



ECDL Computing

Computational Thinking und
Programmieren mit Python

FEICHTINGER Helmut

Dipl. Trainer

Microsoft® Certified
Professional

Vorwort/Lernziele	5	4.5 Grundaufbau eines Programms am Beispiel Python	43
1 Informationen zu diesem Buch	7	4.6 Übungen	48
1.1 Voraussetzungen	7	5 Zahlensysteme und Zeichencodes	49
1.2 Aufbau und Konventionen	8	5.1 Zahlensysteme unterscheiden	49
	9	5.2 Programme basieren auf Daten	51
2 Grundlagen zu Computing, Programmen und Programmiersprachen	10	5.3 Digitales Rechnen	53
2.1 Soft- und Hardware	10	5.4 Zeichencodes	55
2.2 Grundlagen zu Programmen	10	5.5 Übung	57
2.3 Computing und Computational Thinking	11	6 Grundlegende Sprachelemente	58
2.4 Qualitätskriterien und Dokumentation	14	6.1 Syntax und Semantik	58
2.5 Warum programmieren?	18	6.2 Grundlegende Elemente einer Sprache	60
2.6 Klassifizierung von Programmiersprachen	18	6.3 Standarddatentypen (elementare Datentypen)	62
2.7 Die Klassifizierung nach Generationen	19	6.4 Literale für primitive Datentypen	65
2.8 Prozedurale und funktionale Programmiersprachen	21	6.5 Variablen und Konstanten	65
2.9 Objektorientierte Programmiersprachen	23	6.6 Operatoren	71
2.10 Hybride Programmiersprachen und Skriptsprachen	25	6.7 Ausdrücke und Operatorenrangfolgen	75
2.11 Logische Programmiersprachen	27	6.8 Übungen	77
2.12 Erziehungsorientierte Programmiersprachen und Minisprachen	27	7 Kontrollstrukturen	78
2.13 Webprogrammierung und mobile Apps	28	7.1 Anweisungen und Folgen	78
2.14 Übungen	29	7.2 Bedingungen und Kontrollstrukturen	80
3 Programmlogik und Darstellungsmittel für Programmabläufe	31	7.3 Grundlagen zu Verzweigungen	81
3.1 Abstraktion der Wirklichkeit	31	7.4 Bedingte Anweisung	81
3.2 Programmlogik und Programmablauf	31	7.5 Verzweigung	82
3.3 Programmabläufe visualisieren	32	7.6 Geschachtelte Verzweigung	83
3.4 Programmablaufplan	33	7.7 Mehrfache Verzweigung (Fallauswahl)	84
3.5 Datenflussdiagramm	34	7.8 Schleifen	86
3.6 Struktogramme	35	7.9 Zählergesteuerte Schleife (Iteration)	87
3.7 Pseudocode	36	7.10 Kopfgesteuerte bedingte Schleife	89
3.8 Entscheidungstabellen	37	7.11 Fußgesteuerte bedingte Schleife	90
3.9 Übung	38	7.12 Schnellübersicht	92
4 Werkzeuge der Softwareentwicklung	39	7.13 Sprunganweisungen	94
4.1 Programme erstellen	39	7.14 Endlosschleifen	95
4.2 Konzepte zur Übersetzung	40	7.15 Übungen	95
4.3 Entwicklungsumgebungen	42	8 Elementare Datenstrukturen	98
4.4 Standardbibliotheken und Wiederverwendung	43	8.1 Warum werden Datenstrukturen benötigt?	98
		8.2 Arrays	99
		8.3 Eindimensionale Arrays	100

8.4	Records	101	11 Reaktion auf Ereignisse	131
8.5	Zeichenketten	101	11.1 Grundlagen zu Ereignissen und Eventhandlern	131
8.6	Tupel und Listen	102	11.2 Timer und Scheduler	132
8.7	Dictionaries	106	11.3 Ereignisbehandlung bei Programmen mit grafischen Oberflächen	134
8.8	Mengen	107	11.4 Verschiedene Techniken der Ereignisbehandlung	137
8.9	Besondere Datenstrukturen anhand von Stapel (Stack) und Schlangen (Queue)	108	11.5 Schnellübersicht	139
8.10	Übungen	111	11.6 Übung	139
9	Methoden, Prozeduren und Funktionen	113	12 Grundlagen der Softwareentwicklung	140
9.1	Unterprogramme	113	12.1 Software entwickeln	140
9.2	Parameterübergabe	116	12.2 Methoden	142
9.3	Parameterübergabe als Wert	117	12.3 Der Software-Lebenszyklus	143
9.4	Parameterübergabe über Referenzen	117	12.4 Vorgehensmodelle im Überblick	147
9.5	Rückgabewerte von Funktionen oder Methoden	118	12.5 Computergestützte Softwareentwicklung (CASE)	151
9.6	Innere Funktionen – Closures	119	12.6 Fehler finden und identifizieren	152
9.7	Standardbibliotheken und Built-in-Funktionalitäten	119	12.7 Schnellübersicht	162
9.8	Übungen	124	12.8 Übung	162
10	Algorithmen	125	Anhang A: PAP, Struktogramm und Pseudocode	163
10.1	Eigenschaften eines Algorithmus	125	A.1 Beispiel Zinsberechnung	163
10.2	Iterativer Algorithmus	125	A.2 Beispiel Geldautomat	164
10.3	Rekursiver Algorithmus	127	Anhang B: Installationen und Quellangaben	165
10.4	Iterativ oder rekursiv?	128	B.1 Python laden und installieren	165
10.5	Generischer Algorithmus	129	B.2 Quellangaben im Internet	167
10.6	Übung	130	Stichwortverzeichnis	168

Vorwort

Ziele

Die Kandidatinnen und Kandidaten sollen

- Grundlagen des Computing und typische Schritte beim Erstellen eines Programmes verstehen,
- Methoden des Computational Thinking wie Problemzerlegung, Mustererkennung, Abstraktion und algorithmisches Design zur Problemanalyse und Lösungsentwicklung verstehen und anwenden,
- Algorithmen für ein Programm unter Verwendung von Flussdiagrammen und Pseudocode schreiben, testen und bearbeiten,
- Wesentliche Grundsätze und Schlüsselbegriffe des Codings und die Bedeutung von gutstrukturiertem und dokumentiertem Code verstehen,
- Programmierbegriffe wie Variablen, Datentypen und Logik in einem Programm verstehen und verwenden,
- Effizienz und Funktionalität verbessern, indem Iteration, bedingte Anweisungen, Prozeduren und Funktionen sowie Events und Commands in einem Programm eingesetzt werden,
- Programm testen, Fehler bereinigen (debugging) und vor der Auslieferung sicherstellen, dass die erforderlichen Bedingungen erfüllt sind.

Lernziel

1. Begriffe im Bereich Computing

- 1.1. Schlüsselbegriffe

2. Methoden des Computational Thinking

- 2.1. Problemanalyse
- 2.2. Algorithmisches Design

3. Coding

- 3.1. Erste Schritte
- 3.2. Variablen und Daten

4. Konstruktive Verwendung von Code-Elementen

- 4.1. Logik
- 4.2. Schleifen (Iteration)
- 4.3. Bedingte Anweisung
- 4.4. Prozeduren und Funktionen
- 4.5. Ereignisse (Events) und Aufrufe (Commands)

5. Testen, Fehlersuche, Auslieferung

- 5.1. Programm ausführen, testen, Fehler beseitigen
- 5.2. Auslieferung des Programms

6. Ausgabe

- 6.1. Einrichtung
- 6.2. Drucken

1 Informationen zu diesem Buch

In diesem Kapitel erfahren Sie

- ✓ an wen sich dieses Buch richtet
- ✓ welche Vorkenntnisse Sie mitbringen sollten
- ✓ welche Hard- und Software Sie für die Arbeit mit diesem Buch benötigen
- ✓ wie dieses Buch aufgebaut ist
- ✓ welche Konventionen verwendet werden

1.1 Voraussetzungen

Zielgruppe

- ✓ Programmierinsteiger
- ✓ Auszubildende in IT-Berufen
- ✓ Schüler mit IT-Schwerpunkt
- ✓ Teilnehmer ECDL Module und Zertifikate

Empfohlene Vorkenntnisse

Folgende Kenntnisse werden für eine erfolgreiche Benutzung dieses Buchs vorausgesetzt:

- ✓ Grundkenntnisse im Umgang mit Windows, Linux oder MacOS
- ✓ Grundkenntnisse in der Bedienung von Anwendungsprogrammen
- ✓ Grundkenntnisse im Umgang mit dem Internet

Hinweise zur Software

- ✓ Bei den Beispielen im Buch wird überwiegend die Programmiersprache **Python** in der Version 3.6.x verwendet, wenn konkrete Programmierung durchgeführt wird. Python ist eine universelle, höhere Programmiersprache, die üblicherweise interpretiert wird. Python ist den meisten gängigen Programmiersprachen verwandt, wurde aber mit dem Ziel größter Einfachheit und Übersichtlichkeit entworfen. Hinweise zur Installation und Konfiguration von Python finden Sie im Anhang.
- ✓ Bei Sprachelementen, die Python nicht bereitstellt, werden diese bei Bedarf gelegentlich mithilfe von **Java** erläutert, ohne dass sich um eine konkrete Ausführung des Codes gekümmert wird. Dies ist eine an C/C++ angelehnte streng objektorientierte Sprache, die ursprünglich für Geräte der Konsumelektronik entwickelt wurde. Gegenüber C/C++ wurde auf verschiedene (fehlerträchtige) Konstrukte verzichtet. Deshalb ist Java eine relativ schlanke und übersichtliche Sprache. Mit dem Java-Compiler erzeugt man maschinenunabhängigen Code, sogenannten Bytecode, der in einer virtuellen Maschine ausgeführt wird.
- ✓ Als Beispiel für eine komfortable integrierte Entwicklungsumgebung wird **IDLE** verwendet. Diese gehört zum Python-System und wird meist mit Python automatisch installiert. Die Verwendung vereinfacht die Arbeit mit Python erheblich, weshalb im Buch auch nur auf deren Einsatz gesetzt wird. Hinweise zur Installation und Konfiguration von Python finden Sie im Anhang. Als reine Eingabemöglichkeit des Quelltextes kann man im Grunde aber jeden Editor verwenden, der in einem typischen Betriebssystem vorhanden ist. Dieser genügt auch für die Erstellung von Java-Quellcode.

- ✓ Bei einigen Ausblicken werden bei Bedarf verschiedene weitere Programme und Tools verwendet oder erwähnt.
- ✓ Das verwendete Betriebssystem im Buch ist Windows 10, aber Linux und MacOS X werden ebenso berücksichtigt.

1.2 Aufbau und Konventionen

Aufbau und inhaltliche Konventionen des Buchs

- ✓ Am Anfang jedes Kapitels finden Sie die Lernziele und am Ende einiger Kapitel eine Schnellübersicht mit den wichtigsten Funktionen im Überblick.
- ✓ Die meisten Kapitel enthalten Übungen, mit deren Hilfe Sie die erlernten Kapitelinhalte einüben können.

Hervorhebungen im Text

Im Text erkennen Sie bestimmte Programmelemente an der Formatierung. So werden z. B. Bezeichnungen für Programmelemente wie Register immer *kursiv* geschrieben und wichtige Begriffe **fett** hervorgehoben.

<i>Kursivschrift</i>	kennzeichnet alle von Programmen vorgegebenen Bezeichnungen für Schaltflächen, Dialogfenster, Symbolleisten, Menüs bzw. Menüpunkte (z. B. <i>Datei - Schließen</i>) sowie alle vom Anwender zugewiesenen Namen wie Dateinamen, Ordnernamen, eigene Symbolleisten, Hyperlinks und Pfadnamen.
<code>Courier New</code>	kennzeichnet Programmtext, Programmnamen, Funktionsnamen, Variablennamen, Datentypen, Operatoren etc.
<code>Courier New kursiv</code>	kennzeichnet Zeichenfolgen, die vom Anwendungsprogramm ausgegeben oder in das Programm eingegeben werden.
[]	Bei Darstellungen der Syntax einer Programmiersprache kennzeichnen eckige Klammern optionale Angaben.
	Bei Darstellungen der Syntax einer Programmiersprache werden alternative Elemente durch einen senkrechten Strich voneinander getrennt.

Was bedeuten die Symbole im Buch?



Hilfreiche Zusatzinformation



Praxistipp



Warnhinweis

Computing

```
31 def __init__(self, settings):
32     self.file = None
33     self.fingerprints = set()
34     self.logdupes = True
35     self.debug = debug
36     self.logger = logging.getLogger(__name__)
37     if path:
38         self.file = open(os.path.join(path, 'requests.txt'),
39                         'a')
40         self.file.seek(0)
41         self.fingerprints.update(self.requests)
42
43 @classmethod
44 def from_settings(cls, settings):
45     debug = settings.getbool('SUPERFINGER_DEBUG')
46     return cls(job_dir(settings), debug)
47
48 def request_seen(self, request):
49     fp = self.request_fingerprint(request)
50     if fp in self.fingerprints:
51         return True
52     self.fingerprints.add(fp)
53     if self.file:
54         self.file.write(fp + os.linesep)
55
56 def request_fingerprint(self, request):
57     return request_fingerprint(request)
```



2 Grundlagen zu Computing, Programmen und Programmiersprachen

In diesem Kapitel erfahren Sie

- ✓ was es mit den Begriffen „Computing“ und „Computational Thinking“ auf sich hat
- ✓ was Programme sind
- ✓ wie sich die Programmiersprachen geschichtlich entwickelt haben
- ✓ nach welchen Kriterien Programmiersprachen eingeteilt werden

2.1 Soft- und Hardware

Was ist Software?

Der Begriff **Software** wird meist umfassender als der Begriff **Programm** verstanden. Als Software werden alle immateriellen Teile eines Computers verstanden. Das umfasst Programme, aber auch die zugehörigen Daten. Im täglichen Sprachgebrauch werden die Begriffe Software und Programm jedoch oft synonym verwendet.

Was ist Hardware?

Der Begriff **Hardware** wird meist für alle materiellen (physischen) Teile eines Computers verwendet. Also grob gesagt all das, was man anfassen kann. Zur Hardware eines Computers gehören beispielsweise diese Komponenten:

- ✓ Die Grundbestandteile der Rechnerarchitektur: Hauptplatine (auch Motherboard oder Mainboard genannt) mit dem Chip (CPU – Central Processing Unit) bzw. Prozessor und Arbeitsspeicher (RAM – Random-Access Memory)
- ✓ Massenspeicher und Speichermedien
- ✓ Erweiterungskarten (Grafikkarte, Soundkarte, Netzwerkkarte ...)
- ✓ Ergänzende Komponenten wie Netzteil, Gehäuse, Lüfter ...
- ✓ Peripheriegeräte in Form von
 - ✓ Ausgabegeräten (Drucker, Bildschirm, Beamer, Lautsprecher ...)
 - ✓ Eingabe- (Tastatur, Maus, Joystick ...) und Einlesegeräten (Mikrofone, Kartenlesegeräte, Scanner ...)

2.2 Grundlagen zu Programmen

Ein Programm oder Skript ist ganz allgemein ein Lösungsweg zur Bearbeitung einer Aufgabe durch einen Computer oder ein verwandtes System. Im Folgenden wird **Programm** als Synonym für Programm und Skript verwendet, wenn nicht ausdrücklich von einem Skript die Rede ist. Ein Programm wird auch **Applikation (engl. application)** oder **Anwendung** genannt. Die Kurzform der englischen Version von Applikation – **App** – ist vor allem bei der Programmierung für mobile Geräte gebräuchlich, setzt sich aber auch in anderen Bereichen mehr und mehr durch.

Vom Algorithmus über den Quellcode zum Programm

Ein **Programm** bzw. Skript ist also eine Beschreibung der Lösung einer vorgegebenen Aufgabe, muss aber in einer spezifischen Programmiersprache umgesetzt werden.

Die Lösung kann (und wird in der Regel) aus einzelnen Bearbeitungsvorschriften bestehen. Eine solche Bearbeitungsvorschrift wird **Algorithmus** genannt.

Ein Algorithmus muss bei jeder möglichen Eingabe von Daten die Verarbeitung nach **endlich vielen Schritten** beenden und einen **eindeutigen Ablauf** mit einem **reproduzierbaren Ergebnis** besitzen.



Ein Programm besteht aus Algorithmen, deren Arbeitsschritte in einer Programmiersprache (z. B. in Python, Java, C#, oder JavaScript formuliert sind. Dies ist der sogenannte **Quellcode** oder **Quelltext** (engl. **source code** oder kurz **source**). Gelegentlich findet man dafür auch den Begriff **Programmcode**.

Programme verarbeiten Daten

Ein Programm kann Ihnen beispielsweise Steuern berechnen. Sie teilen dem Programm die Daten mit, die für die Berechnung benötigt werden. Das Programm rechnet nach seinem Algorithmus mit Ihren Daten und liefert am Ende ein Ergebnis.

Programme verarbeiten **Daten**, z. B. Benutzereingaben, und liefern Daten zurück. Diesen Datenfluss durch ein Programm nennt man **EVA-Prinzip** (Eingabe - Verarbeitung - Ausgabe).

- ✓ **Eingabe:** In das Programm werden Daten eingegeben, z. B. über eine Tastatur oder eine Datenbank.
- ✓ **Verarbeitung:** Das Programm verarbeitet diese Daten nach einem vorgegebenen Algorithmus.
- ✓ **Ausgabe:** Die Ergebnisse des Programms werden ausgegeben, z. B. auf einen Bildschirm, einen Drucker oder in eine Datenbank.



2.3 Computing und Computational Thinking

Da das ECDL-Modul explizit die Begriffe „Computing“ und „Computational Thinking“ verwendet, sollen diese Begriffe zuerst kompakt erläutert werden.

Computing und Automatisierung

In der IT-Szene wird der Begriff „Computing“ als solches zwar kaum verwendet, aber er ist schon auf einer sehr theoretischen und allgemeinen Ebene standardisiert. Allgemein bezeichnet man damit alle zielorientierten Tätigkeiten, die

- ✓ auf Computern beziehungsweise algorithmischen (berechenbaren, reproduzierbaren) Prozessen aufbauen,
- ✓ von ihnen profitieren oder
- ✓ solche hervorbringen.

Computing ist wie gesagt ein sehr theoretischer und umfassender Begriff. Dieser weist eine Vielzahl unterschiedlicher und teils spezieller Ausprägungen auf. Etwa diese:

- ✓ Der Entwurf von Hardware und Software
- ✓ Die Entwicklung sowie die Herstellung von Hardware und Software
- ✓ Die Verarbeitung, Strukturierung sowie Verwaltung verschiedener Arten von Informationen
- ✓ Das wissenschaftliche Forschen an und mit Computern
- ✓ Das Sammeln von Informationen für bestimmte Zwecke
- ✓ Rechenarbeiten mit hohem Bedarf an Rechenleistung oder Speicherkapazität
- ✓ Die Vernetzung des Alltags mittels „intelligenter“ (smarter) Gegenstände

Aber es gibt noch eine Vielzahl weiterer Ausprägungen, denen letztendlich nur gemein ist, dass sich die damit verbundenen Aufgaben (effizient) **automatisieren** lassen. Und diese Automatisierung basiert auf den gerade eingeführten Algorithmen und Programmen, was im Fokus des Buches stehen soll.

Computational Thinking

Das sogenannte „Computational Thinking“ (computergestütztes Denken) bezeichnet eine der vielen Definitionen für einen computergestützten Denkprozess, die sich über die Jahre entwickelt haben. Bei der so bezeichneten Vorgehensweise ist es das Ziel, ein Problem zu formulieren und seine Lösung(en) so auszudrücken, dass der Prozess von einem Menschen oder einer Maschine bzw. einem Computer gleichermaßen effektiv ausgeführt werden kann.

Eine weitere Charakterisierung des „Computational Thinking“ als theoretisches Konzept versteht dieses als einen iterativen Prozess, der auf drei abstrakten Stufen basiert.

- ✓ Eine Person kann eine Problemstellung identifizieren und abstrakt modellieren.
- ✓ Die Problemstellung kann danach in Teilprobleme oder -schritte zerlegen werden.
- ✓ Es lassen sich darauf aufbauend Lösungsstrategien entwerfen und ausarbeiten und diese formalisiert so darzustellen, dass sie von einem Menschen oder auch einem Computer verstanden und ausgeführt werden können.

Um die Vorgehensweise zu verdeutlichen, kann man sich eine einfache mathematische Aufgabenstellung vorstellen.

Die Aufgabenstellung soll lauten, dass eine Person die Zahlen 1 und 2 addieren und das Ergebnis mit 3 multiplizieren soll. Auch ein Computer soll diesen Vorgang später durchführen können, und es soll ein Muster formuliert werden, mit dem dann auch andere Zahlen so verarbeitet werden können.

1. Problemstellung identifizieren und abstrakt modellieren

Um die einfache Aufgabenstellung nicht unnötig zu verkomplizieren, sei direkt eine explizit abstrakt ausformulierte mathematische Formel notiert, die die einfache Berechnung mit den konkreten Zahlen löst:

Nehme die Zahl 1 und addiere dazu die Zahl 2 und multipliziere dann mit der Zahl 3

Beachten Sie, dass in dem Problem bzw. einer echten mathematischen Darstellung ($1 + 2 * 3$) die Punkt-Vor-Strichrechnung relevant ist – darauf wird gleich eingegangen.

2. In Teilprobleme oder -schritte zerlegen

Die in Schritt 1 formulierte mathematische Formel ist das Resultat von 2 Teilaufgaben, die sich einzeln und nacheinander abarbeiten lassen. Man spricht hier von einer sogenannten **Sequenz** der Teilschritte.

Wenn man diese Aufgabe also zerlegt, gibt es hier die folgenden zwei Teilschritte:

Die Addition von den Zahlen 1 und 2 ($1 + 2$).

Die Multiplikation des Ergebnisses der Addition mit dem Wert 3 ($3 * 3$).

Diese müssen nacheinander ausgeführt werden, da sich die Reihenfolge der Sequenz aus der Zerlegung der abstrakt formulierten mathematischen Formel in Teilschritte von links kommend ergeben hat.

Wir haben hier ein Beispiel für einen Algorithmus (die gesamte Berechnung) und auch die einzelnen Teilschritte sind (kleinere) Algorithmen, aus denen sich eben der gesamte Algorithmus zusammensetzt.

3. In der dritten Phase von „Computational Thinking“ als theoretisches Konzept sollen Lösungsstrategien entworfen und ausgearbeitet sowie formalisiert so dargestellt werden, dass sie von einem Menschen als auch einem Computer verstanden und ausgeführt werden können.

In diesem Beispiel macht die Notwendigkeit der Punkt-Vor-Strichrechnung in der echten mathematischen Formel deutlich, dass diese Darstellung der Aufgabe in Punkt 1 bereits für Menschen missverständlich ist oder nur unter Zuhilfenahme von „Metainformationen“ korrekt gelöst werden kann (in der bisherigen mathematischen Darstellung müsste zuerst die Multiplikation ausgeführt werden). Auch Computer werden normalerweise eine sogenannte Priorität von Operatoren (Vorrang, in welcher Reihenfolge Dinge zu tun sind) beachten und dabei zuerst die Multiplikation und dann erst die Addition durchführen, wenn man den Vorgang als $1 + 2 * 3$ formuliert (also erst $2 * 3$ und dann $1 + 6$).

Wenn man hingegen die nachfolgende Darstellung wählt (unter der Voraussetzung, dass Klammern gültig sind und den Vorrang festlegen), ist das Problem eindeutig für Mensch oder Maschine korrekt zu lösen:

$$(1 + 2) * 3$$

Damit gibt man die Sequenz, in der die Teilalgorithmen auszuführen sind, sauber formalisiert, reproduzierbar und auf andere Situationen übertragbar an.

Oder aber man schreibt explizit die einzelnen Schritte nacheinander bzw. untereinander hin.

Mit der Vorgehensweise hat man in jedem Fall ein Muster in dem gesamten Problem als auch in den Teilproblemen identifiziert und kann Standardlösungen für ähnliche Fälle anbieten und verwenden.

Die Modellierungs- und Problemlösungsprozesse sind dabei von einer Programmiersprache unabhängig.

Zum Abschluss soll nicht verschwiegen werden, dass die Definition des „Computational Thinking“ von vielen Fachleuten stark kritisiert wird.

- ✓ Das Konzept des Computational Thinking wird oft als zu vage kritisiert, da selten klar gemacht wird, wie es sich von anderen Formen des Denkens unterscheidet.
- ✓ Einige Computerwissenschaftler machen sich Sorgen über die Förderung von Computational Thinking als Ersatz für eine breitere Informatikausbildung, da Computational Thinking nur einen sehr kleinen Teil des Feldes darstellt.
- ✓ Andere Forscher befürchten, dass der Schwerpunkt Computational Thinking Computerwissenschaftler dazu ermutigen könnte, bei der Lösung von Problemen viel zu eng vorzugehen und vor allen Dingen die sozialen, ethischen und ökologischen Auswirkungen der von ihnen geschaffenen Technologie zu ignorieren.

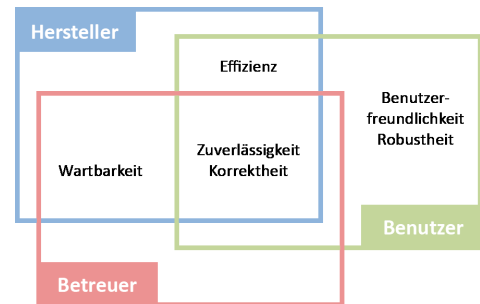
2.4 Qualitätskriterien und Dokumentation

An Software werden hohe Qualitätsansprüche gestellt. Die Software muss dabei den Anforderungen zweier Interessengruppen entsprechen – denen der Hersteller und denen der Kunden. Zur Erfüllung von Qualitätsanforderungen werden **Qualitätskriterien** festgelegt.

Der Kunde achtet bei der Software z. B. auf Benutzerfreundlichkeit und auf den Kundendienst einer Softwarefirma. Für ihn ist es meist wichtig, dass die Software zum bestellten Termin und zum vereinbarten Preis ausgeliefert wird. Außerdem spielt der Einarbeitungsaufwand eine große Rolle.

Dem Hersteller dagegen ist es wichtig, bei der Entwicklung und der nachfolgenden Wartung rentabel zu arbeiten. Sobald das Produkt ausgeliefert ist, geht die Verantwortung vom Hersteller auf den Betreuer der Software, meistens den Hersteller selbst, über. Dieser übernimmt die Wartung der Software, z. B. in Form des Kundendienstes.

Die nebenstehende Abbildung zeigt die Verbindung von Qualitätskriterien.



Korrektheit, Robustheit, Zuverlässigkeit

Die **Korrektheit** stellt bei allen Programmen ein Kriterium dar. Ein Programm ist dann korrekt, wenn es ein vorgegebenes Problem fehlerfrei löst. Die Beurteilung der Korrektheit basiert auf Anforderungen an das Programm, der Produktdefinition (oder dem Pflichtenheft). Die Problematik innerhalb dieses Kriteriums besteht darin, dass Sie nicht entscheiden können, ob ein Programm 100-prozentig korrekt ist. Der vollständige Test aller möglichen Programmzustände ist schon für kleine Programme nur mit erheblichem Aufwand durchführbar. Beim Programmieren sehr umfangreicher Programme scheitert jedoch dieses Verfahren.

Mathematisch gesehen können Sie meist entscheiden, ob ein Programm (die verwendeten Algorithmen etc.) 100-prozentig korrekt ist. Da die Ausführung von Software aber auch noch von anderen Bedingungen abhängt (Hardware, Betriebssystem), kann nicht davon ausgegangen werden, dass sie in Bezug auf diese Bedingungen korrekt ist (Anpassungen sind notwendig). Korrektheit ist somit eher ein theoretisches Maß.

Eine Software ist **robust**, wenn sie beispielsweise bei falschen Eingaben sinnvoll reagiert und bei Eingabefehlern nicht abstürzt. Dies kann von einer einfachen Fehlermeldung bis zu einer automatischen Fehlerkorrektur reichen.



Ein robustes Programm stürzt nicht ab, sondern reagiert auf mögliche Fehler durch eine geeignete Fehlerbehandlung.

Die **Zuverlässigkeit** ist das wichtigste Kriterium von Software bezüglich ihrer Fehlerhaftigkeit. Es kann davon ausgegangen werden, dass jede Software Fehler enthält und diese auch nicht auszuschließen sind. Um zu entscheiden, wie zuverlässig ein Programm ist, müssten die folgenden vier Fragen beantwortet werden:

- ✓ Wie oft tritt ein und derselbe Fehler auf?
- ✓ Wie viele unterschiedliche Fehler gibt es?
- ✓ Sind es Fehler, die bei der Programmierung hätten entdeckt werden müssen? Oder sind es selten auftretende Fehler, die nur schwer zu lokalisieren sind?
- ✓ Führt der Fehler zum Systemabsturz, Verlust von Daten oder zu einem falschen Ergebnis?



Software ist zuverlässig, wenn selten Fehler auftreten und diese nur geringe Auswirkungen haben. Zuverlässige Programme sind auch stets robust.

Benutzerfreundlichkeit

Ist eine Software sowohl von erfahrenen als auch unerfahrenen Benutzern einfach zu bedienen, ist sie **benutzerfreundlich**. Wesentliche Aspekte dabei sind die Entwicklung von Benutzeroberflächen nach **software-ergonomischen Kriterien**, wie beispielsweise der übersichtliche Aufbau und der Inhalt der Bildschirmfenster. Hinweise zur Software-Ergonomie finden Sie in zahlreichen Publikationen. Die ständig verfügbare Hilfefunktion und die Möglichkeit, den Umfang von Funktionen an den Kenntnisstand des Nutzers anzupassen, sind weitere wichtige Gesichtspunkte.

Effizienz

Bei der **Effizienz** (Wirksamkeit und Wirtschaftlichkeit) wird zwischen der Laufzeit- und der Speicherplatzeffizienz unterschieden. Die Grundlage für effiziente Programme sind leistungsfähige Algorithmen und deren Verwendung in der jeweiligen Anwendung. Konzentrieren Sie sich während der Programmierung jedoch zu sehr auf diesen Gesichtspunkt, kann dies zulasten der Übersichtlichkeit gehen. Deshalb sollten Sie bereits eine Vorauswahl für effiziente Algorithmen treffen und diese so implementieren, dass sie einfach durch effizientere austauschbar sind.

Bei der Softwareentwicklung orientieren Sie sich zunächst meist an den anderen Qualitätskriterien. Erst wenn bei Tests der Software eine mangelnde Effizienz festzustellen ist, wird versucht, durch gezieltes Austauschen der Algorithmen eine Verbesserung zu erreichen.

Wartbarkeit

Immens wichtig für die Hersteller von Software ist die **Wartbarkeit**. Wartbarkeit umfasst den Aufwand bei der Fehlersuche, der Fehlerkorrektur und bei funktionalen Erweiterungen, der sich in den anfallenden Kosten widerspiegelt. Für den Benutzer spielt dieses Kriterium in Bezug auf die Softwarequalität jedoch keine direkte Rolle.

Dokumentation, Programmbeschreibung und Programmspezifikation

Softwareprogramme sollten gut dokumentiert werden. Dabei gibt es verschiedene Faktoren, die zu erfassen sind:

- ✓ Benutzerschnittstelle zum Anwender
- ✓ Systemschnittstelle zu anderen Produkten
- ✓ Kommentierter Quelltext für den Hersteller
- ✓ **Dokumentation** für die Anwender und den Hersteller

„Guter“ Programmierstil

Es ist eine Selbstverständlichkeit, dass ein Programm fehlerfrei sein muss und das macht, was es soll. Aber das ist nur eine Facette eines „guten“ Programms. Die Art und Weise, wie der Quellcode geschrieben ist, ist fast genauso wichtig. Eine Sprache wie Python zwingt Programmierer bereits an einigen Stellen zu einem „guten“ Programmierstil, weil sie

- ✓ streng Einrückungen einfordert,
- ✓ Groß- und Kleinschreibung unterscheidet,
- ✓ mehrere redundante oder überflüssige Syntaxelemente weglässt etc.

Doch auch in anderen Programmiersprachen kann man nach einem „guten“ Programmierstil vorgehen. Es gibt ein paar allgemeine Regeln, die man beim Schreiben von Quellcode einhalten sollte und damit auch einer modernen **Softwaredokumentation aus dem Quellcode heraus** Rechnung tragen:

- ✓ Quellcode soll selbsterklärend sein.
- ✓ Die Bezeichner von Variablen und Funktionen sollen für Menschen intuitiv verständlich sein.

- ✓ Wo sich bereits die formale Programmiersprache selbst ausreichend erklärt und die Struktur durch geeignetes Einrücken bereits hinreichend deutlich wird, darf **keine** zusätzliche und unabhängige Beschreibung angefertigt werden.
- ✓ Eine Dokumentation soll so weit wie möglich in den Quellcode eingearbeitet sein. Das kann durch Kommentare und Kommentarzeilen erreicht werden, die in unmittelbarer Nähe der Anweisungen im Quellcode stehen und bei deren Veränderung sofort aktualisiert werden können. Viele Programmiersprachen stellen auch spezielle Dokumentationskommentare zur Verfügung, die automatisch von Dokumentationstools genutzt werden können.
- ✓ Das Einhalten von Konventionen ist extrem wichtig. Das betrifft sowohl die Namensgebung an sich, die Formatierung des Codes, aber auch Groß- und Kleinschreibungen, auch wenn diese von einer Sprache nicht gefordert werden.
- ✓ Man sollte nach dem **DRY**-Prinzip arbeiten (Don't repeat yourself, englisch für „Wiederhole dich nicht“; auch bekannt als *once and only once* – „einmal und nur einmal“). Redundanz ist dabei zu vermeiden oder zumindest zu reduzieren. Das erhöht die Wartbarkeit, denn eventuelle Änderungen brauchen nur an einer Stelle vorgenommen zu werden, und man vermeidet Tipparbeit. Zudem ist so ein Code klarer und weniger fehleranfällig.
- ✓ Konventionen und klare, übersichtliche Formatierungen sollten damit einhergehen, dass Codestrukturen selbst immer möglichst einfach und kurz bleiben. Es gibt eine Regel, die als **KISS**-Prinzip bekannt ist. Das steht für „Keep it simple, stupid“ (englisch für „Halte es einfach und dumm“). Das ist kein Widerspruch zu einer intelligenten Programmierung – im Gegenteil. Denn „intelligent“ bedeutet in der Programmierung einfach und langweilig. Und damit ist die Erstellung des Codes erst einmal wenig fehlerträchtig, aber der Code ist auch leichter wartbar, was mittel- bis langfristig genauso wichtig ist. Einfach bedeutet, dass Sie nicht unnötig komplexe oder lange Strukturen schreiben sollten. Lieber schreiben Sie zwei kurze Funktionen als eine lange Funktion. Wie gesagt, langweilige Programmierung ist fast immer gut. Doch was soll das bedeuten? Als recht leicht merkbare Regel können Sie sich merken, dass Sie immer so programmieren sollten, dass sich jemand Fremdes über Ihren Code nicht wundern würde. Erwartbare Strukturen sind gut, unerwartete Strukturen schlecht. Wenn Sie etwa an einer Stelle eine `while`-Schleife verwenden und in einer vergleichbaren folgenden Situation eine `for`-Schleife, dann ist das unerwartet und schlecht, wenn es keinen unvermeidbaren Grund gibt. Ein erfahrener Programmierer, der den Code sieht, würde sich fragen, warum einmal die eine Schleife und das andere Mal die andere Schleife verwendet wurde. Und dies vollkommen überflüssig, wenn es eben keinen zwingenden Grund gibt. Oder wenn Sie einmal eine Konstante vollkommen groß schreiben und das andere Mal nur den ersten Buchstaben groß schreiben, ist das unerwartet und damit schlecht (was wieder auf die Konventionen zurückkommt). Python erzwingt oder unterstützt zumindest durch seine reduzierten Syntaxstrukturen ohnehin an vielen Stellen die Einhaltung des KISS-Prinzips.

Beschreibung und Dokumentation

Mit einer **Programmbeschreibung** wird bei einer Software beschrieben, was das Programm **macht**. Mit der **Programmspezifikation** wird vorgegeben, was es **machen soll**. Beide Elemente bilden den Kern der **Software-dokumentation**. Diese erklärt eine Software aus unterschiedlicher Sicht. Es gibt in der Regel spezielle Dokumentationen für die Entwickler oder die Anwender, wobei verschiedene Dokumentationsteile nur den Entwicklern zugänglich sind, während andere für die Anwender verfügbar sein müssen.

Man beschreibt darin beispielsweise,

- ✓ wie ein Programm funktioniert (Programmiererdokumentation),
- ✓ wie ein Programm über die Zeit gepflegt wird (Entwicklungsdokumentation),
- ✓ was ein Programm für Ausgaben erzeugt und welche Eingaben notwendig sind (Datendokumentation),
- ✓ wie ein Programm getestet wurde (Testdokumentation),
- ✓ wie ein Programm zu benutzen ist (Benutzerdokumentation),
- ✓ was zum Betrieb eines Programms erforderlich ist (Installationsdokumentation) oder
- ✓ auf welchen Grundlagen die Software entwickelt wurde (Methodendokumentation).

Einige Dokumentationsteile erfüllen neben technischen auch juristische Zwecke und können Vertragsbestandteile im Rahmen eines **Pflichtenhefts** sein. Diese werden auch und im Fall von Gewährleistungsansprüchen berücksichtigt.

Bei der Dokumentation für eine Software werden zwei Arten unterschieden:

- ✓ Systemdokumentation
- ✓ Benutzerdokumentation

Systemdokumentation

Sie beschreibt die Eigenschaften der Software vom Beginn der Entwicklung bis zur letzten Version. Die Systemdokumentation enthält die Definition des Produkts, die Spezifikation der Software, die exakten Beschreibungen der Strukturen. Diese Dokumentation ist sehr detailliert beschrieben und bildet die Grundlage für die Wartung der entsprechenden Software.

Benutzerdokumentation

Hier findet der Anwender ein Handbuch zur Einführung in die Softwarenutzung sowie ein komplettes Nachschlagewerk zu allen angebotenen Funktionen. Sollte es notwendig sein, muss ein gesondertes Handbuch für den Systemadministrator, beispielsweise für die Systeminstallation, in der Benutzerdokumentation integriert sein. Die Nutzungsbedingungen sind ebenso einzufügen wie die Angaben zu den Systemvoraussetzungen, unter denen das Programm effizient arbeitet. Angaben zu den Autoren der Dokumentation bzw. dem Hersteller der Software sind ebenfalls Bestandteil einer guten Dokumentation.

Eine Dokumentation muss klar verständlich sein, der aktuellen Programmversion entsprechen, einen logischen Aufbau besitzen und sachlich präzise geschrieben sein. Ungebräuchliche Bezeichnungen sind in einer Dokumentation zu vermeiden bzw. sollten bei Gebrauch näher erläutert werden.

Zunehmende Qualitätsanforderungen

Software wird heute in vielen Bereichen eingesetzt, in denen ein Ausfall kritisch ist. Softwarefehler können verheerende Folgen haben. In sensiblen Bereichen steht die Forderung nach der Null-Fehler-Software. Trotz Anwendung wissenschaftlicher Ergebnisse bei der Qualitätssicherung können diese Anforderungen (noch) nicht erfüllt werden. Das ergibt sich auch aus der Tatsache, dass umfassende Tests neu entwickelter Software mitunter sehr schwierig und teuer sind.

Zwei Typen der Softwaredokumentation

Zusammenfassend kann die Softwaredokumentation wie folgt unterschieden werden:

- ✓ Die **Projektdokumente** beschreiben, was von den Personen zu tun ist, die an der Entwicklung beteiligt sind, und warum das zu tun ist. Das umfasst beispielsweise die Ziele und Anforderungen, die Methodik, den Zeitrahmen (Planungsdokumente) und die Festlegung, womit die Entwicklung zu erfolgen hat.
- ✓ Die **Systemdokumente** beschreiben, woraus das System besteht, was es tut (Funktionen), was es erzeugt (Ergebnisse), welche Daten es verarbeitet, wie es zu bedienen ist etc.

2.5 Warum programmieren?

Bestehende Programme decken viele Einsatzgebiete ab, wie z. B. die Verwaltung von Geschäftsprozessen, die Textverarbeitung, die Tabellenkalkulation, Spiele usw. Warum sollten Sie noch selbst programmieren?

Standard-Software anpassen

Standardisierte Software, die für ein bestimmtes Einsatzgebiet konzipiert wurde, stößt an ihre Grenzen, wenn spezielle Anforderungen an sie gestellt werden bzw. wenn die Anwendungsgebiete erweitert werden.

Beispiel

Sie haben sich mit einem Programm eine Datenbank mit Ihren Kundendaten aufgebaut. Später müssen Sie auch mit anderen Programmen auf diese Datenbank zugreifen. In diesen Programmen ist keine Möglichkeit vorgesehen, auf die Daten Ihrer Kundendatenbank zuzugreifen. Sie müssten also die Daten wieder manuell eingeben. Oft ist es dann einfacher, wenn Sie ein Programm schreiben, das den Zugriff auf die Daten ermöglicht.

Individual-Software

Hierbei handelt es sich um Software, die eigens für einen bestimmten Anwendungsbereich oder für spezielle Abteilungen innerhalb einer Firma erstellt wird.

Für verschiedene Branchen und Einsatzgebiete gibt es spezielle Anforderungen, sodass nur eine genau auf die Bedürfnisse abgestimmte Software infrage kommt. Individual-Software kann, auf Kundenwunsch hin, erweitert und verändert werden.

2.6 Klassifizierung von Programmiersprachen

Im Gegensatz zu natürlichen Sprachen, z. B. Deutsch, Englisch, gehören **Programmiersprachen** zu den **formalen Sprachen** (künstlichen Sprachen).

Programmiersprachen lassen sich nach verschiedenen Kriterien einordnen. Nach ihrer **historischen Entwicklung** werden verschiedene Generationen unterschieden:

- ✓ Erste Generation: Maschinensprachen (Maschinencode)
- ✓ Zweite Generation: Assembler-Sprachen
- ✓ Dritte Generation: Prozedurale Sprachen
- ✓ Vierte Generation: 4GL (**Generation Language**)
- ✓ Fünfte Generation: Künstliche Intelligenz

Programmiersprachen unterscheiden sich erheblich in der zugrunde liegenden **Programmiertechnik**, auch Programmierparadigma genannt. Nach Programmiertechniken und Konzepten können die Programmiersprachen wie folgt klassifiziert werden, wobei diese Einteilung weder strikt zu trennen noch zwingend ist:

- ✓ Prozedurale und funktionale Programmiersprachen
- ✓ Objektorientierte Programmiersprachen
- ✓ Hybride Programmiersprachen
- ✓ Skriptsprachen
- ✓ Logische Programmiersprachen
- ✓ Erziehungsorientierte Programmiersprachen und Minisprachen

Sprachen lassen sich also nach verschiedenen Kriterien klassifizieren, und zum Teil ist die Zuordnung zu einer bestimmten Gruppe nicht eindeutig. Auch haben sich Klassifizierungen über die Zeit hin und wieder geändert oder werden nicht überall gleich verwendet.

Prozedurale bzw. funktionale sowie logische Programmiersprachen werden auch als **deklarative Programmiersprachen** bezeichnet.



2.7 Die Klassifizierung nach Generationen

Werfen wir einen genaueren Blick auf die Klassifizierung von Programmiersprachen aufgrund ihrer historischen Entwicklung.

Erste Generation: Maschinensprachen (Maschinencode)

Damit ein Computer Probleme lösen kann, muss ihm der Lösungsweg in einer ihm verständlichen Art und Weise mitgeteilt werden. Eine Maschinensprache erfüllt genau diese Bedingung. Intern arbeitet ein Computer mit einem Register aus Befehlen, die zusammen diese Maschinensprache bilden und mit seinem Prozessor verbunden sind.

Beschreibung	Sowohl die Operationen als auch die Daten werden vom Programmierer ausschließlich als Bitfolge aus Nullen und Einsen eingegeben. Eine übliche Operation ist z. B. der Transport von Daten aus dem Speicher in ein Register.
Nachteile	Für jeden Computer müssen die Maschinenbefehle neu entwickelt werden, da diese Sprache von den Eigenschaften der Hardware (Prozessoren) abhängig ist. Programme in Maschinensprache sind schwer lesbar und mit einem hohen Programmieraufwand verbunden. Deshalb wird diese Sprache heute kaum mehr direkt eingegeben (allerdings werden auch heute noch alle Befehle für Computer letztendlich in Maschinenbefehle umgewandelt).
Beispiel	00011010 0011 0100

Zweite Generation: Assembler-Sprachen

Beschreibung	Assembler-Sprachen sind wie Maschinensprachen an bestimmte Prozessoren gebunden. Übersetzungsprogramme, die Assembler-Programme in Maschinencode umwandeln, werden ebenfalls als Assembler bezeichnet.
Einsatz	Assembler-Sprachen werden überwiegend zur Programmierung der Hardware oder für schnelle, zeitkritische Programme eingesetzt.
Vorteile	Im Vergleich zur Maschinensprache bieten Assembler-Sprachen dem Programmierer durch Operationskürzel, z. B. ADD für addieren, wesentliche Erleichterungen. Da Assembler-Sprachen auf die maschinenspezifischen Besonderheiten des jeweiligen Computers abgestimmt sind, verbrauchen die Programme im Allgemeinen weniger Speicherplatz und sind meist auch schneller als ein entsprechendes Programm in einer anderen Programmiersprache.
Besonderheit	Die einzelnen Befehle der Assembler-Sprachen verwenden direkt die internen Befehle des Prozessors. Wer eine Assembler-Sprache erlernt, erfährt dabei viel über die Arbeitsweise des jeweiligen Prozesstyps.
Beispiel	ADD ax, 10

Dritte Generation: Prozedurale Sprachen

Anstoß zur Weiterentwicklung der Programmiersprachen der dritten Generation gaben die mangelhafte Eignung der maschinenorientierten Sprachen zum Erstellen komplexer Anwendungsprogramme und die schlechte Lesbarkeit für Menschen.

Beschreibung	Prozedurale Sprachen sind (weitgehend) unabhängig von einem Computersystem. Da Programmiersprachen ab der dritten Generation vom spezifischen Computersystem abstrahieren, werden sie auch als höhere Programmiersprachen bezeichnet. Damit ein Computer ein Programm in einer höheren Programmiersprache versteht, muss das Übersetzungsprogramm (Compiler oder Interpreter) an das jeweilige System angepasst sein und den entsprechenden Maschinencode erzeugen. Denn der Prozessor muss auch beim Einsatz einer höheren Programmiersprache auf ihn speziell abgestimmten Maschinencode „vorgestellt“ bekommen – nur den versteht ein Prozessor, wie bereits erläutert wurde.
Einsatz	Die Sprachen der dritten Generation sind in ihrer Struktur und ihrem Befehlsvorrat auf bestimmte Anwendungsbereiche zugeschnitten. Allerdings nimmt die Tendenz zu, diese Sprachen immer umfassender auszugestalten.
Vorteile	Höhere Programmiersprachen sind im Allgemeinen leichter zu erlernen als maschinenorientierte Sprachen. Der Programmcode kann auch bei anderen Rechnersystemen wiederverwendet werden – bei einigen (älteren) Sprachen müssen allerdings bei einer Portierung einige Anpassungen vorgenommen werden.
Nachteile	Das Programm der höheren Programmiersprache verbrauchte früher mehr Speicherplatz und war meist auch langsamer als das vergleichbare Maschinenprogramm. Durch optimierte Übersetzung verschwinden diese Nachteile aber bei vielen modernen höheren Programmiersprachen.
Programmiersprachen	Python, Cobol, RPG, Fortran, Pascal, PL/1, Basic, Ada, C/C++, Java und viele mehr
Beispiel (Java)	<code>flaeche = laenge * breite;</code>

Vierte Generation: 4GL (Generation Language)

Während die ersten drei Generationen noch relativ klar voneinander getrennt werden können, fehlen bei der folgenden Generation eindeutige Kriterien.

Beschreibung	Bei 4GL-Programmiersprachen beschreibt der Programmierer lediglich, was das Programm leisten soll, ohne den genauen algorithmischen Weg anzugeben. Eine einzelne Anweisung löst eine ganze Folge von internen Einzelschritten aus.
Einsatz	4GL-Programmiersprachen sind auf spezielle Anwendungsgebiete ausgerichtet, z. B. auf das Bearbeiten und Auswerten von Dateien, Datenbanken, Tabellenkalkulationen oder auf das Erstellen von Bildschirmformularen.
Vorteile	Zum Programmieren stehen fortschrittliche, einfach zu bedienende und leistungsfähige Entwicklungssysteme zur Verfügung. Die Erstellung eines Programms wird dadurch wesentlich erleichtert.
Nachteile	Die größtmögliche Effizienz der Programme ist nicht gegeben. Da bestimmte Folgen von Arbeitsschritten innerhalb einer Anweisung automatisch ausgeführt werden, hat der Programmierer kaum einen Einfluss auf die internen Abläufe. Die Ausführungsgeschwindigkeit dieser Programme ist aufgrund der mächtigeren Sprache langsamer als bei prozeduralen Sprachen.

Programmiersprachen	SQL, Natural, Symphony, Open Access
Beispiel (SQL)	CREATE Adresskartei SELECT Kunde FROM Tabelle WHERE KdNr = 10

Fünfte Generation: Künstliche Intelligenz

Beschreibung	Der Grundgedanke des Forschungsgebietes der künstlichen Intelligenz ist es zu untersuchen, unter welchen Bedingungen Computer menschliche Verhaltensweisen, die auf Intelligenz beruhen, nachvollziehen können. Für diese Forschungszwecke sind einige spezielle Programmiersprachen entwickelt worden. Diese Sprachen gehören meist zu den logischen oder funktionalen Programmiersprachen, die Sie im weiteren Verlauf dieses Kapitels kennen lernen.
Einsatz	Inzwischen umfasst dieses Gebiet mehrere Fachbereiche, beispielsweise die Robotik, die zur Entwicklung von Robotern und deren komplizierten Bewegungsabläufen geführt hat, die Wissensverarbeitung und die Spracherkennung.
Programmiersprachen	Prolog, Lisp, Smalltalk
Beispiel (Prolog)	grossvater(X,Y) :- vater(X,Z), vater(Z,Y).

2.8 Prozedurale und funktionale Programmiersprachen

Als prozedurale Programmiersprachen werden höhere Programmiersprachen bezeichnet, die den Weg zur Problemlösung als eine Folge von Anweisungen angeben, die nacheinander abgearbeitet werden. Anweisungen können dabei auch zusammengefasst werden (in den meisten Sprachen dieser Kategorie als sogenannte **Prozeduren** bezeichnet – daher der Name dieser Art an Programmiersprachen). Und diese Zusammenfassungen (früher oft **Unterprogramme** genannt) können wie integrierte Anweisungen aufgerufen werden. Wer ein Unterprogramm aufruft, wird **Aufrufer** des Unterprogramms genannt.

```

BeginneProgramm
  Anweisung1
  Anweisung2
  Anweisung3
  ...
BeendeProgramm

```

Nun findet man hin und wieder auch den Begriff der „funktionalen Programmiersprache“ im Zusammenhang mit prozeduralen Programmiersprachen, obwohl der Begriff langsam verschwindet. Zwar werden diese beiden Bezeichnungen heutzutage meist synonym verwendet, aber historisch gibt es einen Unterschied, zumal sich Funktionen und Prozeduren im strengen historischen Sinn auch programmiertechnisch unterscheiden. Beide sind zwar Zusammenfassungen von Anweisungen, aber sie verhalten sich etwas unterschiedlich.

Bei funktionalen Programmiersprachen ist das Programm selbst eine Funktion, die sich typischerweise auf einfachere Funktionen als Startkomponente (Wurzel oder root-Element oder auch main-Element) stützt, daher auch der Name „funktionale Programmiersprache“. Das unterscheidet diese Form von Sprachen aber nicht von einer prozeduralen Sprache im engeren Sinn. Im Gegenteil – das ist sogar eine Gemeinsamkeit, weshalb man diese beiden Typen meist eben nicht mehr differenziert, zumal die meisten Sprachen dieser Kategorie sowohl Funktionen als auch Prozeduren bereitstellen.

Die Beziehungen zwischen den Funktionen und/oder Prozeduren untereinander sind einfach:

- ✓ Eine Funktion und/oder Prozedur kann eine andere Funktion und/oder Prozedur aufrufen.
- ✓ Eine Funktion und/oder Prozedur kann das **Ergebnis** einer Funktion (den sogenannten **Rückgabewert**) als Parameter für eine andere Funktion und/oder Prozedur nutzen. Und hier ist der einzige Unterschied zu dem prozeduralen Fall – eine Prozedur liefert **keinen Rückgabewert**. Deshalb kann eine Prozedur nicht als Parameter für eine andere Funktion oder Prozedur genutzt werden.

In einigen alten Programmiersprachen dieser Kategorie musste man bei einer Funktion als Aufrufer den Rückgabewert zwingend verwenden, auch wenn man diesen gar nicht benötigt hatte. Programmierer haben deshalb oft Hilfskonstruktionen erstellt, nur um diesen Rückgabewert „vernichten“ zu können, wenn er nicht benötigt wurde. Moderne Programmiersprachen verzichten auf diesen Zwang.

In dem Buch wird – wie heute üblich – nur noch von prozeduralen Sprachen gesprochen, aber Sie sollten sich merken, dass eine Funktion im engeren Sinn ein Ergebnis liefern wird, das man weiterverwenden kann, während eine Prozedur im engeren Sinn kein Ergebnis an den Aufrufer liefert, sondern die erteilte Aufgabe ohne konkrete Rückmeldung an den Aufrufer erledigt.

Hier sind ein paar Beispiele für prozedurale Sprachen.

Die Programmiersprache COBOL

Bedeutung	COBOL steht für C ommon B usiness O riented L anguage.
Entwicklung	COBOL wurde speziell für betriebswirtschaftliche Anwendungen entwickelt und 1959 eingeführt. 1968 wurde COBOL durch das American National Standard Institute (ANSI) genormt. Seitdem wurde es kontinuierlich erweitert und modifiziert. Der letzte gültige Stand ist Cobol 2002. Diese Version enthält objektorientierte Erweiterungen.
Einsatz	COBOL ist eine noch immer weit verbreitete Programmiersprache. Im Bereich der kaufmännischen Großrechner sind weit über die Hälfte aller Anwendungen in COBOL geschrieben. Selbst auf dem PC werden betriebswirtschaftliche Anwendungen, die auf Großrechnerdaten zugreifen, in COBOL entwickelt.

Die Programmiersprache BASIC/Visual Basic

Bedeutung	BASIC steht für B eginners A ll-Purpose S ymbolic I nstruction C ode.
Entwicklung	BASIC wurde 1965 für Schulungszwecke entwickelt. Die Weiterentwicklung von BASIC führten viele Hersteller allerdings im Alleingang durch, weshalb für viele Rechnertypen eine eigene BASIC-Version entstand. 1991 wurde BASIC zu Visual Basic weiterentwickelt und ab 2002 auch in das .NET Framework, eine Entwicklungsumgebung für Windows-basierte Anwendungen von Microsoft, integriert. Dabei kann Visual Basic auch für objektorientierte Programmierung eingesetzt werden (eine hybride Sprache).
Einsatz	Haupteinsatzbereich für BASIC waren Mikrocomputer. Die objektorientierte Programmiersprache Visual Basic 2008 wird zur Programmierung von Windows-basierten Anwendungen verwendet.

Die Programmiersprache C

Entwicklung	Die Programmiersprache C wurde Anfang der 70er-Jahre von B. W. Kernighan und D. M. Ritchie im Auftrag der Bell Laboratories während der Arbeit an dem Betriebssystem UNIX entwickelt.
Einsatz	C eignet sich für die Entwicklung von systemnahen Programmen. So ist beispielsweise das Betriebssystem UNIX in C geschrieben worden. Die letzte Variante der Sprache ist C99 von 1999.
Hinweise	C hat alle Vorteile einer höheren Programmiersprache. Gleichzeitig kann damit sehr hardwarenah programmiert werden, was sonst nur bei Assembler-Sprachen möglich ist. Bedingt durch die Normierung von C durch die ANSI-Kommission, können Programme relativ leicht auf andere Rechnertypen übertragen (portiert) werden, sofern sich die Programmierer an die normierten Konventionen halten.

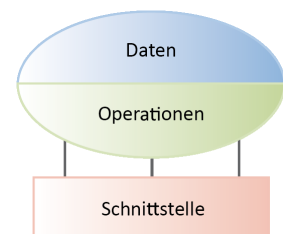
Programme werden wie mathematische Funktionen geschrieben. Eine Funktion hat einen Definitions- und einen Wertebereich. Die Funktion erhält einen Eingabewert und berechnet, mathematisch gesehen, den Wert der Funktion.

2.9 Objektorientierte Programmiersprachen

Die Weiterentwicklung der Computer führte auch zur Entwicklung komplexerer Software. Entsprechend wurde die prozedurale Programmierung zur objektorientierten Programmierung weiterentwickelt. Das Problem der prozeduralen Programmierung ist, dass globale Daten in jedem Teil des Programms manipuliert und überschrieben werden können. Es fehlt eine Verbindung zwischen den Daten eines Programms und den sie manipulierenden Funktionen. Dies führt dazu, dass große Programme sehr leicht unübersichtlich werden und sich schwerer testen lassen. Die Weiterentwicklung der Erkenntnisse und Erfahrungen der strukturierten Analyse führten zur objektorientierten Programmierung (**OOP**).

Was bedeutet objektorientiertes Programmieren?

1970 erkannte David Parnas das Problem und hatte die Idee, jedes einzelne Datenelement in einem Modul zu kapseln. Der direkte Zugriff auf diese Daten wurde nur über eine bestimmte Schnittstelle mit einem Satz von Operationen, wie z. B. über Prozeduren oder Funktionen, erlaubt. Sollen andere Module ebenfalls auf die Variable zugreifen, können sie dies nur indirekt über die Schnittstelle des Moduls tun.

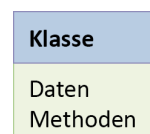


Kernstück der OOP sind sogenannte **Klassen**. Klassen sind Datentypen, die sich an der realen Welt orientieren. In einem Programm, das beispielsweise eine Tabellenkalkulation realisiert, gibt es dementsprechend Klassen, mit denen z. B. ein Tabellenblatt, eine Zeile, eine Spalte oder ein Diagramm beschrieben werden können.

Um die entscheidenden Ziele bei der Entwicklung großer Softwaresysteme, wie z. B. Produktivitäts- und Qualitätssteigerung, zu realisieren, werden bei objektorientierten Programmiersprachen die Wiederverwendbarkeit, die Erweiterbarkeit und die Kompatibilität in den Vordergrund gestellt. Objektorientierte Programmiersprachen bieten Konzepte wie Abstraktion, Datenkapselung, Modularität und Hierarchieprinzipien, um diese Ziele zu erreichen.

Klassen und Objekte

In Klassen werden Daten und Methoden zusammengefasst. Methoden dienen dem Zugriff auf bzw. Verändern von Daten in der Klasse.



Aus Klassen können **Objekte** erzeugt werden. Objekte sind sogenannte **Instanzen** einer Klasse. Der Aufbau und die Funktionsweise eines Objekts werden durch die Daten und Methoden der Klasse festgelegt.

- ✓ Die **Daten** einer Klasse entsprechen den Daten, die ein Objekt aufnehmen kann. Sie sind von einem bestimmten Datentyp, der elementar, strukturiert oder wiederum ein Objekt sein kann (Referenztyp). Wird eine Instanz der Klasse gebildet, besitzt jedes Objekt diese Daten. Die Werte können aber bei jedem Objekt unterschiedlich sein. Auf die Daten kann in der Regel von außen nicht direkt zugegriffen werden, sondern nur über die entsprechenden Methoden (**Datenkapselung**). Daten beschreiben die Eigenschaften von Klassen bzw. Objekten. Diese Daten werden auch **Attribute** genannt.
- ✓ Eine **Methode** (Operation) ist wie eine Prozedur oder Funktion aufgebaut (also eine Art Unterprogramm). Sie wird innerhalb einer Klasse definiert. Methoden können auf die Daten eines Objekts dieser Klasse zugreifen. Methoden beschreiben das Verhalten von Klassen bzw. Objekten und stehen nur über die Klasse bzw. ein Objekt der Klasse zur Verfügung.



Objekte können einen spezifischen **Zustand** besitzen, der die Verfügbarkeit prinzipiell vorhandener Methoden beeinflussen kann. So kann eine Person etwa nicht heiraten, wenn sie bereits verheiratet ist.

Die Programmiersprache C++

Entwicklung	1982 begann Bjarne Stroustrup eine Erweiterung der prozeduralen Programmiersprache C zu entwickeln. 1989 wurde die Basissprache definiert, 1996 der internationale Standard (ISO/IEC 14882) verabschiedet. Die letzte Überarbeitung der Sprache wurde 2003 veröffentlicht.
Einsatz	C++ eignet sich für die Entwicklung von systemnahen Programmen und von komplexen Anwendungen.
Hinweise	Es gibt zur objektorientierten Programmierung mit C++ mächtige Programmierwerkzeuge. C++-Compiler stehen praktisch auf jeder Rechnerplattform zur Verfügung. C++ besitzt mächtige Sprachmittel, um komplexe Anwendungen zu erzeugen. Außerdem stehen umfangreiche Klassenbibliotheken zur Verfügung.

Die Programmiersprache Java

Entwicklung	Eine Gruppe von Ingenieuren bei Sun Microsystems entwickelte 1991 Software für interaktives Fernsehen und andere Geräte der Konsumelektronik. Diese Programmiersprache nannte sich „Oak“. Mit der zunehmenden Verbreitung des Internets wurde Oak dafür angepasst und in Java umbenannt. Java wurde an C/C++ angelehnt, wobei auf verschiedene (fehlerträchtige) Konstrukte verzichtet wurde.
Einsatz	Anwendungen, die auf unterschiedlichen Rechnersystemen laufen sollen, sowie für verteilte Anwendungen
Hinweise	Da nicht alle Sprachelemente von C++ realisiert sind, ist Java eine relativ schlanke und übersichtliche Sprache. Java-Compiler sind für Windows und Linux/Unix sowie MacOS kostenlos erhältlich und erzeugen maschinenunabhängigen Code, sogenannten Bytecode. Dieser wird in einer virtuellen Maschine ausgeführt. Java besitzt eine umfangreiche Programmbibliothek für verschiedene Bereiche.

.NET und die Programmiersprache C#

Entwicklung	Mit .NET bezeichnet Microsoft seine Software-Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen. C# ist eine objektorientierte Programmiersprache für .NET. C# greift Konzepte der Programmiersprachen Java, C++, C und Delphi auf und wurde von Microsoft erstmals im Jahr 2000 standardisiert.
Einsatz	C# eignet sich besonders für eine bequeme Erstellung von sicheren, stabilen und gut an die Zielplattform angepassten Windows-Programmen. Aber auch für andere Betriebssysteme (etwa Linux) gibt es Laufzeitumgebungen.
Hinweise	.NET besteht aus einer Laufzeitumgebung (Common Language Runtime – CLR), in der die Programme ausgeführt werden, sowie einer Sammlung von Klassenbibliotheken, Programmierschnittstellen und Dienstprogrammen (analog Java und seiner virtuellen Maschine). .NET unterstützt neben C# die Verwendung weiterer Programmiersprachen. Unabhängig von der verwendeten Programmiersprache werden .NET-Programme in eine Zwischensprache (Common Intermediate Language – CIL) übersetzt, bevor sie von der Laufzeitumgebung ausgeführt werden.

2.10 Hybride Programmiersprachen und Skriptsprachen

Die strenge Aufteilung in prozedurale, funktionale oder objektorientierte Sprachen ist bei vielen modernen Sprachen nicht möglich oder sinnvoll. Solche Sprachen können oft sowohl prozedural als auch objektorientiert eingesetzt werden. Man nennt diese Sprachen deshalb oft auch **hybrid**. Können hybride Sprachen mit Objekten umgehen, stellen jedoch nicht den vollen Umfang von objektorientierten Sprachen bereit, werden sie **objektbasiert** oder **objektbasierend** genannt. Gerade sogenannte Skriptsprachen sind oftmals sowohl objektbasierend als auch hybrid. Skriptsprachen sind in der Regel von einfacherer Natur und werden meist über einen Interpreter ausgeführt, der die Anweisungen erst zur Laufzeit in Maschinenbefehle übersetzt.

