Informatik I (2F/NF)

Priv.-Doz. Dr. Frank Huch

Institut für Informatik, Technische Fakultät,
Christian-Albrechts-Universität zu Kiel.

Skript zur Vorlesung im Wintersemester 2018/19.

Version vom 7. Februar 2019

Inhaltsverzeichnis

1 Einleitung

Die Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, insbesondere die automatische Verarbeitung mit Rechenanlagen (Computer).

Ursprünge

Mathematik
Berechnen von Folgen, Lösung von Gleichungssystemen

Elektrotechnik
Computer als Weiterentwicklung von Schaltkreisen

Nachrichtentechnik
Datenübertragung im Rechner oder im WWW

Disziplinen der Informatik

Theoretische Informatik
Grundlagen, z.B. Komplexitätstheorie, Berechenbarkeit, Graphentheorie

Technische Informatik
Hardwarenahe Aspekte, z.B. Mikroprozessortechnik, Rechnerarchitekturen, Netzwerksysteme

Praktische Informatik
Lösen von konkreten Problemen durch Algorithmen/Programme
hierbei wichtige Aspekte: Effizienz, Softwaretechnik, Programmiersprachen, Datenbanken

1.1 Grundbegriffe

Algorithmus ist die Beschreibung einer Vorgehensweise zur Lösung von Problemen einer bestimmten Problemklasse.

Beispiele Kochrezept, Berechnung der Quadratwurzel, Dekodierung von DNA-Sequenzen, Berechnung von π, Verwaltungsvorschriften

Beachte dabei:

- in der Regel löst ein Algorithmus eine Klasse von Problemen (d.h. er ist *parametrisiert*), z.B. Quadratwurzel für beliebige Zahlen, Verwaltungsvorschrift für Hausbau. *Parameter* legt konkretes Problem der Klasse fest.

- die Vorgehensweise kann unterschiedlich detailliert formuliert werden, z.B. beim Kochrezept: „Eiweiß steif schlagen“oder „Schüssel holen, Eiweiß und Eigelb trennen, Schneebesen nehmen, . . . “. Es ist eine Kunst, aber auch Erfahrung, den richtigen bzw. geeigneten Grad an Genauigkeit zu finden. Wichtig hierbei: von Details abstrahieren. *Abstraktion* ist eines der wichtigsten Konzepte der Informatik.

Beispiel Zählen von Weizenkörnern (alle gleich schwer)
Hilfsmittel: Apothekerwaage (ohne Gewichte), einen Zettel, Bleistift

Algorithmus Wiederhole folgende Schritte, bis du keine Körner mehr hast:

- Teile Körner in zwei gleich schwere Haufen
- Ist dies nicht möglich lege ein Korn zur Seite. Danach kannst Du sie aufteilen.
- Beschrifte den Zettel mit $\begin{cases} 1, \text{ falls du ein Korn zur Seite gelegt hast} \\ 0, \text{ sonst} \end{cases}$
- Wähle einen Haufen aus, mit dem du im nächsten Schritt weiter machst.

Drehe die Zahl auf dem Zettel um und lies das Ergebnis als Binärzahl ab. Ggf. kannst du diese Zahl in eine Dezimalzahl wandeln.

Beispiel

$$\left.\begin{array}{rcl} 13:2 & = & 6 \text{ R } 1 \\ 6:2 & = & 3 \text{ R } 0 \\ 3:2 & = & 1 \text{ R } 1 \\ 1:2 & = & 0 \text{ R } 1 \end{array}\right\} 1101b = (8+4+1)d = 13d$$

Algorithmen werden in der Informatik häufig mit Hilfe von Programmiersprachen *programmiert*, so dass sie auf Rechnern/Computern ausgeführt werden können. Hierbei ist zunächst keine Abstraktion mehr möglich. Ein Programm muss so konkret sein, dass ein Prozessor (winziger Befehlssatz) den Algorithmus ausführen kann.

Programmierung war zunächst nur in Maschinensprache möglich, was aber wenig komfortabel war/ist. Feste Abstraktionen der höheren *Programmiersprachen* ermöglichen elegantere Programmierung auf abstrakterem Niveau. (In dieser Vorlesung werden wir die Sprache **Python** (in der Version 3) kennenlernen.) Darüber hinaus ist weitere Abstraktion durch *Programmiertechniken* (Softwaretechnik) möglich.

Programmierung ist heute die Kunst/Technik ein Problem und seine Lösung in Teile zu zerlegen, so dass sich ein kompositionelles System ergibt, welches leicht zu verstehen, zu ändern und zu konfigurieren ist.

Programmiersprachen ermöglichen die Kommunikation mit dem Rechner auf möglichst „natürliche“Weise. Programme werden entweder mit Hilfe eines *Compilers* in Maschinensprache übersetzt (z.B. *C*) oder durch ein spezielles Programm (*Interpreter*) interpretiert (z.B. *Python*). Mischformen sind ebenfalls möglich, bei denen ein Compiler in eine dann interpretierte Zwischensprache übersetzt (z.B. *Java*).

Bei der Programmierung unterscheidet man drei Bereiche:

- *Syntax* beschreibt die zulässigen Zeichenfolgen der Programme
- *Semantik* beschreibt, wie die Programme ausgeführt werden, also die Bedeutung der Sprachkonstrukte

- *Pragmatik* beschreibt die Idee, wie die Programmiersprache verwendet werden soll (viele Wege führen nach Rom, aber welche sind gut?). Hier spielt auch das Programmieren als Kunst/Technik hinein.

Beachte, dass bisher wenig über die Effizienz von Programmen gesagt wurde. Diese ist in der Regel *unwichtig*!
Wenige Ausnahmen: zeitkritische Algorithmen
Wichtiger meist: Verständlichkeit, Wartbarkeit, Entwicklungszeit

2 Programmierung

Algorithmus: abstrakte Beschreibung zur Lösung von Problemen

Viele Algorithmen können prinzipiell automatisch durch Computer ausgeführt werden. Hierzu muss der Algorithmus in einer konkreten Programmiersprache *implementiert* werden. Nun wollen wir ein paar grundlegende Konzepte kennenlernen, die in vielen Programmiersprachen verwendet werden und mit deren Hilfe wir einfachste Algorithmen programmieren können.

2.1 Ausdrücke

Aus der Mathematik kennt man Ausdrücke

$$3+4 \qquad x^2+2x+1 \qquad (x+1)^2$$

Woraus bestehen Ausdrücke?

- Basiselemente:
  - Werte (Konstanten)
    z.B. $3, 4 \in \mathbb{N}$ oder $\pi \in \mathbb{R}$
  - Variablen
    z.B. x, y
    repräsentieren beliebige Werte und können später mit konkreten Werten belegt werden.
- Zusammengesetzte Ausdrücke erhält man durch die Anwendung von Funktionen (genauer eigentlich Funktionssymbolen, später hierzu mehr), z.B. $+, -, \cdot$ ($*$ in der Informatik) auf bereits gebildete Ausdrücke. Eigentlich müsste man für die Eindeutigkeit bei jeder Funktionsapplikation Klammern verwenden. Oft kann man hierauf aber verzichten, da Operator-Präzedenzen berücksichtigt werden. Auch zu Präzedenzen später noch mehr.

Die meisten dieser Funktionen sind zweistellig und verknüpfen zwei Ausdrücke zu einem neuen Ausdruck: $3+4$. Zweistellige Funktionen (und manchmal auch einstellige Funktionen), die (meist mit einem Sonderzeichen) zwischen zwei Ausdrücken notiert werden, nennt man auch *Operatoren*. Beispiele für Operatoren sind $+, -$ und $\cdot$.

Was ist aber mit $\frac{\sqrt{x^2+1}}{x}$?

Auch hier finden wir mehrere Funktionsanwendungen, die aber ungewöhnlich notiert werden:

- Die zweistelligen Funktionen $+$, $\hat{}$ (Exponentiation) und / (als Bruchstrich geschrieben)
- und die einstellige Funktionen $\sqrt{}$ (Wurzelfunktion).

Der Computer (die Programmiersprache) erwartet eine genormte Darstellung solcher Ausdrücke:

$$\begin{array}{rcl} x**2 & \text{statt} & x^2 \\ sqrt(x) & \text{statt} & \sqrt{x} \\ a/b & \text{statt} & \frac{a}{b} \end{array}$$

Somit können wir $\frac{\sqrt{x^2+1}}{x}$ schreiben als:

$$sqrt(x**2+1)/x$$

Entspricht dies tatsächlich dem mathematischen Term? Bei der Quadratfunktion ist es auf Grund der Notation klar, dass nur das x quadriert wird. Hier verwenden wir aber die Funktion $**$. Woher wissen wir, dass in diesem Ausdruck x quadriert und nicht „hoch 2+1“gerechnet wird?

Ähnlich wie in der Mathematik verwenden auch die meisten Implementierungen von Ausdrücken in Computern Präzedenzen, die festlegen, welche Operatoren stärker als andere binden. Ein Beispiel für solch eine Präzedenz ist die Punkt-vor-Strich-Rechnung: $*$ und $/$ binden stärker als $+$ und $-$. Sie ermöglicht es uns unter Umständen auf Klammern zu verzichten. Wie ist das aber mit den Operatoren $**$ und $+$?

Hierzu wäre es nützlich, unseren Ausdruck mit diesem Ausdruck:

$$sqrt(x**(2+1))/x$$

zu vergleichen. Wir untersuchen also, wie die Semantik (Bedeutung) dieser beiden Ausdrücke in der Programmiersprache Python ist. Zunächst beginnen wir mit einem einfacheren Beispiel: $3+4$

Um den Wert dieses Ausdrucks zu berechnen, schreiben wir unser erstes Python-Programm. Wir editieren (mit Hilfe eines Editors, z.B. Atom) eine Python-Datei erstesProgramm.py. Wichtig ist hierbei die Endung .py für „Python-Datei“. Der Inhalt unseres ersten Programms sieht wie folgt aus:

```
print(3+4)
```

Der Befehl print dient zur Ausgabe von Werten. Der Ausdruck, den wir auswerten wollen, schreiben wir in die Klammer hinter den print-Befehl. Die Zeile beenden wir mit einem Semikolon.

Nun können wir dieses Programm, wie folgt ausführen:

- in einer Shell tippen wir: python3 erstesProgramm.py
- in Atom drücken wir auf Programm ausführen.

Das Programm wird ausgeführt, der Ausdruck wird ausgewertet und das Ergebnis 7 erscheint auf dem Bildschirm.

Nun wollen wir die Ergebnisse der beiden Ausdrücke $sqrt(x**2+1)/x$ und $sqrt(x**(2+1))/x$ vergleichen. Hierzu ändern wir den Ausdruck in unserem Programm:

```
print(sqrt(x ** 2 + 1) / x)
```

Führen wir dieses Programm aus, erhalten wir folgende Python-Fehlermeldung:

```
Traceback (most recent call last):
  File 'erstesProgramm', line 1, in <module>
NameError: name 'sqrt' is not defined
```

Dies liegt daran, dass Python die Funktion sqrt nicht direkt zur Verfügung stellt. Wir müssen hierzu die Bibliothek math importieren und die Wurzel-Funktion qualifiziert mittels math.sqrt verwenden:

```
import math

print(math.sqrt(x ** 2 + 1) / x)
```

Führen wir dieses Programm aus, erhalten wir eine ähnliche Python-Fehlermeldung:

```
Traceback (most recent call last):
  File 'erstesProgramm.py', line 3, in <module>
    print(math.sqrt(x ** 2 + 1) / x)
NameError: name 'x' is not defined
```

Der Grund ist, dass der Wert der Variable x unbekannt ist und Python den Wert des Ausdrucks nicht berechnen kann. Es gibt ja auch gar nicht „einen Wert“für diesen Ausdruck. Wir müssen also vorher festlegen, für welchen Wert der Variablen x, wir den Ausdruck auswerten wollen. Hierzu belegen wir vorher die Variable mit einem Wert:

```
import math

x = 42          # Belege x mit dem Wert 42
print(sqrt(x ** 2 + 1) / x)
```

Die Belegung einer Variablen bezeichnet man auch als *Zuweisung*. Hierauf werden wir später noch genauer eingehen.

Nun können wir die unterschiedlich geklammerten Versionen unseres Ausdrucks vergleichen und sehen, dass der Operator $**$ stärker als $+$ bindet und die Klammern um den Teilausdruck $** 2$ tatsächlich nicht notwendig sind.

Der Vorteil von Präzedenzen ist, dass viele Klammern weggelassen werden können und Ausdrücke so besser lesbar werden. Der Computer, bzw. die Programmiersprache (oder auch andere Anwendungen, wie z.B. eine Tabellenkalkulation) fügt intern automatisch die fehlenden Klammern hinzu. Die vollständig geklammerte Schreibweise für unseren Ausdruck wäre:

$$(sqrt(((x ** 2) + 1))/x)$$

Jede Operatoranwendung wird geklammert.

Neben der Präzedenz ist es auch noch wichtig, zu verstehen, wie Ausdrücke mit Operatoren mit gleicher Präzedenz geklammert werden. Als Beispiel betrachten wir den Ausdruck $5-3-2$. Wieder gibt es zwei mögliche Interpretationen dieses Terms: $5-(3-2)$ und $(5-3)-2$. Vergleichen wir wieder die unterschiedlichen Ergebnis dieser Ausdrücke, sehen wir dass der Terme $5-3-2$ dem Term $(5-3)-2$ entspricht. Es wird also links geklammert. Man sagt auch der Operator $-$ bindet *links(-assoziativ)*. Andere Operatoren können auch rechts-assoziativ binden, worauf man beim Verständnis von Ausdrücken achten sollte.

Generell gilt: wenn man sich nicht sicher ist, wie ein Operator bindet, sollten ruhig zusätzliche (überflüssige) Klammern verwendet werden.

2.2 Einfachste Algorithmen und ihre Implementierung als Ausdrücke

Nachdem wir Ausdrücke und ihre Implementierung in Python kennengelernt haben, wollen wir Ausdrücke zur Programmierung einfacher Algorithmen verwenden.

Wir betrachten folgendes Problem:

Gegeben: Radius r
Aufgabe: Bestimme die Fläche eines Kreises mit Radius r
Ausdruck: $\pi \cdot r^2$

Entsprechend können wir diesen Algorithmus als Python-Ausdruck implementieren:

```
import math     # Brauchen wir für die Konstante pi.

r = 4           # Hier legen wir den konkreten Radius fest.

print(math.pi * (r ** 2))
```

Ausdrücke spielen in Programmiersprachen eine wichtige Rolle. Sie kommen aber auch in vielen anderen Anwendungen vor. Als weiteres Beispiel betrachten wir Tabellenkalkulationen, wie z.B. Excel, Openoffice oder Libreoffice.

Hier können Ausdrücke als Formeln verwendet werden. Anstelle von Variablen verwendet man die Zellen einer Tabelle. Adressiert werden diese durch einen Buchstaben für die Spalte, gefolgt von einer Zahl für die Zeile. Ist z.B. der Wert für den Radius in der Zelle B1 gespeichert, so können wir den Radius mit Hilfe der Formel (des Ausdrucks) $3.1415*B1*B1$ berechnen.

Um unser Programm noch etwas spannender zu machen, können wir den Radius auch vom Benutzer einlesen und so für beliebige Kreise die Fläche zu berechnen:

```
import math         # Brauchen wir für de Konstante pi.

r = input()         # Hier wird der konkrete Radius vom Benutzer
                    # eingelesen.
print(math.pi * (r ** 2))
```

Als nächstes betrachten wir ein etwas schwierigeres Problem:

Gegeben: Zwei Zahlen n und m
Aufgabe: Bestimme das Maximum von n und m

Diese Aufgabe können wir mit den bisherigen Funktionen nicht lösen, es sei denn, wir gehen davon aus, dass wir eine entsprechende vordefinierte Funktion zur Verfügung haben.

Ein Algorithmus zur Lösung dieses Problems sieht wie folgt aus:

Vergleiche n mit m
Falls $n < m$ ist, dann ist m das Maximum, sonst ist n das Maximum.

2.2.1 Boolesche Werte

Welche neuen Funktionen benötigen wir, um diesen Algorithmus zu implementieren?

Zunächst den Vergleich $<$, aber was ist das Ergebnis von $n < m$?

Eine Möglichkeit:

0 für n nicht kleiner als m
1 für n kleiner als m

So ist dies z.B. in der Programmiersprache C realisiert. Eine bessere Lösung ist aber die Verwendung spezieller *boolescher*[1] *Werte*: False und True.

Für die Vergleichsfunktion $<$ ergibt sich dann: Der Wert False ist Ergebnis für nicht kleiner und
der Wert True ist das Ergebnis für kleiner.

Also: $3 < 4 \rightsquigarrow \mathrm{True}$ $\quad$ $4 < 4 \rightsquigarrow \mathrm{False}$.

Beachte, dass True und False zwar dem intuitiven wahr bzw. falsch entsprechen, aber dennoch Werte, wie 42 oder -15, sind. Genau wie $3 + 4$ zu 7 reduziert wird, wird $3 < 4$ zu True reduziert.

Entsprechend stehen in Python auch Operatoren $<=$ (für $\leq$), $>$, $>=$ (für $\geq$) und $!=$ (für $\neq$) zur Verfügung.

Nun müssen wir aber noch eine Möglichkeit finden, wie wir das „Falls ... dann ... sonst ...'— umsetzen.

Hierzu könnten wir eine Funktion if_then_else, welche drei Argumente benötigt, verwenden. Ihre Semantik ist wie folgt definiert:

Semantik von if_then_else

$$if\_then\_else(True, e_1, e_2) = e_1$$
$$if\_then\_else(False, e_1, e_2) = e_2$$

Beachte, dass das erste Argument ein boolescher Wert sein muss, damit diese Funktion ausgewertet werden kann. D.h. es können hier z.B. Vergleiche verwendet werden.

Damit können wir nun den Maximumsalgorithmus als Ausdruck implementieren:

$$if\_then\_else(n > m, n, m)$$

Werten wir diesen Ausdruck nun für unterschiedliche Variablenbelegungen aus, so ergibt sich:

$$\begin{aligned} n = 7, m = 42 \quad &\Rightarrow \quad if\_then\_else(7 > 42, 7, 42) \\ &= \quad if\_then\_else(False, 7, 42) \\ &= \quad 42 \end{aligned}$$

und

$$\begin{aligned} n = 42, m = 8 \quad &\Rightarrow \quad if\_then\_else(42 > 8, 42, 8) \\ &= \quad if\_then\_else(True, 42, 8) \\ &= \quad 42 \end{aligned}$$

In Python wird if_then_else nicht als Applikation einer dreistelligen Funktion notiert, sondern in einer Mixfixnotation geschrieben, wobei die Bedingung überraschender Weise in der Mitte notiert wird:

e_1 if b else e_2 anstelle von $if\_then\_else(b, e_1, e_2)$

In der Anwendung in unserem Ausdruck für die Maximumsberechnung also

[1] Benannt nach dem englischen Mathematiker George Boole (1815-1864).

```
n = 42
m = 7

print(n if n > m else m)
```

In den meisten anderen Programmiersprachen existiert diese Funktion ebenfalls, wird aber recht unterschiedlich notiert. In Java und C schreibt man beispielsweise

```
b ? e1 : e2
```

und in Ruby

```
if b then e1 else e2 end
```

Die Semantik entspricht aber immer genau der oben skizzierten, dreistelligen Funktion if_then_else.

Auch in Tabellenkalkulationen können boolesche Ausdrücke zur Definition von Formeln verwendet werden. Hier wird das if-then-else auch tatsächlich als 3-stellige Funktion IF_THEN_ELSE oder WENN_DANN_SONST (in der deutschen Variante) notiert. In einigen Programmen wird allerdings ein Semikolon anstelle eines Kommas zur Trennung der Argumente verwendet.

Als weiteres Beispiel betrachten wir die boolesche Funktion OR (Oder).

Gegeben: zwei boolesche Werte in x und y
Gesucht: $OR(x, y)$ mit

OR	$y = True$	$y = False$
$x = True$	$True$	$True$
$x = False$	$True$	$False$

Mit der 3-stelligen if_then_else-Funktion können wir OR als den folgenden Ausdruck definieren:

$$if\_then\_else(x, True, if\_then\_else(y, True, False))$$

Der entsprechende Python-Ausdruck zur Berechnung von OR sieht dann wie folgt aus:

```
x = True
y = False

print(True if x else True if y else False)
```

Es gibt aber auch eine noch kompaktere Definition:

```
print(True if x else y)
```

In Python ist die Funktion OR als Infixoperator or vordefiniert und kann in booleschen Ausdrücken verwendet werden. Entsprechend gibt es auch ein AND (Implementierung als Übung), welches als and zur Verfügung steht. Auch diese Operatoren haben unterschiedliche Präzedenzen (siehe Übung).

Wir werden später noch genauer auf die korrekte Syntax und Semantik von Ausdrücken eingehen. Zunächst soll diese intuitive Idee ausreichen. Wichtig ist es insbsondere zu verstehen, dass ein Ausdruck nur unter Berücksichtigung einer konkreten Belegung aller vorkommenden Variablen ausgewertet werden kann, seine Semantik also von der aktuellen Variablenbelegung abhängt.

2.2.2 Ganze Zahlen

Bisher habe wir Zahlen und boolesche Werte kennengelernt. Die Zahlen wollen wir noch etwas genauer untersuchen. In der Mathematik unterscheidet man unterschiedliche Zahlenmengen: natürliche Zahlen, ganze Zahlen, rationale Zahlen und reelle Zahlen. Keine dieser Zahlenmengen kann in einem Computer repräsentiert werden, da jede Menge unendlich viele Zahlen enthält. Ein Computer besitzt aber nur einen endlichen Speicher, so dass auch nur endlich viele Zahlen dargestellt werden können.

Welche genauen Zahlendarstellungen verwendet werden, hängt von der jeweiligen Anwendung (Programmiersprache oder Tabellenkalkulation) ab. Wir wollen hier aber die wichtigsten kennenlernen.

Ganze Zahlen

Untersuchen wir einmal folgende Ausdrücke in Python:

```
print(2**0)          # -> 1
print(2**1)          # -> 2
print(2**10)         # -> 1024
print(2**100)        # -> 1267650600228229401496703205376
print(2**1000)       # -> 10715086071862673209484250490600...
print(0-2**1000)     # -> -10715086071862673209484250490600...
```

Alle Werte können exakt durch Python berechnet werden. Python stellt tatsächlich ganze Zahlen (fast) beliebiger Größe dar. Die Beschränkung ergibt sich nur aus dem maximal verfügbaren Speicherplatz, was aber in der Praxis so gut wie nie ein Problem darstellt.

