender **R**-Code nutzt zu diesem Zweck die Funktion `barplot()`. Die Höhe des über einem Wert w aufgetragenen Balkens entspricht dabei den

Funktionswerten $h(w)$ bzw. $r(w)$. Die visuelle Einsicht entspricht natürlich der Inspektion des Dataframes H. Beide Häufigkeitsverteilungen zeigen, dass Werte um 18 im vorliegenden Datensatz gehäuft auftreten, wohingegen niedrigere oder höhere Werte wie 14 bzw. 23 dagegen eher selten sind.

```
pdf(                                     # PDF-Datei öffnen
  file   = "./_figures/110-haeufigkeitsverteilungen.pdf", # Dateiname
  width  = 7,                                     # Breite in Zoll
  height = 4)                                     # Höhe in Zoll
par(                                     # Grafikparameter
  mfcol  = c(1,2),                               # Grafikanordnung
  las    = 1,                                     # Achsentextlage
  cex    = 0.7,                                   # Textvergrößerung
  cex.main = 1.3)                                # Titelvergrößerung
h        = H$ah                                  # h(w)-Werte
r        = H$rh                                  # r(w)-Werte
names(h) = H$w                                   # w-Werte als Balkennamen
names(r) = H$w                                   # w-Werte als Balkennamen
barplot(                                       # Balkendiagramm
  h,                                          # absolute Häufigkeiten
  col  = "gray90",                          # Balkenfarbe
  xlab = "w",                               # x-Achsentext
  ylab = "h(w)",                             # y-Achsentext
  ylim = c(0,25),                           # y-Achsgrenzen
  main = "Absolute Häufigkeitsverteilung")   # Titel
barplot(                                       # Balkendiagramm
  r,                                          # relative Häufigkeiten
  col  = "gray90",                          # Balkenfarbe
  xlab = "w",                               # x-Achsentext
  ylab = "r(w)",                             # y-Achsentext
  ylim = c(0,.25),                          # y-Achsgrenzen
  main = "Relative Häufigkeitsverteilung")   # Titel
dev.off()
```

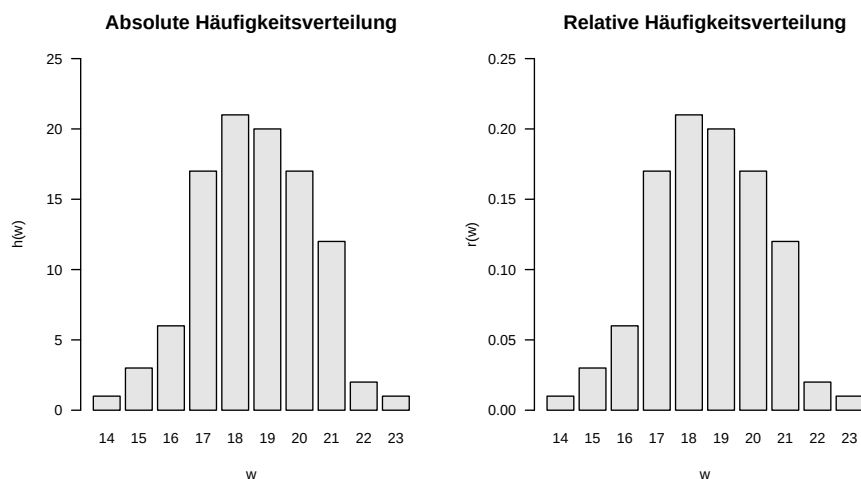


Abbildung 10.1 Absolute und relative Häufigkeitsverteilungen der Pre-Interventions-BDI-II-Werte.

10.2 Histogramme

Wenn in einem Datensatz keine Werte mehrfach vorkommen, nehmen die absoluten und relativen Häufigkeitsverteilungen für jeden dieser Werte den Wert 1 bzw. $1/n$ an und bieten dann keinen Mehrwert über die visuelle Inspektion des Datensatzes hinaus. Die in diesem Abschnitt vorgestellten Histogramme bieten jedoch eine Möglichkeit, auch in diesem Fall visuelle Darstellungen von Häufigkeitsverteilungen zu generieren. Dabei werden verschiedene Werte, die in einem wohldefinierten Sinne “nah beieinander” liegen, zu Klassen zusammengefasst und die absoluten oder relativen Häufigkeiten dieser Klassen bestimmt. Da bei Datensätzen, in denen die gleichen Werte mehrfach vorkommen, diese Werte der Natur nach in der gleichen Klasse liegen, können Histogramme auch in dem in Kapitel 10.1 betrachteten Fall sinnvoll angewendet werden. Histogramme verallgemeinern also die im vorherigen Abschnitt betrachteten Häufigkeitsverteilungen bei Datensätzen mit mehrfach auftretenden Werten.

Wir geben zunächst eine Definition des Histogrammbegriffs. Man beachte, dass in dieser Definition der Vorgang der Klassenbildung, die Bestimmung ihrer absoluten und relativen Häufigkeiten sowie die Visualisierung dieser Häufigkeiten eng verzahnt sind. Dies ist der Konvention geschuldet, dass der Begriff des Histogramms sowohl eine bestimmte Form der Datenanalyse als auch ihre Visualisierung beschreibt.

Definition 10.2 (Histogramm). $y := (y_1, \dots, y_n)$ sei ein Datensatz mit mindestens zwei verschiedenen Werten. Die Klassengrenzen seien $b_0 < b_1 < \dots < b_k$ mit

$$b_0 := \min_{1 \leq i \leq n} y_i \text{ und } b_k := \max_{1 \leq i \leq n} y_i. \quad (10.4)$$

Die Klassen sind definiert als

$$C_1 := \{t \in \mathbb{R} | b_0 \leq t \leq b_1\} \quad (10.5)$$

und für $2 \leq j \leq k$ als

$$C_j := \{t \in \mathbb{R} | b_{j-1} < t \leq b_j\}. \quad (10.6)$$

Wir betrachten zunächst Klassen gleicher Breite $d := b_j - b_{j-1} > 0$. Jeder Datenwert gehöre genau einer Klasse an. Die *absolute Klassenhäufigkeit* und die *relative Klassenhäufigkeit* sind definiert als

$$h_j := |\{i \in \{1, \dots, n\} | y_i \in C_j\}| \text{ und } r_j := \frac{h_j}{n}. \quad (10.7)$$

Ein *Häufigkeitshistogramm* stellt über jeder Klasse C_j ein Rechteck der Breite d und der Höhe h_j beziehungsweise r_j dar. Davon zu unterscheiden ist ein *Dichtehistogramm*. Bei Klassenbreiten $d_j > 0$ hat das Rechteck über C_j die Höhe

$$\frac{r_j}{d_j}. \quad (10.8)$$

Seine Fläche entspricht damit der relativen Klassenhäufigkeit r_j . Die Gesamtfläche aller Rechtecke beträgt eins.

•

Natürlich ist das visuelle Erscheinungsbild eines Histogramms stark davon abhängig, wie genau die Werte eines Datensatzes zu Klassen zusammengefasst werden. Der entscheidende Parameter dafür ist nach Definition 10.2 die Anzahl der Klassen k , die zwischen dem Minimalwert b_0 und dem Maximalwert b_k des Datensatzes gewählt werden. Im Folgenden stellen wir einige konventionelle Werte für die Anzahl der Klassen k vor und berechnen und visualisieren sie am Beispiel des Beispieldatensatzes. Dabei gehen wir durchgängig davon aus, dass die benachbarten Klassen C_j mit $j = 1, \dots, k$ für die Anzahl von Klassen k mit konstanter Breite $d = b_j - b_{j-1}$ gewählt werden.

Zur Berechnung und Visualisierung von Histogrammen nutzen wir die **R**-Funktion `hist()`. Bei dieser werden die Klassengrenzen als das Argument `breaks` festgelegt. Mit `right = TRUE` und `include.lowest = TRUE` werden die rechtsgeschlossenen Klassen aus Definition 10.2 verwendet. Die erste Klasse schließt zusätzlich den Minimalwert ein. Speziell ist `breaks` der **R**-Vektor `c(b_0, b_1, \dots, b_k)` mit Länge $k + 1$.

Mit `prob = TRUE` erzeugt `hist()` ein Dichtehistogramm. Die Fläche jedes Rechtecks entspricht dann der relativen Klassenhäufigkeit; die Gesamtfläche beträgt eins.

Um den Effekt verschiedener Wahlen der Anzahl der Klassen zu veranschaulichen, lesen wir zunächst den Datensatz ein und bestimmen seine minimalen und maximalen Werte b_0 und b_k .

```
# Datensatz
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y = D$PRE # PRE-BDI-II-Daten
n = length(y) # Anzahl der Datenwerte
b_0 = min(y) # Minimum
b_k = max(y) # Maximum
```

```
Minimum b_0 : 14
Maximum b_k : 23
```

Eine erste Möglichkeit der Wahl der Klassen k besteht darin, möglichst genau eine gewünschte konstante Breite d zu erzielen. Hierzu setzt man

$$k := \left\lceil \frac{b_k - b_0}{d} \right\rceil. \quad (10.9)$$

Wie in Definition 10.2 betrachten wir Datensätze mit mindestens zwei verschiedenen Werten. Die Anwendung der Aufrundungsfunktion impliziert, dass die resultierende Klassenbreite höchstens dem gewünschten Wert entspricht, aber auch kleiner sein kann. Folgender **R**-Code verdeutlicht diesen Effekt.

```
# Histogramm mit gewünschter Klassenbreite d = 2
d = 2 # Gewünschte Klassenbreite
k = ceiling((b_k - b_0)/d) # Anzahl der Klassen
b = seq(b_0, b_k, length.out = k + 1) # Klassengrenzen
```

```
Gewünschte Klassenbreite d : 2
Klassenanzahl k : 5
Klassengrenzen b_j : 14 15.8 17.6 19.4 21.2 23
Erzielte Klassenbreite d : 1.8
```

Die Wahl einer gewünschten Klassenbreite von $d = 1.5$ dagegen führt für den vorliegenden Datensatz auf eine resultierende Klassenbreite, die dem Wunsch entspricht.

```
# Histogramm mit gewünschter Klassenbreite d = 1.5
d = 1.5                                # Gewünschte Klassenbreite
k = ceiling((b_k - b_0)/d)             # Anzahl der Klassen
b = seq(b_0, b_k, length.out = k + 1)  # Klassengrenzen
```

```
Gewünschte Klassenbreite d : 1.5
Klassenanzahl k           : 6
Klassengrenzen b_j        : 14 15.5 17 18.5 20 21.5 23
Erzielte Klassenbreite d  : 1.5
```

Eine weitere Möglichkeit ist es, die Anzahl der Klassen abhängig von der Anzahl der Datenpunkte im Datensatz zu machen. Die *Wurzelregel* setzt

$$k := \lceil \sqrt{n} \rceil. \quad (10.10)$$

Folgender **R**-Code implementiert diese Wahl für den vorliegenden Datensatz.

```
# Histogramm mit Wurzelregel
k = ceiling(sqrt(n))                 # Anzahl der Klassen
b = seq(b_0, b_k, length.out = k + 1) # Klassengrenzen
```

```
Anzahl Datenpunkte n      : 100
Klassenanzahl k           : 10
Klassengrenzen b_j        : 14 14.9 15.8 16.7 17.6 18.5 19.4 20.3 21.2 22.1 23
Erzielte Klassenbreite d  : 0.9
```

Ähnlich gelagert wie die Wurzelregel ist die von Sturges (1926) vorgeschlagene Klassenanzahl, welche die Darstellung unter einer impliziten Normalverteilungsannahme optimiert und durch

$$k := \lceil \log_2 n + 1 \rceil \quad (10.11)$$

gegeben ist. Im vorliegenden Fall erhalten wir:

```
# Klassenanzahl nach Sturges (1926)
k = ceiling(log2(n) + 1)             # Anzahl der Klassen
b = seq(b_0, b_k, length.out = k + 1) # Klassengrenzen
```

```
Anzahl Datenpunkte n      : 100
Klassenanzahl k           : 8
Klassengrenzen b_j        : 14 15.125 16.25 17.375 18.5 19.625 20.75 21.875 23
Erzielte Klassenbreite d  : 1.125
```