Programme, die in Skriptsprachen geschrieben sind, werden Skripte (engl. scripts) genannt. Teilweise wird auch die Bezeichnung **Makro** verwendet.

Die Anwendungsgebiete und Eigenschaften konventioneller Programmiersprachen und Skriptsprachen überschneiden sich mittlerweile stark, weshalb eine strikte Trennung nur selten möglich ist.

JavaScript

Entwicklung	<p>JavaScript ist eine Skriptsprache, die ursprünglich von der Firma Netscape für die Erweiterung von HTML in Webbrowsern entwickelt wurde, um Benutzerinteraktionen auszuwerten und dynamisch Inhalte der Webseite zu verändern. Im Jahr 1995 erschien mit dem Netscape Navigator 2.0 der erste Browser mit einer eingebetteten Skriptsprache, die zu diesem Zeitpunkt LiveScript genannt wurde.</p> <p>Im Rahmen einer Kooperation zwischen Netscape und Sun Microsystems wurde LiveScript in JavaScript umbenannt. Mittlerweile unterstützen alle modernen Webbrowser JavaScript, wobei Microsoft einen eigenen Dialekt mit Namen JScript verwendet und Anwender JavaScript im Browser auch deaktivieren können.</p>
Einsatz	<p>JavaScript wurde früher fast ausschließlich clientseitig im Webbrowser eingesetzt. Heutzutage finden Sie JavaScript-Implementierungen aber auch auf Servern, im Umfeld von Java oder .NET, in Datenbanken, Microcontrollern oder bei Apps.</p>
Hinweise	<p>ECMAScript (ECMA 262) bezeichnet den standardisierten Sprachkern von JavaScript. Die Syntax von JavaScript basiert auf C und stimmt in großen Bereichen mit der von Java überein.</p> <p>Aber trotz der Namens- und syntaktischen Ähnlichkeit hat JavaScript konzeptionell nur geringe Gemeinsamkeiten mit Java. JavaScript ist nicht streng objektorientiert, sondern objektbasierend und verwendet etwa bei der Vererbung Prototypen statt Klassen. Man kann mit JavaScript sowohl prozedural als auch rein objektorientiert arbeiten, wenn man dabei gewisse Konventionen einhält, die aber nicht zwingend sind.</p>

Python

Entwicklung	<p>Python ist eine universelle, höhere Programmiersprache, die üblicherweise interpretiert wird. Es gibt also in der Laufzeitumgebung von Python einen Interpreter samt notwendiger weiterer Ressourcen. Python unterstützt sowohl die objektorientierte, die aspektorientierte, die strukturierte als auch die funktionale Programmierung. Das bedeutet, Python zwingt den Programmierer nicht zu einem einzigen Programmierstil, sondern erlaubt, das für die jeweilige Aufgabe am besten geeignete Paradigma zu wählen. Objektorientierte und strukturierte Programmierung werden vollständig unterstützt, funktionale und aspektorientierte Programmierung werden zumindest durch einzelne Elemente der Sprache unterstützt. Ein zentrales Feature ist in Python die dynamische Typisierung samt dynamischer Speicherbereinigung. Damit kann man Python auch als reine Skriptsprache nutzen.</p>
Einsatz	<p>Das Einsatzgebiet von Python ist breit gestreut. Man kann damit so gut wie jede Form von Applikation schreiben, und es gibt für die wichtigsten Betriebssysteme Implementierungen.</p>

Hinweise	<p>Python hat seine Grundlagen in C und ist mit den meisten gängigen Programmiersprachen verwandt, wurde aber mit dem Ziel größter Einfachheit und Übersichtlichkeit entworfen. Zentrales Ziel bei der Entwicklung der Sprache ist die Förderung eines gut lesbaren, knappen Programmierstils. So wird beispielsweise der Code nicht durch geschweifte Klammern (wie in fast allen anderen C-basierenden Sprachen), sondern durch zwingende Einrückungen strukturiert. Zudem ist die gesamte Syntax reduziert und auf Übersichtlichkeit optimiert.</p> <p>Wegen dieser klaren und überschaubaren Syntax gilt Python als einfach zu erlernen, zumal die Sprache mit relativ wenigen Schlüsselwörtern auskommt. Es wird immer wieder zu hören sein, dass sich Python-basierte Skripte deutlich knapper formulieren lassen als in anderen Sprachen. In vielen Ländern hat Python an Universitäten bei Anfängerkursen in Informatik-bezogenen Studiengängen Java abgelöst, was über viele Jahre die Szene beherrscht hatte und im professionellen Umfeld immer noch das Maß aller Dinge darstellt.</p>
----------	--



Bei den konkreten Programmcode-Beispielen in diesem Buch wird wie schon erwähnt überwiegend Python verwendet. Diese Programme können aber im Prinzip mit jeder modernen Programmiersprache umgesetzt werden. Python wird im Buch überwiegend prozedural eingesetzt, aber die entscheidenden Abschnitte – die konkreten Strukturen und Algorithmen – sind von dem Paradigma unabhängig und können auch in einer rein objektorientierten Sprache wie Java oder C# nachvollzogen werden.

Da Python die Basis der meisten Beispiele im Buch darstellt, sollen wichtige Fakten zu Python hier noch vertieft werden.

Was umfasst Python?

Es gibt einmal die **Sprache** Python, die aus den Schlüsselworten, Operatoren, eingebauten Funktionalitäten und Funktionen (Built-in Functions) etc. sowie einer eigenständigen Syntax besteht. Python besitzt zudem eine umfangreiche Standardbibliothek (**API** – Application Programming Interface, was auf Deutsch Programmierschnittstelle oder genauer eine Schnittstelle zur Anwendungsprogrammierung bedeutet) und zahlreiche Pakete im **Python Package Index**, bei deren Entwicklung ebenfalls großer Wert auf Überschaubarkeit, aber auch eine leichte Erweiterbarkeit gelegt wurde. Python-Programme lassen sich deshalb auch in anderen Sprachen als Module einbetten. Umgekehrt lassen sich mit Python Module und Plug-ins für andere Programme schreiben, die die entsprechende Unterstützung bieten.

Die verschiedenen Python-Paradigma

Python unterstützt sowohl die objektorientierte, die aspektorientierte, die strukturierte als auch die funktionale Programmierung. Das bedeutet, Python zwingt den Programmierer nicht zu einem einzigen Programmierstil, sondern erlaubt, das für die jeweilige Aufgabe am besten geeignete Paradigma zu wählen. Ein zentrales Feature ist in Python die dynamische Typisierung samt dynamischer Speicherbereinigung. Damit kann man Python auch als reine Skriptsprache nutzen.

Die Evolution von Python

- ✓ Python wurde Anfang der 1990er-Jahre von Guido van Rossum als Nachfolger für die Sprache ABC entwickelt.
- ✓ Ursprüngliche Zielplattform war das verteilte Betriebssystem Amoeba.
- ✓ Die erste Vollversion erschien im Januar 1994.
- ✓ Python 2.0 erschien Oktober 2000.
- ✓ Python 3.0 (auch Python 3000) erschien Dezember 2008. Da Python 3.0 teilweise inkompatibel zu früheren Versionen ist, wird Python 2.7 derzeit parallel zu Python 3 weiter durch Updates unterstützt.
- ✓ Die Serie 3 ist Mitte 2018 immer noch aktuell (derzeit 3.6).

2.11 Logische Programmiersprachen

Bei logischen Programmiersprachen wird kein Algorithmus zum Lösen eines Problems angegeben, sondern es werden lediglich die Bedingungen für eine korrekte Lösung bestimmt. Stellen Sie dann eine Frage, erhalten Sie aufgrund dieser Wissensbasis eine entsprechende Antwort.

Die Programmiersprache PROLOG

Bedeutung	Der Name PROLOG ist eine Abkürzung für „ PRO gramming in LOG ic“.
Entwicklung	Die Programmiersprache PROLOG wurde 1970 von Alain Colmerauer und Philippe Roussel entwickelt. PROLOG hat sich in den 80er-Jahren verbreitet, jedoch wurde durch die fehlende Entwicklung von Anwendungen die weitere Verbreitung behindert.
Einsatz	Dient als experimentelles Werkzeug für künstliche Intelligenz, wird jedoch hauptsächlich an Universitäten benutzt
Beispiel	<pre> Links_von(Apfel, Birne). Factum Links_von(Pflaume, Apfel). Factum Links_von(X, Y) :- Rechts_von(Y, X). Regel ... Frage: ?-Rechts_von(X, Apfel). Frage Antwort: X = Pflaume Antwort </pre>

2.12 Erziehungsorientierte Programmiersprachen und Minisprachen

Erziehungsorientierte Programmiersprachen und Minisprachen sind als Lerninstrumente für Programmieranfänger und teils sogar Kinder konzipiert, die spielerisch mit der Funktionsweise und den Prinzipien der Programmierung vertraut machen sollen.

Solche Sprachen sind nicht zwingend auf Kinder beschränkt, sondern können sowohl für Jugendliche als auch für die Erwachsenenbildung sinnvoll sein. Ziel vieler Sprachen ist die Erstellung visuell interessanter Anwendungen und Oberflächen ohne „echte“ Programmierung. Andere erziehungsorientierte Konzepte agieren als virtuelle Roboter oder Umgebungen. Dabei werden Objekte wie ein Roboter, ein Hamster oder eine Schildkröte in einer Umgebung gesteuert. Das kann mit einer „erwachsenen“ Programmiersprache im Umfeld einer virtuellen Umgebung erfolgen, aber auch mit vereinfachten Techniken.

Erziehungsorientierte Programmiersprachen sind meist mit integrierten Entwicklungsumgebungen (IDE) verknüpft, um Programme möglichst einfach und menügeleitet zu erstellen. Manchmal wird sogar ganz auf die Eingabe von Text durch den Anwender verzichtet und mit Symbolen gearbeitet, um Kinder mit den Grundzügen der Programmierung vertraut zu machen, ohne die Syntax einer Programmiersprache lernen zu müssen.

Die Programmiersprache Logo

Entwicklung	Als die älteste kindgerechte Programmiersprache gilt die von Seymour Papert 1967 auf der Grundlage von Lisp entwickelte Programmiersprache Logo .
Einsatz	Mit Logo wird die Turtle-Grafik erstellt, bei der eine virtuelle Schildkröte über den Bildschirm kriecht und dabei eine farbige Linie hinter sich herzieht. In einigen Ländern wird Logo noch heute in Schulen eingesetzt.

Die Programmiersprache Scratch

Entwicklung	Im Jahr 2007 wurde Scratch veröffentlicht. Mittlerweile gibt es die Version 2.0. Entwickelt wurde Scratch unter der Leitung von Mitchel Resnick am MIT Media Lab. Die ersten Implementierungen der Scratch-Entwicklungsumgebung basierten auf Squeak. Den folgenden Scratch-Web-Player gibt es auf Basis von Java oder Flash. Scratch 2.0 erschien im Mai 2013 und basiert komplett auf Flash.
Einsatz	Scratch soll Anfänger – besonders Kinder und Jugendliche – motivieren, die Grundkonzepte der Programmierung spielerisch und über interessante Experimente zu erlernen. Die Programmiersprache verwendet eine visuelle Entwicklungsumgebung, die Wert auf eine intuitive Bedienung und leichte Übersetzbarkeit legt und vorgefertigte Bausteine zur Programmierung bietet.



Zur Verdeutlichung verschiedener Konzepte in diesem Buch kann Scratch eine sinnvolle Ergänzung sein. Unter <https://scratch.mit.edu/> finden Sie die wichtigsten Informationen zu Scratch inklusive der Entwicklungsumgebung, die Sie direkt aus einem Browser heraus ausführen können.

2.13 Webprogrammierung und mobile Apps

In den letzten Jahren hat die Entwicklung von Programmen eine massive Veränderung erfahren. Bis vor wenigen Jahren konnten Programme überwiegend in Desktop- und Großrechnerprogramme eingeteilt werden. Mittlerweile sind zwei weitere Programmarten von Bedeutung: interaktive Web-Applikationen (RIAs) und mobile Apps.

Interaktive Web-Applikationen (RIAs)

Seit Anfang der 1990er-Jahre hält das Internet Einzug in das tägliche Leben, und die Bedeutung wächst stetig. Dabei nutzen Anwender neben **E-Mail** hauptsächlich das World Wide Web (**WWW**) über einen Browser. Im Browser werden Webseiten angezeigt, die von einem Webserver geladen werden. Die Webseiten werden fast ausschließlich mit der Dokumentenbeschreibungssprache **HTML (Hypertext Markup Language)** erstellt. Das Layout bei modernen Webseiten wird mittels Formatvorlagen (Style Sheets) festgelegt. Im WWW kommen hauptsächlich **CSS (Cascading Style Sheets)** als Formatvorlagen zum Einsatz. Standardisiert werden die Techniken für das WWW vom W3C (<http://www.w3.org>).

HTML ist **explizit keine Programmiersprache**. Im Laufe der Jahre hat sich das WWW jedoch zu einem System entwickelt, in dem Anbieter von Inhalten sowohl auf dem Webserver als auch dem Client (Browser) programmieren können. Im Client werden dazu HTML-Seiten durch **JavaScript** erweitert, während auf dem Webserver sehr oft **PHP**, aber auch Java oder Sprachen aus dem .NET-Umfeld zum Einsatz kommen. Das einfache Konzept des WWW mit der Anforderung von Webseiten führte durch die Entwicklung von Webseiten mit interaktiven Inhalten zu Problemen. Bei der Anfrage eines Browsers an einen Webserver nach neuen Daten muss dieser immer eine **vollständige Webseite** als Antwort senden. Oder genauer – der Browser versteht die Antwort so, dass er die bisher im Browser angezeigte Seite durch diesen neuen Inhalt vollständig ersetzt. Diese Vorgehensweise ist aufwändig, vor allem wenn nur kleine Änderungen notwendig sind.

Es wurde ein Verfahren benötigt, um eine Reaktion bei einer interaktiven Webseite in (nahezu) Echtzeit zu gewährleisten, obwohl neue Daten vom Webserver angefordert werden. **Ajax (Asynchronous JavaScript and XML)** wurde entwickelt. Ändern sich Daten einer im Browser vorhandenen Webseite, führt eine Ajax-Datenanfrage dazu, dass nur die neuen Daten vom Webserver geschickt werden und diese dann mit JavaScript in die bereits beim Client geladene Webseite „eingebaut“ werden. Dabei wird in der Regel die Interaktion des Benutzers mit der Webseite durch das Laden neuer Daten nicht unterbrochen. Dieses Verfahren erlaubt nun auch im Web die Erstellung von Angeboten, die sehr stark auf Interaktion mit dem Anwender setzen. **Web 2.0** wird als Oberbegriff für die meisten interaktiven Webangebote verwendet. Interaktive Webangebote werden als **RIAs (Rich Internet Applications)** bezeichnet und sind – im Gegensatz zu statischen Webseiten ohne Programmierung – Programme bzw. Applikationen (Web-Applikationen).

Mobile Apps

Mobile Endgeräte und die darauf laufenden Anwendungen (Programme) – sogenannte **Apps** als Abkürzung von „Applications“ – gehören mittlerweile zum Alltag. Mobilität zeigt sich vielfältig, in Form von Handys, Smartphones oder Tablets über E-Book-Reader bis hin zu Netbooks, Ultrabooks und Notebooks. Die Grenzen der Gerätetypen wie auch der mobilen Apps zu klassischen Desktopanwendungen verschwimmen. Bei mobilen Anwendungen ist eine permanente Verbindung mit dem Internet mehr oder weniger ein Muss. Die Anwendungen (Apps) müssen den Besonderheiten mobiler Endgeräte Rechnung tragen.

Sind Netbooks, Ultrabooks oder Notebooks dabei noch „echte“ Computer, die mit Tastatur und Maus bedient werden können, müssen E-Book-Reader, Smartphones, Tablets oder gar Smart-Watches meist ohne solche externen Eingabegeräte auskommen. Die Eingabe bei kleinen Geräten wird auf wenige Hardwaretasten, Touchscreens, Sensoren und Spracheingabe übertragen. Dazu werden spezifische Designs der Benutzeroberflächen benötigt. Mobile Geräte besitzen oft spezifische Hardware, die bei stationären Geräten oder klassischen Computern nicht vorhanden ist. Besondere Hardware Sensoren, z. B. zum Erkennen der Orientierung des Bildschirms oder des Standortes eines Benutzers, unterstützen die mobile Verwendung.

Im mobilen Bereich werden viele verschiedene Techniken eingesetzt. Dies betrifft die Hardware ebenso wie die Bedienkonzepte, die verwendeten Programmiersprachen und die Betriebssysteme.

In der Praxis nutzen die meisten mobilen Endgeräte entweder Android oder iOS als Betriebssystem.



Native Programmierung von mobilen Apps bedeutet, dass Sie sich auf eine Hardware samt speziellem Betriebssystem oder nur eine spezielle Version davon konzentrieren. Dazu verwenden Sie klassische Programmiersprachen wie **Swift** (Regelfall unter iOS, wobei auch Objective-C gelegentlich noch verwendet wird) oder **Java** (meist unter Android eingesetzt).

Wollen Sie mehrere Systeme unterstützen, müssen Sie ggf. für jedes Zielsystem eine eigene App erstellen. Alternativ können Sie mit klassischen Web-Technologien (HTML, JavaScript, CSS) als Basis einer App arbeiten und damit weitgehend neutral von verschiedenen Zielplattformen. Insbesondere HTML5 und CSS3 werden in Verbindung mit JavaScript mittlerweile von allen relevanten Herstellern mobiler Endgeräte unterstützt. Dabei sind Sie erst einmal auf den jeweilig verfügbaren eingebetteten Browser beschränkt und können nicht auf die gesamte mobile Hardware zugreifen. Es gibt aber auch Umgebungen wie Cordova (<https://cordova.apache.org/>), um darüber auf die mobile Hardware zugreifen zu können und auch mit Web-Technologien „echte“ Apps zu erstellen.

2.14 Übungen

Übung 1: Fragen zu Grundlagen der Datenverarbeitung

Übungsdatei: --

Ergebnisdatei: *uebung02.pdf*

1. Was ist ein Programm? Was ist Software?
2. Beschreiben Sie das EVA-Prinzip.

Übung 2: Fragen zu Programmiersprachen

Übungsdatei: --

Ergebnisdatei: *uebung02.pdf*

1. Nach welchen Kriterien können Programmiersprachen klassifiziert werden?
2. Gehören Assembler-Sprachen zu den höheren Programmiersprachen? Begründen Sie Ihre Antwort.

3. Ordnen Sie Programmier Techniken und Programmiersprachen zu:

Programmiersprachen	prozedural	objektorientiert	logisch
Lisp			
C#			
C++			
Java			
Cobol			

Übung 3: Modularisierung und Zerlegung in Teilprobleme

Übungsdatei: --

Ergebnisdatei: *uebung02.pdf*

Ein Bankkunde kommt zu einem Bankautomat und möchte Geld abheben. Dazu muss er über eine Bankkarte verfügen und sich legitimieren können. Der Vorgang kann in einzelne Teilprobleme zerlegt werden.

1. Identifizieren Sie die möglichen Teilprobleme bzw. Schritte. Beschreiben Sie die Teilprobleme und machen Sie sich Gedanken über mögliche Schritte bei fehlerhaften Eingaben durch den Anwender – sowohl in Hinsicht der Legitimierung als auch der auszuzahlenden Summe. Zerlegen Sie das Problem in die identifizierten Teilprobleme.
2. Beschreiben Sie die Verbindungen zwischen den Teilproblemen. Wie folgen die einzelnen Schritte aufeinander?
3. Wie könnten die identifizierten Module der Teilprobleme in anderen Vorgängen verwendet werden?

Übung 4: Fehler in einem Algorithmus finden und verbessern

Übungsdatei: --

Ergebnisdatei: *uebung02.pdf*

Ein Programm soll alle geraden Zahlen zwischen 0 und 100 auf dem Bildschirm ausgeben. Dazu soll folgender Algorithmus in Pseudocode verwendet werden, der eine Schleife simulieren soll.

Beginn des Algorithmus

Teste, ob der Wert der Zahl, die in der Variablen i steht, gerade ist

Wenn ja, dann Ausgabe Wert der Zahl i

Wenn nein, dann keine Ausgabe

Gehe wieder zum Test, solange die Variable i nicht den Wert 100 erreicht

Ende des Algorithmus

1. Identifizieren Sie die möglichen Fehler.
2. Korrigieren Sie die Fehler.

3 Programmlogik und Darstellungsmittel für Programmabläufe

In diesem Kapitel erfahren Sie

- ✓ was eine Abstraktion ist
- ✓ was Programmlogik und Programmablauf bedeuten
- ✓ wie Sie einen Programmablauf visualisieren können
- ✓ was ein Programmablaufplan, ein Datenflussdiagramm und ein Struktogramm sind
- ✓ wie Sie Pseudocode schreiben
- ✓ wie Sie Entscheidungstabellen verwenden können

3.1 Abstraktion der Wirklichkeit

Viele Problemstellungen sind umfangreich und kompliziert. Deshalb erfordern sie eine systematische Vorarbeit, um den Lösungsweg per EDV-System zu beschreiben. Zentral ist, dass man dabei immer nur eine **Abstraktion** der Wirklichkeit umsetzt. Das bedeutet einfach nur, dass man sich auf die Dinge beschränkt, die wirklich für ein Problem relevant sind, und Nebensächlichkeiten sowie unwichtige Dinge ausdrücklich weglässt. Das wird ja z. B. im Konzept des „Computational Thinking“ im ersten Schritt gefordert.

Wenn Sie etwa ein Programm zur Verwaltung eines Bankkontos erstellen, werden Sie zwar beispielsweise den Namen und die Adresse einer Person benötigen, aber nicht deren Hobbies oder Haarfarbe. Sie abstrahieren also die Wirklichkeit und reduzieren sie auf diese Dinge, die in einem speziellen Problem relevant sind.

3.2 Programmlogik und Programmablauf

Der Programmablauf oder Programmfluss beschreibt die Umsetzung eines Algorithmus in einem Programm, das dazu die Folge von Operationen zur Lösung einer Aufgabe durchläuft. Diese Schritte der Folge können dabei auf verschiedene Weisen aufeinanderfolgen und von einer Programmlogik abhängen. Die verschiedenen Arten der Schritte treten in komplexeren Programmen in der Regel in verschiedenen Formen auf.

- ✓ **Sequenziell:** Dabei folgt einfach eine Operation nach der nächsten (eine **Sequenz**), ohne dass sich die Folge der Schritte jemals verändert. Sequenzen treten in fast jedem Programm auf, aber rein sequenziell abgearbeitete Programme sind eher selten und meist nur beim standardisierten Verarbeiten von Arbeitsschritten (ohne Benutzerinteraktion) zu finden. Etwa bei der immer gleichen Verarbeitung großer Datenmengen (Datensicherungen zum Beispiel) oder beim Start eines Betriebssystems.
- ✓ **Iterativ:** Das bedeutet das Wiederholen von Schritten in Form von einer Schleife (**Iteration**). In dem Fall muss aber in der Regel vor jeder Wiederholung geprüft werden, ob eine Wiederholung sinnvoll bzw. gewünscht ist oder nicht. Falls nicht, muss die Iteration beendet werden. Diese Prüfung (meist Bedingung der Schleife genannt) ist Teil der Programmlogik und der Programmablauf ändert sich je nach Ergebnis der Prüfung. Die Prüfung der Bedingung wird oft mittels sogenannter Operatoren umgesetzt. Man kann beispielsweise überprüfen, ob der Wert einer Variablen mit einem Vergleichswert übereinstimmt oder auch größer oder kleiner ist. In Abhängigkeit davon wird der Programmablauf in die eine oder andere Richtung fortgesetzt.

Bedingungen können auch kombiniert werden. Es müssen also beispielsweise mehrere Bedingungen erfüllt sein (eine Und-Verknüpfung) oder aber nur eine von mehreren Bedingungen (eine Oder-Verknüpfung). So kann eine Schleife etwa verlassen oder die Anweisungen der Schleife wiederholt werden.

- ✓ Auswählend auf Grund von Entscheidungen: Mittels Entscheidungsanweisungen wird der Programmfluss eine von mehreren Alternativen auswählen. Auch hier kommt Programmlogik in Form einer Prüfung (meist Bedingung der Entscheidungsanweisung genannt) zum Einsatz und der Programmablauf ändert sich je nach Ergebnis der Prüfung. Wie bei Schleifen wird die Prüfung der Bedingung oft mittels Operatoren umgesetzt. Man kann beispielsweise überprüfen, ob der Wert einer Variablen mit einem Vergleichswert übereinstimmt und dann den einen Zweig im Programmfluss auswählen und im anderen Fall den anderen Zweig.
- ✓ Gezielt im Programmfluss springend in Form von Sprunganweisungen: Damit springt man gezielt eine Stelle im Programmfluss an, die meist nicht als nächste Anweisung im Quellcode folgt.

Bedingungen als logische Tests

Die beschriebenen Bedingungen zur Steuerung des Programmflusses sind als logische Tests zu verstehen. Logische Tests können entweder wahr (true) oder nicht wahr bzw. falsch (false) sein und wie erwähnt auch kombiniert werden. Hier einige Beispiele:

- ✓ Eine Person ist 15 Jahre alt. Dann würde der Test auf das Alter bei der Bedingung „Ist das Alter kleiner als 18 Jahre?“ wahr (true) ergeben.
- ✓ Eine Person ist 19 Jahre alt. Dann würde der Test auf das Alter bei der Bedingung „Ist das Alter kleiner als 18 Jahre?“ nicht wahr (false) ergeben.
- ✓ Betrachten wir konkret einen Jungen von 15 Jahren. Dann würde der Test auf das Alter bei der Bedingung „Ist das Alter kleiner als 18 Jahre?“ natürlich wieder wahr (true) ergeben. Wenn man allerdings noch das Geschlecht miteinbezieht und „Ist das Geschlecht weiblich?“ testet, kommt es auf die Art der Kombination an, ob der Gesamttest wahr oder falsch liefert. Der Einzeltest auf „Ist das Geschlecht weiblich?“ liefert falsch (false).

Der Test auf beide Bedingungen mit einer Und-Verknüpfung liefert falsch (false), weil eine der Bedingungen nicht erfüllt ist.

Der Test mit einer Oder-Verknüpfung liefert jedoch wahr (true), da hier nur eine der beiden Bedingungen erfüllt sein muss.

Für ein Mädchen mit 13 Jahren liefern sowohl die Und-Verknüpfung als auch die Oder-Verknüpfung den Wert true.

3.3 Programmabläufe visualisieren

Die strukturierte Programmierung verlangt eine gut durchdachte Vorbereitung, bevor Sie mit dem eigentlichen Programmieren (Codieren) beginnen. Zudem **zerlegen** Sie in der Regel ein Problem in kleinere Bausteine, die man isoliert betrachten und dann später zu einer Gesamtlösung zusammenfügen kann. Das wird im Konzept des „Computational Thinking“ in der zweiten Phase gefordert.

Beim Entwurf eines Programms werden oftmals grafische Darstellungsmittel für die Logik des Programmablaufs oder eines Teils davon verwendet, mit deren Hilfe sich Kontrollstrukturen und Verzweigungen sowie Algorithmen anschaulich darstellen lassen.

Diese Visualisierung erleichtert es auch erheblich, Fehler in Algorithmen und der Programmlogik zu finden und zu beseitigen oder erst gar nicht entstehen zu lassen, beispielsweise

- ✓ falsche Sequenzen,
- ✓ fehlende Programmelemente oder
- ✓ falsche Entscheidungsergebnisse.

Im Folgenden werden einige Entwurfstechniken vorgestellt, die Sie für Ihren Programmentwurf wählen können.

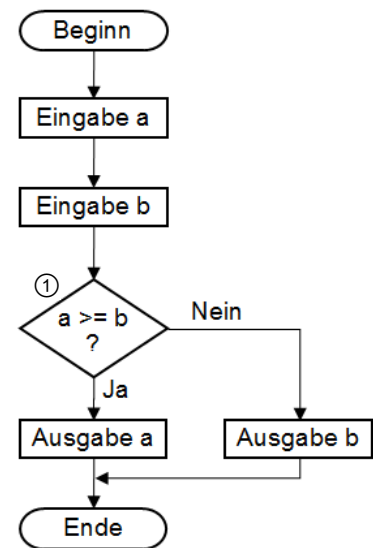
3.4 Programmablaufplan

Ein Programmablaufplan wird auch Ablaufdiagramm, Flussdiagramm oder Blockdiagramm genannt. Der Begriff **Programmablaufplan** (kurz: PAP) ist in der DIN 66001 genormt. Die aktuellste Veröffentlichung ist die Ausgabe 1983-12, siehe auch <http://www.din.de>.

Programmablaufpläne sind grafische Darstellungen mithilfe genormter Symbole. Sie sind weit verbreitete Hilfsmittel bei der Programmentwicklung. Die Programmstruktur lässt sich bildhaft als Ablauf darstellen. Die Pfeile zeigen dabei die Richtung der Verarbeitung an.

Programmablaufpläne werden meist in mehreren Stufen erstellt. Je nach Komplexität des Problems wird zunächst ein grober PAP entworfen, der immer weiter verfeinert wird, bis alle Befehle einzeln aufgeführt sind.

Die einzelnen Symbole haben unterschiedliche Bedeutungen. Die wichtigsten davon werden nachfolgend kurz erklärt.



Programmablaufplan (PAP)

Erläuterung des PAP der obigen Abbildung

Nach Beginn des Programms werden die Werte für die Variablen *a* und *b* eingegeben. Dann folgt der Vergleich der Werte der Variablen *a* und *b* ①. Ist der Wert der Variablen *a* größer als der Wert der Variablen *b* oder gleich groß, ist die Bedingung erfüllt (der Weg *Ja* wird durchlaufen), und es wird der Wert der Variablen *a* ausgegeben. Anderenfalls, wenn der Wert der Variablen *b* größer ist, ist die Bedingung nicht erfüllt (Weg *Nein*), und der Wert der Variablen *b* wird ausgegeben. Danach laufen beide Wege wieder zusammen, und das Programm ist zu Ende.

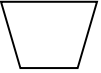





Sollte hier etwa in der späteren Programmierung beim Vergleich von *a* und *b* nur ein Größenvergleich notiert werden, erlaubt ein Abgleich mit dem PAP (oder auch anderen visuellen Darstellungsformaten) die Identifikation des fehlerhaften Laufzeitverhaltens. Auch eine fehlende Eingabe oder Ausgabe oder ein anderer logischer Fehler lässt sich so erkennen.

PAPs wie auch die nachfolgend im Kapitel vorgestellten Diagramme können Sie mit Papier und Stift erstellen. Bequemer ist die Verwendung eines Grafikprogramms. Sie können auch spezialisierte Tools zur Erstellung von Diagrammen verwenden. Neben kommerziellen Produkten gibt es empfehlenswerte Freeware, z. B. den **PapDesigner** (<http://friedrich-folkmann.de/papdesigner/>), den **Dia Diagram Editor** (<http://dia-installer.de/>) oder den **Diagram Designer** (<http://meesoft.logicnet.dk/>). Auch die Zeichnenfunktionen von Word, PowerPoint oder OpenOffice lassen sich sehr gut zum Erstellen verwenden.



Auswahl von Symbolen des PAP

	Verarbeitung Das Rechteck ist für Zuweisungen oder Ein- und Ausgabeoperationen vorgesehen.
	Verbindung Zur Verdeutlichung der Ablaufrichtung werden Linien mit einer Pfeilspitze benutzt.
	Verzweigung Bedingungen werden als Raute dargestellt. Eine Verbindungslinie führt hinein, zwei Verbindungslinien führen heraus. Je nach Wahrheitswert der Bedingung wird der Ablauf in die eine oder andere Richtung fortgeführt.

	Manuelle Verarbeitung Eingaben des Programmnutzers werden durch ein Trapez dargestellt.
	Dokumentation an anderer Stelle Durch dieses Symbol wird auf einen anderen PAP hingewiesen, der z. B. ein Unterprogramm darstellt.
	Verbinder Teilt sich ein Programmablauf und wird dann wieder zusammengeführt, wird der Verbinder (Konnektor) eingesetzt. Somit wird vermieden, dass sich Ablauflinien kreuzen. Damit wird die Übersichtlichkeit erhöht. Für größere PAPs, für die mehrere Seiten benötigt werden, wird der Verbinder am Seitenende der vorherigen und Seitenanfang der folgenden Seite verwendet. Zur Identifizierung wird er mit einer eindeutigen Zahl für diese Verbindungsstelle versehen.
	Grenzstelle Eine Grenzstelle kennzeichnet den Anfang und das Ende eines Programmablaufplans.
 	Schleifenbegrenzer Zur Darstellung von Programmwiederholungen werden diese zwei Symbole benutzt, die den Anfang und das Ende einer Schleife kennzeichnen. In der Praxis verwendet man statt dieser Schleifensymbole oft auch Verbindungen, die wieder zu einem vorherigen Verarbeitungsschritt verweisen.

Geschachtelte Strukturen sind im PAP nicht gut zu erkennen. Ein weiterer Nachteil besteht darin, dass für objektorientierte Programmkonzepte keine Symbole definiert sind. Als vorteilhaft ist jedoch zu werten, dass ...

- ✓ Anweisungsteile der Algorithmen übersichtlich lesbar sind,
- ✓ die Terminierung überprüfbar ist,
- ✓ die Korrektheit überprüfbar ist,
- ✓ die Methodik der schrittweisen Verfeinerung unterstützt wird, da jede Kontrollstruktur als Blackbox betrachtet werden kann.

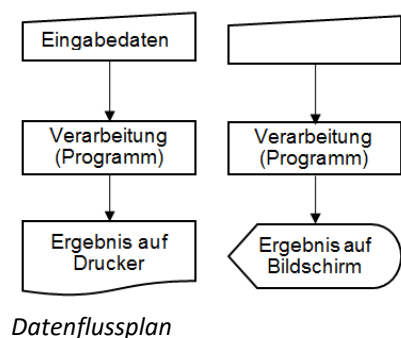


Entwerfen Sie bei der Erstellung des PAP zuerst die Kontrollstrukturen, bevor Sie die elementaren Anweisungen einbauen. Das ist die angedeutete Zerlegung eines umfangreicheren Problems in kleinere Prozesse. Oder aber Sie entwerfen zuerst die übergeordnete Logik und kümmern sich dann um Details. In jedem Fall haben Sie Teilprozesse, die zusammen die vollständige Aufgabenstellung abbilden.

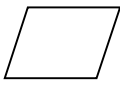

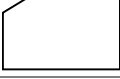


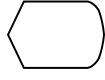
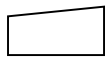
3.5 Datenflussdiagramm

Ein Datenflussplan ist eine grafische Übersicht, welche die Programme und Daten, die zu einer Gesamtaufgabe gehören, miteinander verbindet. Er zeigt, welche Teile eines Programms von den Daten durchlaufen werden und welche Art der Bearbeitung innerhalb der Programme vorgenommen wird.

Ein Datenflussplan besitzt ähnliche Symbole wie ein Programmablaufplan. Zusätzliche Sinnbilder werden vor allem für die Daten eingeführt.



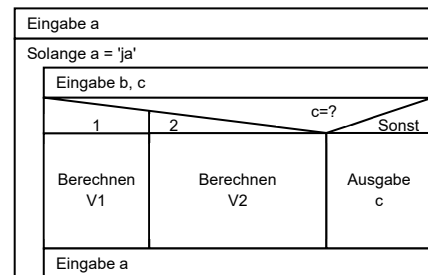
Symbole von Datenflussplänen

	Daten allgemein
	Daten auf einem Schriftstück (z. B. Druckliste)
	Daten auf einer Karte (z. B. Loch- oder Magnetkarte)
	Daten auf einem Speicher mit ausschließlich nacheinander zu verarbeitendem Zugriff (sequenziell), z. B. ein Magnetband
	Daten auf einem Speicher mit direktem Zugriff (wahlweise), z. B. eine Festplatte
	Maschinell erzeugte optische oder akustische Daten, z. B. die Bildschirmausgabe
	Manuelle Eingabe

3.6 Struktogramme

Struktogramme wurden 1973 von I. Nassi und B. Shneiderman als Darstellungsmittel für einen Algorithmus für den strukturierten Programmwurf entwickelt. Sie sind in der DIN 66261 genormt.

Ein Struktogramm ist die grafische Darstellung eines Programmablaufs in Form eines geschlossenen Blocks, der entsprechend den einzelnen logischen Grundstrukturen in verschiedene untergeordnete Blöcke aufgeteilt werden kann. Struktogramme setzen sich aus verschiedenen Symbolen für die verschiedenen Operationsarten zusammen, die von oben nach unten betrachtet werden (top-down).



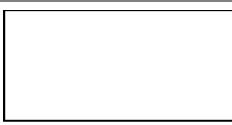
Nassi-Shneiderman-Struktogramm

Erläuterung des obigen Struktogramms

Zuerst wird der Wert *a* eingegeben. Gibt der Benutzer "ja" ein, wird er nach den Werten für die Variablen *b* und *c* gefragt. Die nachfolgende Verzweigung des Programms richtet sich nach dem Wert der Variablen *c*, also nach der Eingabe des Anwenders. Ist *c*=1, wird *V1* berechnet, ist *c*=2, wird *V2* berechnet, ansonsten wird der Wert *c* ausgegeben. Dies wird so lange durchgeführt, bis der Anwender bei der 1. Abfrage nicht "ja" eingibt.

Symbole von Struktogrammen

Ein Programmbaustein aus mehreren logisch zusammengehörenden Befehlen wird als Strukturblock bezeichnet. Ein einzelner Befehl heißt Elementarblock. Zur Darstellung von Programmabläufen verwendet man folgende Sinnbilder:

	Verarbeitung (Prozess) Mit dem Verarbeitungssymbol werden die Struktur- und Elementarblöcke dargestellt, die Ein- und Ausgabebefehle, Berechnungen und Unterprogrammaufrufe enthalten.		
<table><tr><td>Strukturblock 1</td></tr><tr><td>Strukturblock 2</td></tr></table>	Strukturblock 1	Strukturblock 2	Folge (Sequenz) Folgen mit zwei oder mehreren Arbeitsschritten werden durch aneinandergereihte Strukturblöcke dargestellt.
Strukturblock 1			
Strukturblock 2			

	Alternative (Verzweigung) Wird im Programmablauf eine Bedingung gestellt, wird dies mit dem Symbol „Alternative“ dargestellt. Ist die Bedingung erfüllt, wird der Strukturblock 1 ausgeführt, sonst Strukturblock 2.
	Fallauswahl (Mehrfachverzweigung) Eine mehrfache Bedingung im Programmablauf wird mit diesem Symbol dargestellt. Es wird kontrolliert, welche Auswahl vorgenommen wurde, und in den entsprechenden Strukturblock verzweigt. Trifft keine der Bedingungen zu, wird der Alternativblock ausgeführt.
	Wiederholung (Schleife) Diese Wiederholungssymbole dienen der Darstellung von Anweisungen in Schleifen. Der Anweisungsblock (Strukturblock) wird so lange von Neuem ausgeführt, bis die angegebene Bedingung erfüllt ist. Oben: kopfgesteuerte (abweisende) Schleife (while-Schleife oder Iteration) Unten: fußgesteuerte (annehmende) Schleife (do-while-Schleife)

Nach DIN 66261 sind insgesamt elf Symbole in Struktogrammen einsetzbar.

Die grafische Darstellung von linearen Kontrollstrukturen ist im Struktogramm optimal, da keine Sprünge darstellbar sind. Der zeichnerische Aufwand wird durch den Einsatz von Struktogramm-Generatoren begrenzt. Als besonders vorteilhaft ist zu werten, dass Alternativen nebeneinander angeordnet sind und so klare Linien des Programmablaufs erkennbar sind.

Bei Struktogrammen lässt sich die Methode der schrittweisen Verfeinerung anwenden, statt der konkreten Anweisungen wird nur eine grobe Beschreibung eingetragen (z. B. Datensatz verarbeiten). Für solch ein Rechteck wird dann wiederum ein Struktogramm entworfen, welches die Einzelheiten darstellt.

3.7 Pseudocode

Der Pseudocode ist eine halbformale, textuelle Beschreibung eines Programmablaufs, die sich an höhere Programmiersprachen anlehnt, aber keiner konkreten Sprache entsprechen muss. Der Pseudocode ist nicht genormt wie Struktogramme oder Programmablaufpläne. Für Kontrollstrukturen werden meist ähnliche Worte verwendet, wie sie in der Syntax einer Programmiersprache vorkommen können:

`if...then...else...end if, while...end while`

Für die Anweisungen werden entweder verbale Beschreibungen verwendet (z. B.: erhöhe `i` um 1) oder Anweisungen, die einer Programmiersprache ähneln (z. B.: `i := i + 1`). Im Pseudocode können auch Variablen- und Konstantendeklarationen enthalten sein.

```

begin
  BetragPruefen
    Eingabe(a);
    Eingabe(b);
    if a >= b then
      Ausgabe(a);
    else
      Ausgabe(b);
    end if
  end BetragPruefen

```

Pseudocode

Betrachten wir zur weiteren Verdeutlichung den Fall, dass die Zahlen 1 und 2 zu addieren und das Ergebnis mit 3 zu multiplizieren ist. Das könnte man in Pseudocode so formulieren:

```

begin Rechnen
  Definiere zahl1
  Definiere zahl2
  Definiere ergebnis
  Eingabe(zahl1);
  Eingabe(zahl2);
  ergebnis = (zahl1 + zahl2) * 3
  Ausgabe(ergebnis);
end Rechnen

```

Die Passage `(zahl1 + zahl2) * 3` ist bereits eine Formulierung des tatsächlichen Algorithmus, der in vielen „echten“ Programmiersprachen dann genauso notiert wird.

3.8 Entscheidungstabellen

Ein Hilfsmittel zur Darstellung von logischen Verknüpfungen sind Entscheidungstabellen (ET). Darin werden mehrere Bedingungen aufgeführt und die daraus resultierenden Aktionen festgehalten. Die Entscheidungstabellen sind leicht verständlich und können für Absprachen zwischen Programmierern und Anwendern bezüglich der Programmlogik verwendet werden.

Programmablaufpläne oder Struktogramme eignen sich für eine derartige Darstellung weniger, da mit der Anzahl der Bedingungen auch die Anzahl der Verzweigungen steigt, sodass die Pläne sehr unübersichtlich werden.

Aufbau von Entscheidungstabellen

Eine Entscheidungstabelle gibt eine Wenn-Dann-Beziehung wieder.

WENN eine bestimmte Bedingung erfüllt ist,
DANN ist eine bestimmte Aktion auszuführen.

Entscheidungstabellen werden in vier Bereiche eingeteilt. Die Bedingungen werden links oben formuliert. Die Aktionen werden darunter (links unten) aufgeführt. Im Bedingungsanzeigerteil wird die Kombination möglicher Wahrheitswerte eingetragen, und die entsprechend auszuführenden Aktionen werden im Aktionsanzeiger markiert.

Name der ET	Regelnummern
Bedingungen (WENN...)	Bedingungsanzeiger
Aktionen (DANN...)	Aktionsanzeiger

Die nachfolgende Abbildung zeigt Ihnen beispielhaft den schematischen Aufbau einer Entscheidungstabelle. Die Entscheidungsregeln bestehen aus Kombinationen von vier Bedingungen und drei Aktionen. Die im Anzeigerteil erfolgten Eintragungen haben folgende Bedeutung:

j = ja	Bedingung ist erfüllt
n = nein	Bedingung ist nicht erfüllt
x	Aktion wird ausgeführt
-	Aktion wird nicht ausgeführt, Bedingung ist für die Entscheidung nicht relevant

	ET: ET-Name	R1	R2	R3	R4	...
Bedingungsteil WENN	Bedingung 1	-	j	-	j	
	Bedingung 2	j	-	-	-	
	Bedingung 3	j	n	j	-	
	Bedingung 4	-	n	n	-	
Aktionsteil DANN	Aktion 1	-	x	-	-	
	Aktion 2	x	-	-	x	
	Aktion 3	x	x	x	-	

Schematische Darstellung einer Entscheidungstabelle

Die Entscheidungsregeln in der Tabelle sind spaltenweise zu lesen, z. B. für die Spalte R1:

WENN Bedingung 2 erfüllt ist und
Bedingung 3 erfüllt ist
(unabhängig davon, ob Bedingung 1 oder 4 erfüllt ist),

DANN führe Aktion 2 aus und
führe Aktion 3 aus.

Beispiel: Aufpreise bei Produktvarianten

	ET: Aufpreise	R1	R2	R3
<i>Bedingungsteil</i> WENN	Standardverpackung	j	n	n
	Geschenkverpackung	-	j	j
	Menge < 30	-	j	n
	Menge >= 30	-	n	j
<i>Aktionsteil</i> DANN	Aufpreis	-	x	-
	kein Aufpreis	x	-	x

Ein Unternehmen liefert ein Produkt in verschiedenen Verpackungen.

R1: Wird ein Produkt mit der standardmäßigen Verpackung gewünscht, zahlt der Kunde keinen Aufpreis.

R2: Bestellt er weniger als 30 Stück mit einer speziellen Geschenkverpackung, zahlt er einen Aufpreis.

R3: Bei mehr als 30 Stück entfällt dieser Aufpreis wieder.

3.9 Übung

Struktogramm und Entscheidungstabelle erstellen

Übungsdatei: --

Ergebnisdatei: *uebung03.pdf*

- Entwerfen Sie ein Struktogramm, um nach der Eingabe einer Schulnote die Bewertung in Textform auszugeben. Wird eine Note größer als 6 angegeben, soll eine Meldung ausgegeben werden. Die Bewertungen lauten:

1: Sehr gut	3: Befriedigend	5: Mangelhaft
2: Gut	4: Ausreichend	6: Ungenügend
- Entwickeln Sie eine Entscheidungstabelle für die Versandbedingungen eines Versandhauses. Dieses berechnet bis zu einem Einkaufspreis von 150,00 EUR Versandkosten in Höhe von 8,00 EUR; ab einem Einkaufspreis von 300,00 EUR wird zusätzlich ein Gratisgeschenk verschickt.

4 Werkzeuge der Softwareentwicklung

In diesem Kapitel erfahren Sie

- ✓ welche Werkzeuge Sie beim Erstellen eines Programms benötigen
- ✓ was ein Compiler, Interpreter, Binder, Lader ist
- ✓ welchen Grundaufbau ein Programm hat
- ✓ wie Sie ein Programm kompilieren und ausführen
- ✓ wie Sie ein Skript interpretieren lassen

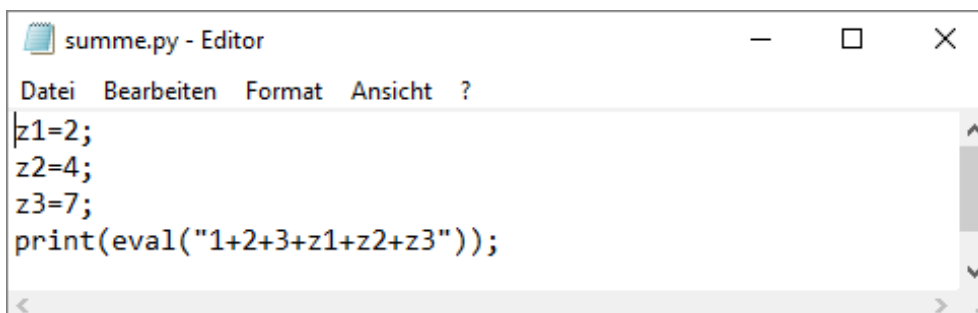
Voraussetzungen

- ✓ Grundlegende Computerkenntnisse

4.1 Programme erstellen

Quelltext eingeben

Bei der Programmierung wird ein Quelltext (auch Code oder Quellcode genannt) in einer bestimmten Programmiersprache erstellt. Zur Eingabe dient ein **Texteditor** oder kurz **Editor**. Ein Editor ist eine Software zur Bearbeitung von Texten. Im Unterschied zu einer Textverarbeitung oder einem Layout-Programm fügt ein Editor keine Formatierungsanweisungen in den Text ein, um ihn z. B. in Absätze zu gliedern oder Textpassagen hervorzuheben.



Editor (Windows) mit Quelltext (Python)

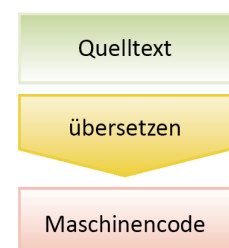
Editoren sind auf allen Computerplattformen verfügbar und besitzen unterschiedliche Funktionsausstattungen. Viele Editoren können z. B. programmiersprachenspezifische Sprachelemente hervorheben.

Entwicklungsumgebungen für eine bestimmte Programmiersprache besitzen einen eigenen, integrierten Editor, der eine Hervorhebung der Syntax und Hilfestellungen für verschiedene Sprachelemente bietet.

Programme übersetzen

Der Quelltext kann von einem Computer nicht direkt ausgeführt werden. Der Computer benötigt – wie bereits erläutert – eine sogenannte Maschinensprache (Maschinencode). Deshalb muss jeder Quelltext in den rechnerartspezifischen Maschinencode übersetzt werden.

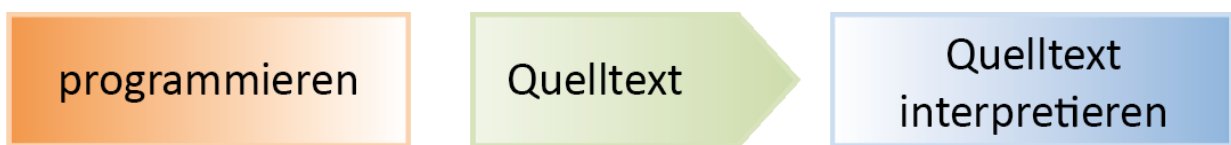
Abhängig von der verwendeten Programmiersprache lassen sich drei grundlegende Verfahren für den Übersetzungsvorgang unterscheiden.



- ✓ Vollständige Übersetzung **bei der Programmausführung** mit einem Interpreter, z. B. bei der Programmiersprache BASIC oder bei der Skriptsprache JavaScript
- ✓ Vollständige Übersetzung **bei der Programmerstellung** mit einem Compiler, z. B. bei der Programmiersprache C
- ✓ **Zweistufige Übersetzung**, z. B. bei der Programmiersprache Java:
 - ✓ Übersetzung in plattformunabhängigen Bytecode bei der Programmerstellung (Compiler)
 - ✓ Übersetzung des plattformunabhängigen Bytecodes bei der Programmausführung (Interpreter)

4.2 Konzepte zur Übersetzung

Interpreter – Übersetzung während der Programmausführung



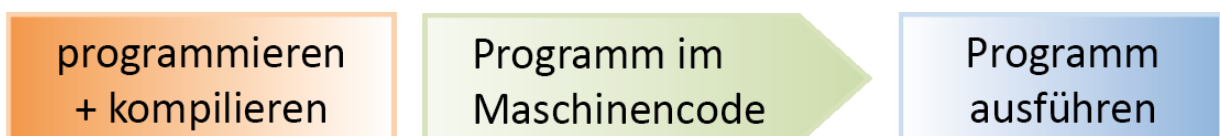
Während der Laufzeit eines Programms oder Skripts (Programmausführung) übersetzt ein **Interpreter** den Quelltext in Maschinencode. Interpreter übersetzen ein Programm zeilenweise, wobei erst dann jede Zeile einer Syntaxprüfung unterzogen wird.

- ✓ Sind die Befehle in der Zeile korrekt, werden sie vom Interpreter in Anweisungen für den Prozessor übersetzt und ausgeführt. Anschließend wird die nächste Zeile geprüft usw.
- ✓ Tritt während des Programmablaufs ein Syntaxfehler auf, wird der Übersetzungsvorgang abgebrochen und eine entsprechende Fehlermeldung ausgegeben. Allerdings ist das Programm schon bis zu der Fehlerstelle ausgeführt worden.

Bei jedem Übersetzungsvorgang wird die Fehlerüberprüfung automatisch durchgeführt. Interpretierte Programme werden aufgrund der Prüfung zur Laufzeit langsamer ausgeführt als vor der Laufzeit übersetzte Programme.

Mittlerweile gibt es Programmiersprachen, deren Interpreter bereits vor dem Start eines Programms die Syntax kontrollieren und verschiedene Optimierungen vornehmen. Diese Programme werden schneller abgearbeitet, da in diesem Fall keine Überprüfung während der Laufzeit erfolgen muss. Ein Vertreter dieser Art ist die Programmiersprache Perl, die zum Beispiel zum Erstellen von dynamischen Internetseiten benutzt wird.

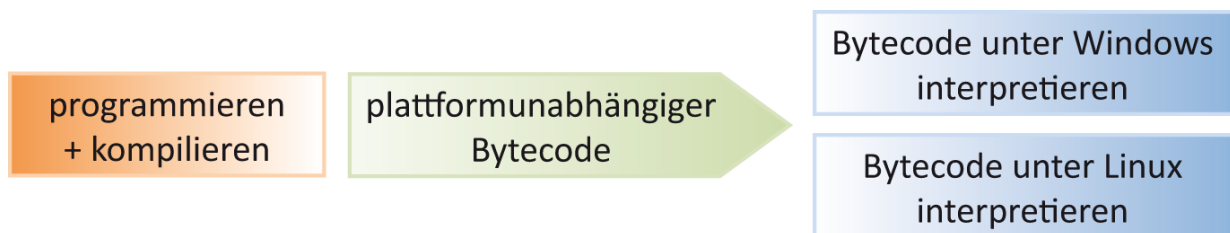
Compiler – Quelltext vor der Laufzeit übersetzen



Um ein Programm schnell ausführen zu können, muss es **vor der Laufzeit** in Maschinencode umgewandelt werden, den der Prozessor ausführen kann. Dies übernimmt ein **Compiler**. Dieser Vorgang wird Kompilieren genannt. Der Compiler übersetzt Programme einer Programmiersprache in einen Maschinencode, ohne jedoch die Befehle dabei schon auszuführen. Der zu übersetzende Code wird auch **Quellprogramm** (Quelltext) und das übersetzte Programm **Zielfprogramm** oder einfach nur **Programm** genannt. Bei der Übersetzung eines Programms durch einen Compiler wird jede Anweisung auf die korrekte Syntax der Befehle geprüft. Im Fehlerfall wird der gesamte Vorgang des Übersetzens abgebrochen, und der Fehler muss beseitigt werden, bevor das Zielfprogramm erstellt werden kann.

Assembler	Ein Übersetzer von einer maschinenorientierten Programmiersprache in eine Maschinsprache wird als Assembler bezeichnet. Einige Compiler höherer Programmiersprachen haben die Möglichkeit, Assemblercode anstatt Maschinencode zu erzeugen.
Präprozessor	Ein Präprozessor modifiziert vor dem Kompilieren den Quellcode. Mit seiner Hilfe werden weitere Quelltextdateien eingebunden, Teile des Codes ignoriert oder ersetzt.
Compiler	Ein Compiler übersetzt ein Programm in der Regel direkt in die Maschinsprache des Zielsystems. Dabei werden verschiedene Optimierungen durchgeführt. Je nach Programmiersprache und Programmierungsumgebung können mehrere Compilerdurchläufe zur Erzeugung des eigentlichen Programms notwendig sein.
Cross-Compiler	Sie nehmen eine Sonderstellung ein, denn sie erzeugen einen Maschinencode für eine andere Computerplattform als die, auf der sie ausgeführt werden.

Zweistufige Übersetzung – Übersetzung mit Zwischencode

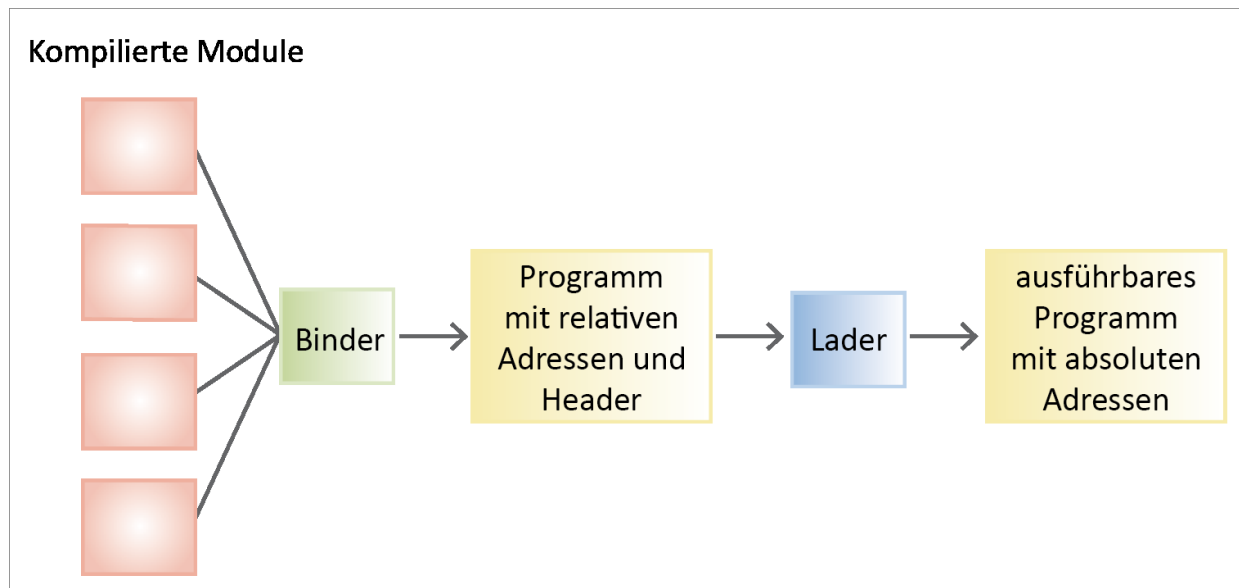


Ein Nachteil bei kompilierten Programmen ist, dass diese Programme maschinenabhängig sind und somit nicht auf jeder Computerplattform, z. B. Linux und Windows, ausgeführt werden können. Java oder die .NET-Plattform gehen in dieser Hinsicht einen anderen Weg. Der Programmcode in Java wird nicht zu einem ausführbaren Programm, sondern in einen **Zwischencode**, den sogenannten **Bytecode**, kompiliert (1. Schritt). Dieser Code ist für alle Plattformen gleich und kann mithilfe des entsprechenden **plattformspezifischen Interpreters** (2. Schritt) auf der jeweiligen Plattform ausgeführt werden. Java-Interpreter werden auch virtuelle Maschinen (JVM) genannt. Bei der .NET-Plattform geht man einen etwas anderen Weg, aber auch hier kommt ein Zwischenformat (Common Intermediate Language) zum Einsatz. Es werden verschiedene Programmiersprachen zunächst in die Common Intermediate Language übersetzt, bevor sie von der Laufzeitumgebung ausgeführt werden.

Binden und Laden

In der letzten Phase der Übersetzung werden die Teile des Codes aus der getrennten Übersetzung der einzelnen Module zum endgültigen ausführbaren Programm zusammengesetzt.

Binder (Linker)	Die einzeln kompilierten Module eines Programms sind noch nicht lauffähig, da die verwendeten Adressen von Funktionen, Prozeduren usw. angepasst werden müssen. Ein Binder verbindet diese einzelnen Programm-Module zu einem vollständigen Programm (relative Adressauflösung innerhalb der Module und Anfügen eines Programmheaders). Er zählt zu den Bibliotheksverwaltungsprogrammen, gehört aber inzwischen zu jedem Betriebssystem, das über Compiler verfügt, bzw. zur verwendeten Programmiersprache.
Lader	Ein Lader (oder auch Ladeprogramm) hat die Aufgabe, ein übersetztes und gebundenes Programm an eine Ladeadresse im Arbeitsspeicher zu bringen und alle Adressangaben dieses Programms auf diese auszurichten, d. h., relative Adressen in absolute Adressen umzusetzen. Das Programm kann von dort ausgeführt werden.
Bindelader	Über einen Bindelader werden die zwei Vorgänge zusammengefasst. Die einzelnen Module werden zu einem vollständigen Programm zusammengesetzt, anschließend wird das Programm in den Arbeitsspeicher geladen.



Ablauf des Bindens und Ladens der kompilierten Module

Ein Programm besitzt vor seiner Ausführung nur relative Adressangaben, d. h., alle Positionen innerhalb des Programms werden relativ zu einer Basisadresse angegeben. Wird ein Programm ausgeführt, werden diese relativen Adressangaben durch absolute ersetzt (wo sich das Programm im Hauptspeicher befindet).



Der Vorgang des Bindens und Ladens wird in modernen Entwicklungsumgebungen meist automatisch vorgenommen und bleibt dem Programmierer weitgehend verborgen.

4.3 Entwicklungsumgebungen

Für viele Programmiersprachen werden mächtige Werkzeuge für die Programmierung zur Verfügung gestellt, die **(integrierte) Entwicklungsumgebungen** (IDE, engl. Integrated Development Environment) oder auch Entwicklungstools genannt werden. Meist beinhalten sie unter einer grafischen Benutzeroberfläche den Quelltexteditor, eine Projektverwaltung, einen Compiler und einen Debugger zum einfachen Erstellen und Testen von Programmen. Professionelle aktuelle Entwicklungsumgebungen sind z. B. Microsoft Visual Studio für verschiedene Programmiersprachen wie Visual Basic, C#, C++ oder JScript und Eclipse (<http://www.eclipse.org>) für Java oder diverse andere Sprachen (über Plugins ist Eclipse erweiterbar).



Im Buch wird bei Programm-Beispielen mit Python gearbeitet. Diese Arbeit können Sie in IDLE leichter erledigen, als wenn Sie mit einem reinen Editor und der Python-Kommandozeile arbeiten. Diese IDE ist im „normalen“ Download-Paket von Python bereits integriert. Bei den Hinweisen zur Installation von Python im Anhang finden Sie genauere Details.

Integrierte Entwicklungsumgebungen bieten eine ganze Reihe an Unterstützung für die Programmierung.

Editoren mit Programmiersprachenunterstützung

Integrierte Quelltexteditoren bieten umfangreiche Leistungsmerkmale, wie zum Beispiel

- ✓ die Hervorhebung der Syntaxstruktur durch Farben oder durch textliche Auszeichnungen,
- ✓ eine integrierte Programmierhilfe sowie
- ✓ eine automatische Code-Vervollständigung.

Mithilfe einer Projektverwaltung können größere Programme komfortabel verwaltet werden. Teilweise sind Versionsverwaltungssysteme und die Unterstützung für die Entwicklung in einem Team integriert.

Debugger

Logische Fehler sind Fehler, die bei der Programmausführung falsche Ergebnisse liefern. Da die Syntax des Programms fehlerfrei ist und auch bei der Programmausführung selbst kein Fehler auftritt, erkennt weder der Compiler den Fehler noch können Sie den Fehler durch einfaches Starten des Programms ermitteln. Der Fehler wird durch eine falsche Programmlogik oder Eingabefehler hervorgerufen. Um diese Fehler aufzufinden, können Sie einen **Debugger** verwenden.

4.4 Standardbibliotheken und Wiederverwendung

Für die meisten Programmiersprachen existiert eine Reihe von Standardbibliotheken (API). Diese Bibliotheken sind Sammlungen von bereits implementierten Funktionalitäten, die Sie in Ihren eigenen Programmen verwenden können, z. B. Grafikroutinen, mathematische Funktionen, Funktionen für den Umgang mit Datum und Uhrzeit oder Funktionen für den Dateizugriff. Dazu müssen diese in das eigene Programm eingebunden werden.

Bei der Verwendung von Bibliotheken kann der Vorteil der **Wiederverwendbarkeit** ausgenutzt werden (DRY-Prinzip).

- ✓ Der Programmcode ist einfacher zu warten.
- ✓ Funktionen können mehrfach verwendet werden.
- ✓ Es muss weniger Quellcode geschrieben werden.
- ✓ Die Fehleranfälligkeit des Programms wird verringert, da der wiederverwendete Quellcode bereits ausführlich getestet wurde.

Es liegen für die meisten Programmiersprachen sehr umfangreiche Bibliotheken für die verschiedenen Anwendungsgebiete vor.

4.5 Grundaufbau eines Programms am Beispiel Python

Jedes Programm, das Sie in einer Programmiersprache schreiben, hat eine gewisse Anordnung von Anweisungen oder gar ein spezielles Grundgerüst, das immer verwendet werden muss. Dabei kann eine ganz bestimmte Methode oder Funktion notwendig sein, die immer beim Start eines Programms als erste Anweisung aufgerufen wird (etwa in Java), oder aber es werden einfach alle Anweisungen im Quellcode von oben nach unten der Reihe nach abgearbeitet (etwa in JavaScript). Zur Erklärung des Grundaufbaus von Programmen am Beispiel der Programmiersprache Python wird das zu Demonstrationszwecken sehr oft verwendete "Hallo Welt"-Programm in drei Versionen angewendet. Dieses gibt zwar nur auf Ihrem Bildschirm "Hallo Welt!" aus, zeigt aber alles, was zum grundsätzlichen Verständnis notwendig ist.