In manchen anderen Programmiersprachen und auch vielen Prozessoren wird nur ein endlicher Zahlenbereich zur Verfügung gestellt. Hierdurch ist gewährleistet, dass alle möglichen Zahlenwerte mit einer festen Anzahl von Bits dargestellt werden können. Je nach verwendeter Bit-Länge können die folgenden Werte dargestellt werden:

Bit-Länge	kleinste Zahl	größte Zahl
8	-128	127
16	-32.768	32.767
32	$-2.147.483.648$	$2.147.483.647$
64	$-9.223.372.036.854.775.808$	$9.223.372.036.854.775.807$

Falls eine Rechnung über die untere oder obere Grenze hinaus geht, gibt es in der Regel einen Überlauf und somit ein falsches Ergebnis. Z.B. würde bei 8 Bit bei der Berechnung 120+30 das Ergebnis -23 herauskommen (weitere Infos hierzu findet man unter dem Stichwort *Zweierkomplement*).

In manchen Systemen wird dieser Überlauf aber auch abgefangen und man erhält eine Fehlermeldung. In Python tritt dieses Problem erst gar nicht auf.

2.2.3 Gleitkommazahlen

Für manche praktischen Probleme ist es aber auch sinnvoll, rationale Zahlen bzw. reelle Zahlen zu betrachten. Taschenrechner und Computer bieten scheinbar auch solche Zahlen an.

In Python heißen diese Zahlen `float` (floating-point numbers oder Fließkommazahlen) und man erhält sie, in dem man mindestens eine Nachkommastelle verwendet (z.B. 3.0 oder 42.75). Alle arithmetischen Operationen liefern dann auch wieder `float`-Zahlen, so dass wir gefühlt mit Dezimalbrüchen rechnen können.

Aus der Mathematik ist klar, dass wir reelle Zahlen, wie z.B. $\sqrt{2}$, nicht exakt im Computer darstellen können. Diese Zahl weist keine Periode auf und ist bereits ein unendliches Objekt, welches nicht im endlichen Speicher des Computers dargestellt werden kann. Das Ergebnis von $\sqrt{2}$ kann also nur angenähert werden.

Entspricht `float` also der Menge der rationalen Zahlen? Dies hat sich in der Praxis ebenfalls nicht bewährt, da Brüche häufig nicht gekürzt werden können und somit nach wenigen Operationen bereits eine Darstellung mit sehr großen Zählern und Nennern entsteht.

Welche Darstellung erscheint also geeignet? Eine fest begrenzte Anzahl von Vor- und/oder Nachkommastellen. Solch eine Darstellung nennt man Festkommazahl. Der Nachteil dieser Darstellung ist aber, dass bei sehr großen bzw. kleinen Werten ein recht großer Teil der Zahlendarstellung nicht für Genauigkeit genutzt werden.

Als Lösung verwendet man deshalb in der Regel *Gleitkommazahlen* (*floating point numbers*, Float). Die Idee ist, dass man sich auf Zahlen einer bestimmten Genauigkeit beschränkt. Hierbei bedeutet Genauigkeit die Anzahl der unterscheidbaren Stellen.

Für eine Genauigkeit mit vier Stellen könnne wir z.B folgende Beispielzahlen nennen:

$$1234, \quad 23,45, \quad 34560000, \quad -0,0004567, \quad 120000$$

Beachte, dass alle Zahlen auch Zahlen einer größeren Genauigkeit sein können, wie man insbesondere am Beispiel 120000 sieht, welche auch eine Zahl mit der Genauigkeit zwei Stellen ist.

Für eine klarere Darstellung der Genauigkeit verwendet man besser eine genormte Darstellung:

$$1234 \cdot 10^{0}, \quad 2345 \cdot 10^{-2}, \quad 3456 \cdot 10^{4}, \quad -4567 \cdot 10^{-7}, 1200 \cdot 10^{2}$$

oder

$$0,1234 \cdot 10^{4}, \quad 0,2345 \cdot 10^{2}, \quad 0,3456 \cdot 10^{8}, \quad -0,4567 \cdot 10^{-3}, \quad 0,12 \cdot 10^{6}$$

Eine Float-Zahl wird also durch zwei Zahlen repräsentiert: die Mantisse (Ziffern der entsprechenden Genauigkeit) und einen Exponenten. Im Rechner steht für beides eine feste Anzahl von Bits (Zahlen im Binärformat) zur Verfügung.

Hiermit ergibt sich die kleinste bzw. größte darstellbare positive Zahl als

$$2.2250738585072014 \cdot 10^{-308}$$

$$1.7976931348623157e + 308 \cdot 10^{45}$$

Das Python-System versucht, Float-Werte möglichst gut für den Menschen lesbar auszugeben. Zum Beispiel beträgt die Lichtgeschwindigkeit

$$\begin{aligned} e &= 2,99792458 \cdot 10^8 \text{m/s} \\ &= 2,99792458\text{e}8\text{m/s} \end{aligned}$$

In Python wird dies als 299792458.0 ausgegeben. Die Gravitationskonstante $G = 6,67384 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$ wird aber genau in dieser Darstellung als $6.67384\text{e}{-11}$ präsentiert.

Auf Grund der Ungenauigkeiten von Float kann es zu Rundungsfehlern kommen, wie das folgende Beispiel zeigt:

```
print(42 if (1-0.2-0.2-0.2-0.2-0.2)==0 else -1)  # -> -1
print(1-0.2-0.2-0.2-0.2-0.2)  # -> 5.551115123125783e-17
```

Bei Abbruchbedingungen sollte man also immer prüfen, ob man bis auf eine bestimmte Nähe an den Zielwert herangekommen ist:

```
x = 1-0.2-0.2-0.2-0.2-0.2

print(42 if x >=-0.0001 or x<=0.0001 else -1)  # -> 42
```

Bem.: or ist der Operator für das boolesche Oder. Der Ausdruck ist erfüllt, wenn mindestens eine der beiden Vergleiche erfüllt ist.

2.2.4 Auswertung von Ausdrücken

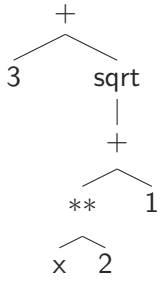
Ausdrücke (*Terme*) haben wir bisher so verstanden, wie sie auch in der Schulmathematik eingeführt wurden. Neu waren dabei die booleschen Werte und die dreistellige if-then-else-Funktion. Um zu verstehen, wie ein Computer mit solchen Ausdrücken umgeht, beschäftigen wir uns nun intensiver mit der Darstellung von Ausdrücken, ihrer Repräsentation im Computer und einer Auswertungsmaschine.

Ausdrücke werden (in der Mathematik) häufig auch als *Terme* bezeichnet. Durch die unterschiedliche Schachtelung der Operatoren und Funktionen bilden sie eine Baumstruktur, welche insbesondere in Form der *Termbaum*-Darstellung verdeutlicht wird.

Hierbei steht die Funktion bzw. der Operator jeweils oberhalb seiner Argumente.

Bsp.: $3 + sqrt(x**2 + 1)$

hat die folgende Termbaum-Repräsentation:



Die inneren Knoten des Termbaums sind mit Funktionen/Operatoren beschriftet, die Blätter mit Werten oder Variablen. Der Verzweigungsgrad der inneren Knoten ergibt sich genau

aus der Stelligkeit der entsprechenden Funktion/Operation, d.h. die Kindknoten eines Funktionsknoten sind mit den Termbäumen der Argumente beschriftet. So ist die Wurzel des gesamten Termbaums mit dem Operator $+$ beschriftet. Sein linkes Kind enthält die Termbaumdarstellung des Teilterms 3 und sein rechtes Kind die Termbaumdarstellung des Teilterms $sqrt(x**2\ +\ 1)$.

Beachte: Es kommen keine Klammern mehr vor! Diese sind nur in der linearen Darstellung als Zeichenfolge notwendig. In der Baumstruktur werden innere/äußere Funktionsanwendung dadurch repräsentiert, dass sie weiter oben bzw. unten im Termbaum auftreten.

Gibt es noch andere Darstellungen für Terme?

Bisher: $f(g(x,y),3)$ und Infixoperatoren $(3+4)$.

Ist die Stelligkeit aller Funktionen bekannt, ist es auch möglich, die Klammern und Kommata ganz wegzulassen. Man erhält die klammerfreie Präfixnotation:

$$f\ g\ x\ y\ 3 \qquad \text{mit } f \text{ und } g \text{ 2-stellig}$$

$$\begin{aligned} +\ 3\ -\ 4\ x &\triangleq +(3,-(4,x)) \triangleq (3+(4-x)) \\ **+sqrt\ x\ 1\ /\ 4\ 2 &\triangleq **(+(sqrt(x),1),/(4,2)) \\ &\triangleq (sqrt(x)+1)**(4/2) \end{aligned}$$

Entsprechend gibt es auch die klammerfreie Postfixnotation, bei der alle Funktionsanwendungen nach den Argumenten folgen:

$$\begin{aligned} x\ y\ g\ 3\ f \\ 3\ 4\ x\ -\ + \\ x\ sqrt\ 1\ +\ 4\ 2\ /** \end{aligned}$$

Wozu sind diese Termdarstellungen gut?

Insbesondere die Postfixnotation kann gut zur automatischen Auswertung des Ausdrucks verwendet werden, wie sie im Prinzip auch in vielen Implementierungen von Ausdrücken in Computern implementiert wird.

Man verwendet hierzu eine sogenannte *Stackmaschine*.

Ein *Stack* (oder Keller, Stapelspeicher) ist eine Struktur, in der beliebig viele Werte abgelegt und wieder herausgenommen werden können. Die Werte liegen hierbei „übereinander'—, so dass immer nur „oben'— auf den zuletzt hinein geschriebenen Werte zugegriffen werden kann.

Es gibt nur zwei Operationen, mit denen ein Stack verändert werden kann:

push v, schiebt v auf den Stack

pop, holt oberstes Element vom Stack

Bsp.: Beginnen wir mit dem leeren Stack erhalten wir nach der Ausführung von push 3:

Nach Ausführung von push 5 erhalten wir:

5
3

Nach Ausführung von push 4 erhalten wir:

4
5
3

Nach Ausführung von pop erhalten wir die 4 als Ergebnis und wieder den Stack, von vorher:

5
3

führen wir push 7 aus:

7
5
3

Mit drei weiteren pop Operationen können wir noch die Werte 7, 5 und 3 nacheinander von Stack holen.

Es werden also der Reihe nach die Werte 4, 7, 5 und 3 vom Stack ge*pop*t.

Im folgenden werden wir Stacks auch horizontal notieren. Wichtig ist aber die Beachtung des LIFO-Prinzips (**l**ast-**i**n-**f**irst-**o**ut)

Einschub - FIFO-Prinzip

Es gibt auch das FIFO-Prinzip (**f**irst-**i**n-**f**irst-**o**ut). Diese Struktur nennt man Queue (oder Schlange).

Bei obigem Beispiel hätten die pop-Operationen die Werte in der folgenden Reihenfolge geliefert:

$$3, 5, 4, 7$$

Später werden wir uns noch ausführlicher mit Queues beschäftigen.

Wie kann nun der Stack in der Stackmaschine verwendet werden, um den Wert eines Ausdrucks zu berechnen?
Hierzu verwendet man am besten die Postfixnotation:

Bsp.: $(3 + 4) * sqrt(2 * *2)$

Postfix: $3\ 4\ +\ 2\ 2 * *sqrt*$

Eine *Konfiguration der Stackmaschine* besteht jeweils aus einem Keller, neben dem der (restliche) Postfixausdruck steht. Begonnen wird mit dem leeren Keller (notieren wir als ε)

und der Postfixnotation des gesamten Ausdrucks:

$$
\begin{aligned}
\varepsilon \mid 3\ 4\ +\ 2\ 2 * *\ sqrt\ * \quad &\Rightarrow \quad 3 \mid 4\ +\ 2\ 2 * *\ sqrt\ * \\
&\Rightarrow \quad 3\ 4 \mid\ +\ 2\ 2 * *\ sqrt\ * \\
&\Rightarrow \quad 7 \mid 2\ 2 * *\ sqrt\ * \\
&\Rightarrow \quad 7\ 2 \mid 2 * *\ sqrt\ * \\
&\Rightarrow \quad 7\ 2\ 2 \mid\ * *\ sqrt\ * \\
&\Rightarrow \quad 7\ 4 \mid sqrt\ * \\
&\Rightarrow \quad 7\ 2 \mid\ * \\
&\Rightarrow \quad \mathbf{14} \mid \\
&\qquad\ \uparrow \qquad\quad \nwarrow \\
&\qquad \text{Ergebnis} \quad \text{leere Eingabe}
\end{aligned}
$$

Allgemein lässt sich die Auswertung der Stackmaschine mit folgenden Regeln beschreiben:

Wir notieren eine Konfiguration der Stackmaschine mit einem S für den Stack und einer Eingabe p (einem Teil eines Postfixausdrucks).

Die *Startkonfiguration der Stackmaschine* besteht aus einem leeren Stack und dem auszuwertenden Term in Postfixnotation (p_0):

leerer Stack — initialer, zu berechnender Term in Postfixnotation

$$\varepsilon \mid p_0$$

Die *Konfigurationsübergänge der Stackmaschine* sind wie folgt definiert:

$S \mid v\ p$ mit v ein Wert (z.B. Zahl, True, False)

$\Rightarrow S\ v \mid p$

$S\ v_1 \ldots v_n \mid f\ p$ mit f eine n-stelliges Funktionssymbol und $\tilde{f}$ die Semantik von f

$\Rightarrow S\ \tilde{f}(v_1 \ldots v_n) \mid p$

Die *Endkonfiguration der Stackmaschine* hat die Form:

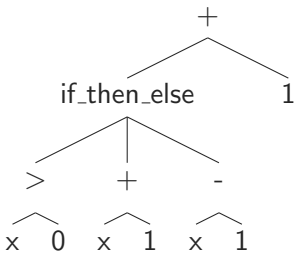
$$
\begin{array}{rl}
 & v \mid \\
 & \uparrow \quad \uparrow \\
\text{nur ein! Element auf Stack} & \text{leere Eingabe} \\
\text{Das Ergebnis der Auswertung ist } v & \text{(Term komplett abgearbeitet)}
\end{array}
$$

Beachte, dass die Argumente der Funktion eigentlich in der falschen Reihenfolge vom Stack gepopt werden. Deshalb wurde für die Termauswertung ursprünglich die Polnische Notation verwendet, in der die Argumente im Vergleich zur Postfixnotation in umgekehrter Reihenfolge angegeben werden. Wir definieren die Stackmaschine hier aber für die Postfixnotation (umgekehrte Polnische Notation) und müssen bei der Funktionsanwendung entsprechend korrekt applizieren.

Weiteres Bsp.:

```
(x + 1 if x > 0 else x - 1) + 1
```

$\rightsquigarrow$ Term-Baum:



$\rightsquigarrow$ Postfix-Notation:

$$x\ 0\ >\ x\ 1\ +\ x\ 1\ -\ if\_then\_else\ 1\ +$$

Wir betrachten die Belegung: $x = 40$

Auswertung mit Stack-Maschine:

$$\begin{array}{rr|l}
 & & 40\ 0\ >\ 40\ 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & 40 & 0\ >\ 40\ 1\ +\ 40\ 1\ -\ if_then_else 1\ + \\
\Rightarrow & 40\ 0 & >\ 40\ 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & True & 40\ 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & True\ 40 & 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & True\ 40\ 1 & +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & True\ 41 & 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow^2 & True\ 41\ 40\ 1 & -\ if_then_else\ 1\ + \\
\Rightarrow & True\ 41\ 39 & if_then_else\ 1\ + \\
\Rightarrow & 41 & 1\ + \\
\Rightarrow & 41\ 1 & + \\
\Rightarrow & 42 &
\end{array}$$

Wir betrachten im Vergleich die Auswertung als Term, mit Python-Syntax:

$$\begin{array}{rl}
 & (40\ +\ 1\ if\ 40\ >\ 0\ else\ 40 - 1)\ +\ 1 \\
\Rightarrow & (40\ +\ 1\ if\ True\ else\ 40 - 1)\ \mathbf{40 - 1}\ end\ +\ 1 \\
\Rightarrow & (40 + 1) + 1 \\
\Rightarrow & 41 + 1
\end{array}$$

Die Berechnung von $40 - 1$ wurde gespart.

Das if_then_else kann auch schon ausgewertet werden, bevor das zweite und dritte Argument ausgewertet wurden, man sagt if_then_else ist **nicht strikt im 2. und 3. Argument**.

if_then_else ist aber auch **strikt** im 1. Argument, genau wie $+$, welches im ersten und zweiten Argument strikt ist.

Da if_then_else entweder das zweite oder das dritte Argument liefert, ist es sinnvoll, diese vor der if_then_else-Auswertung nicht zu berechnen. Hierzu sind kompliziertere Stack-Maschinen notwendig $\rightsquigarrow$ Informatik-Studium

In der Praxis ist es aber auch sinnvoll, dass if_then_else sein zweites und drittes Argument nicht unbedingt auswertet. So können mit Hilfe von if_then_else auch Fehler verhindert werden:

$$0\ if\ b == 0\ else\ a/b$$

verhindert Division durch Null und liefert im eigentlichen Fehlerfall das Ergebnis 0.

2.3 Anweisungen

Ausdrücke können keine Wiederholungen von Vorgängen ausdrücken, welche aber notwendig sind, um viele Algorithmen zu implementieren.

Beispiel: Fakultätsfunktion

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

Bsp:

$$\begin{aligned} 5! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \\ 2! &= 1 \cdot 2 = 2 \end{aligned}$$

D.h. der Ausdruck zur Fakultätsberechnung ist unterschiedlich groß für unterschiedliche Argumente n (wächst mit wachsendem n).

Wie kann das Problem aber algorithmisch gelöst werden? Eine Lösung ist die schrittweise Multiplikation der Faktoren.

n	$n!$
1	1
2	$1 \cdot 2 = 2$
3	$2 \cdot 3 = 6$
4	$6 \cdot 4 = 24$
5	$24 \cdot 5 = 120$
$\vdots$	$\vdots$

Hierbei ist es nicht nötig sich alle vorherigen Ergebnisse zu merken. Das jeweils letzte Ergebnis reicht aus. Man kann von einem Ergebnis auf das nächste schließen:
$n! = (n-1)! \cdot n$

In Programmiersprachen können Werte in Variablen gespeichert werden, wodurch man sich in einem Programm Werte merken kann (Belegung von Variablen mit Werten). Solche Belegungen sind nun nicht mehr nur am Anfang des Programms erlaubt, sondern an jeder beliebigen Stelle. Anstelle von Variablenbelegung sagt man auch *Zuweisung*. In *imperativen Programmiersprachen* können Zuweisungen auch die existierenden Belegungen überschreiben.

Beispiele für Zuweisungen:

```
x = 3
z = 4
y = x if x>z else z
y = y * 2
```

Hierbei dürfen auf der rechten Seite der Zuweisung beliebige Ausdrücke stehen, auf der linken nur Variablen (Syntax).

Die Semantik der Zuweisung ergibt sich wie folgt: Sind alle im Ausdruck verwendeten Variablen belegt, so kann der Ausdruck zu einem Wert ausgewertet werden und die Variable wird mit diesem Wert belegt. Hierbei wird der Ausdruck stets mit der vor der Zuweisung gültigen Variablenbelegung ausgewertet. Außerdem können Zuweisungen hintereinander geschrieben und damit dann nacheinander (*sequentiell*) ausgeführt werden.

```
                <- Hier ist die Variable y noch nicht belegt.
y = 2
                <- Hier ist y mit 2 belegt.
y = y * 2
                <- Hier ist y mit 4 belegt.
```

Möchte man mehreer Zuweisungen in eine Zeile schreiben, muss man sie mit einem Semikolon von einander abtrennen, damit Python erkennt, wo die erste Zuweisung endet und die nächste Zuweisung beginnt.

```
fac = fac * n; n = n+1
```

Um die Semantik eines solchen Programms nachzuvollziehen, können wir die Programmausführung zeilenweise simulieren. Hierzu geben nummerieren wir die Programmzeilen durch:

```
fac = 2          # 1
n = 3            # 2
fac = fac * n    # 3
n = n+1          # 4
print(fac)       #5
```

Danach erstellen wir eine Tabelle, die in der ersten Spalte die Progrmmzeilen protokolliert und für jede vorkommende Variable eine zusätzliche Spalte enthält. Außerdem verwenden wir eine zusätzliche Spalte für Bemerkungen. Danach werden die Zuweisungen der Reihe nach durchgeführt und die Belegung der Variable in der entsprechenden Spalte hinter der aktuellen Programmzeile notiert.

Im Beispiel:

Programmzeile (PZ)	fac	n	Bemerkungen
1	2		Anfangsbelegung der Variablen fac
2		3	Anfangsbelegung der Variablen n
3	6		neue Belegung für fac
4		4	neue Belegung für fac
5			Ausgabe: 6

Um nun nach und nach die Fakultät zu berechnen benötigen wir noch eine Wiederholungsmöglichkeit, auch *Schleife* genannt.

2.3.1 *while*-Schleife:

In jeder imperativen Programmierung findet man eine sogenannte while-Schleife. Wir betrachten das folgende Beispiel einer while-Schleife in Python:

Bsp:

```
n = 1          #1
while n<4:     #2
  n = n + 1    #3

print(n)       #4
```

Hierbei nennt man den Teil zwischen while und dem Doppelpunkt *die Bedingung* und den Teil, der wiederholt weden soll *den Rumpf der Schleife*. Viele Programmiersprachen verwenden Schlüsselwörter (begin-end oder do-end) oder Klammern (meist geschweifte Klammern) um Anfang und Ende des Rumpfs zu markieren. Hierbei hat sich für eine bessere Lesbarkeit des Codes herausgestellt, dass es hilft, den Rumpf in Relation zum Kopf der Schleife einzurücken. Diese Idee wurde in Python forciert, hierfür aber auf Schlüsselworte verzichtet.

In obigem Programm ist es also wichtig, dass die Zeile `n = n + 1` eingerückt ist. Hierbei gibt die Spalte des ersten Symbols nach dem Doppelpunkt an, wie weit alle Zeilen des Rumpfs eingerückt werden müssen. Zum syntaktischen Beenden des Rumpfes muss der Programmierer wieder die Einrückungstiefe des while verwenden und in der ersten Spalte weiterschreiben.