Eine weitere Möglichkeit ist es, Deskriptivstatistiken des Datensatzes in die Wahl der Klassenanzahl einfließen zu lassen. Als Maß für die Streuung der Daten im Datensatz nutzt die Klassenanzahl nach Scott (1979) die empirische Standardabweichung (cf. Kapitel 10.6), um durch das Histogramm eine Dichteschätzung bei Normalverteilung zu realisieren. Wir

wollen den nichtparametrischen Hintergrund dieses Vorgehens hier nicht vertiefen, sondern halten lediglich fest, dass die Wahl der Klassenanzahl nach Scott (1979) durch Festlegung der Klassenbreite zu

$$d := 3.49S/n^{1/3} \quad (10.12)$$

und die anschließende Wahl der Klassenanzahl durch

$$k := \left\lceil \frac{b_k - b_0}{d} \right\rceil \quad (10.13)$$

gegeben ist, wobei S die Standardabweichung des Datensatzes bezeichnet. Im vorliegenden Fall erhalten wir:

```
# Klassenanzahl nach Scott (1979)
S = sd(y)                                # Empirische Standardabweichung
d = 3.49*S/(n^(1/3))                     # Klassenbreite nach Scott
k = ceiling((b_k - b_0)/d)               # Anzahl der Klassen
b = seq(b_0, b_k, length.out = k + 1)    # Klassengrenzen
```

```
Anzahl Datenpunkte n      : 100
Standardabweichung        : 1.740167
Klassenbreite nach Scott d : 1.308428
Klassenanzahl k           : 7
Klassengrenzen b_j        : 14 15.28571 16.57143 17.85714 19.14286 20.42857 21.71429 23
Erzielte Klassenbreite d  : 1.285714
```

Per Default nutzt die in **R** implementierte `hist()`-Funktion eine Modifikation der von Sturges (1926) vorgeschlagenen Klassenanzahl und bietet eine Reihe von weiteren Möglichkeiten zur Wahl der Klassen. Einen Überblick dazu liefert `?hist`. Folgender **R**-Code visualisiert die absoluten Häufigkeiten der Werte des Beispieldatensatzes anhand der oben spezifizierten Klassen.

```
# Grafikparameter
pdf(                                     # PDF-Datei öffnen
  file   = "./_figures/110-histogramme.pdf", # Dateiname
  width  = 6,                             # Breite in Zoll
  height = 9)                             # Höhe in Zoll
par(                                     # Grafikparameter
  mfcol  = c(3,2),                       # Grafikanordnung
  las    = 1,                             # Achsentextlage
  cex    = .7,                             # Textvergrößerung
  cex.main = .9)                         # Titelvergrößerung
x_min    = 12                             # x-Achsendgrenze
x_max    = 25                             # x-Achsendgrenze
y_min    = 0                             # y-Achsendgrenze
y_max    = 45                             # y-Achsendgrenze
label    = c("", "", "Wurzelregel", "Sturges", "Scott") # Titel

# Histogramme mit vordefinierter Klassenanzahl und Klassenbreite
for(i in 1:5){
  hist(                                     # Histogramm
    y,                                     # Datensatz
    right = TRUE,                         # rechtsgeschlossene Klassen
    include.lowest = TRUE,                 # Einschluss des Minimalwerts
    breaks = bs[[i]],                     # breaks-Argument
    xlim   = c(x_min, x_max),             # x-Achsendgrenzen
```

```

ylim      = c(y_min, y_max),          # y-Achsendgrenzen
ylab      = "Häufigkeit",             # y-Achsentext
xlab      = "BDI",                    # x-Achsentext
main      = sprintf(                  # Titel
  paste(label[i], "k = %.0f, d = %.2f"), # Textmuster
  ks[i],                                     # Klassenanzahl
  ds[i]))                                # Klassenbreite
}

# R-Standardhistogramm
hist(                                     # Histogramm
  y,                                     # Datensatz
  right    = TRUE,                      # rechtsgeschlossene Klassen
  include.lowest = TRUE,                # Einschluss des Minimalwerts
  xlim     = c(x_min, x_max),           # x-Achsendgrenzen
  ylim     = c(y_min, y_max),           # y-Achsendgrenzen
  ylab     = "Häufigkeit",              # y-Achsentext
  xlab     = "",                        # x-Achsentext
  main     = "R Default")               # Titel
dev.off()                               # Grafik schließen

```

10.3 Empirische Verteilungsfunktionen

Neben der Frage, wie häufig in einem Datensatz bestimmte Werte oder Klassen von Werten vorkommen, kann es interessant sein, zu untersuchen, wie häufig Werte vorkommen, die kleiner als ein bestimmter Wert sind. Obwohl die Antwort auf diese Frage natürlich in den absoluten und relativen Häufigkeiten bzw. den entsprechenden Histogrammen implizit ist, ist es manchmal von Vorteil, diese Information direkt verfügbar zu haben. Die in diesem Abschnitt eingeführten Begriffe der *kumulativen absoluten und relativen Häufigkeitsverteilung* sowie der *empirischen Verteilungsfunktion* bezeichnen die entsprechenden Analoga zu den in Kapitel 10.1 und Kapitel 10.2 diskutierten Begriffen. Wir beginnen mit der Definition der *kumulativen absoluten und kumulativen relativen Häufigkeitsverteilungen*.

Definition 10.3 (Kumulative absolute und relative Häufigkeitsverteilungen). $y = (y_1, \dots, y_n)$ sei ein Datensatz und $W := \{w_1, \dots, w_k\}$ mit $k \leq n$ sei die Menge der im Datensatz vorkommenden Zahlenwerte. Weiterhin seien h die absolute und r die relative Häufigkeitsverteilung der Werte von y . Dann heißt die Funktion

$$H : W \rightarrow \mathbb{N}, w \mapsto H(w) := \sum_{w' \leq w} h(w') \quad (10.14)$$

die *kumulative absolute Häufigkeitsverteilung* von y und die Funktion

$$R : W \rightarrow [0, 1], w \mapsto R(w) := \sum_{w' \leq w} r(w') \quad (10.15)$$

die *kumulative relative Häufigkeitsverteilung* der Zahlenwerte von y .

•

Im Sinne der Definition Definition 10.1 gilt für die kumulativen Häufigkeitsverteilungen also

$$H(w) = \text{Anzahl der } y_i \text{ aus } y \text{ mit } y_i \leq w \quad (10.16)$$

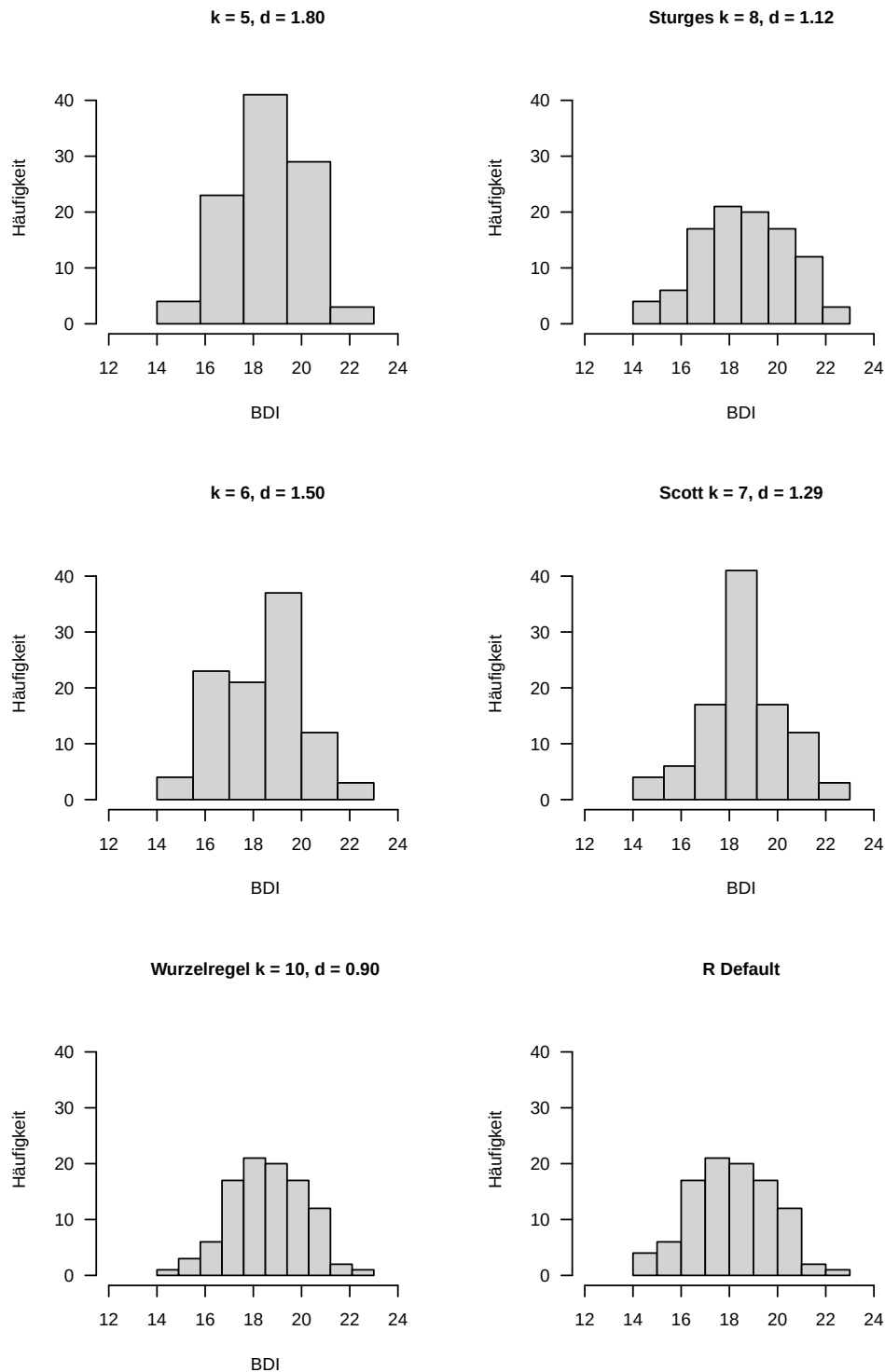


Abbildung 10.2 Histogramme der absoluten Häufigkeiten der Pre-Interventions-BDI-II-Werte. Man beachte, dass der qualitative Eindruck, dass im Datensatz Werte um 18 gehäuft auftreten und niedrigere und höhere Werte seltener sind, über alle Wahlen der Klassenanzahl konstant ist. Das genaue quantitative Erscheinungsbild ist dagegen von der exakten Wahl der Klassenanzahl und Klassenbreite abhängig.

und

$$R(w) = \text{Anzahl der } y_i \text{ aus } y \text{ mit } y_i \leq w \text{ geteilt durch } n. \quad (10.17)$$

In **R** erlaubt die Funktion `cumsum()` die Berechnung kumulativer Summen und ermöglicht damit die Auswertung kumulativer Häufigkeitsverteilungen, wie folgender **R**-Code anhand des Beispieldatensatzes demonstriert.

```
D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                           # PRE-BDI-II-Daten
n      = length(y)                                       # Anzahl der Datenwerte
H      = as.data.frame(table(y))                        # Absolute Häufigkeiten
names(H) = c("w", "h")                                  # Spaltenbenennung
H$H    = cumsum(H$h)                                    # Kumulierte Anzahlen
H$r    = H$h/n                                          # Relative Häufigkeiten
H$R    = cumsum(H$r)                                    # Kumulierte Anteile
print(H, digits = 1)
```

	w	h	H	r	R
1	14	1	1	0.01	0.01
2	15	3	4	0.03	0.04
3	16	6	10	0.06	0.10
4	17	17	27	0.17	0.27
5	18	21	48	0.21	0.48
6	19	20	68	0.20	0.68
7	20	17	85	0.17	0.85
8	21	12	97	0.12	0.97
9	22	2	99	0.02	0.99
10	23	1	100	0.01	1.00