Da Python verschiedene Programmierparadigmen unterstützt, kann man mit dieser Sprache sowohl

- ✓ die rein sequenzielle Abarbeitung von Anweisungen in einem Quellcode,
- ✓ den Aufruf einer Hauptfunktion als auch
- ✓ einen objektorientierten Ansatz

demonstrieren.

Beachten Sie, dass in allen Python-Quellcodes **Spalten** beziehungsweise **Eintrückungen** bei Codes im Editor eine Bedeutung haben. Das ist bei vielen anderen Sprachen nicht der Fall, obgleich es für guten Programmstil auch da dringend zu empfehlen ist. Durch Einrückungen werden in Python Strukturen zusammengefasst. Inkorrekte Einrückungen führen zu Fehlern! In der Vorgabe rückt man Strukturen immer um 4 Leerzeichen ein.



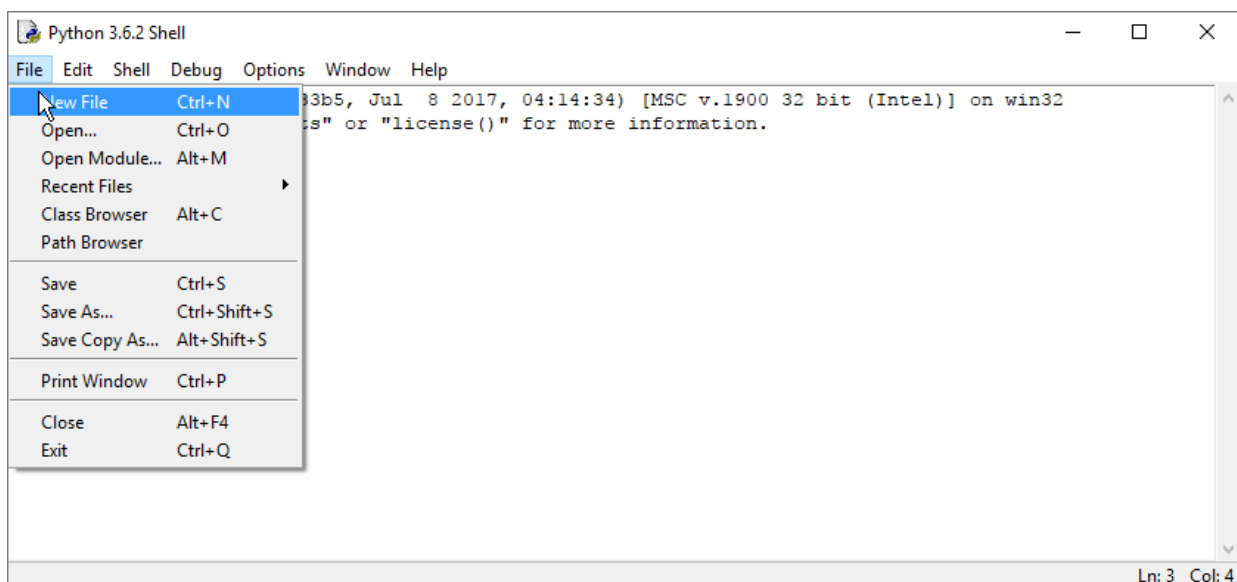
Eine Anwendung mit sequenzieller Abarbeitung erstellen

Bei der prozeduralen Programmierung werden zur Problemlösung eine Folge von Anweisungen angegeben, die dann bei der Ausführung einfach nacheinander abgearbeitet werden. Insbesondere Skripte wie JavaScripts arbeiten meist nach dem Verfahren. Dabei hat eine Quelltextdatei in der Regel keinen vorgegebenen Aufbau, sondern die Anweisungen werden einfach nacheinander notiert und beim Ausführen in genau der Reihenfolge abgearbeitet.



In vielen Programmiersprachen werden Anweisungen mit einem Semikolon beendet. Ein paar wenige Sprachen verbieten hingegen das Semikolon am Ende. In Python **kann** man Anweisungen mit einem Semikolon beenden, **muss** es aber **nicht**. Die verschiedenen Einrückungsebenen kennzeichnen das Ende von Anweisungen in Python bereits eindeutig. Aber um den Kodierungsstil verwandten Sprachen wie C, Java, PHP, JavaScript oder C# anzugleichen, notieren Programmierer auch unter Python gelegentlich das Semikolon am Ende einer Anweisung. Dafür nimmt man in Kauf, dass man im Grunde überflüssige Zeichen notiert.

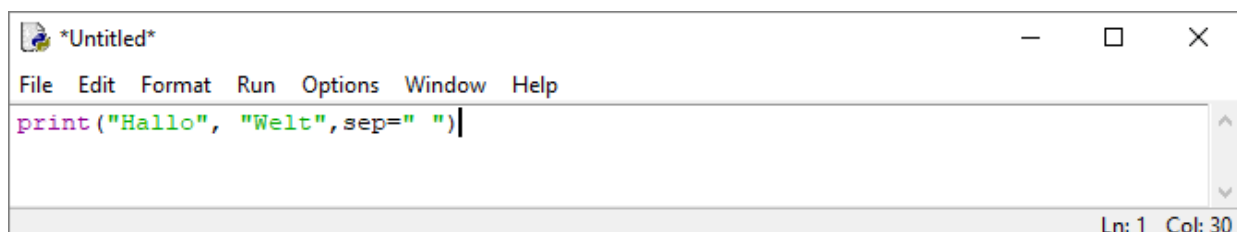
- ✓ Öffnen Sie IDLE. Es startet die sogenannte **Python Shell** oder Python-Kommandozeile, die voll in IDLE integriert ist. Hier sehen Sie insbesondere die Ausgabe eines Programms oder vom Python-Laufzeitsystem selbst.
- ✓ Im Menü *File* der Python Shell wählen Sie den Befehl *New File*. Damit können Sie eine neue Python-Quellcodedatei erzeugen.



In IDLE aus der integrierten Python Shell heraus eine neue Quelltextdatei anlegen

Im nun zusätzlich geöffneten integrierten Editor von IDLE können Sie Python-Quelltext eingeben.

- ✓ Geben Sie im integrierten Editor von IDLE das folgende Beispielprogramm ein.



Der Quelltext im integrierten Editor von IDLE

Eine der wichtigsten Python-Funktionen beziehungsweise -Anweisungen ist `print()`. Damit generiert man eine Ausgabe. Es handelt sich um eine **Python-Built-in-Function**, die immer automatisch in Python verfügbar ist.

Bei der Notation des Aufrufs sind immer eine öffnende und eine schließende Klammer notwendig. Innerhalb der Klammern schreibt man dann die Werte, die man ausgegeben haben möchte. Dabei kann man kommasepariert auch mehrere Werte notieren, die zusammen ausgegeben werden sollen. Diese Werte werden mit dem Zeichen verbunden, das über den **Separator** spezifiziert wird. Über den Optionsparameter `sep` (für Separator) kann man dieses Trennzeichen anpassen. Das macht man mit einer Zuweisung über das Gleichheitszeichen. Wenn man da einen Leerstring angibt, kann das Trennzeichen auch vollständig beseitigt werden. Sie geben einfach das oder die gewünschte(n) Zeichen als weiteren Parameter zusätzlich an.

Die in die Klammern notierten Werte sind die **Parameter** oder **Übergabewerte** der Funktion.



Diese Art der Wertzuweisung bei Parametern wird in Python sehr häufig für die Konfiguration von Funktionen verwendet. Dieses Beispiel mit dem Parameter `sep` kann also verallgemeinert betrachtet werden. Man hat in Python bei Funktionen sehr oft ein Default- beziehungsweise Vorgabeverhalten. Wenn dieses modifiziert werden soll, wird einem bestimmten Attribut ein neuer Wert zugewiesen.

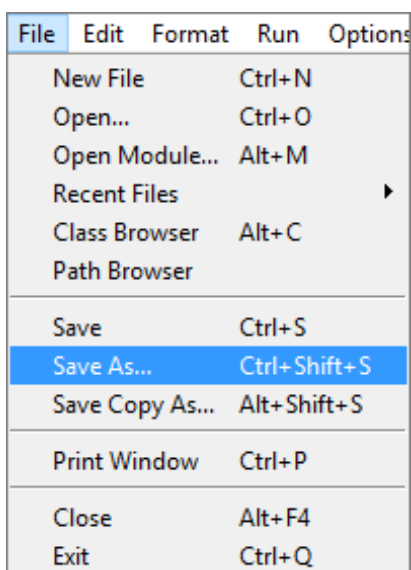


Python beachtet Groß- und Kleinschreibung. Man nennt dies **case-sensitiv**. Das gilt auch für Built-in-Functions und deshalb wäre die Schreibweise `Print()` oder `PRINT()` falsch. Alle Built-in-Functions werden kleingeschrieben! Das trifft in der Regel auch für alle anderen Anweisungen in Python zu.



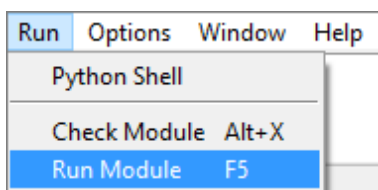
Die Quelltextdatei ist nun zu speichern. Die übliche Dateierweiterung für Python-Quellcode ist `py`. Dabei ist es sinnvoll, dass Sie einen eigenen Ordner für Ihre Python-Quelltexte verwenden. Es ist ebenso sinnvoll, dass Sie darin dann weitere Unterordner für einzelne Projekte erstellen.

- ✓ Speichern Sie die Quelltextdatei in Ihrem Python-Arbeitsverzeichnis unter dem Namen *HalloWelt.py*. Im Menü *File* des integrierten Editors finden Sie die üblichen Befehle zum Speichern.



Speichern aus dem Editor der IDLE

Im Menü *Run* finden Sie den Befehl zum Ausführen des Programms direkt aus IDLE (*Run Module*).



- ✓ Führen Sie das Programm aus. Die Ausgabe erfolgt in der integrierten Python Shell.

Die Ausgabe des Programms in der integrierten Python Shell von IDLE

Sie können erkennen, dass der Interpreter den Quelltext interpretiert und ausgeführt hat. Dabei wird der zusammengesetzte Text ausgegeben.

Eine Python-Anwendung mit Hauptfunktion erstellen

Obwohl es in Python unnötig ist, ziehen es manche Programmierer vor, ein Programm vollständig aus einer Art **Wurzel** (root-Element oder kurz root) zu entwickeln. Damit möchte man auch in Python vom Stil synchron zu vielen anderen Programmiersprachen programmieren, die so eine Funktion benötigen. Man erstellt dabei eine Wurzel- oder Hauptfunktion und ruft daraus alle weiteren Anweisungen des Programms auf. Das Programm zur Ausgabe einer einfachen Meldung kann wie folgt umgeschrieben werden.

- ✓ Öffnen Sie wieder IDLE und erstellen Sie über das Menü *File* der Python Shell und den Befehl *New File* eine neue Python-Quellcodedatei.
- ✓ Geben Sie im integrierten Editor von IDLE das folgende Beispielprogramm ein.

```

① def main():
②     print("Hallo", "Welt", sep=" ")
③ main()

```

- ① Mit dem Schlüsselwort `def` wird in Python eine **Deklaration** (die Einführung oder Definition) einer Funktion eingeleitet. Danach folgen der Name der Funktion und ein Klammernpaar. In die runden Klammern können bei Bedarf Parameter eingefügt werden. Der Doppelpunkt beendet diese Deklarationszeile. Der Bezeichner (Name) der Funktion ist in dem Beispiel `main`. Dieser Bezeichner ist in Python willkürlich, wird aber oft gewählt, weil viele verwandte Sprachen einen solchen Bezeichner für ihre Wurzelfunktion oder Wurzelmethode fordern. Damit bleibt man vom Stil synchron.
- ② Mit einer Einrückung um 4 Leerzeichen wird die Ausgabeanweisung der Funktion zugeordnet. Die Anweisung selbst bleibt unverändert.
- ③ Die Funktion `main()` wird aufgerufen. Beachten Sie, dass der Aufruf nicht eingerückt wird – er zählt zur Hauptebene des Programms.

Die Quelltextdatei ist nun wieder zu speichern.

- ✓ Speichern Sie die Quelltextdatei in Ihrem Python-Arbeitsverzeichnis unter dem Namen *HalloWelt2.py*.

Im Menü *Run* finden Sie den Befehl zum Ausführen des Programms direkt aus IDLE (Run Module).

- ✓ Führen Sie das Programm wieder über das Menü *Run* und den Befehl *Run Module* aus.

Die Ausgabe erfolgt wieder in der integrierten Python Shell und unterscheidet sich nicht von der Ausgabe des vorhergehenden Beispiels.

Eine objektorientierte Python-Anwendung erstellen

In Python kann man alle Informationen, die zur Ausführung eines Programms notwendig sind, in Klassen notieren. Diese Klassen umfassen Attribute (Daten) und Methoden (Funktionalität). Die Methoden beinhalten die Anweisungen, die ausgeführt werden sollen.

Jede einzelne Klasse kann dabei in einer eigenen Datei mit der Dateinamenerweiterung `.py` gespeichert werden. Die Datei erhält dabei oft den Namen der Klasse als eigenen Namen. Im Beispiel wird die einzige Klasse unter dem Namen `HalloWelt3.py` als Datei gespeichert. Die Klasse `HalloWelt3` enthält analog der Vorgehensweise beim Verwenden einer Wurzelfunktion eine Methode mit dem „Hauptprogramm“, die sogenannte **main-Methode**. Die main-Methode besteht erneut aus einer einzigen Anweisung, die den Text `"Hallo Welt!"` auf dem Bildschirm ausgibt. Da Python diese beliebig zu benennende Methode nicht automatisch aufruft (im Gegensatz zu beispielsweise Java, wo die Methode einen vorgegebenen Namen haben muss und diese dann aber auch automatisch aufgerufen wird), muss man allerdings noch ein Objekt der Klasse erzeugen (man nennt das **instanzieren**) und darüber dann die Methode aufrufen. Das entspricht dann dem Aufruf einer Funktion, wie es im vorherigen Beispiel durchgeführt wurde.

- ✓ Öffnen Sie wieder IDLE und erstellen Sie über das Menü *File* der Python Shell und den Befehl *New File* eine neue Python-Quellcodedatei.
- ✓ Geben Sie im integrierten Editor von IDLE das folgende Beispielprogramm ein.

```
① class HalloWelt3:
②     def main(self):
③         print("Hallo", "Welt", sep=" ")
④ HalloWelt3().main()
```

- ① Mit dem Schlüsselwort `class` wird in Python eine **Deklaration** einer Klasse eingeleitet. Danach folgen der Name der Klasse und ein Doppelpunkt.
- ② Mit einer Einrückung von 4 Leerzeichen werden alle Bestandteile der Klasse dieser zugeordnet. In dem Fall ist das eine Methode mit dem Bezeichner `main`. Dieser Bezeichner ist – wie schon im Fall der Wurzelfunktion – in Python willkürlich, wird aber oft gewählt, weil viele verwandte Sprachen einen solchen Bezeichner für ihre Wurzelfunktion oder Wurzelfunktion fordern. Damit bleibt man vom Stil synchron.

Der Parameter `self` muss in Python in jeder Deklaration einer Instanzmethode als erster Parameter notiert werden und verweist auf das aktuelle Objekt. Beim Aufruf der Methode wird dieser Parameter dann aber automatisch gefüllt und darf nicht mit angegeben werden. Der Parameter `self` dient in Python auch dazu, dass man in Methoden auf die anderen Instanzelemente (Elemente, die zusätzlich in der Klasse deklariert werden) zugreifen kann. Man notiert in der Punktnotation einfach `self` und dann die gewünschte Eigenschaft oder Methode. Der Parameter `self` steht also für ein anonymes Objekt.
- ③ Mit einer weiteren Einrückung um 4 Leerzeichen wird die Ausgabeanweisung der Methode `main()` zugeordnet. Die Anweisung selbst bleibt unverändert.
- ④ Aus der Klasse wird ein Objekt erzeugt (instanziiert) und direkt die Methode `main()` aufgerufen. Beachten Sie, dass der Aufruf auch hier nicht eingerückt wird – er zählt zur Hauptebene des Programms.

Die Quelltextdatei ist nun wieder zu speichern.

- ✓ Speichern Sie die Quelltextdatei in Ihrem Python-Arbeitsverzeichnis unter dem Namen `HalloWelt3.py`.
- ✓ Führen Sie das Programm wieder über das Menü *Run* und den Befehl *Run Module* aus.

Die Ausgabe erfolgt wieder in der integrierten Python Shell und unterscheidet sich nicht von der Ausgabe der beiden vorhergehenden Beispiele.

Im Buch wird in der Folge überwiegend mit dem sequenziellen Aufbau der Quelltextdatei gearbeitet. Das bedeutet aber nicht, dass keine Funktionen oder Objekte verwendet werden.



Die Kommandozeile von Python

Sowohl die integrierte Python Shell in IDLE als auch eine „normale“ Kommandozeile des Betriebssystems (Terminal, Shell, Konsole oder ganz früher unter Windows DOS-Box genannt) erlauben die Ausführung von Python-Anweisungen, wenn man dort den **Interaktivmodus** von Python verwendet. Dabei gibt man in der Python Shell Anweisungen ein, die von einem Python-Interpreter unmittelbar ausgeführt werden. Das wird in dem Buch aber nicht weiterverfolgt.

4.6 Übungen

Übung 1: Fragen zu Werkzeugen der Softwareentwicklung

Übungsdatei: --

Ergebnisdatei: *uebung04.pdf*

1. Warum kann ein in einer höheren Programmiersprache codiertes Programm nicht sofort nach der Codierung ausgeführt werden?
2. Worin unterscheidet sich die Arbeitsweise eines Compilers von der eines Interpreters?
3. Warum ist es sinnvoll, in eigenen Programmen fertige Programmteile aus Bibliotheken zu verwenden?

Übung 2: Programm zur Ausgabe einer Textzeile erstellen

Übungsdatei: --

Ergebnisdatei: *ErsteAusgabe.py*

1. Erstellen Sie ein Programm, das folgende Ausgabe erzeugt: Python-Programme können auch aus Klassen bestehen.
2. Speichern Sie das Programm unter dem Namen *ErsteAusgabe.py*.
3. Kompilieren Sie das Programm und führen Sie das Programm anschließend aus.

5 Zahlensysteme und Zeichencodes

In diesem Kapitel erfahren Sie

- ✓ welche verschiedenen Zahlensysteme es gibt
- ✓ welche Zeichencodes es gibt
- ✓ wie Daten innerhalb des Computers dargestellt werden

Voraussetzungen

- ✓ Mathematische Grundkenntnisse

5.1 Zahlensysteme unterscheiden

Wozu werden Zahlensysteme benötigt?

Zahlensysteme werden zur Darstellung quantitativer Merkmale von Gegenständen, Vorgängen etc. verwendet. Diese Systeme basieren auf Zeichen, aus denen die Zahlen gebildet werden. Die bekanntesten Zahlensysteme sind das arabische (0, 1, 2 ...) und das römische (I, II, III, IV ...) Zahlensystem. Das arabische Zahlensystem ist in Europa das jüngere der beiden und das leistungsfähigere. Rechenoperationen sind dort sehr viel leichter möglich als mit dem römischen Zahlensystem.

Die Ziffern bzw. Zeichen, die in dem Zahlensystem vorkommen, werden **Nennwerte** genannt. Alle Zahlen des Zahlensystems setzen sich aus diesen Nennwerten zusammen. Die Anzahl der Nennwerte entspricht der **Basis** des Zahlensystems.



Dezimalsystem

Da Menschen zehn Finger zum Rechnen zur Verfügung stehen, entwickelten sie das Dezimalsystem. Die Basiszahl des Dezimalsystems ist die Zahl 10. Somit werden zehn Ziffern (Nennwerte) zur Darstellung aller Zahlen dieses Systems benötigt: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**.

Innerhalb einer Zahl erhält jede Ziffer einen sogenannten **Stellenwert**. Der Stellenwert nimmt beim Dezimalsystem von Position zu Position um den Faktor 10 zu. Das Dezimalsystem und alle weiteren hier vorgestellten Zahlensysteme gehören im Gegensatz z. B. zum römischen Zahlensystem zu den Stellenwertsystemen.

Jede Dezimalzahl lässt sich auch als Summe von Potenzen der Basis 10 darstellen.

Beispiel: Dezimalzahl als Summe von 10er-Potenzen

234,3	=	$3 * 10^{-1}$	→	0,3	niedrigster Stellenwert
	+	$4 * 10^0$	→	4	
	+	$3 * 10^1$	→	30	
	+	$2 * 10^2$	→	200	höchster Stellenwert
			=	234,3	Summe ergibt Dezimalzahl

Dualsystem

Das bereits im 17. Jahrhundert entwickelte Dualsystem (auch als Binärsystem bezeichnet) arbeitet mit den beiden Nennwerten **0** und **1**. Durch Kombination dieser Ziffern lässt sich jede Dezimalzahl darstellen.

Für die EDV bot sich das Binärsystem an. Da es für elektronische Signale zwei Zustände („ein“ und „aus“) gibt, lassen sich diese durch die Ziffern 1 (für „ein“) und 0 (für „aus“) oder umgekehrt darstellen.

Das Dualsystem ist ein Stellenwertsystem mit der Basis 2. Der Stellenwert einer Ziffer innerhalb einer Zahl des Dualsystems nimmt von Position zu Position um den Faktor 2 zu. Daher lässt sich jede Dualzahl auch als Summe von Potenzen zur Basis 2 darstellen. Das Ergebnis der Summe entspricht der Dezimalzahl dieser Dualzahl.

Beispiel: Dualzahl als Dezimalzahl

(1101)₂	=	1 * 2 ⁰	→	1	niedrigster Stellenwert
	+	0 * 2 ¹	→	0	
	+	1 * 2 ²	→	4	
	+	1 * 2 ³	→	8	höchster Stellenwert
			=	(13)₁₀	Summe ergibt Dezimalzahl



Um Missverständnissen vorzubeugen bei der Frage, welchem Zahlensystem eine Zahl angehört, wird die Zahl in Klammern gesetzt und die Basis des Zahlensystems tiefergestellt, beispielsweise (750)₁₀ oder (101)₂.

Umwandlung einer Dezimalzahl in eine Dualzahl

Um eine Dezimalzahl in eine Dualzahl umzuwandeln, wird die Dezimalzahl durch 2 dividiert und der Rest notiert; das ganzzahlige Ergebnis wird wieder durch 2 dividiert und der Rest notiert, so lange, bis das ganzzahlige Ergebnis 0 beträgt.

Beispiel: Dezimalzahl als Dualzahl

(22)₁₀	=	22 / 2	→	11	Rest 0	0	niedrigster Stellenwert
		11 / 2	→	5	Rest 1	1	
		5 / 2	→	2	Rest 1	1	
		2 / 2	→	1	Rest 0	0	
		1 / 2	→	0	Rest 1	1	höchster Stellenwert
			=			(10110)₂	Umwandlung in Dualzahl

Hexadezimalsystem

Da große Dualzahlen schwer lesbar sind, wurden Zahlensysteme entwickelt, die mehrere Stellen einer Dualzahl zusammenfassen, das Hexadezimal- und das Oktalsystem. Im Hexadezimalsystem werden jeweils vier Stellen einer Dualzahl codiert, wodurch sich sechzehn verschiedene Kombinationen von 0 und 1 ergeben.

Die Basiszahl des Hexadezimalsystems ist die Zahl 16. Das Hexadezimalsystem enthält die folgenden sechzehn Nennwerte: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**.



Da das arabische Zahlensystem nur zehn Ziffern kennt, wurde vereinbart, die übrigen Ziffern mit Buchstaben zu bezeichnen: 10=A, 11=B, 12=C, 13=D, 14=E, 15=F.

Der Stellenwert einer Ziffer nimmt von Position zu Position um den Faktor 16 zu. Eine Hexadezimalzahl lässt sich daher als Summe von Potenzen der Basis 16 darstellen.

Die Umwandlung einer Dezimalzahl in eine Hexadezimalzahl erfolgt wie bei den bereits besprochenen Dualzahlen mit dem Unterschied, dass durch die Zahl 16 dividiert wird. Zur Umwandlung einer Hexadezimalzahl in eine Dualzahl wird jede Ziffer der Hexadezimalzahl als Dualzahl dargestellt.

Beispiel: Hexadezimalzahl als Dezimalzahl

$(7B1)_{16}$	=	$1 * 16^0$	→	$1 * 1$	=	1	niedrigster Stellenwert
	+	$B * 16^1$	→	$11 * 16$	=	176	
	+	$7 * 16^2$	→	$7 * 256$	=	1792	höchster Stellenwert
					=	$(1969)_{10}$	Summe ergibt Dezimalzahl

Beispiel: Hexadezimalzahl als Dualzahl und umgekehrt

$(7B1)_{16}$	=	$(7)_{10}$	$(11)_{10}$	$(1)_{10}$	
	=	$(0111)_2$	$(1011)_2$	$(0001)_2$	
					= $(0111\ 1011\ 0001)_2$ Umwandlung in Dualzahl

Oktalsystem

Dieses Zahlensystem codiert drei Stellen einer Dualzahl und ist auf der Basiszahl 8 (Nennwerte: 0, 1, 2, 3, 4, 5, 6, 7) aufgebaut. Daraus ergeben sich acht verschiedene Kombinationen aus 0 und 1, von 000 bis 111. Diese Darstellungsweise wurde früher in der Datenverarbeitung verwendet, ist jedoch mittlerweile veraltet.

5.2 Programme basieren auf Daten

Analoge und digitale Daten

Bei analogen Daten handelt es sich um Daten, die durch eine kontinuierliche Funktion, wie z. B. eine Zeigerstellung ① (Uhr) oder eine Skala (z. B. Thermometer), dargestellt werden. Der Begriff **analog** steht für entsprechend bzw. vergleichbar.

Digitale (engl. digit = Ziffer) Daten werden immer durch Ziffern ② dargestellt. Innerhalb eines Computers lassen sich Daten nur digital verarbeiten.

Probleme mit digitalen Daten können z. B. bei Divisionsrechnungen auftreten:

$$(2.0 / 3.0) * 3.0 = ?$$

Die mathematische Berechnung ergibt 2. Ein Computer rechnet in der Regel als Ergebnis z. B. 1.999999998 aus, da eine digitale Zahl im Computer nur mit einer bestimmten Anzahl von Stellen dargestellt werden kann. Das analoge Ergebnis von $2/3$ lautet 0.66666, mit unendlichen Stellen nach dem Komma. Der Speicher eines Computersystems ist jedoch begrenzt. Also wird die Zahl bei der digitalen Verarbeitung an einer bestimmten Dezimalstelle abgeschnitten. Nach einer erneuten Multiplikation wird mit dem digitalen Näherungswert gerechnet, wodurch die Abweichung des Ergebnisses zustande kommt.



Vorteile des Dualsystems für die rechnerinterne Darstellung

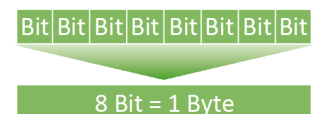
Die Verwendung eines Systems, das nur mit zwei Ziffern arbeitet, hat erhebliche Vorteile. Alle Daten, die im Dualsystem codiert sind, lassen sich problemlos und mit sehr niedriger Fehleranfälligkeit auf verschiedene Arten darstellen, weitergeben und verarbeiten. In der folgenden Übersicht finden Sie einige Beispiele der Verarbeitung und Weitergabe von digitalen Daten.

Nutzungsart	Ziffer 0 entspricht	Ziffer 1 entspricht
Speicherung mit Spannung	Keine Spannung	Ca. 6 Volt Spannung
Speicherung mit Magnetismus	Magnetisches Feld vorhanden	Veränderung der Polarität
Übertragung mit Tönen	Kurzer Ton	Langer Ton
Speicherung mit Relais	Schaltung offen	Schaltung geschlossen
Optische Speicherung auf Lochstreifen, Karte	Medium undurchsichtig (kein Loch)	Medium durchsichtig (Loch)
Optische Speicherung auf CD, DVD	Kein Übergang von oder zu einer Vertiefung	Übergang von oder zu einer Vertiefung

Bits und Bytes

Zwei wichtige Begriffe, die in der EDV immer wieder auftauchen, sind das **Bit** und das **Byte**.

- ✓ Die kleinste Speichereinheit in einem Computer oder auf einem Datenträger (z. B. Festplatte) wird als **Bit** bezeichnet. Bit ist die Abkürzung für Binary Digit, auch Binärziffer oder Dualziffer genannt.
- ✓ Zur Darstellung der internen Zeichen werden 8 Bit zu einem **Byte** zusammengefasst. Ein Byte kann genau ein Zeichen speichern. Zeichen können Ziffern, Buchstaben, Sonder- und Steuerzeichen entsprechen. Sie werden im Computer durch Zahlencodes dargestellt, z. B. $(65)_{10}$ für den Buchstaben A, $(53)_{10}$ für die Zahl 5 oder $(33)_{10}$ für das Ausrufezeichen ! beim ASCII-Code.
- ✓ Eine Folge von Bits wird als **Bitmuster** bezeichnet.



Oft benötigte Größenangaben in Byte:

Abkürzungen	1 KByte / KiB	1 MByte / MiB	1 GByte / GiB	1 TByte / TiB
Bezeichnung	1 Kilobyte	1 Megabyte	1 Gigabyte	1 Terabyte
Umrechnung in Byte	1.024 Byte	1.024 KByte	1.024 MByte	1.024 GByte
	2^{10} Byte	2^{20} Byte	2^{30} Byte	2^{40} Byte

Paritätsbit als Kontrollmöglichkeit

Bei der Übertragung von Bitmustern können technisch bedingte Fehler zu einer Veränderung des Bitmusters führen, indem beispielsweise ein Bit durch einen Schaltfehler umkippt, d. h., aus 0 wird 1 oder umgekehrt. Um solche Fehler zu erkennen, wird oft ein sogenanntes Prüfbit, auch Paritätsbit genannt, an das Bitmuster angehängt. Bei der geraden Parität ist vereinbarungsgemäß das Paritätsbit 0, wenn die Anzahl der Einsen gerade ist. Bei der ungeraden Parität ist es umgekehrt. Das Paritätsbit ist 0, wenn die Anzahl der Einsen ungerade ist.

Beispiel

Das erste Bit ist das zusätzliche Paritätsbit (PB), die übrigen acht Bit stellen das zu übertragende Bitmuster dar.

Der Operator modulo ermittelt den Rest bei der Division mit Ganzzahlen. 5 dividiert durch 2 ist 2, Rest 1.

1 10001111

PB Bitmuster

Summe der Einsen = 5

5 modulo 2 = 1 => Das Paritätsbit ist 1.

5.3 Digitales Rechnen

Im Computer liegen sämtliche Daten im Dualsystem vor, egal ob es sich um Zahlen, Speicheradressen oder um Maschinenbefehle handelt. In dieser Form werden sie gespeichert und verarbeitet.

Addieren von Dualzahlen

Die Addition von dualen Zahlen ist der schriftlichen Addition von Dezimalzahlen sehr ähnlich.

- ✓ Die Ziffern werden stellenweise von rechts beginnend addiert.
- ✓ Wenn ein Übertrag entsteht, geht dieser auf die höherwertige Stelle über.
- ✓ Die Regeln sind nebenstehend dargestellt. Der Übertrag wird beim Computer in einem speziellen Bit gespeichert, dem Carrybit.

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0, Übertrag 1

Beispiel

45 + 58 <hr/> Übertrag: 110 103	101101 + 111010 <hr/> Übertrag: 1110000 1100111
--	--

Schriftliches Addieren im Dezimalsystem

Addieren im Dualsystem

Komplementbildung

Eine sehr häufig benötigte Operation des Computers ist die Komplementbildung. Sie wird bei der Ausführung verschiedener Rechenoperationen eingesetzt, z. B. bei der Subtraktion. Es gibt zwei Arten von Komplementbildungen. Das Komplement der positiven Zahl Z mit der Basis b kann bezüglich b-1 ((b-1)-Komplement) oder bezüglich b (b-Komplement) gebildet werden. Das sind beispielsweise bei Dualzahlen (mit der Basis 2) das **1er-** und das **2er-Komplement**.

Das (b-1)-Komplement wird für gebrochene Zahlen (Zahlen mit Nachkommastellen) verwendet.

Das b-Komplement der positiven n-stelligen Zahl Z mit der Basis b wird wie folgt gebildet:

- ✓ $b^n - Z$ für $Z <> 0$
- ✓ 0 für $Z = 0$

Beispiel:

10er-Komplement der Zahl 4637	10^4	- 4637	= 5363
2er-Komplement der Zahl 11010101	2^8	- 11010101	=
	100000000	- 11010101	= 00101011

Subtraktion von Dualzahlen

Es soll der Ausdruck $Z1 - Z2$ berechnet werden. Das Verfahren der schriftlichen Subtraktion ist für digitale Computer ungeeignet. Es wird stattdessen eine Addition der Zahl $Z1$ mit dem Komplement der Zahl $Z2$ ausgeführt.

$0 - 0 = 0$
 $1 - 0 = 1$
 $0 - 1 = 1$, Übertrag -1
 $1 - 1 = 0$
 $0 - 1$ mit Übertrag -1 = 0, Übertrag -1

- ✓ Komplement der Zahl $Z2$ bilden
- ✓ Addition von $Z1$ und Komplement $Z2$
- ✓ Der Übertrag an der höchsten Stelle wird weggelassen.
- ✓ (Tritt an der höchsten Stelle kein Übertrag auf, wird vom Ergebnis der Addition das Komplement gebildet und mit einem negativen Vorzeichen versehen.)
- ✓ Der negative Übertrag wird beim Computer in einem speziellen Bit gespeichert, dem Borrowbit.

Beispiele

Aufgabe 1 (Dezimalsystem):	$58 - 45 =$
1. Komplement von 45:	$Z1 = 58; Z2 = 45$
2. Addieren	$10^2 - 45 = 100 - 45 = 55$
3. Übertrag streichen	$58 + 55 = 113$
	1 13
→ Ergebnis	$58 - 45 = 13$

Aufgabe 2 (Dualsystem):	$111010 - 101101 =$
1. Komplement von 101101:	$Z1 = 111010; Z2 = 101101$
2. Addieren	$(2^6)_{10} - 101101 = (64)_{10} - 101101 = 1000000 - 101101 = 10011$
3. Übertrag streichen	$111010 + 10011 = 1001101$
	1 001101
→ Ergebnis	$111010 - 101101 = 001101$

Aufgabe 3 (Dualsystem):	$101001 - 110001 =$
1. Komplement von 110001:	$Z1 = 101001; Z2 = 110001$
2. Addieren	$(2^6)_{10} - 110001 = (64)_{10} - 110001 = 1000000 - 110001 = 1111$
3. Kein Übertrag, daher das Komplement des Ergebnisses bilden:	$101001 + 1111 = 111000$
	$= 1000000 - 111000 = 1000$
→ Ergebnis	$101001 - 110001 = -1000$

Multiplikation von Dualzahlen

Die Multiplikation von Dualzahlen wird im Rechner durch bitweise Verschiebungen der Zahlen und Additionen realisiert. Das funktioniert ähnlich wie die schriftliche Multiplikation.

0 * 0 = 0
1 * 0 = 0
0 * 1 = 0
1 * 1 = 1

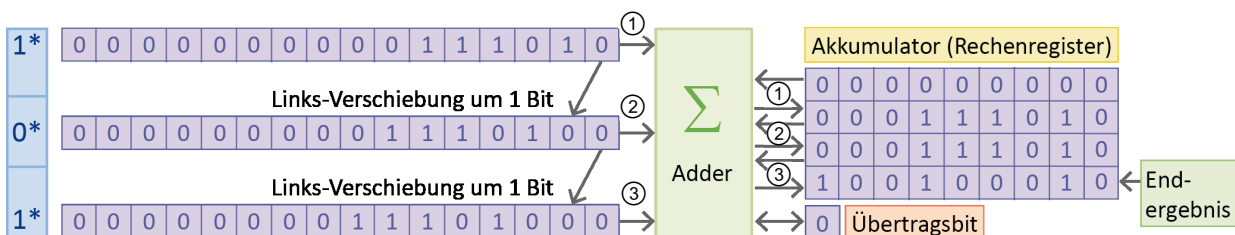
Beispiel

$\begin{array}{r} 58 * 115 \\ + \quad 58 \\ + \quad 58 \\ + \quad 290 \\ \hline 6670 \end{array}$	$\begin{array}{r} 111010 * 101 \\ + \quad 111010 \\ + \quad 000000 \\ + \quad 111010 \\ \hline 100100010 \end{array}$
---	---

Schriftliches Multiplizieren im Dezimalsystem

Multiplizieren im Dualsystem

Im Computer werden die Zwischenergebnisse der Multiplikation sofort addiert. Diese Aufgabe übernimmt der sogenannte Addierer (Adder). Die Zwischenergebnisse werden in einem Akkumulator (dem Speicher für die Rechenergebnisse) gespeichert. Das nächste Zwischenergebnis und der Inhalt des Akkumulators werden im nächsten Schritt addiert, und das Ergebnis wird im Akkumulator gespeichert. Wenn alle Zwischenergebnisse addiert wurden, steht das Endergebnis im Akkumulator. Tritt bei der Berechnung ein Übertrag auf, wird dieser in einem Übertragsbit gespeichert und fließt bei der nächsten Addition mit ein.



Multiplikation wird im Computer durch Linksverschiebung und Addition realisiert. Die Ziffern in den Kreisen kennzeichnen die Reihenfolge der Abarbeitung. Die am weitesten rechts stehende Ziffer des Multiplikators wird als Erste zur Berechnung herangezogen, gerade umgekehrt wie bei der schriftlichen Multiplikation.

5.4 Zeichencodes

Ein Zeichencode stellt eine Zuordnungsregel dar, mit deren Hilfe jedes Zeichen eines Zeichenvorrates eindeutig durch bestimmte Zeichen eines anderen Zeichenvorrates dargestellt werden kann. Zur Darstellung von Zeichen in Computern wurden verschiedene Zeichencodes entwickelt und teilweise standardisiert, z. B. ASCII-Code und Unicode.

Der ASCII-Code

ASCII steht für **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange (Amerikanische Standardverschlüsselung für Datenaustausch). Der ASCII-Code wurde 1967 als 7-Bit-Zeichencode standardisiert. In der Weiterentwicklung wurde der ASCII-Code auf 8 Bit erweitert. Als Standard-ASCII-Code sind nur die 128 Zeichen des 7-Bit-Codes festgelegt. Mit 7 Bit können genau $2^7 = 128$ Zeichen dargestellt werden. Der Zeichencode umfasst druckbare Zeichen wie Buchstaben, Ziffern und Sonderzeichen und nicht druckbare Zeichen wie Übertragungs-, Format-, Geräte- und Informationssteuerungszeichen.

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	\$	37	%	38	&	39	'
40	(41)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60		61		62		63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[92	\	93]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	DEL

Die ASCII-Code-Tabelle (jeweils rechts der Dezimalwert des ASCII-Zeichens)

Die wichtigsten, nicht darstellbaren Zeichen sind folgende:

9	HT	horizontal tabulation – Horizontaltabulator
10	NL	new line – neue Zeile
13	CR	carriage return – Wagenrücklauf (analog zur Schreibmaschine)
127	DEL	delete – Löschen, Entfernen



Im Windows-Betriebssystem erhalten Sie das zugehörige ASCII-Zeichen, indem Sie bei gedrückter **[Alt]**-Taste den ASCII-Wert über den numerischen Tastaturblock eingeben.

EBCDI-Code

EBCDI steht für **E**xtended **B**inary-**C**oded **D**ecimal **I**nterchange. Dieser 8-Bit-Code wird vor allem auf Großrechnern verwendet, die in der kommerziellen Datenverarbeitung eingesetzt werden. Er wurde von IBM entwickelt und existiert in mehreren Varianten, um Sprachunterschiede auszugleichen. Für eine Codeumwandlung von ASCII-Code in EBCDI-Code müssen Kodierungstabellen verwendet werden.

ANSI-Code

Um Zeichen wie ä, ö, ü, æ und ê darstellen zu können, wurde der 8-Bit-ANSI-Code entwickelt. Die Abkürzung ANSI steht für **A**merican **N**ational **S**tandards **I**nstitute. Das ANSI ist das nationale Institut für Normung im Datenverarbeitungsbereich in den USA.



Der ANSI-Code ist in großen Teilen mit dem ASCII-Code identisch. Da aber z. B. die Umlaute (ä, ö, ü) im ASCII-Code nicht enthalten sind, treten bei Textkonvertierungen zwischen verschiedenen Betriebssystemen (z. B. MS-DOS (ASCII) und Microsoft Windows (ANSI)) Probleme auf.

Unicode

Unicode ist ein Zeichencode, in dem fortlaufend Zeichen, beispielsweise aller bekannten Sprachen weltweit, codiert werden. In der aktuellen Version 10.0 (Stand: Juni 2017) von Unicode sind mehr als 135 000 Zeichen codiert.

Unicode vereint Zeichencodes. So können z. B. hebräische und lateinische Buchstaben ohne Wechsel des Zeichensatzes geschrieben werden. Steuerzeichen wie Silbentrennungszeichen, erzwungene Leerzeichen oder Tabulatorzeichen sowie Zeichen mathematischer Formeln sind ebenfalls in Unicode enthalten.

Das Unicode-System ist konform zur internationalen Norm ISO/IEC 10646. Erstellt wurde es vom Unicode-Konsortium und einer ISO-Arbeitsgruppe (**ISO** – **I**nternational **S**tandard **O**rganization). Zu weiteren Informationen vgl. auch <http://www.unicode.org>.

5.5 Übung

Fragen zu Zahlensystemen und Zeichencodes

Übungsdatei: --

Ergebnisdatei: *uebung05.pdf*

1. Erstellen Sie eine Tabelle mit vier Spalten. Tragen Sie in die erste Spalte die Dezimalzahlen von 1 bis 16 ein. In die zweite Spalte sind dann die entsprechenden Dualzahlen, in die dritte Spalte die Oktalzahlen und in die vierte Spalte die Hexadezimalzahlen einzutragen.
2. Wandeln Sie die Dezimalzahl 12345 in eine hexadezimale Zahl um.
3. Geben Sie die hexadezimale Zahl 1F binär im Dualsystem an.
4. Welchen dezimalen Wert hat die Oktalzahl 1357?
5. Rechnen Sie die Dezimalzahl 735 in das Dualsystem um.
6. Stellen Sie die Binärzahl 1101 1100 1011 1010 in hexadezimaler Form dar.
7. Was verstehen Sie unter den Begriffen Bit und Byte?
8. Addieren Sie die Dualzahlen 11011011 und 1010011001.
9. Lösen Sie folgende Aufgaben: 1101111 - 111111 und 1011000 - 1100011.
10. Multiplizieren Sie die Zahlen 1011110 und 110010.
11. Rechnen Sie um: 3600 Byte in KByte sowie 1,44 MByte in Byte.

6 Grundlegende Sprachelemente

In diesem Kapitel erfahren Sie

- ✓ was Syntax und Semantik bedeuten
- ✓ was Datentypen, Variablen und Konstanten sind
- ✓ welche Operatoren es gibt
- ✓ was unter Ausdrücken zu verstehen ist

6.1 Syntax und Semantik

Programmiersprachen sind, wie auch natürliche Sprachen, nach definierten Regeln aufgebaut. Diese Regeln legen fest, welche Zeichen verwendet werden dürfen, wie die Zeichen angeordnet sein müssen und welche Bedeutung bestimmte Zeichenfolgen haben.

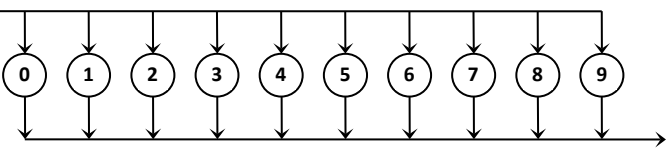
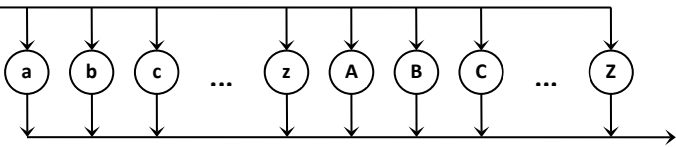
- ✓ Die **Syntax** einer Sprache bestimmt den Aufbau der Sätze. Auf Programmiersprachen bezogen legt die Syntax z. B. fest, wie Anweisungen aufgebaut sind.
- ✓ Die **Semantik** erklärt die Bedeutung der Sätze. Auf Programmiersprachen bezogen wird durch die Semantik z. B. beschrieben, was eine Anweisung bedeutet.

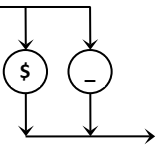
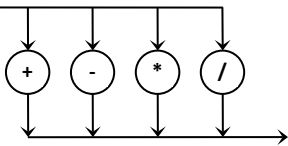
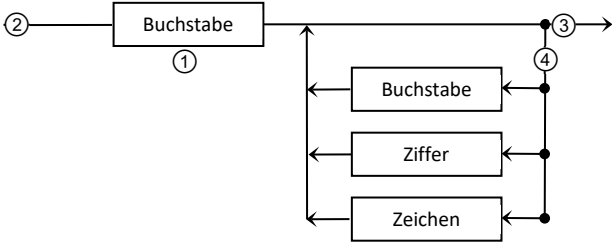
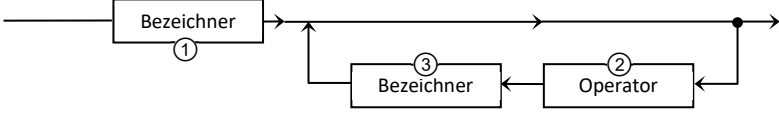
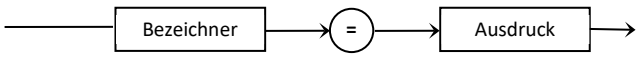
Die Syntax einer Sprache kann mithilfe von Syntaxdiagrammen oder der Backus-Naur-Form dargestellt werden. In Syntaxdiagrammen wird die Grammatik einer Sprache grafisch dargestellt und ist dadurch leichter lesbar. Die Backus-Naur-Form (BNF) verwendet eine textuelle Darstellung der Syntax und ist dadurch mit jedem Texteditor zu erfassen. In Sprachbeschreibungen wird häufig die erweiterte Backus-Naur-Form (EBNF) verwendet, die im Vergleich zur BNF mehr Möglichkeiten bietet.

Darstellung der Syntaxdiagramme

Die Syntaxdiagramme einer Sprache sind sehr umfangreich, da alle Konstrukte einer Sprache beschrieben werden. Das folgende Beispiel zeigt, wie ein numerischer oder textbasierender Ausdruck einer Sprache dargestellt werden kann.

Beispiel: Syntaxdiagramme

Syntaxbegriff	Syntaxdiagramm und Bedeutung
<Ziffer>	 <p>Eine Ziffer kann ein Wert von 0 bis 9 sein.</p>
<Buchstabe>	 <p>Ein Buchstabe kann jeder kleine und große Buchstabe des Alphabets sein.</p>

Syntaxbegriff	Syntaxdiagramm und Bedeutung
<Zeichen>	 <p>Ein Zeichen kann entweder $\\$ oder $-$ sein.</p>
<Operator>	 <p>Ein Operator ist eines der Zeichen $+$, $-$, $*$ oder $/$.</p>
<Bezeichner>	 <p>Ein Bezeichner muss mit einem Buchstaben ① beginnen. Anschließend können weitere Bestandteile folgen oder nichts. Dies wird durch die Pfeile und die Linien veranschaulicht. Wenn Sie die Linie vom Startpunkt ② aus verfolgen, kommen Sie auf jeden Fall zum Buchstaben ①. Vom Buchstaben aus gelangen Sie an einen Knotenpunkt ③. Verfolgen Sie die Linie geradeaus weiter, erreichen Sie das Ende (der Bezeichner besteht in diesem Fall aus genau einem Buchstaben). Verfolgen Sie aber die vertikale Linie ④, gelangen Sie entweder zu einem weiteren Buchstaben, einer Ziffer oder einem Zeichen. Danach kehren Sie wieder auf die ursprüngliche Linie und somit zu dem Knotenpunkt zurück. Nun können Sie den Weg beenden oder wieder den vertikalen Weg gehen.</p>
<Ausdruck>	 <p>Ein Ausdruck kann ein Bezeichner ① sein oder ein Bezeichner ①, gefolgt von einem Operator ② und einem Bezeichner ③. Ein Ausdruck kann aber auch mehr als zwei (beliebig viele) Bezeichner, die jeweils durch Operatoren miteinander verknüpft sind, beinhalten.</p>
<Zuweisung>	 <p>Eine Zuweisung besteht aus einem Bezeichner, dem Zuweisungsoperator $=$ und einem Ausdruck.</p>

Das Beispiel bezieht sich auf keine spezielle Sprache, sondern soll das Darstellungsprinzip verdeutlichen.



Darstellung der EBNF

Symbol	Beschreibung
<begriff>	Syntaxbegriff
::=	„ist definiert als“ oder linke Seite „kann ersetzt werden durch“ rechte Seite
{ }	Der in Klammern eingeschlossene Teil kann beliebig oft wiederholt werden oder entfallen.
[]	Der in eckigen Klammern eingeschlossene Teil ist optional (kann auch entfallen).
<a> 	Alternative (logisches Oder) bedeutet: <a> <i>oder</i> wird verwendet.
"x"	Zeichen oder Wörter in Anführungszeichen sind Bestandteile der Sprache und müssen genau so geschrieben werden.

Beispiel: EBNF

Die oben aufgeführten Syntaxdiagramme wurden hier in die EBNF umgesetzt.

```

<ziffer>      ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
<buchstabe>   ::= "a"|"b"|"c"|\...|"z"|"A"|"B"|"C"|\...|"Z"
<zeichen>    ::= "$"|"_"
<operator>   ::= "+"|"-"|"*"|"\/"
<bezeichner> ::= <buchstabe>{<buchstabe>|<ziffer>|<zeichen>}
<ausdruck>   ::= {<bezeichner> <operator>} <bezeichner>
<zuweisung> ::= <bezeichner> "=" <ausdruck>

```

Die Beschreibung einer Sprache in der EBNF muss immer bis in die kleinste syntaktische Einheit (Zeichen der Sprache) auflösbar sein. Bevor ein Ausdruck, wie

```

<alternative> ::= "if (" <logischer ausdruck> ") " <anweisung> ["else"
                                     <anweisung>]

```

angegeben werden kann, müssen zuvor die Syntaxbegriffe <logischer ausdruck> und <anweisung> erklärt sein.

Hinweis zur Schreibweise der Syntax im Buch

- ✓ Schlüsselwörter werden fett hervorgehoben.
- ✓ Optionale Angaben stehen in eckigen Klammern [] .
- ✓ Drei Punkte (...) kennzeichnen, dass weitere Angaben folgen können.
- ✓ Sofern eckige Klammern oder drei Punkte (...) als Bestandteil des Quelltextes erforderlich sind, wird darauf in der Erläuterung explizit hingewiesen.

6.2 Grundlegende Elemente einer Sprache

Alle Sprachen werden durch ihre eigene Syntax und Semantik beschrieben. Sprachen können sich z. B. hinsichtlich des Aufbaus von Bezeichnern, der zulässigen Datentypen und der verfügbaren Operationen unterscheiden. Einige Elemente oder Mechanismen sind Bestandteil vieler Sprachen und sich sehr ähnlich. Diese wurden bei der Verwendung in formalen Syntaxdiagrammen angesprochen und werden im weiteren Verlauf des Buchs exemplarisch wie auch in Verbindung mit konkreten Programmiersprachen (insbesondere Python, aber gelegentlich auch Java) erklärt.

Bezeichner

Bezeichner (engl. identifier) geben Dingen in Programmen, z. B. Variablen, einen Namen, über den sie im Programm angesprochen werden können. Die Anzahl der Zeichen und welche Zeichen zulässig sind, ist in den Sprachen verschieden. Einige Sprachen unterscheiden auch zwischen Groß- und Kleinschreibung (**case-sensitive**).

In Programmen können Sie als Bezeichner beispielsweise sowohl englische als auch deutsche Namen wählen. Innerhalb eines Programms sollte aber eine einheitliche Schreibweise gewählt werden. Englische Namen haben den Vorteil, dass sie oft kürzer sind, keine Umlaute enthalten, zu den reservierten Wörtern passen und, falls Programme international eingesetzt werden sollen, besser verständlich sind.

Bezeichner in Python

Betrachten wir Python als konkretes Beispiel für Bezeichner. Für den Aufbau eines Bezeichners in Python gelten folgende Regeln:

- ✓ Bezeichner müssen mit Buchstaben oder '_' beginnen und können dann mit Zahlen fortgesetzt werden.
- ✓ Bezeichner beinhalten keine Leerzeichen und keine Sonderzeichen.
- ✓ Python unterscheidet zwischen Groß- und Kleinschreibung. `Name`, `NAME` und `name` sind drei unterschiedliche Bezeichner für Elemente in Python.
- ✓ Man vermeidet deutsche Umlaute oder das ß. Man sollte also nur Buchstaben aus dem ASCII-Zeichensatz, Zahlen oder den Unterstrich für diese Bezeichner verwenden.
- ✓ Python verwendet den Bezeichner in seiner kompletten Länge (alle Stellen sind signifikant – bis auf technische Begrenzungen).

Es ist oft sehr wichtig, möglichst aussagekräftige Bezeichner für Variablen, Funktionen, Methoden, Klassen etc. zu wählen. In professionellen Projekten gibt es dazu meist streng verbindliche Vorgaben, aber auch sonst sollten Sie feste Regeln einhalten und etwa bei Variablen die Bedeutung und/oder den Datentyp in den Bezeichner einfließen lassen (z.B. `zahl1`, `wertNumber`, ...) oder bei Funktionen die Bedeutung (etwa `Rechnen`, `Ausgabe`, `Eingabe`, ...). Wobei das im Widerspruch dazu steht, dass man möglichst kurze Bezeichner wählen sollte. Oft wählt man einen Kompromiss. Etwa `BtnOK` für einen Button (eine Schaltfläche), der mit `OK` beschriftet ist.

Reservierte Wörter

In jeder Sprache ist eine Anzahl von Wörtern als Schlüsselwörter definiert. Diese reservierten Wörter haben in der Programmiersprache eine spezielle Bedeutung.

Alle reservierten Wörter in Python

<code>False</code>	<code>def</code>	<code>if</code>	<code>raise</code>
<code>None</code>	<code>del</code>	<code>import</code>	<code>return</code>
<code>True</code>	<code>elif</code>	<code>in</code>	<code>try</code>
<code>and</code>	<code>else</code>	<code>is</code>	<code>while</code>
<code>as</code>	<code>except</code>	<code>lambda</code>	<code>with</code>
<code>assert</code>	<code>finally</code>	<code>nonlocal</code>	<code>yield</code>
<code>break</code>	<code>for</code>	<code>not</code>	
<code>class</code>	<code>from</code>	<code>or</code>	
<code>continue</code>	<code>global</code>	<code>pass</code>	

- ✓ Die beiden großgeschriebenen reservierten Wörter (`False` und `True`) stellen konstante vordefinierte Werte dar und werden als **Literale** bezeichnet. Alle anderen Schlüsselwörter in Python werden kleingeschrieben.

- ✓ Man kann mit booleschen Literalen `True` und `False` rechnen, denn diese repräsentieren in Python die Zahlen 1 und 0.
- ✓ Alle anderen hier aufgeführten reservierten Wörter sind sogenannte **Schlüsselwörter**.
- ✓ Alle reservierten Wörter dürfen **nicht** als Bezeichner verwendet werden.

Kommentare

Zur Dokumentation eines Quelltextes verwenden Sie Kommentare. Diese sollen Ihnen und anderen Mitarbeitern an einem Projekt unter anderem helfen, den Quelltext auch nach längerer Zeit noch zu verstehen. Kommentare sollten beispielsweise verwendet werden, um

- ✓ den Zweck von Variablen zu beschreiben,
- ✓ die Verwendung einer Methode (bzw. Funktion oder Prozedur) zu erläutern,
- ✓ einen nicht selbsterklärenden Algorithmus zu beschreiben oder
- ✓ Teile vom Quelltext auszukommentieren (beispielsweise im Fall der Fehlersuche).



Durch Kommentare wird im Compiler-Fall die Größe des ausführbaren Programms nicht erhöht. Kommentare werden vom Compiler nicht berücksichtigt. Da im Interpreter-Fall der Quellcode jedoch zur Laufzeit abgearbeitet wird, ist dieser durch Kommentare umfangreicher als ohne Kommentare.

Die verschiedenen Programmiersprachen lassen unterschiedliche Arten von Kommentaren zu. Kommentare werden durch spezielle für die Programmiersprache festgelegte Zeichen eingeleitet bzw. eingeschlossen.

Beispiel: Kommentartypen in Python

Einzeilige Kommentare	#	Kommentar bis zum Ende der Zeile Alle Zeichen der Zeile hinter # werden vom Compiler/Interpreter überlesen. Die meisten anderen Programmiersprachen verwenden <code>//</code> als Kennzeichnung eines einzeiligen Kommentars.
(Mehrzeiliger) Kommentarblock	<code>'''</code> <code>...</code> <code>'''</code>	Für mehrzeilige Kommentare setzt man drei einfache oder auch doppelte Hochkommata. Diese müssen dann aber auch wieder abgeschlossen werden. Die meisten anderen Programmiersprachen verwenden <code>/* ... */</code> als Kennzeichnung eines mehrzeiligen Kommentars.

6.3 Standarddatentypen (elementare Datentypen)

In einem Programm verarbeiten Sie Daten, die sich in ihrer Art unterscheiden:

- ✓ Zahlen – numerisch
- ✓ Zeichen – alphanumerisch
- ✓ boolesche Daten – logisch

Ein **Datentyp** ist eine Menge darstellbarer Werte. Viele Programmiersprachen besitzen vordefinierte elementare Datentypen, auch **primitive Datentypen** genannt. Diese Datentypen unterscheiden sich in der Art der Daten und in dem zulässigen Wertebereich. Ebenso unterscheiden sich Programmiersprachen darin, welche Datentypen sie bereitstellen und wie diese genau implementiert sind.



Objektorientierte Sprachen verwenden oft auch einen Objekt-Datentyp. Dieser wird in der Regel als **Referenztyp** umgesetzt. Variablen eines Referenztyps stellen Referenzen (pointer – engl. für Zeiger) dar. Dies bedeutet, dass die gesuchte Information an einer anderen Stelle steht, nämlich an der Stelle, auf welche die Referenz verweist.

Numerische Datentypen

Numerische Datentypen lassen sich, wie in der Mathematik, in ganze Zahlen (Integer) und reelle Zahlen (Real) unterteilen. Dabei sind die Wertebereiche und die Genauigkeit der Zahlen durch die Festlegung des Speicherbereichs begrenzt. Numerische Datentypen werden z. B. für Berechnungen eingesetzt.

Integer-Datentypen

Integer-Datentypen sind **Ganzzahlen** und besitzen keine Nachkommastellen. Einige Programmiersprachen bieten mehrere Integer-Datentypen an, die sich durch die Bezeichnung und die Größe des Wertebereichs unterscheiden. Die Größe des Wertebereichs ist von der Größe des Speicherplatzes abhängig, der für den Datentyp vorgesehen ist, z. B. 2, 4 oder 8 Byte. Andere Sprachen (etwa Python) haben nur einen Typ für ganze Zahlen.

Der kleinste bzw. größte darstellbare Wert eines **vorzeichenlosen Integer-Datentyps** kann wie folgt berechnet werden. Für x wird dabei die zulässige Speichergröße in Bit eingesetzt: $0 \dots 2^x - 1$

Für **Integer-Datentypen mit Vorzeichen** kann folgende Formel verwendet werden: $-2^{x-1} \dots 2^{x-1}-1$
Ein Bit wird dabei für die Speicherung des Vorzeichens verwendet.

Beispiel: Integer-Datentypen in Python

Ganzzahlige Werte werden in Python durch den Datentyp `int` repräsentiert.

Beispiel:

```
n1 = 78829
```

- ✓ Integer-Datentypen werden im Computer immer genau dargestellt.
- ✓ Es treten keine Rundungsfehler bei der Darstellung der Zahlen auf.

Gleitkomma-Datentypen

Für **Fließkommazahlen** (Dezimalzahlen, Gleitkommazahlen oder Gleitpunktzahlen beziehungsweise Floating-Point-Zahlen genannt) werden Gleitkomma-Datentypen mit Vorzeichen verwendet. Der Computer kann jedoch nicht jede Zahl genau darstellen. Dies führt auch bei einfachen Rechnungen zu Rundungsfehlern. Je nach verwendetem Typ lassen sich nur Zahlen mit einer bestimmten Genauigkeit abbilden. Für eine höhere Genauigkeit wird aber auch mehr Speicherplatz benötigt. Einige Programmiersprachen besitzen zwei Gleitkomma-Datentypen: Single Precision (einfache Genauigkeit) und Double Precision (doppelte Genauigkeit). Andere Sprachen (wie etwa Python) haben für den Zweck nur einen Datentyp. Einige Sprachen bieten zusätzlich eine Extended Precision (höhere Genauigkeit).

Zahlen mit Nachkommastellen werden in Python durch den Datentyp `float` repräsentiert. Die Nachkommastellen werden dabei – wie im englischen Sprachraum üblich – in nahezu allen Programmiersprachen nicht durch ein Komma, sondern durch einen Punkt von dem ganzzahligen Anteil getrennt (man spricht deshalb von Floating-Point-Zahlen). Zudem ist es möglich, sehr große oder sehr kleine `float`-Zahlen mittels `e` oder `E` in Exponential-Schreibweise anzugeben. Die Zahl hinter dem `e` gibt dabei an, um wie viele Stellen der Dezimalpunkt innerhalb der Zahl verschoben wird.

```
n2 = 4.3e5 # 430000
```

Komplexe Datentypen

Etwas ungewöhnlich in Python ist, dass es einen extra Datentyp für **komplexe Zahlen** gibt – in den meisten anderen Programmiersprachen gibt es so einen Datentyp nicht. Diese bestehen aus einem Realteil und einem Imaginärteil. Der Imaginärteil besteht aus einer reellen Zahl, die mit der imaginären Einheit `i` multipliziert wird. Beachten Sie, dass die imaginäre Einheit in Python durch ein `j` statt einem `i` dargestellt wird. Um eine komplexe Zahl zu definieren, gibt man in Python etwa Folgendes ein:

```
n4 = 4 + 3j
```

Zeichen-Datentyp, Zeichenketten und Zeichenliterale

In vielen Programmiersprachen gibt es Zeichen-Datentypen. Diese können beliebige Zeichen enthalten. Es ist von der Sprache abhängig, ob Zeichen intern als ASCII- oder Unicode-Zeichen dargestellt werden.

Zeichenketten beziehungsweise Strings sind eine Folge von Zeichen, die wahlweise in einfachen oder doppelten Anführungszeichen geschrieben werden und in Python vom Typ `str` sind.

Beispiel:

```
str1 = 'Hallo'
str2 = "Welt!"
```

Die einzelnen Zeichen innerhalb einer Zeichenkette sind **Zeichenliterale**, die auch **maskiert** werden können. Das bedeutet, dass man eine kodierte Darstellung verwendet – die **Escape-Darstellung**.

- ✓ Die Escape-Zeichenliterale beginnen immer mit dem Backslash-Zeichen.
- ✓ Diesem folgt eines der Zeichen (b, t, n, f, r, ", ' oder \) oder eine Serie von Ziffern, denen ein Zeichen für die Art der Darstellung vorangeht.

In der nachfolgenden Tabelle sehen Sie einige Beispiele für besondere Zeichen in verschiedenen Darstellungen:

Escape-Literal	Bedeutung
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\	Backslash
\b	Rückschritt (Backspace)
\f	Formularvorschub (Formfeed)
\n	Neue Zeile (New line)
\r	Wagenrücklauf (Return)
\t	Tabulatur (Tab)
\a	ASCII Bell – Klingeln
\uxxx	Ein Zeichen im Unicode (UTF-16) mit drei hexadezimalen Zahlen. Die Darstellung macht auf einigen Python-Plattformen Schwierigkeiten.
\Uxxxxxxxx	Ein Zeichen im Unicode (UTF-32) mit acht hexadezimalen Zahlen. Die Darstellung macht auf einigen Python-Plattformen Schwierigkeiten.
\ooo	Ein Zeichen in der Oktal-Darstellung mit drei oktalen Zahlen.
\xhh	Ein Zeichen in der Hex-Darstellung mit zwei hexadezimalen Zahlen.