Bedeutung: So lange die Bedingung gilt (also zu True ausgewertet wird), wird der Rumpf wiederholt. Die Bedingung wird jedesmal vor der Ausführung des Rumpfs überprüft.

Programmausführung:

PZ	n	Bemerkung
1	1	
2		Bedingung: n<4 $\rightsquigarrow$ True
3	2	
2		Bedingung: n<4 $\rightsquigarrow$ True
3	3	
2		Bedingung: n<4 $\rightsquigarrow$ True
3	4	
2		Bedingung: n<4 $\rightsquigarrow$ False
4		Ausgabe: 4

Am Ende des Programms hat die Variable n den Wert 4, welchen wir mit Hilfe von print ausgeben.

Beachte hierbei, dass in den Argumentklammern von print auch hier ein Ausdruck steht, nämlich der Ausdruck, welcher nur aus der Variablen n besteht.

Nun können wir die Fakultät berechnen:

```
n_max = 4          #1    # zu berechnende Fakultaet
n = 0              #2    # Zaehler
fac = 1            #3    # (Teil-)Ergebnis
while n < n_max:   #4
  n = n + 1        #5
  fac = fac * n    #6

print(fac)         #7
```

Wieder geben wir das Ergebnis der Berechnung mit Hilfe von print am Ende des Programms aus.

Programmsimulation: Hierzu haben wir in unserem Programm alle Semikolons durchnummeriert (im Kommentar):

PZ	n_max	n	fac	Bemerkung
1	4			
2		0		
3			1	
4				n < n_max ⇝ True
5		1		
6			1	
4				n < n_max ⇝ True
5		2		
6			2	
4				n < n_max ⇝ True
5		3		
6			6	
4				n < n_max ⇝ True
5		4		
6			24	
4				n < n_max ⇝ False
7				Ausgabe: 24

Das Ergebnis lautet also 24 und wird ausgegeben.

Zuweisungen, Sequenzen und while-Schleifen nennt man auch **Anweisungen** (Statements) und sie sind neben Ausdrücken eine weitere wichtige Struktur in imperativen Sprachen. Zunächst betrachten wir ihre Syntax nur an Beispielen. Später werden wir diese auch noch formalisieren.

Als nächstes wollen wir uns noch einmal mit dem Ausdruck zur Berechnung des Maximums beschäftigen. Zunächst hatten wir das Ergebnis dieses Ausdrucks nur ausgegeben. Nun können wir dieses Ergebnis auch in einer Variablen speichern und es so in der weiteren Berechnung verwenden:

```
n = ...
m = ...

max = n if n>=m else m
```

Die Verzweigung mittels `if_then_else` ist nicht nur in Ausdrücken möglich. Wir können Sie auch auf Anweisungsebene verwenden und zwischen unterschiedlichen möglichen Anweisungen Verzweigen:

```
n = ...
m = ...

if n >= m:
  max = n
else:
  max = m

print(max)
```

Die Bedingung ist auch hier weiterhin ein Ausdruck, der einen booleschen Wert als Ergebnis liefert. Der then- und der else- Teil sind nun aber beliebige Anweisungen, welche entsprechend ausgeführt werden, wenn die Bedingung zu True bzw. False ausgewertet wird. Wieder verwendet Python Einrückung, um die entsprechenden Rümpfe um then- und else-Fall abzugrenzen.

Bei Verzweigungen auf Anweisungsebene kann es manchmal auch sinnvoll sein, den else-Fall weg zu lassen. Dies entspricht einer Verzweigung, bei der der else-Fall leer wäre. Es handelt sich also nur noch um eine so genannte $if - then$-Anweisung oder *bedingte Anweisung*, wobei der Begriff bedingte Anweisung ausdrückt, das die Anweisung(ssequenz) im then-Fall nur unter einer gegeben Bedingung ausgeführt wird.

Bei unserer Maximumsberechnung können wir die bedingte Anweisung wie folgt einsetzen:

```
n = ...
m = ...

max = n
if m > max:
    max = m
```

Will man das Maximum von drei Werten bestimmen, erkennt man den Vorteil der bedingten Anweisung noch deutlicher:

```
n = ...
m = ...
o = ...

max = n
if m > max:
    max = m
if o > max:
    max = o
```

Eine alternative (platzsparendere) Einrückung sieht wie folgt aus:

```
n = ...
m = ...
o = ...

max = n
if m > max: max = m
if o > max: max = o
```

Diese empfiehlt sich aber nur, wenn der Rumpf nur eine Zeile lang ist, da man sonst bei Veränderung der Bedingung nachträglich ale Zeilen des Rumpfes anpassen muss.

Durch das schrittweise Ändern der Variablen max benötigen wir nur zwei bedingte Anweisungen, während wir in dem $if - then - else$-Ausdruck für das Maximum von drei Werten insgesamt vier (verschachtelte) $if - then - else$ Anwendungen benötigt haben.

Zum Abschluss dieses Abschnitts wollen wir noch einmal die Fakultätsberechnung betrachten. Anstatt die Faktoren hochzuzählen können wir auch runterzählen (mit Kommutativität von $\cdot$):

$$1 \cdot 2 \cdot \ldots \cdot n = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

Im Programm können wir diese Idee verwenden, um eine Variable, hier konkret n_max, zu sparen:

```
n = 4              #1      zu berechnende Fakultaet
fac = 1            #2
while n>0:         #3
  fac = fac * n    #4
  n = n-1          #5

print(fac)         #6
```

PZ	n	fac	Bemerkung
1	4		
2		1	
3			$n > 0 \rightsquigarrow$ True
4		4	
5	3		
3			$n > 0 \rightsquigarrow$ True
4		12	
5	2		
3			$n > 0 \rightsquigarrow$ True
4		24	
5	1		
3			$n > 0 \rightsquigarrow$ True
4		24	
5	0		
3			$n > 0 \rightsquigarrow$ False
6			Ausgabe: 24

Beachte: Zwischenergebnisse sind keine Fakultätsergebnisse mehr. Die Eingabevariable n wird verändert $\Rightarrow$ Wert geht verloren. Ungünstig, falls er nochmals benötigt wird!

2.3.2 Größter gemeinsamer Teiler

Um das Programmieren mit Schleifen weiter zu üben, wollen wir uns die Bestimmung des größten gemeinsamen Teilers zweier gegebener Zahlen anschauen. Der größte gemeinsame Teiler zweier Zahlen wird z.B. beim Kürzen von Brüchen verwendet, wo man Nenner und Zähler durch ihren größten gemeinsamen Teiler dividiert.

Def.: (ggT)

Gegeben: $a, b \in \mathbb{Z}$

Gesucht: $ggT(a, b) = c \in \mathbb{N}$, so dass c teilt a ohne Rest und c teilt b ohne Rest und für alle weiteren Teiler d von a und b gilt $c > d$.

Als Beispiel betrachten wir folgende Zahlen:
21 hat die Teiler: 1, 3, 7, 21
18 hat die Teiler: 1, 2, 3, 6, 9, 18
Somit ist der größte gemeinsame Teiler: $ggt(18, 21) = 3$

0 hat die Teiler: 1, 2, 3, 4, 5, 6, 7, ...
Somit gilt für alle $a \neq 0$: $ggt(a,0) = ggt(0,a) = a$.

$ggt(0,0)$ ist nicht definiert, da alle Zahlen die 0 ohne Rest teilen und es somit keine größte Zahl gibt, die 0 teilt.

Wie könnte nun eine mögliche Lösung dieses Problems aussehen? Bevor wir eine geschickte Lösung mit einem etwas geschickteren Algorithmus verwenden, lernen wir eine Methode kennen, die in vielen Fällen (allerdings oft nicht besonders geschickt) zum Ziel führt.

2.3.3 Aufzählen und Überprüfen

Ein großer Vorteil eines Computers gegenüber einem Menschen ist die Fähigkeit, viele Werte sehr schnell aufzählen und gewisse Eigenschaften für diese Werte überprüfen zu können. Somit können viele Probleme, bei denen der Bereich der möglichen Lösungen endlich ist und aufgezählt werden kann, mit der Programmiertechnik Aufzählen und Überprüfen gelöst werden. Dies ist auch für den ggT der Fall.

Der ggT von zwei Zahlen liegt sicherlich zwischen 1 und der kleineren der beiden Zahlen. Wir können also diese Werte der Reihe nach aufzählen und jeweils überprüfen, ob die entsprechende Zahl beide gegebenen Zahlen ohne Rest teilt.

Für die Überprüfung, ob eine Zahl eine andere ohne Rest teilt, ist der Modulo-Operator sehr hilfreich, welcher den Rest einer ganzzahligen Division liefert. Falls a und b ganzzahlige Werte sind, so liefert / die ganzzahlige Division und % den Rest der ganzzahligen Division.

Bsp.:

$$12/9 \rightsquigarrow 1 \quad 12\%9 \rightsquigarrow 3$$
$$16/3 \rightsquigarrow 5 \quad 16\%3 \rightsquigarrow 1$$

Der Algorithmus, welcher alle möglichen Teiler aufzählt und überprüft kann wie folgt in Python realisiert werden:

```
a = 12              # natuerliche Zahlen, fuer die der
b = 9               # ggT bestimmt werden soll
test_grenze = a if a<b else b
i = 1
ggt = 1
while i<=test_grenze:
  if a%i==0 and b%i==0:
    ggt=i

  i = i + 1

print(ggt)
```

Beachte, dass der Ausdruck a%i==0 && b%i==0 wegen der Präzedenzen der verwendeten Operatoren, so geklammert ist: ((a%i)==0) && ((b%i)==0), d.h. &&(logisches Und) bindet schwächer als == bindet schwächer als %. Außerdem heben wir mit der Zuweisung i = i + 1 den Rumpf der bedingten Anweisung auf, gehen aber zum Rumpf der while-Schleife zurück, da die Zuweisung exakt unter dem if steht.

Da wir den größten gemeinsamen Teiler suchen ist es für diese Aufgabe aber sinnvoller die Zahlen von oben nach unten aufzuzählen, da man dann bei der ersten Zahl, die beide gegebenen Zahlen ohne Rest teilt aufhören kann und den ggT ausgeben kann. Das Speichern des letzten Teilers wird überflüssig.

Es ergibt sich folgendes, einfacheres ggT-Programm:

```
a = 12              # natuerliche Zahlen, fuer die der
b = 9               # ggT bestimmt werden soll
ggt = a if a<b else b
while a%ggt!=0 or b%ggt!=0:
  ggt = ggt-1

print(ggt)
```

Beachte wieder, dass der Ausdruck a%ggt!=0 *or* b%ggt!=0 wegen der Präzedenzen der verwendeten Operatoren, so geklammert ist: ((a%ggt)!=0) *or* ((b%ggt)!=0), d.h. *or* (logisches Oder) bindet schwächer als != bindet schwächer als %.

Problematisch sind nun noch die Randfälle $a = 0$ und/oder $b = 0$. Hier liefert das Programm einen Laufzeitfehler. Diese müssen nun noch explizit vor der Schleife abgefangen werden, was das Programm aber leider etwas aufbläht:

```
a = 12              # natuerliche Zahlen, fuer die der
b = 9               # ggT bestimmt werden soll
if a==0:
  if b==0:
    print('ggT nicht definiert')
  else:
    print(b)
else:
  if b==0:
    print(a)
  else:
    ggt = a if a<b else b
    while a%ggt!=0 or b%ggt!=0:
      ggt = ggt-1

    print(ggt)
```

Hier zeigt sich, dass es beim Testen der Programme auch wichtig ist, alle Randfälle systematisch zu überprüfen.

2.3.4 Euklidischer Algorithmus

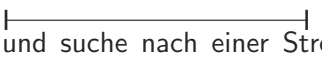
Bei großen Zahlen liefert die Aufzählen-und-Testen-Methode die Lösung leider nicht mehr in akzeptabler Zeit. Wir betrachten deshalb eine effizientere Lösung, wie sie bereits ca. 300 v. Chr. von dem griechischen Mathematiker Euklid gefunden wurde.

Gegeben zwei Strecken

Bestimme eine Strecke, mit der man beide Strecken „messen'— kann, d.h. die in beide Strecken ganz „hineinpasst'—.
Beide gegebenen Strecken sollen also **Vielfache** der gesuchten Strecke sein. Hier:

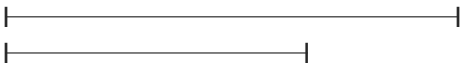
Wie findet man so eine Strecke?
Wenn beide Strecken gleich lang sind, passen sie natürlich in die jeweils andere hinein und es ist die gesuchte Strecke.
Wenn nicht: ziehe die kürzere Strecke von der Längeren ab.

Ergibt im Beispiel die Strecke:



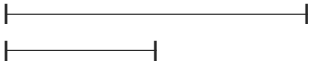
und suche nach einer Strecke, die in die kürzere Strecke und in die Strecke, die durch Abziehen der kürzeren von der längeren Strecke entsteht, hineinpasst.
Da die gesuchte Strecke sowohl in die kürzere als auch in die längere Strecke hineinpassen soll, muss sie auch in die Differenz der beiden Strecken hineinpassen.

Finde also Strecke, die in diese beiden Strecken passt:

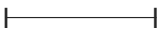


Nach Abziehen erhalten wir die Strecke:

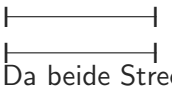
Finde also eine Strecke, die in diese beiden Strecken passt:



Die Differenz ergibt:



Als nächstes suchen wir also die Strecke, die in die verbleibenden beiden Strecken passen:



Da beide Strecken gleich lang sind, ist dies die gesuchte Strecke.

Wir setzen das Verfahren also so lange fort, bis beide Strecken gleich lang sind. Diese Strecke ist dann die größte mögliche Strecke, die beide Strecken teilt. Alternativ kann man auch noch ein weiteres Mal abziehen und enden, wenn eine der beiden Strecken nur noch die Länge 0 hat.

Der Euklidische Algorithmus eignet sich also insbesondere auch zur Bestimmung des ggTs. Wir arbeiten ähnlich wie bei der Idee mit den Strecken, allerdings enden wir nicht wenn beide Zahlen gleich sind, sondern erst, wenn eine der beiden 0 ist. Hierdurch bestimmt unser Algorithmus auch den ggt korrekt, falls eine der beiden Zahlen 0 ist. Den Fall, dass beide Zahlen 0 sind, müssen wir dann noch separat überprüfen. **Bsp.:** $ggT(15, 10)$

$$\begin{aligned} 15 \geq 10 \quad &\Rightarrow \quad 15 - 10 = 5 \\ 10 \geq 5 \quad &\Rightarrow \quad 10 - 5 = 5 \end{aligned}$$

Also $ggT(15, 10) = 5$.

Bsp.: $ggT(12, 9)$

$$\begin{array}{rcl} 12 \geq 9 & \Rightarrow & 12 - 9 = 3 \\ 9 \geq 3 & \Rightarrow & 9 - 3 = 6 \\ 6 \geq 3 & \Rightarrow & 6 - 3 = 3 \\ 3 \geq 3 & \Rightarrow & 3 - \underline{3} = 0 \end{array}$$

Also $ggT(12, 9) = 3$.

$$\begin{array}{rcl} ggT(12, 9) & = & ggT(3, 9) \\ & = & ggT(3, 6) \\ & = & ggT(3, 3) \\ & = & ggT(0, 3) \\ & = & 3 \end{array}$$

Wie können wir diese mathematische Definition nun in Python realisieren? Wir wiederholen immer wieder die gleichen Schritte, so dass wir für die Programmierung eine while-Schleife verwenden sollten. Im Rumpf der Schleife muss entweder a=a−b oder b=b−a gerechnet werden, je nachdem, welcher Wert größer ist. Der Schleifenrumpf sollte so lange wiederholt werden, wie weder a noch b Null sind:

```
a = 12                          #1   # initiale Werte
b = 9                           #2
while a != 0 and b != 0:        #3
  if a < b:                     #4
    b = b - a                   #5
  else:                         #6
    a = a - b                   #7
```

Wenn wir also das Schleifenende erreichen, wissen wir, dass mindestens eine der Variablen den Wert Null hat. Die andere Variable enthält dann den ggT. Nun müssen wir nur noch herausfinden, welche Null ist. Hierbei identifizieren wir insbesondere noch den Fall, dass beide Null sind und der ggT nicht definiert ist.

```
a = 12                                  #1   # initiale Werte
b = 9                                   #2
while a != 0 and b != 0:                #3
  if a < b:                             #4
    b = b - a                           #5
  else:                                 #6
    a = a - b                           #7

if a == 0:                              #8
  if b == 0:                            #9
    print('ggT nicht definiert')        #10
  else:                                 #11
    print(b)                            #12
else:                                   #13
  print(a)                              #14
```

Die Programmausführung sieht dann wie folgt aus:

PZ	a	b	Bemerkung
1	12		
2		9	
3			a!=0 and b!=0 $\rightsquigarrow$ True
4			a < b $\rightsquigarrow$ False
6			
7	3		
3			a!=0 and b!=0 $\rightsquigarrow$ True
4			a < b $\rightsquigarrow$ True
5		6	
3			a!=0 and b!=0 $\rightsquigarrow$ True
4			a < b $\rightsquigarrow$ True
5		3	
3			a!=0 and b!=0 $\rightsquigarrow$ True
4			a < b $\rightsquigarrow$ False
6			
7	0		
8			a==0 $\rightsquigarrow$ True
13			
14			Ausgabe: 3

Betrachte initiale Belegung: $a = 3$, $b = 0$

PZ	a	b	Bemerkung
1	3		
2		0	
3			a!=0 and b!=0 $\rightsquigarrow$ False
8			a==0 $\rightsquigarrow$ False
13			
14			Ausgabe: 3

Weitere sinnvolle Testfälle wären: $a = 0$, $b = 3$ und $a = 0$, $b = 0$. Außerdem können wir auch für größere Werte die Ausgaben unserer unterschiedlichen ggT-Implementierungen vergleichen und uns so von der Korrektheit unser Programme überzeugen.

Der Algorithmus kann auch noch weiter optimiert werden, wenn man die Subtraktion durch eine Division mit Restbildung ersetzt. Näheres hierzu in der Übung.

2.3.5 *for*-Schleife

Bei vielen Iterationen weiß man genau, wie oft der Schleifen-Rumpf ausgeführt werden soll. Außerdem benötigt man oft eine Zählvariable, welche die Iterationen zählt und automatisch inkrementiert wird (n bei der ersten Version der Fakultät). Zu diesem Zweck kann man *for*-**Schleifen** verwenden. Die for-Schleife in Python verwendet man wie folgt:

Bsp.:

```
n = 4                          #1
fac = 1                        #2
for i in range(1,n+1):         #3
  fac = i * fac                #4

print(fac)                     #5
```

Das Schlüsselwort for leitet die for-Schleife ein. Als nächstes wird die Zählvariable (hier i) festgelegt. Hinter dem Schlüsselwort in gibt man dann die Schleifengrenzen (Startwert und Wert hinter dem Endwert) mit Hilfe der zweistelligen Funktion range an. Nach dem Doppelpunkt folgt dann eingerückt der Rumpf, welcher ggf. wiederholt wird. Hierbei nimmt die Zählvariable der Reihe nach die Werte vom Start- bis zum Endwert, aber exklusive des Endwertes an.

Mit Hilfe der Programmzeilentabelle können wir die Ausführung des Programms simulieren:

PZ	n	fac	i	Bemerkung
1	4			
2		1		
3			1	
4		1		
3			2	
4		2		
3			3	
4		6		
3			4	
4		24		
3				Würde i weiter erhöht, wäre der Endwert (5) erreicht
5				Ausgabe: 24

Die Zählvariable wird vor jeder nächsten Iteration hochgezählt. Der letzte Durchlauf wird mit dem Wert vor dem Endwert als Belegung für die Zählvariable durchgeführt.

Beachte: Eine *for*-**Schleife** wird immer beendet (terminiert immer), im Gegensatz zur *while*-Schleife:

Bsp.:

```
while True:
  ...
end
```

terminiert nicht.

Neben falschen Ergebnissen oder Programmabstürzen, kann ein Programm auch fehlerhafter Weise in eine Endlosschleife laufen. Dies ist z.B. möglich, wenn vergessen wird, die Zählvariable zu verändern:

```
while n>0:
  fac = fac * n    #n=n-1 vergessen
end
```

Dies ist bei der for-Schleife nicht möglich.

In Python terminiert die for-Schleife tatsächlich immer (außer der Rumpf terminiert nicht, z.B. weil er eine nicht-terminierende $while$-Schliefe enthält). Selbst wenn wir im Rumpf die Zählvariable verändern (was man aber auf keinen Fall machen sollte, da es zu unverständlichen Programmen führt!), wird der Wert der Zählvariable, vor dem nächsten Schleifendurchlauf auf den nächsten Zählwert gesetzt.

Dies gilt nicht für alle imperativen Programmiersprachen. Dennoch sollte man auch in anderen Sprachen die Zählvariable nicht verändern und for-Schleifen so verwenden, wie hier vorgestellt.

Genau wie bei der while-Schleife kann es aber auch bei der for-Schleife sein, dass der Rumpf gar nicht durchlaufen wird. Bei der $while$-Schleife wird die Bedingung überprüft, bevor der Rumpf ausgeführt wird. Bei der for-Schleife wird ebenfalls vorher überprüft, ob der Endwert beim weiteren erhöhen erreicht wurde:

```
x = 0
for i in range(5,3):
  x = x+1

print(x)
```

gibt 0 aus.

Bei

```
for i in range(5,6):
  ...
end
```

wird der Rumpf genau einmal mit $i = 5$ ausgeführt.

Nun kennen wir die beiden wichtigsten Schleifenarten: while- und for-Schleife. Beide finden sich so ähnlich in fast allen imperativen Programmiersprachen. Als Pragmatik der imperativen Programmierung gilt, dass eine for-Schleife verwendet werden sollte, wenn bei Schleifenbeginn bekannt ist, wie oft eine Wiederholung statt finden soll. Dies bedeutet nicht zwangsläufig, dass bereits zur Zeit der Programmierung die genaue Anzahl bekannt sein muss und die Schleifengrenzen immer feste Zahlen sein müssen. Es kann auch sein, dass die Grenze für die Wiederholung nur als Wert in einer Variablen bekannt ist und z.B. das Ergebnis einer anderen Berechnung oder auch einer Benutzereingabe (bekommen wir später) sein kann.