Man beachte dabei insbesondere die Gegenüberstellung von `h` und `H` sowie von `r` und `R`, die die kumulativen Summen in Definition 10.3 verdeutlichen. Folgender **R**-Code ermöglicht die Darstellung der so bestimmten kumulativen Häufigkeitsverteilungen wie in Abbildung 10.3 gezeigt. Offensichtlich sind die absolute und die relative Häufigkeit von Werten kleiner als der kleinste Wert im Datensatz null. Die absolute und relative Häufigkeit von Werten kleiner oder gleich dem größten Wert im Datensatz ist dagegen n bzw. $n/n = 1$.

```
pdf(                                           # PDF-Datei öffnen
  file = "./_figures/110-kumulative-haeufigkeitsverteilungen.pdf", # Dateiname
  width = 10,                                # Breite in Zoll
  height = 5)                                # Höhe in Zoll
Hw = H$H                                     # H(w)-Werte
Rw = H$R                                     # R(w)-Werte
names(Hw) = H$w                              # Balkennamen
names(Rw) = H$w                              # Balkennamen
par(                                          # Grafikparameter
  mfcol = c(1,2),                           # Grafikanordnung
  las = 1,                                  # Achsentextlage
  cex = 1)                                  # Textvergrößerung
barplot(                                     # Balkendiagramm
  Hw,                                       # H(w)-Werte
  col = "gray90",                         # Balkenfarbe
  xlab = "w",                             # x-Achsentext
  ylab = "H(w)",                          # y-Achsentext
  ylim = c(0,110),                       # y-Achsendaten
  las = 1,                               # Achsentextlage)
```

```

main      = "Absolute kumulative Häufigkeit PRE")           # Titel
barplot(  # Balkendiagramm
  Rw,    # R(w)-Werte
  col     = "gray90",  # Balkenfarbe
  xlab    = "w",       # x-Achsentext
  ylab    = "R(w)",    # y-Achsentext
  ylim    = c(0,1),   # y-Achsgrenzen
  las     = 1,         # Achsentextlage
  main    = "Relative kumulative Häufigkeit PRE")          # Titel
dev.off() # Grafik schließen

```

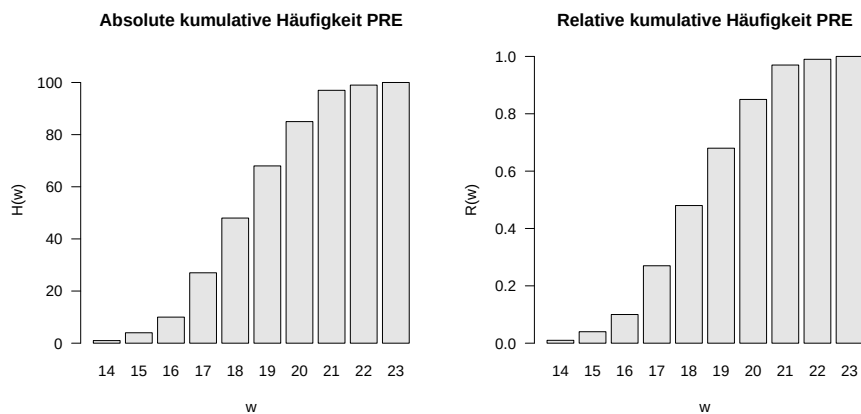


Abbildung 10.3 Absolute und relative kumulative Häufigkeitsverteilungen der Pre-Interventions-BDI-II-Werte.

Das kumulative Analogon zu einem Histogramm ist die *empirische Verteilungsfunktion*. Im Unterschied zu einem Histogramm ist für die Bestimmung einer empirischen Verteilungsfunktion allerdings keine Definition von Klassen erforderlich. An die Stelle der Intervalleinteilung der reellen Zahlen zwischen dem Minimum und dem Maximum eines Datensatzes treten bei der empirischen Verteilungsfunktion einfach die reellen Zahlen, wie folgende Definition verdeutlicht.

Definition 10.4 (Empirische Verteilungsfunktion). $y = (y_1, \dots, y_n)$ sei ein Datensatz. Dann heißt die Funktion

$$F : \mathbb{R} \rightarrow [0, 1], x \mapsto F(x) := \frac{\text{Anzahl der } y_i \text{ aus } y \text{ mit } y_i \leq x}{n} \quad (10.18)$$

die empirische Verteilungsfunktion (EVF) von y .

•

Die empirische Verteilungsfunktion wird manchmal auch die *empirische kumulative Verteilungsfunktion* genannt. Allgemein beschränkt man sich bei der empirischen Verteilungsfunktion wie in Definition 10.4 auf die Angabe der relativen Häufigkeiten. Natürlicherweise kommen in Datensätzen nicht alle reellen Zahlen vor. Dies impliziert, dass die relative Häufigkeit, die reellen Zahlen zugeordnet wird, die zwischen zwei benachbarten Werten des Datensatzes liegen, derjenigen des kleineren der beiden Werte entspricht. Typischerweise sind empirische Verteilungsfunktionen also Treppenfunktionen. In **R** kann die Funktion `ecdf()` zur Evaluation und Visualisierung der empirischen

Verteilungsfunktion eingesetzt werden, wie folgender **R**-Code anhand des Beispieldatensatzes demonstriert und in Abbildung 10.4 visualisiert ist. Man liest aus Abbildung 10.4 zum Beispiel ab, dass die relative Häufigkeit von Werten kleiner als 17.9 etwa 0.3 ist. Die gleiche relative Häufigkeit haben Werte kleiner als 17.8.

```
D      = read.csv("../_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                             # PRE-BDI-II-Daten
evf    = ecdf(y)                                          # Evaluation der EVF
pdf(    # PDF-Datei öffnen
  file  = "../_figures/110-ecdf.pdf",                    # Dateiname
  width = 8,                                              # Breite in Zoll
  height = 5)                                           # Höhe in Zoll
plot(    # EVF darstellen
  evf,    # ecdf-Objekt
  xlab    = "BDI",                                       # x-Achsentext
  ylab    = "Relative Häufigkeit",                      # y-Achsentext
  main    = "Empirische Verteilungsfunktion")           # Titel
dev.off()                                              # Grafik schließen
```

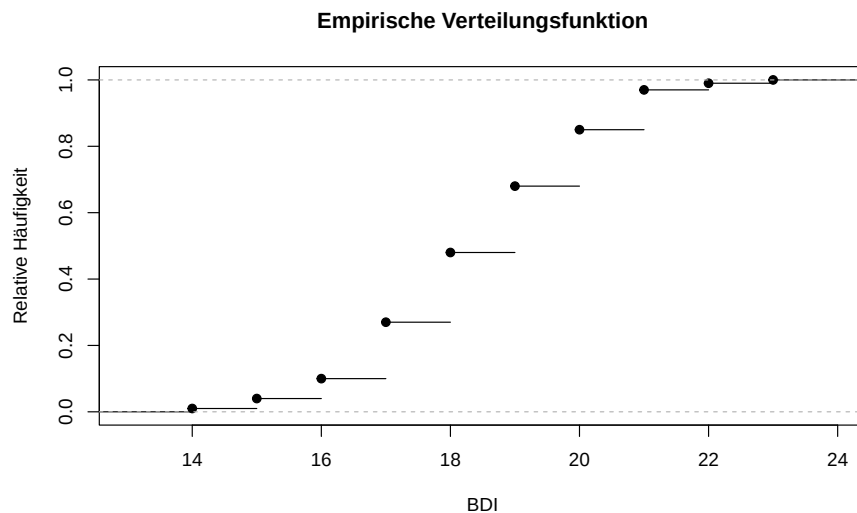


Abbildung 10.4 Empirische Verteilungsfunktion der PRE-Werte.

Die mit `ecdf()` erzeugte Funktion lässt sich auch an einzelnen Stellen auswerten.

```
evf = ecdf(D$PRE) # Empirische Verteilungsfunktion: PRE-Werte
evf(17)           # Anteil der Werte kleiner oder gleich 17
```

```
[1] 0.27
```

```
evf(19)           # Anteil der Werte kleiner oder gleich 19
```

```
[1] 0.68
```

Im Beispieldatensatz sind 27 % der BDI-PRE-Werte kleiner oder gleich 17 und 68 % kleiner oder gleich 19.

10.4 Quantile und Boxplots

Die empirische Verteilungsfunktion gibt eine zumindest grafisch dargestellte Antwort auf die Frage, wie groß der relative Anteil der Werte eines Datensatzes ist, die kleiner als ein gegebener Wert sind. Die in diesem Abschnitt eingeführten Quantile können als Inversion dieser Datensatzinterrogation verstanden werden. Bei Quantilen gibt man einen relativen Anteil $0 \leq p \leq 1$ vor und bestimmt einen Wert, sodass mindestens $p \cdot 100\%$ der Datenwerte kleiner oder gleich diesem Wert und mindestens $(1 - p) \cdot 100\%$ der Datenwerte größer oder gleich diesem Wert sind. Wir nutzen folgende Definition des Begriffs des p -Quantils.

Definition 10.5 (p -Quantil). $y := (y_1, \dots, y_n)$ sei ein Datensatz mit $n \geq 1$ und $y_{(1)} \leq \dots \leq y_{(n)}$ seien seine aufsteigend sortierten Werte. Weiterhin bezeichne $\lfloor \cdot \rfloor$ die Abrundungsfunktion. Dann heißt für $0 \leq p \leq 1$ die Zahl

$$y_p := \begin{cases} y_{(1)} & p = 0, \\ y_{(\lfloor np+1 \rfloor)} & 0 < p < 1, \ np \notin \mathbb{N}, \\ \frac{y_{(np)} + y_{(np+1)}}{2} & 0 < p < 1, \ np \in \mathbb{N}, \\ y_{(n)} & p = 1 \end{cases} \quad (10.19)$$

das p -Quantil des Datensatzes y .

•

Man beachte, dass das p -Quantil y_p nach Definition 10.5 entweder ein Wert des Datensatzes oder ein Wert zwischen zwei Werten des Datensatzes ist. Nach Konstruktion von y_p gilt, dass mindestens $p \cdot 100\%$ aller Werte des Datensatzes kleiner oder gleich y_p und mindestens $(1 - p) \cdot 100\%$ aller Werte größer oder gleich y_p sind. Diese Anteile müssen nicht exakt erreicht werden. Insbesondere können wiederholt auftretende Datenwerte dazu führen, dass die jeweiligen Anteile größer sind. Je nach Wahl von p erhalten die p -Quantile spezielle Bezeichnungen:

- $y_{0.25}, y_{0.50}, y_{0.75}$ heißen in dieser Reihenfolge *unteres Quartil*, *Median* und *oberes Quartil*,
- $y_{j \cdot 0.10}$ für $j = 1, \dots, 9$ heißen *Dezile* und
- $y_{j \cdot 0.01}$ für $j = 1, \dots, 99$ heißen *Perzentile*.

Um den Begriff des p -Quantils zu verdeutlichen, betrachten wir ein an Henze (2018), Kapitel 5 orientiertes Beispiel. Dazu seien zunächst der in untenstehender Tabelle dargestellte Datensatz und seine aufsteigend sortierte Version gegeben.

i	1	2	3	4	5	6	7	8	9	10
y_i	8.5	1.5	75	4.5	6.0	3.0	3.0	2.5	6.0	9.0
$y_{(i)}$	1.5	2.5	3.0	3.0	4.5	6.0	6.0	8.5	9.0	75

Wir wollen zunächst das untere Quartil, also das 0.25-Quantil, dieses Datensatzes bestimmen. Es ist offenbar $n = 10$ und wir haben $p := 0.25$ gewählt. Basierend auf Definition 10.5 finden wir, dass

$$np = 10 \cdot 0.25 = 2.5 \notin \mathbb{N}. \quad (10.20)$$

Also gilt nach Definition 10.5, dass

$$y_{0.25} = y_{(\lfloor 2.5+1 \rfloor)} = y_{(3)} = 3.0. \quad (10.21)$$

Das untere Quartil des betrachteten Datensatzes ist also der Wert 3.0. Die vier Datenwerte 1.5, 2.5, 3.0 und 3.0 sind kleiner oder gleich diesem Wert. Damit sind 40% der Datenwerte kleiner oder gleich dem unteren Quartil. Dieser Anteil übersteigt die geforderten 25%, was mit der Quantildefinition vereinbar ist.