Logischer Datentyp

George Boole entwickelte im 19. Jahrhundert eine nach ihm benannte Algebra, die boolesche Algebra. Der Grundgedanke ist, dass es nur die beiden Wahrheitswerte *wahr* (true) und *falsch* (false) gibt. Eine solche Variable, die nur diese beiden Werte annehmen kann, ist vom logischen Datentyp, der in vielen Sprachen `boolean` oder `bool` (Python) genannt wird.

Nicht in allen Sprachen gibt es einen speziellen logischen Datentyp. Er wird dort meist mithilfe von ganzzahligen Werten verwaltet. Dabei steht 0 für *falsch* und 1 oder gar jeder andere Wert für *wahr*. Einige Sprachen (etwa JavaScript, Python oder PHP) stellen einen logischen Datentyp zur Verfügung, interpretieren aber auch in Vergleichssituationen numerische Werte als *wahr* und *falsch*.



Nahezu alle Programmiersprachen erlauben die Abfrage eines Datentyps. In Python kann der Datentyp eines Objekts oder einer Variablen jederzeit mittels der Built-in-Funktion `type()` ermittelt werden.



6.4 Literale für primitive Datentypen

Werte, die Sie im Quelltext eingeben (z. B. Zahlen), werden als **Literale** bezeichnet. Für deren Schreibweise gelten entsprechende Regeln, damit der Datentyp ersichtlich ist.

Beispiel: Literale für numerische Datentypen in Python

- ✓ Für numerische Werte des Datentyps `int` können Sie beispielsweise die Vorzeichen `+` bzw. `-` und die Ziffern `0` ... `9` verwenden (das Vorzeichen `+` kann entfallen).
- ✓ Als Dezimaltrennzeichen bei Fließkommawerten verwenden Sie einen Punkt `.`.

Beispiel: Literale für booleschen Datentyp in Python

Für die Eingabe eines logischen Wertes existieren lediglich die beiden Literale `True` (wahr) und `False` (falsch). Beachten Sie den Sonderfall, dass diese großgeschrieben werden.

6.5 Variablen und Konstanten

In einer Programmiersprache werden **Variablen** benötigt, um Daten darin zu speichern. Variablen haben einen Namen, über den sie im Programm angesprochen werden, z. B. `Kilometerstand`, um darin die Anzahl der gefahrenen Kilometer eines Autos zu speichern.

Bei der Namensgebung der Variablen sind die Vorgaben für Bezeichner einzuhalten. Oft gibt es zusätzliche Empfehlungen für die Namensgebung, z. B. durch die Programmiersprache selbst, durch Projektabsprachen oder Firmenrichtlinien.



In vielen Programmiersprachen können in Variablen nur Daten eines bestimmten Datentyps gespeichert werden. Ein Datentyp gibt an, welche Art von Werten in der Variablen gespeichert werden dürfen; im Beispiel der Variablen `Kilometerstand` sollen Zahlen gespeichert werden. Das ist beispielsweise in Java, C oder C# der Fall. Bei anderen Sprachen wie Python, JavaScript oder PHP kann sich der Datentyp einer Variablen ändern.

Der Wert der Variablen kann sich im Laufe des Programms ändern, er ist variabel.

Gültigkeitsbereich von Variablen

Variablen können in einem Programm verschiedene Gültigkeitsbereiche besitzen, d. h., Sie können auf diese nur in bestimmten Programmbereichen zugreifen.

Lokale Variablen	Variablen, die innerhalb von Funktionen und Anweisungsblöcken vereinbart werden, können Sie auch nur dort ansprechen. Die Lebensdauer lokaler Variablen endet mit dem Verlassen des Blocks, in dem sie deklariert wurden. In anderen Funktionen sind diese Variablen nicht bekannt.
-------------------------	---

Statische Variablen	Auch wenn der Gültigkeitsbereich der statischen Variablen verlassen wird, behalten diese den aktuellen Wert bei. Statische Variablen haben eine Lebensdauer über die gesamte Programmausführung hinweg. Sie ähneln globalen Variablen, sind aber in objektorientierten Sprachen an Klassen gebunden.
Externe Variablen	Wenn eine Variable als extern deklariert wird, bedeutet dies, dass die Variable in einer anderen Datei oder Bibliothek schon vorhanden ist und Sie diese nutzen möchten. Externe Variablen besitzen ebenfalls eine Lebensdauer über das gesamte Programm.
Globale Variablen	Dies sind Variablen, die einmalig außerhalb von Funktionen deklariert werden. Sie können in allen Funktionen angesprochen werden und besitzen eine globale Gültigkeit und eine Lebensdauer über die gesamte Programmausführung hinweg.



Nicht alle Programmiersprachen unterstützen jeden dieser Variablentypen. So kennt Java beispielsweise keine globalen Variablen. JavaScript hingegen kennt keine statischen Variablen.



In Python gibt es die Besonderheit, dass globale Variablen erst einmal **nicht** in einer Funktion zur Verfügung stehen. Man muss sie über das Schlüsselwort `global` in der Funktion bekannt geben. Erst dann kann man über den Bezeichner in der Funktion auf die globale Variable zugreifen.

Deklaration und Typisierung

In fast allen Programmiersprachen muss eine Variable, bevor sie in einem Programm verwendet werden kann, deklariert werden. Mit der **Deklaration** wird der Name (Bezeichner) der Variablen festgelegt.

Das Konzept der dabei verwendeten **Typisierung** bezeichnet ein Prinzip zur Bestimmung der Art einer Variablen, das allerdings nicht eindeutig definiert ist. In der Fachliteratur finden sich unterschiedlich strenge Definitionen. Man kann aber generell zwischen Sprachen ohne oder mit sehr schwacher Typisierung (z. B. JavaScript oder PHP) und solchen mit stärkerer Typisierung (u. a. Java oder C++) unterscheiden.

- ✓ In Sprachen mit einer **strengen Typisierung** (auch **starke Typisierung** genannt) wie Java, C oder C# wird zu dem Bezeichner bei der Deklaration mit spezieller Syntax der Datentyp bestimmt und über den Datentyp der Speicherplatz festgelegt, den dieser Datentyp bei der Programmausführung benötigt. Der Datentyp kann sich nicht mehr ändern.
- ✓ In anderen Sprachen wie Python, JavaScript oder PHP erfolgt die Festlegung des Datentyps jedoch implizit durch den Datentyp des zugewiesenen Wertes. Dabei werden Variablen ohne Typangabe deklariert. Das nennt man **lose Typisierung**, und oft wird hier auch von einer **dynamischen Typisierung** gesprochen, wobei man auch dabei unterschiedlich vorgehen kann.

Bei JavaScript oder PHP werden etwa die Datentypen von Variablen direkt aus dem zugewiesenen Wert geschlossen, und die Datentypen können sich auch nachträglich ändern.

Python, das im Hintergrund ausschließlich mit Objekttypen arbeitet, verfolgt etwa das Konzept des **Duck-Typings**, bei dem der Typ eines Objektes nicht durch seine Klasse beschrieben wird, sondern durch das Vorhandensein bestimmter Methoden oder Attribute. Beim Duck-Typing wird zur Laufzeit des Programms geprüft, ob ein Objekt die entsprechenden Merkmale unterstützt. Dies führt wie bei allen dynamischen Typsystemen zu einer erhöhten Flexibilität, reduziert aber ebenso die Möglichkeit, statisch zur Übersetzungszeit Fehler im Programm zu finden. Aber unter der Oberfläche ist dieses System typsicher, weil sich der Datentyp einer Variablen nicht wirklich ändern kann.

Für die Deklaration von Variablen gibt es allgemein einige verbindliche Regeln.

- ✓ Alle Variablennamen müssen im jeweiligen Gültigkeitsbereich eindeutig sein. Sie können z. B. in Python denselben Variablennamen in mehreren Anweisungsblöcken verwenden, da sich die Gültigkeitsbereiche von aufeinander folgenden Anweisungsblöcken nicht überschneiden.
- ✓ Globale Variablen müssen hingegen einen eindeutigen Namen besitzen, da sie für das gesamte Programm gelten.

Wertzuweisung und Initialisierung

Wenn Sie eine globale, statische oder externe Variable deklarieren, wird vom Programm ein bestimmter Speicherbereich bereitgestellt, dessen Größe vom entsprechenden Datentyp abhängig ist.

Bevor Sie auf den Wert einer Variablen zugreifen können, müssen Sie dieser Variablen einen Wert zuweisen. Bei der Deklaration einer Variablen geschieht dies nicht immer automatisch. Die erste Wertzuweisung nach der Deklaration einer Variablen wird **Initialisierung** genannt. Bevor Sie eine Variable auf der rechten Seite einer Anweisung nutzen, muss dieser ein gültiger Wert zugewiesen worden sein. Um einer Variablen einen Wert zuzuweisen, verwenden Sie den **Zuweisungsoperator**, der je nach Sprache durch `:` `=` oder wie in Python, Java, JavaScript oder C durch `=` dargestellt wird.

Weisen Sie jeder Variablen einen Anfangswert zu, bevor Sie diese das erste Mal benutzen, auch wenn einige Programmiersprachen Variablen automatisch initialisieren. So beugen Sie Fehlern vor und können sicher sein, dass die Variable den gewünschten Wert für weitere Berechnungen hat.



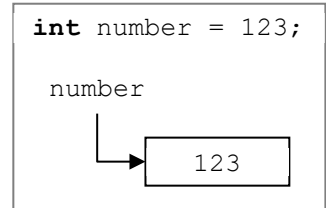
Einer Variablen in einer streng typisierten bzw. typsicheren Sprache dürfen nur Werte zugewiesen werden, die dem vereinbarten Datentyp entsprechen. Möchten Sie einer Variablen in solchen Sprachen den Wert einer anderen Variablen zuweisen, die einen anderen Datentyp besitzt, muss der Wert in den Zieldatentyp konvertiert werden. Dazu besitzen die Sprachen entsprechende Funktionen oder führen auch interne Typumwandlungen durch.

Beispiel: Variable in Java deklarieren und initialisieren

Syntax der Deklaration	<code>type identifier[, identifier1...];</code>
Syntax der Deklaration und Initialisierung	<code>type identifier = value;</code>

Es wird eine ganzzahlige Variable `number` vom Datentyp `int` deklariert und mit dem Wert 123 initialisiert. Dabei wird zunächst ein entsprechend großer Speicherplatz für die Variable reserviert und dann der Wert 123 dort eingetragen.

Über den Namen `number` können Sie später im Programm auf den gespeicherten Wert zugreifen.

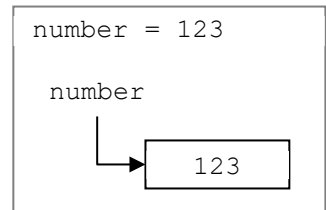


Beispiel: Variable in Python deklarieren und initialisieren

Syntax der Deklaration und Initialisierung	<code>identifier = value</code>
--	---------------------------------

Es wird eine ganzzahlige Variable `number` deklariert und mit dem Wert 123 initialisiert. Dabei wird zunächst ein entsprechend großer Speicherplatz für die Variable reserviert und dann der Wert 123 dort eingetragen. Der Datentyp des Literals legt indirekt den Datentypen der Variablen fest.

Über den Namen `number` können Sie später im Programm auf den gespeicherten Wert zugreifen.



Beispiel: *VariableDefinition.py*

```
# Richtige Variablendeklarationen
number = 0 # deklariert eine Variable number vom Typ int
print(type(number), number, sep=",")

price = 0.0 # deklariert eine Variable price vom Typ float
print(type(price), price, sep=",")

size = 0e0 # deklariert eine Variable size vom Typ float
print(type(size), size, sep=",")

name = "" # deklariert eine Variable name vom Typ str
print(type(name), name, sep=",")

inumber = 0 + 0j # deklariert eine Variable inumber vom Typ complex
print(type(inumber), inumber, sep=",")

wahr = True # deklariert eine Variable wahr vom Typ bool
print(type(wahr), wahr, sep=",")

# Fehlerhafte Variablendeklarationen - auskommentiert
'''
&count = 42 # Bezeichner von Variablen dürfen kein & enthalten
1number = 42 # Bezeichner von Variablen dürfen nicht mit einer Zahl
beginnen
as = 1 # Bezeichner dürfen nicht mit Schlüsselwörtern übereinstimmen
'''
```

Das ist die Ausgabe des Beispiels:

```
<class 'int'>,0
<class 'float'>,0.0
<class 'float'>,0.0
<class 'str'>,
<class 'complex'>,0j
<class 'bool'>,True
```

Konstanten

Konstanten sind Variablen, die nicht verändert werden dürfen. Sobald während der Programmausführung eine Wertzuweisung erfolgt ist, ist diese endgültig. Sie können also keine zweite Wertzuweisung an eine Konstante vornehmen, nachdem diese initialisiert wurde.



Nicht alle Programmiersprachen besitzen Konstanten. So kennt JavaScript beispielsweise keine Konstanten, und auch in Python kann man diese nicht direkt erstellen.

Konstanten vereinfachen die Lesbarkeit und vermeiden Fehler: Der Wert einer Konstanten wird nur **einmal** eingegeben. Im weiteren Verlauf des Programms arbeiten Sie nur noch mit dem Namen der Konstanten. Sofern der Wert der Konstanten korrigiert werden muss, ist diese Änderung nur an einer einzigen Stelle im Quelltext vorzunehmen.

Wie für eine lokale Variable wird auch für jede Konstante Speicherplatz im Arbeitsspeicher Ihres Computers reserviert. Im Programm greifen Sie auf diesen Bereich über den Namen der Konstanten zu. Jede Konstante, die Sie in Ihrem Programm verwenden, muss vorher definiert werden.

Explizite und implizite Typumwandlung

Unter **Casting** beziehungsweise **Typkonvertierung** versteht man die Umwandlung von einem Datentyp in einen anderen Datentyp. Es gibt verschiedene Situationen, in denen so etwas notwendig ist:

- ✓ Sie wollen zwei unterschiedliche Datentypen verbinden. Beispielsweise sollen eine ganze Zahl und eine Gleitkommazahl addiert oder ein String und eine Zahl verbunden werden.
- ✓ Das Ergebnis einer Operation kann nicht mehr in dem Typ dargestellt werden, den die beteiligten Operanden haben. Beispielsweise ergibt die mathematische Division der ganzen Zahl 5 durch die ganze Zahl 2 keine ganze Zahl, sondern eine Gleitkommazahl.
- ✓ An einer bestimmten Stelle im Programm wird ein bestimmter Datentyp erwartet, aber ein anderer Datentyp geliefert. Beispielsweise erwartet eine Funktion als Parameter eine Gleitkommazahl, bekommt aber eine ganze Zahl übergeben.

Implizite Typumwandlungen

Manche Sprachen wandeln Datentypen im Hintergrund automatisch um, wenn das notwendig ist (implizit). Beispielsweise lassen sich in Python alle primitiven Typen implizit umwandeln.

Beispiel: *ImCast.py*

```
# Variablendeklarationen
i = 0 # Datentyp int
j = 0.0 # Datentyp float
k = 0 + 0j # Datentyp complex
l = 'a' # Datentyp str
m = False # Datentyp bool

# Testausgaben der Datentypen und Werte
print("i", type(i), i, sep=" ")
print("j", type(j), j, sep=" ")
print("k", type(k), k, sep=" ")
print("l", type(l), l, sep=" ")
print("m", type(m), m, sep=" ")

# Variablenumwandlungen
i = j # float zu int-Variable
print("i", type(i), i, sep=" ")

k = j # float zu complex-Variable
print("k", type(k), k, sep=" ")

l = m # bool zu str-Variable
print("l", type(l), l, sep=" ")
```

Das ist die Ausgabe des Beispiels:

```
i: <class 'int'>: 0
j: <class 'float'>: 0.0
k: <class 'complex'>: 0j
l: <class 'str'>: a
m: <class 'bool'>: False
i: <class 'float'>: 0.0
k: <class 'float'>: 0.0
l: <class 'bool'>: False
```

Diese Ad-hoc-Konvertierungen ohne Zutun des Programmiers sind sehr bequem, aber es gibt Situationen, in denen auch eine explizite Konvertierung notwendig ist, um absichtlich den Datentyp eines Wertes zu verändern. Auch das unterstützen moderne Programmiersprachen in der Regel. Entweder mit Operatoren oder Funktionen bzw. Methoden.

Explizite Typumwandlung in Python

Für eine explizite Typumwandlung stellt Python Built-in-Functions zur Verfügung, deren Namen sich an den verfügbaren Datentypen orientieren.

- ✓ Mit `int()` wandelt man einen Parameter in einen `int`-Datentyp.
- ✓ Mit `float()` wandelt man einen Parameter in einen `float`-Datentyp.
- ✓ Mit `str()` wird ein Parameter in einen String überführt.

Darüber hinaus gibt es noch Konvertierungsfunktionen für Listen und Tupel (sequenzielle Datentypen, die in einem späteren Kapitel behandelt werden).

Beispiel: *ExCast.py*

```
# Deklaration der Variablen
var_1 = 4
var_2 = 3.14
var_3 = "123"

# Typbestimmung
print(type(var_1))
print(type(var_2))
print(type(var_3))

# Typumwandlung
print(type(float(var_1)))
print(type(float(var_3)))
print(type(int(var_2)))
print(type(int(var_3)))
print(type(str(var_1)))
print(type(str(var_2)))
```

Das ist die Ausgabe des Beispiels:

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'float'>
<class 'float'>
<class 'int'>
<class 'int'>
<class 'str'>
<class 'str'>
```

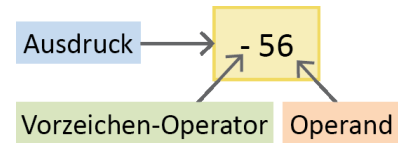


Wenn man unter die Oberfläche schaut, muss man deutlich sagen, dass es in Python in Wirklichkeit gar keine primitiven Datentypen gibt. In Python ist intern alles ein Objekt. Selbst ganze Zahlen oder boolesche Variablen sind in Python Objekte. Damit wird in Python intern mit Referenztypen gearbeitet, auch wenn die Art des Zugriffs wie bei primitiven Datentypen erfolgt. Die Ausgaben der letzten Beispiele haben das auch schon verdeutlicht. Es ist weder Zufall noch bedeutungslos, dass in der Ausgabe des Beispiels immer „class“ in den spitzen Klammern bei dem Datentyp auftaucht.

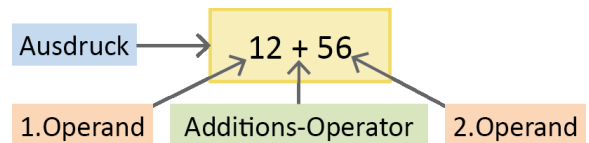
6.6 Operatoren

Operatoren sind Zeichen, die zusammen mit Operanden für die Berechnung eines Ausdrucks (engl. expression) bzw. die Ermittlung eines Wertes verwendet werden. Die meisten Sprachen besitzen unäre und binäre Operatoren. Ausdrücke lernen Sie im nachfolgenden Abschnitt kennen.

Unäre (einstellige) Operatoren werden auf nur **einen** Operanden angewendet. Der Vorzeichen-Operator '-' (Minus) ist ein unärer Operator. Er ändert das Vorzeichen des ihm folgenden Operanden. Beim Ausdruck -1 wird der Vorzeichen-Operator auf die Konstante 1 angewendet und diese so in eine negative Zahl umgewandelt.



Binäre (zweistellige) Operatoren werden auf **zwei** Operanden angewendet. Bei der Addition werden beispielsweise zwei Operanden über den Additions-Operator verknüpft.



Seltener gibt es auch **ternäre** (dreistellige) Operatoren. In den Sprachen C/C++, JavaScript oder Java existiert beispielsweise nur der Fragezeichen-Operator mit drei Operanden. Liefert in der Anweisung `Ausdruck ? b : c` der Ausdruck den Wert *wahr*, wird der Wert von *b* zurückgegeben, anderenfalls der Wert von *c*.

Die nachfolgenden Beispiele für Operatoren verwenden für die konkreten Beispiele explizit Python als Syntax. Bei der Nennung der Operatoren finden Sie allerdings gelegentlich verschiedene andere Schreibweisen, die in anderen Sprachen zum Einsatz kommen.



Vorzeichen-Operatoren

Mit Vorzeichen-Operatoren lässt sich das Vorzeichen einer Zahl bzw. einer Variablen festlegen.

Operator	Name	Bedeutung	Beispiel	Erläuterung
+	Positives Vorzeichen	+a ist gleichbedeutend mit a	a = +4;	a erhält den Wert 4
-	Negatives Vorzeichen	-a kehrt das Vorzeichen der Variable a um	a = -4;	a erhält den Wert -4

Arithmetische Operatoren

Diese Operatoren werden für mathematische Berechnungen in Ausdrücken eingesetzt. Als Operanden sind numerische Variablen oder Konstanten anzugeben. Arithmetische Operatoren liefern ein numerisches Ergebnis zurück.

Operator	Name	Bedeutung	Beispiel	Erläuterung
+	Addition	a + b ergibt die Summe von a und b	a = 4 b = 2 c = a + b	c erhält den Wert 6
-	Subtraktion	a - b ergibt die Differenz von a und b	a = 4 b = 2 c = a - b	c erhält den Wert 2
*	Multiplikation	a * b ist das Produkt aus a und b	a = 4 b = 2 c = a * b	c erhält den Wert 8

Operator	Name	Bedeutung	Beispiel	Erläuterung
/	Division	a / b ist der Quotient von a und b	$a = 4$ $b = 2$ $c = a / b$	c erhält den Wert 2
% (oder MOD)	Modulo	$a \% b$ ist der Rest der ganzzahligen Division von a und b	$a = 5$ $b = 3$ $c = a \% b$	c erhält den Wert 2
**	Potenz	Der erste Operator wird mit dem zweiten Operator zur Potenz genommen.	$a = 3 ** 2$	a erhält den Wert 9
//	Ganzzahldivision	Bei dieser Form der Division wird das Ergebnis eine ganze Zahl sein. Ein eventueller Nachkommateil wird abgeschnitten.	$a = 7 // 2$	a erhält den Wert 3

In einigen Programmiersprachen (z. B. in C, JavaScript und Java) gibt es eine Kurzschreibweise, mit der eine Variable hoch- oder heruntergezählt werden kann.

$x = x + 1$ (Inkrement) oder $x = x - 1$ (Dekrement)

Dafür kennen manche Sprachen keine expliziten Operatoren für die Potenz oder die Ganzzahldivision.

Vergleichsoperatoren

Diese Operatoren, auch relationale Operatoren genannt, vergleichen Ausdrücke miteinander und liefern ein logisches Ergebnis, entweder `True` oder `False`.

Operator	Name	Bedeutung	Beispiel	Erläuterung
== (oder =)	Gleichheit	$a == b$ ergibt <code>True</code> , wenn a und b gleich sind.	$a = 4$ $b = 4$ $a == b$	$a == b$ liefert <code>True</code>
!= (oder <>)	Ungleichheit	$a != b$ ergibt <code>True</code> , wenn a und b ungleich sind.	$a = 4$ $b = 4$ $a != b$	$a != b$ liefert <code>False</code>
<	Kleiner	$a < b$ ergibt <code>True</code> , wenn a kleiner b ist.	$a = 3$ $b = 4$ $a < b$	$a < b$ liefert <code>True</code>
>	Größer	$a > b$ ergibt <code>True</code> , wenn a größer b ist.	$a = 3$ $b = 4$ $a > b$	$a > b$ liefert <code>False</code>
<=	Kleiner gleich	$a <= b$ ergibt <code>True</code> , wenn a kleiner oder gleich b ist.	$a = 4$ $b = 4$ $a <= b$	$a <= b$ liefert <code>True</code>
>=	Größer gleich	$a >= b$ ergibt <code>True</code> , wenn a größer oder gleich b ist.	$a = 4$ $b = 4$ $a >= b$;	$a >= b$ liefert <code>True</code>

Einige Programmiersprachen kennen noch die **Identität** als Vergleichsoperator. In JavaScript werden dafür zum Beispiel drei Gleichheitszeichen (===) genutzt. Dabei werden sowohl eine Wert- als auch eine Datentypgleichheit überprüft. Die Umdrehung in JavaScript ist !==.



Logische Operatoren

Vergleichsoperatoren vergleichen Ausdrücke miteinander, deren Typen Zahlen oder logische Operatoren sein können. Mit logischen Operatoren werden Wahrheitswerte miteinander verknüpft. Das Ergebnis der Vergleichsoperation beider Operationen ist immer ein logischer Wert.

Die boolesche Logik basiert auf dem binären Zahlensystem und kann nur zwei Zustände annehmen: logisch 1 und logisch 0 – wahr und falsch. Abhängig von der Sprache werden logische Operatoren als Zeichen oder als Wort (Abkürzung) dargestellt. Die in der folgenden Tabelle zusammengestellten Operatoren sind nicht in allen Sprachen implementiert.

Operator	Name	Bedeutung	Beispiele
not (oder !)	Logisches NICHT Math. Zeichen: \neg	Ergibt die Umkehrung des Wahrheitswertes	not True \rightarrow False not False \rightarrow True
and (oder &&)	Logisches UND Math. Zeichen: \wedge	Ergibt True, wenn beide Operanden True sind, ansonsten False	True and True \rightarrow True True and False \rightarrow False False and True \rightarrow False False and False \rightarrow False
or (oder)	Logisches ODER Math. Zeichen: \vee	Ergibt True, wenn mindestens einer der beiden Operanden True ist	True or True \rightarrow True True or False \rightarrow True False or True \rightarrow True False or False \rightarrow False

- ✓ Manche Programmiersprachen kennen noch Operatoren für ein exklusives ODER (xor). Dabei gibt ein Vergleich nur dann True, wenn nur einer der beiden Operanden True ist.
- ✓ Ebenso gibt es Sprachen mit einem Operator für eine Implikation. Dabei gilt, wenn der erste Operand True ist, muss auch der zweite Operand True sein; wenn der erste Operand False ist, ist der Wert des zweiten Operanden nicht relevant.

Bitweise Operatoren

Verschiedene Programmiersprachen besitzen Bit-Operatoren, die eine hardwarenahe Programmierung unterstützen. Eingesetzt werden Bit-Operationen beispielsweise auch für die Optimierung von Multiplikation und Division. Zum Beispiel kann statt einer Multiplikation mit 2 eine einfache Linksverschiebung der Bits verwendet werden, die schneller abläuft als die arithmetische Operation.

Beispiel

Im Folgenden wird die Zahl $(10)_{10}$ mit dem Wert 2 multipliziert. Die Bitfolge $(01010)_2$ wird dabei um ein Bit nach links verschoben, und es wird eine 0 angehängt $\Rightarrow (10100)_2$ (entspricht $(20)_{10}$).

Operator	Name	Bedeutung	Beispiel	Erläuterung
~	Einerkomplement bitweise Negation	Invertiert alle Bits des Operanden	B1 = 153 # 10011001 B1 = ~B1 # 01100110	B1 erhält den Wert 102
	ODER	a b verknüpft die jeweiligen Bits von a und b nach der ODER-Logik	B1 = 145 # 10010001 B2 = 133 # 10000101 B3 = B1 B2 # 10010101	B3 erhält den Wert 149

Operator	Name	Bedeutung	Beispiel	Erläuterung
&	Bitweises UND	a & b verknüpft die jeweiligen Bits von a und b nach der UND-Logik	B1 = 145 # 10010001 B2 = 133 # 10000101 B3 = B1 & B2 # 10000001	B3 erhält den Wert 129
^	Bitweises Exklusiv-ODER	a ^ b verknüpft die jeweiligen Bits von a und b nach der XOR-Logik	B1 = 145 # 10010001 B2 = 129 # 10000001 B3 = B1 ^ B2 # 00010000	B3 erhält den Wert 16
>>	Rechtsschieben	a >> x verschiebt die Bits der Variablen a um x Stellen nach rechts. Dies entspricht der Division durch 2 ^x bei vorzeichenlosen Zahlen (Überlauf beachten!).	B1 = 144 # 10010000 B1 = B1 >> 2 # 00100100	B1 erhält den Wert 36
<<	Linksschieben	a << x verschiebt die Bits der Variablen a um x Stellen nach links. Dies entspricht der Multiplikation mit 2 ^x bei vorzeichenlosen Zahlen (Überlauf beachten!).	B1 = 84 # 01010100 B1 = B1 << 1 # 10101000	B1 erhält den Wert 168

In einige Sprachen gibt es noch das Rechtsschieben ohne Vorzeichen.

Zuweisungsoperatoren

Um einer Variablen einen Wert zuzuweisen, wird der Zuweisungsoperator verwendet. Der Wert, der einer Variablen zugewiesen wird, ergibt sich meist aus einem Ausdruck. Sie können einer Variablen in der Regel nur Werte des Datentyps zuweisen, den sie selbst besitzt.

Operator	Bedeutung	Beispiele
= (oder :=)	Der Variablen auf der linken Seite dieser Anweisung wird der Wert des Ausdrucks der rechten Seite zugewiesen.	zahl = 123 * 5 richtig = zahl > 100 zeichen = 'b'

Bei einigen Programmiersprachen werden Zuweisungsoperatoren mit anderen Operatoren gekoppelt, um den Schreibaufwand zu verringern.

So kann z. B. in Python, C, JavaScript und Java statt `x = x + 72` die Kurzschreibweise `x += 72` verwendet werden.

Operatoren	Bedeutung	Initialisierung	Beispiel
+= -= *= /= &= %= = ^= <<= //= **= >>=	a *= b weist der Variablen a den Wert von a * b zu und liefert a * b als Rückgabewert. Dies ist gleichzusetzen mit der Funktion a = a * b. Diese Funktionsweise ist bei all diesen Operatoren gleich.	a = 4 b = 5	a *= b a erhält den Wert 20

Der String-Verkettungsoperator

In vielen Sprachen kann man Strings verknüpfen. Der Plusoperator wird in Python auch zur String-Verkettung verwendet. Das verbindet zwei Strings zu einem einzigen String. So einen Operator gibt es in vielen Sprachen, und meist wird auch dort das Pluszeichen verwendet (eine der wenigen Ausnahmen ist PHP). Aber es gibt in Python eine Besonderheit zu beachten.

Sobald einer der Operanden des Plusoperators kein String ist, wird in Python **keine** automatische String-Verkettung genommen. In vielen anderen Sprachen ist das der Fall. In Python müssen Sie gegebenenfalls einen unpassenden Operanden erst mit der Built-in-Function `str()` in einen String wandeln. Sonst kommt es zum Fehler.



Beispiel: *StringVerkettung.py*

```
var_1 = "Hallo" # String
var_2 = 1 # int
var_3 = "1" # String mit einer Zahl als Inhalt

print(var_1 + var_3) # String-Verkettung ohne Umwandlung
print(var_1 + str(var_2)) # Umwandlung des int var2
print(str(var_2) + var_3) # Umwandlung des int var2
```

Das ist die Ausgabe des Beispiels:

```
Hallo1
Hallo1
11
```

Vor der Verknüpfung muss jeder Datentyp wie erwähnt ein String sein. Aus dem Grund wird ab der zweiten Ausgabe mit der Funktion `str()` gearbeitet. Deshalb gibt auch die dritte Ausgabe das Ergebnis 11 und nicht 2. Es ist ausdrücklich eine String-Verkettung und keine arithmetische Addition. Nur bei der ersten Ausgabe kann man den Plusoperator ohne vorherige String-Umwandlung verwenden. In `var_3` steht zwar eine Zahl als Inhalt, aber der Datentyp ist ein String.

6.7 Ausdrücke und Operatorenrangfolgen

In den meisten Programmiersprachen gehören Ausdrücke zu den kleinsten ausführbaren Einheiten eines Programms. Sie werden verwendet, um Variablen einen Wert zuzuweisen, numerische Berechnungen durchzuführen oder logische Bedingungen zu formulieren.

Ein Ausdruck ist eine gültige Kombination von Operanden und Operatoren, die ein Ergebnis liefert. Bei der Anwendung der Operatoren gilt wie in der Mathematik die Regel „Punktrechnung geht vor Strichrechnung“, geklammerte Ausdrücke werden zuerst ausgewertet.

Beispiele

<code>a</code>	Auch eine Variable ist ein Ausdruck.
<code>a * (b+c)</code>	Zuerst wird die Klammer berechnet und danach das Ergebnis mit dem Wert <code>a</code> multipliziert.
<code>((a>100) and (b<1000)) or (c>200)</code>	Zuerst werden die Ausdrücke <code>a>100</code> und <code>b<1000</code> mit dem logischen Operator <code>and</code> (UND) verknüpft. Dieses Ergebnis wird dann mit dem Ergebnis des Ausdrucks <code>c>200</code> über den <code>or</code> -Operator (ODER) verknüpft.

Rangfolge der Operatoren

Oft ist es notwendig, mehr als zwei Operanden miteinander zu vergleichen, z. B.:
`(amount1 > 500) or (amount2 < 801) and !(amount3 == 400)`

Die Abarbeitungsreihenfolge für diese Anweisung kann in verschiedenen Programmiersprachen unterschiedliche Ergebnisse liefern. Das liegt daran, dass der Ausdruck nach bestimmten Vorrangregeln für die Operatoren abgearbeitet wird. Einige Sprachen arbeiten die Ausdrücke prinzipiell von links nach rechts ab.



Wollen Sie sichergehen, dass die Ausdrücke in der richtigen Reihenfolge abgearbeitet werden, setzen Sie Klammern. Wie in der Mathematik werden die Bedingungen in Klammern **immer** zuerst ausgewertet. Auch die Lesbarkeit Ihres Programms wird durch das Setzen von Klammern erhöht.

Details zum Operatorvorrang und Ausdrucksbewertung

Operatorvorrang bedeutet die Beachtung der Priorität von Operatoren. Denn diese sind entsprechend geordnet. Die Operatoren in Python haben eine festgelegte Rangordnung, die immer dann angewandt wird, wenn in einem Ausdruck mehrere Operatoren verwendet werden und keine Klammern gesetzt werden (Klammern sind in der Rangfolge mit aufgenommen und haben die höchste Priorität). Als Beispiel kennen Sie bereits die Punkt-vor-Strich-Regel. Die Prioritäten sind von der höchsten Wertigkeit bis zur niedrigsten abwärts angegeben.

Operator	Beschreibung
<code>**</code>	Exponentialoperator
<code>~ + -</code>	Komplement
<code>* / % //</code>	Punktrechnung
<code>+ -</code>	Strichrechnung
<code>>> <<</code>	Rechts und links bitweise verschieben
<code>&</code>	Bitweise 'UND'
<code>^ </code>	Bitweise exklusive 'ODER' und regelmäßige 'ODER'
<code><= < > >=</code>	Vergleichsoperatoren
<code>== !=</code>	Gleichstellungsbetreiber
<code>= %= /= //=- += *= **=</code>	Zuweisungsoperatoren
<code>is is not</code>	Identitätsoperatoren
<code>in not in</code>	Mitgliedschaftsoperatoren
<code>not or and</code>	Logische Operatoren

Bewertung von Ausdrücken

Wenn Sie einen Ausdruck mit unterschiedlichen Operatoren haben, muss Python entscheiden, wie Ausdrücke bewertet werden. Dazu wird zuerst der gesamte Ausdruck analysiert. Hier ist das Beispiel von Punkt-vor-Strich-Rechnung aus der Mathematik wieder sinnvoll. Python hält sich strikt an Regeln der Operatorvorrangigkeit. Je höher ein Operator priorisiert ist, desto eher wird er bewertet.

Die sogenannte **Operatorassoziativität** ist die einfachste der Bewertungsregeln, aber sie kann von Bedeutung sein. Es geht darum, dass alle arithmetischen Operatoren Ausdrücke in der Vorgabe von links nach rechts bewerten (assoziiieren). Das heißt, wenn derselbe Operator oder Operatoren gleicher Priorität in einem Ausdruck mehr als einmal auftauchen – wie beispielsweise der `+`-Operator bei dem Ausdruck `1 + 2 + 3 + 4 + 5` –, dann wird der am weitesten links erscheinende zuerst bewertet, gefolgt von dem rechts daneben und so weiter.

Unterziehen wir folgende arithmetische Zuweisung einer näheren Betrachtung:

```
x= 1 + 2 + 3 + 4 + 5
```

In diesem Beispiel wird der Wert des Ausdrucks auf der rechten Seite des Gleichheitszeichens zusammengerechnet und der Variablen `x` auf der linken Seite zugeordnet. Für das Zusammenrechnen des Werts auf der rechten Seite bedeutet die Tatsache, dass der Operator `+` von links nach rechts assoziiert, dass der Wert von `1 + 2` zuerst berechnet wird. Erst im nächsten Schritt wird zu diesem Ergebnis dann der Wert `3` addiert und so fort, bis zuletzt die `5` addiert wird. Anschließend wird das Resultat dann der Variablen `x` zugewiesen. Immer, wenn Operatoren derselben Priorität mehrfach benutzt werden, können Sie die Assoziativitätsregel anwenden.

6.8 Übungen

Übung 1: Arbeiten mit grundlegenden Sprachelementen

Übungsdatei: --

Ergebnisdatei: *uebung06.pdf*

1. Was verstehen Sie unter einer Variablen, und welche Gültigkeitsbereiche gibt es?
2. Welche Zahlenart kann eine als `int` deklarierte Variable in Python darstellen?
3. Zwei Variablen, `var1` und `var2`, vom Datentyp `char` (character) sind gegeben. Die Werte der beiden Variablen sollen vertauscht werden. Beschreiben Sie den Algorithmus im Pseudocode.
4. Stellen Sie die arithmetischen, logischen und Vergleichsoperatoren in einem Syntaxdiagramm und in der erweiterten BNF dar. Fassen Sie dabei unäre und binäre Operatoren in je einer Darstellung zusammen. Verwenden Sie jeweils eines der möglichen Zeichen pro Operator. Erzeugen Sie das Syntaxdiagramm und die erweiterte BNF für einen Ausdruck. Verwenden Sie dabei unäre und binäre Operatoren.
5. Werten Sie folgende Bedingungen aus:
 - ✓ `(value1 > 500) or (value2 < 800) and not (value3 == 400)`
 - ✓ `(value1 > 500) or ((value2 < 800) and not (value3 == 400))`

Die Variablen sollen folgende Werte enthalten:

```
value1 = 501           value2 = 799           value3 = 400
```

In den Bedingungen wird angenommen, dass die Abarbeitung von links nach rechts erfolgt.

Übung 2: Zwei Variablen vertauschen

Übungsdatei: --

Ergebnisdatei: *ChangeValues.py*

1. Setzen Sie den Pseudocode zum Vertauschen der zwei Variablen, `var1` und `var2`, vom Datentyp `str` (String) in Python um.
Mit dem Befehl `print(var1)` können Sie den Wert einer Variablen ausgeben. Um verschiedene Werte zusammen mit Text auszugeben, verwenden Sie den Befehl wie folgt: `print("Wert der ersten Variablen", var1, "hier kann noch mehr Ausgabertext stehen.", var2)`

7 Kontrollstrukturen

In diesem Kapitel erfahren Sie

- ✓ wie Sie das Ausführen von Anweisungen von Bedingungen abhängig machen
- ✓ welche Möglichkeiten der Fallauswahl es gibt
- ✓ wie Sie Anweisungen wiederholt ausführen können

Voraussetzungen

- ✓ Arbeiten mit Variablen, Operatoren
- ✓ Verwendung der Datentypen

7.1 Anweisungen und Folgen

Was sind Anweisungen?

Ein Programm setzt sich aus einer Menge von Anweisungen zusammen, um eine bestimmte Aufgabe zu lösen. Anweisungen arbeiten dabei mit den im Programm festgelegten Variablen und führen damit Operationen aus. Die einzelnen Programmiersprachen stellen unterschiedliche Anweisungen zur Verfügung; dabei finden sich jedoch folgende Grundstrukturen wieder:

- ✓ Einfache Anweisungen, z. B. Zuweisung
- ✓ Verzweigungen
- ✓ Schleifen



In Python stehen diverse Kontrollstrukturen, die es in anderen modernen Programmiersprachen gibt, nicht oder nur eingeschränkt zur Verfügung. Das mag Umsteigern als Schwäche oder Einschränkung erscheinen, gilt aber im Gegenteil als einer der Hauptvorteile von Python. Der Verzicht auf redundante Syntaxstrukturen führt zu einem klareren und effizienteren Quellcode und zwingt Programmierer zu einem sauberen Stil. Allerdings sollte man als Programmierer auch diejenigen typischen Kontrollstrukturen kennen, die in Sprachen wie Java, JavaScript, PHP, C, C# etc. zum unabdingbaren Standard gehören. Deshalb werden sie in dem Kapitel zumindest vorgestellt und beim Fehlen in Python mit Java-Quellcode skizziert.

Einfache Anweisungen

Eine Anweisung (engl. statement) ist die kleinste ausführbare Einheit eines Programms. In Java, JavaScript und vielen anderen Programmiersprachen wird eine Anweisung mit einem Strichpunkt (Semikolon) abgeschlossen. In einigen Sprachen wird pro Zeile eine Anweisung geschrieben. Ist eine Anweisung so lang, dass sie nicht in eine Zeile passt, wird sie in der nächsten Zeile fortgesetzt. Einige Programmiersprachen verlangen, dass diese Fortsetzungszeile dann mit einem speziellen Fortsetzungszeichen beginnt.



Die folgenden Beispiele zeigen die Programmierung unter Verwendung von Python, und das Anweisungsende wird nicht mit einem Semikolon abgeschlossen.

Die leere Anweisung

Die einfachste Form der Anweisung ist die **leere Anweisung**.

Syntax	Bedeutung
<code>pass</code> (oder <code>;</code>)	Eine leere Anweisung besteht in den meisten Sprachen (Java, JavaScript, PHP ...) nur aus dem Semikolon und hat keinerlei Auswirkung auf das laufende Programm. Eine leere Anweisung kann dort verwendet werden, wo syntaktisch eine Anweisung erforderlich ist, aber von der Programmlogik her nichts zu tun ist. Python verwendet dafür aber ein eigenes Schlüsselwort <code>pass</code> und kann das Semikolon nicht als leere Anweisung verwenden.

Anweisungsblöcke

Bei einer Reihe von syntaktischen Sprachkonstrukten darf nicht mehr als eine Anweisung angegeben werden. Die Logik des Programms kann es jedoch erfordern, dass eine Folge von zwei oder mehr Anweisungen ausgeführt werden muss. In diesen Fällen muss ein **Anweisungsblock** (Verbundanweisung) verwendet werden.

In den nachfolgenden Beschreibungen der Syntax wird jeweils nur der Begriff `statement` verwendet. Dies kann dann eine einzelne Anweisung oder ein Anweisungsblock sein.

Syntax	Bedeutung
Einrückung oder <pre>{ statement1 statement2 }</pre> oder <pre>Begin statement1 statement2 End</pre>	<p>Ein Anweisungsblock ist eine Zusammenfassung von Anweisungen, die nacheinander ausgeführt werden. Alle im Block zusammengefassten Anweisungen gelten in ihrer Gesamtheit als eine einzelne Anweisung. Ein Block wird eingerückt oder in Schlüsselwörter oder in definierte Zeichen eingeschlossen.</p> <p>In den meisten populären Sprachen wird ein Block in geschweifte Klammern <code>{ }</code> eingeschlossen.</p> <p>Einige Sprachen verwenden die Worte <code>begin</code> und <code>end</code>, die manchmal klein geschrieben werden, manchmal auch groß oder die Schreibweise spielt keine Rolle.</p> <p>In Python wird die Strukturierung in Blockanweisungen durch Einrückung von vier Zeichen (Vorgabe, die aber geändert werden kann) vorgenommen. Das unterscheidet das Strukturierungsprinzip von Python deutlich von anderen Programmiersprachen. Dafür spart man sich die zusätzlichen Schlüsselwörter beziehungsweise Trennzeichen.</p> <p>Trotz der unterschiedlichen Konzepte zum Schreiben eines Blocks – die Funktion eines Blocks ist in allen Sprachen identisch.</p>

Wenn Sie eine Berechnung durchführen oder eine Eingabe eines Anwenders speichern möchten, erfolgt dies über die **Zuweisungsanweisung**.

Syntax	Bedeutung
<code>i = i + 1</code>	Einer Variablen wird ein berechneter Wert des Ausdrucks <code>i + 1</code> zugewiesen.

Die Ausgabe von Werten beispielsweise auf dem Bildschirm können Sie über **Ausgabeeanweisungen** realisieren.

Syntax	Bedeutung
<code>print("text")</code>	Die Ausgabe eines Wertes oder Textes erfolgt über eine von der Sprache vorgegebene Ausgaberoutine.

Zur Eingabe werden entsprechende **Eingabeanweisungen** verwendet.

Syntax	Bedeutung
<code>name = input ("Eingabe des Namens")</code>	Die Eingabe eines Wertes oder Textes erfolgt über eine von der Sprache vorgegebene Eingaberoutine.

Diese Arten von Anweisungen werden auch **einfache Anweisungen** genannt.

Folgen

Eine **Folge**, auch **Sequenz** genannt, ist eine Liste von Anweisungen, die nacheinander (sequenziell) abgearbeitet werden.

Beispiel: Programm zur Berechnung des Bruttopreises – *Amount.py*

Der Nettopreis wird im Beispiel vorgegeben. Daraus wird der Bruttopreis berechnet und ausgegeben.

```
netAmount = 7.84 # Nettopreis
VAT = 0.19 # Mehrwertsteuer
totalAmount = netAmount * (1 + VAT)
print("Berechneter Bruttopreis: " , totalAmount)
```

Ein Python-Beispielprogramm für eine Sequenz

7.2 Bedingungen und Kontrollstrukturen

Was sind Bedingungen?

Durch eine Bedingung (engl. condition) wird der Ablauf eines Programms beeinflusst. Es wird ein logischer Ausdruck ausgewertet. Für die Formulierung solcher Bedingungen stehen die Vergleichsoperatoren und die logischen Operatoren zur Verfügung. Eine Bedingung kann entweder mit „Ja“ beantwortet werden (`True` bzw. wahr) oder mit „Nein“ (`False` bzw. falsch). Vom Ergebnis der Bedingung ist abhängig, welche Anweisungen des Programms danach abgearbeitet werden und welche nicht.

Beispiele

Es wird mithilfe des Modulo-Operators geprüft, ob eine Zahl gerade ist:	<code>(Zahl % 2) == 0</code>
Es wird geprüft, ob eine Zahl zwischen 100 und 500 liegt (ohne die Werte 100 und 500):	<code>(Zahl > 100) and (Zahl < 500)</code>
Es wird geprüft, ob ein Zeichen eine Ziffer ist:	<code>(Zeichen >= '0') and (Zeichen <='9')</code>
Es wird geprüft, ob der Mitarbeiter „Meier“ seine 30 Urlaubstage bereits angebrochen hat:	<code>(Mitarbeiter == "Meier") and (Urlaubstage < 30)</code>

Kontrollstrukturen im Überblick

Sollen Programmteile mehrmals oder gar nicht ausgeführt werden, werden Sprachelemente benötigt, mit denen der Programmablauf gesteuert werden kann, sogenannte **Kontrollstrukturen**. Die Entscheidung, nach welchen Kriterien der Ablauf gesteuert wird, wird in **Bedingungen** formuliert.

Es werden zwei Gruppen von Kontrollstrukturen unterschieden:

Verzweigungen	Es werden alternative Programmteile angeboten, in die – abhängig von einer Bedingung – beim Programmablauf verzweigt wird.
Schleifen (Wiederholung oder Iterationen)	Ein Programmteil kann – abhängig von einer Bedingung – mehrmals durchlaufen werden (Schleife = engl. loop) oder gar nicht.

7.3 Grundlagen zu Verzweigungen

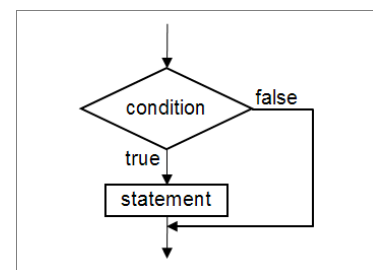
Eine Verzweigung kann eine der folgenden Ausprägungen besitzen:

Bedingte Anweisung („Einseitige“ Verzweigung)	Je nachdem, ob eine Bedingung erfüllt ist oder nicht, wird ein Programmteil ausgeführt oder übersprungen.
Verzweigung	In Abhängigkeit vom Ergebnis einer Bedingung wird ein Programmteil oder ein anderer Programmteil ausgeführt.
Geschachtelte Verzweigung	Innerhalb der alternativen Programmteile einer bedingten Anweisung oder einer Verzweigung wird eine weitere Verzweigung integriert. Mit einer solchen geschachtelten Verzweigung lassen sich mehrere Alternativen bereitstellen.
Mehrfache Verzweigung	Es wird ein Ausdruck ausgewertet. Je nachdem, welchem Wert der Ausdruck entspricht, verzweigt das Programm fallweise (case) in einen entsprechenden Programmteil. Eine solche Kontrollstruktur kann somit ebenfalls mehrere Alternativen bereitstellen.

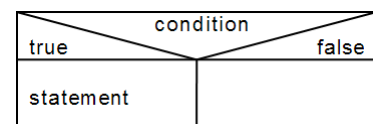
7.4 Bedingte Anweisung

Die bedingte Anweisung ist dadurch gekennzeichnet, dass nach einer Bedingungsabfrage durch einen Ausdruck eine Anweisung oder ein Anweisungsblock (mehrere Anweisungen) ausgeführt wird oder nicht.

Wenn die Bedingung mit *Ja* (wahr) beantwortet wird, **dann** wird die Anweisung (statement) ausgeführt. Wird die Bedingung mit *Nein* beantwortet (falsch), erfolgt keine Aktion des Programms. In beiden Fällen wird nach dem Ende der Alternative mit der nächsten Anweisung fortgesetzt. Die bedingte Anweisung entspricht in den meisten Programmiersprachen einer *if*-Anweisung mit optionalem alternativen Zweig.



PAP



Struktogramm

Einige ältere Programmiersprachen verwenden *if* in Verbindung mit *then* hinter der Bedingung. Erst danach folgt die auszuführende Anweisung oder der Block.



Syntax zur bedingten Anweisung in Python

- ✓ Eine bedingte Anweisung wird mit dem Schlüsselwort `if` eingeleitet.
- ✓ Nach dem Schlüsselwort `if` wird (bei Bedarf auch in Klammern `[]`), wobei das in Python eher untypisch ist) als Bedingung ein logischer Ausdruck angegeben, dessen Auswertung `True` oder `False` sein muss.
- ✓ In Python folgt dann ein Doppelpunkt.
- ✓ Ist die Bedingung erfüllt (`True`), wird die folgende Anweisung ① bzw. der eingerückte Anweisungsblock ② ausgeführt.

```
if condition:
    statement ①
```

```
if condition:
    statement1 ②
    statement2
    ...
```

Beispiel: *Discount.py*

Wenn (`if`) ein Kunde einen Auftrag höher als 1000 EUR erteilt, bekommt er 3 % Rabatt und der neue Preis wird berechnet. Bei Aufträgen bis 1000 EUR wird kein Rabatt gewährt. Die Überprüfung der Bedingung kann nur das Ergebnis *Ja* bzw. *Nein* ergeben, in Python entsprechend `True` oder `False`.

```
price = 1497.56
if price > 1000:
    price = price * (1 - 0.03)
print("Berechneter Discount-Preis: " , price)
```

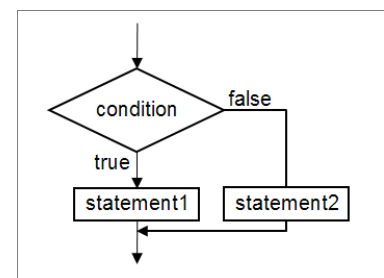
Python-Beispiel

7.5 Verzweigung

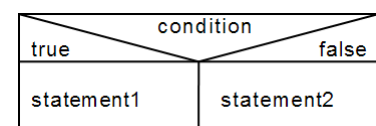
Bei einer sogenannten Verzweigung (auch Alternative genannt) wird einer von zwei möglichen Anweisungsblöcken in Abhängigkeit von einer Bedingung ausgeführt.

Wenn die Bedingung erfüllt ist, **dann** wird Anweisungsblock A (`statement1`) ausgeführt, **sonst** wird Anweisungsblock B (`statement2`) ausgeführt. Die Verzweigung entspricht in den meisten Programmiersprachen der `if`-Anweisung mit `else`-Zweig.

Sie können die Verzweigung verwenden, wenn es genau **zwei** Auswahlmöglichkeiten gibt.



PAP



Struktogramm

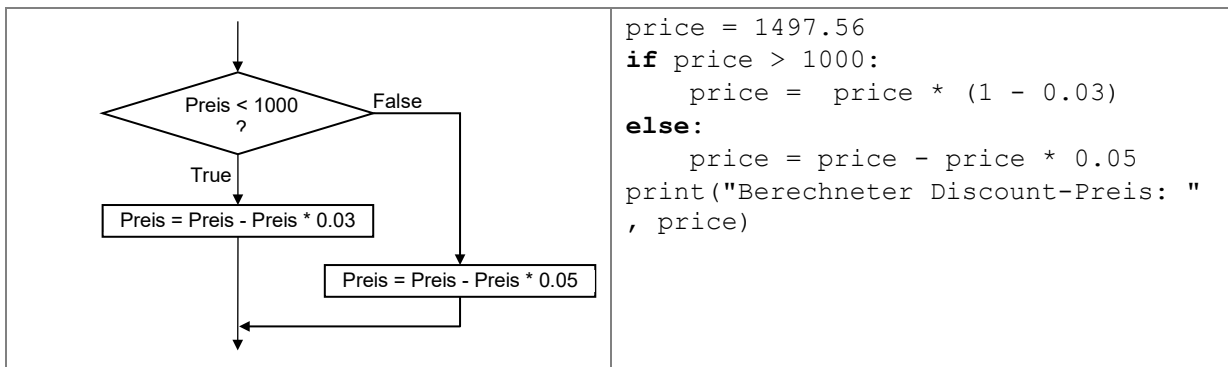
Syntax zur Verzweigung in Python

- ✓ Nach der Bedingung in der `if`-Anweisung und der Anweisung ① folgt das Schlüsselwort `else`.
- ✓ Nach `else` schließt sich die Anweisung ② an, die ausgeführt wird, wenn die Bedingung in der `if`-Anweisung nicht zutrifft (Ausdruck liefert `False`).

```
if condition:
    statement1 ①
else:
    statement2 ②
```

Beispiel: Discount2.py

Bei einem Einkaufswert von weniger als 1000 EUR wird ein Rabatt von 3 % auf den Einkaufspreis gewährt. Liegt der Einkaufspreis bei 1000 EUR oder mehr, erhält der Kunde 5 % Rabatt. Der neue Preis wird berechnet.



PAP

Python

7.6 Geschachtelte Verzweigung

Innerhalb eines `if`-Blocks oder eines `else`-Blocks dürfen wiederum `if`-Anweisungen stehen, d. h., sie werden geschachtelt. Damit kann die Ausführung von Anweisungen von mehreren Bedingungen abhängig gemacht werden. Zur Vereinfachung geschachtelter `if`-Anweisungen bieten Programmiersprachen oft den `else-if`-Block. Python stellt das Schlüsselwort `elif` als Verkürzung bereit.

Ist die erste Bedingung wahr, wird kontrolliert, ob auch die nächste Bedingung wahr ist, usw.

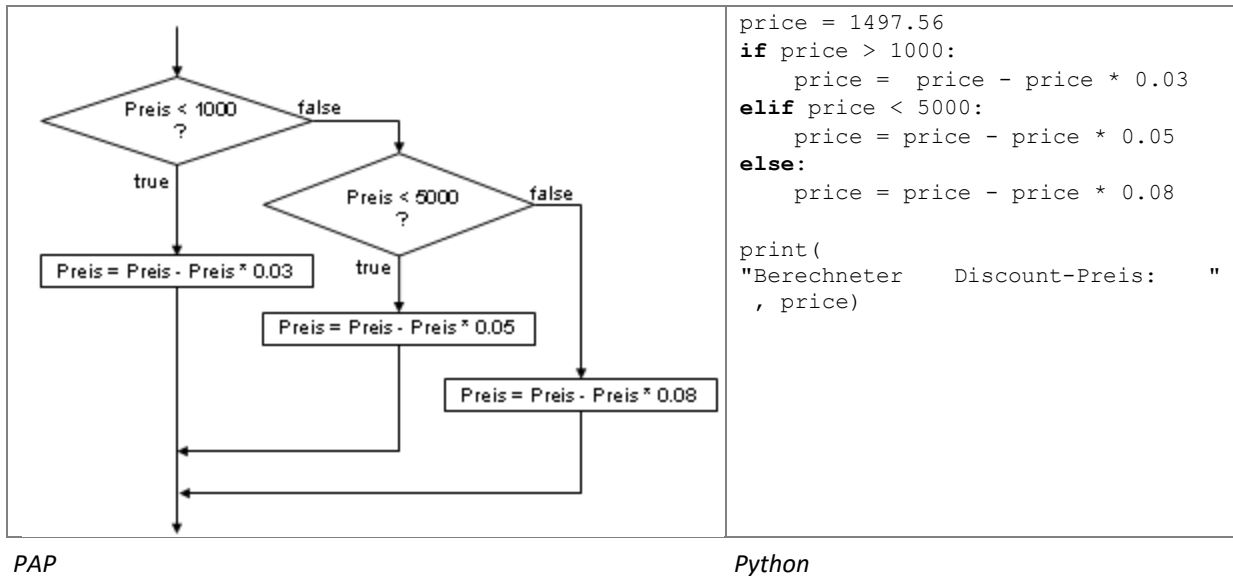
<pre> if condition1: statement1 if condition2: statement2 else: statement3 else: statement4 </pre>	<pre> if condition1: statement1 elif condition2: statement2 else: statement3 </pre>	<pre> if condition1: statement1 else: statement2 if condition2: statement3 else: statement4 </pre>
--	---	--

Möglichkeiten für geschachtelte Verzweigungen in Python

Für die geschachtelten `if`-Anweisungen gelten die gleichen Regeln wie für eine einfache `if`-Anweisung.

Beispiel: Discount3.py

Sie können die mehrfache Verzweigung verwenden, wenn es **mehr als zwei** Auswahlmöglichkeiten gibt. Das Beispiel zur Berechnung des Rabattes wird dazu noch etwas erweitert. Bei einem Einkaufswert von weniger als 1000 EUR wird ein Rabatt von 3 % auf den Einkaufspreis gewährt. Liegt der Einkaufspreis bei 1000 EUR oder mehr, erhält der Kunde 5 % Rabatt. Ab 5000 EUR erhält er 8 % Preisnachlass.

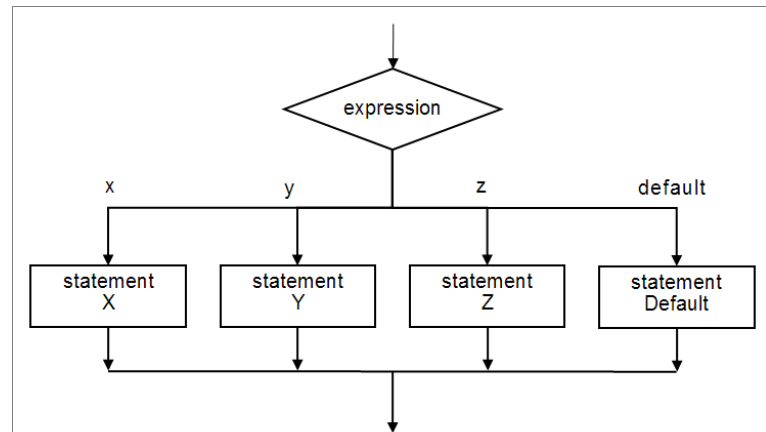


Vermeiden Sie, wenn möglich, geschachtelte Verzweigungen, die mehr als drei Stufen umfassen. Ihr Programm wird sonst zu unübersichtlich und ist schlecht lesbar.

7.7 Mehrfache Verzweigung (Fallauswahl)

Bei einer mehrfachen Verzweigung wird der Wert einer Variablen ausgewertet und in Abhängigkeit von diesem Wert eine Anweisung bzw. ein Anweisungsblock ausgeführt. Die Variable, deren Inhalt geprüft werden soll, wird auch als **Selektor** bezeichnet. In der Programmierung wird auch von einer **Fallauswahl** (oder Selektion) gesprochen.

Ist der Wert der auszuwertenden Variablen gleich dem ersten Wert (in der Abbildung: *x*), wird die zugehörige Anweisung (hier: *statementX*) ausgeführt. Entspricht der Variablenwert dem zweiten Wert (*y*), wird Anweisung (*statementY*) ausgeführt usw. Ist keiner der Werte gleich dem Wert der Variablen, wird der Default-Block *statementDefault* abgearbeitet.



expression			
x	y	z	
statement X	statement Y	statement Z	statement Default

Struktogramm



Python kennt für die mehrfache Verzweigung keine eigenständige Syntax, sondern verwendet `if` in Verbindung mit `elif`. Da aber nahezu alle anderen Programmiersprachen eine eigenständige Syntax für diese Art der Verzweigung kennen, soll sie anhand von Java skizziert werden.

Syntax zur mehrfachen Verzweigung

- ✓ Die mehrfache Fallauswahl wird in vielen Sprachen mit dem Befehl `switch` eingeleitet.
- ✓ Danach folgt in Klammern der Selektor, ein Ausdruck (`expression`), der ordinal (abzählbar) sein muss, z. B. vom Typ `int`. Der Wert des Selektors bestimmt, welcher Auswahlblock ausgeführt wird.
- ✓ Jeder Auswahlblock beginnt in Java mit `case` und einem konstanten Wert, der den gleichen Typ hat wie der Wert des Ausdrucks im Selektor.
- ✓ Stimmt der Wert des Selektors mit einem der aufgeführten Auswahlwerte überein, wird der Anweisungsblock hinter dem Wert ausgeführt. Damit in Java bzw. JavaScript nicht auch die nachfolgenden Auswahlblöcke ausgeführt werden, schließen Sie einen Auswahlblock mit `break` ab.
- ✓ Die Überprüfung des Auswahlwertes wird nur einmal durchgeführt.
- ✓ Stimmt der Wert des Selektors mit keinem Auswahlwert überein, werden die Anweisungen des `default`-Zweigs ausgeführt. Wird dieser Defaultblock weggelassen, führt das Programm die nächste Anweisung nach dem Ende der Fallauswahl durch.

```
switch (expression)
{
    case constantValue1: statement1
                        break;
    case constantValue2: statement2
                        break;
    ...
    [default: statement]
}
```

Ausnahmsweise soll ein Java-Listing diese Art der Fallunterscheidung demonstrieren.

Beispiel: *Grading.java*

Ein Programm soll jeweils für eine Schulnote (`grade`) eine entsprechende Beurteilung als Text ausgeben. Eine gut strukturierte Fallauswahl ist dadurch gekennzeichnet, dass

- ✓ die Fälle nach ihrer Häufigkeit angeordnet sind. Der am häufigsten vorkommende Fall steht an erster Stelle, der seltenste Fall am Ende. Dadurch wird die Fallauswahl schneller abgearbeitet;
- ✓ alle Auswahlfälle, die nicht in ihrer Häufigkeit vergleichbar sind, alphabetisch oder numerisch geordnet sind;
- ✓ der letzte `default`-Zweig (auch `otherwise`-Zweig genannt) für unvorhergesehene Fälle und Fehlermeldungen verwendet wird. Trifft keine der vorherigen Auswahlmöglichkeiten zu, kann ein Fehler über den `default`-Zweig abgefangen werden.
- ✓ die Anweisungen innerhalb von Auswahlblöcken möglichst kurz und einfach gehalten sind. So bleibt die Struktur leichter lesbar und verständlich.

```
...
int grade = 4; //Note
String text = "";
switch (grade)
{
    case 1: text = "sehr gut"; break;
    case 2: text = "gut"; break;
    case 3: text = "befriedigend"; break;
    case 4: text = "ausreichend"; break;
    case 5: text = "mangelhaft"; break;
    case 6: text = "ungenügend"; break;
    default: text = "FEHLER";
}
System.out.println(grade + " entspricht " + text);
...
```

Java

Beispiel: *Grading.py*

Das Beispiel mit dem analogen Ergebnis in Python sieht wie folgt aus.