2.4 Syntaxbeschreibung

Bevor wir weiter in die Feinheiten der Programmierung einsteigen, wollen wir uns noch mit der Frage beschäftigen, was eigentlich genau gültige Python-Programme sind. Bisher haben wir die Syntax an Beispielen kennen gelernt und dann entsprechend auf andere Programme übertragen. Es stellt sich aber die Frage, ob man auch genau festlegen kann, welches die gültigen Pythonprogramme sind. Hierzu bietet die Informatik unterschiedliche Formalismen, welche es ermöglichen Sprachen formal zu definieren.

Die Ansätze gehen in der Regel auf den amerikanischen Linguisten Noam Chomsky zurück. Formal gesehen ist eine Sprache eine (möglicherweise unendliche) Menge von Wörtern.[2] Als Beispiel betrachten wir die folgenden formalen Sprachen:

- Die leere Sprache: $\emptyset$ enthält gar kein Wort.
- Eine endliche Sprachen mit genau drei Wörtern: $\{Studierende, hallo, liebe\}$
- Die Sprache, aller Wörter, die aus den Buchstaben G, C, T, A bestehen: $\{\varepsilon$[3]$, G, C, T, A, GG, GC, GT, GA, CG, \ldots\}$. Diese Sprache ist die Sprache unseres Erbguts (der DNA). Jeder Buchstabe repräsentiert hierbei eine Nukleinbase.
- Die Sprache aller Palindrome, also der Wörter, die von vorne und hinten gelesen gleich sind: $\{\varepsilon, a, b, c, \ldots, aa, bb, cc, \ldots, aaa, aba, aca, \ldots, aaaa, abba, \ldots, aaaaa, aabaa, abcba, \ldots, otto, \ldots\}$

Zur formalen Beschreibung solcher Sprachen verwendet man in der Informatik (und auch Linguistik) unterschiedliche Formalismen: Syntaxdiagramme, Grammatiken und die Backus-Naur-Form (BNF), welche wir hier näher betrachten werden.

Die *Backus-Naur-Form (BNF)* stellt einen Formalismus zur Beschreibung von Sprachen (insbesondere Programmiersprachen) dar. Es werden Nichtterminalsymbole (beginnen mit Großbuchstaben) und Terminalsymbole (Zeichen in ' ') unterschieden. Nichtterminalsymbole stellen keine Elemente der zu definierenden Sprache dar. Vielmehr sind sie Strukturelemente, welche weiter verfeinert und letzendlich zu einer Folge von Terminalsymbolen abgeleitet werden.

Als erstes Beispiel betrachten wir eine BNF, die die Sprache des Erbguts beschreibt. Wir verwenden nur ein Nichtterminalsymbol DNA.

$$DNA \quad ::= \quad 'G'\ DNA \ \mid \ 'C'\ DNA \ \mid \ 'T'\ DNA \ \mid \ 'A'\ DNA \ \mid$$

Die möglichen Ableitungen für das Nichterminalsymbol DNA werden hinter dem speziellen Symbol ::= notiert. Hierbei gibt es fünf unterschiedliche Möglichkeiten, wie wir das Nichtterminalsymbol DNA ableiten können. Wir trennen die unterschiedlichen Möglichkeiten durch |. Jede einzelne Möglichkeit bezeichnen wir als *Regel* und sprechen von der Regel $DNA ::= \ 'C'\ DNA$, wenn wir eine einzelne Regel benennen. Die letzte Regel lautet $DNA ::=$.

[2] Man kann auch natürliche Sprachen, z.B. die deutsche Sprache, als formale Sprache sehen. Die Sätze würde man dann als Wörter der Sprache bezeichnen. Die einzelnen Wörter des Deutschen wären dann die Buchstaben der formalen Sprache.

[3] Die Sprache enthält insbesondere auch das leere Wort, also ein Wort, welches aus keinem Zeichen besteht. Dieses notieren wir hier als ε.

Nun wollen wir die spezielle Nukleinbasensequenz CAG mit Hilfe dieser BNF herleiten. Dazu beginnen wir mit dem Nichtterminalsymbol, aus dem wir die Sequenz ableiten wollen. In jedem Ableitungsschritt darf jeweils nur ein vorkommendes Nichtterminalsymbol durch eine seiner rechten Seiten ersetzt werden. Kommen keine Nichtterminalsymbole mehr vor, hat man ein gültiges Wort der beschriebenen Sprache abgeleitet:

$$\begin{aligned} & DNA \\ \Rightarrow\ & C\ DNA \\ \Rightarrow\ & C\ A\ DNA \\ \Rightarrow\ & C\ A\ G\ DNA \\ \Rightarrow\ & C\ A\ G \end{aligned}$$

Im letzten Schritt verwenden wir die fünfte Regel DNA ::= und ersetzen das Nichtterminalsymbol DNA durch die leere Zeichenfolge, wodurch das Nichtterminalsymbol DNA verschwindet und die gewünschte Zeichenfolge erzeugt wurde.

Als nächstes Beispiel betrachten wir die folgende BNF:

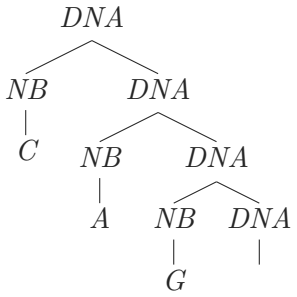
$$\begin{aligned} DNA &::= NB\ DNA \mid \\ NB &::= \ 'G' \mid 'C' \mid 'T' \mid 'A' \end{aligned}$$

Auch diese Grammatik ermöglicht es uns, alle möglichen DNA-Sequenzen herzuleiten. Als Beispiel betrachten wir noch einmal eine Ableitung der Nukleinbasensequenz CAG:

$$\begin{aligned} & DNA \\ \Rightarrow\ & NB\ DNA \\ \Rightarrow\ & NB\ NB\ DNA \\ \Rightarrow\ & NB\ NB\ NB\ DNA \\ \Rightarrow\ & NB\ NB\ NB \\ \Rightarrow\ & C\ NB\ NB \\ \Rightarrow\ & C\ A\ NB \\ \Rightarrow\ & C\ A\ G \end{aligned}$$

Auf Grund des zusätzlich verwendeten Nichtterminalsymbols NB ist die Ableitung länger geworden. Dennoch beschreibt diese BNF die gleiche Sprache, wie die erste BNF. Wir haben über das Nichtterminalsymbol NB lediglich eine weitere Struktur eingeführt, die besser verdeutlicht, dass alle vier Buchstaben Nukleinbasen repräsentieren.

Vergleichen wir die beiden Ableitungen des Wortes CAG noch einmal, stellen wir fest, dass die Stelle, an der wir weiter ersetzen können, nicht mehr eindeutig definiert ist. Vielmehr gibt es mehrere mögliche Ableitungen, welche das Wort CAG herleiten und auch der Schreibaufwand wächst, da immer nur ein Nichtterminalsymbol pro Schritt ersetzt werden darf. Einfacher ist es deshalb anstelle der Ableitung den Ableitungsbaum für ein gegebenes Wort hinzu schreiben:



Die inneren Knoten sind mit Nichtterminalsymbolen beschriftet. Die Blätter mit Terminalsymbolen (bzw. dem leeren Wort). Die Kinder eines Nichtterminalsymbolknotens, sind jeweils durch alle Symbole der rechten Seite der Regel definiert. Hierbei müssen Anzahl der entsprechenden Symbole und deren Reihenfolge beachtet werden. Das abgeleitete Wort erhält man, wenn man die Blätter des Ableitungsbaums von links nach rechts liest.

Die Reihenfolge, in der die Nichtterminalsymbole durch ihre rechten Seiten ersetzt werden, ist im Ableitungsbaum egal, so dass er in der Regel kompakter ist, als eine schrittweise Ableitung.

BNF für Python-Werte

Als nächstes Beispiel betrachten wir eine BNF, die die Sprache aller möglichen Python-Werte definiert. Zunächst beschränken wir uns auf (ganze) Zahlen und die booleschen Werte als Nichtterminalsymbol Val:

$$\begin{aligned} Val &::= Num \mid '-'\ Num \mid 'True' \mid 'False' \\ Num &::= Dig \mid Dig\ Num \\ Dig &::= '0' \mid \ldots \mid '9' \end{aligned}$$

Beachte, dass wir für ganze Zahlen keine führenden Nullen verbieten. Dies wäre auch möglich, soll aber, genau wie Gleitkommazahlen, als Übung verfeinert bzw. hinzu genommen werden.

BNF für Python

Nachdem wir nun Werte, wie sie in Python verwendet werden, definiert haben, können wir dieses Beispiel erweitern und die Sprache aller gültigen (Python-)Ausdrücke definieren. Hierbei beschreiben wir der Einfachheit halber nur vollständig geklammerten Ausdrücke Exp[4]:

$$\begin{aligned} Exp &::= Var \\ &\mid Val \\ &\mid '('\ Exp\ Op\ Exp\ ')' \\ &\mid Fun\ '('\ Exps\ ')' \\ &\mid '('\ Exp\ 'if'\ Exp\ 'else'\ Exp\ ')' \\ Exps &::= Exp \\ &\mid Exp\ ','\ Exps \\ Op &::= '+' \mid '-' \mid '*' \mid '/' \mid '**' \\ Fun &::= 'math.sqrt' \mid 'math.sin' \mid \ldots \\ Var &::= 'x' \mid 'y' \mid 'z' \mid \ldots \end{aligned}$$

[4] Wir gehen hier zunächst davon aus, dass es keine Präzedenzen gibt und die Ausdrücke vollständig geklammert werden müssen. Wir werden später noch sehen, wie Präzedenzen in der BNF ausgedrückt und Klammern vermieden werden können.

Die Variablen definieren wir hier nur beispielhaft. Eine genauere Definition soll ebenfalls als Übung erfolgen.

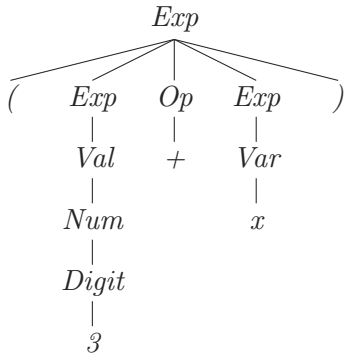
Als Beispiel für eine Ableitung in dieser BNF zeigen wir, dass das Wort $(math.sqrt((x * * 2))/x)$ ein vollständig geklammerter Ausdruck ist, d.h. dieses Wort aus dem Nichtterminal Exp abgeleitet werden kann.

Wieder darf in jedem Ableitungsschritt nur ein vorkommendes Nichtterminalsymbol durch eine seiner rechten Seiten ersetzt werden. Kommen keine Nichtterminalsymbole mehr vor, hat man ein gültiges Wort der beschriebenen Sprache abgeleitet:

$$
\begin{aligned}
& Exp \\
\Rightarrow\ & (Exp\ Op\ Exp) \\
\Rightarrow\ & (Exp\ /\ Exp) \\
\Rightarrow\ & (Exp\ /\ Var) \\
\Rightarrow\ & (Exp\ /\ x) \\
\Rightarrow\ & (Fun(Exps)\ /\ x) \\
\Rightarrow\ & (Fun(Exp)\ /\ x) \\
\Rightarrow\ & (math.sqrt(Exp)\ /\ x) \\
\Rightarrow\ & (math.sqrt((Exp\ Op\ Exp))\ /\ x) \\
\Rightarrow\ & (math.sqrt((Exp * * Exp))\ /\ x) \\
\Rightarrow\ & (math.sqrt((Exp * * Val))\ /\ x) \\
\Rightarrow\ & (math.sqrt((Exp * * Num))\ /\ x) \\
\Rightarrow\ & (math.sqrt((Exp * * Dig))\ /\ x) \\
\Rightarrow\ & (math.sqrt((Exp * * 2))\ /\ x) \\
\Rightarrow\ & (math.sqrt(((Var * * 2))\ /\ x) \\
\Rightarrow\ & (math.sqrt((x * * 2))\ /\ x)
\end{aligned}
$$

Beachte, dass die Zeichenfolge $x * * 2$ in diesem vollständig geklammerten Ausdruck zwei mal geklammert werden muss. Eine Klammer für die Funktionsanwendung ($math.sqrt$) und eine Klammer für die Operatoranwendung ($**$).

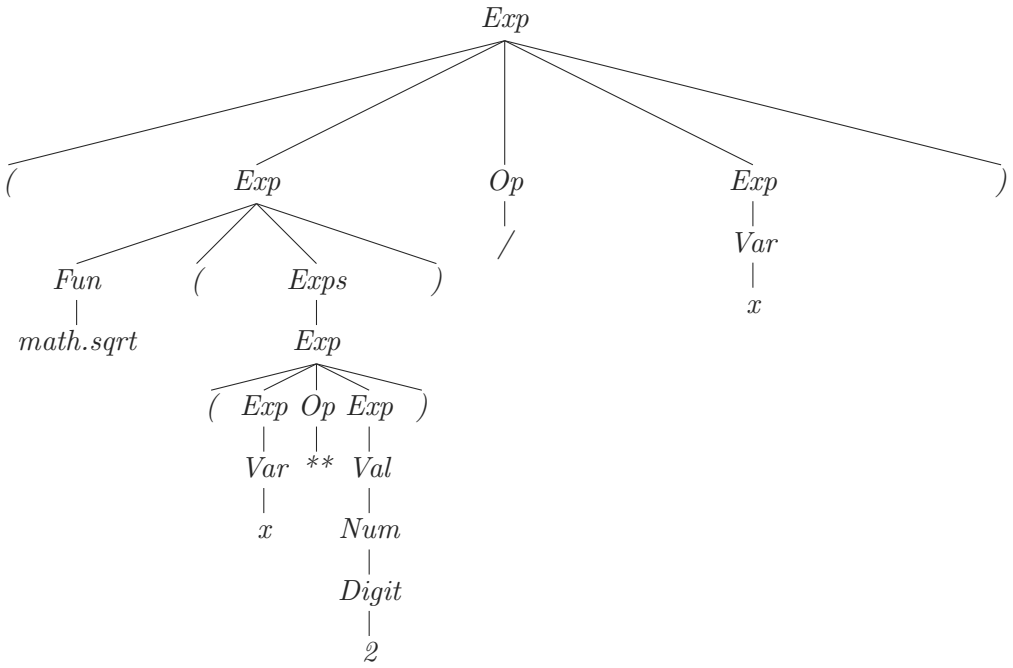
Bevor wir dieses Beispiel als Ableitungsbaum darstellen, betrachten wir den Ableitungsbaum für den Ausdruck $(3 + x)$:



Wieder sind die Nichtterminalsymbole die inneren Knoten des Ableitungsbaums. Die Wurzel ist mit dem Nichtterminal beschriftet, aus welchem man das Wort ableiten möchte (hier z.B. Exp). Die Kinder eines Nichtterminalknotens, entsprechen jeweils den Symbolen der rechten Seite der verwendeten Regel. So wurde bei der Wurzel zunächst die Regel $Exp\ ::=\ '('\ Exp\ Op\ Exp\ ')'$ angewendet, weshalb die Wurzel fünf Kindknoten hat, die von links nach rechts mit den entsprechenden Symbolen beschriftet sind.

Im fertigen Ableitungsbaum sind alle Blätter mit Terminalsymbolen beschriftet. Das abgeleitete Wort ergibt sich, indem man die Blätter des Baumes von links nach rechts abliest, hier also das Wort $(3 + x)$.

Der Ableitungsbaum für den Ausdruck $(math.sqrt((x ** 2))/x)$ sieht wie folgt aus:



Mit der hier vorgestellten BNF haben wir nun schon als wichtigen Teil von Pythons Syntax Ausdrücke definiert. Im folgenden werden wir diese schrittweise erweitern und auch Anweisungen in Python beschreiben.

BNF für Palindrome

Um vorher aber noch einmal klar zu machen, dass die BNF ein universeller Formalismus zur Beschreibung von Sprachen ist, wollen wir ihn vorher noch verwenden um eine Sprache zu beschreiben, die gar nichts mit Python zu tun hat, die Sprache der Palindrome.

Ein Palindrom ist ein Wort, welches von vorne und von hinten gelesen gleich ist. Beispiele sind **otto**, **rentner** oder (wenn man die Leer-/Satzzeichen ignoriert) **o genie, der herr ehre dein ego**. Wenn man Palindrome formal spezifizieren will, so kann man dies mit Hilfe folgender BNF machen:

$$\begin{aligned} Pal \;\; ::= \;\; & \; 'a' \; Pal \; 'a' \; \mid \; \ldots \; \mid \; 'z' \; Pal \; 'z' \\ \mid \;\; & \; 'a' \; \mid \; \ldots \; \mid \; 'z' \; \mid \end{aligned}$$

Hierbei werden natürlich nicht nur gültige Palindrome der deutschen Sprache beschrieben, sondern vielmehr alle Wörter (über dem Alphabet $'a'$ bis $'z'$), die von vorne und hinten gleich aussehen. Die letzte Regel Pal ::= wird verwendet, da auch das leere Wort eine Palindrom ist. Außerdem findet sie Anwendung, falls ein Palindrom hergeleitet wird, welches keinen einzelnen Buchstaben in der Mitte hatte, wie z.B. **otto**.

Untersucht man alle BNF die wir nun programmiert haben genauer, fällt auf, dass immer wieder ähnliche Konstruktionen auftreten, wie z.B. das optionale Vorkommen oder die

Wiederholung von Teilausdrücken. Um solche Strukturen einfacher ausdrücken zu können wurden zur BNF spezielle Konstrukte hinzugefügt, was die *erweiterte BNF (EBNF)* ergibt:

- Optionales Vorkommen eines Wertes: $[\mathbf{e}]$

  Dies können wir für die Definition von Werten verwenden:

  $$Val \;\; ::= \;\; [\,'-'\,]\; Num$$

- Optionale Wiederholung $\{\alpha\}$, d.h. α kann 0-mal, 1-mal, 2-mal,... vorkommen.
- Außerdem besteht noch die Möglichkeit die Alternative (|) auch in Gruppierungen zu verwenden. Ein Beispiel hierzu ist

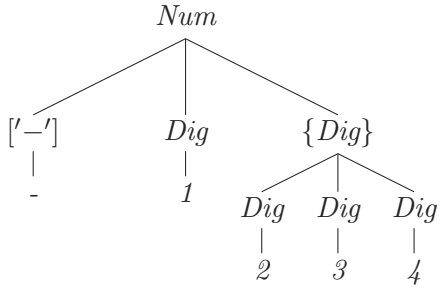
  $$S \;\; ::= \;\; 'a'\; (\; 'b' \mid 'c'\;)\; d$$

  mit abd und acd aus S ableitbar.

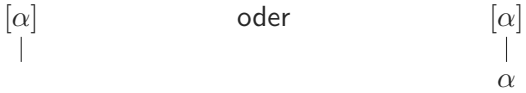
Mit diesen Abkürzungen können wir unsere BNF für Python-Ausdrücke an einigen Stellen kompakter aufschreiben:

$$\begin{array}{rcl} Exp & ::= & Var \\ & \mid & Val \\ & \mid & '('\; Exp\; Op\; Exp\; ')' \\ & \mid & Fun\; '('\; Exp\; \{\;,\; Exp\;\}\; ')' \\ Val & ::= & [\,'-'\,]\; Dig\; \{\; Dig\; \}\; \mid\; 'True'\; \mid\; 'False' \end{array}$$

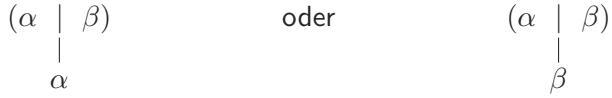
Beachte, dass sich für eine EBNF auch die Ableitungsbäume etwas ändert. Die inneren Knoten können nun auch mit den neuen Konstrukten der EBNF beschriftet sein, wobei außen immer ein neues Konstrukt steht. In einem Schritt erhält man dann so viele Kindknoten, wie benötigt werden, um dieses Konstrukt aufzulösen. Als Beispiel betrachten wir die Regel für Num bei der Ableitung des Wortes -1234:



Für die anderen Regeln gilt entsprechend:



und



Hierbei stehe α und β für beliebige Folgen von Terminal und Nichtterminalsymbolen und erweiterten Konstrukten der EBNF. Angewendet auf die konkreten Folgen, wie sie in der entsprechenden EBNF-Regel vorkommen.

Um nun auch die Syntax kompletter Python-Programme formal zu beschreiben, müssen wir uns zunächst noch einmal Gedanken um die Layout-Regeln von Python machen. Diese können leider nicht mit Hilfe einer EBNF beschrieben. Es ist aber möglich einen Präprozessor zu definieren, der die Einrückungen in Python analysiert und in passende Endemarkierungen für die definierten Rümpfe umwandelt. Hierbei wird jedes Ende eines Rumpfes durch ein passendes Symbol ‘¡:‘ (wir verwednen dieses Symbol als gegenstück zur Anfangsmarkierung mittels ‘:‘).

Als Beispiel betrachten wir das Ergebnis der Präprozessors für eine erste Version des euklidischen ggT-Programms (ohne Betrachtung der Randfälle):

```
a = 12                          #1
b = 9                           #2
while a != 0 and b != 0:        #3
  if a < b:                     #4
    b = b - a                   #5
  else:                         #6
    a = a - b                   #7
```

wird durch den Präprozessor in folgendes Codefragment übersetzt:

```
a = 12                          #1
b = 9                           #2
while a != 0 and b != 0:        #3
  if a < b:                     #4
    b = b - a                   #5
  <:                  # Ende des then-Rumpfs
  else:                         #6
    a = a - b                   #7
  <:                  # Ende des else-Rumpfs
<:                    # Ende des while-Rumpfs
```

Tatsächlich verwendet Python in der Implementierung einen ähnlichen Präprozessor. Beachte aber, das das Ergebnis kein gültiges Python mehr ist, aber zur weiteren syntaktischen Analyse verwendet werden kann.

In der nun erzeuugten Darstellung ist das Layout nicht mehr relevant und wir können die Syntax von Anweisungen in Python für diese Sprache definieren. Wir definieren das Nichtterminalsymbol Stm für Anweisungen (*statemants*):

$$
\begin{aligned}
Stm \;::=\; & \; Stm\ (\ ';'\ |\ '\backslash n'\ )\ Stm \\
& |\ Var\ '='\ \ Exp \\
& |\ 'while'\ Exp\ ':'\ Stm\ '<:' \\
& |\ 'for'\ Var\ 'in'\ 'range'\ '('\ Exp\ ','\ \ Exp\ ')'\ ':'\ Stm\ '<:' \\
& |\ 'if'\ Exp\ ':'\ Stm\ '<:'\ [\ 'else'\ ':'\ Stm\ '<:'\ ]\ \\
& |\ 'print'\ '('\ Exp\ ')'
\end{aligned}
$$

Wir haben gesehen, dass die Verwendung von EBNF-Konstrukten in einigen Fällen die Syntaxbeschreibung vereinfacht. Es stellt sich aber direkt die Frage, ob diese Erweiterung eine echte Erweiterung darstellt oder nur mehr Komfort bedeutet und dieselben Definitionen nicht auch mit einer BNF möglich sein. Wir fragen und also, ob Sprachen/Eigenschaften in der EBNF beschrieben werden können, welche in der BNF nicht möglich waren?