Als Nächstes wollen wir das 0.80-Quantil des Datensatzes bestimmen. Es ist offenbar weiterhin $n = 10$ und wir haben nun $p := 0.80$ gewählt. Wir finden in diesem Fall, dass

$$np = 10 \cdot 0.80 = 8 \in \mathbb{N}. \quad (10.22)$$

Also folgt nach Definition 10.5

$$y_{0.80} = \frac{1}{2} (y_{(8)} + y_{(8+1)}) = \frac{1}{2} (y_{(8)} + y_{(9)}) = \frac{8.5 + 9.0}{2} = 8.75. \quad (10.23)$$

Das 0.80-Quantil des betrachteten Datensatzes ist mit 8.75 also der Wert zwischen den Werten 8.5 und 9 des Datensatzes. In **R** wertet man die so bestimmten Quantile wie folgt aus.

```
y = c(8.5, 1.5, 75, 4.5, 6.0, 3.0, 3.0, 2.5, 6.0, 9.0) # Beispieldaten
n = length(y) # Anzahl Datenwerte
y_s = sort(y) # sortierter Datensatz
p = 0.25 # np ist nicht ganzzahlig
y_p = y_s[floor(np + 1)] # 0.25-Quantil
cat("0.25-Quantil: ", round(y_p, 2)) # Ausgabe
```

0.25-Quantil: 3

```
p = 0.80 # np ist ganzzahlig
y_p = (1/2)*(y_s[np] + y_s[np + 1]) # 0.80-Quantil
cat("0.80-Quantil: ", round(y_p, 2)) # Ausgabe
```

0.80-Quantil: 8.75

Mit `quantile()` stellt **R** eine Funktion bereit, die die Quantilbestimmung eines Datensatzes automatisiert. Die in Definition 10.5 angegebene Quantildefinition entspricht in **R** der Einstellung `type = 2`. Mit `quantile(y, p, type = 2)` lassen sich daher die nach dieser Definition bestimmten Quantile berechnen. Die Einstellung `type` erlaubt auch die Auswahl anderer Quantildefinitionen. Einen Überblick geben Hyndman & Fan (1996) auf theoretischer Ebene und `?quantile` auf Implementationsebene. Wir bestimmen das 0.25-Quantil des Pre-Interventions-BDI-II-Beispieldatensatzes mit `type = 2`.

```
D = read.csv("../_data/110-deskriptive-statistik.csv") # Datensatz laden
y = D$PRE # PRE-BDI-II-Daten
y_p = quantile(y, 0.25, type = 2) # 0.25-Quantil nach Typ 2
```

0.25-Quantil (Type = 2): 17

Ein *Boxplot* visualisiert eine Zusammenfassung eines Datensatzes durch fünf Kennzahlen. Im folgenden Beispiel reichen die Whisker aufgrund der Einstellung `range = 0` bis zum Minimum und Maximum. Die Grenzen der zentralen Box bilden die *untere Tukey-Hinge* und die *obere Tukey-Hinge*. Der Median wird als Strich innerhalb der Box dargestellt.

Zur Bestimmung der Tukey-Hinges wird der Datensatz zunächst aufsteigend sortiert. Bei gerader Anzahl der Datenwerte wird er in zwei gleich große Hälften geteilt. Bei ungerader Anzahl wird der mittlere Datenwert beiden Hälften zugeordnet. Die untere Tukey-Hinge ist der Median der unteren Hälfte, die obere Tukey-Hinge der Median der oberen Hälfte. Diese Konstruktion gehört zur *Fünffahnenzusammenfassung* nach Tukey (1977), Abschnitt 2C, und wird in **R** durch `fivenum()` berechnet.

Die Tukey-Hinges stimmen nicht für jeden Datensatz mit den Quartilen der oben angegebenen Quantildefinition überein. Für den Datensatz

$$y := (1, 2, 3, 4, 5, 6, 7) \quad (10.24)$$

sind die beiden Hälften $(1, 2, 3, 4)$ und $(4, 5, 6, 7)$. Ihre Mediane und damit die Tukey-Hinges betragen

$$h_u := \frac{2+3}{2} = 2.5 \text{ und } h_o := \frac{5+6}{2} = 5.5. \quad (10.25)$$

Die Quartile nach Definition 10.5 betragen dagegen $y_{0.25} = 2$ und $y_{0.75} = 6$. Der folgende **R**-Code vergleicht die Tukey-Hinges und Quartile für dieses separate Beispiel.

```
y_beispiel = 1:7                                # Separates Beispiel
fivenum(y_beispiel)                             # Fünffahnenzusammenfassung
```

```
[1] 1.0 2.5 4.0 5.5 7.0
```

```
quantile(y_beispiel, c(0.25, 0.75), type = 2)    # Quartile nach der Definition
```

```
25% 75%
 2   6
```

Der Interquartilsabstand $d_Q := y_{0.75} - y_{0.25}$ ist ein Maß für die Streuung der Daten. Er ist nicht allgemein mit der Differenz der Tukey-Hinges identisch. Im separaten Beispiel $y = (1, \dots, 7)$ beträgt der Interquartilsabstand $6 - 2 = 4$, während die Differenz der Tukey-Hinges $5.5 - 2.5 = 3$ beträgt.

Folgender **R**-Code erstellt einen Boxplot des Pre-Interventions-BDI-II-Datensatzes.

```
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
pdf(                                                    # PDF-Datei öffnen
  file = "./_figures/110-boxplot.pdf",                # Dateiname
  width = 8,                                           # Breite in Zoll
  height = 5)                                          # Höhe in Zoll
boxplot(                                              # Boxplot
  D$PRE,                                              # Datensatz
  horizontal = T,                                     # Horizontal darstellen
  range = 0,                                          # Antennen: Min./Max.
  xlab = "BDI",                                       # x-Achsentext
  main = "Boxplot")                                   # Titel
dev.off()                                             # Grafik schließen
```

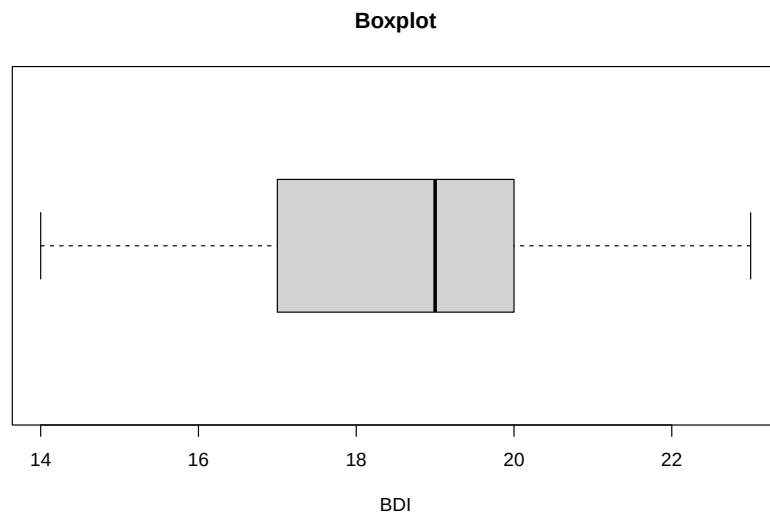


Abbildung 10.5 Boxplot der Pre-Interventions-BDI-Werte des Beispieldatensatzes.

Die fünf im Boxplot dargestellten Kennzahlen erhält man mit `fivenum()`. Für die BDI-PRE-Werte ergeben sich 14, 17, 19, 20 und 23.

```
fivenum(D$PRE)           # Min., untere Hinge, Median, obere Hinge, Max.
```

```
[1] 14 17 19 20 23
```

In [Abbildung 10.5](#) werden $\min y$ und $\max y$ als Whiskerendpunkte dargestellt. Die untere und die obere Tukey-Hinge bilden die Grenzen der zentralen grauen Box. Der Median $y_{0.50}$ wird als Strich innerhalb der Box abgebildet. Es gibt viele Boxplotvariationen, vgl. McGill et al. (1978). Eine genaue Erläuterung der grafisch dargestellten Datensatzeigenschaften ist daher immer nötig. Schließlich sei angemerkt, dass die Darstellung von Datensätzen mithilfe von Boxplots in den letzten zwanzig Jahren etwas aus der Mode gekommen ist und man sie nur noch sehr selten in aktuellen wissenschaftlichen Publikationen findet.

10.5 Maße der zentralen Tendenz

In diesem Abschnitt betrachten wir einige grundlegende Kennzahlen von Datensätzen, die für die deskriptive Statistik zentral sind und Aussagen zu dem “durchschnittlichen” Wert eines Datensatzes machen. Wir beginnen mit der Definition des Mittelwerts.

10.5.1 Mittelwert

Definition 10.6 (Mittelwert). $y = (y_1, \dots, y_n)$ sei ein Datensatz. Dann heißt

$$\bar{y} := \frac{1}{n} \sum_{i=1}^n y_i \quad (10.26)$$

der *Mittelwert* von y .

•

Der Mittelwert eines Datensatzes ist also die Summe der Datenwerte geteilt durch die Anzahl der Datenwerte. Der hier definierte Mittelwert wird auch als *arithmetisches Mittel* oder *Durchschnitt* bezeichnet. Im Kontext der Frequentistischen Inferenz werden wir den Mittelwert vor dem Hintergrund des Modells der Zufallsstichprobe als *Stichprobenmittel* bezeichnen. Insbesondere erlauben die Modelle der Frequentistischen Inferenz auch, die funktionale Form des Mittelwertes tiefer zu begründen, wie wir an gegebener Stelle sehen werden. Mithilfe der Summenfunktion berechnet man den Mittelwert eines Datensatzes in **R** wie folgt.

```
D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                           # PRE-BDI-II-Daten
n      = length(y)                                       # Anzahl der Werte
y_bar  = (1/n)*sum(y)                                    # Mittelwertsberechnung
```

```
Mittelwert der PRE-BDI-II-Daten  18.61
```

Mit der `mean()`-Funktion stellt **R** natürlich auch eine Funktion zur automatisierten Berechnung des Mittelwerts bereit.

```
cat("Mittelwert der PRE-BDI-II-Daten ", mean(y)) # Ausgabe
```

```
Mittelwert der PRE-BDI-II-Daten  18.61
```

In folgendem Theorem halten wir zunächst zwei Eigenschaften des Mittelwerts eines Datensatzes fest, die seine Einordnung als Maß der zentralen Tendenz eines Datensatzes plausibel machen.