```
grade = 4 # Note
text = ""
if grade == 1:
    text = "sehr gut"
elif grade == 2:
    text = "gut"
elif grade == 3:
    text = "befriedigend"
elif grade == 4:
    text = "ausreichend"
elif grade == 5:
    text = "mangelhaft"
elif grade == 6:
    text = "ungenügend"
else:
    text = "FEHLER"

print(grade , "entspricht" , text, sep= " ")
```

Python

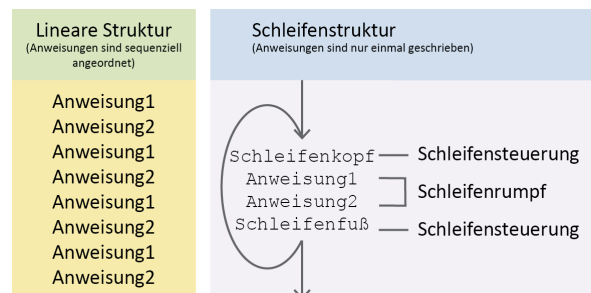
Sie erkennen, dass man mit `if`, `elif` und `else` eine identische Logik umsetzen kann und es nicht wirklich eine Notwendigkeit für eine zusätzliche Syntaxstruktur gibt.

7.8 Schleifen

Zur Umsetzung von Algorithmen müssen die gleichen Anweisungen meist mehrmals wiederholt werden. Häufig ist es dabei auch nicht vorhersehbar, wie oft diese Anweisungen ausgeführt werden sollen. In jeder Programmiersprache gibt es verschiedene Strukturen, die eine wiederholte Ausführung von Anweisungen ermöglichen. Diese Kontrollstrukturen werden als **Schleifen** bezeichnet.

Eine Schleife besteht aus einer **Schleifensteuerung** und einem **Schleifenrumpf**. Die Schleifensteuerung gibt an, wie oft oder unter welcher Bedingung die Anweisungen abgearbeitet werden. Innerhalb der Schleifensteuerung befindet sich der Schleifenrumpf, der die zu wiederholenden Anweisungen enthält.

Drei Arten von Schleifenstrukturen werden unterschieden:



Zählergesteuerte Schleife

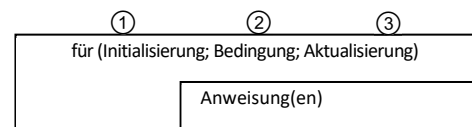
Die Schleife wird über einen Zähler gesteuert. Ein Zähler, dessen Startwert vorgegeben ist, wird für jeden Schleifendurchlauf um einen bestimmten Wert erhöht. **Vor** der Ausführung des eingeschlossenen Programmteils wird überprüft, ob der festgelegte Endwert erreicht ist. Ist der Endwert erreicht, wird das Programm hinter der Struktur fortgesetzt.

Bedingte Schleife	Kopfgesteuerte Schleife: (auch vorprüfende oder abweisende Schleife genannt) Solange eine Bedingung erfüllt ist, wird der in der Struktur eingeschlossene Programmteil ausgeführt. Anschließend wird das Programm hinter der Struktur fortgesetzt. Die Überprüfung der Bedingung erfolgt vor der Ausführung des eingeschlossenen Programmteils.
	Fußgesteuerte Schleife: (auch nachprüfende oder annehmende Schleife genannt) Ein Programmteil wird ausgeführt. Anschließend wird die Bedingung geprüft. Ist die Bedingung erfüllt, wird der Programmteil erneut ausgeführt. Erst wenn die Bedingung nicht mehr erfüllt ist, wird das Programm hinter der Struktur fortgesetzt.

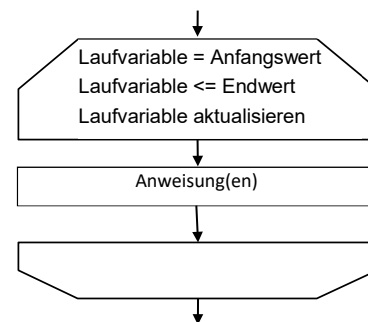
7.9 Zählergesteuerte Schleife (Iteration)

Die zählergesteuerte Schleife (auch Iteration, Zählschleife oder Laufanweisung genannt) ist dadurch gekennzeichnet, dass die Anzahl der Schleifendurchläufe durch einen Anfangs- und Endwert festgelegt ist.

- ✓ Die Variable, die die Schleifendurchläufe zählt, wird als Laufvariable oder Zähler bezeichnet.
- ✓ Die Laufvariable wird beim Eintritt in die Schleife mit dem Anfangswert initialisiert ①.
- ✓ Der Endwert kann durch eine Konstante, eine Variable oder einen Ausdruck festgelegt werden.
- ✓ Bei jedem Schleifendurchlauf wird in der Bedingung ② im Schleifenkopf geprüft, ob die Laufvariable den Endwert erreicht hat oder nicht. Ist die Bedingung erfüllt, d. h., der Endwert ist noch nicht erreicht, werden die Anweisungen im Schleifenrumpf durchgeführt.
- ✓ Der Wert, um den die Laufvariable in jedem Schritt verändert wird, heißt Schrittweite. Bei jedem Durchlauf des Schleifenkopfs wird der Wert der Laufvariablen automatisch aktualisiert ③, indem zu der Laufvariablen die festgelegte Schrittweite addiert wird (oder subtrahiert bei negativer Schrittweite).
- ✓ Wird beim Prüfen der Bedingung festgestellt, dass die Laufvariable den Endwert erreicht hat, wird die Schleife verlassen.
- ✓ Laufvariable und Endwert müssen vom selben Datentyp sein.



Struktogramm



PAP

Sie können die zählergesteuerte Schleife einsetzen, wenn z. B. die Anzahl der Schleifendurchläufe bekannt ist bzw. genau ermittelt werden kann.



Python kennt keine zählergesteuerte `for`-Schleife in klassischem Sinn, sondern nur eine Abwandlung für einen speziellen Zweck. Deshalb wird hier eine Syntax vorgestellt, wie sie in Sprachen wie Java, JavaScript, PHP, C, C# etc. üblich ist. Die konkrete Syntax soll Java sein. Dabei ist diese Schleife kopfgesteuert – also abweisend.



Syntax zur zählergesteuerten **for**-Schleife in Java

```
for (initStatement; condition; nextStatement)
    statement
```

- ✓ Das Schlüsselwort **for** leitet die Zählschleife ein.
- ✓ Danach folgt die Schleifensteuerung, die in runde Klammern eingeschlossen wird.
- ✓ Die Schleifensteuerung besteht aus drei Teilen, die jeweils mit einem Semikolon getrennt werden: dem **Initialisierungsteil** (`initStatement`), dem **Bedingungsausdruck** (`condition:`) und dem **Aktualisierungsteil** (`nextStatement`).
- ✓ Im Initialisierungsteil (`initStatement`) wird die Laufvariable definiert und mit dem Anfangswert initialisiert.
- ✓ Die Bedingung `condition:` legt das Abbruchkriterium für die Schleife fest. Ist eine Bedingung erfüllt (`True`), wird der Schleifenrumpf ausgeführt. Ist die Bedingung nicht erfüllt, wird das Programm hinter der Schleife fortgesetzt.
- ✓ Die Bedingungen werden vor einem Schleifendurchlauf geprüft, es handelt sich daher um eine kopfgesteuerte Schleife.
- ✓ Am Ende eines Schleifendurchlaufs wird die Anweisung im Aktualisierungsteil ausgeführt. Dies ist üblicherweise das Erhöhen oder Verringern der Laufvariablen. Der Schleifenrumpf kann aus einer einzelnen Anweisung oder einem Anweisungsblock bestehen, der in geschweiften Klammern eingeschlossen wird.



Einige Programmiersprachen bieten die Möglichkeit, für das Aktualisieren der Laufvariablen eine Schrittweite anzugeben. Damit kann die Laufvariable beispielsweise in Fünfer-Schritten inkrementiert bzw. dekrementiert werden. Steht diese Möglichkeit nicht zur Verfügung, kann im Bedarfsfall eine zusätzliche Variable mit der gewünschten Schrittweite initialisiert und verwendet werden. Durch Verringern der Laufvariablen haben Sie die Möglichkeit, eine Schleife zu erstellen, bei der Sie die Laufvariable herunterzählen. In Java können Sie im Aktualisierungsteil eine beliebige Anweisung integrieren.

In Python gibt es wie gesagt auch die **for**-Schleife, nur mit folgender Syntax für eine spezielle Anwendung (die Iteration über sequenzielle Datenstrukturen).

Syntax zur **for**-Schleife in Python

```
for x in Datenstruktur:
    statement
```

- ✓ Das Schlüsselwort **for** leitet auch hier die Schleife ein.
- ✓ Danach folgt die Schleifensteuerung, aber nicht auf Basis einer Zählvariablen, sondern eines Iterators, der eine komplette Datenstruktur durchläuft.



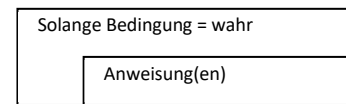
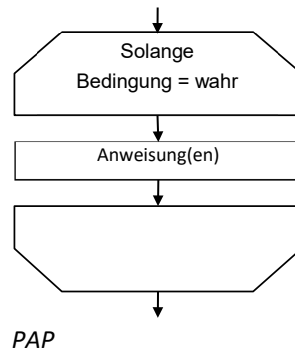
Wenn man in Python eine zählergesteuerte Schleife benötigt, verwendet man dort die **while**-Schleife.



Fehler entstehen bei **for**-Schleifen oft durch falsche Initialisierung oder falsche Angabe für die Abbruchbedingung.

7.10 Kopfgesteuerte bedingte Schleife

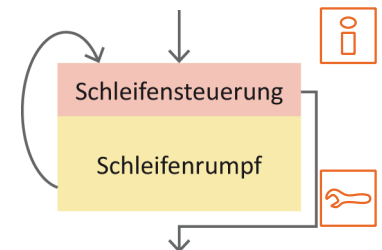
Bei einer **kopfgesteuerten** bedingten Schleife wird eine Bedingung ausgewertet, **bevor** die Anweisungsfolge innerhalb des Schleifenrumpfs ausgeführt wird. Wenn das Ergebnis der Bedingung falsch ist, bevor die Schleife das erste Mal durchlaufen werden soll, werden die darin enthaltenen Anweisungen **nicht** ausgeführt (abweisende Schleife).



Wenn das Ergebnis der Bedingung wahr ist, wird die Anweisungsfolge so lange ausgeführt, bis die Bedingung nicht mehr zutrifft.

Eine kopfgesteuerte Schleife wird auch abweisende Schleife genannt. Auch die Zählschleife ist eine kopfgesteuerte Schleife. Wenn die Laufvariable den Endwert bereits bei der Initialisierung überschreitet, werden die Anweisungen der Schleife nie ausgeführt.

Sie sollten kopfgesteuerte Schleifen einsetzen, wenn das Programm nur dann die Anweisungen der Schleife abarbeiten soll, wenn eine bestimmte Bedingung zutrifft.



Syntax zur kopfgesteuerten **while**-Schleife

- ✓ Die kopfgesteuerte bedingte Schleife wird durch das Schlüsselwort **while** eingeleitet. Danach folgt eine Bedingung **condition:**, die **am Anfang** der Schleife geprüft wird.
- ✓ **Solange** die Bedingung **condition:** im Schleifenkopf erfüllt ist, wird die Schleife ausgeführt.
- ✓ Falls die Bedingung nicht erfüllt ist, wird die Schleife beendet.
- ✓ **statement** kann eine einzelne Anweisung oder ein Anweisungsblock sein.

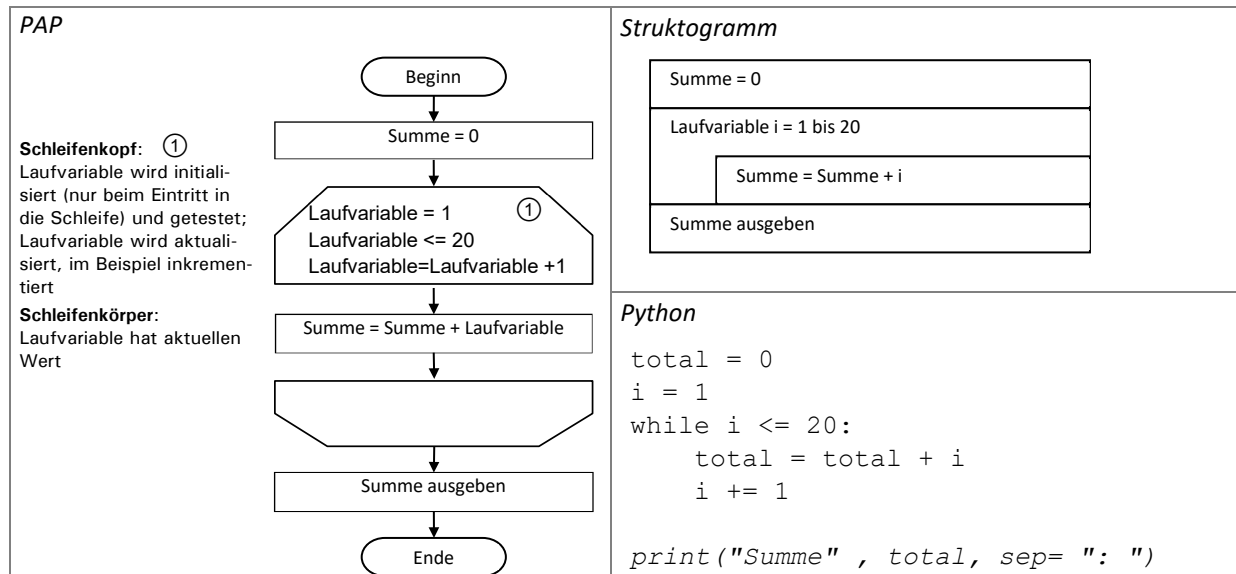
```
while condition:
    statement
```

Die **while**-Schleife besitzt nicht per se eine Zählvariable. Aber man kann jederzeit eine Zählvariable definieren und damit den Schleifenablauf steuern. In Python ist das auch der Weg zu einer zählergesteuerten Schleife.



Beispiel: AddNumbers.py

Es wird ein Programm benötigt, das die Summe der Zahlen von 1 bis 20 berechnet.



Mit der Formel $n(n+1)/2$ für die Summe der natürlichen Zahlen können Sie das Ergebnis schnell überprüfen.



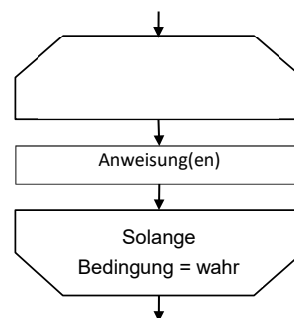
Die Abbruchbedingung muss innerhalb des Schleifenkörpers irgendwann erfüllt sein, damit die Bedingung im Schleifenkopf irgendwann nicht mehr erfüllt ist und die Schleife stoppt.

7.11 Fußgesteuerte bedingte Schleife

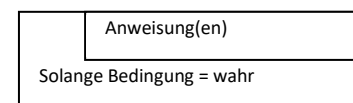
Fußgesteuerte Schleife

Bei der **fußgesteuerten** Schleife findet die Bedingungsprüfung **am Ende** der Schleife statt.

Erst **nach** Abarbeitung der Anweisungen wird am Ende der Schleife die Bedingung geprüft, ob der Schleifenrumpf noch einmal durchlaufen werden soll oder nicht (annahmende Schleife).



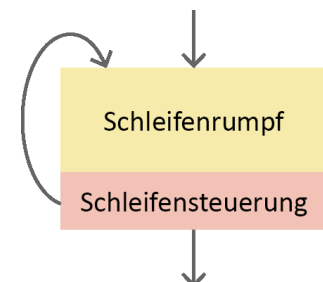
PAP



Struktogramm



Setzen Sie fußgesteuerte Schleifen ein, wenn die Anweisungen innerhalb des Schleifenrumpfs **mindestens einmal** ausgeführt werden sollen. Erst danach entscheidet eine Bedingungsprüfung, ob die Schleife erneut ausgeführt wird.



Python kennt keine fußgesteuerte Schleife in klassischem Sinn. Deshalb wird hier eine Syntax vorgestellt, wie sie in Sprachen wie Java, JavaScript, PHP, C, C# etc. üblich ist. Die konkrete Syntax soll wieder Java sein. Aber noch einmal zur Verdeutlichung – der Verzicht auf diese Art einer Schleife in Python ist keine Schwäche, sondern ein sehr großer Vorteil! Es gibt keinen Grund, warum man zwingend eine fußgesteuerte Schleife in einer Programmiersprache benötigt. Man muss lediglich eine Situation beim Erreichen der Schleife so formulieren, dass auch mit einer abweisenden Schleife die Schleife auf jeden Fall einmal durchlaufen wird, etwa durch geschickte Wahl des Wertes der Zählvariable. In Python sorgt der Verzicht also nur für klareren Code und einfachere Syntax.



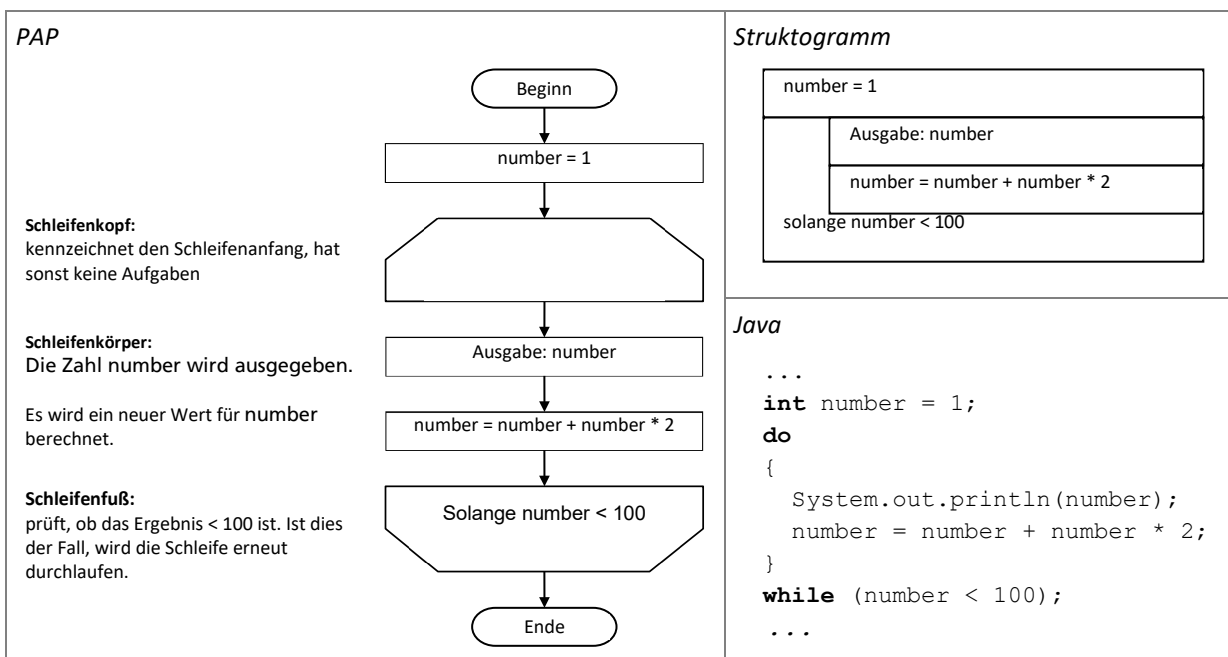
Syntax zur fußgesteuerten bedingten Schleife in Java

- ✓ Die fußgesteuerte bedingte Schleife wird mit dem Schlüsselwort `do` eingeleitet.
- ✓ Danach folgt direkt der Schleifenrumpf (`statement`). `statement` kann eine einzelne Anweisung oder ein Anweisungsblock sein.
- ✓ Die Anweisungen innerhalb des Schleifenrumpfs werden **mindestens einmal** ausgeführt.
- ✓ Der Schleifenfuß enthält die Schleifensteuerung. Sie beginnt mit dem Schlüsselwort `while`.
- ✓ Anschließend folgt die Bedingung, die in runde Klammern gesetzt wird. Der Bedingungsausdruck muss einen logischen Wert (`true` oder `false` – hier kleingeschrieben, weil Java) liefern.
- ✓ Die Schleife wird ausgeführt, **solange** die Bedingung zutrifft.

```
do
    statement
while (condition)
```

Beispiel: *PlusTwice.java*

Zu einer Zahl soll, beginnend bei 1, immer wieder das Doppelte der Zahl addiert werden, solange sie kleiner als 100 ist.



Da Python keine annehmende Schleife besitzt, soll gezeigt werden, wie man die im Ergebnis vollkommen gleiche Logik mit der abweisenden Schleife `while` umsetzen kann.

Beispiel: *PlusTwice.py*

```

number = 1
while number < 100:
    print(number);
    number = number + number * 2

```

Python



Schleifen können wie die Verzweigungsstrukturen ineinander verschachtelt werden. Genauso können sie auch die verschiedenen Kontrollstrukturen miteinander verbinden, wenn es die Aufgabenstellung erfordert.

In einigen Sprachen wird bei Schleifen auch das Schlüsselwort `repeat` zu finden sein, oft in Verbindung mit Endeanweisungen für die Schleife.

7.12 Schnellübersicht

Was bedeutet ...?	
Anweisung	Dies ist die kleinste ausführbare Einheit eines Programms.
Folge (Sequenz)	Die Anweisungen werden nacheinander, jede einmal , abgearbeitet.
Bedingung	Eine Bedingung beeinflusst den Ablauf eines Programms durch eine Ja-Nein-Entscheidung.
bedingte Anweisung	<p>In Abhängigkeit von einer oder mehreren Bedingungen wird ein Anweisungsblock einmal oder gar nicht ausgeführt.</p> <p>WENN ... DANN ...</p> <p>In Python:</p> <pre> if (...): ... </pre>
Verzweigung	<p>Zwei alternative Anweisungsblöcke stehen zur Auswahl. In Abhängigkeit von einer oder mehreren Bedingungen wird der eine oder der andere Anweisungsblock einmal ausgeführt.</p> <p>WENN ... DANN ... SONST ...</p> <p>In Python:</p> <pre> if (...): ... else: ... </pre>

Was bedeutet ...?	
geschachtelte Verzweigung	<p>In Abhängigkeit von einer oder mehreren Bedingungen wird von mehreren Anweisungsblöcken einer ausgeführt.</p> <p>WENN ... DANN ... SONST WENN ... DANN ...</p> <p>In Python:</p> <pre> if (...): ... elif (...): ... else : ... </pre>
mehrfache Verzweigung (Fallauswahl)	<p>In Abhängigkeit von einem Variablenwert wird einer von mehreren möglichen Auswahlblöcken einmal ausgeführt.</p> <p>FALLAUSWAHL ... FALL A ... FALL B ... SONST FALL ...</p> <p>In Java:</p> <pre> switch (...) { case a: ... break; case b: ... break; ... default ... } </pre> <p>In Python wird das Konzept mit <code>if</code> und <code>elif</code> umgesetzt.</p>
zählergesteuerte Schleife	<p>Es ist im Voraus bekannt oder berechenbar, wie oft ein Anweisungsblock wiederholt werden soll. Eine Zählvariable wird automatisch erhöht (oder vermindert).</p> <p>FÜR ... BIS ... (MIT DER SCHRITTWEITE ...)</p> <p>Jede Schleife kann als zählergesteuerte Version verwendet werden, wobei viele Sprachen die <code>for</code>-Schleife explizit dafür vorsehen.</p>
kopfgesteuerte Schleife	<p>Die Anweisungen in der Schleife werden in Abhängigkeit von einer oder mehreren Bedingungen 0 ... n mal ausgeführt. Die Bedingung wird am Anfang der Schleife geprüft.</p> <p>SOLANGE ... DURCHLAUFE SCHLEIFE</p> <p>In Python:</p> <pre> while (...): ... </pre>

Was bedeutet ...?	
fußgesteuerte Schleife	<p>Die Anweisungen in der Schleife werden in Abhängigkeit von einer oder mehreren Bedingungen 1 ... n mal abgearbeitet. Die Bedingungsprüfung findet am Ende der Schleife statt.</p> <p>DURCHLAUFE SCHLEIFE ... SOLANGE ...</p> <p>In Java:</p> <pre>do { ... } while (...);</pre> <p>In Python gibt es diese Schleifenform nicht.</p>

7.13 Sprunganweisungen

Nahezu alle modernen Sprachen stellen eine gewisse Anzahl an Sprunganweisungen zur Verfügung. Diese verlassen eine umgebende syntaktische Struktur (beispielsweise eine Schleife) und führen dazu, dass mit der direkten Anweisung hinter der syntaktischen Struktur weitergemacht wird. Hier soll ein kurzer Überblick anhand der Syntax von Python die Konzepte verdeutlichen.

Abbruch mit **break**

Die Anweisung `break` verlässt eine Syntaxstruktur sofort, wenn diese Stelle im Quelltext erreicht wird. Das kann man in Schleifen nutzen.

Fortsetzen mit **continue**

Mit der Sprunganweisung `continue` können Sie an einer bestimmten Stelle innerhalb des Schleifenblocks unmittelbar den nächsten Schleifendurchlauf erzwingen und die nachfolgenden Anweisungen innerhalb des Schleifenblocks ignorieren. Die Verwendung von `continue` findet man in der Praxis nicht sonderlich oft. Alternativ kann man fast immer Bedingungen so formulieren, dass man darauf verzichten kann.

Rückgabe mit **return**

Im Zusammenhang mit Funktionen und Methoden gibt es die Sprunganweisung `return`. Diese verlässt eine Funktion oder Methode und gibt in der Regel einen Rückgabewert zurück.

Unterbrechung mit **raise**

Die vierte Sprunganweisung bei Python nennt sich `raise`. Damit werfen Sie eine sogenannte Ausnahme (Exception) aus, die den normalen Programm- beziehungsweise Skriptablauf unterbricht und zu einer Behandlungsroutine springt. Man spricht hier vom „Werfen“ einer Ausnahme. Ausnahmebehandlung gehört zu den fortgeschrittenen Programmier Techniken und wird im Zusammenhang mit Fehlerbehandlung im Heft nur kurz angesprochen.



Die meisten Sprachen werfen Ausnahmen mit dem Schlüsselwort `throw`.

7.14 Endlosschleifen

Endlosschleifen (Infinite loop) sind Schleifen, die nach jeder Abarbeitung erneut abgearbeitet werden, falls die Ausführung nicht durch äußere Einflüsse oder eine Anweisung aus dem Inneren (etwa die Sprunganweisung `break`) abgebrochen wird.

Endlosschleifen können durch Fehler entstehen, wenn die Abbruchbedingung nicht definiert ist oder nicht eintreten kann. Es gibt aber auch sinnvolle Anwendungen. So kann etwa die permanente Abfrage der Mausposition bei einer grafischen Oberfläche eine gewollte Endlosschleife sein. Eine andere Situation ist, dass man eine Schleife eine unbekannte oder unbestimmte Anzahl von Durchläufen ausführen möchte und der Abbruch dann durch eine Eingabe erfolgen soll. Das soll die folgende Python-Übung demonstrieren.

Eine Endlosschleife in Python (Infinite.py)

```
while True:
    eingabe = input("Geben Sie etwas ein:\n")
    print("Ihre Eingabe", eingabe, sep=": ")

    if eingabe == "a":
        print("Ihre Eingaben werden beendet.")
        break;
```

- ✓ Wegen der Anweisung `while True:` wird die Bedingung der Schleife immer wahr sein. Deshalb kann man die Schleife nur aus dem Inneren heraus verlassen, wenn man nicht das Programm von außen beenden will.
- ✓ Dazu finden Sie die Anweisung `break`, die bedingt ausgelöst wird. Wenn der Anwender den Kleinbuchstaben `a` eingibt, wird die Endlosschleife beendet.

7.15 Übungen

Übung 1: Verzweigung

Übungsdatei: --

Ergebnisdateien: *uebung07.pdf, Maximum.py*

1. Erstellen Sie ein Struktogramm, um die Zahlen 543 und 246 zu vergleichen und die größere von beiden auszugeben.
2. Setzen Sie das Struktogramm in Python-Code um.

Übung 2: Geschachtelte Verzweigung

Übungsdatei: --

Ergebnisdateien: *uebung07.pdf, Anrede.py*

1. Für eine Nachricht soll von einem Programm automatisch die Anrede formuliert werden. Folgende Variablen existieren:

- ✓ name: Name
- ✓ sex: Geschlecht
- ✓ currentHour: Uhrzeit (Stundenangabe)

Die Anrede soll je nach Tageszeit mit „Guten Morgen“ (0–9 Uhr), „Guten Tag“ (10–17), „Guten Abend“ (18–0 Uhr) beginnen und anschließend mit „Herr xxx“ bzw. „Frau xxx“ fortgesetzt werden. Für xxx soll der entsprechende Name eingesetzt werden.

Schreiben Sie ein entsprechendes Struktogramm.

2. Erstellen Sie aus dem Struktogramm ein Python-Programm. Für Zeichenketten stellt Python den Datentyp String zur Verfügung. Der Name und das Geschlecht sollen durch Variablen vorgegeben werden.

Übung 3: Benutzereingaben überprüfen

Übungsdatei: --

Ergebnisdateien: *uebung07.pdf, Zugang.py*

1. In Python gibt es die Built-in Funktion `input()`, um eine Benutzereingabe von der Konsole entgegenzunehmen (`input(„Geben Sie den Username ein\n“)`). Dieser kann in einer Variablen gespeichert werden. Nehmen Sie auf diese Weise einen Benutzernamen und ein Passwort entgegen.
2. Vergleichen Sie den Benutzernamen mit dem Wert „gast“ und das Passwort mit dem Wert „geheim“.
3. Wenn die Eingaben in Kombination übereinstimmen, zeigen Sie die Meldung „Zugang erlaubt“ an.
4. Wenn die Eingaben nicht stimmen, zeigen Sie die Meldung „Zugang verboten“ an.

Übung 4: Gerade Zahlen in der Konsole anzeigen

Übungsdatei: --

Ergebnisdateien: *Geradezahlen1.py, Geradezahlen2.py, Geradezahlen3.py, uebung07.pdf*

1. Geben Sie in der Konsole die geraden Zahlen von 0 bis 100 (exklusive) aus.
2. Überlegen Sie sich mehrere Wege zur Umsetzung.

Übung 5: Kopfgesteuerte bedingte Schleife

Übungsdatei: --

Ergebnisdateien: *uebung07.pdf, DivideBy5.py*

1. Schreiben Sie ein Programm, das zu einer eingelesenen Integer-Zahl ermittelt, wie oft mit dieser Zahl eine Division durch 5 durchgeführt werden kann. Die ermittelte Anzahl wird ausgegeben. Initialisieren Sie die einzulesende Integer-Zahl wie folgt:

```
number = int(input("Zahl\n"));
```

2. Beim Einlesen können Sie die Funktion `input()` verwenden, die einen Text über die Standardeingabe einliest. Diese Eingabe muss aber vor einer mathematischen Berechnung in eine Zahl konvertiert werden. Deshalb verwendet man in Python die Konvertierungsfunktion `int()`.
3. Zum Test, ob eine Zahl durch 5 geteilt werden kann, kann man das Moduloverfahren verwenden.
4. Beachten Sie, dass Python bei der Division eine Gleitkommazahl erzeugt, wenn sich die Operanden nicht ganzzahlig dividieren lassen. Sie müssen deshalb ggf. mit `int()` eine ganze Zahl erzwingen.

Übung 6: Verzweigungen und Schleifen

Übungsdatei: --

Ergebnisdateien: *uebung07.pdf*, *Days.py*

1. Erstellen Sie ein Programm, das mit dem gegebenen letzten Tag des Vorjahres (Eingabe über die Konsole) alle Sonntage ausgibt, die auf den ersten Tag eines Monats fallen. Verwenden Sie Zahlen von 1 (Montag) bis 7 (Sonntag) für die Tage (auch als Eingabewert für den letzten Tag des Vorjahres) und 1 (Januar) bis 12 (Dezember) für die Monate.

Sie können nach diesem Verfahren vorgehen, aber auch eigene Ansätze versuchen:

- ✓ Lassen Sie den Anwender den letzten Tag des Vorjahres eingeben. Achten Sie auf den Datentyp.
- ✓ Mit einer äußeren Schleife iterieren Sie über alle Monate des Jahres. Die Werte 1 ... 12 stehen für Januar bis Dezember.
- ✓ Da die Monate unterschiedlich viele Tage haben, müssen Sie im Inneren der Schleife pro Monat diese Anzahl ermitteln. Dabei sind Schaltjahre zu berücksichtigen. Das können Sie aber rein formal implementieren und den Wert 1 für ein Schaltjahr und 0 für ein sonstiges Jahr vorgeben. Sie geben ein, ob es sich um ein Schaltjahr handeln soll oder nicht. Diese Eingabe wird aber nicht verifiziert, und die Verantwortung auf Richtigkeit liegt beim Anwender.
- ✓ Eine innere Schleife überprüft, ob ein Tag der 1. des Monats und gleichzeitig Sonntag ist. Dazu kann das Moduloverfahren zum Einsatz kommen.

8 Elementare Datenstrukturen

In diesem Kapitel erfahren Sie

- ✓ was Arrays, Records, Tupel, Listen, Dictionaries, Mengen sowie andere elementare Datenstrukturen sind
- ✓ wie Sie Datenstrukturen definieren und einsetzen
- ✓ welche speziellen sequenziellen Datenstrukturen Python bereitstellt
- ✓ wie Sie spezielle Datenstrukturen, wie Stapel, Schlangen und Listen, implementieren

Voraussetzungen

- ✓ Deklaration von Variablen und Konstanten
- ✓ Kontrollstrukturen

8.1 Warum werden Datenstrukturen benötigt?

Mithilfe der ganzzahligen Datentypen und der Gleitkommatypen können Sie in Python Zahlen darstellen, ebenso Texte in Form von Strings. Alle Sprachen stellen solche Konstruktionen (teils implizit) zur Verfügung.

Häufig werden in der Programmierung aber auch **zusammengehörige** Daten verwendet, wie beispielsweise verschiedene Zahlen, die zu einem Vektor bzw. einer Matrix gehören, oder Adressen mit ihren Bestandteilen. Viele Programmiersprachen besitzen dafür **elementare Datenstrukturen**, die meist auch **sequenzielle Datentypen** oder auch **aggregierte Datentypen** genannt werden. Es gibt beispielsweise folgende sequenzielle Datentypen:

- ✓ Ein **Array**, auch Feld genannt, wird in manchen Sprachen für Elemente **gleichen** Datentyps verwendet, z. B. für einen Vektor. Andere lassen auch Elemente unterschiedlichen Datentyps zu, und damit gibt es dort keine Unterscheidung zu einem Record.
- ✓ Ein **Record**, auch Verbund oder Struktur (Struct) genannt, wird für eine Anzahl Elemente **unterschiedlichen** Datentyps verwendet, z. B. für eine Adresse. Verbundstrukturen werden syntaktisch oft unterschiedlich umgesetzt.



Nahezu alle Programmier- und Skriptsprachen unterstützen solche sequenziellen Datentypen, wenngleich nicht immer die gleichen Strukturen. Allgemein bestehen beispielsweise in Python sequenzielle Datentypen einfach aus einer Aneinanderreihung von einzelnen Werten. Damit kann man Variablen anlegen, die mehrere Werte zur Verfügung stellen. Beachten Sie, dass gerade Python den Umgang mit sequenziellen Datenstrukturen erheblich vereinfacht und viele äußerst mühsame und schwierige Schritte mit der Steuerung von Zeigern, wie sie in anderen (vor allem alten) Sprachen notwendig sind, überflüssig macht. Dennoch sollten Sie die theoretischen Grundlagen kennen, die hinter den verschiedenen sequenziellen Datenstrukturen liegen.

Zuerst werden die Datenstrukturen Array und Record behandelt, wobei bewusst einkalkuliert wird, dass Python ausdrücklich beide Strukturen nicht bereitstellt. Nur da sie in vielen anderen Programmiersprachen vorkommen und auch von historischer Bedeutung sind, muss man sie als Programmierer zumindest grundsätzlich kennen. Wie sonst auch, wenn es keine Python-Syntax für eine Struktur gibt, werden wir bei Arrays (dem wichtigeren der beiden Typen) auf Java für schematische Listings zurückgreifen.

8.2 Arrays

Arrays nutzen

Stellen Sie sich eine Berechnung vor, die 20 Ergebnisse liefert. Die Ausgabe der 20 Werte soll erfolgen, wenn alle Berechnungen abgeschlossen wurden. Auf jedes Ergebnis soll der Zugriff möglich sein. Sie könnten zum Speichern der Ergebnisse 20 Variablen definieren ($x_1, x_2, x_3, x_4 \dots$). Bei der Berechnung müssten Sie dann jeder Variablen einzeln einen Wert zuweisen und bei der Ausgabe jede Variable einzeln ansprechen – das sind 40 verschiedene Anweisungen.

Durch die Verwendung eines Arrays können Sie **alle** Ergebnisse unter **einem** Bezeichner speichern. Der Zugriff auf die einzelnen Ergebnisse erfolgt über einen sogenannten **Index**. Der Index entspricht dabei der aktuellen Position eines Ergebnisses im Array. Zum Beispiel befindet sich das Ergebnis der 13. Berechnung auf Position 12 des Arrays (Index startet meist mit 0).

Der Aufwand für obige Problemlösung reduziert sich dann mithilfe einer Schleife auf wenige Anweisungen, unabhängig davon, wie viele Elemente das Array besitzt.

Merkmale von Arrays

- ✓ Ein Array hat einen Namen.
- ✓ Arrays bestehen in einigen Sprachen (z. B. Java) immer aus Elementen **desselben** Datentyps. In nicht streng typisierten Sprachen wie JavaScript oder PHP müssen die Elemente explizit **nicht** vom selben Datentyp sein, weshalb dort die Grenzen zu einem Record vollkommen aufgehoben sind.
- ✓ Ein Element eines Arrays wird über einen **Index** angesprochen; die einzelnen Elemente haben keinen eigenen Namen.
- ✓ In den meisten Programmiersprachen (z. B. C, C#, Java, JavaScript) hat das erste Element eines Arrays den Index 0, das zweite Element hat den Index 1, das dritte Element hat den Index 2 usw. Man nennt das **nullindiziert**. Der Index eines Arrays mit n Elementen reicht dann von 0 bis $n-1$. Es gibt allerdings auch Sprachen, bei denen der erste Index mit 1 beginnt.
- ✓ Wird die Anzahl der Elemente bei der Definition des Arrays festgelegt, wird ein statisches Array erzeugt. Wird die Anzahl nicht angegeben, wird ein dynamisches Array erzeugt. Nicht alle Programmiersprachen erlauben dynamische Arrays. In Java beispielsweise sind (vollständig) dynamische Arrays nicht erlaubt. Auch statische Arrays sind nicht in allen Programmiersprachen erlaubt. In JavaScript sind Arrays beispielsweise **immer** dynamisch.

Die Schreibweise für Arrays kann in verschiedenen Programmiersprachen unterschiedlich sein.



Ein- und mehrdimensionale Arrays

Einfache Arrays bestehen aus n Elementen, die über einen Index angesprochen werden können. Das Array kann beispielsweise einen Vektor speichern. Das Array repräsentiert dann eine Dimension. Besteht ein Array-Element wiederum aus einem Array, z. B. um eine Tabelle zu speichern, handelt es sich um ein zweidimensionales Array. Ein zweidimensionales Array besteht aus $m \times n$ Elementen, die über zwei Indizes angesprochen werden. Ein Array kann auch mehr als zwei Dimensionen besitzen. Für jede weitere Dimension wird ein weiterer Index für den Zugriff auf die Elemente benötigt.

8.3 Eindimensionale Arrays

Syntax für die Definition von Arrays in Java

Sie definieren in Java eine Array-Variable, indem Sie dem Datentyp ^① ein Klammerpaar `[]` anfügen.

```
①type[] identifier;
```

Der Name des Arrays und der Datentyp seiner Elemente sind somit bekannt. Unbekannt ist noch, wie viele Elemente das Array enthalten soll. Bevor ein Array verwendet werden kann, muss es erzeugt werden.

Syntax für die Erzeugung von Arrays in Java

- ✓ Um ein Array zu erzeugen, verwenden Sie den Operator `new` ^②. Die Array-Größe (die Anzahl der Array-Elemente) wird dazu in eckigen Klammern hinter dem Datentyp angegeben. Sie können in Java die Größe später **nicht** mehr ändern.

```
②identifier = new type[size];
```

- ✓ Die Größe muss vom Datentyp `int` sein. Sie können sie mit einer Zahl, mit einer Konstanten oder einem Ausdruck angeben. Der Ausdruck wird zur Laufzeit ausgewertet, und somit wird auch die Größe erst zur Laufzeit festgelegt.
- ✓ Bei der Erzeugung des Arrays mit `new` werden die Elemente automatisch mit Default-Werten für den jeweiligen Datentyp initialisiert. So erhalten z. B. Elemente vom Typ `int` den Wert 0.
- ✓ Sie können die Definition und das Erzeugen des Arrays in einer Anweisung durchführen ^③.

```
type[] identifier1 = new type[size1];③
```



Die Array-Variable `identifier` enthält nicht das Array selbst, sondern einen sogenannten **Zeiger** auf das erzeugte Array. Zeiger, auch Referenz genannt, lernen Sie im weiteren Verlauf dieses Kapitels kennen.

Syntax für die Erzeugung und Initialisierung von Arrays in Java

- ✓ Bereits bei der Erzeugung des Arrays können Sie die Elemente mit Daten initialisieren.

```
type[] identifier1 = {value0,  
value1,...valueN};
```

- ✓ Geben Sie auf der rechten Seite des Gleichheitszeichens in geschweiften Klammern `{ }` die gewünschten Werte ein. Trennen Sie dabei die einzelnen Werte durch Kommata.
- ✓ Durch die Anzahl der Werte, die Sie innerhalb der geschweiften Klammern angeben, wird gleichzeitig die Größe des Arrays festgelegt.

Beispiele für die Definition, Erzeugung und Initialisierung von Arrays

```
①double[] field1;  
field1 = new double[15];           //Array mit 15 Elementen vom Typ double  
②int[] field2 = new int[20];        //Array mit 20 Elementen vom Typ int  
③double[] field3 = {1.2, 42.3, 27.0, 12.567}; //Elemente mit Werten  
                                           initialisieren  
④int a = 1;  
int b = 6;  
int c = 4;  
int[] field4 = {a, b, c, a + b + c}; //Ausdruecke verwenden
```

- ① Definition und Erzeugung eines Arrays in zwei separaten Anweisungen
- ② Definition und Erzeugung eines Arrays in einer einzigen Anweisung
- ③-④ Definition, Erzeugung und Initialisierung eines Arrays mit individuellen Werten

Mit Arrays arbeiten

Wie z. B. Variablen primitiver Datentypen können Sie auch Array-Elementen Werte zuweisen und die Inhalte auslesen.

Syntax für Wertzuweisungen und das Auslesen von Array-Elementen

- ✓ Der Zugriff auf Array-Elemente erfolgt durch den Index, der bei 0 beginnt und entsprechend für das letzte Array-Element den Wert Array-Größe -1 besitzt.

```
identifier[intExpression] = expression; ①  
type identifier1 = identifier[intExpression1]; ②
```
- ✓ Der Index kann ein beliebiger Ausdruck sein, der ein Ergebnis vom Typ `int` liefert.
- ✓ Sie können einem über den Index bestimmten Array-Element einen Wert zuweisen ①.
- ✓ Über den Index (`intExpression1`) können Sie den Wert des entsprechenden Array-Elements auslesen ② und beispielsweise in einer anderen Variablen speichern.

8.4 Records

Gerade ältere Sprachen verwenden für die sequenzielle Speicherung von Daten eine Struktur, die Record oder Struct genannt wird. Allerdings kann sich beispielsweise die Schreibweise für Records in den verschiedenen Programmiersprachen unterscheiden.

- ✓ In vielen objektorientierten Programmiersprachen, wie z. B. Java, gibt es keine Records – dafür stehen Klassen zur Verfügung.
- ✓ In Sprachen ohne strenge Typüberwachung wie JavaScript oder PHP sind Records und Arrays meist identisch.

Merkmale von Records

- ✓ In einem Record lassen sich Elemente zusammenfassen, die im Gegensatz zu Arrays in manchen Sprachen explizit aus **verschiedenen** Datentypen bestehen können.
- ✓ Jedes Element eines Records hat einen Namen und einen Datentyp. Der Zugriff auf die Elemente des Records erfolgt über die Namen.

8.5 Zeichenketten

Grundlagen zu Zeichenketten

Zeichenketten, auch **Strings** genannt, sind eine Folge von Zeichen. Die maximale Länge einer Zeichenkette ist von der Sprache abhängig. Zeichenketten werden in den verschiedenen Sprachen unterschiedlich gespeichert und gehandhabt.

- ✓ Die meisten Sprachen besitzen einen Datentyp `String` oder ähnlich (wie Python `str`). Dieser kann von primitiver Natur sein, oder aber die Zeichenkette ist ein Objekt. In einigen Sprachen gibt es beide Varianten.
- ✓ In einigen Sprachen muss bei der Definition der Zeichenkette deren maximale Länge angegeben werden, in anderen Sprachen ist die Länge variabel bzw. wird dynamisch festgelegt.
- ✓ Meist werden Zeichenketten in doppelten Hochkommata dargestellt. Einige Sprachen (etwa Python) gestatten auch einfache Hochkommata.
- ✓ In einigen Sprachen wird die Zeichenkette mit 0 abgeschlossen, um deren Ende zu kennzeichnen (Null-terminierte Zeichenkette).

Informationen zur Arbeit mit Zeichenketten

Bearbeitet werden Zeichenketten immer über Funktionen bzw. Methoden, da Zeichenketten in der Regel nicht zu den elementaren Datentypen zählen. Die meisten Sprachen bieten Funktionen bzw. Methoden, welche

- ✓ Zeichenketten-Variablen einen Wert (eine Zeichenkette) zuweisen,
- ✓ die Länge einer Zeichenkette bestimmen,
- ✓ Teilzeichenketten ermitteln,
- ✓ Zeichen in die Zeichenkette einfügen bzw. daraus entfernen,
- ✓ Zeichenketten bzw. Teilzeichenketten kopieren,
- ✓ Zeichenketten vergleichen.



In Python sind Zeichenketten eine sequenzielle Ansammlung von Zeichenliteralen. Dementsprechend kann man darauf die gemeinsamen Techniken zur Bearbeitung verwenden.

8.6 Tupel und Listen

Tupel und Listen kommen nicht in allen Programmiersprachen explizit als sequenzielle Datentypen vor, aber in Python sind sie die zentrale sequenzielle Datenstruktur und im Grunde das, was in anderen Sprachen Arrays darstellen. Sie sind wie Strings eine Aneinanderreihung von beliebigen Werten.

Zuerst soll gezeigt werden, wie man in Python diese beiden sequenziellen Datenstrukturen anlegt, bevor die theoretischen Grundlagen erläutert werden. Sie werden sehen, wie Python den Umgang mit diesen sequenziellen Datenstrukturen erheblich vereinfacht und den Umgang intuitiv werden lässt. Dennoch sollten Sie auch die theoretischen Grundlagen verstehen, die hinter den Listen und Tupeln liegen.

Tupel

Tupel entstehen in Python ganz einfach, wenn man in runden Klammern Daten kommagetrennt hintereinanderschreibt.

```
zahlen = (2, 3, 5, 7, 11)
```

Es können beliebige Datentypen in einem Tupel auftauchen.

```
person = ("Hans", "Dampf", 45, True, None)
```

Tupel können auch verschachtelt werden. Das bedeutet, es tauchen Tupel in Tupel auf.

```
tupel2 = ((1, 2), (3, 4))
```

Einträge in Tupel ansprechen

Nun stellt sich die Frage, wie man einzelne Werte in einem Tupel ansprechen kann? Es geht über den Index. Dieser muss in eckigen Klammern angegeben werden.

Beispiel:

```
zahlen[2]
```

Tupel sind nullindiziert. Der erste Wert in einem Tupel hat also den Index 0. Das geht mit so gut wie allen modernen Sprachen einher, die Datenstrukturen immer nullindizieren.

Bei verschachtelten Tupeln werden die Indizes in aufeinander folgenden eckigen Klammern notiert, Ganz so wie es in den meisten anderen Programmiersprachen bei Arrays üblich ist.

Beispiel:

```
tupel2[1][1]
```

Beispiele für die Definition, Erzeugung und Ausgabe von Tupeln

```
① primzahlen = (2, 3, 5, 7, 11, 13, 17, 19)
② person = ("Hans", "Dampf", 45, True, None)
③ tupel1 = (primzahlen, person)
④ tupel2 = ((1,2), (3,4))
print(primzahlen[0])
print(person[1])
print(tupel1[0])
print(tupel1[1][0])
print(tupel2[1][1])
```

- ① Ein Tupel mit Zahlen wird angelegt.
- ② Ein Tupel mit unterschiedlichen primitiven Datentypen wird angelegt.
- ③ Ein Tupel mit zwei anderen Tupeln, die an anderer Stelle deklariert wurden
- ④ Ein Tupel mit zwei anderen Tupeln, die innerhalb des äußeren Tupels deklariert werden

Führen Sie das Programm aus.

- ✓ Zuerst sprechen wir den ersten Eintrag in dem Tupel `zahlen` an. Das ist der Wert 2.
- ✓ Dann wird der zweite Eintrag in dem Tupel `person` genommen ("Dampf").
- ✓ Mit `tupel1[0]` wird das Tupel `primzahlen` in dem Tupel `tupel1` angesprochen. Das geht auch und damit bekommt man das vollständige geschachtelte Tupel.
- ✓ Dann wird mit `tupel1[1][0]` genau ein Element in dem verschachtelten Tupel `tupel1` angesprochen ("Hans").
- ✓ Zum Schluss wird auf den Wert 4 zugegriffen, das zweite Element in dem zweiten verschachtelten Tupel.

Wenn Sie bei dem Index einen Bereich angeben, können Sie damit gezielt Elemente aus einem sequenziellen Datentyp extrahieren. Sie geben dazu in den eckigen Klammern den Anfangs- und den Endeindex an:

Beispiel:

```
person = "Hans"
print(person[1:2])
```

Das würde nur „an“ ausgeben. Das Verfahren funktioniert auch bei allen anderen sequenziellen Datentypen.



Die Anzahl der Elemente in einer sequenziellen Liste oder auch in anderen Datentypen dieser Art erhalten Sie in Python mit der Built-in-Funktion `len()`.

Dynamische Listen

Nun gibt es in Python noch Listen als sequenziellen Datentyp. Auf den ersten Blick sind Listen und Tupel fast identisch, nur werden Listen in eckige Klammern gesetzt. Im einfachsten Fall erzeugt man eine leere Liste durch eckige Klammern ohne Inhalt.

Beispiel:

```
zahlen = []
```

Die Liste müsste man nun dynamisch füllen. Aber man kann auch Vorgabewerte direkt bei der Deklaration angeben. So würde das Beispiel von oben mit Listen aussehen:

Beispiele für den Umgang mit Listen in Python

```
primzahlen = [2, 3, 5, 7, 11, 13, 17, 19]
person = ["Hans", "Dampf", 45, True, None]
tupel1 = [primzahlen, person]
tupel2 = [[1, 2], [3, 4]]

print(primzahlen[0])
print(person[1])
print(tupel1[0])
print(tupel1[1][0])
print(tupel2[1][1])
```

Warum Listen und Tupel?

Doch wozu gibt es noch Listen, wenn es schon Tupel gibt?

- ✓ Tupel können nach der Erzeugung nicht mehr geändert werden (immutable), während Listen nachträglich veränderbar (mutable) sind.
- ✓ Tupel sind performanter, Listen dynamisch.

Methoden für Listen

Wir werden nun die OOP ins Spiel nehmen müssen, aber für den Einsatz von Listen sind die Listenmethoden elementar. Und Methoden gehören explizit zur OOP. Python erlaubt explizit Programmierung nach dem OO-Paradigma, auch gemischt mit anderen Programmierstilen.

Methoden versteht man als Beschreibung der objekt- und klassenbezogenen Funktionalität. Sie repräsentieren das, was Objekte tun. Sie sind das OO-Analogon zu Prozeduren/Funktionen in der prozeduralen/funktionalen Programmierung. Man ruft Methoden über das Voranstellen des Objekts oder der Klasse auf. Das Objekt oder die Klasse werden mit einem Punkt abgedreht (Punkt- oder Dot-Notation).

Für Listen stehen in Python einige spannende Methoden zur Verfügung:

Methode	Beschreibung
<code>append(x)</code>	Fügt am Ende der Liste ein Element hinzu
<code>extend(L)</code>	Erweitert die Liste, indem alle Elemente in der angegebenen Liste angehängt werden
<code>insert(i, x)</code>	Fügt einen Wert an einer bestimmten Stelle ein. Das erste Argument ist der Index des Elements, vor dem man einfügen soll.
<code>remove(x)</code>	Entfernen des ersten Elements aus der Liste, dessen Wert x ist
<code>pop([i])</code>	Entfernen des Elements an der angegebenen Position in der Liste. Es wird zurückgegeben. Wenn kein Index angegeben ist, entfernt <code>pop()</code> das letzte Element in der Liste und gibt es zurück.
<code>index(x)</code>	Rückgabe des Index vom ersten Element, dessen Wert x ist
<code>count(x)</code>	Anzahl des Vorkommens von x in der Liste
<code>sort(cmp=None, key=None, reverse=False)</code>	Sortieren der Elemente der Liste (die Argumente können für Sortierung verwendet werden, müssen aber nicht)
<code>reverse()</code>	Umdrehen der Elemente der Liste

Beispiel: *Liste2.py*

```
a = [66.25, 333, 333, 1, 1234.5]

print(a.count(333), a.count(66.25), a.count('x'))
a.insert(2, -1)
a.append(333)
print(a)
print(a.index(333))
print(a.remove(333))
print(a)
a.reverse()
print(a)
a.sort()
print(a)
print(a.pop())
print(a)
```

Das ist die Ausgabe des Beispiels:

```
2 1 0
[66.25, 333, -1, 333, 1, 1234.5, 333]
1
None
[66.25, -1, 333, 1, 1234.5, 333]
[333, 1234.5, 1, 333, -1, 66.25]
[-1, 1, 66.25, 333, 333, 1234.5]
1234.5
[-1, 1, 66.25, 333, 333]
```

Operatoren bei sequenziellen Datentypen

Mit dem **String-Verkettungsoperator** kennen Sie bereits einen Operator, der im Grunde ein allgemeiner Verkettungsoperator bei sequenziellen Datentypen ist. Er hat nur bei Zeichenketten die recht triviale Wirkung der String-Verkettung.

Pythons **Membership-Operatoren** (Mitgliedschaftsoperatoren) testen die Mitgliedschaft eines Operanden in einer Sequenz wie z. B. String, Liste oder Tupel. Es gibt zwei Mitgliedsoperatoren.

- ✓ Der Operator `in` testet, ob sich der erste Operand in dem zweiten Operanden (der Sequenz) befindet.
- ✓ Der zweite Mitgliedsoperator ist streng genommen kein eigener Operator, sondern die Kombination aus `not` und `in` und testet, ob sich der erste Operand nicht in dem zweiten Operanden befindet.

8.7 Dictionaries

Neben den sequenziellen Datentypen Listen, Strings und Tupel gibt es in Python eine weitere Kategorie von Datentypen, die allgemein „Mapping“ genannt wird und derzeit nur einen implementierten Typ besitzt: das Dictionary. Das ist ein assoziatives Feld, etwas wie ein Hash, Map oder assoziatives Array, wie es in anderen Sprachen genannt wird. Ein Dictionary besteht aus Schlüssel-Objekt-Paaren (Key-Value-Paaren). Zu einem bestimmten Schlüssel gehört immer ein Objekt beziehungsweise Wert. Es handelt sich um eine in Python sehr wichtige Struktur, die aber ziemlich einfach ist.

Wie Listen können Dictionaries dynamisch zur Laufzeit verändert werden. Sie haben aber ein paar interessante zusätzliche Methoden zur Verfügung, die sich aus dem assoziierten Schlüssel ergeben. Anlegen kann man im einfachsten Fall ein leeres Dictionary durch geschweifte Klammern ohne Inhalt.

Beispiel:

```
map = { }
```

Wenn bereits Inhalt vorhanden sein soll, gibt man kommasepariert die gewünschten Vorgabewerte an. Die Inhalte können wie bei Listen von verschiedenen Werten sein. Bei den Schlüsseln gilt jedoch die Einschränkung, dass nur Instanzen unveränderlicher (immutable) Datentypen verwendet werden können. Damit fallen Listen und Dictionaries als Schlüssel heraus. Beachten Sie nur, dass Sie immer ein Key-Value-System angeben, das durch den Doppelpunkt getrennt wird. Das kennt man so ja von diversen ähnlichen Konzepten.

Beispiel:

```
person = { "name" : "Marvin", "alter" : 31, 1 : True }
```

Der Zugriff auf einzelne Werte erfolgt wie bei anderen sequenziellen Datentypen in Python über den Index in eckigen Klammern.

Beispiel: *Dic1.py*

```
a = [66.25, 333, 333, 1, 1234.5]

print(a.count(333), a.count(66.25), a.count('x'))
a.insert(2, -1)
a.append(333)
print(a)
print(a.index(333))
print(a.remove(333))
print(a)
```

```
a.reverse()
print(a)
a.sort()
print(a)
print(a.pop())
print(a)
```

Das ist die Ausgabe des Beispiels:

```
Hans
<class 'dict'>
{'vname': 'Hans', 'nname': 'Dampf'}
Hans
Hans
Dampf
False
False
```

- ✓ Zuerst wird in dem Beispiel ein Tupel `person` angelegt und danach auf den ersten Eintrag darin zugegriffen.
- ✓ Dann sehen Sie ein Dictionary `person2`. Die Ausgabe mit `type()` zeigt, dass es vom Typ `dict` ist.
- ✓ Die `print()`-Funktion kann direkt auf alle sequenziellen Datentypen angewendet werden und gibt dann deren vollständigen Inhalt aus. Das geht auch bei Dictionaries.
- ✓ Im Folgeschritt wird über den Textindex auf genau einen Wert im Dictionary zugegriffen.
- ✓ Mit dem Membership-Operator `in` kann man auch hervorragend in einer `for`-Schleife über ein Dictionary iterieren. Die Variable der Schleife ist der Wert.
- ✓ Mit dem Dictionary `daten` sehen Sie, dass man auch Zahlen als Schlüssel verwenden kann.
- ✓ Sogar boolesche Schlüssel sind möglich, was das Dictionary logisch demonstriert.
- ✓ Selbst Tupel kann man verwenden, was Sie mit dem Dictionary auch noch logisch erkennen können.

8.8 Mengen

Mengen bedeuten eine ungeordnete Zusammenfassung von bestimmten wohlunterschiedenen Dingen. Der Datentyp `set` dient in Python zum Erstellen solch einer ungeordneten Sammlung von einmaligen und unveränderlichen Elementen. In anderen Worten: Ein Element kann in einem `set`-Objekt nicht mehrmals vorkommen, was bei Listen und Tupel jedoch möglich ist.

Das Erzeugen einer Menge geschieht mit der Built-in-Funktion `set()`. Als Parameter können beliebige Daten angegeben werden.

Beispiel:

```
x = set("Die Antwort ist 42.")
```

Sets sind wie gesagt so implementiert, dass sie keine veränderlichen (mutable) Objekte erlauben. Damit sind beispielsweise keine Listen als Elemente erlaubt.

Auch wenn Sets keine veränderlichen Elemente enthalten können, sind sie selbst veränderlich. Wir können zum Beispiel neue Elemente einfügen. Dazu gibt es die Methode `add()`.

Beispiel:

```
staedte = set(["Mainz", "Eppstein"])
staedte.add("Bodenheim")
```

8.9 Besondere Datenstrukturen anhand von Stapel (Stack) und Schlangen (Queue)

Aufbauend auf den sequenziellen Grundstrukturen gibt es eine Vielzahl an besonderen Datenstrukturen, die eine Art „Metalogik“ der enthaltenen Elemente darstellen. Es wird dabei in der Regel zu den Elementen eine übergeordnete Verfahrensweise definiert, beispielsweise wie Elemente in eine Struktur hineingespeichert und wieder entnommen werden. Dabei sind zwei dieser Datenstrukturen besonders wichtig:

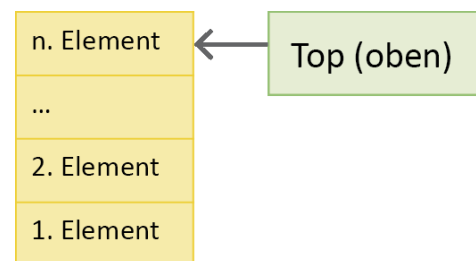
- ✓ Der Stack
- ✓ Die Queue

Manche Sprachen wie Java stellen eigene Klassen für diese Datenstrukturen bereit. Andere Sprachen lassen diese Datenstrukturen durch den Programmierer über die Anwendung der sequenziellen Grundstrukturen aufbauen (beispielsweise Python).

Beschreibung der Datenstruktur Stapel

Ein Stapel (engl. **stack**), auch Stapelspeicher oder **Keller** bzw. Kellerspeicher genannt, ist eine Datenstruktur, in der Daten nach dem LIFO-Prinzip verwaltet werden. LIFO steht für **Last In First Out** und bedeutet: Was zuletzt auf den Stack kommt (Last In), verlässt den Stack als Erstes (First Out).

Stellen Sie sich einen Spielkartenstapel vor. Wenn Sie eine Karte ziehen möchten, dürfen Sie nur die oberste Karte vom Stapel nehmen. Wollen Sie eine Karte ablegen, wird sie stets oben auf den Stapel gelegt. Die zuletzt auf den Stapel gelegte Karte (Last in) muss als Erste vom Stapel genommen werden (First Out).



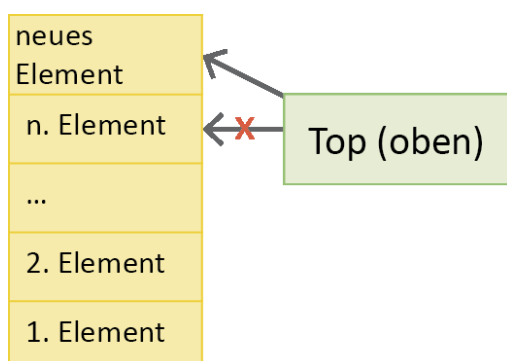
Stack mit n Elementen

Operationen für einen Stack

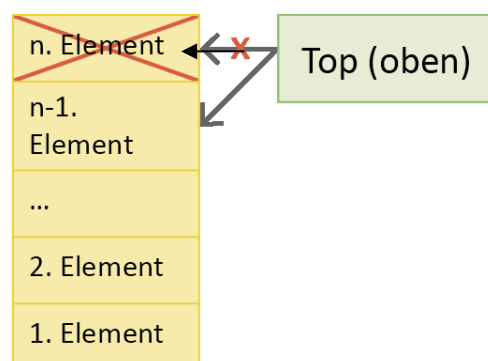
Um einen Stack umzusetzen, benötigt man in der Regel einige Operationen:

- ✓ Es muss sich ein neues Element oben in den Stack hinzufügen lassen – **push**.
- ✓ Es muss sich ein Element von oben entfernen lassen – **pop**.
- ✓ Man muss prüfen können, ob der Stack leer ist – **empty**: Ein Stapelunterlauf tritt ein, wenn versucht wird, aus einem Stack zu lesen, obwohl dieser leer ist. Deshalb sollte vor dem Löschen eines Elements überprüft werden, ob der Stack noch Elemente besitzt. Für Stacks, die eine feste maximale Größe besitzen, ist auch das Gegenteil möglich, der Stapelüberlauf.

Wichtig ist auch, dass **keine** davon abweichenden Operationen bereitgestellt oder zumindest nicht programmiert werden.



Push – Element in einen Stack einfügen



Pop – Element von einem Stack entfernen

Beispiel für die Erstellung eines Stapels mit einer Liste in Python: *Stack.py*

Unabhängig davon, dass viele Sprache eigene Klassen oder Strukturen für Stacks bereitstellen, ist ein Stack bloß ein Stapel mit Werten. Was zuerst auf den Stapel gelegt wurde, wird zuletzt wieder vom Stapel genommen. Die Listenmethoden machen es in Python sehr einfach, eine Liste als Stapel zu verwenden, wobei das letzte hinzugefügte Element wieder als erstes Element abgerufen wird. Das macht man so lange weiter, bis der Stapel leer ist.

Um ein Element zum Anfang des Stapels hinzuzufügen, verwenden Sie die Methode `append()`. Um ein Element von der Oberseite des Stapels abzurufen, verwenden Sie `pop()` ohne einen expliziten Index.

In dem folgenden Listing erzeugen wir einen Stapel mit 5 Elementen und bauen ihn Element für Element von oben wieder ab.