Dies ist aber nicht der Fall, denn jede EBNF kann in eine BNF übersetzt werden, welche die gleiche Sprache beschreibt. Gehe hierzu wie folgt vor:
Falls es eine Regel gibt mit

$$N \ ::= \ \alpha \ [\ \beta \ ] \ \gamma$$

Dann ersetze diese durch

$$N \ ::= \ \alpha \ \gamma \mid \alpha \ \beta \ \gamma$$

Falls es eine Regel gibt mit

$$N \ ::= \ \alpha \ \{ \ \beta \ \} \ \gamma$$

Dann ersetze diese durch

$$\begin{aligned} N \ &::= \ \alpha \ M \ \gamma \\ M \ &::= \ \beta \ M \ \mid \ \varepsilon, \end{aligned}$$

wobei M ein neues Nichtterminalsymbol der EBNF ist, also noch nicht in der aktuellen EBNF verwendet worden sein darf.

Die BNF wird auch von Compilern zur Analyse der Programmiersprache verwendet, Aufgabe ist es hierbei zu einem gegebenen Wort (Programm) einen passenden Ableitungsbaum zu konstruieren. Der Compiler kann dann alle möglichen Ableitungen ausprobieren:

Bsp: Finde Ableitung für $(3 + 4)$

$$\begin{aligned} Exp \ &\Rightarrow Var \ \lightning \\ &\Rightarrow Num \Rightarrow Digit \Rightarrow Digit \ Num \ \ \Rightarrow 0 \ \lightning \\ &\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow 1 \ \lightning \\ &\qquad\qquad\qquad\qquad\qquad\quad \ldots \\ &\Rightarrow (\ Exp \ Op \ Exp \ ) \Rightarrow (\ Val \ Op \ Exp \ ) \Rightarrow' -' \ Num \Rightarrow (\ Val \ Op \ Exp \ ) \Rightarrow \ldots \end{aligned}$$

Diese Suchverfahren, bei denen der Reihe nach alle möglichen Kodierungen durchgetestet werden bezeichnet man als *Backtracking*. Wir werden uns diese Programmiertechnik später noch genauer eingehen.

2.4.1 Präzedenzen in der BNF

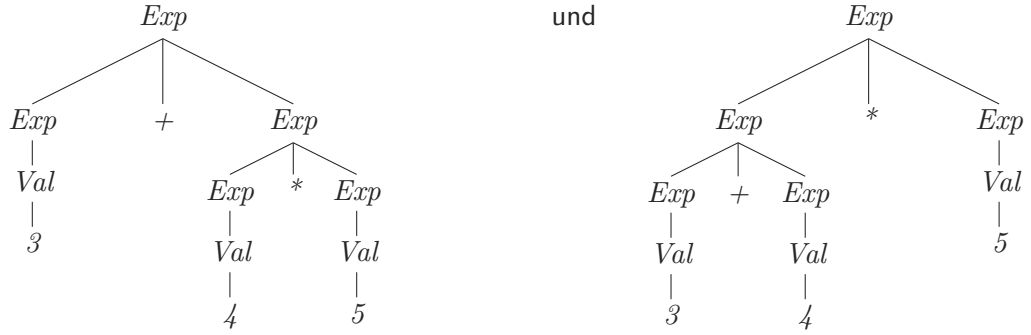
Wir wollen uns noch einmal mit der Klammerung für arithmetische Ausdrücke beschäftigen. Bisher haben wir ja in der EBNF nur vollständig geklammerte Ausdrücke betrachtet. Python erlaubt aber auch Ausdrücke, bei denen auf Grund von Präzedenzen Klammern fehlen, z.B.:

$$3 + 4 * 5 \text{ statt } (3 + (4 * 5)) \quad \textbf{und} \quad 3 * 4 - 2 \text{ statt } ((3 * 4) - 2)$$

Es ist tatsächlich möglich, auch für Ausdrücke eine EBNF anzugeben, so dass die Präzedenzen berücksichtigt werden. Zur Vereinfachung, betrachten wir hier nur die beiden Operatoren $+$ und *. Als ersten Ansatz könnten wir folgende BNF wählen:

$$
\begin{aligned}
Exp \;::=\; & Var \\
& \mid Val \\
& \mid Exp\ '+'\ Exp \\
& \mid Exp\ '*'\ Exp \\
& \mid '('\ Exp\ ')'
\end{aligned}
$$

Dann könnte der Ausdruck $3+4*5$ aber auf zwei unterschiedliche Wege abgeleitet werden:



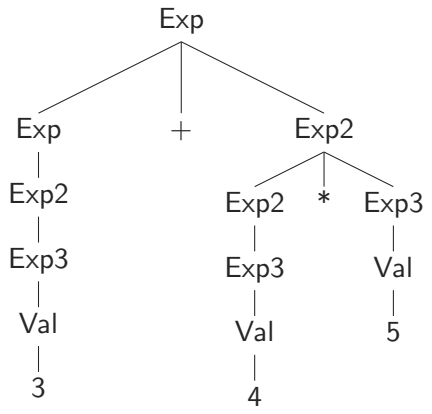
wobei der rechte Baum aber eben nicht der Strukturierung der Präzedenzen entspricht: ((3+4)*5).

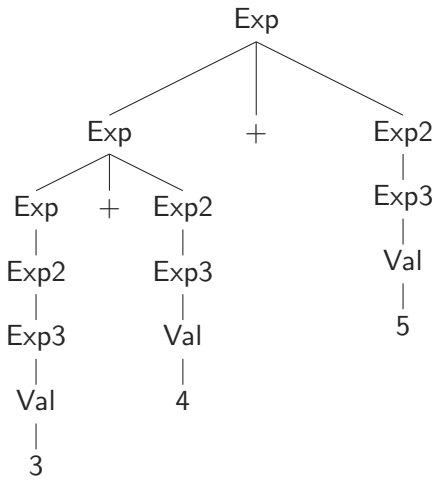
Zur Realisierung von Präzedenzen können wir unterschiedliche Ebenen in der EBNF vorsehen. Hierbei sind innerhalb von Multiplikationen dann eben Additionen nur noch erlaubt, wenn diese geklammert werden. Außerdem können wir gleichzeitig noch eine Linksklammerung bei gleichen Operatoren vorsehen, d.h. 3+5+6 wird interpretiert als (3+5)+6 und eben nicht als 3+(5+6). Man sagt der Operator bindet linksassoziativ.

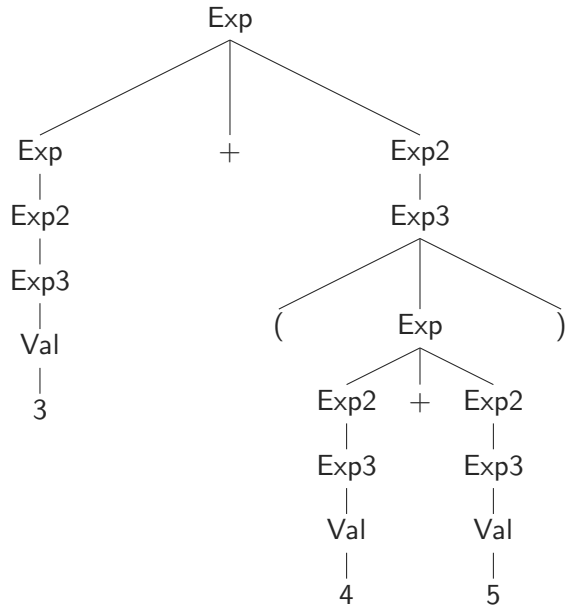
Die EBNF sieht dann wie folgt aus:

$$
\begin{aligned}
Exp \;::=\; & Exp\ '+'\ Exp2 \\
& \mid Exp2 \\
Exp2 \;::=\; & Exp2\ '*'\ Exp3 \\
& \mid Exp3 \\
Exp3 \;::=\; & '('\ Exp\ ')' \\
& \mid Var \\
& \mid Val
\end{aligned}
$$

Die zusätzlich hinzugenommen Nichtterminale *Exp2* und *Exp3* dienen dazu, zwischen beliebigen Ausdrücken (Exp), Ausdrücken ohne direkte Addition (*Exp2*) und Ausdrücken ohne Multiplikation und Addition zu unterscheiden. Außerdem ist bei der Regel für die Addition, eine weitere direkte Addition nur im linken Argument erlaubt, wodurch wir die Linksassoziativität des +-Operators realisieren. Als Beispiele betrachten wir folgende Ableitungen:

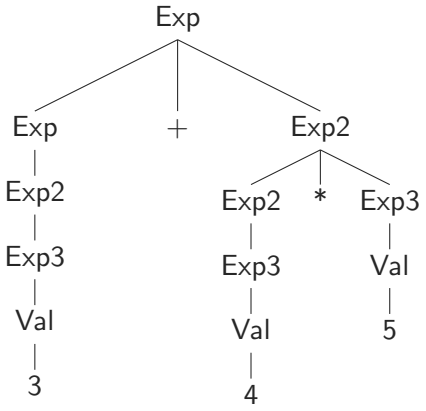




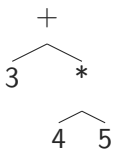


Für die explizite Rechtklammerung der Addition sind also Klammern notwendig: 3+(4+5), während die Linksklammerung auch ohne Klammern abgeleitet werden kann.

Zusammen mit Postfixnotation und Stackmaschine ergibt sich nun ein vollständiges Bild der Auswertung von Ausdrücken in Python. Eine gegebene Zeichenkette (z.B. 3+4*5) wird zunächst mit Hilfe einer EBNF mit Präzedenzen anlysiert. Wir erhalten den konkreten Ableitungsbaum:



Dieser wird dann in einen Termbaum (auch abstrakten Syntaxbaum genannt) umgebaut:



Dieser kann dann mittels Postfixnotation (3 4 5 * +) und Stackmaschine ausgewertet werden:

$$\begin{aligned}
\varepsilon \mid 3\ 4\ 5\ *\ + &\Rightarrow\ 3 \mid 4\ 5\ *\ + \\
&\Rightarrow\ 3\ 4 \mid 5\ *\ + \\
&\Rightarrow\ 3\ 4\ 5 \mid *\ + \\
&\Rightarrow\ 3\ 20 \mid +\ \\
&\Rightarrow\ \mathbf{23} \mid
\end{aligned}$$

2.5 Ausdrucksstärke unterschiedlicher Statements

Bisher haben wir folgende Anweisungen kennengelernt:

$$\begin{aligned}
Stm\ ::=\ & Stm\ (\ ';'\ \mid '\backslash n'\ )\ Stm \\
\mid\ & Var\ '='\ \ Exp \\
\mid\ & 'while'\ Exp\ ':'\ Stm\ '<:' \\
\mid\ & 'for'\ Var\ \ 'in'\ 'range'\ '('\ Exp\ ','\ \ Exp\ ')'\ ':'\ Stm\ '<:' \\
\mid\ & 'if'\ Exp\ ':'\ Stm\ '<:'\ [\ 'else'\ ':'\ Stm\ '<:'\ ] \\
\mid\ & 'print'\ '('\ Exp\ ')'
\end{aligned}$$

Im Folgenden wollen wir uns mit der Frage beschäftigen, ob diese wirklich alle notwendig sind oder durch andere simuliert werden können. Hierbei interessieren uns insbesondere die Kontrollstruktur $while$, die beiden if-Varianten, sowie die *for*-Schleife.

Auf die $while$-Schleife als einzige Schleife (Wiederholungsmöglichkeit), mit der wir eine Endlosschleife programmieren können, können wir sicherlich nicht verzichten. Da die for-Schleife immer terminiert werden wir nicht alle Programme mit ihr realisieren können. Können aber die if_then_else-Anweisungen simuliert werden?

if_then ohne $else$ kann durch ein if_then_else mit leerer $else$-Anweisung ersetzt werden. Wie konstruiert man eine leere Anweisung?

Falls eine Variable v bereits verwendet wird, können wir eine Zuweisung $v = v$ als „leere'— Anweisung verwenden.

Falls noch gar keine Variable verwendet wird, können wir eine Variable v, welche im Programm nicht verwendet wird initialisieren $v = 0$.

Ist es umgekehrt auch möglich, das if_then_else durch das if_then darzustellen?

Wir können einmal die Bedingung und dann die negierte Bedingung überprüfen:

```
if b:
  s1
else:
  s2

    ↓

if b:
  s1
if not b:
  s2
```

Dies ist so aber leider in einigen Fällen falsch, da sich eine Variable verändern kann, so dass b einen anderen Wert hat:

```
x = 5
if x>4:
  x = 2          Endbelegung: x = 2
else:
  x = 7
    ↓
x = 5
if x>4:
  x = 2          Endbelegung: x = 7
if not (x>4):
  x = 7
```

Wichtig ist es, beobachten zu können, ob $s1$ ausgeführt wurde und sonst $s2$ auszuführen: Eine mögliche Lösung ist die Verwendung eines booleschen Flags:

```
executed = False  # neue Variable
if b:
  s1
  executed=True
if not executed
  s2
```

Alternativ können wir auch das Ergebnis der booleschen Berechnung in einer Variablen zwischenspeichern und dann zweimal im $if-then$ abfragen. Zum Speichern des Wertes der Bedingung verwenden wir eine neu, im Programm noch nicht vorkommende Variable *bool*:

```
bool = b  # neue Variable
if bool:
  s1
if !bool:
  s2
```

Nun stellt sich die Frage, ob wir die if_then-Anweisung auch durch eine $while$-Schleife simulieren können. Mit der vorherigen Übersetzung können wir dann auch das if_then_else mit $while$ simulieren. ($\rightsquigarrow$ Übung)

```
if b:
  s
```

```
    ↓
while b:
  s
```

Probleme: s wird in der Regel mehrfach ausgeführt.

Lösung: Wir kombinieren, diese Ideen mit der Verwendung eines booleschen Flags um mehrfache Ausführung zu verhindern:

```
bool = b  # die Variable bool darf noch nicht verwendet sein
while bool:
  s
  bool = False
```

Als nächsten betrachten wir die Simulation von *if-then* mit Hilfe einer *for*-Schleife:

```
if b:
  s

    ↓

for i in range(0,1 if b else 0):
  s
```

Hierbei haben wir die *if-then* Anweisung zwar ersetzen können. Die Verzweigung konnte allerdings nur auf Ausdrucksebene verschoben werden, da unter Verwendung der *for*-Schleife sonst keine Möglichkeit existiert, in Abhängigkeit einer booleschen Bedingung zu verzweigen.

Nun wollen wir noch die Simulation der *for*- mit Hilfe der *while*-Schleife betrachten. Hierbei müssen aber einige wichtige Punkte berücksichtigt werden: Veränderungen der Zählvariablen bzw. der Bereichsgrenzen haben keinerlei Auswirkungen auf das Zählverhalten der Schleife. Um dies zu realisieren, Verwenden wir eine separate Zählvariable, deren Wert unabhängig von Veränderungen der eigentlichen Zählvariable hochgezählt werden kann. Außerdem wird der Endwert der Schleife einmalig vor Schleifenbeginn berechnet und zwischengespeichert:

```
for z in range(e1,e2):
  s

    ↓

end_value = e2      # neue Variable
zaehler   = e1      # neue Variable

while zaehler < end_value:
  z = zaehler
  s
  zaehler = zaehler+1
```

Abschließend wollen wir noch überlegen, ob wir die *while*-Schleife mit anderen Kontrollstrukturen simulieren können. Da die *while*-Schleife als einizige Kontrollstruktur nicht terminieren kann, ist dies nicht möglich. Wir können sie aber mit Hilfe der bedingten Anweisung abwickeln:

```
s1
while b:
  s2
```

ist äquivalent zu

```
s1
if b:
  s2
  while b:
    s2
```

Wiederholt man dieses Abwickeln, so stellt man fest, dass die *while*-Schleife einer "unendlichen Verschachtelung bedingter Anweisungen" entspricht:

```
s1
if b:
  s2
  if b:
    s2
    if b:
      s2
      if b:
        s2
        ...
```

2.6 Zeichenketten

Bei der Programmierung muss auch häufig mit Texten (Zeichenfolgen) gearbeitet werden. Hierzu stellen die meisten Programmiersprachen Zeichenketten (Strings) als Werte zur Verfügung.

In Python heißt die Klasse $String$ und Werte dieser Klasse können wie folgt definiert werden:

'Hallo'	'Dies ist ein "Text".'
'noch ein Text'	"ein ganz 'komischer' Text"

Strings können ebenfalls mittels $print()$ ausgegeben werden:

print('Hallo')	produziert die Ausgabe	Hallo
print('a"b"c')	produziert die Ausgabe	a"b"c

Operationen auf Strings:

Strings können mit der Operation + verbunden werden:

'Hallo' + '␣' + 'Leute' $\rightsquigarrow$ 'Hallo␣Leute'

Es gibt noch eine Reihe weiterer Funktionen für Strings. Die erste, die wir kennen lernen, dient zur Bestimmung der Länge eines Strings. Sie heißt len:

```
len('Hallo') ⇝ 5
len('Info ist toll') ⇝ 13
```

Man kann Strings auch mit einer Zahl multiplizieren, was die vorkommenden Zeichen mit der entsprechenden Häufigkeit zusammen fügt:

```
'Hi' * 3 ⇝ 'HiHiHi'
3 * 'Hallo' ⇝ 'HalloHalloHallo'
```

Außerdem können Strings auch mittels input() vom Benutzer eingelesen werden:

Bsp.:

```
str = input()
print(len(str))
print(str)
```

Strings können neben normalen Zeichen auch Steuerzeichen enthalten. Ein wichtiges ist hierbei der Zeilenumbruch \n. Kommt er in einem String vor, wird bei der Ausgabe des Strings ein Zeilenumbruch vorgenommen.

Weitere spezielle Steuerzeichen in Strings:

\t'— Tabulator
\r'— Wagenrücklauf (ggf. bei Zeilenumbruch unter Windows)

Außerdem ist es möglich, auf Teilstrings zuzugreifen: $str[a:e]$, wobei a die Anfangsposition und e die Positon hinter dem letzten gewählten Zeichen ist (ähnlich zur 'for'-Schleife). Beachte hierbei aber, dass die Positionszählung bei Null beginnt, man das erste Zeichen also mit abcdef'—[0 : 1] und nicht mit abcdef'—[1 : 2] bekommt. Wir beginnen das Zählen also bei 0:

```
'abcdef' [3:5] ⇝ 'de'
'abcdef' [2:3] ⇝ 'c'
'abcdef' [3:13] ⇝ 'def'
```

Jetzt wollen wir mit Strings programmieren.

Aufgabe: Zähle, wie oft ein bestimmter Buchstabe in einem Text vorkommt.

```
text = input('Text: ')

letter = input('Buchstabe: ')[0:1]
     #wir speichern nur den ersten Buchstaben
n = 0
for i in range(0,len(text)):
  if text[i] == letter:
    n = n + 1

print("Der Buchstabe '" + letter +
      "' kommt " + str(n) + ' mal vor.')
```

Die einstellige Funktion str wandelt eine Zahl in einen String um (später mehr).

```
str(3) ⇝ '3'
```

Wir haben ähnliche Funktionen bereits zuvor verwendet, um Strings in Zahlen umzuwandeln.

```
int('123') ⇝ 123\\
int('-3a') ⇝ Syntax error\\
```

```
int('Hallo') ⇝ Syntax error\\
float('-2.34') ⇝ -2.34\\
int('-2.34') ⇝ -2\\
```

Dies haben wir z.B. im Fakultätsprogramm genutzt:

```
n_str = input()
n = int(n_str)
...
```

oder direkt:

```
n = int(input())
```

Ausführen unseres Programms zum Buchstanbenzählen:

```
> Python countletter.py
Text: Informatik ist ein schoenes Fach
Buchstabe: i
Der Buchstabe 'i' kommt 3 mal vor.
>
```

Beachte, dass das großgeschriebene 'I' bei Informatik nicht mitgezählt wurde. Groß- und Kleinbuchstaben werden also unterschieden.

Man sieht, dass die zunächst etwas merkwürdige Angabe der Grenzen der for-Schleife sehr gut zu dem iterieren durch einen String passt. Die Positon der Länge des Strings existiert nicht, da die Positionszählung mit Null beginnt. Wir können die Länge des Strings also optimal als Endposition angeben.

Da man häufig über Strings (und später auch über Listen und Listen) mit Hilfe von for-Schleifen iteriert, bietet Python noch zwei praktische Abkürzungen an, mit denen man dieses Programm vereinfachen kann. Zunächst können wir bei der 'range'-Funktion auch nur den Endwert angeben. Der Aufruf range(n) entspricht dann dem Aufruf range(0,n). Außerdem möchte man häufig auf nur ein Zeichen in einem String zugreifen, weshalb wir auch str[n] anstatt str[n:n+1] schreiben können. Beachte aber, dass str[n] einen Teilstring der Länge eins selektiert und nicht, wie in manchen anderen Programmiersprachen einen Character, was manchmal der Datentyp eines Zeichens ist. Mit diesen Vereinfachungen sieht unser Programm wie folgt aus:

```
text = input('Text: ')

letter = input('Buchstabe: ')[0]
     #wir speichern nur den ersten Buchstaben
n = 0
for i in range(len(text)):
  if text[i] == letter:
    n = n + 1

print("Der Buchstabe '" + letter +
      "' kommt " + str(n) + ' mal vor.')
```

Als nächstes Problem wollen wir untersuchen, ob ein String in einem anderen String vorkommt.

Bsp.:

```
text='Die Giraffe trinkt Kaffee.'
sub='kaffe'
i=0
while i < len(text) and text[i:i + len(sub)] != sub:
  i = i+1

print('\'' + sub + '\' kommt in \'' + text +
      '\' ' + ('nicht' if i==len(text) else '') +' vor.')
```

Abschließen wollen wir dieses Kapitel mit einem Programm, welches einen gegebenen String umdreht.

```
text = input()

rev = ''    # in rev bauen wir den umgedrehten String auf

for i in range(len(text)):
   rev = text[i]+rev

print(rev)
```

2.7 Prozedurale Abstraktion

Bereits bei der Einführung des Algorithmenbegriffs haben wir folgende Aspekte diskutiert:

- Detailliertheit der Beschreibung (Verständlichkeit) (nimm Ei aus Verpackung, schlage es gegen Schüsselkante vs. trenne Ei, schlage Eiweiß)
- Wiederverwendbarkeit (Sauce Hollandaise oder Rührteig anderswo definiert)

Auch bei unseren Programmen wäre eine solche Strukturierung wünschenswert. Hierzu bieten Programmiersprachen in der Regel **Prozeduren** (ohne Rückgabewert) oder **Funktionen** (mit Rückgabewert).