Theorem 10.1 (Zentralitätseigenschaften des Mittelwerts). $y = (y_1, \dots, y_n)$ sei ein Datensatz und $\bar{y}$ sei der Mittelwert von y . Dann gelten

(1) Die Summe der Abweichungen vom Mittelwert ist null,

$$\sum_{i=1}^n (y_i - \bar{y}) = 0. \quad (10.27)$$

(2) Die absoluten Summen negativer und nichtnegativer Abweichungen vom Mittelwert sind gleich. Die Indexmengen

$$J := \{i \in \{1, \dots, n\} | y_i - \bar{y} < 0\} \quad (10.28)$$

und

$$K := \{i \in \{1, \dots, n\} | y_i - \bar{y} \geq 0\} \quad (10.29)$$

erfassen die negativen beziehungsweise nichtnegativen Abweichungen. Dann gilt

$$\left| \sum_{i \in J} (y_i - \bar{y}) \right| = \left| \sum_{i \in K} (y_i - \bar{y}) \right|. \quad (10.30)$$

◦

Beweis. (1) Es gilt

$$\begin{aligned}
 \sum_{i=1}^n (y_i - \bar{y}) &= \sum_{i=1}^n y_i - \sum_{i=1}^n \bar{y} \\
 &= \sum_{i=1}^n y_i - n\bar{y} \\
 &= \sum_{i=1}^n y_i - \frac{n}{n} \sum_{i=1}^n y_i \\
 &= \sum_{i=1}^n y_i - \sum_{i=1}^n y_i \\
 &= 0.
 \end{aligned} \tag{10.31}$$

(2) Weil J und K die Datenpunktindizes vollständig und ohne Überschneidung aufteilen, gilt mit Aussage (1)

$$0 = \sum_{i=1}^n (y_i - \bar{y}) = \sum_{i \in J} (y_i - \bar{y}) + \sum_{i \in K} (y_i - \bar{y}). \tag{10.32}$$

Also gilt

$$\sum_{i \in K} (y_i - \bar{y}) = - \sum_{i \in J} (y_i - \bar{y}) \tag{10.33}$$

und damit

$$\left| \sum_{i \in J} (y_i - \bar{y}) \right| = \left| \sum_{i \in K} (y_i - \bar{y}) \right|. \tag{10.34}$$

□

Nach Theorem 10.1 liegt der Mittelwert eines Datensatzes also gerade so, dass sich in Summe die Abweichungen der Datenpunkte von diesem Wert zu null ausgleichen und dass sich positive und negative Abweichungen vom Mittelwert die Waage halten. In **R** kann man Theorem 10.1 am Beispieldatensatz wie folgt verifizieren.

```
# Datensatz
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y = D$PRE # PRE-BDI-II-Daten
s = sum(y - mean(y)) # Summe der Abweichungen
s_1 = sum(y[y < mean(y)] - mean(y)) # Negative Abweichungen
s_2 = sum(y[y >= mean(y)] - mean(y)) # Nichtneg. Abweichungen
```

```
Summe der Abweichungen vom Mittelwert : 0
Summen der negativen und nichtnegativen Abweichungen : -71.28 71.28
```

Theorem 10.2 (Summationseigenschaften des Mittelwerts). *Gegeben seien zwei Datensätze $y = (y_1, \dots, y_n)$ und $z = (z_1, \dots, z_n)$ gleichen Umfangs und es sei durch*

$$y + z := (y_1 + z_1, \dots, y_n + z_n) \tag{10.35}$$

die Summe dieser Datensätze definiert. Weiterhin seien $\bar{y}$ und $\bar{z}$ die Mittelwerte der Datensätze y und z in dieser Reihenfolge. Dann gilt

$$\overline{y + z} = \bar{y} + \bar{z} \tag{10.36}$$

◦

Beweis. Es gilt

$$\begin{aligned}
 \overline{y+z} &:= \frac{1}{n} \sum_{i=1}^n (y_i + z_i) \\
 &= \frac{1}{n} \sum_{i=1}^n y_i + \frac{1}{n} \sum_{i=1}^n z_i \\
 &=: \bar{y} + \bar{z}.
 \end{aligned} \tag{10.37}$$

□

Betrachtet man die Summe zweier Datensätze, so ist es für die Bestimmung ihres Mittelwerts also unerheblich, ob man zunächst die Datensätze aufaddiert und dann den Mittelwert des summierten Datensatzes bestimmt oder ob man für jeden der beiden Datensätze den Mittelwert bestimmt und diese aufaddiert. Wir verifizieren dieses Ergebnis exemplarisch mithilfe der Pre- und Post-Interventions-BDI-II-Daten des Beispieldatensatzes in **R** wie folgt.

```

D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                             # PRE-BDI-II-Daten
y_bar  = mean(y)                                           # Mittelwert PRE
z      = D$POS                                             # POS-BDI-II-Daten
z_bar  = mean(z)                                           # Mittelwert POS
yz_bar_1 = mean(y + z)                                     # Mittelwert von PRE+POS
yz_bar_2 = y_bar + z_bar                                   # Summe der Mittelwerte

```

```

Mittelwert von y + z          : 31.68
Summe der Mittelwerte von y und z : 31.68

```

Theorem 10.3 (Mittelwert bei linear-affiner Transformation). $y = (y_1, \dots, y_n)$ sei ein Datensatz und $\bar{y}$ sei der Mittelwert von y . Weiterhin sei für $a, b \in \mathbb{R}$ mit

$$ay + b := (ay_1 + b, \dots, ay_n + b) \tag{10.38}$$

ein linear-affin transformierter Datensatz von y definiert. Dann gilt, dass

$$\overline{ay + b} = a\bar{y} + b \tag{10.39}$$

◦

Beweis. Es gilt

$$\begin{aligned}
 \overline{ay + b} &:= \frac{1}{n} \sum_{i=1}^n (ay_i + b) \\
 &= \sum_{i=1}^n \left(\frac{1}{n} ay_i + \frac{1}{n} b \right) \\
 &= \left(\sum_{i=1}^n \frac{1}{n} ay_i \right) + \left(\sum_{i=1}^n \frac{1}{n} b \right) \\
 &= a \left(\frac{1}{n} \sum_{i=1}^n y_i \right) + \frac{1}{n} \sum_{i=1}^n b \\
 &= a\bar{y} + \frac{1}{n} nb \\
 &= a\bar{y} + b.
 \end{aligned} \tag{10.40}$$

□

Eine linear-affine Transformation eines Datensatzes transformiert den Mittelwert des Datensatzes also linear-affin. Dies kann man sich zunutze machen, wenn man sich überlegt, wie der Mittelwert lauten sollte, wenn man einen Datensatz neu skaliert, also z. B. von Gramm in Kilogramm umrechnet. Natürlich kann man zur Sicherheit aber auch einfach den Mittelwert des neu skalierten Datensatzes bestimmen. Wir demonstrieren diese Eigenschaft exemplarisch durch Reskalierung des Pre-Interventions-BDI-II-Datensatzes mit dem Wert $a = 2$ unter gleichzeitiger Addition der Konstante $b := 5$.

```
D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                           # PRE-BDI-II-Daten
y_bar  = mean(y)                                         # Mittelwert PRE
a      = 2                                               # Skalierungsfaktor
b      = 5                                               # Addierter Wert
z      = a*y + b                                         # PRE-Werte: a*y + b
z_bar  = mean(z)                                         # Mittelwert nach a*y+b
ay_bar_b = a*y_bar + b                                  # a*Mittelwert + b
```

```
Mittelwert des transformierten Datensatzes : 42.22
Transformation des Mittelwertes           : 42.22
```

10.5.2 Median

In den oben definierten Mittelwert geht jeder Wert eines Datensatzes summativ mit gleichem Gewicht $1/n$ ein. Dies betrifft insbesondere auch solche Werte, die im Vergleich zu den übrigen Werten des Datensatzes sehr untypisch sind. Eine Möglichkeit, die “Mitte” eines Datensatzes unabhängig von solchen untypischen Werten zu bestimmen, besteht in der Bestimmung seines Medians, den wir nachfolgend definieren. Wir haben den Median bereits in Abschnitt Kapitel 10.4 als das 0.5-Quantil kennengelernt, mit dem er identisch ist.

Definition 10.7 (Median). $y = (y_1, \dots, y_n)$ sei ein Datensatz und $y_s = (y_{(1)}, \dots, y_{(n)})$ der zugehörige aufsteigend sortierte Datensatz. Dann ist der Median von y definiert als

$$\tilde{y} := \begin{cases} y_{((n+1)/2)} & \text{falls } n \text{ ungerade,} \\ \frac{1}{2} (y_{(n/2)} + y_{(n/2+1)}) & \text{falls } n \text{ gerade.} \end{cases} \quad (10.41)$$

•

Definition 10.7 impliziert, dass mindestens 50% aller Datenwerte y_i des Datensatzes kleiner oder gleich $\tilde{y}$ sind und dass gleichzeitig mindestens 50% aller Datenwerte y_i des Datensatzes größer oder gleich $\tilde{y}$ sind. Anstelle eines formalen Beweises dieser Aussagen verweisen wir auf Abbildung 10.6. Wie in der Definition des Medians unterscheiden wir dort die Fälle, dass die Anzahl der Datenpunkte ungerade ist (hier $n = 5$, Abbildung 10.6 A) oder gerade ist ($n = 6$, Abbildung 10.6 B). Im Fall der ungeraden Anzahl an Datenpunkten ist der Median der Datenwert mit der Indexzahl, die in der Mitte der Indexzahlen des sortierten Datensatzes liegt, im Beispiel Abbildung 10.6 A also der Wert $y_{(3)}$. Damit sind im vorliegenden Fall 3/5, also 60% der Datenwerte (speziell $y_{(1)}$, $y_{(2)}$ und $y_{(3)}$) kleiner oder gleich dem Median, gleichzeitig aber auch 3/5, also 60% der Datenwerte (speziell $y_{(3)}$, $y_{(4)}$ und $y_{(5)}$) größer oder gleich dem Median. Im Fall einer geraden Anzahl von Datenpunkten ist die Lage des Medians etwas intuitiver: Der Median liegt in der Mitte

der beiden mittleren Werte des sortierten Datensatzes (in Abbildung 10.6 B speziell in der Mitte der Werte $y_{(3)}$ und $y_{(4)}$). Damit sind dann $3/6$, also 50% der Datenwerte (speziell $y_{(1)}$, $y_{(2)}$ und $y_{(3)}$) kleiner als der Median und $3/6$, also 50% der Datenwerte (speziell $y_{(4)}$, $y_{(5)}$ und $y_{(6)}$) größer als der Median.

A

Beispiel für n ungerade

$$n := 5 \Rightarrow \left(\frac{5+1}{2}\right) = (3) \Rightarrow \tilde{y} := y_{(3)}$$

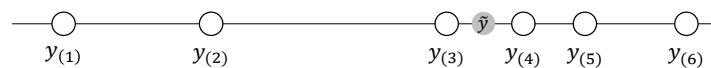


$$y_{(1)}, y_{(2)}, y_{(3)} \leq \tilde{y} \leq y_{(3)}, y_{(4)}, y_{(5)}$$

B

Beispiel für n gerade

$$n := 6 \Rightarrow \left(\frac{6}{2}\right) = (3), \left(\frac{6}{2} + 1\right) = (4) \Rightarrow \tilde{y} := \frac{1}{2}(y_{(3)} + y_{(4)})$$



$$y_{(1)}, y_{(2)}, y_{(3)} < \tilde{y} < y_{(4)}, y_{(5)}, y_{(6)}$$

Abbildung 10.6 Bestimmung und Eigenschaften des Medians.

Folgender **R**-Code bestimmt den Median des Beispieldatensatzes anhand von Definition 10.7.

```
D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                           # PRE-BDI-II-Daten
n      = length(y)                                       # Anzahl der Werte
y_s    = sort(y)                                         # Aufsteigend sortieren
if(n %% 2 == 1){                                         # n ungerade, n mod 2 == 1
  y_tilde = y_s[(n+1)/2]                                 # Median für ungerades n
} else {                                                # n gerade, n mod 2 == 0
  y_tilde = (y_s[n/2] + y_s[n/2 + 1])/2}                # Median für gerades n
```

Median der PRE-BDI-II-Daten: 19

Natürlich stellt **R** mit `median()` auch eine Funktion zur direkten Bestimmung des Medians bereit, wie folgender **R**-Code demonstriert.

```
cat("Median der PRE-BDI-II-Daten: ", median(y)) # Ausgabe
```

Median der PRE-BDI-II-Daten: 19

Folgende Simulation zeigt beispielhaft, dass der Median als Maß der Mitte eines Datensatzes weniger anfällig für Ausreißer ist als der Mittelwert.

```
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y = D$PRE                                           # PRE-BDI-II-Daten
```

Mittelwert und Median f<U+00FC>r Datensatz ohne Ausrei<U+00DF>er: 18.61 19

```
z = y                                           # neuer simulierter Datensatz
z[1] = 10000                                  # ... mit einem Extremwert
```

Mittelwert und Median f<U+00FC>r Datensatz mit Ausrei<U+00DF>er: 118.44 19

Obwohl der Median also als robusteres Maß für die zentrale Tendenz eines Datensatzes als der Mittelwert erscheint, ist die Bestimmung von Mittelwerten sicherlich das üblichere Vorgehen. Dies ist einerseits dadurch begründet, dass sich der Mittelwert aufgrund seiner mathematisch weniger komplexen Definition einfacher im analytischen Kontext zur Gestaltung von probabilistischen Modellen eignet. Andererseits ist das Auftreten von Extremwerten üblicherweise ein Anzeichen von Datenerhebungsfehlern (z. B. sollte ein BDI-II-Wert von 10.000 schlicht nicht vorkommen) oder großen Unterschieden zwischen den betrachteten experimentellen Einheiten, die durch visuelle Inspektion des Datensatzes meist vor der Bestimmung von Maßen der zentralen Tendenz auffallen. Es gibt allerdings ein ganzes Unterfeld der Statistik, das sich mit – im weitesten Sinne – ausreißerunabhängigen Maßen beschäftigt, vgl. z. B. Huber (1981).