```
primzahlen = []
primzahlen.append(2)
primzahlen.append(3)
primzahlen.append(5)
primzahlen.append(7)
primzahlen.append(11)
print("Aktueller Stack nach dem Hinzufügen von 5 Elementen: ", primzahlen)
print("Oberstes Element vom Stack geholt: ", primzahlen.pop())
print("Aktueller Stack: ", primzahlen)
print("Oberstes Element vom Stack geholt: ", primzahlen.pop())
print("Aktueller Stack: ", primzahlen)
print("Oberstes Element vom Stack geholt: ", primzahlen.pop())
print("Aktueller Stack: ", primzahlen)
print("Oberstes Element vom Stack geholt: ", primzahlen.pop())
print("Aktueller Stack: ", primzahlen)
print("Oberstes Element vom Stack geholt: ", primzahlen.pop())
print("Aktueller Stack: ", primzahlen)
```

Das ist die Ausgabe des Beispiels:

```
Aktueller Stack nach dem Hinzufügen von 5 Elementen: [2, 3, 5, 7, 11]
Oberstes Element vom Stack geholt: 11
Aktueller Stack: [2, 3, 5, 7]
Oberstes Element vom Stack geholt: 7
Aktueller Stack: [2, 3, 5]
Oberstes Element vom Stack geholt: 5
Aktueller Stack: [2, 3]
Oberstes Element vom Stack geholt: 3
Aktueller Stack: [2]
Oberstes Element vom Stack geholt: 2
Aktueller Stack: []
```

Wo werden Stacks angewendet?

Stacks werden beispielsweise zur Programmierung folgender Anwendungen eingesetzt:

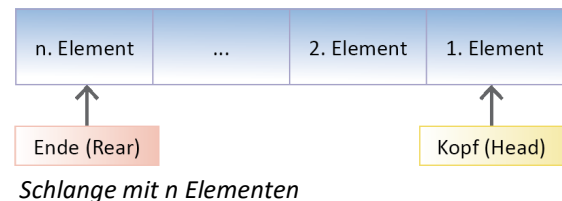
- ✓ Auswerten von geklammerten Ausdrücken
- ✓ Ausführen rekursiver Prozeduren
- ✓ Durchsuchen von Baumstrukturen (traversieren)

Beschreibung der Datenstruktur Schlange

Schlangen (engl. **queue**), sind wie Stapel Datenstrukturen mit eingeschränkten Zugriffsmöglichkeiten. Im Gegensatz zu Stacks arbeiten sie jedoch nach dem FIFO-Prinzip (**F**irst **I**n, **F**irst **O**ut), das heißt, dass das Element, welches als Erstes in die Schlange eingefügt wurde, sich somit am Kopf der Schlange befindet, auch als Erstes wieder ausgelesen wird. Neue Elemente werden stets am Ende der Queue angefügt.

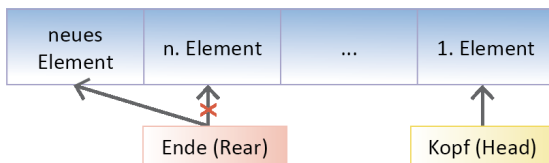
Das ist der gleiche Vorgang wie bei einer Warteschlange von Autos an einer roten Ampel. Hinzukommende Fahrzeuge reihen sich hinten ein, das Fahrzeug am Anfang der Schlange fährt als Erstes weiter.

Bei einer Queue werden nur das erste und das letzte Element betrachtet. Die Elemente werden genau in der Reihenfolge aus der Queue ausgelesen, in der sie eingefügt wurden.

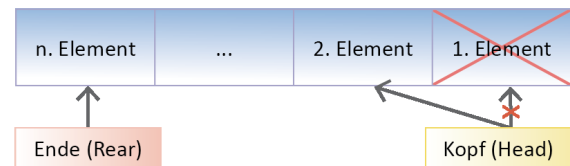


Operationen für eine Queue

- ✓ Neues Element hinzufügen – **enqueue**: Das neue Element wird am Ende der Queue eingefügt. Der Zeiger auf das Ende der Queue wird entsprechend angepasst.
- ✓ Element entfernen – **dequeue**: Das erste Element wird aus der Queue gelöscht. Der Zeiger auf den Kopf der Queue wird entsprechend angepasst.



Enqueue – ein Element einer Queue hinzufügen



Dequeue – ein Element aus einer Queue löschen

Beispiel für die Erstellung einer Queue mit einer Liste in Python: Queue.py

Zwar sind Listen für diesen Zweck nicht effizient, aber es ist auch möglich, mit Listenmethoden eine Warteschlange (Queue) zu erzeugen. Dabei wird das erste Element hinzugefügt und auch wieder als erstes Element abgerufen. Neue Elemente werden hinten an die Schlange einsortiert (FIFO-Prinzip: First In – First Out (englisch für der Reihe nach)).

Das nachfolgende Listing erzeugt eine Liste mit 5 Elementen und baut diese Element für Element von unten/vorne wieder ab. Die Funktion `len()` wird genutzt, um die Anzahl der Elemente in der Schlange zu überwachen und gezielt damit einen Index ansprechen, um mit `pop()` ein Element von unten zu entfernen.

```
queue = []
queue.append(2)
queue.append(3)
queue.append(5)
queue.append(7)
queue.append(11)
print("Queue nach Fertigstellung: ", queue)
while len(queue) > 0:
    print(queue.pop(0))
    print("Queue: ", queue)
```

Das ist die Ausgabe des Beispiels:

Queue nach Fertigstellung: [2, 3, 5, 7, 11]

2

Queue: [3, 5, 7, 11]

3

Queue: [5, 7, 11]

5

Queue: [7, 11]

7

Queue: [11]

11

Queue: []

Wo werden Queues angewendet?

Queues werden beispielsweise zur Programmierung folgender Anwendungen eingesetzt:

- ▶ Verwaltung von sequenziell abzuarbeitenden Aufgaben (Druckaufträge, Prozessverwaltung, Tastatureingaben usw.)
- ▶ Speicherverwaltung (Paging)

8.10 Übungen

Übung 1: Tupel und Listen

Übungsdatei: --

Ergebnisdateien: *uebung08.pdf, Searchnumbers.py*

1. Welchen Index hat das n-te Element eines Tupels oder einer Liste? Begründen Sie Ihre Aussage.
2. Schreiben Sie ein Python-Programm, um in einem Tupel `numbers = (0, 10, 12, 4, 7, 20, 21, 13)` nach einer Zahl zu suchen, die ein Anwender eingibt. Das Ergebnis der Suche soll mit der Position ausgegeben werden, an der sich die Zahl befindet. Falls die Zahl nicht vorhanden ist, soll auch dazu eine Meldung ausgegeben werden. Die Anzahl der Elemente in einem Tupel erhalten Sie in Python mit der Built-in-Function `len()`.

Übung 2: Tupel und Listen

Übungsdatei: --

Ergebnisdateien: *uebung08.pdf, FindPrimes.py*

1. Primzahlen sind Zahlen aus dem Zahlenbereich der natürlichen Zahlen größer oder gleich 2, die außer sich selbst und der Zahl 1 keine weiteren Teiler besitzen. Alle natürlichen Zahlen lassen sich in Faktoren von Primzahlen zerlegen. Erstellen Sie einen Algorithmus, der die Primzahlen von 2 bis n mithilfe des Siebs des Eratosthenes ermittelt. Bei dieser Methode werden alle Zahlen gestrichen, die sich aus anderen Zahlen zusammensetzen.
 - ✓ Schreiben Sie die Zahlen von 1 bis zur gewünschten Zahl der Größe nach geordnet in eine Liste.
 - ✓ Sie starten mit der kleinsten natürlichen Primzahl, der Zahl 2.
 - ✓ Setzen Sie alle Listeninhalte, die ein Vielfaches der Zahl 2 sind, auf 0.
 - ✓ Ist die gesamte Liste durchlaufen, wird von der Zahl 2 aus die nächstgrößere Zahl gesucht (3).
 - ✓ Alle Listeninhalte, die ein Vielfaches dieser Zahl sind, werden anschließend auf 0 gesetzt.

- ✓ Führen Sie diese Schritte so lange durch, bis das Quadrat der aktuellen Zahl größer als die letzte Zahl der Liste ist.
- ✓ In allen Listenelementen größer 1, deren Inhalt nicht auf 0 gesetzt wurde, befinden sich die gesuchten Primzahlen.

Siebzahl 2: alle Vielfache von 2 entfallen

alt	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...
neu	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...

Siebzahl 3: zusätzlich entfallen alle Vielfache von 3

alt	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...
neu	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...

usw.

2. Setzen Sie den Algorithmus aus Aufgabe 1 in Python-Code um. Verwenden Sie zum Einlesen der Zahl, bis zu der die Primzahlen ermittelt werden sollen, die Funktion `input()`. Nutzen Sie die `append()`-Methode zum Aufbau der Liste. Speichern Sie das Programm unter *FindPrimes.py*.

Übung 3: Listen

Übungsdatei: --

Ergebnisdateien: **Warenkorb1.py**

1. Erstellen Sie auf Basis einer Liste ein Python-Programm für einen Warenkorb.
2. Die Anwendung soll in einer Schleife ein Produkt als freie Texteingabe angeben (mit der `input()`-Funktion).
3. Das Produkt wird mit der `append()`-Methode der Liste (dem Warenkorb) hinzugefügt.
4. Lassen Sie den Anwender entscheiden, wie daran weiter vorgegangen wird:
 - ✓ weiter einkaufen
 - ✓ bestellen oder
 - ✓ die Bestellung abbrechen

Übung 4: Dictionaries

Übungsdatei: --

Ergebnisdateien: **Warenkorb2.py**

1. Erstellen Sie auf Basis eines Dictionaries ein Python-Programm für einen Warenkorb.
2. Die Anwendung soll in einer Schleife ein Produkt als freie Texteingabe angeben (mit der `input()`-Funktion).
3. Dem Produkt soll eine Anzahl zugeordnet werden, die ebenso als freie Texteingabe mit der `input()`-Funktion eingegeben wird.
4. Produkt und Anzahl werden mit der `update()`-Methode dem Dictionary (dem Warenkorb) hinzugefügt.
5. Lassen Sie den Anwender entscheiden, wie daran weiter vorgegangen wird:
 - ✓ weiter einkaufen
 - ✓ bestellen oder
 - ✓ die Bestellung abbrechen
6. Geben Sie zu jeder Entscheidung des Anwenders eine passende Meldung in der Konsole aus.

9 Methoden, Prozeduren und Funktionen

In diesem Kapitel erfahren Sie

- ✓ wie Sie Methoden, Prozeduren und Funktionen schreiben
- ✓ welche Möglichkeiten der Parameterübergabe existieren
- ✓ wie Sie Rückgabewerte verwenden
- ✓ was Ihnen Standardbibliotheken und automatisch vorhandene Methoden, Prozeduren und Funktionen bieten

Voraussetzungen

- ✓ Deklarieren von Variablen und Konstanten
- ✓ Verwenden von Datentypen
- ✓ Zeiger und Arrays

9.1 Unterprogramme

Warum Methoden, Prozeduren und Funktionen verwendet werden

Code mehrfach nutzen

Häufig werden in einem Programm bestimmte Anweisungen mehrmals benötigt. Dazu werden Anweisungen, die eine bestimmte Funktion erfüllen, zusammengefasst und erhalten einen Namen. Diese funktionalen Einheiten werden bei objektorientierten Programmiersprachen **Methoden**, in prozeduralen Sprachen **Funktionen** oder **Prozeduren** genannt. Diese funktionalen Einheiten werden auch mit **Unterprogramm** oder **Subroutine** bezeichnet. Unterprogramme können beliebig oft in einem Programm verwendet werden, indem sie über ihren Namen aufgerufen werden (das DRY-Konzept). Die Unterprogramme können über Bibliotheken auch für andere Programme zur Verfügung gestellt werden (**Mehrfachverwendbarkeit**). Unterprogramme aus Bibliotheken sind bereits getestet. Somit wird durch die Verwendung dieser Unterprogramme die Korrektheit von Programmen erhöht, da sie meist weniger Fehler enthalten als entsprechend neu erstellte Programme.

Code strukturieren

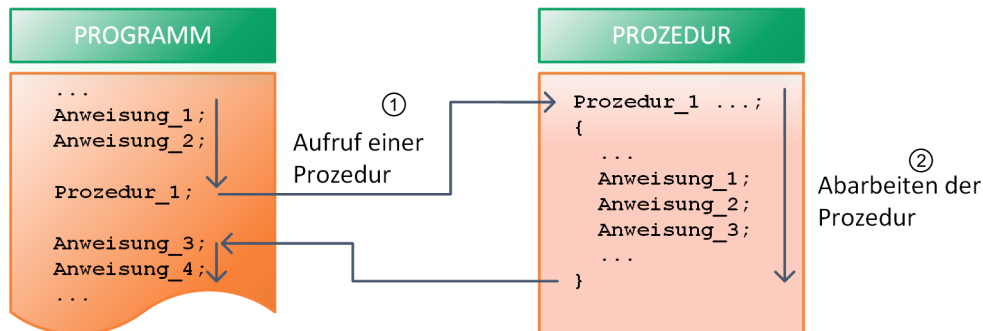
Unterprogramme sind gemäß dem Modularitätsprinzip überschaubare Einheiten eines Programms. Ohne diese Strukturierung könnten große Programme nicht verstanden und gewartet werden.

Mit Prozeduren arbeiten

Unterprogramme, die in der prozeduralen Programmierung **keinen Rückgabewert** liefern, heißen **Prozeduren**, wobei dieser Ausdruck eher in der Vergangenheit genutzt wurde und heutzutage nur noch selten explizit zum Einsatz kommt.

Soll eine Prozedur bei jedem Aufruf mit anderen Daten arbeiten, benötigt eine Prozedur **Parameter**. Parameter, die eine Prozedur zur Verfügung stellt, heißen **formale Parameter**. Beim Aufruf einer Prozedur über den Prozedurnamen können dann entsprechende Werte an die formalen Parameter übergeben werden. Die beim Prozeduraufruf übergebenen Werte werden **aktuelle Parameter** genannt. Soll eine Prozedur z. B. verschiedene Texte ausgeben, kann der Text als Parameter übergeben werden.

Wird beim Programmablauf ein Prozeduraufruf ① erreicht, kommt es zur Unterbrechung des Programmflusses an dieser Stelle, und es wird in die zugehörige Prozedur verzweigt. Die Anweisungen der Prozedur werden ausgeführt ②, anschließend wird die dem Prozeduraufruf folgende Anweisung, im Beispiel Anweisung_3, ausgeführt.



Aufruf und Abarbeiten einer Prozedur

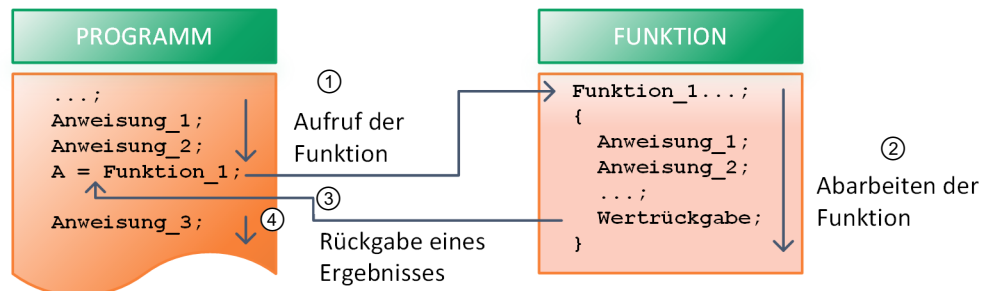


Die meisten Programmiersprachen bieten Standardprozeduren. Sie erledigen häufig benötigte Standardaufgaben, z. B. Prozeduren für Ausgaben auf den Bildschirm.

Mit Funktionen arbeiten

Funktionen liefern in der prozeduralen Programmierung im Unterschied zu Prozeduren **einen Rückgabewert** an das aufrufende Programm zurück. Dadurch können Sie Funktionen auch in Ausdrücken angeben.

Funktionen werden, analog zu Prozeduren, mit dem Funktionsnamen aufgerufen. Auch Funktionen können Sie bei Bedarf Parameter übergeben. Wird eine Funktion aufgerufen ①, werden die Anweisungen der Funktion abgearbeitet ②, und ein Rückgabewert ③ wird an das aufrufende Programm übergeben. Dieser Rückgabewert kann danach z. B. einer Variablen zugewiesen oder in einer Berechnungsformel verwendet werden. Anschließend wird die dem Funktionsaufruf folgende Anweisung ④, im Beispiel Anweisung_3, ausgeführt.



Aufruf und Abarbeiten einer Funktion



Heutzutage fasst man Prozeduren und Funktionen fast immer unter dem Begriff Funktion zusammen.

Aufbau von Prozeduren und Funktionen

Prozeduren und Funktionen bestehen aus einem Kopf und einem Rumpf.

- ✓ Im Kopf wird der Name der Prozedur bzw. Funktion zusammen mit der optionalen Parameterliste festgelegt. Der Kopf mit der Parameterliste ist die **Schnittstelle** der Prozedur bzw. Funktion. Die Schnittstelle, auch **Signatur** genannt, legt fest, wie die Prozedur bzw. Funktion benutzt werden kann.
- ✓ Der Rumpf enthält die einzelnen Anweisungen der Prozedur bzw. Funktion.
- ✓ Prozeduren und Funktionen können in ihrem Rumpf ebenfalls Prozeduren oder Funktionen aufrufen.

Wichtige Unterschiede zwischen Prozeduren und Funktionen

Aktion	Funktion	Prozedur
Ergebnisrückgabe	ja	nein
Verwendung in einer Wertzuweisung	ja	nein
Verwendung in einem Ausdruck	ja	nein
Verwendung als Vergleichsoperator in Bedingungsausdrücken	ja	nein

Syntax für die Beschreibung einer einfachen Funktion in Python

- ✓ Die Beschreibung einer Funktion erfolgt durch den Funktionskopf ① und den Funktionsrumpf ②.

```
def functionsname(Parameterliste): ①
    ...                             ②
```

Die Deklaration einer eigenen Funktion beginnt in Python mit dem Schlüsselwort `def`. Dem Schlüsselwort folgen der Name der Funktion und runde Klammern, in die bei Bedarf Parameter eingefügt werden. Das ist in Python ganz so wie in nahezu allen Programmiersprachen, die Funktionen oder Methoden deklarieren lassen. Die Besonderheit in Python ist wieder der nun folgende Doppelpunkt, mit dem der Anweisungsblock eingeleitet wird. Dieser wird lediglich eingerückt und **nicht** – wie in vielen anderen Sprachen – durch besondere Bezeichner oder Zeichen gekennzeichnet.

Mit Methoden arbeiten

Unterprogramme, die in der objektorientierten Programmierung verwendet werden, sind an Objekte oder Klassen gebunden. Um sie deutlich von prozeduralen Unterprogrammen abzugrenzen, nennt man sie **Methoden**. Dabei wird nicht zwischen Methoden, die keinen Rückgabewert liefern, und solchen, die einen Rückgabewert liefern, unterschieden. Auch an Methoden können Sie bei Bedarf Parameter übergeben.

Gelegentlich taucht auch bei der objektorientierten Programmierung der Begriff der Funktion oder Prozedur auf, obwohl eine Methode gemeint ist. Mit der nötigen Vorsicht kann man das machen, obgleich es eine etwas un-saubere Formulierung darstellt. Sie sollten bei der objektorientierten Programmierung die korrekte Bezeichnung Methode verwenden.



Syntax für die Beschreibung einer einfachen Methode in Python

- ✓ Die Beschreibung einer Methode erfolgt durch den Methodenkopf ① und den Methodenrumpf ②.

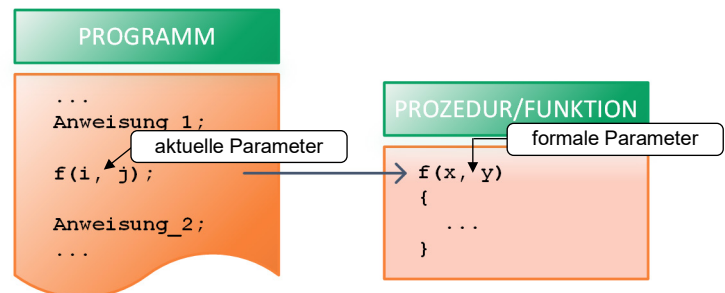
```
def functionsname(self, Parameterliste): ①
{
    ...                                 ②
}
```

Außer der Tatsache, dass sich Methoden in Python innerhalb einer Klasse befinden, unterscheiden sich die Deklaration von Funktionen und Methoden in Python nur in einer Sache. Das ist der zwingend notwendige Parameter `self`. Dieser muss in Python in jeder Deklaration einer Instanzmethode als erster Parameter notiert werden und verweist auf das aktuelle Objekt. Beim Aufruf wird dieser Parameter dann aber automatisch gefüllt. Dafür darf kein Wert angegeben werden. Das bedeutet, dass beim Aufruf einer Instanzmethode der erste übergebene Wert immer für den zweiten (!) Parameter steht.

9.2 Parameterübergabe

Aktuelle und formale Parameter

Oft ist es notwendig, Werte an Unterprogramme zu übergeben, damit die Methode, Prozedur oder Funktion mit den aktuellen Werten des aufrufenden Programms arbeiten kann. Diese Werte werden **aktuelle Parameter** oder auch **Argumente** genannt.



Bei der Definition der Methode, Prozedur oder Funktion wird in vielen Programmiersprachen genau festgelegt, welche Parameter für die Berechnung benötigt werden. Diese Parameter werden auch **formale Parameter** genannt. Die Liste mit allen formalen Parametern wird rechts vom Namen des Unterprogramms angegeben.

Bedingungen für die Übergabe der aktuellen Parameter

Beim Aufruf des Unterprogramms mit den aktuellen Parametern müssen Sie beachten, dass in vielen Sprachen aktuelle und formale Parameter in den folgenden Punkten genau übereinstimmen. In anderen Sprachen wie JavaScript oder PHP muss das allerdings nicht sein.

Anzahl der Parameter	Die Anzahl der aktuellen Parameter muss der Anzahl der formalen Parameter entsprechen, wenn die Sprache (etwa Java) das fordert. Bei weniger strengen Sprachen wie JavaScript oder PHP muss das nicht sein.
Reihenfolge der Parameter	Der erste aktuelle Parameter wird an den ersten formalen Parameter übergeben, der zweite aktuelle Parameter an den zweiten formalen Parameter usw.
Datentypen der Parameter	Es hängt an der Art der Sprache, ob die Datentypen der Parameter von Bedeutung sind. In Sprachen ohne strenge Typbindung (JavaScript, PHP etc.) ist der Datentyp vollkommen irrelevant. In strengen Sprachen wie Java hingegen ist der Datentyp wichtig. Der Datentyp des ersten aktuellen Parameters muss dort mit dem Datentyp des ersten formalen Parameters übereinstimmen, der Datentyp des zweiten aktuellen Parameters mit dem Datentyp des zweiten formalen Parameters usw. Jeder Parameter kann einen anderen Datentyp haben.

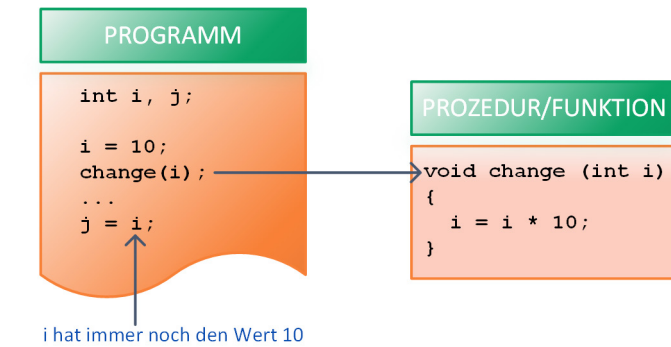


- ✓ Bei Entwurf von Programmen mit PAP, Struktogramm oder Pseudocode finden meist nur die ersten beiden Punkte Beachtung. Der letzte Punkt – die Übereinstimmung der Datentypen – ist bei der Umsetzung in die Programmiersprache zu beachten.
- ✓ Im Pseudocode können Prozeduren beispielsweise durch das Wort `procedure` gekennzeichnet werden, dem der Name der Prozedur folgt. Nach dem Prozedurnamen kann in Klammern eine Liste mit Parametern angegeben werden, die 1 bis n Parameter, durch Komma getrennt, enthalten kann. Bei Funktionen wird entsprechend das Wort `function` zur Kennzeichnung verwendet.

9.3 Parameterübergabe als Wert

Einen Parameter als Wert übergeben

Der Wert (engl. value) einer übergebenen Variablen kann in dem Unterprogramm bei der Parameterübergabe als Wert, auch **call by value** genannt, nicht verändert werden. Bei der Parameterübergabe wird nur eine **Kopie** des Wertes an den formalen Parameter des Unterprogramms übergeben.



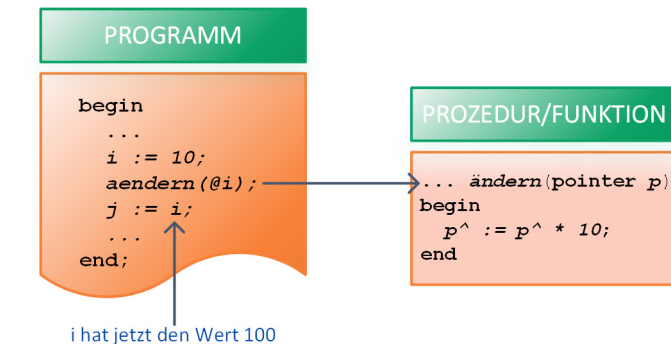
Parameterübergabe als Wert

9.4 Parameterübergabe über Referenzen

Einen Parameter als Referenz übergeben

Bisher wurden die aktuellen Parameter an ein Unterprogramm nur als Kopien der Werte der aktuellen Variablen übergeben. Die Änderung des Wertes eines Parameters innerhalb des Unterprogramms hatte keine Auswirkung auf den Wert der Variablen, die im aufrufenden Programm als aktueller Parameter übergeben wurde.

Die Parameterübergabe über Referenzen, auch **Call-by-Reference** genannt, stellt eine Möglichkeit dar, Werte der Variablen, die als aktuelle Parameter übergeben werden, innerhalb der Methode, Prozedur bzw. Funktion zu ändern, sodass sie auch nach dem Verlassen des Unterprogramms den neuen Wert behalten. Beim Aufruf des Unterprogramms wird nun statt des Wertes die Adresse (Referenz) der Variablen übergeben, die diesen Wert beinhaltet. Diese verweist dann auf dieselbe Speicherstelle wie die Variable des aufrufenden Programms. Ändern Sie den Wert der Variablen nun in der Methode, Prozedur bzw. Funktion, wird die Änderung auch im aufrufenden Programm sichtbar.



Parameterübergabe als Referenz (Pseudocode)

Die Möglichkeiten der Parameterübergabe und deren Darstellung sind in den einzelnen Sprachen recht unterschiedlich. In Sprachen, in denen Sie mit Zeigern arbeiten können, ist eine Unterscheidung der Übergabeart möglich, z. B. in Pascal oder C++. Java stellt keine Parameterübergabe über Referenzen zur Verfügung. In Python gilt von der Wirkung Call-by-Value. Es wird also eine Kopie übergeben, und Änderungen in der Funktion verändern nicht die globale Variable. Nur ist das nicht ganz so trivial, denn Python benutzt einen Mechanismus, den man als Call-by-Object, Call-by-Object-Reference oder auch Call-by-Sharing bezeichnet. Beim Aufruf einer Funktion werden bei Python nicht Zeiger auf Variablen, sondern Zeiger auf die zugrunde liegenden Objekte übergeben. Insofern könnte man von einer Art Referenzübergabe sprechen. Zuerst verhält sich Python wie bei Call-by-Reference, in dem Ergebnis jedoch wie Call-by-Value, weil das Objekt ausgewertet wird. Der Vorteil ist, dass man beliebige Ausdrücke übergeben kann.



9.5 Rückgabewerte von Funktionen oder Methoden

Die Rückgabewerte von Funktionen oder Methoden können z. B. primitive Datentypen oder Referenzen sein. Der Rückgabewert wird mit dem Schlüsselwort `return` zurückgegeben. Das ist in der Regel die letzte Anweisung im Funktionsblock, sofern das nicht bedingt gemacht wird.

Syntax für Methoden mit Rückgabewert in Java

```
def functionsname(Parameterliste):
    ...Anweisungen
    return Ergebnis
```



In streng typisierten Sprachen geben Sie bei der Deklaration an, ob es einen Rückgabewert gibt und welchen Typ des Rückgabewertes ein Unterprogramm liefert.

Beispiel für einen primitiven Datentyp als Rückgabewert: *FindMax.py*

Im folgenden Programm soll der größere von zwei Variablenwerten, die als Parameter übergeben werden, ermittelt werden. Der Vergleich wird von einer Funktion ausgeführt. Als Ergebnis ist der größere der beiden Werte zurückzugeben.

```
③ def returnMax(a, b):
④     if (a > b):
⑤         return a
⑤     else:
⑤         return b
⑥ n1 = 1234;
⑥ n2 = 5678;
⑥ max = 0;
① def main():
②     max = returnMax(n1, n2)
②     print("Maximum: ", max)
⑦ main()
```

- ① Das Programm arbeitet mit einer Wurzelfunktion `main()`. Aus dieser entwickelt sich der gesamte Programmablauf. Hier steht die Funktionssignatur, mit der die Funktion beginnt.
- ② Innerhalb des Funktionsrumpfs von `main()` wird die Methode `returnMax()` aufgerufen, um den größeren der beiden Werte zu ermitteln. Als Parameter werden die beiden Variablen `n1` und `n2` übergeben.
- ③ Die Methode `returnMax()` ist so definiert, dass sie mit zwei Parametern aufgerufen werden muss.
- ④ Die Werte der beiden Parametervariablen werden verglichen.
- ⑤ Ist der Wert der Variablen `a` größer, wird `a` als Rückgabewert an das Hauptprogramm geliefert, sonst `b`.
- ⑥ Die Variablen `n1` und `n2` sowie `max` werden initialisiert.
- ⑦ Die Wurzelfunktion `main()` wird aufgerufen und nun das Maximum ausgegeben. Die Ausgabe lautet: *Maximum: 5678*.

9.6 Innere Funktionen – Closures

Manche Programmiersprachen erlauben den Aufbau verschachtelter Funktionen. Dabei wird eine Funktionsdeklaration direkt in eine andere Funktion hinein notiert – so auch Python. In dem Fall gelten einige spezielle Regeln (hier speziell für Python):

- ✓ Die innere verschachtelte Funktion kann nur über die äußere aufgerufen werden und ist für direkte Zugriffe unzugänglich.
- ✓ Die innere Funktion kann die Variablen oder andere innere Funktionen der äußeren Funktion verwenden.
- ✓ Die äußere Funktion hat keinen Zugang zu den lokalen Variablen der inneren Funktion.

Zur Laufzeit erzeugt der Python-Interpreter beim Aufruf der äußeren Funktion den Code der inneren Funktion. Dabei entsteht ein sogenanntes Closure (Einschluss) der inneren Funktion – das ist der Code der Funktion und eine Referenz auf alle Variablen, die von der inneren Funktion benötigt werden. Ein Closure kombiniert den Programmcode mit der lexikalischen Umgebung – d. h., das Closure „merkt“ sich die Umgebung, in der es erzeugt wurde.

Beispiel für ein Closure: *MyClosure.py*

```
① def outside():  
④     t = "Purple"  
②     def inside():  
⑦         print("Deep", t)  
⑤     inside()  
outside()
```

- ① Die Deklaration der äußeren Funktion `outside()` beginnt.
- ② Innerhalb des Funktionsrumpfs der äußeren Funktion wird die Deklaration der inneren Methode `inside()` notiert.
- ⑦ Die Funktion `print()` in der inneren Funktion `inside()` nutzt die Variable `t`, die in der äußeren Funktion `outside()` deklariert wurde.
- ④ Die Deklaration der Variablen `t` in der äußeren Funktion `outside()`
- ⑤ Die innere Funktion `inside()` wird in der äußeren Funktion `outside()` aufgerufen.

Mit Closures kann man in Python auch rein funktional das Konzept der Datenkapselung unterstützen, das im Rahmen der objektorientierten Programmierung zu den Grundprinzipien zählt.



9.7 Standardbibliotheken und Built-in-Funktionalitäten

Neben der Syntax beinhalten fast alle Programmiersprachen meist eine Reihe an „eingebauten“ Funktionalitäten. Erst darüber kann man entweder bestimmte Dinge überhaupt machen (etwa Benutzerdaten entgegennehmen, eine Ausgabe vornehmen, eine Datei schreiben oder lesen, etc.) oder aber viele Schritte vereinfachen, weil man das Rad nicht immer neu erfinden muss.

Python besitzt beispielsweise einige vorinstallierte Funktionen (**Built-in-Functions**), die automatisch in Python bereitstehen und überall im Quellcode verwendet werden können. Mit anderen Worten: Sie müssen diese vor einer Verwendung nicht selbst deklarieren und auch sonst keine besonderen Vorbereitungen vor der Verwendung treffen.

Eine der wichtigsten Python-Funktionen beziehungsweise -Anweisungen ist sicher `print()`. Damit generiert man eine Ausgabe in der Konsole und kann den letzten Teil des EVA-Prinzips umsetzen.

Um das EVA-Prinzip vollständig umzusetzen, brauchen Sie in Python aber natürlich noch eine Eingabemöglichkeit. Und dazu kann man eine weitere Built-in-Funktion anwenden, die sich `input()` nennt.

Beispiel für die Eingabe einer UserID und eines Passworts und Ausgabe eines entsprechenden Ergebnisses: *BuiltInFunctions.py*

- ✓ Das nachfolgende Beispiel verwendet ein verschachteltes Tupel mit einigen Kombinationen aus Benutzernamen und einem zugehörigen Passwort.
- ✓ Der Anwender kann unter zweifacher Verwendung der Built-in-Funktion `input()` eine solche Kombination eingeben.
- ✓ Ist die Eingabekombination korrekt, wird eine entsprechende Meldung ausgegeben.

```
User:
hans
Passwort:
geheim
Vorhanden
>>>
```

Die Eingabekombination ist korrekt, und es wird eine entsprechende Meldung ausgegeben.

- ✓ Ist die eingegebene Kombination falsch, wird eine andere Meldung angezeigt.
- ✓ Im Fehlerfall kann der Anwender unter Verwendung der Built-in-Funktion `input()` für die Eingabe einer Kennung entweder einen neuen Versuch starten oder die Eingabe abbrechen.

```
User:
otto
Passwort:
geheim
Falsch
Vorgang abbrechen (J)|
```

Die Eingabekombination ist falsch, und es wird eine entsprechende Meldung ausgegeben.

```
User:
otto
Passwort:
geheim
Falsch
Vorgang abbrechen (J)
User:
hans
Passwort:
geheim
Vorhanden
>>>
```

Die Eingabekombination war im ersten Versuch falsch und im zweiten Versuch dann korrekt.


```
User:
otto
Passwort:
geheim
Falsch
Vorgang abbrechen (J) J
Vorgang wird abgebrochen
>>> |
```

Der Vorgang wurde nach einer falschen Eingabekombination abgebrochen.

- ✓ Nach maximal drei falschen Versuchen wird der Vorgang beendet und eine entsprechende Meldung ausgegeben.

```
User:
otto
Passwort:
geheim
Falsch
Vorgang abbrechen (J)
User:
otto
Passwort:
geheim
Falsch
Vorgang abbrechen (J)
User:
otto
Passwort:
Geheim
Falsch
Nach 3 falschen Versuchen wird der Account gesperrt
>>> |
```

Nach einer zu großen Anzahl an Fehlversuchen wird der Vorgang abgebrochen.

```
① zugangsliste =
  (("hans", "geheim"), ("otto", "passwort"), ("fred", "feuerstein"))
  i=0
② def zugang():
⑦     global i
        global zugangsliste
        while i < 3:
            user = input("User:\n")
            pw = input("Passwort:\n")
            vorhanden = False
            p = (user, pw)
④         for v in zugangsliste:
                if v == p:
                    vorhanden = True
```

```

⑤      i+=1
      if vorhanden :
          print("Vorhanden")
          break
      else:
          print("Falsch")

          if i > 2:
              print("Nach", i,
                    "falschen Versuchen wird der Account gesperrt")
              break
          if input("Vorgang abbrechen (J) ")=="J":
              print("Vorgang wird abgebrochen")
              break

zugang()

```

- ① Es werden globale Variablen deklariert, einmal ein Tupel `zugangsliste` mit den Kombinationen aus UserID und Passwort und eine Zählvariable `i`, um die Anzahl der Versuche zu protokollieren.
- ② Die Deklaration der Funktion beginnt.
- ⑦ Die globalen Variablen werden in der Funktion mit dem Schlüsselwort `global` bekannt gegeben. In der Schleife werden unter Verwendung der Built-in-Funktion `input()` die UserID und das Passwort entgegengenommen und in Variablen gespeichert. Zudem wird eine boolesche Variable `vorhanden` angelegt. Darüber wird in der Folge entschieden, wie der Programmfluss weiter verläuft. Aus der UserID und einem Passwort wird ein neues Tupel `p` erstellt.
- ④ Mit dem Membership-Operator `in` wird überprüft, ob das Tupel `p` in dem Tupel `zugangsliste` vorhanden ist.
- ⑤ Wenn eine Kombination vorhanden ist, wird eine entsprechende Meldung angezeigt und das Programm beendet. Ist die Kombination falsch, kann der Anwender mit der Eingabe von `J` das Programm abbrechen oder einen neuen Versuch starten. Nach maximal 3 Versuchen wird das Programm abgebrochen.

Built-in-Functions von Python

Nachfolgend finden Sie alphabetisch sortiert eine Tabelle aller Built-in-Functions von Python 3.6.x. Für Details sei auf die Dokumentation unter <https://docs.python.org/2/library/functions.html> verwiesen.

<code>abs()</code>	<code>divmod()</code>	<code>input()</code>	<code>open()</code>	<code>staticmethod()</code>
<code>all()</code>	<code>enumerate()</code>	<code>int()</code>	<code>ord()</code>	<code>str()</code>
<code>any()</code>	<code>eval()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>sum()</code>
<code>basestring()</code>	<code>execfile()</code>	<code>issubclass()</code>	<code>print()</code>	<code>super()</code>
<code>bin()</code>	<code>file()</code>	<code>iter()</code>	<code>property()</code>	<code>tuple()</code>
<code>bool()</code>	<code>filter()</code>	<code>len()</code>	<code>range()</code>	<code>type()</code>
<code>bytearray()</code>	<code>float()</code>	<code>list()</code>	<code>raw_input()</code>	<code>unichr()</code>
<code>callable()</code>	<code>format()</code>	<code>locals()</code>	<code>reduce()</code>	<code>unicode()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>long()</code>	<code>reload()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>map()</code>	<code>repr()</code>	<code>xrange()</code>
<code>cmp()</code>	<code>globals()</code>	<code>max()</code>	<code>reversed()</code>	<code>zip()</code>
<code>compile()</code>	<code>hasattr()</code>	<code>memoryview()</code>	<code>round()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hash()</code>	<code>min()</code>	<code>set()</code>	
<code>delattr()</code>	<code>help()</code>	<code>next()</code>	<code>setattr()</code>	
<code>dict()</code>	<code>hex()</code>	<code>object()</code>	<code>slice()</code>	
<code>dir()</code>	<code>id()</code>	<code>oct()</code>	<code>sorted()</code>	

Die Built-in-Functions in Python

In ähnlicher Form stellen auch andere Sprachen Funktionen oder Prozeduren oder aber über Standardklassen Methoden und Eigenschaften bereit. Diese sind dann Teil der **Standardbibliothek** der Sprache.

Darüber hinaus gibt es – vor allen Dingen in objektorientierten Sprachen – oft auch Funktionalitäten, die nicht automatisch im Quelltext zur Verfügung stehen, aber „hinzugebunden“ bzw. importiert werden können. Auch solche Funktionalität zählt man zur Standardbibliothek, wenn sie mit der Sprache selbst ausgeliefert wird.

In Python kann Quellcode beispielweise in verschiedene **Module** verteilt werden. Das sind einfach .py-Dateien mit Klassendefinitionen, Variablen oder Funktionen. In Dateien, wo diese Module verwendet werden sollen, notiert man eine `import`-Anweisung (ggf. in Kombination mit einer `from`-Anweisung, um den Zugriff auf Elemente zu verkürzen) und gibt den Namen des Moduls (der Dateiname ohne Erweiterung) an. Der Zugriff erfolgt dann über den Namen und die Punktnotation. Für verschiedene praktische Beispiele vgl. Kapitel 11.



Beispiel die Erstellung einer Zufallszahl unter Verwendung eines Moduls: *Random.py*

Das folgende Python-Programm generiert eine Zufallszahl im Bereich von 1 bis 49 und gibt diese anschließend mit `print()` aus.

①	<code>import random</code>
②	<code>random.seed()</code>
⑦	<code>x = random.randint(1,49)</code>
④	<code>print(x)</code>

- ① Zum Generieren einer zufälligen Zahl in Python wird das Modul `random` verwendet. Dieses wird mit der `import`-Anweisung eingebunden.
- ② Dann wird der Zufallsgenerator mit `random.seed()` initialisiert.
- ③ Die Funktion `randint()` wird mit zwei Parametern aufgerufen und liefert eine zufällige Ganzzahl aus dem durch die Parameter definierten Bereich.
- ④ Der Zufallswert wird ausgegeben.

9.8 Übungen

Übung 1: Funktionen, Prozeduren und Methoden ohne Rückgabewert

Übungsdatei: --

Ergebnisdateien: *uebung09.pdf, MoveRight.py*

1. Schreiben Sie eine Prozedur in Pseudocode, die alle Elemente eines Feldes um jeweils eine Position nach rechts verschiebt. Das letzte Element soll das erste Element werden.
2. Setzen Sie den Pseudocode aus Aufgabe 1 in Python-Code um, wobei Sie eine Funktion ohne Rückgabewert erstellen.
3. Erstellen Sie eine Prozedur in Pseudocode, der als Parameter der Radius einer Kugel übergeben wird. In der Prozedur sind der Umfang der Kugel, die Mantelfläche und das Volumen zu berechnen. Die Werte sollen anschließend dem aufrufenden Programm zur Verfügung stehen.

Umfang einer Kugel (Kreis): $2 \cdot \pi \cdot R$ π entspricht der Konstanten $\pi = 3.14$

Mantelfläche der Kugel: $4 \cdot \pi \cdot R^2$ R entspricht dem Radius

Volumen einer Kugel: $4 / 3 \cdot \pi \cdot R^3$

Übung 2: Funktionen und Methoden mit Rückgabewert

Übungsdatei: --

Ergebnisdateien: *uebung09.pdf, FindMin.py, UsePower.py*

1. Schreiben Sie eine Funktion in Pseudocode, die das Minimum dreier verschiedener Werte bestimmt und an das Hauptprogramm zurückgibt.
2. Lösen Sie die Aufgabe 1 mit einem Python-Programm. Speichern Sie das Programm unter *FindMin.py*.
3. Erstellen Sie eine Funktion in Pseudocode, die die n-te Potenz einer Zahl x berechnet ($y = x^n$). Die Zahlen x und n sollen der Funktion als Parameter übergeben werden, wobei x und n natürliche Zahlen (0, 1, 2, 3 ...) sind. Das Ergebnis soll im Hauptprogramm ausgegeben werden.
4. Erstellen Sie für die Aufgabe 3 ein Python-Programm unter Verwendung passender Methoden. Die Methode soll mit Eingaben des Anwenders für die Basis und den Exponenten getestet werden. Speichern Sie das Programm unter *UsePower.py*.

10 Algorithmen

In diesem Kapitel erfahren Sie

- ✓ welche verschiedenen Arten von Algorithmen es gibt
- ✓ den Unterschied zwischen einem iterativen und rekursiven Algorithmus

Voraussetzungen

- ✓ Methoden, Prozeduren und Funktionen

10.1 Eigenschaften eines Algorithmus

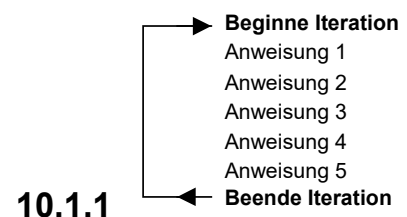
Der Begriff des Algorithmus ist zentral in der Programmierung und er wurde bereits von Anfang an in einer einfachen Form verwendet. In dem Kapitel betrachten wir seine exakte Bedeutung in der Informatik genauer. Damit ein Algorithmus zur Lösung einer Aufgabe in Programmen eingesetzt werden kann, muss er bestimmte Eigenschaften besitzen.

endlich	Der Algorithmus muss vollständig mit einer endlichen Anzahl von Anweisungen beschrieben werden können.
allgemein	Der Algorithmus muss für die gleichen Eingabedaten immer das gleiche Ergebnis liefern.
eindeutig	Jeder Schritt des Algorithmus muss eindeutig ausgeführt werden können.
terminiert	Der Algorithmus muss nach einer endlichen Anzahl von Schritten zum Ende kommen.

Algorithmen können nach verschiedenen Kriterien unterschieden werden, z. B. nach der Art der Abarbeitung oder dem Umgang mit Datentypen.

10.2 Iterativer Algorithmus

Ein **iterativer Algorithmus** löst eine Aufgabe mit Schleifendurchläufen.



Beispiel für einen iterativen Algorithmus: *Factorial.py*

Die Fakultät einer Zahl x , geschrieben $x!$, ist das Produkt aller ganzen positiven Zahlen von 1 bis x . Die Fakultät von 3 berechnet sich beispielsweise wie folgt: $3! = 1 * 2 * 3 = 6$.

```

① def compFact(x):
②     result = 1
③     while (x > 1):
④         result = result * x
⑤         x = x - 1
⑥     return result
print(compFact(4))

```

- ① Die Methode besitzt den Namen `compFact`. Der formale Parameter ist `x`.
- ② Die Variable `result` (Ergebnis) wird deklariert und gleich mit dem Wert 1 initialisiert.
- ③ Die Schleife wird so lange durchlaufen, bis die Variable `x` kleiner oder gleich 1 ist. Die letzte Multiplikation (mit dem Wert 1) kann entfallen, da eine Multiplikation mit 1 das Ergebnis nicht verändert.
- ④ Die Berechnung der Fakultät wird durchgeführt. Das bisherige Resultat in der Variablen `result` wird mit der Variablen `x` multipliziert.
- ⑤ Die Variable `x` wird um den Wert 1 verringert (dekrementiert).
- ⑥ Das Endergebnis `result` wird mit der `return`-Anweisung zurückgegeben.

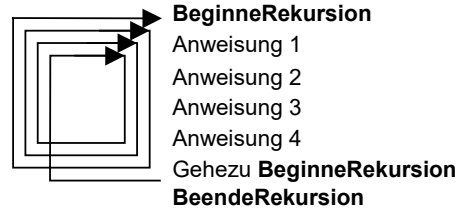
Die Iteration Schritt für Schritt

Zur Verdeutlichung des iterativen Algorithmus werden die Variablenwerte bei den einzelnen Schleifendurchläufen für den aktuellen Parameterwert 4 betrachtet.

Schleifendurchlauf	Programmschritte	Erklärung
	<code>compFact(4);</code>	Methodenaufruf
	<code>result = 1;</code>	Initialisierung der Variablen
1. Iteration	$4 > 1 ?$ $result = 1 * 4 = 4$ $x = 4 - 1 = 3$	Wahr → 1. Schleifendurchlauf
2. Iteration	$3 > 1 ?$ $result = 4 * 3 = 12$ $x = 3 - 1 = 2$	Wahr → 2. Schleifendurchlauf
3. Iteration	$2 > 1 ?$ $result = 12 * 2 = 24$ $x = 2 - 1 = 1$	Wahr → 3. Schleifendurchlauf
4. Iteration	$1 > 1 ?$ $result = 24$	Falsch → Schleifendurchlauf wird beendet
	<code>return result</code>	Rückgabe des Ergebnisses der Methode

10.3 Rekursiver Algorithmus

Die meisten Programmiersprachen unterstützen sogenannte rekursive Funktionsaufrufe – auch Selbstaufrufe genannt. Das Adjektiv „rekursiv“ stammt vom lateinischen Verb „recurrere“, was „zurücklaufen“ bedeutet. Rekursive Aufrufe haben eine gewisse Ähnlichkeit mit Schleifen, da auch damit Anweisungen wiederholt werden. Die Idee des rekursiven Algorithmus ist es, einen Algorithmus durch sich selbst zu beschreiben. Die Rekursion muss jedoch an einer definierten Stelle beendet werden, sonst ist der Algorithmus nicht endlich.



Bei rekursiven Algorithmen werden Funktionen oder Methoden bzw. deren Zustand vor der konkreten Abarbeitung auf einem Stapel (Stack) abgelegt. Die Ausführung erfolgt, indem der Stapel von oben nach unten abgearbeitet wird. Der letzte rekursive Schritt wird also als Erstes ausgeführt.



Nun ist die FILO-Reihenfolge bei Rekursion nicht immer von Bedeutung. Im einfachen Fall ruft sich eine Funktion im Laufe der Abarbeitung bloß selbst wieder auf und macht irgendetwas, was vom vorherigen Aufruf nicht abhängt, oder man liefert keine Ausgabe, die den aktuellen Status anzeigt. Aber wenn ein Aufruf von dem vorherigen abhängt, dann ist die Reihenfolge natürlich massiv von Bedeutung. Gerade wenn eine rekursive Funktion einen Rückgabewert liefert, ist das FILO-Prinzip wichtig.

Fakultätsberechnung mit rekursivem Algorithmus

Die Fakultät einer ganzen Zahl x berechnet sich

- ✓ für alle $x > 0$: $x! = x * (x-1)!$
- ✓ für $x = 0$: $0! = 1$

Das Beispiel zur Berechnung von $4!$ ergibt nach Auflösung aller Fakultäten:

$$4! = 4 * (4-1)! = 4 * 3 * (3-1)! = 4 * 3 * 2 * (2-1)! = 4 * 3 * 2 * 1 * (1-1)! = 24.$$

Beispiel für einen rekursiven Algorithmus: *FactorialRec.py*

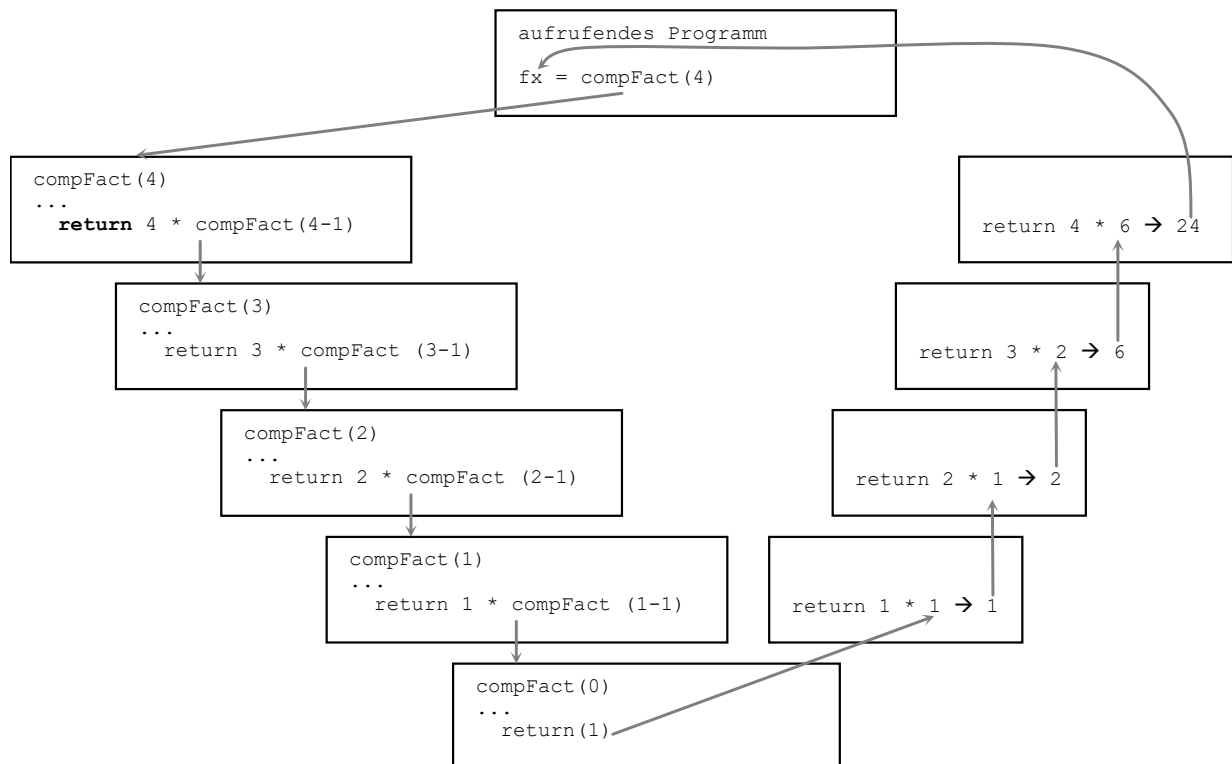
Mit dem rekursiven Algorithmus $x! = x * (x-1)!$ und dem Abbruchkriterium der Rekursion $0! = 1$ kann die Fakultät in Python wie folgt berechnet werden:

```
def compFact(x):
    ① if (x > 0):
    ②     return x * compFact(x-1)
    else:
    ③     return 1
```

- ① Die Abbruchbedingung wird dadurch definiert, dass die Rekursion nur so lange durchgeführt wird, wie der Parameter x einen Wert größer als null hat.
- ② Die Berechnung der Fakultät wird durchgeführt. Dabei ruft sich die Methode selbst mit dem neuen Parameter $(x-1)$ auf. Erst wenn der Methodenaufruf zurückkehrt und das Methodenergebnis liefert, kann die weitere Berechnung erfolgen.
- ③ Hat x den Wert 0, wird der Rückgabewert 1 zurückgeliefert, da gilt: $0! = 1$. Es findet kein neuer Methodenaufruf statt. Die Rekursion wird beendet.

Die Rekursion Schritt für Schritt

Zur Verdeutlichung des rekursiven Algorithmus wird die Methode anhand der übergebenen Daten betrachtet.



Der Aufruf der Methode `compFact(4)` erfolgt mit dem Wert 4 als Argument. Nach der Regel $x * (x-1)!$ ist die folgende Aktion $x * \text{compFact}(3)$. Die Methode `compFact` wird diesmal mit dem Wert 3 als Argument aufgerufen (1. Rekursion). Dies wird so lange wiederholt, bis in der 4. Rekursion festgestellt wird, dass das Argument 0 ist. Die Rekursion endet hier. Die einzelnen Resultate werden an die übergeordneten Rekursionen zurückgegeben. Nach dem Auflösen aller Rekursionen wird das Endergebnis der Methode zurückgeliefert: $4! = 24$.

10.4 Iterativ oder rekursiv?

Die iterativen und rekursiven Algorithmen spielen in der Programmierung eine wesentliche Rolle, da viele Probleme je nach Anforderung entweder durch Iteration oder Rekursion gelöst werden können.

Vor- und Nachteile iterativer und rekursiver Algorithmen

	Iterativ	Rekursiv
Vorteil	<ul style="list-style-type: none"> ✓ Keine hohen Systemanforderungen, da lokale Variablen benutzt werden 	<ul style="list-style-type: none"> ✓ Einfacher und kurzer Quellcode durch Reduzierung des Problems auf seinen einfachsten Fall ✓ Die Lösung ist schneller entwickelt. ✓ Passt sich an die Komplexität der Lösungsmöglichkeiten an (Sie müssen z. B. nicht die Anzahl der Schritte vorher berechnen und sich keine Zwischenschritte merken.)

	Iterativ	Rekursiv
Nachteil	<ul style="list-style-type: none"> ✓ Bei größeren Aufgaben kann ein komplexer Algorithmus entstehen. ✓ Zu langer und unübersichtlicher Quellcode 	<ul style="list-style-type: none"> ✓ Je nach Rekursionstiefe besteht ein hoher Ressourcenbedarf, z. B. an Hauptspeicher. ✓ Durch das ständige Merken der rekursiven „Sprünge“ in ein und dieselbe Funktion auf dem Stack kann es bei komplexen Aufgaben zu einem Speichermangel im System kommen (Stapelüberlauf). Die Folge ist Programmabsturz.

Welches Verfahren Sie verwenden, hängt von den allgemeinen Rahmenbedingungen ab. Rekursive Lösungen können schneller entwickelt werden, da der Algorithmus nicht komplex wird und somit übersichtlicher ist. Iterative Lösungen enthalten in den meisten Fällen einen komplexeren Algorithmus, können jedoch leichter optimiert werden.

10.5 Generischer Algorithmus

„Generisch“ bedeutet „die Gattung betreffend“. In der Programmierung bezieht sich diese Aussage auf die entsprechenden Datentypen. Üblicherweise sind Algorithmen in Programmen für spezielle Datentypen ausgelegt. Dies bedeutet, der Algorithmus kann entweder nur Daten vom Typ `Integer` oder vom Typ `String` bearbeiten. Falls sich der Algorithmus selbst für verschiedene Datentypen nicht unterscheidet, liegt es nahe, diese Anweisungen für alle Datentypen nur einmal zu implementieren. Die datentypabhängigen Funktionsteile werden herausgelöst und für die verschiedenen Datentypen jeweils separat bereitgestellt. Dadurch kann ein generischer Algorithmus mit verschiedenen Datentypen arbeiten.

Generische Algorithmen werden beispielsweise in den Standardbibliotheken verschiedener Programmiersprachen zur Verfügung gestellt. Sie dienen z. B. dem Sortieren von Feldern und Verwalten von Listen. Ein typischer Vertreter dieser Gattung ist ein „Sortier-Algorithmus“, der oft als Funktion `sort()` implementiert ist. Er sortiert auf- oder absteigend eine Menge von Ganzzahlen (`Integer`), Fließkommazahlen (`Real`) sowie Zeichenketten (`String`) und liefert das Ergebnis an das Programm zurück. Damit der Algorithmus Daten eines beliebigen Typs sortieren kann, müssen Sie eine separate Vergleichsfunktion (`compare`) für die einzelnen Datentypen entwickeln. Dieser werden zwei Datenelemente des betreffenden Typs als Parameter übergeben. Für jeden Datentyp wird die Vergleichsfunktion definiert, z. B. können Namen nach dem Alphabet sortiert werden und Adressen nach der Postleitzahl.

10.6 Übung

Iterative und rekursive Algorithmen

Übungsdatei: --

Ergebnisdateien: *uebung10.pdf*, *AddRec.py*, *AddIter.py*,
KingChess.java,
GreatestCommonDivisor.java

1. Die Multiplikation der ganzen Zahlen x und y kann auch als x -maliges Addieren der Zahl y gelöst werden. Zum Beispiel ist $3 * 4 = 4 + 4 + 4 = 12$.
Erstellen Sie eine rekursive Funktion für diese Operation.
2. Lösen Sie die Übung 1 mithilfe eines iterativen Algorithmus.
3. Erstellen Sie für die Aufgaben 1 und 2 jeweils ein Python-Programm. Speichern Sie die Programme unter *AddRec.py* und *AddIter.py*.
4. Berechnen Sie mit einem rekursiven Algorithmus die Anzahl der Weizenkörner, ausgehend von einem Schachspiel. Das erste Feld soll mit einem Korn belegt sein, das zweite Feld mit zwei Körnern und so fort. Die Zahl der Körner soll auf jedem Feld verdoppelt werden.
5. Lösen Sie die Aufgabe 4 mit einem Python-Programm. Speichern Sie das Programm unter *KingChess.py*.
6. Erstellen Sie einen iterativen Algorithmus, der den größten gemeinsamen Teiler von zwei vorgegebenen ganzen Zahlen ermittelt. Verwenden Sie dazu den Algorithmus von Euklid, der die folgenden Schritte umfasst.

- ✓ Ausgangspunkt sind zwei ganze Zahlen, z. B. 75 und 54.
- ✓ Bilden Sie durch die ganzzahlige Division den Quotienten aus der größeren Zahl (Dividend) und der kleineren Zahl (Divisor), z. B. $75/54$.
- ✓ Ist der Rest der ganzzahligen Division verschieden von 0, wird ein neues Zahlenpaar aus der kleineren Zahl und dem Rest der ganzzahligen Division gebildet, z. B. 54 und 21.
- ✓ Bilden Sie davon analog zum ersten Zahlenpaar den Quotienten, z. B. $54/21$.
- ✓ Wiederholen Sie den Vorgang so lange, bis der Rest der ganzzahligen Division 0 ist.
- ✓ Der größte gemeinsame Teiler ist der Divisor der letzten Division.

Zahlenpaar		Quotient	Rest	
75	54	1	21	>0
54	21	2	12	>0
21	12	1	9	>0
12	9	1	3	>0
9	3	3	0	=0

größter gemeinsamer Teiler

7. Lösen Sie die Aufgabe 6 mit einem Python-Programm. Speichern Sie das Programm unter *GreatestCommonDivisor.py*.
8. Die Verzeichnisse und Dateien in einem Betriebssystem sind in der Regel innerhalb einer Baumstruktur angeordnet. Um eine bestimmte Datei zu suchen, werden die Ordner und Unterordner nacheinander durchsucht, bis die Datei gefunden wird. Wie kann ein Algorithmus zum Suchen einer Datei in einer solchen Baumstruktur aussehen? Geben Sie eine iterative und eine rekursive Lösung an.

11 Reaktion auf Ereignisse

In diesem Kapitel erfahren Sie

- ✓ was Timer und Scheduler sind
- ✓ wie Sie auf Aktionen eines Benutzers reagieren können
- ✓ was eine Ereignisbehandlung ist – insbesondere bei Programmen mit grafischen Oberflächen
- ✓ welche Arten von Ereignissen es gibt
- ✓ welche verschiedenen Techniken der Ereignisbehandlung es gibt

11.1 Grundlagen zu Ereignissen und Eventhandlern

Ein Programm arbeitet nach dem EVA-Prinzip. Dabei kann es nach einem Start rein sequenziell Daten verarbeiten, aber es gibt selbstverständlich mehr Situationen als das reine Starten des Programms mit streng chronologischer Abarbeitung von Anweisungen. Es gibt Situationen, in denen gezielt eine Funktion oder Anweisung aufgerufen werden sollte. Aber diese können in modernen Programmen zu einem beliebigen Zeitpunkt eintreten – entweder aufgrund einer Zeitsteuerung oder einer Aktion, die auch ein Anwender auslösen kann.

Ereignisse und Eventhandler

Hier kommt der Begriff **Ereignis** (engl. **event**) ins Spiel. Unter einem Ereignis können Sie alles verstehen, was während der Lebenszeit eines Programms auftreten kann. Das können

- ✓ ein Zeitpunkt oder eine Zeitspanne,
- ✓ das Starten oder Beenden des Programms,
- ✓ eine beliebige Aktion eines Anwenders oder
- ✓ ein Schritt des Programms

sein.

Beispielsweise versteht man den Klick auf eine Schaltfläche oder das Überstreichen eines Teils der grafischen Benutzeroberfläche mit dem Mauszeiger als Ereignis. Solche Ereignisse bieten sich auf Basis für Aufrufe von Anweisungen ideal an.

Aber man braucht einen Mechanismus, über den auf solche Ereignisse gezielt reagiert werden kann und ggf. eine Anweisung, Funktion oder eine Methode ausgelöst wird. Dieser Mechanismus wird allgemein **EventHandler** (Ereignisbehandler) genannt. Mittels der unterschiedlichen Eventhandler reagiert man auf verschiedenartige Ereignisse, die der Eventhandler-Mechanismus über festgelegte Verfahren identifiziert.

Dabei nehmen **zeitgesteuerte** Aktionen in vielen Programmiersprachen einen Sonderstatus ein. Das liegt nicht zuletzt daran, dass man solche zeitgesteuerten Aktionen schon vor der Etablierung von Eventhandlern zur Verfügung hatte und sie auch heute noch in Techniken und Sprachen bereitstehen, die gar kein Eventhandling im eigentlichen Sinn gestatten.

Das Event-Objekt

Nun liegt hinter der Ereignisbehandlung in den meisten Programmiersprachen ein spezielles Objekt – das **Ereignisobjekt** beziehungsweise **Event-Objekt**. Solche Ereignisobjekte werden von einem Laufzeitsystem unablässig erzeugt. Ein solches Objekt kann zur Reaktion auf Ereignisse gezielt verwertet werden. Das Ereignisobjekt stellt dazu in der Regel eine Reihe interessanter Eigenschaften mit spezifischen Informationen sowie diverse Methoden bereit. Betrachten wir als Beispiel die Situation, wenn ein Anwender mit der Maus in irgendeinen Bereich einer grafischen Oberfläche klickt. Das erzeugt ein Ereignisobjekt, das unter anderem folgende Informationen enthalten kann:

- ✓ die verwendete Maustaste,
- ✓ eventuell gedrückte Zusatz Tasten ((Strg), (Alt), (CapsLock)),
- ✓ die Koordinaten des Klicks.

Andere Ereignisobjekte, die bei weiteren Ereignissen erzeugt werden, beinhalten natürlich andere Informationen, die dem Ereignis angepasst sind. So steht beispielsweise bei einem Tastendruck die gedrückte Taste als abzufragende Information bereit. Allgemein beinhaltet ein Ereignisobjekt zahlreiche sinnvolle Informationen, die Sie zur Erstellung gut angepasster Applikationen nutzen können.

11.2 Timer und Scheduler

Zuerst sollen nun zeitgesteuerte Aktionen betrachtet werden. Als **Timer** (engl. für Zeitmesser) oder Zeitgeber im engeren Sinne bezeichnet man ein Steuerelement, das zur Realisierung der unterschiedlichsten zeitbezogenen Funktionen sowie in Zählern eingesetzt wird.

Ein sogenannter **Scheduler** nutzt einen Timer und arbeitet Programmstücke nach einer festen Zeitspanne oder zu einem fixen Zeitpunkt einmal oder wiederholt ab. So etwas kann man sogar auf Ebene des Betriebssystems finden (und gerade da), wo bestimmte Aufgaben regelmäßig zu bestimmten Zeitpunkten ausgeführt werden müssen.

The Epoch

Beim Umgang mit Datum und Uhrzeit arbeiten fast alle Computersysteme gleich. Der 01. Januar 1970, 0.00 Uhr, ist der interne Zeitnullpunkt, der in nahezu allen Skript- und Programmiersprachen als Speicherungs- und Berechnungsbasis für alle Operationen mit einem Datum dient. Ab diesem Zeitpunkt wird die Zeit numerisch gezählt – sowohl rückwärts bei Terminen davor als auch natürlich in die Zukunft, entweder ganz genau in Millisekunden oder zumindest in Sekunden. Und da ein jedes Datum immer intern in einer ganzen Zahl verwaltet wird, können Sie mit Datumsobjekten meist auch wie mit gewöhnlichen Zahlen rechnen.

Man nennt dieses Verfahren auch die **Unixzeit**, die für das Betriebssystem Unix entwickelt und als POSIX-Standard festgelegt wurde. Vor der Unix Version 6, die 1975 abgelöst wurde, zählte die Unix-Uhr in Hundertstelsekunden. Seit der Unix-Version 6 zählt die Unixzeit die vergangenen Sekunden seit dem Zeitnullpunkt. Dieses Startdatum wird auch als „The Epoch“ bezeichnet.

Die Python-Module zum Umgang mit Datum und Uhrzeit

Python stellt in seiner **Standardbibliothek** Module bereit, die den Umgang mit Zeit und Datum erlauben und so einen Scheduler erstellen lassen. Die Funktionalitäten sind weitgehend in den Modulen `datetime`, `calendar` oder `time` zu finden und in der Dokumentation unter <https://docs.python.org/3/library/datatypes.html> beschrieben.

In Dateien, wo diese Module verwendet werden sollen, notiert man eine `import`-Anweisung und gibt den Namen des Moduls (der Dateiname ohne Erweiterung) an. Der Zugriff erfolgt dann über den Namen und die Punktnotation. Das folgende Beispiel soll einfach die Abarbeitung des Programms an einer Stelle kurz pausieren lassen.

Beispiel: MyTimer.py