2.7.1 Funktionen

Als Beispiel betrachten wir die Definition einer Funktion zur Berechnung der Fakultät.

```
def fac(n) :
  res = 1
  for i in range(1,n+1):
    res = res * i

  return res

print(fac(5)) # Ausgabe: 120
print(fac(6)) # Ausgabe: 720
```

Hierbei ist def ein Schlüsselwort, welches den Beginn einer Funktionsdefinition anzeigt. Wie immer wird der Rumpf der Funktionsdefinition durch Einrückung festgelegt. Nach dem Schlüsselwort def steht der Name der definierten Funktion und danach kommen die *f*ormalen Parameter (Variablen, falls mehrere durch Kommas getrennt), welche von den *k*onkreten Parametern] beim Funktionsaufruf abstrahieren. Nach den Parametern kann das Verhalten der Funktion in Form normaler Anweisungen definiert werden. Diesen Teil bezeichnet man als *Funktionsrumpf*.

Der *R*ückgabe-Wert der Funktionsdefinition wird mit Hilfe des Schlüsselwortes return bestimmt. Die Ausführung der return-Anweisung beendet die Funktion. Um den Code übersichtlicher zu gesatlten sollte man aber darauf achten, dass return immer am Ende einer Funktionsdefinition steht und bei Verzweigungen in allen möglichen Zweigen ein Rückgabewert mittels return festgelegt wird.

Die Fakultätsfunktion kann nach der Definition beliebig häufig im weiteren Programm verwendet werden und mit unterschiedlichen Werten aufgerufen werden.

Ein Funktionsaufruf liefert genau wie eine vordefinierte Funktionen einen Rückgabewert. Um den Ergebniswert in die weiteren Berechnungen einbauen zu können, werden Funktionen im Rahmen von Ausdrücken aufgerufen, z.B.

```
n = fac(4)

if fac(5)>100 :
  print('gross')
else :
  print('klein')

print('Fakultaet von ' + str(n) + ' lautet ' + str(fac(n)))
```

Beim Aufruf einer definierten Funktion, werden zunächst die Argumente ausgewertet. Danach werden die formalen Parameter (Variablen der Funktionsdefinition) an die aktuellen Parameter (Argumente) gebunden. Mit dieser Bindung wird der Funktionsrumpf ausgewertet und, nach Beendigung dieser Auswertung, der Aufruf durch den Rückgabewert (Wert hinter return) ersetzt.

Zum Beispiel würde der Ausdruck $\mathrm{fac}(\mathrm{fac}(3))+10$ zunächst $\mathrm{fac}(3)$ berechnet, wozu mit der Belegung $\mathrm{n}{=}3$ die Fakultätsfunktion ein erstes Mal ausgeführt wird. Die Variable res wird für die Belegung $\mathrm{n}{=}3$ beim return an 6 gebunden sein und der Aufruf zu $\mathrm{fac}(6)+10$ ausgewertet werden. Danach erfolgt der nächste Aufruf mit der Belegung $\mathrm{n}{=}6$. Disesmal erhalten wir die finale Belegung $\mathrm{res}{=}720$ und die Berechnung wird mit $720{+}10$ fortgesetzt, was abschließend noch zu 730 ausgerechnet wird.

Bei der Definition eigener Funktionen kann man in Python auch Variablen von außerhalb verwenden, man kann sie aber nicht ändern. Zum besseren Programmverständnis sollte man dies aber auch nicht machen, sondern alle relevanten Werte als Parameter an die Funktion übergeben. Tatsächlih gibt es innerhalb von Funktionsdefinitionen einen lokalen Namensraum, in dem lokal definierte Variablen gleichnamige Variablen von außen verdecken. Dies erkennt man an dem folgenden Beispiel:

```
x = 42

def bla() :
  x = 73
```

```
  return x

print(bla())   # Ausgabe: 73
print(x)       # Ausgabe: 42
```

Innerhalb der Funktion bla wird also eine lokale Variable x definiert, welche unabhängig von der Variablen x des Hauptprogramms existiert und verändert wird. Die Veränderung der lokalen Varibalen hat im Gegenzug auch keine Auswirkung auf die Belegung von x im Hauptprogramm.

Zur Auswertung von Funktionen werden wir in der Vorlesung eine Erweiterung der Programmpunkttabellen kennen lernen, welche für aufgerufene Funktion eine separate Funktionstabelle anlegt.

2.7.2 Prozeduren

Im Gegensatz zu Funktionen haben Prozeduren keinen Rückgabewert, das bedeutet, dass sie auch keine Ergebnisse liefern (keine return!). Auch in Prozeduren können globale Variablen in aufgerufenem Programmcode nicht verändert werden. Aber Prozeduren können z.B. Ausgaben erzeugen.

Bsp.: Ausgabe eines Quadrats gegebener Größe

```
def put_quadrat(n) :
  line = ''              #
  for i in range(0,n) : # line = 'X' * n
    line = line + 'X'    #
                         #
  for i in range(0,n) :
    print(line)

put_quadrat(5)

n = int(input())
put_quadrat(n)
```

Da Prozeduren keinen Ergebniswert haben (sie machen höchstens Ausgaben auf dem Bildschirm, welche aber nicht als Ergebnis der Unterberechnung weiterverwendet werden können), können Prozeduren nur als Anweisungen verwendet werden:

```
n = int(input())
put_quadrat(n)

for i in range(0,5) :
  put_quadrat(i)
```

Versucht man eine Prozedur als Funktion zu verwenden, sieht man, dass Python für Prozeduren einen speziellen Rückgabetyp verwedendet, welcher als None ausgegeben wird.

Beachte: Prozeduren und Funktionen können auch andere Prozeduren und Funktionen aufrufen.

Bsp.:

```
def n_spaces_in_between(str,n) :
  return str + ' ' * n + str

def triangle(n) :
  print('*')

  for i in range(0,n-2) :
    print(n_spaces_in_between('*',i))

  print('*' * n)

triangle(3)
print()          #erzeuge eine Leerzeile/Zeilenumbruch
triangle(7)
```

Wichtig ist nur, dass der erste Aufruf einer Funktion unterhalb aller sich wechselseitig aufrufenden Funktionen erfolgt. Die Funktion triangle kann also nicht direkt hinter der Funktionsdefinition von n_spaces_in_between erfolgen. Auch wäre es nicht erlaubt, die Funktion n_spaces_in_between hinter dem Hauptprogramm zu definieren. Die Reighenfolge der beiden Funktionendefinitionen kann aber vertauscht werden.

Als nächstes wollen wir noch untersuchen, ob Funktionen und Prozeduren auch innerhalb von anderen Prozeduren definiert werden können. Dies ist z.B. sinnvoll, wenn man eine Hilfsprozedur definieren möchte, die man aber eigentlich nur innerhalb der Prozedure, aber nicht von außen verwenden soll.

In unserem letzten Beispiel könnte dies die Prozedur n_spaces_in_between sein, welche man vielleicht nur innerhalb von triangle verwenden sollte:

```
def triangle(n) :

  def n_spaces_in_between(str,n) :

    return str + ' ' * n + str

   # Ende der Def. von n_spaces_in_between

  print('*')

  for i in range(0,n-2) :
    print(n_spaces_in_between('*',i))

  print('*' * n)

 # Ende der Definition von triangle

triangle(3)
print()          #erzeuge eine Leerzeile/Zeilenumbruch
triangle(7)
```

Beachte, dass die Einrückung klar zeigt, wo der Rumpf der Unterprozedur n_spaces_in_between endet und wo der Rumpf von triangle weitergeht. Die Verwendung lokaler Funktions- und

Prozedurdefinitionen wird aber erst bei größeren Programmen relevant, so dass Programmieranfänger hierauf zunächst verzichten können.

Wenn Funktionen andere Funktionen aufrufen können, können sie sich dann auch selbst aufrufen?

Antwort: Ja, **Rekursion**

2.8 Rekursion

Bsp.: Fakultätsfunktion

Die Fakultätsfunktion ist ja mathematisch definiert als:

$$n! = \begin{cases} 1, \text{ falls } n = 0 \\ n \cdot (n-1)! \text{ , sonst} \end{cases}$$

Dies lässt sich direkt als rekursive Funktion umsetzen:

```
def fac(n) :
  if n==0 :
    return 1
  else :
    return n * fac(n-1)

print(fac(5))    # 120
print(fac(4))    # 24
```

Das Verhalten eines rekursiven Programms kann am Beispiel des Aufrufs print(fac(3)) wie folgt verdeutlicht werden:

print(fac(3)) hier muss zunächst das Ergebnis von fac(3) bestimmt werden:
fac(3)
↪ Code von fac mit Belegung n=3 auswerten
die Bedingung ist nicht erfüllt, deshalb weiter im else-Zweig
return 3*fac(2) hier muss zunächst das Ergebnis von fac(2) bestimmt werden:
fac(2)
↪ Code von fac mit Belegung n=2 auswerten
die Bedingung ist nicht erfüllt, deshalb weiter im else-Zweig
return 2*fac(1) hier muss zunächst das Ergebnis von fac(1) bestimmt werden:
fac(1)
↪ Code von fac mit Belegung n=1 auswerten
die Bedingung ist nicht erfüllt, deshalb weiter im else-Zweig
return 1*fac(0) hier muss zunächst das Ergebnis von fac(0) bestimmt werden:
fac(0)
↪ Code von fac mit Belegung n=0 auswerten
die Bedingung ist erfüllt, deshalb weiter im then-Zweig
return 1 die Rekursion terminiert, Ergebniswert bekannt
↩ der Aufruf von fac(0) wird durch 1 ersetzt
return 1*1 ausrechnen: 1*1 → 1
return 1 die Rekursion terminiert, Ergebniswert bekannt
↩ der Aufruf von fac(1) wird durch 1 ersetzt

    return 2*1 ausrechnen: 2*1 → 2
    return 2 die Rekursion terminiert, Ergebniswert bekannt
  ↩ der Aufruf von fac(2) wird durch 2 ersetzt
  return 3*2 ausrechnen: 3*2 → 6
  return 6 die Rekursion terminiert, Ergebniswert bekannt
↩ der Aufruf von fac(3) wird durch 6 ersetzt
print(6) ausgeben ⇝ 6

In der Vorlesung werden wir dies noch zusätzlich mit Hilfe von Programmzeilen Tabellen verdeutlichen.

Beachte, dass es nicht immer der Fall sein muss, dass eine Funktion nach dem rekursiven Aufruf selber auch direkt terminieren muss. Hier können noch weitere Funktionen (auch rekursiv!) aufgerufen oder beliebiger anderer Code ausgeführt werden.

Zur Implementierung der Rekursion wird ein Stack (Keller), wie wir ihn in der Stack-Maschine kennen gelernt haben, verwendet. Bei der Ausführung von Prozeduren und Funktionen nennt man diesen Laufzeitstack (oder Laufzeitkeller). Auf ihm wird zu einem Funktionsaufruf die umgebenden Berechnungen gespeichert. Nach der Beendigung einer (nicht nur rekursiven) Funktion wird die Berechnung in der auf dem Laufzeitkeller gespeicherten Umgebung fortgesetzt. In der Ausführung oben haben wir dies durch die Einrückung dargestellt, d.h. weniger stark eingerückte Ausführungeteile liegen noch auf dem Laufzeitkeller und werden noch fertig ausgeführt, wenn die Unterberechnung (z.B. fac(2)) beendet wurde.

Bemerkung: Rekursion ist genauso ausdrucksstark wie die Verwendung von Schleifen, welche auch als **Iteration** bezeichnet wird. Es gibt sogar Programmiersprachen, welche nur Rekursion und keine Schleifen anbieten (z.B. Funktionale Programmiersprachen).

Um zu verstehen, wie man zu iterativen Programmen äquivalente rekursive Programme entwickeln kann, betrachten wir noch einmal das Umdrehen eines Strings:

```
def reverse(str) :

  if len(str)<=1 :
    return str
  else :
    return reverse(str[1:len(str)]) + str[0]

print(reverse('abcde'))
```

Alternativ können wir auch den String unverändet lassen und im rekursiven Aufruf nur den Index, welchen wir im aktuellen Schritt übernehmen wollen, runterzählen. Da die Hilfefunktion, welche den Index als zusätzlichen Parameter nimmt nur innerhalb unseres Algorithmus sinnvoll verwendet werden kann, verwenden wir hier eine lokale Funtionsdefinition rev:

```
def reverse(str) :

  def rev(str,i) :
    if i<0 :
      return ''
    else :
      return str[i]+rev(str,i-1)
```

```
  return rev(str,len(str)-1)

print(reverse('abcde'))
```

Da neben dem Index auch der String, welchen wir umdrehen wollen, im Rumpf der rekursiven Funktion bekannt sein muss, erhält die Funktion rev zwei Parameter. Die eigentliche Funktion reverse mit einem Argument startet dann die zweistellige Hilfsfunktion rev mit einem passenden Wert für den ersten Index.

Innerhalb von rev wird die Variable str nicht verändert, sondern nur gelesen. Deshalb kann man sie als Parameter auch weglassen und einfach die Variable str aus dem äußeren Scope verwenden:

```
def reverse(str):

  def rev(i):
    if i<0:
      return ''
    else:
      return str[i]+rev(i-1)

  return rev(len(str)-1)

print(reverse('abcde'))
```

3 Modellierung von Daten

3.1 Listen

Bisher haben wir nur vordefinierte Datentypen kennengelernt. Oft ist es aber auch wichtig mehrere Werte zu neuen zusammengesetzten Werten zu machen. Ein hierfür gängiger Datentyp in imperativen Sprachen ist das Liste, welches mehrere Werte in einem Speicherbereich hintereinander repräsentiert und die einzelnen Werte über einen Index adressierbar macht.

Python bietet allerdings nicht direkt die Möglichkeit mit Listen zu programmieren. Hierfür ist die Verwendung einer speziellen Bibliothek namens NumPy notwendig, welche wir aber erst später betrachten werden. Die Standard Datenstruktur zum zusammenfassen mehrer Werte wie in einem Liste, sind in Python Listen.

Listen sind sehr viel dynamischer als Listen und können im Gegensatz zu gängigen Liste Implementierungen auch erweitert werden.

Listen können beliebig viele Python-Werte aufnehmen. Die Werte sind dann für eine Liste "l", wie bei Strings, über die Indizes 0 bis len(l) zugreifbar. Wir betrachten ein einfaches Beispiel:

```
list = [42,73,7]

print(list)          # Ausgabe [42,73,7]
print(len(list))     # Ausgabe 3

list = list + list
print(list)          # Ausgabe [42,73,7,42,73,7]

print(list[1:4])     # Ausgabe [73, 7, 42]
```

Wir sehen, dass viele Funktionen, die wir bereits von Strings kennen genau so für Listen funktionieren. Wir können Listen mittels + verbinden und Teillisten mittels list [start:end] ausschneiden.

Ein kleiner Unterschied ergibt sich aber, wenn man bei Listen nur auf einen Index und nicht auf einen Teilbereich zugreift. Während wir bei Strings einen Teilstring der Länge eins erhalten haben, wird bei Listen das Element an dem Index zurückgegeben. Keine einelementige List mit dem Element.

In obigem Beispiel ergäbe sich also:

```
list = [1,2,3,4,5]
print(list[1])    #  Ausgabe 73
```

Die Programmierung mit Listen kann sehr ähnlich, wie die Programmierung mit Strings erfolgen. Als Beispiel betrachten wir noch einmal die Funktion reverse, mit welcher wir nun

eine Liste umdrehen wollen. Diesesmal programmieren wir sie aber iterativ:

```
list = [1,2,3,4,5,6,7]

def reverse(l):

  rev = []
  for i in range(len(list)):
    rev = [list[i]] + rev

  return rev

print(reverse(list))    # Ausgabe [7,6,5,4,3,2,1]
```

Die Definition für Listen muss nur an eienr Stelle anders vorgenommen werden, als für Strings. Bei der Konstruktion der neuen Liste im Schleifenrumpf, erhalten wir mit `list[i]`— ja das Element an der Postion i. Um dies wieder mittels + mit der bereits berechneten Liste in der Variablen `rev` verbinden zu können, müssen wir das Element in eine einelementige Liste einfügen (`[list[i]]`).

3.2 Objektorientierte Programmierung

Bei unseren bisherigen Programmen haben wir zwischen Werten und Funktionen (Operationen die wir auf Werte anwenden können) unterschieden. Enige Funktionen werden in Python aber in einer andren Form, als sogenannte Methoden, angeboten. In diesem Abschnitt wollen wir uns die Zusammenhänge etwas genauer anschauen.

Um die Fülle von Funktionen zu strukturieren wurde die *objektorientierte* Sicht von Daten und Funktionen auf diesen Daten erfunden.

Idee: Die Welt besteht aus **Objekten** (z.B. Bello, Waldi, Mietzi) die in **Klassen** z.B. Hund, Katze eingeteilt werden. Als weiteres Beispiel gibt es die Klasse der ganzen Zahlen, welche z.B. die Werte 42, 0 und -7 enthalten, sowie die Klasse der Strings, welche z.B. die Werte "Halloünd Informatikënthalten. Objekte werden dann zu einer Klasse zusammengefasst, wenn sie gleiche Eigenschaften bzw. Operationen besitzen, so können Hunde bellen, Zahlen addiert werden und Strings konkateniert werden bzw. haben eine Länge.

Operationen auf diesen Objekte bezeichnet man auch als *Methoden*.

Bsp.: Objekte der Klasse Hund könnten z.B. die Methoden hat_rasse, hat_groesse, hat_geschlecht zum Abfragen bestimmter Eigenschaften haben. Methoden, die das Verhalten von Hunden ändern könnten bellen oder spiel_mit sein.

Auch Zahlen haben Methoden +, -, * usw., genau wie Stings. In Python haben wir viele Methoden bis jetzt aber als Funktionen oder Operatoren notiert und Python ist hier weniger konsequent, als z.B. Ruby, wo alle Operationen und Funktionen auch Methoden einer bestimmten Klasse sind. Dennoch gibt es viele Funktionen in Python, welche nur als Methoden verfügbar sind. Bei der Notation einer Methode pbergibt man das Objekt, für welches man die Methode aufruft nicht als Argument sondern notiert zunächst das Objekt, gefolgt von einem Punkt und dann der Name der Methode, welche optional noch weitere Parameter erhalten kann.

Ein erstes Beispiel ist die nullstellige Methode upper der Klasse String, welche alle Zeichen eines Strings in Großbuchstaben umwandelt. Ihre Verwedenung sollte an dem folgenden Beispiel klar werden:

'abcd'.upper() $\rightsquigarrow$ 'ABCD'

Obwohl wir der Methode keinen Parameter übergeben, erhält sie das Objekt (hier den String 'abcd') als Parameter und kann auf diesem operieren. Entsprechend gibt es auch eine nullstellige Methode lower in der Klasse String:

'AbCd'.lower() $\rightsquigarrow$ 'abcd'

Eine Beispiel für eine Methode, welche zusätzlich noch Parameter verwendet ist die zweistellige Methode replace, mit welcher man in einem String alle Vorkommen eines Teilstrings durch einen anderen String ersetzen kann:

'abcdefbcghbcij'.replace('bc','xxx') $\rightsquigarrow$ 'axxxdefxxxghxxxij'

Natürlich muss das Objekt, für welches man die Methode aufruft nicht explizit angegeben sein. Wir können auch eine Variable oder das Ergebnis einer anderen Berechnung verwenden:

```
string = input()

print(string.replace('ize''ise').replace('yze','yse').lower())
```

Für die Eingabe 'Realize and analyze this surprise.' liefert das Programm die Ausgabe 'realise and analyse this surprise.'. Die Methoden werden also nacheinander auf das Ergebnis der vorherigen Methodenanwendung angewendet.

Betrachten wir noch einmal unser Beispiel aus der Welt der Hasutiere:

Die Größe des Objekts Bello könnte man hier durch Zugriff auf die Methode groesse(), welches alle Objkete der Klasse Hund zur Verfügung stellen wie folgt zugreifen: Bello.groesse().

Außerdem könnten wir Bello bellen lassen, indem wir die Methode Bello.bellen() aufrufen oder ihn mit Waldi spielen lassen: Bello.spiel_mit(Waldi).

In Python können wir dies im Moment noch nicht mit Hunden und Katzen durchführen, werden dies aber später noch kennen lernen.

Für Zahlen bietet Python keine speziellen Methoden an. Alle Funktionalitäten stehen als Funktionen oder Operatoren zur Verfügung. Dies ist in vielen anderen imperativen Programmiersprachen anders.

3.3 Objekte und ihre Identität

Nach dem wir auf sehr abstrakter Ebene Klassen, Objekte und Methoden kennengelernt haben, wollen wir einige weitere Methoden genauer untersuchen. Die Objektorientierte Programmierung (Objektorientierte Programmiersprachen) macht nämlich noch mehr aus, als Methoden in Klassen zu sammeln und mit der Punkt-Schreibweise zu notieren.

Objekte haben eine Identität und können verändert werden. Ihr Zustand ist durch ihre Attribute definiert.

Bsp.:
Bello könnte ein Objekt der Klasse Hund sein. Ein Attribut könnte z.B. sein Gewicht sein.

Dann könnte es eine Methode fressen geben, welche natürlich sein Gewicht verändert. Man beachte hierbei, dass sich nicht Bello verändert und wir nach dem Fressen auch keine Kopie von Bello erhalten. Das Objekt Bello wird verändert.

Ein anderes Beispiel ist ein Konto. Wenn man von einem Konto Geld abhebt oder Geld einzahlt, erhält man kein neues Konto. Vielmehr verändert sich der Zustand des Kontos, welcher durch ein Attribut, den Kontostand repräsentiert wird.