10.5.3 Modalwert

Wenn in einem Datensatz manche Werte wiederholt auftreten, ist mit dem *Modalwert* als dem in einem Datensatz am häufigsten vorkommenden Wert ein drittes Maß der zentralen Tendenz gegeben. Wir nutzen folgende Definition.

Definition 10.8 (Modalwert). $y := (y_1, \dots, y_n)$ sei ein Datensatz, W die Menge seiner verschiedenen Zahlenwerte und h seine absolute Häufigkeitsverteilung. Dann heißt jeder Wert $w \in W$, für den

$$h(w) = \max_{v \in W} h(v), \quad (10.42)$$

gilt, ein *Modalwert* von y . Die Menge aller Modalwerte ist

$$M := \{w \in W \mid h(w) = \max_{v \in W} h(v)\}. \quad (10.43)$$

Enthält M genau einen Wert, ist der Modalwert eindeutig.

•

In **R** kann man alle Modalwerte eines Datensatzes wie folgt bestimmen.

```
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y = D$PRE                                           # PRE-BDI-II-Daten
n = length(y)                                       # Anzahl der Werte (100)
H = as.data.frame(table(y))                        # Absolute Häufigkeiten
names(H) = c("a", "w")                             # konsistente Benennung
mod = as.numeric(as.character(H$a[H$w == max(H$w)])) # alle Modalwerte
```

```
cat("Modalwerte der PRE-BDI-II-Daten: ", mod) # Ausgabe der Modalwerte
```

```
Modalwerte der PRE-BDI-II-Daten: 18
```

10.5.4 Datenverteilungsformen und Maße zentraler Tendenz

Wie sinnvoll es ist, mit dem Mittelwert, dem Median oder dem Modalwert eine Angabe über die zentrale Tendenz eines Datensatzes zu geben, hängt stark von der Beschaffenheit des Datensatzes ab. Hat man beispielsweise einen Datensatz hochaufgelöster Daten vorliegen, in dem kein Datenwert mehrfach auftaucht, eignet sich der Modalwert sicherlich nicht, um eine quantitative Aussage über die zentrale Tendenz des Datensatzes zu machen. Man sollte die Angabe eines Maßes der zentralen Tendenz also immer im Kontext der Gesamtbeschaffenheit des Datensatzes sowie, wenn zutreffend, des inferenzstatistischen Modells, das man zur Interrogation des Datensatzes nutzt, betrachten. In [Abbildung 10.7](#) geben wir vier Beispiele, wie sich die oben betrachteten Maße der zentralen Tendenz in verschiedenen Szenarien der Datenwertverteilung verhalten können. Diese vier Szenarien sind natürlich nicht erschöpfend und jeder Datensatz ist in Hinblick auf ein sinnvolles Maß der zentralen Tendenz kritisch zu betrachten.

[Abbildung 10.7 A](#) visualisiert die Lage von Mittelwert, Median und Modalwerten in einem Datensatz, der sich symmetrisch um einen Wert von etwa $y = 50$ verteilt und für den Werte nah bei diesem zentralen Wert häufig vorkommen und Werte fern von diesem zentralen Wert eher selten. Wie wir später sehen, entspricht dies den typischen Charakteristika normalverteilter Datenwerte. Die Werte 53 und 55 kommen mit jeweils 45 Beobachtungen gleich häufig und am häufigsten vor; beide sind daher Modalwerte dieses Datensatzes. In diesem Fall liegen Mittelwert, Median und die beiden Modalwerte recht nahe beieinander und beschreiben die zentrale Tendenz des Datensatzes.

[Abbildung 10.7 B](#) visualisiert die Lage von Mittelwert, Median und Modalwerten in einem Datensatz, in dem die Datenwerte um zwei Werte von etwa $y = 25$ und $y = 75$ gehäuft auftreten und jeweils in positive und negative Richtung von diesen Werten entfernt eher selten auftreten. So liegen zum Beispiel um $y = 50$ recht wenige Datenpunkte dieses Datensatzes. Mittelwert und Median nehmen nun aber gerade Werte um $y = 50$ an. Als solche geben sie bei diesem Datensatz also keine Auskunft darüber, welche Werte besonders häufig vorkommen, sondern lediglich darüber, wo sich der mittlere “physikalische Schwerpunkt” des Datensatzes befindet. Die beiden Modalwerte 72 und 74 kommen mit jeweils 27 Beobachtungen am häufigsten vor und liegen beide beim größeren der beiden Datenhäufungspunkte. Insgesamt betrachtet beschreibt keines der drei Maße bei einem solchen Datensatz, den man auch *bimodal* nennt, die zentrale Tendenz der Datenwerte wirklich zufriedenstellend.

[Abbildung 10.7 C](#) visualisiert die Lage von Mittelwert, Median und Modalwert in einem Datensatz, in dem die Datenwerte in der betrachteten Breite etwa durchgängig gleich häufig auftreten. Hier liegt der Modalwert relativ arbiträr bei etwa $y = 28$, da sich dort eine minimale Häufung von Datenwerten zeigt. Mittelwert und Median liegen wiederum im Sinne des “physikalischen Schwerpunkts” des Datensatzes bei etwa $y = 50$, auch wenn die Werte dieses Datensatzes dort keinerlei Tendenz haben, häufiger als andernorts aufzutreten. Wirklich zufriedenstellend ist auch hier die Beschreibung der zentralen Tendenz des Datensatzes durch die betrachteten Maße nicht. Zusammen betrachtet kann man anhand von

Abbildung 10.7 B und Abbildung 10.7 C vielleicht festhalten, dass, wenn ein Datensatz keine klare zentrale Tendenz hat, die Maße der zentralen Tendenz diese auch nicht abbilden können.

Abbildung 10.7 D zeigt die Lage von Mittelwert, Median und Modalwert für einen Datensatz mit *rechtsschiefer Verteilung*. Die Daten werden aus einer Gammaverteilung mit Formparameter 2 und Rate 2 simuliert. Die zugehörige Dichte hat ihren Modalwert bei 0.5. Höhere Werte treten mit abnehmender Dichte auf, wobei die Verteilung nach rechts einen langen Ausläufer besitzt. Im psychologischen Kontext haben insbesondere Reaktionszeiten oft eine ähnliche Verteilung, da sie nicht negativ sein können, in den meisten Fällen etwa im Bereich von 0.3 bis 0.6 Sekunden liegen, aber in manchen Fällen durch Unaufmerksamkeit oder Stimulusvariabilität auch sehr viel länger sein können. In diesem Fall beschreibt der Median die tatsächliche zentrale Tendenz der Datenwerte etwas besser als der Mittelwert, da dieser durch die selten vorkommenden Ausreißerwerte etwas zu höheren Werten verschoben ist.

Insgesamt ist die Angabe eines oder mehrerer Maße der zentralen Tendenz also immer kritisch vor dem Hintergrund der Verteilung der Werte eines Datensatzes zu betrachten und qualitativ einzuordnen.

10.6 Maße der Variabilität

Neben der Frage, welchen “durchschnittlichen Wert” ein Datensatz annimmt, ist es oft von Interesse zu quantifizieren, wie sehr die Werte im Datensatz streuen oder anders ausgedrückt, wie “variabel” sie sind. Als quantitative Maße für die Variabilität eines Datensatzes betrachten wir hier die *Spannbreite*, die *empirische Varianz* und die *empirische Standardabweichung*. Wie auch für den Mittelwert erschließt sich die volle Bedeutung letzterer Maße vor allem im Kontext inferenzstatistischer Verfahren.

10.6.1 Spannbreite

Die Spannbreite eines Datensatzes ist die Differenz zwischen seinem größten und seinem kleinsten Wert.

Definition 10.9 (Spannbreite). $y = (y_1, \dots, y_n)$ sei ein Datensatz. Dann ist die *Spannbreite* von $y_1, \dots, y_n$ definiert als

$$r := \max(y_1, \dots, y_n) - \min(y_1, \dots, y_n). \quad (10.44)$$

•

In **R** bestimmt man die Spannbreite entweder durch Evaluation des Maximums und des Minimums des Datensatzes mithilfe der `max()`- und `min()`-Funktionen oder direkt mithilfe der `range()`-Funktion. Folgender **R**-Code demonstriert die Anwendung der `max()`- und `min()`-Funktionen.

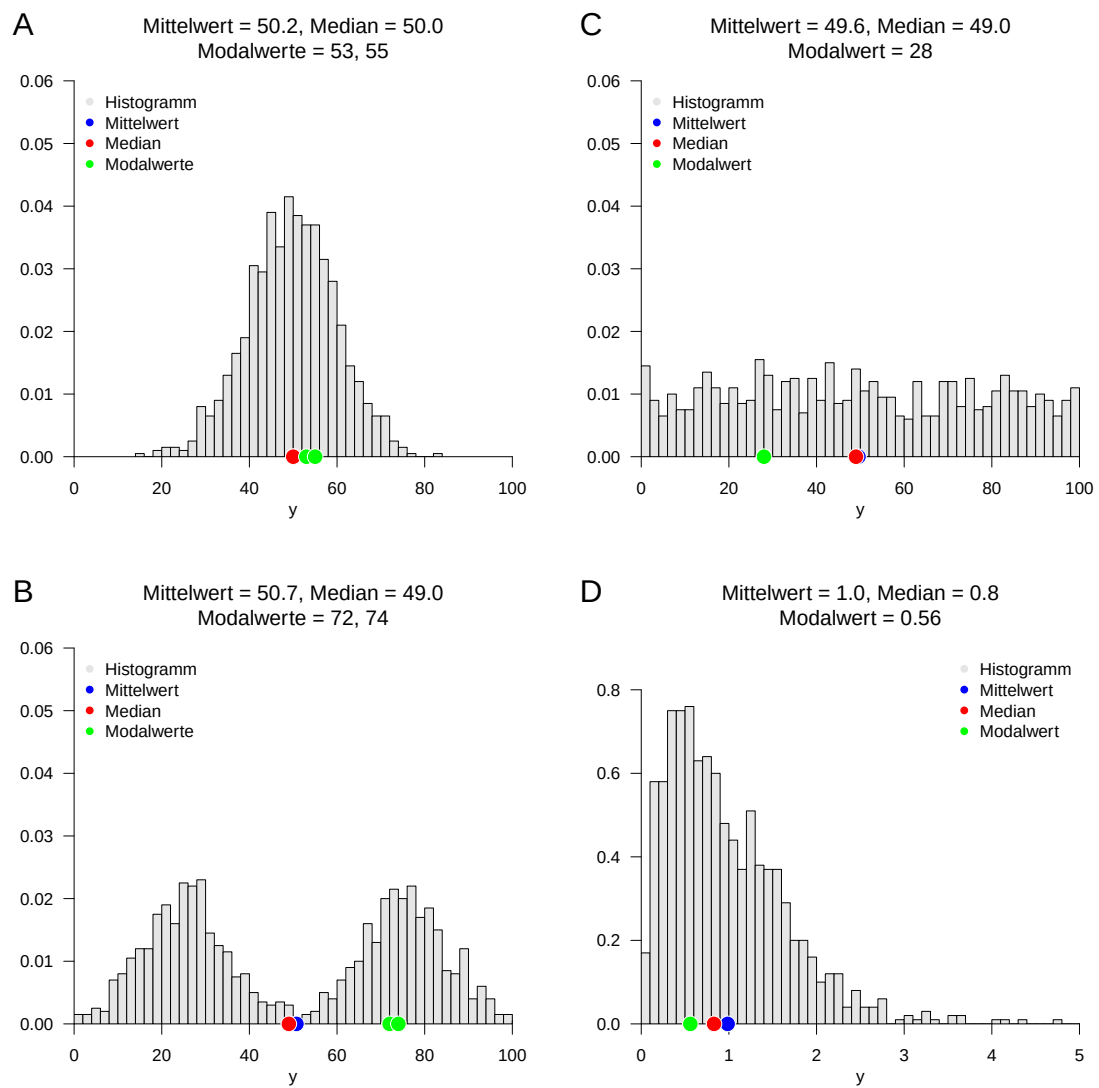


Abbildung 10.7 Visuelle Intuition zu Maßen zentraler Tendenz bei verschiedenen Datenverteilungsformen

```

D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                           # PRE-BDI-II-Daten
y_max  = max(y)                                          # Maximum der PRE-Werte
y_min  = min(y)                                          # Minimum der PRE-Werte
r      = y_max - y_min                                   # Spannweite