```
import time

print(1)
time.sleep(2)
print(2))
```

- ✓ Zuerst wird im Quellcode dieser Datei das `time`-Modul mit `import` eingebunden. Da es sich um ein Modul der Standardbibliothek handelt, kann hier einfach dessen Name notiert werden.
- ✓ Die Ausgabe der 1 erfolgt direkt nach dem Start des Programms. Der Timer sorgt über die Methode `sleep()` dafür, dass die Abarbeitung des Programms zwei Sekunden pausiert und dann erst die 2 ausgegeben wird.

Diese Funktionalität kann man auch zur Erzeugung einer kleinen Uhr mit einer Konsolenausgabe verwenden.

Beispiel: MyClock.py

```
import time

i = 0
while(i < 10):
    localtime = time.asctime( time.localtime(time.time()) )
    print( "Aktuelle Zeit:", localtime)
    i += 1
    time.sleep(1)
```

- ✓ Zuerst wird im Quellcode dieser Datei das `time`-Modul mit `import` eingebunden.
- ✓ Mit der Zählvariable wird die Anzahl der Schleifendurchläufe überwacht. Es sollen zehn Ausgaben der Uhrzeit erfolgen.
- ✓ Mit der Methode `asctime()` aus dem Modul `time` wird der Zeitwert formatiert, der mit der Methode `time()` ermittelt wurde. Dieser Wert wird mit `localtime()` auf die lokale Zeitzone angepasst.
- ✓ Dann sorgt der Timer über die Methode `sleep()` dafür, dass die Abarbeitung der Schleife eine Sekunde pausiert.

Die Ausgabe sieht dann ungefähr so aus:

```
Aktuelle Zeit: Wed Jan 24 07:55:15 2018
Aktuelle Zeit: Wed Jan 24 07:55:16 2018
Aktuelle Zeit: Wed Jan 24 07:55:17 2018
Aktuelle Zeit: Wed Jan 24 07:55:18 2018
Aktuelle Zeit: Wed Jan 24 07:55:19 2018
Aktuelle Zeit: Wed Jan 24 07:55:20 2018
Aktuelle Zeit: Wed Jan 24 07:55:21 2018
Aktuelle Zeit: Wed Jan 24 07:55:22 2018
Aktuelle Zeit: Wed Jan 24 07:55:23 2018
Aktuelle Zeit: Wed Jan 24 07:55:24 2018
```

11.3 Ereignisbehandlung bei Programmen mit grafischen Oberflächen

Insbesondere moderne Programme mit grafischen Oberflächen sind ereignisgesteuert. Kaum ein modernes Clientprogramm kommt ohne grafische Oberfläche (GUI – Graphical User Interface) daher. Natürlich kann man auch in Python solche grafischen Oberflächen erstellen. Dazu stellt Python ein Framework mit allen zu erwartenden Elementen bereit. Dieses Framework zählt zur Standardbibliothek von Python und soll die ereignisgesteuerte Programmierung praktisch demonstrieren.

Hintergrundinformationen zu modernen grafischen Oberflächen

Die GUI-Programmierung basiert in den meisten modernen Programmiersprachen auf ähnlichen Konzepten. Wie bei fast allen modernen Programmiersprachen stellt auch das GUI-Framework von Python allgemeine Komponenten der Anwenderschnittstelle wie Fenster, Schaltflächen oder Menüs zur Verfügung. Es gibt also Komponenten und Container, in denen die Komponenten integriert werden müssen, damit eine vollständige und sinnvolle Anwenderschnittstelle erstellt werden kann.

Der äußerste Container ist in der Regel ein Frame. Ein weiterer bedeutender Bestandteil ist die Technik der Layout-Manager, die sich um das grundsätzliche Layout einer Applikation kümmern. Zusätzlich braucht es aber auch einen Mechanismus, um auf Ereignisse reagieren zu können, die vom Anwender über Komponenten ausgelöst werden. Das sind die besagten Eventhandler.

Eine Python-GUI basiert genau auf dem Grundkonzept, dass jedes Fenster mehrere verschachtelte Ebenen besitzt, die aus Komponenten aufgebaut sind. Dies beginnt mit dem äußersten Fenster und endet mit dem kleinsten eigenständigen Bestandteil der Benutzeroberfläche. Diese Verschachtelung bildet eine Hierarchiestruktur. Die Klassen des GUI-Frameworks sind so entwickelt worden, dass deren Struktur eine vollständige Benutzeroberfläche abbilden kann, wobei das vollständige Konzept in dem Kapitel nur angedeutet werden kann.

Die einzelnen GUI-Komponenten sind die Elemente, über die eine Interaktion mit dem Endanwender konkret realisiert wird. Sie sind aus den unterschiedlichen grafischen Benutzerschnittstellen bekannt. Typische Komponenten sind

- ✓ Schaltflächen,
- ✓ Label,
- ✓ Kontrollkästchen,
- ✓ Optionsfelder,
- ✓ Listen,
- ✓ Auswahlfelder,
- ✓ Textfelder,
- ✓ Textbereiche oder
- ✓ Menüs.

Oberflächenprogrammierung umfasst aber wie gesagt auch die Reaktion auf Benutzeraktionen. Wenn ein Anwender etwa auf eine Schaltfläche klickt, muss im Programm hinterlegt sein, was daraufhin zu tun ist. Das Eventhandling von Python basiert auf Ereignismethoden, die mit Komponenten verbunden werden.

Konkrete GUI-Konzepte in Python und das Modul `tkinter`

Python-GUI-Anwendungen basieren auf dem Modul `tkinter` (<https://docs.python.org/3/library/tk.html>). Fenster und Steuerelemente werden in dem Framework als solche gerade beschriebenen GUI-Elemente betrachtet, die sich ineinander verschachteln lassen. Um eine GUI-Anwendung zu schreiben, muss man also wissen, wie sich die jeweiligen GUI-Elemente erzeugen lassen und wie man sie dann ineinander verschachtelt.

Ein Fenster vom Typ Tk als Basis jeder GUI-Applikation

Die einfachste Variante einer GUI nutzt ein Objekt vom Typ Tk und eine sogenannte `mainloop`, mit der die Anwendungssteuerung an die GUI übergeht. So ein minimaler Code erzeugt bereits ein grafisches Fenster mit einer Vorgabegröße und allen üblichen Features wie der Möglichkeit zum Vergrößern, Verschieben oder Ablegen in der Taskleiste.

Beispiel: *MinimalGUI.py*

```
from tkinter import *  
master=Tk()  
master.mainloop()
```

Wichtig ist die erste Zeile, mit der das `tkinter`-Modul importiert wird.

Die Layout-Manager bzw. Geometry-Manager

Das Hinzufügen von Elementen einer grafischen Oberfläche läuft bei `tkinter` beziehungsweise in Python über das Konzept von Layout-Managern oder Geometry-Managern.

Moderne grafische Oberflächen folgen oft einem Konzept von verschachtelten Anordnungs-Containern. Ein bedeutender Bestandteil des Konzepts ist dabei meist die Technik solcher Layout-Manager. Wenn man so ein Konzept im GUI-Framework einsetzt, muss man sich als Programmierer beim Hinzufügen von Komponenten (Schaltflächen, Eingabefelder etc.) zu Container-Objekten in vielen Fällen (scheinbar) nicht um die genauen Positionen innerhalb der Container oder vor allen Dingen deren Größe kümmern. Diese Komponenten werden einfach „irgendwo“ auf der Oberfläche platziert. Sie haben irgendeine, scheinbar zufällige Größe, die sich manchmal auch verändern kann, wenn sich die Größe vom umgebenden Fenster, der Komponente selbst oder dem Anordnungs-Container ändert.

Dieses Verhalten ist jedoch kein Zufall und keinesfalls ohne System. Positionen und Größen werden nur automatisch im Hintergrund gemanagt. Das ist eine der großen Stärken dieses GUI-Konzepts.

Bei konventionellen Fensterkonzepten muss ein Programmierer sämtliche denkbaren Auflösungen, Fenstergrößen und Orientierungen berücksichtigen (zumindest diejenigen, die er unterstützen möchte), diese zur Laufzeit abfragen und für jede Maske eine individuelle Oberfläche für jede Auflösung generieren. Das ist schon ein erheblicher Aufwand und dennoch immer noch keine befriedigende Lösung, denn unter Umständen hat man nicht alle Situationen bedacht, und es kann sogar zum Verdecken von Komponenten der Oberfläche kommen.

Layout-Manager bzw. Geometry-Manager lösen diese Probleme auf eine intelligente Art und Weise. Das GUI-System kümmert sich weitgehend selbstständig um Größenanpassung und Positionierung der Komponenten auf der Oberfläche. Je nach Plattform und Bedingungen werden die Komponenten auf der Oberfläche optimal angepasst. Und wenn man Container verschachtelt, hat man genügend Kontrolle über Position und Größe von Komponenten, ohne sich so weit festzulegen, dass eine veränderte Bedingung ein System zur Unfähigkeit degradieren kann.

Ein Layout-Manager bzw. Geometry-Manager findet nach gewissen Regeln heraus, an welche Stellen die Komponenten am besten passen und ermittelt die besten Größen der Komponenten. Aber man kann in einem GUI-Framework natürlich auch Layout-Manager erstellen, die eine genaue Positionierung von Komponenten gestatten.

Layout-Manager bzw. Geometry-Manager können verschachtelt werden. Sehr oft kombiniert man flexible Layout-Manager bei äußeren Containern mit fixierenden Layout-Managern im Inneren. Das werden wir hier aber nicht ausarbeiten, da GUI-Programmierung nur ein Thema von vielen darstellt.



Das genaue Aussehen einer solchen flexiblen Oberfläche wird im Allgemeinen von mehreren Aspekten festgelegt.

- ✓ Der Plattform und den genauen Bedingungen dort. Darüber kann ein Programmierer keine Aussagen machen und muss es auch nicht. Darum geht es ja gerade.
- ✓ Der Reihenfolge, in der die Komponenten in einem Container eingefügt werden. Dieses Faktum ist naheliegend. Einmal kann gelten: Wer zuerst kommt, malt zuerst. Das soll heißen, dass zuerst eingefügte Komponenten auch vorher angeordnet werden. Allerdings kann es auch sein, dass eine später eingefügte Komponente eine vorherige ersetzt oder überlagert oder dass die Reihenfolge der Einfügung gar keine Rolle spielt. Wie das genau abläuft, hängt an dem entsprechenden GUI-Konzept.

Richtig interessant ist die Art des Layout-Managers. Die drei bzw. Geometry-Manager von `tkinter` sind diese:

- ✓ `place` (spezifiziert die Position von einem Element durch Angabe von x/y-Koordinaten)
- ✓ `grid` (Gitterlayout)
- ✓ `pack` (schafft ein fließendes Layout, das den verfügbaren Platz optimal auszunutzen versucht)

Die Ereignisbehandlung

Oberflächenprogrammierung umfasst auch die Reaktion auf Benutzeraktionen und einige andere Ereignisse. Wenn ein Anwender etwa auf eine Schaltfläche klickt, muss im Programm hinterlegt sein, was daraufhin zu tun ist. Dabei gibt es verschiedene Modelle, wie auf Ereignisse zu reagieren ist. Aber die meisten modernen Programmiersprachen fokussieren sich auf ein ähnliches Konzept.

Wenn Ereignisse entstehen, sind das Mitteilungsobjekte. Und die müssen irgendwie ausgewertet werden. Dazu gibt es entsprechend aufgebaute Mechanismen, über die eine Reaktion des Programms bewirkt wird. Das Auftreten eines Ereignisobjekts selbst ist also noch keine Reaktion des Programms. Es muss zu einem Auswertungsobjekt transportiert werden.

Dies funktioniert schematisch so:

- ✓ Ein Ereignis tritt bei einem Quellobjekt (dem sogenannten Event Source) auf, etwa einem Button oder einem Menüeintrag.
- ✓ Das entstandene Ereignisobjekt wird an ein Zuhörerobjekt (das nennt man in manchen Konzepten einen Event Listener oder kurz Listener oder in anderen Konzepten einen Event Handler beziehungsweise kurz Handler) weitergeleitet.
- ✓ Erst dort wird über die konkrete Ereignisbehandlung entschieden und die Reaktion auch ausgelöst.

Die konkrete Ereignisbehandlung in Python

Für die Ereignisbehandlung in Python gibt es die `command`-Anweisung. Diese kann als Attribut beim Instanzieren eines GUI-Objekts wie ein Button angegeben werden.

Beispiel:

```
Button(master, text='OK', width=20, command=self.action)
```

Oder man gibt die Referenz auf eine Methode als Wertzuweisung der Eigenschaft an.

Beispiel:

```
self.okButton['command']=self.action
```

Die Referenz auf konkrete Aktion steht in beiden Codes über das Attribut `action` bereit, das eine Funktionsreferenz darstellt.

Beispiel: GUI.py

```
from tkinter import *

def action():
    lbl1.config(text = "Hallo Welt")

master=Tk()
Button(master,text='OK',width=20,command=action).grid(row=0, column=0)
lbl1=Label(master, fg="blue")
lbl1.grid(row=1, column=0)
master.mainloop()
```

Wenn das Programm startet, zeigt die Oberfläche eine Schaltfläche an. Beim Klick auf die Schaltfläche wird dann eine Meldung in einem Label angezeigt.



Eine grafische Applikation mit Ereignisbehandlung

Da ein Gitter-Layout vorliegt, wird mit dem Attribut `row` die Zeile und mit `column` die Spalte angegeben, wo ein Element angeordnet werden soll.

Die Ereignisbehandlung wird mit dem `command`-Attribut bei der Schaltfläche implementiert. Sie sehen da eine Referenz auf die Methode `action()`, in der dann ein Text mit der `config()`-Methode bei dem Label `lbl1` angezeigt wird.

11.4 Verschiedene Techniken der Ereignisbehandlung

Nun gibt es bei der Behandlung von Ereignissen einige unterschiedliche Techniken, deren Konzepte hier zum Abschluss kompakt vorgestellt werden. Beachten Sie, dass diese nicht in allen Programmiersprachen in vollem Umfang vorhanden sind.

Blubbern von Ereignissen und die Bubble-Phase

Der Begriff der Bubble-Events beziehungsweise der Bubble-Phase klingt vielleicht komisch, ist aber grundlegend in der Ereignisbehandlung. Wie erwähnt, erzeugt ein Programm mit einer ereignisgesteuerten Programmierung meist unablässig Ereignisobjekte. Denken Sie nicht nur an die wenigen Ereignisse wie den Klick mit der Maus oder Tastatureingaben. Das Verschieben der Maus über den Anzeigebereich des Programms findet quasi permanent statt. In kürzester Zeit können Tausende von Ereignisobjekten entstehen und wieder vergehen. Die meisten solcher Ereignisse werden also von der Applikation nicht verwertet, sondern schlicht und einfach ignoriert. Oder warum sollte man – außer in speziellen Fällen – auf jeden Millimeter einer Mausbewegung reagieren? Aber auf manche Ereignisse will man gezielt reagieren. Und dann stellt sich die Frage, wer für die Behandlung zuständig ist.

Wenn nun ein Ereignis bei einem Knoten in einer Baumstruktur wie bei einer `tkinter`-GUI auftritt, stellt sich die Frage, welches der Objekte im Baum nun für ein aufgetretenes Ereignis zuständig ist. Knoten sind immerhin im Baum sehr tief ineinander verschachtelt. Wenn ein Anwender auf eine Schaltfläche in einer Oberfläche klickt, hat er ja auch gleichzeitig auf das Fenster selbst geklickt. Ist das Fenster oder die Schaltfläche für die Behandlung zuständig? Und wie erfährt der Knoten des Frames gegebenenfalls davon, dass auf den untergeordneten Knoten der Schaltfläche ein Klick erfolgt ist. Oder was soll passieren, wenn mehrere ineinander geschachtelte Objekte den gleichen Eventhandler besitzen und deshalb auf das Ereignis reagieren könnten?

Die Probleme, beispielsweise Fragen, werden über das sogenannte Event-Bubbling gelöst. Ein Ereignis wird bei diesem Konzept immer zuerst im innersten Element behandelt, bei dem es aufgetreten ist – sofern es dort einen geeigneten Eventhandler gibt!

Das innerste Element ist in der Hierarchie des GUI-Baums am weitesten von der Wurzel entfernt. Hat dieses Element keinen geeigneten Eventhandler, wird das Ereignisobjekt an das nächsthöhere Objekt im GUI-Baum (dem direkten Vorfahren) weitergeleitet usw. Es steigt wie eine Blase (Bubble) über alle Vorfahren hinauf bis zur Wurzel. Es blubbert durch den Baum, bis es behandelt wird.

Datenbindung

Zur Reaktion auf Ereignisse kennt man auch den Mechanismus der Datenbindung (Data binding). Man kann damit eine unmittelbare und direkte Beziehung zwischen zwei Variablen, beispielsweise Ausdrücken, erstellen. Damit assoziieren Sie den Wert eines Ziels mit dem Wert eines gebundenen Ausdrucks. So ein gebundener Ausdruck kann ein einfacher Wert eines beliebigen Typs, ein Objekt, das Ergebnis einer Funktion oder eines Ausdrucks sein.

Das Verfahren ist so ähnlich zu verstehen wie Referenzbezüge zwischen Zellen in einer Tabelle einer Tabellenkalkulation wie OpenOffice Calc oder Microsoft Excel, insbesondere wenn diese in Formeln verwendet werden und sich diese Formeln auf andere Zellen beziehen. Wenn sich der Wert der referenzierten Zelle auf irgendeine Weise ändert, werden sich alle anderen daran gebundenen Ausdrücke in der Weise ändern, wie die Bindungsvorschrift (die Formel) das verlangt – ohne Verzögerung und ohne dass die Aktualisierung noch manuell ausgelöst werden muss.

Trigger

Zur Reaktion auf bestimmte Ereignisse gibt es ansonsten den Mechanismus der sogenannten Trigger (Auslöser). Ein solcher Trigger wird in der Regel bei Ereignissen ausgelöst, die sich auf Grund von Modifikationen von Daten ergeben. Trigger sind sinnvoll, wenn der auszuführende Code bei einer Wertänderung einer Variablen flexibel oder komplex ist.

Delegation

Das Delegationsereignismodell ist eine weitere der vielen Techniken, die verwendet werden, um Ereignisse in einer GUI programmiertechnisch zu behandeln. Im Delegationsereignismodell erzeugt eine Ereignisquelle (event source) ein Ereignis und sendet es an einen oder mehrere Zuhörer (listener). Die Verantwortung für die Bearbeitung des Veranstaltungsprozesses wird dem Listener übergeben. Ein Listener wartet permanent darauf, aktiv zu werden, wenn ein Ereignis stattgefunden hat, das für ihn interessant ist. Dieses Design der Ereignisbehandlung ist sauber von der Hauptanwendungslogik entkoppelt, die das Ereignis erzeugt, und sehr effizient. Allerdings müssen sich die Zuhörer registrieren oder sich mit der Ereignisquellenklasse abstimmen, um eine Benachrichtigung zu erhalten. Dies bedeutet, dass dann eben ein bestimmtes Ereignis nur von einem bestimmten Zuhörer verarbeitet wird. Eine Komponente kann so auf zahlreiche Arten von Ereignissen reagieren, wenn jeweils passende Listener registriert werden.

Nun muss man beachten, dass der Begriff der „Delegation“ nicht wirklich standardisiert ist und in Sprachen wie C# und .NET, Swift, JavaScript oder in Java unterschiedlich verstanden werden kann. Vereinfacht gesagt bedeutet es aber in allen Fällen, dass die Ereignisse, die eigentlich einem Element A zugehörig sind, von einem Element B behandelt werden. Man reicht also die Verantwortung von einem Element an ein anderes Element weiter (dem Delegaten).

11.5 Schnellübersicht

Was bedeutet ...?	
Event	Ein Ereignis, auf das reagiert werden kann
Scheduler	Eine zeitgesteuerte Aufgabe
Event Handler	Ein Reaktionsmechanismus auf ein Ereignis
Bubble-Phase	Das Weiterreichen eines Eventobjekts in einem verschachtelten Baum an möglichen Listener.

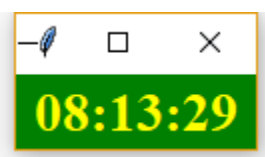
11.6 Übung

Fragen zur Ereignisbehandlung

Übungsdatei: --

Ergebnisdateien: *uebung11.pdf, MyClockGUI.py*

1. Erstellen Sie als Python-Programm eine Uhr mit einer grafischen Oberfläche.
2. Verwenden Sie das `time`- und das `tkinter`-Modul.
3. Deklarieren Sie eine Funktion, in der die Zeit ermittelt wird. Es gibt verschiedene Möglichkeiten, um in Python ein Datumsobjekt für einen beliebigen Zeitpunkt zu erstellen. In der Dokumentation finden Sie entsprechende Informationen (<https://docs.python.org/3/library/time.html>). Verwenden Sie die Methode `strptime()` aus dem Modul `time`. Diese nimmt sogenannte Direktiven zur Aufbereitung des Datumsobjekts an. Mit `%H` erhalten Sie die Stunden im 24-Stunden-Format, mit `%M` die Minuten und mit `%S` die Sekunden.
4. Wenn das Programm startet, soll die Oberfläche ein Label anzeigen, in dem die aktuelle Uhrzeit zu sehen ist.
5. Verwenden Sie den pack-Geometry-Manager (<https://docs.python.org/3/library/tk.html>). Damit wird die Fenstergröße auf den Raum optimiert, den das Label benötigt.



So soll die grafische Uhr aussehen

12 Grundlagen der Softwareentwicklung

In diesem Kapitel erfahren Sie

- ✓ wie Sie konzeptionell beim Entwurf einer Software vorgehen können
- ✓ was das Phasenmodell des Software-Lebenszyklus ist
- ✓ nach welchen Vorgehensmodellen Software entwickelt werden kann
- ✓ welche Anforderungen an ein Programm gestellt werden
- ✓ wie man Software testen kann
- ✓ wie man Fehler in Software findet

12.1 Software entwickeln

Vorgehensmodelle einsetzen

Abhängig von der zu erstellenden Software werden bei der professionellen Programmierung **Vorgehensmodelle** gewählt, die festlegen, wie die Software entwickelt wird. Ein Vorgehensmodell beschreibt, welche Prinzipien, Methoden und Darstellungsmittel eingesetzt werden. Vorgehensmodelle sind in großen Projekten mit mehreren beteiligten Personen oder Gruppen/Abteilungen notwendig, um die Entwicklung der Software in verwaltbare und kontrollierbare Teile zu gliedern.

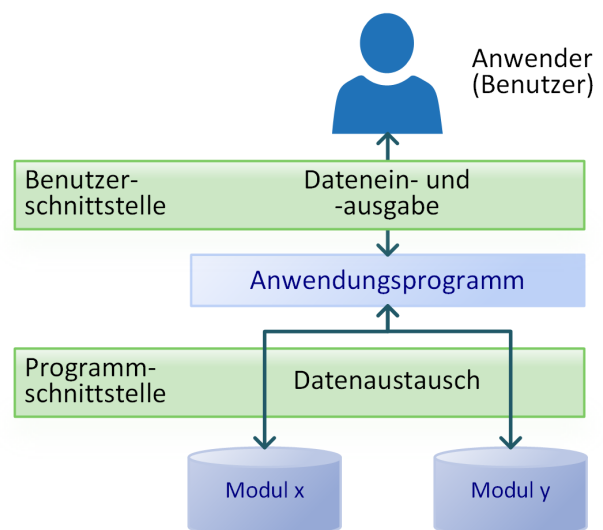
Prinzipien lassen sich aus Erfahrungen und Erkenntnissen herleiten. Prinzipien sind nicht speziell auf ein Anwendungsgebiet begrenzt. So lässt sich beispielsweise das Abstraktionsprinzip in der Informatik ebenso anwenden wie in der Mathematik oder Philosophie. Von grundlegender Bedeutung für die Entwicklung von Software sind das Modularitäts- und das Abstraktionsprinzip.

Wie eine große Aufgabe überschaubar wird

Das Prinzip der **Modularität** besagt, dass eine Gesamtaufgabe in Teilaufgaben (**Module**) zerlegt wird. Dadurch erhöht sich die Verständlichkeit und reduziert sich die Komplexität.

Durch die Modularisierung z. B. eines Programms können die Module einzeln programmiert und getestet werden.

Module besitzen eine definierte Schnittstelle zu deren Verwendung. Als **Schnittstelle** (engl. interface) wird die Verbindungsstelle zwischen Modulen selbst oder zwischen Programm und Anwender bezeichnet, über die ein Daten- bzw. Informationsaustausch stattfindet.



Vorteile der Modularisierung

Beschreibung	Vorteil
Bei der Programmierung von Software, die aus einzelnen Modulen besteht, können diese unabhängig voneinander programmiert werden. Die Module können somit in Teams entwickelt werden.	verkürzte Entwicklungszeit, bessere Verteilung von Arbeiten und Ressourcen
Jedes Modul kann einzeln auf seine Korrektheit getestet werden. Fehler, die nach dem Zusammenfügen der einzelnen Module auftreten, lassen sich relativ leicht finden, da sie bei Modulen, die selbst korrekt arbeiten, dann meist nur an den Schnittstellen auftreten. Die definierten Schnittstellen erlauben es, ein Modul durch ein anderes Modul mit identischer Schnittstelle auszutauschen.	verbesserte Wartbarkeit
Das Modul selbst kann in verschiedenen Programmen verwendet werden, in denen ein Modul mit der entsprechenden Funktionalität benötigt wird.	Wiederverwendbarkeit
Die Datenkapselung in Modulen stellt einen wichtigen Aspekt bezüglich Sicherheit und Zuverlässigkeit eines Programms dar. Unkontrollierte Zugriffe auf ein Modul sind bei einer sauberen Umsetzung (weitgehend) ausgeschlossen, Manipulationen sind nicht möglich.	Sicherheit und Zuverlässigkeit

Die Wiederverwendbarkeit von Modulen bzw. Softwarekomponenten spielt eine große Rolle bei der Einsparung von Entwicklungszeit und -kosten.



Beispiel: Modularisierung

Weltweit arbeiten viele Entwickler an sogenannter OpenSource-Software mit. Das ist Software, deren Quellcode für jedermann zugänglich ist, z. B. Linux oder Gimp. Die Entwicklung dieser Software ist nur durch Aufteilung in Module möglich, die dann von einzelnen Entwicklern bearbeitet werden können. Diese Module werden dann zu einem größeren System zusammengefasst und bilden damit erst als zusammengefügte Einheit das konkrete Programm.

Reduktion auf das Wesentliche durch Abstraktion

Das **Abstraktionsprinzip** ist wichtig, um unwesentliche Informationen von Aufgabenstellungen auszuschließen. Somit liegen nach der Abstraktion nur die zur Lösung einer vorgegebenen Aufgabe notwendigen Informationen vor. Eine Abstraktion ist damit eine Auswahl relevanter Informationen aus einer größeren Menge von verfügbaren Informationen.

Beispiel: Abstraktion

Das Durchschnittsalter von Schülern einer Klasse soll mithilfe eines Programms automatisch berechnet werden. Bekannt sind die Geburtsdaten der Schüler, deren Anzahl und Namen, das Geschlecht sowie das Datum des ersten Schultags. Die Namen der Schüler sowie deren Geschlecht und das Datum des ersten Schultags sind für die Berechnung des Durchschnittsalters unwesentlich. Durch die Abstraktion werden sie bei der Lösung der Aufgabe nicht berücksichtigt. Als notwendige Information für die Berechnung des Durchschnittsalters bleiben die Geburtsdaten und die Anzahl der Schüler. Die Geburtsdaten und die Anzahl sind die für den konkreten Fall gewünschte Abstraktion.

12.2 Methoden

Eine **Methode** ist eine systematische Vorgehensweise, um bestimmte Aufgaben im Rahmen festgelegter Prinzipien zu lösen. Ausgehend von den gegebenen Bedingungen wird ein Ziel mithilfe definierter Arbeitsschritte erreicht.



Beachten Sie, dass hier nicht die Definition der Methode gemeint ist, wie sie in der objektorientierten Programmierung Verwendung findet.

Schrittweise Verfeinerung (stepwise refinement)

Zunächst wird auf einer hohen Abstraktionsebene die Aufgabenstellung des Problems formuliert. Damit wird festgelegt, **was** gemacht werden soll.

Mit der nachfolgenden schrittweisen Verfeinerung der Aufgabenstellung wird die Frage beantwortet: **Wie** ist die Aufgabenstellung zu lösen?

Bleibt noch die Frage: **Womit** ist die Aufgabenstellung zu lösen, das heißt, mit welcher Programmiersprache, welchen Algorithmen und anderen Softwarekomponenten kann die Aufgabenstellung realisiert werden? Es existieren verschiedene Methoden des strukturierten Programmentwurfs:

Top-down-Methode	<p>Ausgangspunkt ist die Gesamtaufgabe, die in Teilaufgaben zerlegt wird. Bei komplexen Aufgabenstellungen können die Teilaufgaben in weitere Teilaufgaben aufgeteilt werden.</p> <p>Bei der Entwicklung eines einzelnen Programms ist diese Methode oft vorzuziehen.</p>
Bottom-up-Methode	<p>Diese Methode ist für Vorgehensweisen interessant, bei denen eine Aufgabenstellung nicht exakt beschrieben ist bzw. noch Änderungen erwartet werden. Einzelne entwickelte Module werden später zu einem großen Modul zusammengesetzt.</p> <p>Die Bottom-up-Methode ist meist beim Entwurf großer Programmsysteme sinnvoll.</p> <p>Bei dieser Methode können Probleme beim Zusammensetzen der Module zum Gesamtsystem auftreten, da zu Beginn das Gesamtsystem nicht genau festgelegt war. Der Vorteil besteht darin, dass bereits entwickelte Module eingesetzt werden können.</p>
Up-down-Methode (Middle-Out, Gegenstromverfahren)	<p>Bei dieser Strukturierungsmethode wird die Gesamtaufgabe durch Top-down-Methoden verfeinert, und es werden Teilaufgaben bottom-up abstrahiert. Auf diese Weise können kritische Teilaufgaben zuerst getestet werden.</p>

12.3 Der Software-Lebenszyklus

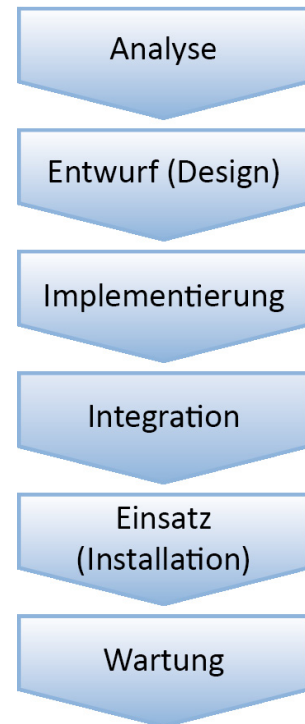
Professionelle Software wird zunehmend komplexer. Um die Erstellung trotzdem planen und kontrollieren zu können, wurden strukturierte Vorgehensmodelle entwickelt. Grundlage dieser Modelle ist die Einteilung des Software-Lebenszyklus in sogenannte **Phasen**.

In jeder Phase werden bestimmte Aufgaben erledigt, getestet und dokumentiert. Tests in den jeweiligen Phasen dienen dazu, Fehler bei der Entwicklung der Software zu einem möglichst frühen Zeitpunkt zu erkennen.

Je später im Entwicklungsprozess ein Fehler erkannt wird, umso höher ist der Aufwand, z. B. Kosten oder Zeit, zu seiner Beseitigung.

Eine Phase sollte beendet sein, bevor die nächste begonnen wird. In der Praxis kann die strikte Abfolge und Trennung der Phasen meist nicht eingehalten werden. Es kann vorkommen, dass bestimmte Aufgaben verschiedenen Phasen zugeordnet werden können. Ebenso sind nicht alle Schritte einer jeweiligen Phase immer zwingend notwendig.

Sowohl für die Entwicklung neuer Software als auch für die Wartung und Pflege bereits existierender Software ist das Phasenmodell verwendbar. Der Software-Lebenszyklus ist dadurch charakterisiert, dass einzelne Phasen aufgrund von Entscheidungen, die aus nachfolgenden Phasen resultieren, zyklisch wiederholt zu durchlaufen sind.



Programme testen

Das Testen eines Programms ist ein sehr wichtiger Schritt in der Softwareentwicklung. Dabei gibt es Abschlusstests, aber auch Tests, die während der Entwicklungszeit durchgeführt werden. Gerade die Tests während der unterschiedlichen Phasen sind elementar und werden deshalb explizit bei jeder Phase beschrieben.

Analysephase

Aufgabe	<p>Was soll die Software tun?</p> <p>Folgende Anforderungen werden in dieser Phase definiert:</p> <ul style="list-style-type: none"> ✓ Funktionsumfang und Qualitätsmerkmale des Programms ✓ Art der Benutzeroberfläche ✓ Schnittstelle der Systemumgebung ✓ Eingesetzte Hardware während der Programmierung ✓ Software, die zum Erstellen des Programms eingesetzt wird (entfällt beim programmiersprachenunabhängigen Entwurf) ✓ Umfang der Dokumentation ✓ Ein-/Ausgabe- und Testdaten festlegen <p>Außerdem können Wirtschaftlichkeitsbetrachtungen angestellt, Verantwortlichkeiten bestimmt und Termine für die Fertigstellung der einzelnen Phasen festgelegt werden.</p>
Test	<p>Anforderungstest</p> <p>Bei großen Softwaresystemen werden die Anforderungen in einer formalisierten Sprache erstellt und können so mit geeigneten Testverfahren automatisch kontrolliert werden. Als formale Sprache kommt im Fall der objektorientierten Programmierung beispielsweise UML (Unified Modeling Language) zum Einsatz, die Methoden für verschiedene Phasen zur Verfügung stellt.</p>
Ergebnis	Pflichtenheft mit eindeutig beschriebenen Anforderungen

Beispiel: Analysephase

Ein Busunternehmen besitzt mehrere Busse und Taxen. Die Fahrzeuge können für beliebige Fahrten bestellt werden. Das Busunternehmen stellt Fahrer und Fahrzeug für die gewünschte Zeit. Das Unternehmen ist stark gewachsen, und der Inhaber möchte für die Planung, Verwaltung und Rechnungslegung eine neue Software einsetzen.

In der Analysephase werden im Beispiel folgende Fragen gestellt und beantwortet:

Wozu soll das Programm dienen?	Zur Planung und Verwaltung der Fahrzeuge einschließlich Rechnungslegung
Welche Informationen sind vorgegeben? Wovon wird ausgegangen?	Anzahl der Busse und Taxen, Anzahl der Fahrer, Mietpreise für einzelnes Fahrzeug inklusive Fahrer, Kundenanfragen
Welche Informationen soll das Programm ausgeben?	Planungstabellen, Angebote, Verträge, Rechnungen, Mahnungen
Wie können diese Informationen ermittelt werden?	Kundenanfragen bzw. Kundendaten müssen erfasst werden. Die Daten für Busse, Taxen und Fahrer enthält eine Datenbank.

Entwurfsphase

Aufgabe	Wie ist die Software zu realisieren? ✓ Strukturierter Softwareentwurf, z. B. mithilfe der Top-down-Methode ✓ Beschreibung der Algorithmen mithilfe eines dafür geeigneten Darstellungsmittels ✓ Festlegung der Programmiersprache für die Implementierung ✓ Festlegung von Programmierstrukturen ✓ Anlegen der entsprechenden Dokumentation
Test	✓ Entwurfstest der Dokumente und des Designs ✓ Funktionstest des Prototyps, falls vorhanden
Ergebnis	Beschreibung des Entwurfs bzw. Softwarespezifikation

Der Schwerpunkt beim Entwurf liegt auf der Bestimmung der Aufgabe der einzelnen Module sowie auf dem Zusammenspiel und der Kommunikation der Module untereinander. Je nach festgelegter Programmiersprache kann sich die Vorgehensweise bei der Fortführung des Entwurfs ändern.

Den Entwurf kann man in drei verschiedene Phasen unterteilen:

- ✓ Grobentwurf
- ✓ Feinentwurf
- ✓ Implementationsentwurf



Ein guter Entwurf zahlt sich langfristig aus und verringert die Entwicklungszeit.

Beim **Grobentwurf** werden die Hauptmodule eines Programms festgelegt und ihre Beziehungen untereinander bestimmt und dokumentiert. Dabei ist es wichtig, dass die einzelnen Module bereits so ausgearbeitet werden, dass ihre Schnittstellen zur Wiederverwendung im Verlauf der weiteren Entwicklung nicht mehr geändert werden müssen.

Im **Feinentwurf** werden die Module des Grobentwurfs weiter untergliedert. Die Zerlegung wird so lange vorangetrieben, bis die Komplexität und die Größe aller Module so weit reduziert sind, dass sie präzise und ohne überflüssige Redundanz (mehrfache Speicherung von Informationen) definiert werden können.

Weiterhin werden beim Feinentwurf die Struktur der Daten und die benötigten Anweisungen präzisiert, und der Aspekt der späteren Wiederverwendung wird mit einbezogen. Das Ergebnis des Feinentwurfs wird in der implementationsunabhängigen Softwarespezifikation festgehalten.

Mit dem Ergebnis des Implementationsentwurfs liegt die Softwarespezifikation vor. In der nachfolgenden Implementierung werden die Module in eine Programmiersprache umgesetzt.

Beispiel: Entwurfsphase

Wie könnte der Grobentwurf für das Busunternehmen aussehen?

Welche Informationen sind vorgegeben?	Kundendaten, Mietpreis, verfügbare Busse, Taxen und Fahrer
Was soll ausgegeben werden?	Planungstabellen, Angebote, Verträge, Rechnungen, Mahnungen
Welche Funktionen sind dafür nötig?	<ul style="list-style-type: none"> ✓ Eingabemaske für die Erfassung der Kundendaten ✓ Eingabemaske für die Busse und Taxen ✓ Eingabemaske für die Fahrer ✓ Anzeigen der verfügbaren (freien) sowie der ausgebuchten Fahrzeuge und Fahrer ✓ Anzeigen der Kundendaten ✓ Planungsfunktion: Zuordnen von Fahrer, Fahrzeug und Kunde für entsprechende Zeit ✓ Abrechnungsfunktion für durchgeführte Einsätze ✓ Druckfunktion für Kundendaten, Angebote, Verträge, Fahrzeuge und Fahrer (Fahrauftrag), Rechnungen, Mahnungen
Welche zusätzlichen Funktionen sollen eingebaut werden?	<ul style="list-style-type: none"> ✓ Druckfunktion für Grußkarten zu Weihnachten und Ostern an die Kunden ✓ Fehlermeldung für ungültige Eingaben (z. B. unvollständige Adressen)

Die Aufgaben können nun auf mehrere Programmierer verteilt werden. So kann beispielsweise ein Mitarbeiter die Eingabemasken erstellen und ein weiterer Mitarbeiter sich um die Ausgaben kümmern.

Implementierungsphase

Aufgabe	<ul style="list-style-type: none"> ✓ Programmcode in einer Programmiersprache erstellen und dokumentieren ✓ Programmcode testen und dokumentieren
Test	Funktionstest/Modultest
Ergebnis	<ul style="list-style-type: none"> ✓ Programmcode ✓ Dokumentation zum Programmcode ✓ Dokumentation zum Test des Programmcodes

Diese Phase wird **Codierung** genannt. In dieser Phase wird der Programmcode, auch Quelltext oder Quellcode (engl. source code) genannt, erstellt. Dazu werden die Module der Entwurfsphase in einer bestimmten Programmiersprache implementiert.

Bei den Tests wird zwischen sogenannten Whitebox- und Blackbox-Tests unterschieden.

- ✓ Whitebox-Tests prüfen die innere Funktionsweise von Komponenten, z. B. Modulen.
- ✓ Im Gegensatz dazu wird das Innere einer Komponente beim Blackbox-Test nicht betrachtet, sondern das Zusammenspiel der einzelnen Komponenten gemäß der Spezifikation.



Entgegen vielen Erwartungen beansprucht die Phase der Implementierung in der Regel den geringsten Anteil der Gesamtentwicklungszeit.

Integrationsphase

Aufgabe	<ul style="list-style-type: none"> ✓ Zusammenfügen der Einzelaufgaben zur Gesamtaufgabe ✓ Dokumentation
Test	<ul style="list-style-type: none"> ✓ Integrationstest, testet das Zusammenspiel der Einzelteile ✓ Systemtest, der gewährleisten soll, dass das Endprodukt die festgelegten Anforderungen des Pflichtenhefts erfüllt. Stimmt das erstellte Programm mit den Anforderungen der ursprünglichen Beschreibung überein?
Ergebnis	Komplettes System und Dokumentation für den Benutzer



Treten während der Integrationsphase Fehler auf, wird wieder die Implementierungsphase mit nochmaligen Tests durchlaufen. Dieser „Kreislauf“ wird so lange wiederholt, bis alle Tests fehlerfrei durchgeführt werden konnten und somit gravierende Fehler der Software ausgeschlossen werden können.

Einsatz und Wartung

Diese Phasen gehören nicht mehr zum eigentlichen Entwicklungsprozess, sondern umfassen den Zeitraum von der ersten Installation beim Auftraggeber bis zum Einsatzende der Software.

Einsatzphase (Installation)

Aufgabe	<ul style="list-style-type: none"> ✓ Software für die Auslieferung fertigstellen ✓ Beenden der Dokumentation
Test	Akzeptanztest/Abnahmetest – Software wird durch den Kunden geprüft
Ergebnis	Software im Einsatz

Wartungsphase

Aufgabe	<ul style="list-style-type: none"> ✓ Korrektur auftretender Fehler ✓ Anpassung an die Systemumgebung ✓ Änderung oder Erweiterung von Funktionen, die einen zusätzlichen Nutzen bringen würden
Test	Konfigurationstest, prüft z. B. Software nach einer Anpassung
Ergebnis	Änderung oder Erweiterung der Funktionen



Bei fehlender Dokumentation und ungenügender Weitsicht für die Einsatzdauer eines Programms können die Kosten für die Wartung einer Software schnell ansteigen.

12.4 Vorgehensmodelle im Überblick

Für die Darstellung des Software-Lebenszyklus gibt es verschiedene Vorgehensmodelle. Sie unterscheiden sich in

- ✓ den Beziehungen zwischen den einzelnen Phasen,
- ✓ der Anordnung der einzelnen Phasen,
- ✓ der Art und dem Inhalt der einzelnen Phasen,
- ✓ dem betrachteten Projektumfang.

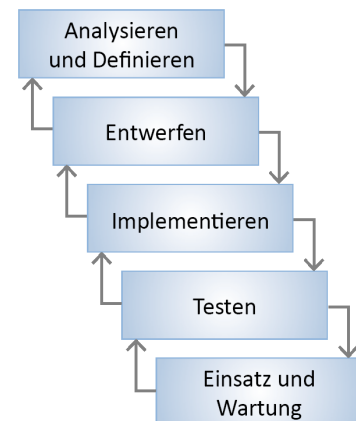
Nachfolgend lernen Sie einige Vorgehensmodelle kennen:

- ✓ das Wasserfallmodell
- ✓ das V-Modell
- ✓ das Prototyping-Modell
- ✓ das Spiralmodell
- ✓ agile Modelle

Das Wasserfallmodell

Das Wasserfallmodell ist ein lineares Modell. Die Phasen sind stufenartig angeordnet. Am Ende jeder Phase steht ein Teilprodukt, das in der nächsten Phase weiterentwickelt wird. Das Modell ist streng sequenziell. Jede Phase besitzt nur eine Rückkopplung zur vorherigen Phase. Diese dient dazu, die bei der Weiterentwicklung gefundenen Schwachstellen aus den vorangegangenen Phasen zu beseitigen. Der Name „Wasserfallmodell“ beruht darauf, dass die Ergebnisse jeder Stufe wasserfallartig auf die nächste Stufe „fallen“.

Die im Wasserfallmodell beschriebene Struktur wurde früher in vielen Projekten verwendet, hat aber durch die mangelnde Flexibilität mittlerweile nicht mehr die Bedeutung vergangener Zeiten. Dennoch gibt es die Vorgehensweise gerade in großen Firmen noch häufiger.



Vorteile	Nachteile
<ul style="list-style-type: none"> ✓ Geringer Managementaufwand ✓ Leicht verständlich 	<ul style="list-style-type: none"> ✓ Späte Änderungen sind nur mit hohem Aufwand realisierbar ✓ Sehr unflexibel durch die strenge Sequenz

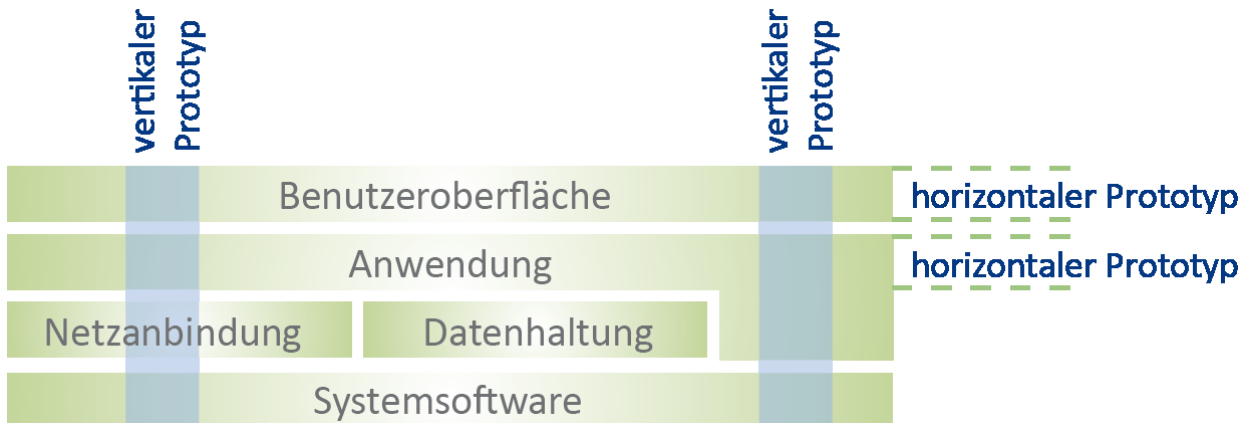
Das V-Modell

Das V-Modell (nach Boehm) kann als eine Weiterentwicklung des Wasserfallmodells betrachtet werden. Die V-Form entsteht dadurch, dass die Phase, die jeweils weiter ins Detail geht, treppenartig versetzt unter der vorhergehenden Phase angeordnet wird. Die Testphase verläuft vom Test des Details hin zum gesamten System – also wieder nach oben. Die Phase und die zugehörige Testphase können als eine Hierarchieebene betrachtet werden. In das Modell sind Validierung (Gültigkeitsprüfung) und Verifikation (Nachweis der Richtigkeit/Korrektheit) integriert.

Es gibt verschiedenen Arten von Prototypen:

- ✓ Demonstrations-Prototyp: wird meist für die Akquisition von Kunden entwickelt, die darüber bereits einen ersten Eindruck von dem späteren Produkt gewinnen (Rapid Prototyping)
- ✓ Labor-Prototyp: wird zur Klärung konstruktionsbezogener und technischer Fragen entwickelt
- ✓ Pilot-Prototyp: bildet den Kern eines Systems, das zum Produkt weiterentwickelt werden kann

Bei der Entwicklung von Prototypen wird unterschieden, ob es sich um einen vertikalen oder einen horizontalen Prototyp handelt. Bei horizontalen Prototypen wird eine Ebene vollständig implementiert, aber ohne Funktionalität. Ein vertikaler Prototyp besitzt die vollständige Funktionalität ausgewählter Teile des Systems.



Vorteile	Nachteile
<ul style="list-style-type: none"> ✓ Entwicklungsrisiko wird reduziert ✓ Prototypen lassen sich in andere Vorgehensmodelle integrieren ✓ Lösungsalternativen werden aufgezeigt ✓ Einbeziehung des Benutzers 	<ul style="list-style-type: none"> ✓ Mitunter zusätzliche Entwicklung und damit erhöhter Entwicklungsaufwand ✓ Häufig keine klaren Beschränkungen für Prototypen

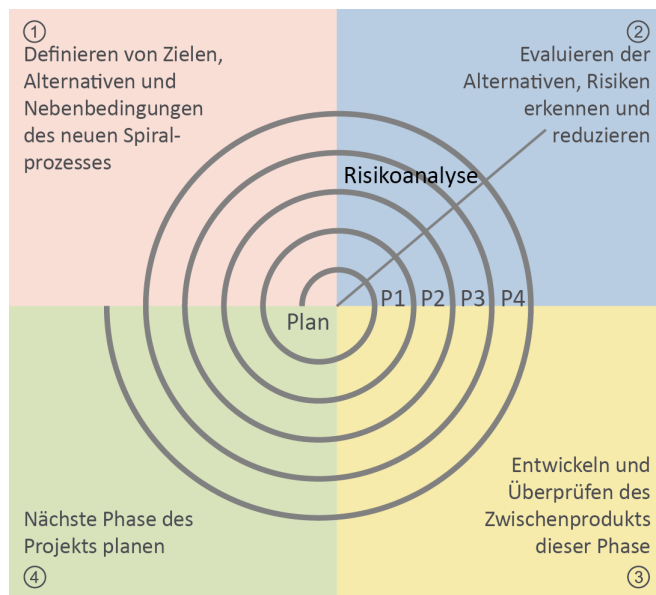
Das Spiralmodell

Das Spiralmodell vereint das klassische lineare Modell (Wasserfallmodell) und evolutionäre Modelle (z. B. Prototyping). Dieses Modell findet heutzutage in sehr vielen Projekten Verwendung.

Die Softwareentwicklung mithilfe des Spiralmodells läuft in vier Schritten ab, die so lange wiederholt werden, bis das Softwareprodukt fertiggestellt ist.

- ① Definition von Zielen, Alternativen und Nebenbedingungen des neuen Spiralprozesses
 - ✓ Zu Beginn jeder Phase (jeder neuen Windung der Spirale) werden inhaltliche Vorgaben für das Teilprodukt definiert.
 - ✓ Es werden alternative Vorgehensweisen ausgearbeitet.
 - ✓ Die Randbedingungen (Nebenbedingungen), wie z. B. Personal, Finanzen und Zeit, sind festzulegen.
- ② Evaluieren der Ziele und Alternativen, Risiken erkennen und reduzieren
 - ✓ Zu Beginn des zweiten Schrittes werden die Alternativen unter Berücksichtigung der Ziele und Randbedingungen evaluiert.
 - ✓ Risikofaktoren werden benannt und so weit wie möglich reduziert.
 - ✓ Es können dazu unterschiedliche Techniken verwendet werden (z. B. Entwicklung von Prototypen). Das Ergebnis ist der Prototyp (P1, P2 ...).

- ③ Entwickeln und Überprüfen des Zwischenprodukts dieser Phase
- ✓ In diesem Schritt wird das Teilprodukt (z. B. einzelne Module oder in der letzten Phase die komplette Software) unter Einhaltung des Ziels und der geplanten Ressourcen realisiert und getestet.
 - ✓ Die Methode für die Realisierung kann flexibel unter Beachtung des Restrisikos gewählt werden.
 - ✓ Die fertig getesteten Module sind beispielsweise Ergebnis dieses Schrittes.
- ④ Nächste Phase des Projekts planen
- ✓ Als letzter Schritt einer Phase wird die nächste Phase inhaltlich und organisatorisch geplant.
 - ✓ Diese Planung kann mehrere Phasen umfassen, sodass unabhängige Teilprojekte entstehen. Diese werden zu einem späteren Zeitpunkt wieder zusammengeführt.
 - ✓ In einem **Review** (Prüfung) werden die Schritte 1 bis 3 analysiert, und es werden Schlussfolgerungen für die weitere Entwicklung gezogen. Sind die technischen oder wirtschaftlichen Risiken einer Projektfortsetzung zu hoch, kann das Projekt abgebrochen werden.
 - ✓ Am Ende der Spirale liegt die fertige Software vor, die bezüglich der Anfangshypothese getestet wird.



Das Spiralmodell

Vorteile	Nachteile
<ul style="list-style-type: none"> ✓ Fehler werden frühzeitig erkannt. ✓ Zwischenprodukte werden regelmäßig überprüft. ✓ Flexibel (Änderungen sind leicht möglich.) ✓ Alternativen werden betrachtet. ✓ Risiken werden minimiert. 	<ul style="list-style-type: none"> ✓ Hoher Managementaufwand ✓ Mitarbeiter benötigen bei Anwendung einer Risikoabschätzung Kenntnisse der Risikoanalyse.

Agile Modelle

Klassische Vorgehensmodelle zeigen in verschiedenen Situationen im Praxiseinsatz auch Schwächen: falls zu Beginn der Softwareerstellung nicht genau bekannt ist, was zu erstellen ist, z. B. wegen der Ungewissheit unternehmenspolitischer Entscheidungen, oder falls sich während der Erstellung der Software Änderungen ergeben, z. B. Markteinbrüche.

Um diese Probleme zu umgehen, wurden sogenannte **agile Modelle** entwickelt. Diese Vorgehensmodelle versuchen mit flexiblen Prozessen, Software in kurzer Zeit in kleinen Teams unter Einbeziehung des Kunden zu entwickeln. Die Software kann bei Bedarf nach und nach ausgebaut werden.

Agile Programmierung stellt folgende Leitlinien in den Fokus:

- ✓ Individuen und Interaktion bzw. Kommunikation sind wichtiger als Prozesse und Werkzeuge.
- ✓ Lauffähige Software ist wichtiger als umfangreiche Dokumentation.
- ✓ Teamwork und Zusammenarbeit mit dem Kunden ist wichtiger als Vertragsverhandlungen.
- ✓ Auf Änderungen reagieren ist wichtiger, als einem vorher festgelegten Plan zu folgen.

Positive Eigenschaften agiler Methoden sind folgende Kriterien:

- ✓ Einfach einzuführen und zu kontrollieren, wenig Administration, wenig Dokumente
- ✓ Schnelle Auslieferung von Funktionen, Analyse- und Designphase im Lauf des Projekts
- ✓ Informelle Kommunikation und gemeinsame Entscheidungen

Es gibt aber auch einige Nachteile:

- ✓ Die Ergebnisse hängen stark von individuellen Fähigkeiten und Motivationen sowie der Disziplin der Mitarbeiter ab.
- ✓ Durch fehlende Strukturen kann es bei großen, sicherheitskritischen Projekten Probleme geben.
- ✓ Das Einbeziehen externer Ressourcen und Mitarbeiter ist schwierig, da wenige formale Rahmen vorhanden sind.
- ✓ Es besteht durch die fehlenden formalen Rahmen die Gefahr von Mehraufwand durch missverständliche Kommunikation oder verspätete Entscheidungen.

12.5 Computergestützte Softwareentwicklung (CASE)

Was ist CASE?

CASE ist die Abkürzung für Computer Aided Software Engineering – computergestützte Softwareentwicklung. Bei der Entwicklung von Software werden computergestützte Hilfsmittel eingesetzt mit dem Ziel, die Produktivität und die Qualität der Software zu verbessern und das Management zu unterstützen. Die computergestützten Hilfsmittel – die CASE-Plattform – bestehen aus einer CASE-Umgebung und CASE-Werkzeugen, die in die CASE-Umgebung integriert sind. CASE-Werkzeuge sind z. B.

- ✓ grafische Editoren, in denen Sie Ihre Softwareprojekte mithilfe verschiedener Vorgehensmodelle gliedern und bearbeiten können;
- ✓ Testfunktionen, um erzeugte Modelle auf Richtigkeit zu überprüfen;
- ✓ Werkzeuge zur Fehlersuche;
- ✓ die automatische Quelltexterstellung in verschiedenen Programmiersprachen.

Beispiele für CASE-Werkzeuge:

- ✓ Eclipse (Eclipse Foundation)
- ✓ VisualStudio (Microsoft)
- ✓ PsPad
- ✓ IDLE
- ✓ Modellierungsprogramme zum Erstellen von Diagrammen

12.6 Fehler finden und identifizieren

Fehler in Programmen sind ein leidiges Thema, weil unvermeidbar. Falls ein Programm Fehler enthält, müssen diese gefunden und beseitigt werden, bevor das Projekt freigegeben werden kann. Aber auch schon während der eigentlichen Erstellungsphasen müssen diese beseitigt werden, wenn sie entdeckt werden. Nachfolgend konzentrieren wir uns auf die Fehler, die in der Implementierungsphase auftreten können.

Welche Fehler gibt es?

Es gibt verschiedene Situationen, die Fehler bewirken:

- ✓ Fehler zur Laufzeit, welche auf äußere Umstände (Situationen, die erst zur Laufzeit entstehen, die Umgebung des Programms oder den Anwender) zurückzuführen sind
- ✓ Typografische Fehler beim Schreiben des Programms
- ✓ Syntaktische Fehler beim Schreiben des Programms
- ✓ Fehler zur Laufzeit, welche auf logische Fehler im Programmaufbau zurückzuführen sind

Die Beseitigung der Fehlerpunkte 2 bis 4 ist hauptsächlich das, was man unter **Debugging** versteht. Der Begriff Debugging geht angeblich auf das englische Wort Bug zurück, was übersetzt Käfer oder Wanze heißt. Es gibt verschiedene Anekdoten, woher der Name für die Fehlersuche kommt. Die bekannteste Anekdote hat zum Inhalt, dass in Zeiten von Röhrengroßrechnern ein Programmfehler auf einen toten Käfer zwischen den heißen Röhren zurückzuführen war, der mit dem Körper Schaltkreise störte und damit das Programm unerwartet ablaufen ließ.

Punkt 1 zählt teilweise ebenfalls dazu, was man Debuggen nennt, wird jedoch bei modernen Programmen weitgehend mit einem Mechanismus behandelt, der **Ausnahmebehandlung** genannt wird.

Fehler zur Laufzeit und Ausnahmebehandlung

Unter einer Ausnahme (Exception) kann man eine Unterbrechung des normalen Programmablaufs aufgrund einer besonderen Situation verstehen, die eine isolierte und unverzügliche Behandlung notwendig macht. Der normale Programmablauf kann erst fortgesetzt werden, wenn diese Ausnahme behandelt wurde. Andernfalls wird das Programm beendet.

Ein klassisches Beispiel, das eine solche Situation provoziert, ist der Versuch, auf einen Wechseldatenträger zuzugreifen, in dem kein Medium eingelegt oder kein Speicherplatz mehr vorhanden ist. Natürlich kann man vor einem Zugriff auf einen Wechseldatenträger überprüfen, ob ein Medium eingelegt ist oder genügend Speicherplatz zur Verfügung steht.

Aber nicht jede Situation kann man so prophylaktisch abfangen. Denn viele potenzielle Probleme kann man auch gar nicht so leicht erkennen. Dann ist das „Werfen“ einer Ausnahme im Fall eines Problems sehr sinnvoll. In einer objektorientierten Sprache wie Python ist eine Ausnahme aber auch ein Mitteilungsobjekt. Es gibt Informationen zurück, welcher Fehler genau vorliegt, und aufgrund dieser Information kann der Programmierer Gegenmaßnahmen ergreifen. Weil das Auftreten einer Ausnahme ohne Gegenmaßnahmen ein Programm beendet, wird diese Reaktion meist sogar erzwungen.

Ein Ausnahmekonzept ist aber nicht unabdingbar. Immerhin hatten ältere Programmiersprachen so ein Konzept ja nicht implementiert. Man kann auch mit selbst programmierten Kontrollmechanismen jeden Fehler in einem Programm abfangen, den Sie als potenzielle Fehlerquelle erkennen. Aber da haben wir den entscheidenden Schwachpunkt – Sie müssen die potenzielle Fehlerquelle erkennen, was oft nahezu unmöglich ist. Gerade in komplexen Programmen können Sie nie alle kritischen Zusammenhänge überblicken. Es werden immer Fehlerquellen da sein, die an kaum beachteten Stellen auftreten. Außerdem ist es sehr viel Arbeit, bei jeder Anweisung im Quelltext zu überlegen, ob da eine Gefahr lauert, und diese dann sicher abzufangen. Es ist sogar so, dass Sie in der überwiegenden Anzahl von auffangbaren Fehlern das Rad neu erfinden, denn es gibt sehr oft schon Standardmaßnahmen. Zu guter Letzt setzt diese individuelle Technik der Laufzeitfehlerbehandlung oft voraus, dass der Fehler selbst nicht eintritt, sondern vorher erkannt und umgelenkt wird.

Typografische Fehler

Wenn Sie sich beim Schreiben des Quellcodes vertippt oder ein Schlüsselwort, eine Variable oder sonst etwas Entscheidendes falsch geschrieben haben, können im Wesentlichen zwei Fälle auftreten.

- ✓ Sofern Sie ein Wort komplett falsch geschrieben haben, ist die Situation oft leicht in den Griff zu bekommen. Wenn der Schreibfehler zu keinem vernünftigen Ausdruck (Token) führt, wird der Ausdruck in der Regel einen syntaktischen Widerspruch innerhalb des Programms erzeugen, und das System wird bei seiner Arbeit den Fehler entdecken. Das System gibt meist eine Information zur Art des Fehlers und eine Stelle zurück, wie Sie den Fehler finden können. Falls Sie direkt an dieser Adressangabe den Fehler finden, haben Sie allerdings Glück, denn leider ist diese Adressangabe nur die Stelle, wo sich ein Fehler auswirkt, und nicht unbedingt der Entstehungsort. Dies muss nicht immer identisch sein, aber zumindest ist der Fehler schon einmal eingegrenzt.
- ✓ Schlimmer ist, wenn Sie zufällig einen solchen Schreibfehler gemacht haben, sodass der falsche Begriff wieder einen sinnvollen Token und keinen syntaktischen Widerspruch ergibt. Denn das wird ein viel ernsteres Problem. Nehmen wir ein Beispiel aus der normalen Sprache. Sie wollten beispielsweise „lehr“ schreiben und haben „leer“ getippt – ein ebenfalls sinnvolles Wort. Die Rechtschreibprüfung Ihrer Textverarbeitung wird Ihnen beispielsweise den Fehler nicht anzeigen. Und in einem analogen Fall würde das System bei der Programmierung kein Problem melden, wenn Sie nur Token vertauschen und sich kein syntaktischer Widerspruch ergibt.

Betrachten Sie folgendes Fragment:

Beispiel: *Fehler.py*