Entsprechend können auch einige der Python Objekte verändert werden. Ein wichtiges Beispiel hierbei sind Listen. Für Listen gibt es eine einstellige Methoden append, welche eine Liste um ein Element erweitert.

```
list = [1,2,3,4,5]
list.append(42)
print(list)        # Ausgabe: [1,2,3,4,5,42]
```

Man beachte, dass wir einfach nur die Methode append für die Liste in der Variablen list aufrufen. Wir weisen keine Ergebnis einer Variablen zu und die Variable list wird auch gar nicht verändert. Vielmehr wird das Objekt, an welches die Variable list gebunden ist verändert. Dies erkennt man, wenn man mehrere Variablen an dasselbe Objekt bindet ,wie das folgende Python-Programm verdeutlich:

```
x = [1,2,3,4]     # Definition der Liste
y = x             # Bindung der Variablen y an das Objekt,
                  # an welches x gebunden ist
x.append(42)      # Mutation der Liste, an die x gebunden ist
print(x)          # Ausgabe: [1,2,3,4,42]
print(y)          # Ausgabe: [1,2,3,4,42]
```

Die Variablen verweisen beide auf das gleiche Listen-Objekt, welches mittels append verändert wird. Methoden, die ein Objekt verändern beziehnet man auch als *mutierende Methoden* bzw. *Mutationen*.

Im Gegensatz hierzu

```
x = [1,2,3,4]     # Definition der Liste
y = [1,2,3,4]     # Bindung der Variablen y an eine Liste,
                  # die zufaellig wie die in x aufgebaut ist
x.append(42)      # Mutation der Liste, an die x gebunden ist
print(x)          # Ausgabe: [1,2,3,4,42]
print(y)          # Ausgabe: [1,2,3,4]
```

x und y zeigen auf zwei unterschiedliche Listen-Objekte, die zwar beide gleich aussehen, aber eben unterschiedliche Objekte sind (wie Zwillinge). Somit führt eine Mutation des Objekts, auf das die Varibale x verweist, zu keiner Mutation des anderen Objekts, auf das y verweist.

Entsprechend wollen wir zum Vergleich auch noch einmal das entsprechende Programm, welches die Listkonkatenation verwendet, untersuchen.

```
x = [1,2,3,4]     # Definition der Liste
y = x             # Bindung der Variablen y an das Objekt,
                  # an welches x gebunden ist
y = x + y         # Zuweisung, so dass die Variable y neu
                  # gebunden wird
```

```
print(x)          # Ausgabe: [1,2,3,4]
print(y)          # Ausgabe: [1,2,3,4,1,2,3,4]
```

Hier zeigen x und y zwar auf das gleiche Listen-Objekt. In der darauf folgenden Zuweisung y = x + y] wird aber kein Objekt verändert, sondern ein neues String-Objekt konstruiert und die Variable y neu, an dieses Objekt gebunden. x verweist weiter auf das alte, unveränderte Listen-Objekt. Die Funktion + (im Python-Kontext of auch als Methode bezeichnet) ist also nicht mutierend. Sie liefert ein neues Objekt und verändert keines ihrer Argumente.

Objekte besitzen also eine Identität und bestimmte Methoden (hier die 1-stellige Methode append) können das Objekt verändern. Solche Methoden werden auch als *mutierende Methoden* bezeichnet. Variablen beinhalten nicht das gesamte Objekt sondern nur einen Verweis auf ein Objekt. Für Listen gibt es noch eine weitere häufig genutze mutierende Methode zum Mutieren einer Liste an einer bestimmten Postion. welche eine spezielle Syntax besitzt:

```
x = [1,2,3,4]     # Definition der Liste
y = [1,2,3,4]     # Bindung der Variablen y an eine Liste,
                  # die zufaellig wie die in x aufgebaut ist
x[2] = 42         # Mutation der Liste, an die x gebunden ist
print(x)          # Ausgabe: [1,2,42,4]
print(y)          # Ausgabe: [1,2,3,4]
```

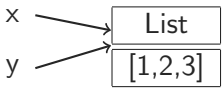
In der dritten Zeile mutieren wir das Listen-Objekt. An der Position 2 wird der vorhandene Wert durch den neuen Wert 42 ersetzt. Wie wir an den beiden Ausgaben sehen, wird das Listen-Objekt tatsächlich verändert und kein neues Objekt generiert. Die Syntax der Modifikation einer Liste an einer gegeben Position suggeriert zwar, dass es sich um eine Zuweisung handelt, die Variable x bleibt aber in Wirklichkeit unverändert (referenziert dasselbe Listen-Objekt). Einzig das Objekt wird mutiert. Es handelt sich also eher um eine mutierende Methode.

Die zuweisungsähnliche Schreibweise geht auf die Idee zurück, dass eine Liste eigentlich eine Liste von Variablen ist, welche nicht direkt benannt, sondern indirekt über den Index addressiert werden. In diesem Sinn kann man diese mutierende Methode tatsächlich als eine Art Zuweisung, aber eben nicht auf die Variable x, sondern auf die Variable x mit dem Offset 2 sehen. Diese "Varibale" x[2] wird dann auch tatsächlich an einen neuen Wert gebunden. Zum Verständniss ist es aber einfacher diese Operation als mutierende Methode zu sehen, welche eine benutzerfreundlich Schreibweise hat. In manchen anderen Sprachen findet man solch eine Methode auch tatsächlich, z.B. in der Form x.update(2,42).

Zur weiteren Verdeutlichung von Objektidentität und mutierenden Methoden verdeutlichen wir, wie der Speicher des Python-Systems nach der Ausführung der folgenden beiden Python-Anweisungen aussieht:

```
x = [1,2,3]       # Definition der Liste
y = x             # Bindung der Variablen y an eine Liste,
                  # die zufaellig wie die in x aufgebaut ist
```

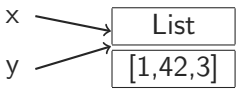
Im Speicher wurde ein Listen-Objekt angelegt. Dieses Objekt ist das Ergebnis des Ausdrucks [1,2,3] und auch das Ergebnis des Ausdrucks x. Entsprechend werden beide Variablen in den Zuweisungen an dieses Objekt gebunden, was wir als Pfeil darstellen:



Wird nun eine Mutation des Objektes in x vorgenommen

```
x[1] = 42          # Mutation der Liste, an die x gebunden ist
```

so wird zunächst der Wert von x nachgschlagen, was das Listen-Objekt ist. Auf diesem Objekt wird dann die Mutation an der Positon 1 durchgeführt und das Objekt verändert sich:



Hierbei wird die Belegung beider Variablen nicht verändert. Sie zeigen immernoch auf dasselbe Objekt, welches jetzt allerdings verändert wurde. Werden nun x und y ausgegeben erhält man für beide Varibalen die veränderte Liste [1,42,3] .

Die List-Klasse verfügt noch über weitere mutierende Methoden, wie zum Beispiel reverse:

```
x = [1,2,3]
y = x
y.reverse()
print(y)      # Ausgabe [3,2,1]
print(x)      # Ausgabe [3,2,1]
```

Oft sind mutierende Methoden auch als Funktionen implementiert, die das veränderte Objekt auch als Ergebnis liefern. Da sie aber Objekte, welche außerhalb der Methodendefinition existieren, auch verändern, würde es eigentlich auch ausreichen, sie als Prozeduren ohne Rückgabewert zu definieren. Prozeduren können also nicht nur mehrere Ausgaben zusammenfassen. Sie können auch übergebene Objekte mutieren. Somit ergeben sich mit mutierenden Updates auch neue sinnvolle Möglichkeiten Prozeduren selber zu definieren.

Mit [] = können auch Teillisten ersetzt werden. Hierfür gibt man wie beim Slektieren von Teillisten (und auch Teilstrings) einen *Slice* an:

```
l = [1,2,3,4,5,6]
l[2:4] = [10,11,12]
print(l)              # Ausgabe: [1,2,10,11,12,5,6]
l[2:5] = []
print(l)              # Ausgabe: [1,2,5,6]
```

Falls man bei eine Slice mit der Startposition 0 beginnt bzw. bei der Endposition Länge der Liste endet, gibt es weitere abkürzende Schreibeweisen:

```
l = [0,1,2,3,4,5,6]
print(l[2:4])         # Ausgabe: [2,3]
print(l[:4])          # Ausgabe: [0,1,2,3]
print(l[3:])          # Ausgabe: [3,4,5,6]
```

Genau so können Slices auch für Strings verwendet werden. Außerdem können diese Slices auch zur Mutation von Listen verwendet werden:

```
l = [1,2,3,4,5,6]
l[3:] = [10,11,12]
print(l)              # Ausgabe: [1,2,3,10,11,12]
l[:4] = []
print(l)              # Ausgabe: [11,12]
```

Um zu verstehen, wie man mit Listmutationen programmiert, implementieren wir die mutierende *reverse* Methode selber. Zunächst müssen wir überlegen, ob es sich um eine Funktion oder eine Prozedur handeln soll. Zwar wollen wir die übergebene Liste verändern, dies wollen wir aber als Seiteneffekt machen, was ja das Wesen einer Mutation ist. Einen Rückgabewert benötigen wir also nicht und programmieren eine Prozedur.

```
def rev(l):
  for i in range(len(l)):
    l[i] = l[len(l)-i-1]
```

Führt man dieses Programm aus ergibt sich folgendes:

```
l = [1,2,3,4]
rev(l)
print(l)          # Ausgabe: [4,3,3,4]
l = [1,2,3,4,5]
rev(l)
print(l)          # Ausgabe: [5,4,3,4,5]
```

Das Programm verhält sich also noch nicht korrekt. Der Grund ist, dass das Kopieren zunächst Werte überschreibt, die später noch gebraucht werden.

Lösung: Tauschen von Werten

Hierzu verwendet man eine Hilfsvariable, in der man einen Wert zwischen speichert. Außerdem dürfen wir, wenn wir tauschen nur noch das halbe Liste durchlaufen.

```
def rev(l):
  for i in range(len(l) // 2):  # ganzzahlige Division
    h = l[i]
    l[i] = l[len(l)-i-1]
    l[len(l)-i-1] = h

l = [1,2,3,4]
rev(l)
print(l)          # Ausgabe: [4,3,2,1]
l = [1,2,3,4,5]
rev(l)
print(l)          # Ausgabe: [5,4,3,2,1]
```

Klassen, bei welchen alle Methoden, ein Objekt nicht mutieren, nennt mann auch *immutable*. Beispiele sind alle Klassen für Zahlen, wie wir sie bisher kennen gelernt haben, die Klasse für boolesche Werte und in Python auch die Klasse String. Dies muss so aber nicht für alle Programmiersprachen gelten. In Ruby können Strings zum Beispiel mutiert werden.

Die Kombination von mutierenden und nicht mutierenden Methoden innerhalb einer Klasse birgt einige Gefahren, welche wir uns später noch genauer ansehen werden.

3.4 Objektorientierte Datenmodellierung

Nachdem wir die Eigenheiten der Objektidentität und mutierenden Methoden verstanden haben, wollen wir uns die Definition eigener Klassen und Objekte anschauen.

In einer Klasse werden Objekte gleicher Art zusammen gefasst. Die Klasse stellt sozusagen das Gerüst für Objekte gleicher Bauart dar. Die Daten, welche in einem Objekt einer Klasse gespeichert werden, heißen Attribute und können in jedem einzelnen Objekt mit anderen Werten belegt werden. Eine Klassedefinition verfügt außerdem über Methoden, welche später für jedes konkrete Objekt der Klasse aufgerufen werden können und über die der Zugriff auf die Attribute erfolgen kann. Als erstes Beispiel wollen wir als neuen Zahlentyp Brüche (Fractional) definieren.

Zunächst definieren wir das Gerüst der Klasse Fractional. Als Konvention beginnen Klassennamen in Python mit einem Großbuchstaben.

```
class Fractional :

  def __init__(self,zaehler,nenner) :
    self.zaehler = zaehler
    self.nenner  = nenner
```

Die spezielle Methode `__init__` heißt in der objektorientierten Programmierung *Konstruktor* und wird aufgerufen, wenn ein neues Objekt der Klasse Fractional angelegt wird. Zur Generierung eines Bruchs verwenden wir zwei Argumente, den Zähler und den Nenner. Außerdem erhalten alle Methoden in Python, und damit auch der Konstruktor, einen weiteren ersten Parameter, welcher das Objekt selber darstellt. Für den Moment stellen wir uns einfach vor, dass dieser Parameter das Objekt selber ist, also der knokrete Bruch, den wir konstruieren wollen. Als Varibalenname verwendet man für diesen Parameter eigentlich immer `self`, was in Python aber kein Schlüsselwort ist, sondern eine Variable.

Die beiden übergebenen Werte speichern wir in Attributvariablen. Diese Attributvariablen sind private Felder des Objekts (an welches ja die Variable `self` gebunden ist) und werden durch `self.`*variablenname* verwendet. Wir können uns dies so vorstellen, dass eine Klasse einen Satz von Varibalen als Zustand hält. Man kann später

Als nächstes wollen wir eine Methode zur Klassendefinition hinzufügen, welche einen Bruch ausgibt:

```
  def show(self) :
    print(str(self.zaehler) + '/' + str(self.nenner))
```

Mit Hilfe der Variablen `self` können wir die zuvor im Konstruktor gesetzten Attributvariablen `zaehler` und `nenner` auslesen.

Dann können wir unsere Klasse bereits wie folgt verwenden:

```
x = Fractional(5,3)
x.show()                    # --> 5/3
```

Der Konstruktoraufruf Fractional(5,3) erzeugt ein neues Objekt, welches dann durch den Konstruktor der Klasse Fractional initialisiert wird, also die Attributevariblen zaehler auf 5 und nenner auf 3 gesetzt werden. Für eigene Objekte erkennt man so ganz genau, an welchen Stellen neue Objekte erstellt werden. Für vordefinierte Objekte, erfolgt dies implizit, wie z.B. bei 42 oder 'Hallo'.

Beim Aufruf einer Objektmethode, wird das Objekt für die Variable self nicht explizit übergeben. Es handelt sich aber immer um das Objekt, zu welchem man die Methode aufruft, also das Objekt vor dem Punkt. Wenn wir also für ein String-Objekt die Methode upper() aufrufen:

```
s = "Hallo"
s.upper()
```

so können wir uns die Implementierung in der Klasse String wie folgt vorstellen:

```
class String:

    ...

    def upper(self) :
```

Im Rumpf von upper () ist die Varibale self an das Objekt gebungen, an welches die Varibale s oben gebunden ist.

Kommen wir noch einmal zurück zur Definition der show-Methode in unser Klasse Fractional. In Python ist es eher ungewöhnlich, spezielle Ausgabemethoden zu definieren. Statt dessen definiert man für seine Objekte meist eine spezielle Methode zur Umwandlung in einen String, welcher dann ausgegeben werden kann. Hierbei können wir direkt auch unsere Ausgabe optimieren und unechte Brüche (Zähler > Nenner) als gemischte Brüche ausgeben.

```
def __str__(self) :
  if self.nenner == 1 :
    return str(self.zaehler)
  else :
    return (('-' if self.zaehler < 0 else '') +
            (str(self.zaehler // self.nenner) + ' '
             if self.zaehler > self.nenner
             else '') +
            str(self.zaehler % self.nenner) + '/' +
            str(self.nenner))
```

Beachte hierbei, dass das Ergebnis von $-2\%3$ 1 ist und wir das Vorzeichen somit explizit ausgeben müssen.

Durch die Verwendung des Methodennamen __str__, können Objekte unserer Klasse auch mit print ausgegeben und mit Hilfe der Funktion str in einen String umgewandelt werden. Wir können wir unsere Klasse nun also wie folgt verwenden:

```
x = Fractional.new(5,3)
print(x)                    # Ausgabe: 1 1/3
print("Mein Bruch lautet: "+str(x))
```

Unschön ist aber noch, dass es mehrere redundante Darstellungen der gleichen Brüche gibt:

```
x = Fractional(5,3)
y = Fractional(10,6)
print(x)                    # Ausgabe: 1 1/3
print(y)                    # Ausgabe: 1 2/6
```

Es wäre besser, wenn Brüche immer direkt gekürzt würden, wenn sie angelegt werden. Dies ist aber mit Hilfe der ggt-Funktion kein großes Problem:

```
def __init__(self,zaehler,nenner) :
  if nenner==0 :
    print('Achtung Division durch 0!')

  ggt = self._ggt(zaehler,nenner)
  if nenner < 0 :
    ggt = -ggt
  self.zaehler = zaehler // ggt
  self.nenner  = nenner  // ggt

def _ggt(self,a,b) :
  a = abs(a)
  b = abs(b)
  while a!=0 and b!=0:
    if a>b :
      a=a%b
    else :
      b=b%a

  if a==0 :
    return b
  else :
    return a
```

Wir erhalten in obigem Programm nun auch für den Bruch $\mathrm{Fractional}(10{,}6)$ die Ausgabe $1\ 1/3$. Die ggt-Funktion ist nur eine Hilfsfunktion innehalb unserer Klasse und soll eigentlich nicht von außen aufgerufen werden. In Python gibt es hierfür eine Konvention, dass eine solche Methode mit zwei Unterstrichen beginnt, wie es auch schon für die Methode __str__ galt. Gleiches gilt in Python auch für ein geheimes Attribut. Andere Programmiersprachen bieten hier komplexere Konzepte, welche es tatsächlihc erlauben Attribute bzw. Methoden vom externen Zugriff zu schützen. Python unterstützt solche Konzepte nicht, erwartet aber vom Programmierer eine gewisse Programmierdisziplin.

Im nächsten Schritt können wir versuchen die Multiplikation von Brüchen zu definieren. Wie immer benötigt der Operator * einen weiteren Bruch als Argument. Da die Methode * das Objekt selber nicht mutieren soll, müssen wir hier ein neues Fractional Objekt anlegen, welches das Produkt der beiden Brüche repräsentiert:

```
def __mul__(self,other) :
  return Fractional(self.zaehler * other.zaehler,
                    self.nenner * other.nenner)
```

Ähnlich wie bei __str__ gibt es in Python die Konvention, dass der Operator * in einen Aufruf der Methode __mult__ der zugehörigen Klasse übersetzt wird.

Wie bei alle Zahlen-Objekte üblich realisieren wir auch unsere Brüche als immutable Klasse. Wir schreiben ein kleines Testprogramm und erhalten:

```
x = Fractional(5,3)
y = Fractional(6,10)
print((x*y))                    # Ausgabe: 1
```

Beachte, dass es nicht notwendig ist, das Ergebnis noch einmal explizit zu kürzen, da der Konstruktor des neuen Objekts dies für uns übernimmt.

Entsprechend können wir auch die Addition und noch weitere fehlende Methoden definieren:

```
  def __add__(self,other) :
    neuer_zaehler = self.zaehler * other.nenner +
                    other.zaehler * self.nenner
    neuer_nenner  = self.nenner * other.nenner
    return Fractional(neuer_zaehler,neuer_nenner)

  def __sub__(self,other) :
    neuer_zaehler = self.zaehler * other.nenner -
                    other.zaehler * self.nenner
    neuer_nenner  = self.nenner * other.nenner
    return Fractional(neuer_zaehler,neuer_nenner)

  def __neg__(self) :     # fuer das unäre Minus
    return Fractional(-self.zaehler,self.nenner)

  def __truediv__(self,other) : # fuer den Operator /
    return Fractional(self.zaehler * other.nenner,
                      self.nenner * other.zaehler)
```

Das Programm kann auf der Web-Seite der Vorlesung heruntergeladen werden.

Als weiteres Beispiel wollen wir eine Klasse definieren, bei der es mutierende Methoden gibt: ein Konto.

```
class Konto :

  def __init__(self, betrag) :
    self.kontostand = betrag

  def get_kontostand(self) :
    return self.kontostand

  def abheben(self, betrag) :
    self.kontostand = self.kontostand - betrag

  def einzahlen(self, betrag) :
    self.kontostand = self.kontostand + betrag
```

Um den aktuellen Kontostand sehen zu können, könnten wir auch von außen auf das Attribut kontostand zugreifen. Dieses möchte man aber in der Regel vermeiden und würde seine Attribute eigentlich gerne verstecken. Deshalb bieten wir für diese Klasse eine sogenannte

getter-Methode an, welche es dem Benutzer dieser Klasse ermöglicht, den Kontostang abzufragen. Auch in der Definition der Klasse Fractional hätten wir getter-Methoden für den Zähler und den Nenner eines Bruchs definieren können. Wir haben dort darauf verzichtet, da wir dem Benutzer nur die Möglichkeit geben wollten einfache Operationen über Brüchen durchzuführen.

Die Methoden abheben und einzahlen verändern den Zustand unserer Objekte, d.h. es wird keine neues Objekt konstruiert. Vielmehr wird unsere Attributvariable self.kontostand verändert.

Eine Verwendung des Kontos sieht dann wie folgt aus:

```
k1 = Konto(200)
k2 = Konto(50)
k1.abheben(50)
print(k1.get_kontostand())          # Ausgabe: 150
k2.einzahlen(1000)
print(k2.get_kontostand())          # Ausgabe: 1050
```

Das Beispiel verdeutlicht auch die Objektidentität unserer Konten. Wir könenn mehrere Konten anlegen, welche alle einzelnen, unabhängig von einander verändert werden können.

Ähnlich wie getter-Methoden, bietet man dem Benutzer einer Klasse oft auch setter-Methoden für bestimmte Attribute an. In unserem Beispiel wäre sie eine Methode zum setzten des Kontostands auf einen bestimmten Wert:

```
  def set_kontostand(self, betrag):
    self.kontostand = betrag
```

Da solch eine Methode bei einem Konto nicht sinnvoll erscheint (wir würden ja Geld vernichten bzw. generieren), lassen wir solch eine setter-Methode hier weg und erlauben nur das Ein- und Auszahlen. Um dennoch den Begriff der setter-Methode zu verstehen, haben wir diese Methode auch vorgestellt.