```

Spannweite der PRE-BDI-II-Daten: 9

Folgender **R**-Code demonstriert die Anwendung der `range()`-Funktion.

```

MinMax = range(y)                                     # Minimum und Maximum
r      = MinMax[2] - MinMax[1]                         # Spannweite

```

Spannweite der PRE-BDI-II-Daten: 9

10.6.2 Empirische Varianz

Ein typisches statistisches Maß für die Variabilität von Datensätzen ist die standardisierte Summe der quadrierten Differenzen zwischen den einzelnen Datensatzwerten und ihrem Mittelwert. Die quadrierten Differenzen zwischen den einzelnen Datensatzwerten und ihrem Mittelwert werden auch als *Abweichungsquadrate* bezeichnet. Je nach Form der Standardisierung, also der Bildung eines Durchschnitts der Abweichungsquadrate, unterscheidet man eine *unkorrigierte empirische Varianz* und eine *empirische Varianz*, wie folgende Definition festsetzt.

Definition 10.10 (Unkorrigierte empirische Varianz und empirische Varianz). $y := (y_1, \dots, y_n)$ sei ein Datensatz mit $n \geq 1$ und $\bar{y}$ sei sein Mittelwert. Dann ist die *unkorrigierte empirische Varianz* von y definiert als

$$\tilde{s}^2 := \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 \quad (10.45)$$

Für $n > 1$ ist die *empirische Varianz* von y definiert als

$$s^2 := \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2. \quad (10.46)$$

•

Wir haben in Theorem 10.1 gesehen, dass die Summe der Differenzen zwischen den Datenwerten und ihrem Mittelwert immer gleich null ist. Die Quadrierung der Differenzen

$$y_i - \bar{y} \text{ für } i = 1, \dots, n \quad (10.47)$$

in Definition 10.10 führt nun gerade dazu, dass sich negative und positive Differenzen von Datenpunkten nicht ausgleichen und, je nach Höhe der positiven und negativen Differenzen, mit

$$\sum_{i=1}^n (y_i - \bar{y})^2 \quad (10.48)$$

ein höheres oder geringeres Maß für die Gesamtabweichung der Datenwerte von ihrem Mittelwert bestimmt werden kann. Dabei ist die Summe der Abweichungsquadrate gerade dann null, wenn alle Datenwerte identisch und damit auch identisch mit ihrem Mittelwert sind. Alternativ wäre zum Beispiel auch die Bestimmung der Absolutbeträge $|y_i - \bar{y}|$ denkbar, allerdings erfüllt das so entstehende Maß der Variabilität andere Eigenschaften als das Maß der quadrierten Abweichungen in Definition 10.10, weshalb dieses, auch in Hinblick auf seine Nähe zu probabilistischen Normalverteilungsmodellen, in der Regel bevorzugt wird. Die unkorrigierte empirische Varianz $\tilde{s}^2$ und die (korrigierte) empirische Varianz s^2 unterscheiden sich dann hinsichtlich der Form der Durchschnittsbildung der Abweichungsquadrate. Für $\tilde{s}^2$ wird die Summe der Abweichungsquadrate durch n , für s^2 durch $n - 1$ geteilt. Die unkorrigierte empirische Varianz $\tilde{s}^2$ ist höchstens so groß wie die empirische Varianz s^2 . Für $n > 1$ gilt

$$\tilde{s}^2 = \frac{n-1}{n} s^2 \leq s^2. \quad (10.49)$$

Bei positiver Varianz ist $\tilde{s}^2 < s^2$. Sind alle Datenwerte identisch, sind beide Varianzen null. Allerdings wird dieser Unterschied für großes n eher gering ausfallen, man denke beispielsweise an den numerischen Unterschied zwischen $\frac{1}{3}$ und $\frac{1}{2}$ bei kleinem n sowie zwischen $\frac{1}{1001}$ und $\frac{1}{1000}$ bei großem n . Man beachte, dass die empirische Varianz für $n = 1$ nicht definiert ist, die unkorrigierte empirische Varianz jedoch auch in diesem Spezialfall prinzipiell bestimmt werden kann. Der Begriff *unkorrigiert* (und implizit *korrigiert*) kann sich an dieser Stelle noch nicht erschließen, sondern ergibt nur vor dem Hintergrund eines frequentistischen Inferenzmodells Sinn und wird in Kapitel „Parameterschätzung“ im PDWP-Lehrbuch erläutert werden.

Wir fassen die obige Diskussion im folgenden Theorem zusammen.

Theorem 10.4 (Verhältnis von unkorrigierter empirischer Varianz und empirischer Varianz). $y = (y_1, \dots, y_n)$ sei ein Datensatz mit $n > 1$ und $\tilde{s}^2$ und s^2 seien in dieser Reihenfolge seine unkorrigierte empirische Varianz und seine empirische Varianz. Dann gelten

(1) (Umrechnung)

$$\tilde{s}^2 = \frac{n-1}{n} s^2 \text{ und } s^2 = \frac{n}{n-1} \tilde{s}^2. \quad (10.50)$$

(2) (Ungleichung)

$$0 \leq \tilde{s}^2 \leq s^2. \quad (10.51)$$

◦

Beweis. (1) Nach Definition 10.10 gilt zum einen

$$\begin{aligned} \tilde{s}^2 &= \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 \\ \Leftrightarrow \frac{n-1}{n-1} \tilde{s}^2 &= \frac{n-1}{n-1} \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 \\ \Leftrightarrow \tilde{s}^2 &= \frac{n-1}{n} \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2 \\ \Leftrightarrow \tilde{s}^2 &= \frac{n-1}{n} s^2 \end{aligned} \quad (10.52)$$

und zum anderen

$$\begin{aligned}
 s^2 &= \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2 \\
 \Leftrightarrow \frac{n}{n} s^2 &= \frac{n}{n} \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2 \\
 \Leftrightarrow s^2 &= \frac{n}{n-1} \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 \\
 \Leftrightarrow s^2 &= \frac{n}{n-1} \tilde{s}^2.
 \end{aligned} \tag{10.53}$$

(2) Mit der Nichtnegativität der Quadratfunktion sowie $\frac{1}{n} > 0$ und $\frac{1}{n-1} > 0$ für $n > 1$ ergeben sich $\tilde{s}^2 \geq 0$ und $s^2 \geq 0$ direkt. Schließlich gilt für $\sum_{i=1}^n (y_i - \bar{y})^2 \neq 0$

$$\frac{1}{n} < \frac{1}{n-1} \Leftrightarrow \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 < \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2 \Leftrightarrow \tilde{s}^2 < s^2. \tag{10.54}$$

und für $\sum_{i=1}^n (y_i - \bar{y})^2 = 0$ offenbar $\tilde{s}^2 = s^2$. Insgesamt gilt also

$$0 \leq \tilde{s}^2 \leq s^2. \tag{10.55}$$

□

Folgender **R**-Code demonstriert die Bestimmung der unkorrigierten empirischen Varianz und der empirischen Varianz anhand der in Definition 10.10 festgelegten Formeln.

```

D      = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y      = D$PRE                                           # PRE-BDI-II-Daten
n      = length(y)                                       # Anzahl der Werte
s2_tilde = (1/n)*sum((y - mean(y))^2)                  # Unkorrigierte Varianz
s2      = (1/(n-1))*sum((y - mean(y))^2)               # empirische Varianz

```

```

Unkorrigierte empirische Varianz der PRE-BDI-II-Daten : 2.998
Empirische Varianz der PRE-BDI-II-Daten              : 3.028

```

Die **R**-Funktion `var()` bestimmt per Default die empirische Varianz. Mithilfe von Aussage (1) in Theorem 10.4 kann basierend auf dieser Funktion auch die unkorrigierte empirische Varianz bestimmt werden.

```

s2      = var(y)                                         # empirische Varianz
s2_tilde = ((n-1)/n)*var(y)                             # Unkorrigierte Varianz

```

```

Unkorrigierte empirische Varianz der PRE-BDI-II-Daten: 2.998
Empirische Varianz der PRE-BDI-II-Daten              : 3.028

```

Folgendes Theorem besagt, wie sich die empirische Varianz bei linear-affiner Transformation eines Datensatzes verhält.

Theorem 10.5 (Varianz bei linear-affinen Transformationen). $y := (y_1, \dots, y_n)$ sei ein Datensatz mit $n > 1$ und empirischer Varianz s_y^2 und $z = (ay_1 + b, \dots, ay_n + b)$ sei der mit $a, b \in \mathbb{R}$ linear-affin transformierte Datensatz mit empirischer Varianz s_z^2 . Dann gilt

$$s_z^2 = a^2 s_y^2. \tag{10.56}$$

◦

Beweis.

$$\begin{aligned}
 s_z^2 &= \frac{1}{n-1} \sum_{i=1}^n (z_i - \bar{z})^2 \\
 &= \frac{1}{n-1} \sum_{i=1}^n (ay_i + b - (a\bar{y} + b))^2 \\
 &= \frac{1}{n-1} \sum_{i=1}^n (ay_i + b - a\bar{y} - b)^2 \\
 &= \frac{1}{n-1} \sum_{i=1}^n (a(y_i - \bar{y}))^2 \\
 &= \frac{1}{n-1} \sum_{i=1}^n a^2 (y_i - \bar{y})^2 \\
 &= a^2 \frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2 \\
 &= a^2 s_y^2.
 \end{aligned} \tag{10.57}$$

□

Beispielsweise ändert also das Umrechnen eines Datensatzes von Meter in Zentimeter die Varianz des Ausgangsdatsatzes um den Faktor 100^2 . Wir reproduzieren Theorem 10.5 beispielhaft mithilfe folgenden **R**-Codes.

```

y = D$PRE # PRE-BDI-II-Daten
s2y = var(y) # Varianz von y
a = 2 # Skalierungsfaktor
b = 5 # Addierter Wert
z = a*y + b # z_i = ay_i + b
s2z = var(z) # Varianz von z
cat( # Ausgabe
  "Empirische Varianz von z_1,...,z_n durch direkte Berechnung : ", # Beschriftung
  round(s2z,3)) # Gerundeter Wert

```

Empirische Varianz von z_1,...,z_n durch direkte Berechnung : 12.113

```

s2z = a^2*s2y # Varianz von z
cat( # Ausgabe
  "Empirische Varianz von z_1,...,z_n nach Theorem : ", # Beschriftung
  round(s2z,3)) # Gerundeter Wert

```

Empirische Varianz von z_1,...,z_n nach Theorem : 12.113

Folgendes Theorem zeigt eine Möglichkeit auf, die unkorrigierte empirische Varianz eines Datensatzes allein durch die Bestimmung von Mittelwerten zu bestimmen. Das Theorem findet seine analytische Entsprechung im Verschiebungssatz der Varianz, [Verschiebungssatz der Varianz im PDWP-Lehrbuch](#).