```
meineVariable = 42;  
mieneVariable = 55;  
print(meineVariable)
```

Die Ausgabe ist das:

42

Das Programm zeigt eine syntaktisch fehlerfreie Python-Struktur. Geplant soll sein, dass eine Variable `meineVariable` eingeführt, im Wert geändert und dann ausgegeben wird. Nur gibt es bei der zweiten Anweisung einen Buchstabendreher im Bezeichner der Variablen. Durch die lose Typisierung wird das aber zu keinem Fehler führen. Es wird nur einfach temporär **eine neue Variable** `mieneVariable` erzeugt und der Wert der eigentlich zu ändernden Variablen `meineVariable` nicht geändert. Und bei der Ausgabe wundern Sie sich dann. Oder noch schlimmer – Sie wundern sich nicht und der Fehler wird gar nicht bemerkt!

In solch einem Fall haben Sie ohne Hilfsmittel nur die Chance, dass Sie den Fehler im Quelltext durch aufmerksames Lesen oder Analyse des Resultats finden (sehr schwer – lassen Sie sich nicht durch dieses extrem einfache Exempel täuschen). Oder aber Sie greifen auf Testausgaben oder gar einen Debugger zurück (s. u.).

Syntaktische Fehler

Bei syntaktischen Fehlern in einem Programm (Fehler bei der Verwendung der Sprachsyntax) wird der Compiler oder Interpreter in der Regel die Abarbeitung unterbrechen. Viele syntaktische Fehler beruhen auf typografischen Fehlern, aber beileibe nicht alle. Denn dieser Fehlertyp beinhaltet auch die falsche Verwendung von korrekten Ausdrücken oder Token (beispielsweise so etwas wie `while (i == 0)` – ein Vergleich statt einer Wertzuweisung der Zählvariable).

Typische syntaktische Fehler sind folgende:

- | | |
|--|--------------------------------------|
| ✓ Falsche Schreibweise von Bezeichnern | ✓ Falsche oder fehlende Kommasetzung |
| ✓ Falsches oder fehlendes Semikolon | ✓ Falsche Klammerung |
| ✓ Falsche Wahrheitswerte | ✓ Falsche Datentypen |

Logische Laufzeitfehler

Es gibt auch Programmierfehler, die sich erst zur Laufzeit durch ein fehlerhaftes Verhalten des Programms auswirken, unter Umständen sogar nur in einer ganz spezifischen Konstellation. Solche Fehler werden auf logische Unzulänglichkeiten im Programmaufbau zurückzuführen sein. Hierbei haben Sie ein syntaktisch vollkommen korrektes Programm geschrieben, das aber Fehler zur Laufzeit zulässt.

Sie müssen zur Beseitigung der Fehlersituation die Stelle im Quelltext suchen, an der dieses Fehlverhalten auftritt. Dort müssen Sie dann versuchen, den Aufbau und mögliche Fehler noch einmal durchzuspielen.

Beispiel: Logische Fehler

Sie haben ein Programm geschrieben, das einen Ausdruck berechnen soll. Die Programmausführung liefert jedoch nicht das gewünschte Ergebnis. Der Fehler wird hier durch fehlende Klammern hervorgerufen. Da das Programm syntaktisch richtig ist, wird beim Übersetzen keine Fehlermeldung erzeugt.

Statt

<code>Ergebnis = (a+b) * (a / (b-a)) ;</code>	gewünschte Klammerung
---	-----------------------

haben Sie

<code>Ergebnis = (a+b) * (a/b-a) ;</code>	falsche Klammerung
---	--------------------

eingegeben. Da bei der Reihenfolge der Berechnung die mathematischen Grundregeln (Punkt vor Strich) gelten, wird in der zweiten Klammer zuerst der Bruch a/b berechnet und vom Ergebnis wird a subtrahiert. In der gewünschten Berechnung wird durch Klammerung zuerst die Berechnung der Differenz erzwungen, bevor der Bruch berechnet wird.

Mit dem Debugger können Sie die Werte von a und b auslesen. Durch die Auswertung der Berechnungen innerhalb der Klammern werden Sie den logischen Fehler schnell finden.



Führen Sie deshalb stets nach erfolgreicher Kompilierung und Ausführung eines Programms Tests mit Beispieldaten durch, um derartige Fehler auszuschließen und die Funktionsweise Ihres Programms zu überprüfen.

Vorbeugen statt heilen

Fehler beim Programmieren können nie vollständig verhindert werden. Selbst Profis unterlaufen permanent Fehler. Aber man kann die Anzahl der Fehler beim Kodieren reduzieren und gleichzeitig die Wartbarkeit von Projekten im Griff halten. Wartbarkeit und Reduktion der Fehler beim Kodieren gehen direkt einher. Diese Maßnahmen können auch Einsteiger gleich von Anfang an berücksichtigen, auch bei einfachen Projekten und Quellcodes, um sich gleich einen guten Programmierstil anzugewöhnen.



Wartung und Weiterentwicklung von professionellen Projekten nimmt bis zu 90 % des Aufwands ein, der über die Laufzeit in ein Projekt investiert wird. Damit wird deutlich, wie wichtig ein gut wartbarer Code werden kann.

- ✓ Um keine unnötigen Fehler bei der Erstellung eines Programms entstehen zu lassen, hilft zum Beispiel schon ganz banal eine akkurate Vorbereitung. Dies beginnt mit der Programmplanung und Vorüberlegungen zum Programmablauf (eventuell mit Ablaufdiagrammen und der Notation der geplanten Struktur und des Ziels der Applikation auf Papier) und geht hin zum Bereitlegen von Nachschlagewerken und genügend Infomaterial. In der professionellen Programmierung benötigt man auch für die Planung und Konzeption oft ein Vielfaches der Zeit, die letztendlich die konkrete Codeerstellung dauert. Oder anders ausgedrückt: Bevor die erste Codezeile erstellt wird, ist oft die wichtigste Arbeit schon gelaufen. So weit brauchen Sie nicht zu gehen, aber Sie sollten die Vorbereitung nicht links liegen lassen – nicht nur wegen der Reduzierung von Fehlern.
- ✓ Ganz wichtig sind vernünftige Entwicklungswerkzeuge mit diversen Features (CASE-Tools).

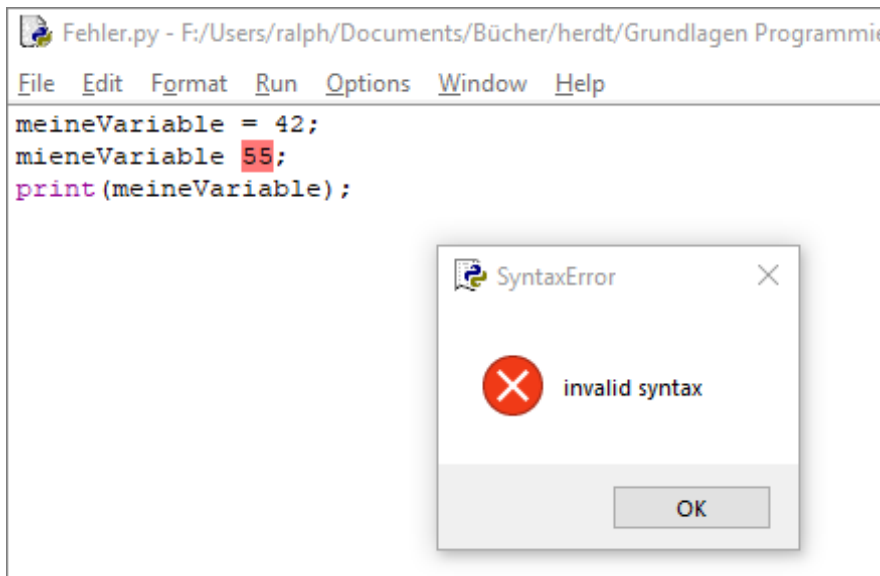
- ✓ Es ist immer viel weniger fehlerträchtig, wenn man bereits fehlerfreien Code einfach kopieren kann. Entweder aus den Vorschlägen der IDE oder Passagen, die Sie bereits ohne Fehler fertig haben oder die Ihnen sonst zur Verfügung stehen – das ist in Vollendung das schon angesprochene DRY-Prinzip (Don't repeat yourself).
- ✓ Das Einhalten von Konventionen, klare, übersichtliche Formatierungen und die Einhaltung des KISS-Prinzips erhöhen nicht nur die Wartbarkeit, sondern vermindern die Fehlerhäufigkeit.
- ✓ Nutzen Sie weiter beim Erstellen Ihrer Quellcodes ausführliche Kommentare, die Sie an den unterschiedlichsten Stellen im Quelltext notieren. Erstens erleichtert es die spätere Wartung, und zweitens machen Sie sich beim Ausformulieren eines Kommentars noch einmal Gedanken über die Quelltextstruktur. Dabei fallen oft auch Fehler auf.

Fehler suchen

Wenn nun in Ihrem Programm trotz sorgfältiger Arbeit ein Fehler vorhanden ist, bleibt nur die Fehlerbeseitigung. Dabei stellt sich zuerst das Problem, den Fehler überhaupt zu lokalisieren. Also geht es eigentlich um die Suche nach dem Fehler. Dies ist der größte und wichtigste Teil dessen, was Debugging genannt wird – das Lokalisieren und Identifizieren eines Fehlers. Der meist einfachere Teil ist die dann folgende Beseitigung des Fehlers.

Man kann auch ohne Hilfsmittel erfolgreich auf die Jagd nach Käfern gehen. Das funktioniert ganz gut, wenn man gewisse Konzepte zur Suche verwendet. Die Fehlersuche ohne Debugger hat die wesentlichen Vorteile, dass Sie außer dem Editor und dem Browser kein zusätzliches Programm benötigen und sich vor allem nicht in dessen Bedienung einarbeiten müssen. Zudem lernen Sie eine Menge über Programmierung.

Dabei helfen die Fehlermeldungen der Programmierungsumgebung. Diese können schon vor der Übersetzung vorhanden sein.



Bereits vor der Übersetzung kann der Fehler lokalisiert werden

Oder aber es wird versucht, das Programm auszuführen, und dann kommt es zu einem Fehler.

```
RESTART: F:/Users/ralph/Documents/Bücher/herdt/Grundlagen Programmierung Python
/Quellcodes/Kap12/Fehler.py
Traceback (most recent call last):
  File "F:/Users/ralph/Documents/Bücher/herdt/Grundlagen Programmierung Python/Q
uellcodes/Kap12/Fehler.py", line 3, in <module>
    print(meineVariable/8);
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>> |
```

In der Python-Konsole sieht man eine Fehlermeldung mit einer Zeilenangabe, wo der Fehler sich auswirkt

Sie sehen in solchen Fällen in der Regel eine Information, wo die Fehlerstelle ist.



Die Stelle, wo sich ein Fehler auswirkt, muss wie schon erwähnt nicht die Stelle sein, wo Sie einen Fehler machen. Das gilt vor allen Dingen im Interpreter-Fall. Wenn Sie mit einem Auto aus einer Kurve fliegen, unterscheiden sich auch die Stelle, an der Sie den Fehler machen, und die Stelle, wo er sich auswirkt (wenn man dann wieder irgendwo aufschlägt). Sie werden an der Fehlermeldung meist nur die Stelle erkennen, wo ein Fehler zum Tragen kommt. Man kann aber davon ausgehen, dass Sie irgendwo vorher im Programm einen Fehler gemacht haben.

Kontrollausgaben

Der klassische Weg, um nicht offensichtliche Fehler ohne einen Debugger zu finden, sind jedoch Kontrollausgaben, welche im endgültigen Quellcode beseitigt werden. Gängige Praxis ist, vor einem vermuteten Fehler eine Bildschirmausgabe zu erzeugen. Man kann entweder den Wert einer Variablen vermerken, in der man die Fehlerursache vermutet, oder man notiert einfach eine Meldung, um zu sehen, ob das Programm eine bestimmte Stelle erreicht.

Man kann selbstverständlich an den unterschiedlichsten Stellen im Programm solche Testausgaben einfügen, um eine Variable über mehrere Schritte hinweg zu verfolgen, oder um zu sehen, wie weit ein Programm läuft. Geben Sie einfach an verschiedenen Stellen im Programm eindeutige Testausgaben aus (beispielsweise Zahlen). Sobald eine Testausgabe nicht mehr erscheint, muss davor ein Fehler vorliegen. Dann suchen Sie von der Stelle der letzten erfolgreichen Testausgabe bis zu der Stelle, wo die Testausgabe unterdrückt wurde.

Teile des Quellcodes auskommentieren

Eine andere Technik zur Fehlerlokalisierung beruht darauf, eine größere Menge an Anweisungen, in denen man die Wanze vermutet, auszukommentieren. Hat ein Programm einen Fehler und läuft nach der Auskommentierung (natürlich ohne den auskommentierten Part), muss sich der Fehler in den auskommentierten Anweisungen befinden. Nun verkleinert man Schritt für Schritt (sinnvoll) den auskommentierten Bereich und testet nach jeder Verkleinerung das Programm. Nach dem Schritt, nachdem das Programm nicht mehr läuft, hat man die fehlerhafte Anweisung mit hoher Wahrscheinlichkeit lokalisiert.

Fehlersuche mit einem Debugger

Ein spezielles Programm bzw. Tool zum Auffinden von Fehlern wird **Debugger** genannt. Debugger gibt es für nahezu alle Programmiersprachen und für viele Programmsprachen – auch für Python über IDLE.

Mit einem Debugger können Sie beispielsweise die Programmausführung an einer definierten Stelle unterbrechen, indem Sie sogenannte **Haltepunkte** (Breakpoints) im Programm setzen. An den Haltepunkten können Sie z. B. Überprüfungen im Quellcode vornehmen, den Quellcode ändern, den Quellcode Zeile für Zeile weiterlaufen lassen und dabei bestimmte Teile des Quellcodes beobachten.



Ein Werkzeug wie ein Debugger ist äußerst flexibel. So werden Sie hier nur einige wichtige Möglichkeiten vorgestellt bekommen.

Die grundsätzliche Arbeit mit einem Debugger

Zuerst soll schematisch die Arbeit mit einem Debugger durchgesprochen werden. Alle Debugger funktionieren in etwa ähnlich. Man kann beispielsweise ein Programm

- ✓ schrittweise verfolgen (sogenanntes Steppen),
- ✓ es gezielt anhalten (mit Haltepunkten beziehungsweise Breakpoints) oder
- ✓ einzelne Variablen und den Zustand des Programms auswerten.

In leistungsfähigen Tools können Sie, auch während das Programm unterbrochen ist, einen Wert in einem Ausdruck manuell ändern. Innerhalb eines Debuggers können Sie in der Regel durch verschiedene Schaltflächen bzw. Menübefehle den Ablauf des Programms verfolgen und kontrollieren. Als Werkzeuge bietet das Programm unter anderem eine Übersicht der aktuell geöffneten Dokumente und eine Möglichkeit, Befehle einzugeben.

Wir wollen einige elementare Arbeitsschritte mit einem Debugger an einem konkreten Beispiel durchspielen. Als erste Beispieldatei zum Umgang mit dem Debugger verwenden wir folgendes Listing, das bewusst eine Problemstellung enthält.

Beispiel: *ZumDebuggen.py*

```
i = 1;
while i != 10:
    print(i)
    i += i
```

Wenn das Programm ausgeführt wird, wird eine Endlosschleife erzeugt.

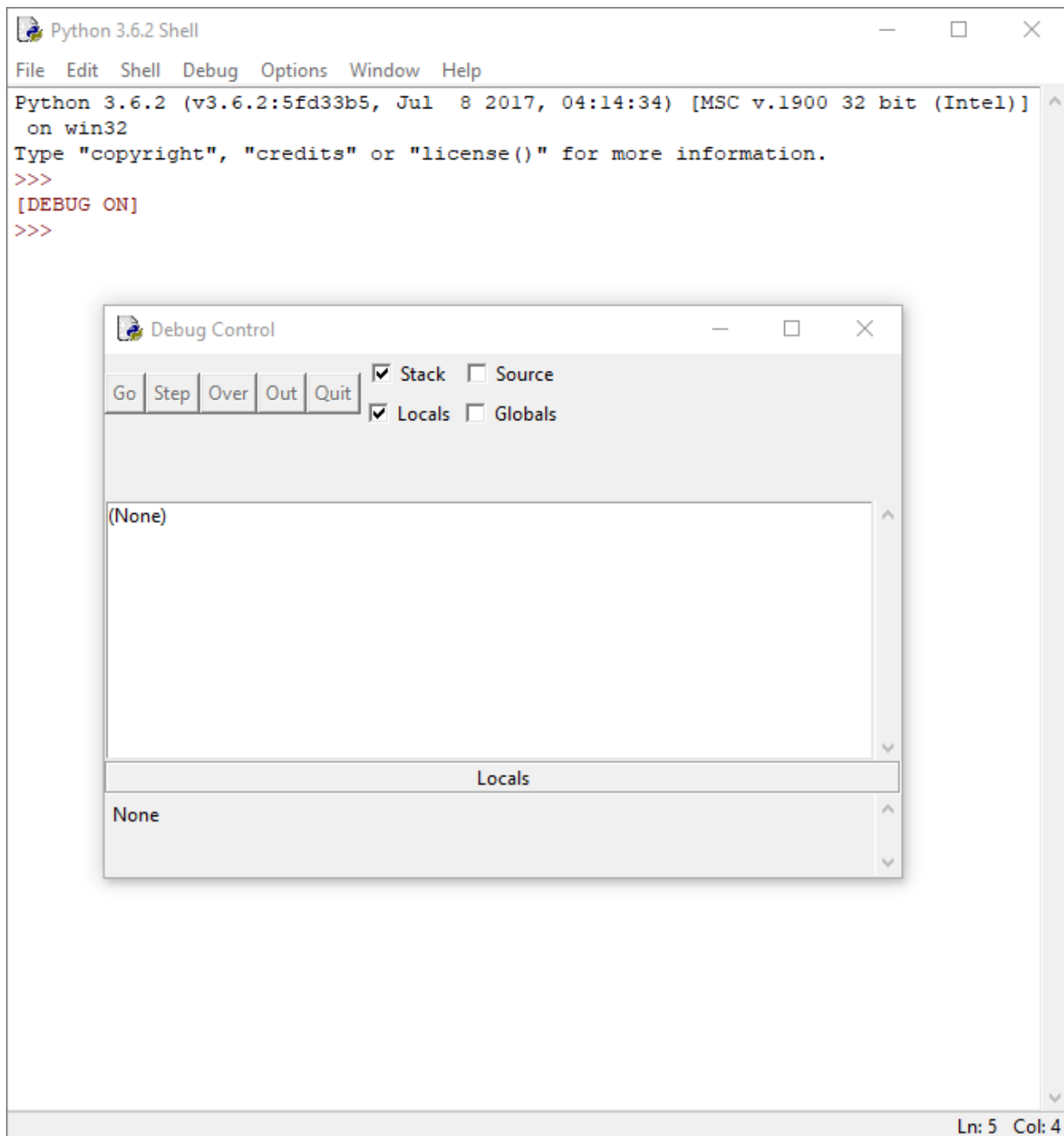
```
*Python 3.6.2 Shell*
File Edit Shell Debug Options Window Help
47175259353065980237219922126478577866076726993401113182069890109075783062762894
02919570188543408116024591824535090515659685981964548006537827248489356756234478
54186595085924618012738712361848919937493876073507331472922728084353530828042821
901750088645899833896337408
86002821250588657563255943553879586660813849504254630951541783780836928846845024
65205593809798850276703340160867881397988379883110023886054225135564376859512591
94841500993383711348363849824784478200230511953188242758694262362517763971154602
94350518706131960474439844252957155732153453986802226364139780218151566125525788
05839140377086816232049183649070181031319371963929096013075654496978713512468957
08373190171849236025477424723697839874987752147014662945845456168707061656085643
803500177291799667792674816
17200564250117731512651188710775917332162769900850926190308356756167385769369004
93041118761959770055340668032173576279597675976622004777210845027112875371902518
38968300198676742269672769964956895640046102390637648551738852472503552794230920
58870103741226392094887968850591431146430690797360445272827956043630313225105157
61167828075417363246409836729814036206263874392785819202615130899395742702493791
41674638034369847205095484944739567974997550429402932589169091233741412331217128
7607000354583599335585349632
34401128500235463025302377421551834664325539801701852380616713512334771538738009
86082237523919540110681336064347152559195351953244009554421690054225750743805036
77936600397353484539345539929913791280092204781275297103477704945007105588461841
17740207482452784189775937701182862292861381594720890545655912087260626450210315
22335656150834726492819673459628072412527748785571638405230261798791485404987582
83349276068739694410190969889479135949995100858805865178338182467482824662434257
5214000709167198671170699264
68802257000470926050604754843103669328651079603403704761233427024669543077476019
72164475047839080221362672128694305118390703906488019108843380108451501487610073
55873200794706969078691079859827582560184409562550594206955409890014211176923682
35480414964905568379551875402365724585722763189441781091311824174521252900420630
44671312301669452985639346919256144825055497571143276810460523597582970809975165
66698552137479388820381939778958271899990201717611730356676364934965649324868515
0428001418334397342341398528
13760451400094185210120950968620733865730215920680740952246685404933908615495203
94432895009567816044272534425738861023678140781297603821768676021690300297522014
71174640158941393815738215971965516512036881912510118841391081978002842235384736
47096082992981113675910375080473144917144552637888356218262364834904250580084126
08934262460333890597127869383851228965011099514228655362092104719516594161995033
13339710427495877764076387955791654379998040343522346071335272986993129864973703
00856002836668794684682797056
Ln: 1616 Col: 372
```

Das Programm wird nicht abgebrochen

Der Quellcode enthält im Programmbereich wie gesagt ein Problem. Die Abbruchbedingung der Schleife wird niemals `True` ergeben, denn der überprüfte Wert der Variable `i` wird den Wert 10 niemals exakt erreichen. Er ist entweder kleiner oder größer. Das ist ein klassischer logischer Fehler.

Das Auffinden mit einem Debugger

- ✓ Öffnen Sie aus IDLE heraus die Python Shell.
- ✓ Klicken Sie den Menübefehl *Debug* und dort *Debugger* an. Das Fenster *Debug Control* wird angezeigt. Darüber können Sie das Debuggen steuern.
- ✓ In der Python Shell sehen Sie die Meldung `[DEBUG ON]`, was zudem anzeigt, dass sich IDLE jetzt im Debug-Modus befindet.

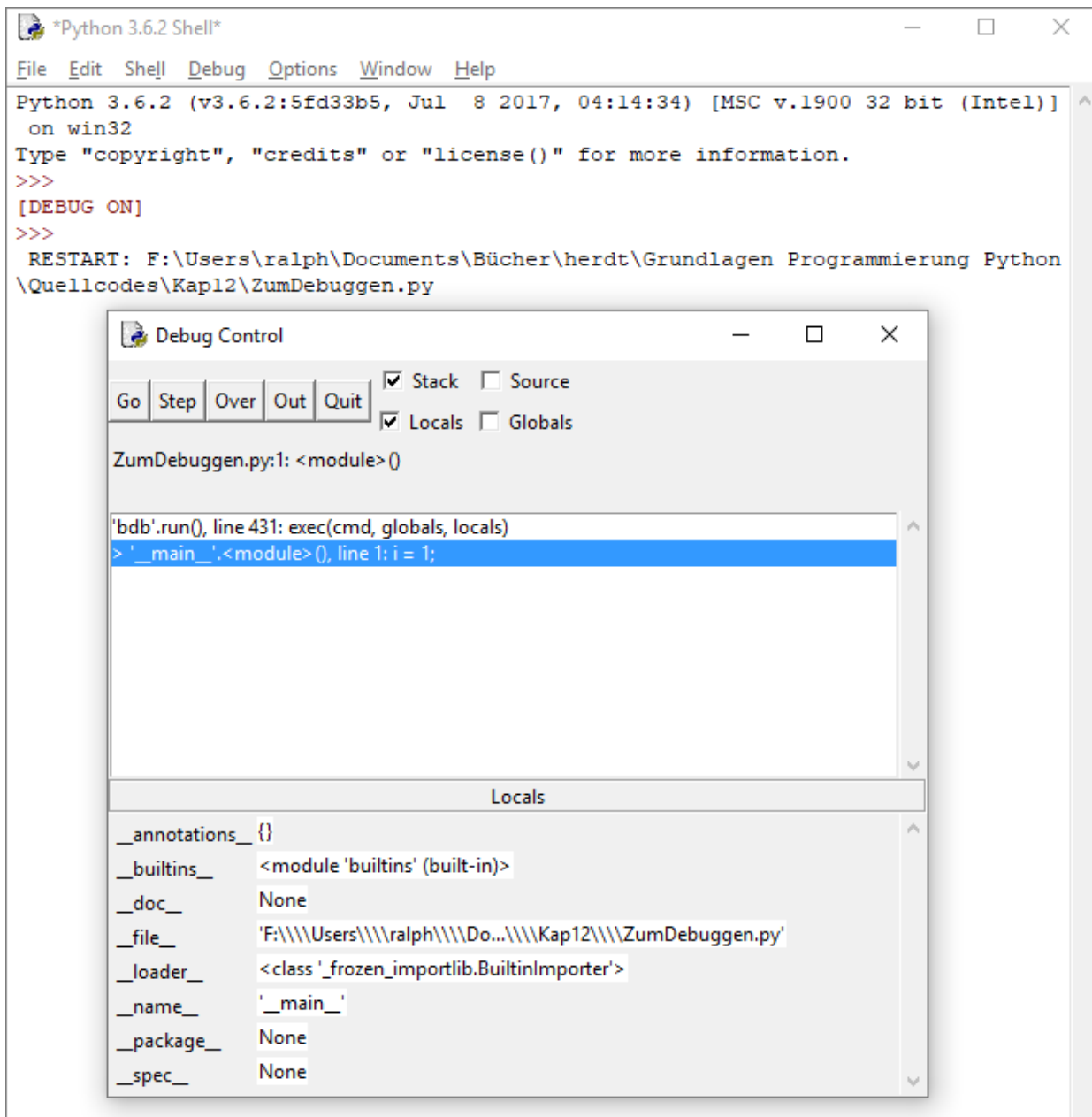


Das Fenster Debug Control – Python ist im Debug-Modus

Man kann in IDLE mit dem Kontextmenü einen Haltepunkt (Breakpoint) setzen, aber das ist in dem Fall nicht zwingend notwendig. Denn wenn der Debug-Modus eingeschaltet ist, wird die Ausführung des Programms unmittelbar nach der ersten Anweisung angehalten, wenn Sie das Programm aus IDLE heraus ausführen.



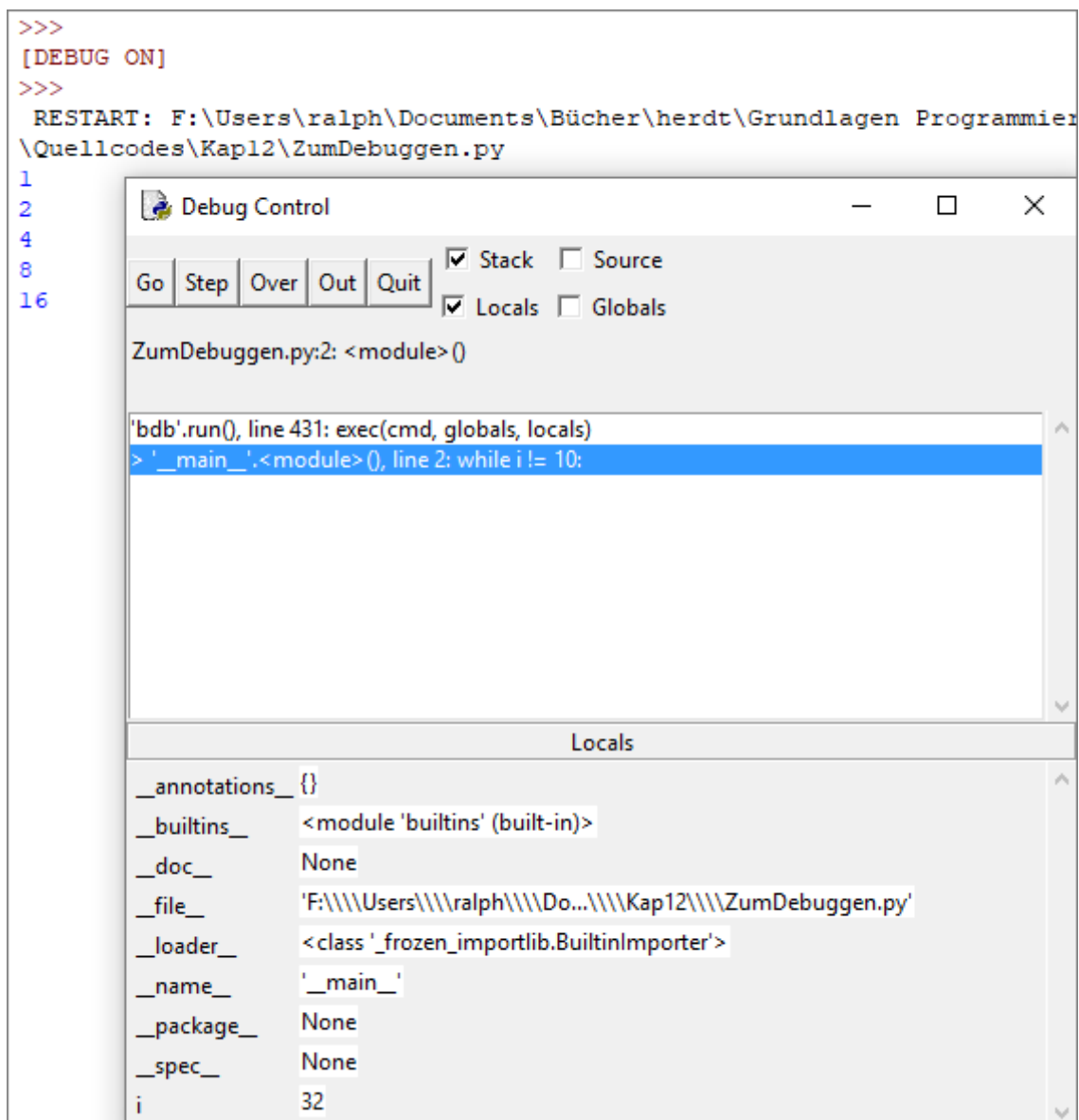
- ✓ Starten Sie wie üblich in IDLE das Programm erneut.
- ✓ Im Debug-Fenster erkennt man diverse Informationen zum Zustand des Programms zu dem Zeitpunkt, u. a. auch den Wert der Variable `i`.



Das Fenster Debug Control

Mit den Schaltflächen oben im Debug Control-Fenster können Sie nun die beschriebenen Schritte beim Debuggen durchführen:

- ✓ Zu einer Stelle gehen (Go)
- ✓ Durch jede Zeile des Programmcodes vorwärtsgehen (Step). Damit gehen Sie jedoch bei einem Funktionsaufruf in die Deklaration der Funktion hinein.
- ✓ Zu jeder neuen Anweisung springen und Unterschlüsse übergehen (Over). Damit arbeiten Sie die folgenden Anweisungen des Programms ab, ohne in die Deklaration von aufgerufenen Funktionen hineinzugehen.
- ✓ Aus einer Unterstruktur herausgehen (Out)
- ✓ Die Programmausführung abbrechen (Quit)



Analyse im Fenster Debug Control

Wenn Sie nun durch den Quellcode mit der Befehlsschaltfläche *Over* voranschreiten, arbeiten Sie die Schleife Schritt für Schritt ab und erkennen, dass der Wert von `i` niemals den Wert 10 annimmt. Sie haben den Fehler gefunden. Aber es liegt an Ihnen, diese Information richtig zu interpretieren und die geeigneten Fehlerbeseitigungsmaßnahmen zu finden.

12.7 Schnellübersicht

Was bedeutet ...?	
Modul/Komponente	Dies ist ein abgeschlossener Baustein einer Software. Eine Komponente kann auch ein eigenständiges Programm sein.
Software-Lebenszyklus	Gliedert die Entwicklung der Software in einzelne Phasen. In jeder Phase wird eine bestimmte Aufgabe erledigt, getestet und dokumentiert.
Vorgehensmodell	Modell für den Planungs- und Entwicklungsprozess von Software
CASE	Computer Aided Software Engineering , computergestützter Softwareentwurf mit CASE-Werkzeugen
Debugger	Ein Programm zur Analyse eines Programms zur Laufzeit, das zur Suche nach Fehlern genutzt werden kann

12.8 Übung

Fragen zur Softwareentwicklung

Übungsdatei: --

Ergebnisdatei: *uebung12.pdf*

1. Benennen Sie die einzelnen Teilphasen des Software-Lebenszyklus.
2. Welche Angaben sind in einem Pflichtenheft festzuhalten?
3. Welche Vorgehensmodelle kennen Sie? Wodurch unterscheiden sie sich?
4. Welche Arten von Dokumentationen für Software gibt es? Warum ist die Dokumentation von so großer Bedeutung?
5. Was bedeutet CASE, und wozu dient es?
6. Was bedeutet „Steppen“?

Anhang A: PAP, Struktogramm und Pseudocode

A.1 Beispiel Zinsberechnung

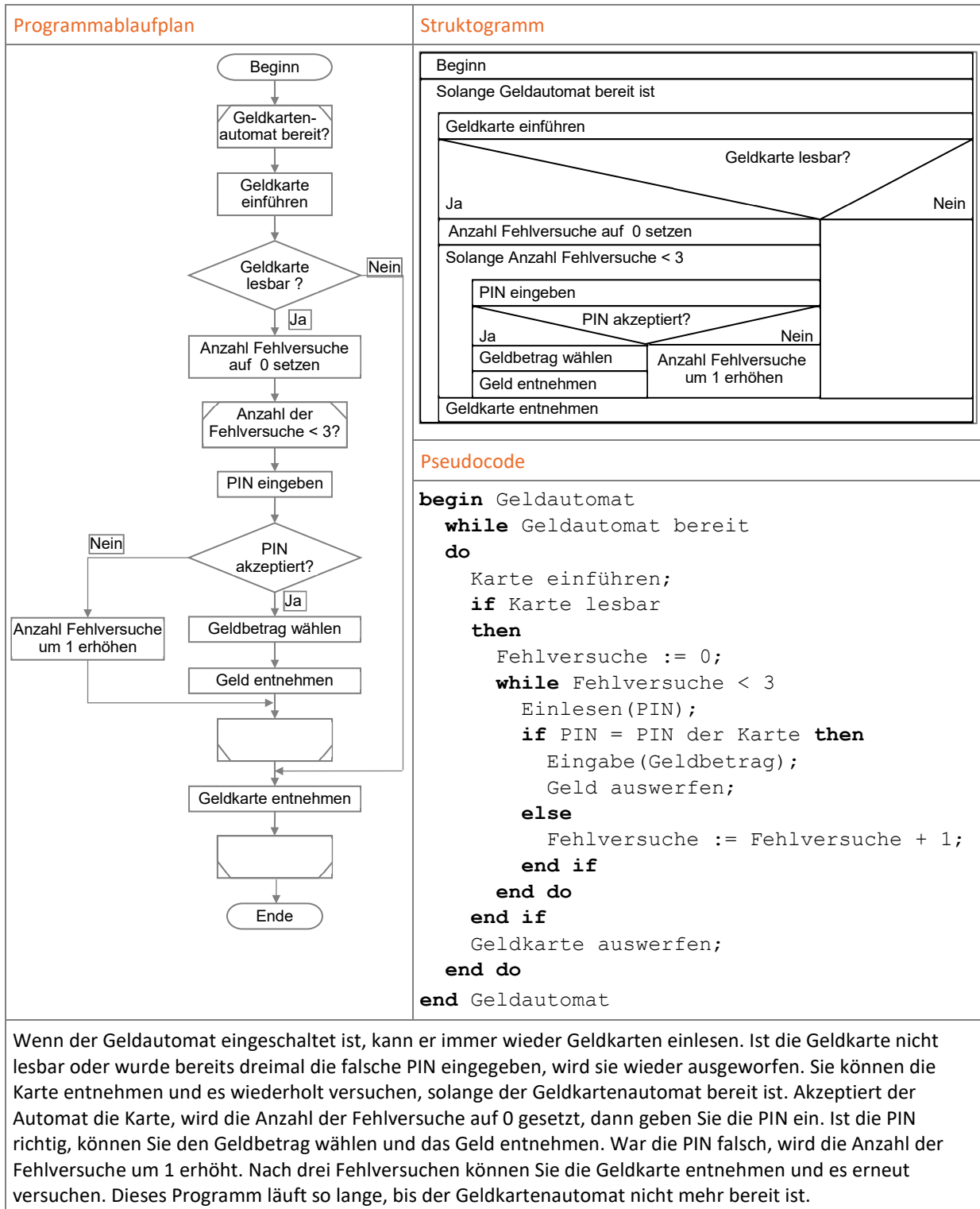
Bei Algorithmen wie der Zinsberechnung werden Schleifen eingesetzt. Das Beispiel zeigt die Zinsberechnung als PAP, Struktogramm und im Pseudocode.

Programmablaufplan	Struktogramm							
<pre>graph TD Start([Beginn]) --> Init[Jahre = 0] Init --> Input[Eingabe von Kapital, Laufzeit, Zinssatz] Input --> Decision{Jahre < Laufzeit} Decision -- Ja --> Output1[Ausgabe Jahre, Kapital] Output1 --> LoopBack Decision -- Nein --> Calc[Kapital ← Kapital + Kapital * Zinssatz] Calc --> Inc[Jahre ← Jahre + 1] Inc --> LoopBack subgraph LoopBack [] direction TB Output1 Calc Inc end LoopBack --> Output1 Output1 --> End([Ende])</pre>	<table><tr><td>Beginn</td></tr><tr><td>Jahre = 0</td></tr><tr><td>Eingabe von Kapital, Laufzeit und Zinssatz</td></tr><tr><td>Solange Jahre < Laufzeit</td></tr><tr><td> Ausgabe Jahre, Kapital</td></tr><tr><td> Kapital ← Kapital + Kapital * Zinssatz</td></tr><tr><td> Jahre um 1 erhöhen</td></tr></table>	Beginn	Jahre = 0	Eingabe von Kapital, Laufzeit und Zinssatz	Solange Jahre < Laufzeit	Ausgabe Jahre, Kapital	Kapital ← Kapital + Kapital * Zinssatz	Jahre um 1 erhöhen
Beginn								
Jahre = 0								
Eingabe von Kapital, Laufzeit und Zinssatz								
Solange Jahre < Laufzeit								
Ausgabe Jahre, Kapital								
Kapital ← Kapital + Kapital * Zinssatz								
Jahre um 1 erhöhen								
	<p>Pseudocode</p> <pre>begin Zinsberechnung Jahre := 0; Eingabe(Kapital, Laufzeit, Zinssatz); while Jahre < Laufzeit do Ausgabe(Jahre, Kapital); Kapital := Kapital + Kapital * Zinssatz; Jahre := Jahre + 1; end do Ausgabe(Jahre, Kapital); end Zinsberechnung</pre>							

Die Variable `Jahre`, die im Ablauf des Programms für die Zählung der Jahre benötigt wird, erhält den Initialwert 0. Der Benutzer gibt das Anfangskapital, die Laufzeit und den Zinssatz ein. Diese Größen werden für die Berechnung benötigt. In einer Schleife erfolgt die Ausgabe des aktuellen Jahres und Kapitals. Das Kapital wird neu berechnet. Es ergibt sich aus dem bisherigen Kapital zuzüglich der Zinsen, die sich aus Kapital mal Zinssatz errechnen. Danach wird das Jahr um 1 weitergezählt. Für jedes Jahr erfolgt ein Schleifendurchlauf. Die Schleife wird so lange immer wieder abgearbeitet, bis die Anzahl der Jahre die Laufzeit erreicht hat. Danach erfolgt die Ausgabe der endgültigen Ergebnisse der Berechnung, und das Programm wird beendet.

A.2 Beispiel Geldautomat

Der Vorgang zum Benutzen eines Geldautomaten (stark vereinfacht) ist im Folgenden als PAP, Struktogramm und im Pseudocode dargestellt.



Anhang B: Installationen und Quellangaben

B.1 Python laden und installieren

Download und Installation von Python

- ▶ Laden Sie Python von der Webseite <http://www.python.org>. Das ist das zentrale Webportal von Python. Hier findet man die wichtigsten Informationen zu Python und die notwendigen Ressourcen. Über den Menüpunkt DOWNLOADS erhalten Sie die aktuellen, aber auch frühere Versionen für verschiedene Betriebssysteme. Empfehlenswert ist, immer die neueste Version von Python für Ihr Betriebssystem herunterzuladen.
- ▶ Um Python zu installieren, folgen Sie den Schritten des Installationsassistenten. Im Folgenden wird die Installation für Windows, MacOS und Linux erklärt. Durch den Installer ist die Installation in der Regel geradezu trivial.

Die Installation unter Windows

Unter Windows starten Sie den Installer, den Sie auf Ihren PC geladen haben. Dieser startet wiederum mit einem Auswahldialog. Wenn Sie wollen, passen Sie dort in einem extra Assistenten mit zwei weiteren Fenstern Pfade und Einstellungen sowie die zu installierenden Bestandteile an. Die Einstellungen können aber meist in der Voreinstellung verbleiben.

Sehr zu empfehlen ist hingegen das Hinzufügen von Python zur Path-Angabe – dem Suchpfad von Windows. Dazu klicken Sie in dem Dialog die entsprechende Option an – bei der im Buch verwendeten Version von Python lautet diese ADD PYTHON 3.6 ZU PATH.



Bei Bedarf anpassen der Installation

Wenn Sie den Installationsbefehl geben (*Install Now*), läuft die Installation ohne weitere Interaktion. Auf dem Rechner hat der Python-Installer danach alle Dateien, die er für die Ausführung von Python-Programmen benötigt, im gewählten Verzeichnis installiert. Wenn Sie die entsprechende Option ausgewählt haben (und das ist empfohlen), wurde das Installationsverzeichnis in den Suchpfad von Windows aufgenommen.

Damit findet Windows insbesondere zwei Programme, die von zentraler Bedeutung für Python sind:

- ✓ Die Python-Kommandozeile, Python-Interpreter genannt. In der Python-Kommandozeile arbeitet man in einem Interaktivmodus.
- ✓ Die Python-GUI **IDLE** (Python's Integrated DeveLopment Environment) besteht im Wesentlichen aus einem Multi-Fenster-Texteditor mit Dateizugriff, Syntax-Hervorhebung, Autovervollständigung, intelligentem Einzug und ein paar anderen Features sowie einer Shell beziehungsweise Kommandozeile mit Syntax-Hervorhebung. Dazu gibt es einen integrierten Debugger und die Möglichkeit zur Ausführung des Programms direkt aus der IDE. Die IDE ist im Vergleich zu Eclipse, XCode, NetBeans oder Visual Studio ziemlich einfach gehalten, was den Umgang mit IDLE erleichtert. Wenn IDLE startet, wird in der Regel die Python-Kommandozeile mit dem Interaktivmodus angezeigt. Darüber hinaus gibt es jedoch einige weitere Möglichkeiten, die über verschiedene Menübefehle angeboten werden: vom Anpassen von IDLE selbst über den Aufruf der Dokumentation bis zum Laden von Quelltext und dem Ausführen aus der IDE heraus.



Unter dem Menüpunkt DOCUMENTATION und dort DOCS finden Sie auf der Webseite zu Python zahlreiche Informationen inklusive einer kompletten Dokumentation, FAQs (Frequently Asked Questions, engl. für häufig gestellte Fragen) und verschiedene Tutorials. Man kann diese Materialien von dort herunterladen, aber das ist im Grunde gar nicht notwendig. Da Sie ja meist online sind, finden Sie die Dokumentation unter <https://docs.python.org>. Die Dokumentation ist auch im Installationspaket von Python enthalten beziehungsweise wird mit installiert, wenn Sie das bei der Installation auswählen.

Installieren auf einem Mac

Für die Installation auf einem Mac steht ebenfalls ein Installer zur Verfügung, der genauso einfach arbeitet wie der Installer unter Windows. Allerdings müssen Sie bei der Installation auf einem Mac eine ganze Reihe an Lizenzbedingungen explizit bestätigen und vor der Installation diverse Hinweise beachten. Sie sollten auch auf einem Mac darauf achten, dass Python im Suchpfad zur Verfügung steht. Das sollte aber automatisch der Fall sein.

Installieren auf einem Linux-System

Python gibt es auch für Linux. Dabei muss man einschränken, dass es verschiedene Formate für die unterschiedlichen Linux-Distributionen gibt. Das macht die Installation etwas unübersichtlich. Aber dafür sind Sie flexibel und können Python aus Quellcodes, individuell für ein System angepasst, erstellen.

Dies ist oft jedoch nicht notwendig. Viele Linux-Distributionen installieren Python bereits automatisch mit. Sie können Python auch über die Anwendungsverwaltung zur Verfügung stellen, wodurch die Installation problemlos erfolgt.

Allerdings müssen Sie beachten, dass in den Installationspaketen nicht immer die aktuellste Version von Python bereitsteht. Falls Sie die neuste Version nicht über die Anwendungsverwaltung Ihrer Linux-Distribution erhalten, können Sie eine individuelle Installationsdatei laden. Die Installation dieser Dateien kann sich unterscheiden, und es muss auf die entsprechende Dokumentation verwiesen werden.

B.2 Quellangaben im Internet

Alles rund um Python	http://www.python.org
Cordova	https://cordova.apache.org/
Dia Diagram Editor	http://dia-installer.de/
Diagram Designer	http://meesoft.logicnet.dk/
Eclipse	http://www.eclipse.org
JDK	http://www.oracle.com/technetwork/java/javase/downloads/index.html
PapDesigner	http://friedrich-folkmann.de/papdesigner/
PSPad	http://www.pspad.com/de/
W3C	http://www.w3.org
Die Python-Dokumentation	https://docs.python.org

[
[]	100	Arrays, statische	99	calendar	132
[DEBUG ON]	158	Arrays, Zugriff auf	101	Call by reference	117
<		Arrays, zweidimensionale	99	Call by value	117
<Ausdruck>	59	ASCII-Code	55	Carrybit	53
<Bezeichner>	59	asctime()	133	Cascading Style Sheets (CSS)	28
<Buchstabe>	58	Assembler	19, 41	case	85
<Operator>	59	Assembler-Sprache	18	CASE	151
<Zeichen>	59	Assoziatives Array	106	case-sensitiv	45
<Ziffer>	58	Assoziativitätsregel	77	CASE-Werkzeuge	148, 151
<Zuweisung>	59	Asynchronous JavaScript and XML (Ajax)	28	Casting	69
4		Attribute	23	Central Processing Unit	10
4GL	18, 20	Aufrufer	21	Chip	10
A		Ausdruck	71, 75	CIL	24
ABC	26	Ausgabeanweisungen	79	class	47
Ablaufdiagramm	33	Ausnahme	94, 152	Closures	119
Abnahmetest	146	Ausnahmebehandlung	94, 152	CLR	24
Abstraktion	31	Automatisierung	12	COBOL	22
Abstraktionsprinzip	141	B		Codierung	145
add()	107	Backslash: Maskierung	64	command	136
Addierer	55	Backspace: Maskierung	64	Common Intermediate Language	24, 41
Addition	71	Backus-Naur-Form	58	Common Language Runtime	24
Aggregierte Datentypen	98	Backus-Naur-Form, erweiterte	58	Compiler	20, 40, 41
Agile Modelle	150	BASIC	22	Computational Thinking	12
Ajax	28	Bedingte Anweisung	81, 82	Computing	11
Akkumulator	55	Bedingungen	80, 92	Condition	80
Algorithmen	11, 12, 15	Bedingungen, Kontrollstrukturen	80	continue	94
Algorithmus von Euklid	130	Bedingungsprüfung	90	CPU	10
Algorithmus, allgemeiner	125	Beispiel, Zinsberechnung	163	Cross-Compiler	41
Algorithmus, eindeutiger	125	Benutzerdokumentation	17	CSS	28
Algorithmus, endlicher	125	Benutzereingaben	96	D	
Algorithmus, generischer	129	Benutzerfreundlichkeit	14, 15	Data binding	138
Algorithmus, iterativer	125	Bezeichner	61	Dateierweiterung: Python	45
Algorithmus, rekursiver	127	Bezeichner, Python	61	Daten	23
Algorithmus, terminiert	125	Bibliotheken	113	Daten, analoge	51
Amoeba	26	Binärsystem	50	Daten, digitale	51
Analyse	143	Bindelader	41	Datenbindung	138
Android	29	Binder	41	Datenflussplan	34
Anführungszeichen: Maskierung	64	Bit	52	Datenkapselung	23, 119, 141
ANSI-Code	56	Bitmuster	52	Datenstrukturen, elementare	98
Anweisung	78	Blockdiagramm	33	Datentyp bestimmen	62, 65
Anweisung, einfache	80	BNF	58	Datentyp, boolescher	65
Anweisung, leere	79	Boole	64	Datentyp, logischer	64
Anweisungsblock	79	boolean	74	Datentyp, numerischer	63
Anwendung	10	Boolesche Logik	73	Datentypen, sequenzielle	98
API	26, 43	Boolescher Datentyp	65	Datentypen, aggregiert	98
App	10	Borrowbit	54	Datentypen, numerische	65
append()	112	Bottom-up-Methode	142	datetime	132
Applikation	10	break	94, 95	Debug Control: IDLE	158
Apps, mobile	29	Breakpoint	156	Debugger	43, 156
Arbeitsspeicher	10	Bubble-Phase	137	Debugging	152, 155
Argumente	116	Bug	152	def	46, 115
Array	98	Built-in Functions	26, 44, 119	default-Zweig	85
Arrays definieren	100	Byte	52	Deklaration	46, 47
Arrays, dynamische	99	Bytecode	24, 41	Deklaration, Variable	66
Arrays, eindimensionale	99, 100	C		Delegation	138
		C#, Programmiersprache	24	dequeue, Schlange	110
		C, Programmiersprache	22	Dezimalsystem	49
		C++, Programmiersprache	24	Dezimalzahlen	63
				Dictionaries	106
				Division	72
				Dokumentation	15

Dokumentation: Python	166	Folge, Sequenz	80	Interaktivmodus	48, 166
Don't repeat yourself	16, 155	Formfeed	64	Interpreter	20, 40
Doppeltes Anführungszeichen:		Formularvorschub: Maskierung	64	iOS	29
Maskierung	64	from	123	Iterationen	31, 81, 87
do-while-Anweisung	91	Funktionen	114, 119		
DRY	16, 155	Funktionen aufrufen	114	J	
Dualsystem	50	Funktionskopf	115	Java	24
Dualzahl	50	Funktionsrumpf	115	JavaScript	25
Dualzahlen, Addition	53			JScript	25
Dualzahlen, Multiplikation	55	G			
Dualzahlen, Subtraktion	54				
Duck-Typing	66	Ganzzahldivision	72	K	
Dynamische Typisierung	66	Gegenstromverfahren	142		
		Geometry Manager	135	Keller	108
E		Geschachtelte Verzweigung	81	KISS-Prinzip	16, 155
EBCDI-Code	56	Gleitkomma-Datentyp	63	Klassen	23, 47
EBNF	58	Gleitkommazahlen	63	Kommandozeile: Python	48, 166
ECMAScript	25	Gleitpunktzahlen	63	Kommentare: Python	62
Editor	39	global	122	Kompilieren	40
Editoren, grafische	151	global: Python	66	Komplementbildung	53
Effizienz	15	Globale Variable	66	Komplexe Zahlen	63
Einarbeitungsaufwand	14	Grafikkarte	10	Konstanten	68
Eingabeweisungen	80	graphical user interface	134	Kontrollausgaben	156
Eingabemöglichkeit	120	Grobentwurf	144	Kontrollstrukturen	32, 86
Einrückungen	43	Groß- und Kleinschreibung	61	Korrektheit	14
elif	83	GUI	134	Korrektheit von Programmen	113
Endlosschleifen	95, 157	Gültigkeitsbereich: Variablen	65	Kriterien, software-ergonomische	15
Endwert	87			Kundendienst	14
enqueue, Schlange	110	H		Künstliche Intelligenz	18, 21
Entscheidungstabelle	37	Haltepunkte	156	L	
Entwicklungsumgebung	42	Handler	136		
Entwurf	144	Hardware	10	Lader	41
Entwurfstechnik	33	Hash	106	Laufzeiteffizienz	15
Eratosthenes	111	Hauptfunktion	46	Laufzeitfehler	154
Ereignisobjekt	132	Hauptplatine	10	Layout-Manager	135
Ereignisse	131	Hexadezimalsystem	50	len()	104, 110, 111
Erweiterungskarten	10	HTML	28	LIFO	108
Escape-Darstellung	64	Hybride Programmiersprachen	25	Linker	41
Euklid	130	Hypertext Markup Language (HTML)	28	Listen	102, 104
EVA-Prinzip	11			Listener	136
Event	131	I		Literale	61, 65
Event Listener	136			LiveScript	25
Event Source	136	Identifizier	61	localtime()	133
Eventhandler	131	IDLE	42, 166	Logo	27
Event-Objekt	132	if-Anweisung	81, 82, 83	Lokale Variable	65
Exception	94, 152	if-else-Anweisung	82	Lose Typisierung	66
Expression	71	Imaginäre Einheit	63		
Externe Variable	66	Imaginärteil	63	M	
		Implementierung	145		
F		import	123, 132	Mainboard	10
Fakultät	127	in	106	Makro	25
Fallauswahl	84	in: Membership-Operator	122	Map	106
Fehler, logische	43, 154	Index	99	Mapping	106
Fehlersuche: Auskommentieren	62	Individual-Software	18	Maschinencode	19, 39
Feinentwurf	145	Infinite loop	95	Maschinensprache	18
Feld, Array	98	Initialisierung, Variable	67	Massenspeicher	10
FIFO	110	Innere Funktionen	119	Mehrfache Verzweigung	81, 84
Fließkommazahlen	63	input()	80, 96, 120	Mehrfachverwendbarkeit	113
float()	63, 70	Instanzen	23	Membership-Operatoren	106
Floating Point-Zahlen	63	int()	63, 70, 96	Mengen	107
Flussdiagramm	33	Integer-Datentyp	63	Methoden	23, 113, 142
		Integrationstest	146	Methoden erstellen	115

Methodenkopf 115
 Methodenrumpf 115
 Minisprachen 18, 27
 Modell, lineares 147
 Modelle, agile 150
 Modularitätsprinzip 113, 140
 Module 132, 140
 Modul: Python 123
 Modulo 72
 Motherboard 10
 Multiplikation 71

N

Nassi-Shneiderman-Struktogramm 35
 Netscape 25
 Netscape Navigator 25
 Netzwerkkarte 10
 Neue Zeile (New line) 64
 new 100
 New line: Maskierung 64
 Nullindiziertes Array 99
 Numerische Datentypen 65

O

Oak 24
 Objektbasiert 25
 Objekte 23
 Objektorientierte Programmierung 23
 Oktalsystem 51
 once and only once 16
 OOP 23
 Operator, arithmetische Vorzeichen 71
 Operator, binärer 71
 Operator, bitweiser 73
 Operator, logischer 73
 Operator, unärer 71
 Operatorassoziativität 76
 Operatoren: sequenzielle
 Datentypen 106
 Operatoren, Rangfolge 76
 Operatorvorrang 76
 Operatoren 31

P

PAP 33, 163
 PapDesigner 33
 Parameter 45, 113
 Parameter als Referenz 117
 Parameter übergeben 116
 Parameter, aktuelle 113
 Parameter, formale 113, 116
 Parameterübergabe als Wert 117
 Paritätsbit 52
 pass 79
 Pflichtenheft 14, 17, 143
 Pointer 62
 pop, Stapel 108
 Potenz 72
 Präprozessor 41
 Primitive Datentypen 62

Primzahlen 111
 print() 44
 Produktdefinition 14
 Programmierparadigma 18
 Programm 11, 12
 Programm, Grundaufbau 43
 Programm, maschinenabhängiges 41
 Programmablauf 31
 Programmablauf, Logik 32
 Programmablaufplan 33
 Programmbeschreibung 15
 Programmcode 11
 Programmwurf 33
 Programmfluss 31

Programmierschnittstelle 26, 43
 Programmiersprache 11, 18
 Programmiersprache, höhere 20
 Programmiersprachen,
 Klassifizierung 18
 Programmiersprachen, deklarative 19
 Programmiersprachen,
 erziehungsorientierte 18, 27
 Programmiersprachen,
 funktionale 18, 21
 Programmiersprachen, hybride 18, 25
 Programmiersprachen, logische 18, 27
 Programmiersprachen,
 objektorientierte 18, 23, 25

Programmiersprachen, prozedurale 18
 Programmierstil 15
 Programmlogik 31
 Programmspezifikation 15
 PROLOG 27
 Prototyp, horizontaler 149
 Prototyp, vertikaler 149
 Prototyping-Modell 148
 Prozedurale Sprache 20
 Prozeduren 21, 113
 Prozeduren aufrufen 113
 Prozessor 10
 Pseudocode 36, 163
 push, Stapel 108
 Python 25
 Python, Dokumentation 166
 Python, Kommandozeile 48
 Python 3000 26
 Python Package Index 26
 Python Shell 44
 Python-Interpreter 166
 Python-Kommandozeile 166
 Python's Integrated DeveLopment
 Environment (IDLE) 42, 166

Q

Qualitätskriterien 14
 Quellcode 11, 39
 Quellprogramm 39
 Quelltext 11, 39
 Quelltexterstellung, automatische 151
 Queue 110

R

raise 94
 RAM 10
 randint() 124
 random 124
 Random-Access Memory 10
 Realteil 63
 Records 98, 101
 Redundanz 145
 Redundanz: Vermeidung 16
 Referenztyp 62
 Register 19
 Rekursive Funktionsaufrufe 127
 repeat 92
 Reservierte Wörter 61
 return 94
 Return 64
 RIA 28
 Risikoanalyse 150
 Robustheit 14
 root-Element 46
 Rückgabewert 21, 94
 Rückgabewerte, Funktionen 114, 118
 Rückschritt 64
 Run Module 45, 46

S

Scheduler 132
 Schlangen 110
 Schleife, abweisende 89
 Schleife, bedingte 87
 Schleife, fußgesteuerte 87, 90
 Schleife, kopfgesteuerte 87, 89
 Schleife, zählergesteuerte 86
 Schleifen 81, 86
 Schleifen, Fehler bei 88
 Schleifenkörper 86
 Schleifensteuerung 86
 Schlüsselwörter 61, 62
 Schnittstelle 114, 140, 141
 Scratch 28
 seed() 124
 Selbstaufrufe 127
 Select Case 93
 Selektion 82, 84
 Selektor 84
 self 47, 115
 Semantik 58
 Semikolon: Anweisungsende 44
 Separator: print() 45
 Sequenz 12, 31, 80
 Sequenzielle Datentypen 98
 set() 107
 Sieb des Eratosthenes 111
 Signatur 114
 Skript 10
 Skriptsprachen 18
 sleep() 133
 Software 10
 Software, Einsatz 146
 Software, individuelle 18

Software, Integration	146	Typkonvertierung	69	Zeiger	62
Software, standardisierte	18	Typumwandlung	69	Zeitgeber	132
Software, Wartung	146			Zeitmesser	132
Softwaredokumentation: aus dem		U		Zeitzone	133
Quellcode heraus	15			Zerlegung	32
Software-Lebenszyklus,		Übergabewerte	45	Zufallszahl	123
Phasenmodell	143	UML	143	Zustand: Objekt	24
Softwarespezifikation	144	Unicode	57	Zuverlässigkeit	14
Soundkarte	10	Unixzeit	132	Zuweisungsanweisung	79
Source	11	Unterprogramm	21	Zuweisungsoperator	67, 74
Sourcecode	11	Unveränderliche Variable	68	Zwischencode	41
Speichermedien	10	update ()	112		
Speicherplatzeffizienz	15	Up-down-Methode	142		
Spiralmodell	149				
Sprache, formale	18	V			
Sprache, natürliche	18				
Sprache, prozedurale	18, 20	Validierung, V-Modell	147		
Sprunganweisungen	94	van Rossum, Guido	26		
Stack	108	Variable	65		
Standardbibliothek	43, 123, 132, 134	Variable, Deklaration	66		
Standard-Software	18	Variable, Gültigkeitsbereich	65		
Stapelüberlauf	108, 129	Variable, Initialisierung	67		
Stapelunterlauf	108	Variable, Lebensdauer	65		
Starke Typisierung	66	Variable, unveränderliche	68		
Statement	78	Verbund	98		
Statische Variable	66	Verfeinerung, schrittweise	142		
Stellenwert	49	Vergleichsoperator	72		
Steppen	157	Verifikation, V-Modell	147		
Stepwise refinement	142	Verzweigung, geschachtelte	83		
str ()	70, 75	Verzweigung, mehrfache	84		
Strenge Typisierung	66	Verzweigungen	32, 81, 82		
Strings	64, 101	V-Modell	147		
String-Verkettung	106	Vorgehensmodelle	140, 147		
String-Verkettungsoperator	75, 106	Vorzeichen	71		
Struktogramm	35, 163				
Struktur	98	W			
Subtraktion	71				
switch	85	Wagenrücklauf	64		
Syntaktische Fehler	153	Wartbarkeit	15		
Syntax	58	Wasserfallmodell	147		
Syntaxdiagramme	58	Web 2.0	28		
Syntaxfehler	40	Web-Applikationen	28		
Systemdokumentation	17	Webserver	28		
Systemtest	146	Wertzuweisung	67		
		while-Anweisung	89		
T		Wiederverwendbarkeit	43, 123, 141		
Tabulatur: Maskierung	64	World Wide Web (www)	28		
Testreihen	154	Wörter, reservierte	61		
Texteditor	39	Wurzelfunktion	46		
The Epoch	132				
then	81	Z			
time ()	132, 133				
Timer	132	Zahlensystem, arabisches	49		
tkinter	134	Zahlensystem, Basis	49		
Top-down-Methode	142	Zahlensystem, Nennwerte	49		
Trigger	138	Zahlensystem, römisches	49		
Tupel	102, 120	Zeichen, druckbare	55		
Turtle-Grafik	27	Zeichen, nicht druckbare	55		
type ()	65	Zeichencodes	55		
Typisierung, dynamische	66	Zeichen-Datentyp	64		
Typisierung, streng	66	Zeichenketten	64, 101		
		Zeichenliterale	64		