Theorem 10.6 (Verschiebungssatz zur unkorrigierten empirischen Varianz). $y = (y_1, \dots, y_n)$ sei ein Datensatz, $y^2 := (y_1^2, \dots, y_n^2)$ sei sein elementweises Quadrat und $\bar{y}$ und $\overline{y^2}$ seien die respektiven Mittelwerte. Dann gilt

$$\tilde{s}^2 = \overline{y^2} - \bar{y}^2 \tag{10.58}$$

◦

Beweis.

$$\begin{aligned}
 \tilde{s}^2 &:= \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 \\
 &= \frac{1}{n} \sum_{i=1}^n (y_i^2 - 2y_i\bar{y} + \bar{y}^2) \\
 &= \frac{1}{n} \sum_{i=1}^n y_i^2 - 2\bar{y} \frac{1}{n} \sum_{i=1}^n y_i + \frac{1}{n} \sum_{i=1}^n \bar{y}^2 \\
 &= \overline{y^2} - 2\bar{y}\bar{y} + \frac{1}{n} n \bar{y}^2 \\
 &= \overline{y^2} - 2\bar{y}^2 + \bar{y}^2 \\
 &= \overline{y^2} - \bar{y}^2.
 \end{aligned} \tag{10.59}$$

□

Wir reproduzieren Theorem 10.6 beispielhaft mithilfe folgenden **R**-Codes.

```

y      = D$PRE                                # PRE-BDI-II-Daten
s2_def = sum((y - mean(y))^2)/length(y)        # Varianz nach Definition
s2_tilde = mean(y^2) - mean(y)^2              # Per Verschiebungssatz
s2      = var(y)                              # Empirische Varianz

cat(                                           # Ausgabe
  "Unkorrigierte Varianz nach Definition      :", # Beschriftung
  round(s2_def,6),                            # Gerundeter Wert
  "\n")                                       # Zeilenumbruch

```

Unkorrigierte Varianz nach Definition : 2.9979

```

cat(                                           # Ausgabe
  "Unkorrigierte Varianz nach Verschiebungssatz:", # Beschriftung
  round(s2_tilde,6),                            # Gerundeter Wert
  "\n")                                       # Zeilenumbruch

```

Unkorrigierte Varianz nach Verschiebungssatz: 2.9979

```

cat(                                           # Ausgabe
  "Empirische Varianz                        :", # Beschriftung
  round(s2,6),                               # Gerundeter Wert
  "\n")                                       # Zeilenumbruch

```

Empirische Varianz : 3.028182

Man beachte, dass der Verschiebungssatz der unkorrigierten empirischen Varianz für die empirische Varianz nicht zutrifft, da in diesem Fall der Multiplikationsfaktor $\frac{1}{n-1}$ nicht auf die entsprechenden Mittelwerte führt. Insbesondere haben wir oben schon gesehen, dass für den betreffenden Beispieldatensatz gilt, dass $s^2 = 3.028$.

10.6.3 Empirische Standardabweichung

Die empirische Varianz ergibt sich wie oben gesehen als durchschnittliches Abweichungsquadrat eines Datensatzwertes von seinem Mittelwert. Ist die Einheit des Datensatzes dabei zum Beispiel in einer Reaktionszeitaufgabe die Millisekunde, so ergibt sich durch Differenzbildung und anschließende Quadrierung die Einheit Millisekunde², eine nicht sehr intuitive Einheit. Um ein Streuungsmaß zu erhalten, dessen Einheit der Einheit des Ausgangsdatsatzes entspricht, wird auf die empirische Varianz häufig noch die Quadratwurzelfunktion angewendet. Dies führt auf den Begriff der *empirischen Standardabweichung*.

Definition 10.11 (Empirische Standardabweichung). $y := (y_1, \dots, y_n)$ sei ein Datensatz mit $n \geq 1$. Die *unkorrigierte empirische Standardabweichung* von y ist definiert als

$$\tilde{s} := \sqrt{\tilde{s}^2}. \quad (10.60)$$

Für $n > 1$ ist die *empirische Standardabweichung* von y definiert als

$$s := \sqrt{s^2}. \quad (10.61)$$

•

In Analogie zu den Eigenschaften der empirischen Varianz und ihrer unkorrigierten Version kann auch die unkorrigierte empirische Standardabweichung leicht in die empirische Standardabweichung umgerechnet werden und umgekehrt. Für $n > 1$ gelten

$$\tilde{s} = \sqrt{\frac{n-1}{n}} s \text{ und } s = \sqrt{\frac{n}{n-1}} \tilde{s}. \quad (10.62)$$

Folgender **R**-Code demonstriert die Bestimmung der empirischen Standardabweichung anhand der in Definition 10.11 festgelegten Formel.

```
y = D$PRE           # PRE-BDI-II-Daten
n = length(y)        # Anzahl der Werte
s = sqrt((1/(n-1))*sum((y - mean(y))^2)) # Standardabweichung
```

Empirische Standardabweichung : 1.74

Die **R**-Funktion `sd()` bestimmt per Default die empirische Standardabweichung.

```
s = sd(y)
```

Empirische Standardabweichung : 1.74

Transformiert man einen Datensatz linear-affin, so geht die Multiplikationskonstante der Transformation lediglich im Sinne ihres Betrages in die Transformation der empirischen Standardabweichung ein. Dies ist die Aussage des folgenden Theorems.

Theorem 10.7 (Empirische Standardabweichung bei linear-affinen Transformationen).
 $y := (y_1, \dots, y_n)$ sei ein Datensatz mit $n > 1$ und empirischer Standardabweichung s_y und
 $z = (ay_1 + b, \dots, ay_n + b)$ sei der mit $a, b \in \mathbb{R}$ linear-affin transformierte Datensatz mit
 empirischer Standardabweichung s_z . Dann gilt

$$s_z = |a|s_y. \quad (10.63)$$

◦

Beweis.

$$\begin{aligned} s_z &:= \left(\frac{1}{n-1} \sum_{i=1}^n (z_i - \bar{z})^2 \right)^{1/2} \\ &= \left(\frac{1}{n-1} \sum_{i=1}^n (ay_i + b - (a\bar{y} + b))^2 \right)^{1/2} \\ &= \left(\frac{1}{n-1} \sum_{i=1}^n (a(y_i - \bar{y}))^2 \right)^{1/2} \\ &= \left(\frac{1}{n-1} \sum_{i=1}^n a^2 (y_i - \bar{y})^2 \right)^{1/2} \\ &= (a^2)^{1/2} \left(\frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2 \right)^{1/2}. \end{aligned} \quad (10.64)$$

Also gilt $s_z = as_y$, wenn $a \geq 0$ und $s_z = -as_y$, wenn $a < 0$. Dies aber entspricht $s_z = |a|s_y$. $\square$

Wir reproduzieren Theorem 10.7 beispielhaft mithilfe folgenden **R**-Codes einmal für den Fall $a \geq 0$ und einmal für den Fall $a < 0$.

```
# a >= 0
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y = D$PRE # PRE-BDI-II-Daten
a = 2 # Skalierungsfaktor
b = 5 # Addierter Wert
z = a*y + b # z_i = ay_i + b
sz = sd(z) # Standardabweichung z
```

Empirische Standardabweichung nach Transformation : 3.48

```
sz = a*sd(y) # Empirische Standardabweichung nach Theorem
```

Empirische Standardabweichung nach Transformation : 3.48

```
# a < 0
D = read.csv("./_data/110-deskriptive-statistik.csv") # Datensatz laden
y = D$PRE # PRE-BDI-II-Daten
a = -3 # Skalierungsfaktor
b = 10 # Addierter Wert
z = a*y + b # z_i = ay_i + b
sz = sd(z) # Standardabweichung z
```

Empirische Standardabweichung nach Transformation : 5.221

```
sz = -a*sd(y) # Empirische Standardabweichung nach Theorem
```

Empirische Standardabweichung nach Transformation : 5.221

Referenzen

- Becker, R. A., Chambers, J. M., & Wilks, A. R. (1988). *The New S Language: A Programming Environment for Data Analysis and Graphics* (Reprint). Chapman & Hall.
- Bertram, M. G., Sundin, J., Roche, D. G., Sánchez-Tójar, A., Thoré, E. S. J., & Brodin, T. (2023). Open Science. *Current Biology*, 33(15), R792–R797. <https://doi.org/10.1016/j.cub.2023.05.036>
- Cotton, R. (2013). *Learning R* (First Edition). O'Reilly.
- Dahm, M. (2005). *Grundlagen Der Mensch-Computer-Interaktion*. Pearson Studium.
- Henze, N. (2018). *Stochastik für Einsteiger*. Springer Fachmedien Wiesbaden. <https://doi.org/10.1007/978-3-658-22044-0>
- Huber, P. J. (1981). *Robust Statistics*. Wiley.
- Hyndman, R. J., & Fan, Y. (1996). Sample Quantiles in Statistical Packages. *The American Statistician*, 50(4), 361. <https://doi.org/10.2307/2684934>
- Ihaka, R., & Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3), 2999–2314.
- McGill, R., Tukey, J. W., & Larsen, W. A. (1978). Variations of Box Plots. *The American Statistician*, 32(1), 12. <https://doi.org/10.2307/2683468>
- Miedema, F. (2022). *Open Science: The Very Idea*. Springer Netherlands. <https://doi.org/10.1007/978-94-024-2115-6>
- Rat für Informationsinfrastrukturen. (2017). *Datenschutz Und Forschungsdaten - Aktuelle Empfehlungen*.
- Riley, J. (2017). *Understanding Metadata: What Is Metadata, and What Is It For*. National Information Standards Organization.
- Rosenblatt, B. (1997). The Digital Object Identifier: Solving the Dilemma of Copyright Protection Online. *Journal of Electronic Publishing*, 3(2). <https://doi.org/10.3998/3336451.0003.204>
- Scott, D. W. (1979). *On Optimal and Data-Based Histograms*. 6.
- Sturges, H. A. (1926). The Choice of a Class Interval. *Journal of the American Statistical Association*, 21(153), 65–66. <https://doi.org/10.1080/01621459.1926.10502161>
- Toelch, U., & Ostwald, D. (2018). Digital Open Science—Teaching Digital Tools for Reproducible and Transparent Research. *PLOS Biology*, 16(7), e2006022. <https://doi.org/10.1371/journal.pbio.2006022>
- Tukey, J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley Pub. Co.
- Ulrich, H., Kock-Schoppenhauer, A.-K., Deppenwiese, N., Gött, R., Kern, J., Lablans, M., Majeed, R. W., Stöhr, M. R., Stausberg, J., Varghese, J., Dugas, M., & Ingenerf, J. (2022). Understanding the Nature of Metadata: Systematic Review. *Journal of Medical Internet Research*, 24(1), e25440. <https://doi.org/10.2196/25440>
- Van Rooij, I., & Wareham, T. (2012). Intractability and Approximation of Optimization Theories of Cognition. *Journal of Mathematical Psychology*, 56(4), 232–247. <https://doi.org/10.1016/j.jmp.2012.05.002>
- Von Neumann, J. (1945). *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania. <https://doi.org/10.5479/sil.538961.39088011475779>

- Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L., François, R., Grolemond, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T., Miller, E., Bache, S., Müller, K., Ooms, J., Robinson, D., Seidel, D., Spinu, V., ... Yutani, H. (2019). Welcome to the Tidyverse. *Journal of Open Source Software*, 4(43), 1686. <https://doi.org/10.21105/joss.01686>
- Wilkinson, M. D., Dumontier, M., Aalbersberg, Ij. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., Bouwman, J., Brookes, A. J., Clark, T., Crosas, M., Dillo, I., Dumon, O., Edmunds, S., Evelo, C. T., Finkers, R., ... Mons, B. (2016). The FAIR Guiding Principles for Scientific Data Management and Stewardship. *Scientific Data*, 3(1), 160018. <https://doi.org/10.1038/sdata.2016.18>