3.5 Mutierende und nicht mutierende Methoden

Wir haben mit mutierenden und nicht mutierenden Methoden zwei unterschiedliche Programmierstile kennen gelernt, welche in der objektorientierten Programmierung kombiniert werden. Es zeigt sich aber, dass die Kombination der beiden Ansätze neue Gefahren birgt, wo man sie bisher eigentlich nicht vermutet hat. Als Beispiel wollen wir ein Programm schreiben, welches mehrere Zeilen vom Benutzter einliest und in einer Liste speichert. Wir deinieren also zunächst die folgende Funktion:

```
def liesBisLeerzeile():

  res = []
  s = input()
  while s != "":
    res = res + [s]
    s = input()

  return res
```

```
l = liesBisLeerZeile()
```

Als nächstes wollen wir alle Vorkommen eines beliebigen Strings in dieser Liste löschen, wofür eine Funktion delete definieren. Wir entschließen uns, diese Funktion nicht mutierend zu realisieren und entwickeln das folgende Programm:

```
def delete(x,l) :

  i = 0

  while i < len(l) :

    if l[i] == x :
      l = l[0:i] + l[i+1:]  # hier schneiden wir das zu loeschende Element aus
      i = i + 2
    else :
      i = i + 1

  return l
```

Wir können nun weiter an unserem Hauptprogramm schreiben und ergänzen die die folgenden Zeilen:

```
l = liesBisLeerZeile()

loeschen = input("Was soll ich loeschen? ")

l1 = delete(loeschen,l)

l.reverse()

print('Nach dem Loeschen ergibt sich: ')
print(l1)

print('Nach dem Umdrehen ergibt sich: ')
print(l)
```

Startet man dieses Programm, so erhält man den folgenden möglichen Programmablauf. Zunächst geben wir ein paar Zeilen eine, z.B.

```
Das
Gras
ist
sehr
grún
.

Was soll ich loeschen? sehr
Nach dem Loeschen ergibt sich:
['Das', 'Gras', 'ist', 'grún', '.']
Nach dem Umdrehen ergibt sich:
['.', 'gruen', 'sehr', 'ist', 'Gras', 'Das']
```

Oder bei einem zweiten Durchlauf:

```
Das
Gras
ist
ein
bisschen
braun
.

Was soll ich loeschen? sehr
Nach dem Loeschen ergibt sich:
['.', 'braun', 'bisschen', 'ein', 'ist', 'Gras', 'Das']
Nach dem Umdrehen ergibt sich:
['.', 'braun', 'bisschen', 'ein', 'ist', 'Gras', 'Das']
```

Das Programm hat also im letzten Durchlauf nicht nur die Liste in der Variablen l, sondern auch die Liste in der Variablen l1 umgedreht, was wir nicht erwartet haben. Dies kann nur durch die mutierende Methode reverse geschehen sein. Da sie nur einmal aufgerufen wird, kann das aber nur bedeuten, dass beide Variablen an dasselbe Objekt gebunden sind. Wie kommt es dazu?

Der Fehler findet sich tatsächlich in der nicht-mutierenden Methode delete, welche für den Fall, dass der zu löschende Wert x nicht in der Liste vorkommt, einfach die Liste unverändert zurück gibt, also dasselbe Objekt als Rückgabewert hat, wie wir es als Parameter übergeben haben. Dies bedeutet, dass ein und dasselbe Objekt an ganz unterschiedlichen Programmstellen vorkommt, ohne dass die Programmiererin sich dessen bewusst ist. Mutiert sie nun ein Liste, werden gefühlt alle Vorkommen dieser Liste im Programm mutiert. Solche Fehler treten in großen realen Systemen gelegentlich und sind sehr schwer zu finden. Jeder Test (man führt sogenannte Unittests für definierete Funktionen aus) für die Methode delete wäre erfolgreich, da Tests in der Regel nur überprüfen, ob eine gegebene Funktion für alle Eingaben das korrekte Ergebnis liefert. Obejktreferenzen werden hierbei in der Regel nicht verglichen.

Es zeigt sich also, dass wir auch bei der Programmierung von nicht mutierenden Funktionen und Methoden, sehr vorsichtig vorgehen müssen, wenn im System weitere mutierende Methoden für denselben Datentypen existieren. Untersucht man die anderen nicht mutierenden Methoden für Listen, so sieht man, dass Python hierbei tatsächlich vorsichtiger vorgeht, wie die folgenden Beispiele zeigen:

```
l = [1,2,3,4]
l1 = l + []     # l = l * 1 oder l = [] + l oder l.reverse
l[1] = 42
print(l)        # [1, 42, 3, 4]
print(l1)       # [1, 2, 3, 4], unverändert
```

Die Liste l1 bleibt also in allen Fällen von der Veränderung des Objekts in l in der dirtten Zeile unverändert. Alle verwendenten nicht mutierenden Methoden der Klasse List also immer eine eine Kopie ihres Parameterobjekts, auch wenn beide Listen identisch sind.

Für unser Programm bedeutet dies also, dass wir dafür sorgen müssen, dass unsere Funktion delete auch für die Fälle, in denen wir kein Element aus der Liste löschen, eine Kopie anlegen. Objektorientierte Programmiersprachen bieten hierzu meist Methoden zum *Klonen*

von Objekten an. In Python bietet die Klasse List gleiche mehrere Methoden hierfür an: l + "", l [:], list (l).

Hiermit können wir unsere Funktion delete korrigieren:

```
def delete(x,l) :

  i = 0

  while i < len(l) :

    if l[i] == x :
      l = l[0:i] + l[i+1:]  # hier schneiden wir das
                            # zu loeschende Element aus
    else :
      i = i + 1

  return list(l)      # So garantieren wir, dass das
                      # Objekt in l kopiert wird.
```

Nun erhalten wir auch einen sinnvolles Programmverhalten, wenn wir einen String löschen, der vorher gar nicht eingegeben wurde.

Bei der Programmierung von nicht mutierenden Methoden muss man also folgendes beachten:

> **Merksatz:** Falls es für einen gleichen Datentypen auch noch mutierende Methoden gibt, müssen die nicht mutierenden Methoden für alle Eingaben neue Objekte als Ergebnis liefern, d.h. sie dürfen keine übergebenen Objekte unkopiert zurückgeben. Funktionen und Methoden, die sich an diese Regel nicht halten, müssen als inkorrekt angesehen werden.

3.6 Dateien lesen und schreiben

Bisher haben wir Zeichenfolgen immer nur vom Benutzer eingelesen oder auf der Konsole ausgegeben. Interessanter werden Programme aber, wenn man auch Dateien lesen und schreiben kann. In Python stehen hierzu sogenannte Datei-Händler zur verfügung, welche man sich als eine Art Zeiger auf eine Datei vorstellen kann.

Zum Einlesen geht man wie folgt vor:

```
f = open('file.txt')
text = f.read()
f.close()
```

und zum Schreiben:

```
f = open('file.txt','w')
f.write(text)
f.close()
```

Hierbei muss text ein String sein. Die Methode write ist also weniger flexibel als die Methode print und konvertiert andere Datentypen nicht auotomatisch.

3.7 Heterogene Listen

Bisher haben wir Listen in der Regel dazu verwendet, gleichartige Daten zusammenzufassen. Dies ist in Python aber nicht notwendig. Wir können auch unterschiedliche Arten von Daten in einer Liste zusammenfassen. Dann sollten wir uns aber darüber im Klaren sein, an welcher Stelle wir welche Art von Daten abgelegt haben, um diese später sinnvoll verwenden zu können. Als Beispiel können wir eine Liste verwenden um z.B. ein Tripel bestehend aus einem Namen (String), der Größe der Person (Zahl) und ihrer aktuellen Anwesenheit (booleschen Wert) zu definieren:

```
person1 = ['Werner',188,True]
person2 = ['Kalr',168,False]

if person1[1]>person2[1]
  then print(person1[0]+' ist groesser als '+person2[0])
  else print(person2[0]+' ist groesser als '+person1[0])
end
```

Eine alternative und sicherlich schönere Variante wäre natürlich die Definition einer eigenen Klasse $Person$, mit entsprechenden Attributen, getter/setter-Methoden.

Da Listen selber auch wieder Werte darstellen, können wir tatsächlich auch wieder Listen als Einträge in einer Liste verwenden, wie folgendes Beispiel verdeutlicht:

```
a = [2,True,[42,[3,'Hallo'],'Leute'],5.0]

print(len(l))      # -> 4
print(a[0])        # -> 2
print(a[1])        # -> True
print(a[2])        # -> [42,[3,'Hallo'],'Leute']
print(a[3])        # -> 5.0

print(a[2][2])     # -> 'Leute'
print(a[2][1][1]) # -> 'Hallo'
```

Wir können also durch mehrfache Liste-Zugriffe auch nach und nach in der verschachtelten Liste-Struktur absteigen.

Beachte folgenden semantischen Unterschied beim Einfügen einer Listen in eine Liste:

```
a = [1,2,3,4,5]
a[3:4] = [42,43,44] # Ersetzen einer TeilListe
a[2] = [0,0]        # Ersetzen eines Elements durch eine Liste
print(a)            # -> [1,2,[0,0],42,43,44,5]
```

Betrachten wir noch einmal das vorherige Beispiel der Personen. Wenn wir nicht nur zwei Personen speichern wollen, sondern beliebig viele, sollten wir anstelle zweier konkreter Personen besser ein Liste von Personen verwenden:

```
personen = [['Werner',188,True],['Karl',162,False],
            ['Elfriede',168,False]]
```

Wie kann man solch ein Liste von Listen interpretieren?

Es handelt sich um eine Tabelle, d.h. wir haben mehrere Zeilen, für die in jeder Spalte gleichartige Information steht, was man auch mit folgender Darstellung noch unterstreichen kann:

```
personen = [['Werner'   ,188,True ],
            ['Karl'     ,172,False],
            ['Elfriede',168,False]]
```

Solche Tabellen kennen wir auch schon aus Tabellenkalkulationen. Haben alle Einträge in der Liste die gleiche Struktur nennt man solche Listen auch mehrdimensionale Felder (hier speziell zweidimensional). Einige Programmiersprachen unterstützen mehrdimensionale Felder explizit und sichern durch statische Typisierung auch zu, dass alle inneren Listen die gleiche Struktur und Größe besitzen. In Python ist dies nicht der Fall und der Programmierer ist hierfür selber verantwortlich.

Ein Austausch solcher Tabellen auch über verschiedene Programme hinweg, wäre natürlich wünschenswert. Hierzu verwendet man gerne Dateien des Formats *comma separated values* (csv), in welchem Tabellen zeilenweise in einer Datei gespeichert werden. Die einzelnen Spalten sind dann durch ein Trennsymbol, in der Regel ein Komma (in der Deutschen Version von Tabellenkalkulationsprogrammen standardmäßig durch ein Semikolon) getrennt. In obigem Beispiel, sähe der zugehörige Dateiinhalt wie folgt aus:

```
Frank,188,True
Steven,172,False
Sandro,168,False
```

Um nun aus unserem zweidimensionalen Feld eine entsprechende csv-Datei zu erzeugen, können wir folgende Prozedur definieren:

```
def csv_write(file,tab):
  str = ''
  for i in range(len(tab)):
    for j in range(len(tab[i])-1):
      str = str + tab[i][j] + ','

    str = str + str(tab[i][len(tab[i])-1]) + '\n'

  f = open(file,"w")
  f.write(str)
  f.close()
```

Das einlesen einer csv-Datei gestalltet sich etwas schwieriger. Wir können aber für eine erste Version die String-Methode `split` verwenden, welche einen String über angegebenen Zeichenfolgen aufsplittet. Die Semantik der Methode sollte an den folgenden Beispielen klar werden:

```
print('a,b,c,d,ef,g,,h'.split(','))
# -> ['a','b','c','d','ef','g','','h']
print('ab\ncd,ef\n,ghi\n'.split('\n'))
# -> ['ab','cd,ef',',ghi','']
print('Die Katze und der Hund').split('und')
# -> ['Die Katze ',' der H','']
```

Mit Hilfe dieser Methode läßt sich dann recht einfach eine csv-Datei in eine zweidimensionale Tabelle einlesen:

```
def csv_read(file) :

  # Einlesen der Datei
  f = open(file)
  str = f.read()
  f.close()

  # Falls die Datei mit einem Zeilenumbruch
  # endet, loeschen wir diesen.
  if str[len(str)-1] == '\n' :
    str = str[:len(str)-1]

  # Zerteilen in Zeilen
  lines = str.split('\n')

  # Zerteilen jeder Zeile in Spalten
  for i in range(len(lines)) :
    lines[i] = lines[i].split(',')

  return lines
```

Mit diesen beiden Funktionen ist es nun möglich zwiedimensionale Tabellen in csv-Dateien umzuwandeln und umgekehrt. Dieses Dateiformat kann auch von Tabellenkalkulationen gelesen und geschrieben werden, so dass ein Austausch mit Tabellenkalkulationsprogrammen möglich ist.

Unsere Implementierung ist noch nicht perfekt, da wir vorkommende Strings, die Kommas enhalten, nicht korrekt im csv-Format repräsentieren. Hierzu verwendet man auch in csv-Dateien Anführungszeichen. Wir werden später vordefinierte csv-Bibliotheken verwenden, welche dies korrekt umsetzen.

In der Vorlesung entwickeln wir außerdem ein etwas komplexeres Beispiel, mit welchem wir eine Mitbringlisten-Anwendung realisieren. Hierbei verwenden wir Tabellen (zweidimensionale Felder), welche wir als csv-Datei auf der Festplatte speichern und so auch in einer Tabellenkalkulation bearbeiten können.

3.8 Reguläre Ausdrücke

Ähnlich wie die EBNF sind reguläre Ausdrücke ein mächtiges Mittel zur Sprachbeschreibung, wenngleich sie weniger ausdrucksstark sind. Mit Hilfe von regulären Ausdrücken können, sehr einfach Teilausdrücke, aber auch bestimmte Muster von Zeichenfolgen in Zeichenreihen gefunden werden. Ein regulärer Ausdruck wird in Python durch /-Symbole begrenzt. Reguläre Ausdrücke sind auch Werte, d.h. sie können auch in Variablen und Datenstrukturen gespeichert werden.

Die einfachsten reguläre Ausdrücke sind Zeichenfolgen, z.B. 'cd' oder 'Hallo'. Mit Hilfe der Funktionen re.match und re.search können reguläre Ausdrücke gegen Strings *gematcht* werden. Dies bezeichnet man auch als *Pattern Matching*, da der reguläre Ausdruck als Mus-

ter (engl. Pattern) in einem String gesucht wird (engl. match). Das Ergebnis des Matchings ist ein Match-Objekt, welches Informationen zum erfolgreichen Match enthält. Im Fall von match wird das Pattern Matching direkt am Anfang des Strings durchgeführt. Bei search, wird überprüft, ab der reguläre Ausdruck an irgendeiner Stelle des Strings passt. Falls das Matching nicht erfolgreich ist, liefern beide Funktionen das Ergebnis None:

```
import re

m = re.match('abc','abcdcd')
print(m)  # -> <_sre.SRE_Match object; span=(0, 3), match='abc'>
m = re.match('cd','abcdcd')
print(m)  # -> None
m = re.search('cd','abcdef')
print(m)  # -> <_sre.SRE_Match object; span=(2, 4), match='cd'>
```

Das Match-Objekt enthält eine Reihe von Informationen zum gematchten String:

- den Span, von wo bis wo das Matching erfolgreich war
- der gematchten String (hier ist der reguläre Ausdruck und der String noch identisch, was sich aber ändern wird)
- falls mehrere Matches möglich sind, liefert search das linkeste mögliche Matching

Für diese Komponenten bietet die Python-Bibliothek für reguläre Ausdrücke auch passende getter-Methoden:

```
m = re.search('cd','abcdefcdef')
print(m)          # -> <_sre.SRE_Match object; span=(2, 4), match='cd'>
print(m.group())  # -> cd
print(m.start())  # -> 2
print(m.end())    # -> 4
```

wobei start und end Anfangs- und Endpositonen des gematchten Spans sind. Die zweite Position, an der der reguläre Ausdruck gepasst hätte, bleibt unberücksichtigt.

Diese Funktionen erscheint im Moment noch überflüssig, da wir bisher nur Teilstrings gesucht haben. Es wird aber nützlich sein, wenn allgemeinere Muster formuliert werden können, welche ggf. auch unterschiedliche Teilstrings matchen können. Als erstes allgemeineres Beispiel betrachten wir den regulären Ausdruck, der auf alle Ziffern passt $\backslash d$:

```
m = re.search('c\dd','abc7def')
print(m.group())   # -> c7d
```

In diesem Beispiel ist schon der gemathcte String interessant, da er z.B. aus einer Eingabe kommen kann.

Für andere Klassen von Zeichen gibt es entsprechende reguläre Ausdrücke, z.B. für Whitespaces:

```
re.search('c\sd','abc d').group()    ↪ 'c d'
re.search('c\sd','abc\ndef').group() ↪ 'c\nd'
re.search('c\sd','abc \ndef')        ↪ None
```

Mit $\backslash w$ können Buchstaben und Ziffern gematcht werden:

```
re.search('c\we','abcdef').group()    ⇝ 'cde'
re.search('c\wd','abc3def').group()   ⇝ 'c3d'
re.search('c\sd','abc(de)f')  ⇝ None
```

Der Punkt . steht für ein beliebiges Zeichen:

```
re.search('c.e','abcdef').group()    ⇝ 'cde'
```

Außerdem ist es möglich für einzelne Buchstabe Bereiche anzugeben, z.B. $[p-t]$ oder $[a-zA-Z]$ oder $[0-9]$ oder konkrete mögliche Buchstaben aufzuzählen, z.B. $[Python]$:

```
re.search('[c-d]','abcdef').group()        ⇝ 'c'
re.search('[c-d][c-d]','abcdef').group()   ⇝ 'cd'
re.search('[Info]','abcdef').group()       ⇝ 'f'
```

Außerdem können bestimmte Buchstaben oder Bereiche ausgenommen werden:

```
re.search('[^ab]','abcdef').group()    ⇝ 'c'
```

Es ist auch möglich, Alternativen zu definieren:

```
re.search('c|d','abcdef').group()              ⇝ 'c'
re.search('c|d','abCdef').group()              ⇝ 'd'
re.search('Hi|Hallo','Hi Willi').group()       ⇝ 'Hi'
re.search('Hi|Hallo','Hallo Willi').group()    ⇝ 'Hallo'
```

Der Alternativen-Operator | bindet hierbei schwächer als die Hintereinanderschreibung von Buchstaben, so dass der reguläre Ausdruck

$'Hi|Hallo'$ identisch mit $'(Hi)|(Hallo)'$ ist, aber eben nicht identisch mit $'H(i|H)allo'$.

Mit Hilfe von Klammern können reguläre Ausdrücke auch strukturiert werden, z.B.:

```
re.search('b(c|x)d','abcdef').group()    ⇝ 'bcd'
re.search('b(c|x)d','abxdef').group()    ⇝ 'bxd'
```

Was bedeutet dieser reguläre Ausdruck: $/a(|x)b/$?

```
re.search('a(|x)b','ab').group()    ⇝ 'ab'
re.search('a(|x)b','axb').group()    ⇝ 'axb'
```

Links vom | steht hier der leere reguläre Ausdruck (entspricht dem leeren Wort), welcher auf den String der Länge Null match.

So kann man also einfach ausdrücken, dass etwas (beschrieben durch einen regulären Ausdruck) optional ist. Alternativ kann man hierfür auch das ? als nachgestellten Operator (Postfix) verwenden:

```
re.search('cd?e','abcdef').group()      ⇝ 'cde'
re.search('cx?d','abcdef') .group()     ⇝ 'cd'
re.search('(cd)?e','abcdef').group()    ⇝ 'cde'
re.search('(cx)?e','abcdef') .group()   ⇝ 'e'
```

Beachte, dass ? stärker bindet, als das Hintereinanderschreiben, sich also nur auf den letzten regulären Ausdruck vor dem ? bezieht (oft Buchstabe):

/ab?c/ ist also idetisch mit /a(b?)c/, aber eben nicht identisch mit /(ab)?c/.

Abschließend kann man noch wie bei der EBNF Wiederholungen $(+)$ oder optionale Wiederholungen $(*)$ zulassen. Man beachte, dass ein Stern in Postfixnotation anstelle geschweifter Klammern wie in der EBNF verwendet wird:

```
re.search('cx*d','abcdef').group()      ⇝ 'cd'
re.search('cx+d','abcdef')              ⇝ None
re.search('cx+d','abcxxxdef').group()   ⇝ 'cxxxd'
```

Für Wiederholungen stellt sich nun die Frage, bis woin eine passender Match geht:

```
re.search('cx*','abcdef').group()       ⇝ 'c'
re.search('cx+','abcxxxdef').group()    ⇝ 'cxxx'
```

Es wird also nicht nur das linkeste Matching gesucht, sondern auch noch das von dort mögliche längste Matching. Im zweoiten Beispiel also nicht 'c' oder 'cx', sondern 'cxxx'.

Zusätzlich zum Überprüfen, ob ein regulärer Ausdruck passt und an welcher Stelle er passt möchte man häufig auch noch weitere Informationen erhalten.

Bsp.:
Gegeben ist ein String str der Form

Name: Telefonnummer\n Name: Telefonnummer\n
(z.B.: Sandro Esquivel: 1234\n Frank Huch: 7277\n)

Nun suchen wir Franks Telefonnummer.

```
str = 'Sandro Esquivel: 1234\n Frank Huch: 7277\n Ulrike ...'

eintrag = re.search('Frank Huch: [0-9]+',str).group()
        ⇝ 'Frank Huch: 7277'
```

Hieraus können wir nun die Telefonummer extrahieren, z.B. wieder mit einen regulären Ausdruck:

```
re.search('[0-9]+',eintrag).group()   ⇝ '7277'
```

Eine Alternative und oft noch komfortablere Möglich, dieses Problem zu lösen bilden sogenannte *capture groups*. Die Idee ist, dass beim Matchen des regulären Ausdrucks, auch die Information vorgehalten wird, gegen welche Teilstrings die verwendeten geklammerten Telausdrücke gematcht wurden. So können wir im Beispiel einfach die Telefonnummer im regulären Ausdruck klammern:

```
m = re.search('Frank Huch: ([0-9]+)',str)
```

Im nächsten Schritt können wir dann die Telefonnummer einfach an einem Index abfragen:

```
telefonnummer = m.group(1)        ⇝ '7277'
```

Die Idee ist, dass alle capture groups von links nach rechts durchnummeriert werden und das Matchobjekt diese an den entsprechenden Indizes zur Verfügung stellt. In der Realisierung leifert ein Match-Objekt eine Liste, welches dann indiziert wird. Listen werden wir im nächsten Kapitel genauer kenen lernen und dann sicherlich auch die Notation besser verstehen können.

Das folgende Beispiel mit mehreren capture groups verdeutlicht die Indizes, über welche anschließend auf die Teilmatches zugegriffen werden kann:

Bsp.:

```
                    1     23        4
                    ↓     ↓↓        ↓
m = re.search('(a|b)((b|a)+)(..)','abbbacbb')
```

liefert für die folgenden Capture Groups:

```
m.group()  ⇝ 'abbbacb'
m.group(0) ⇝ 'abbbacb'
m.group(1) ⇝ 'a'
m.group(2) ⇝ 'bbba'
m.group(3) ⇝ 'a'
m.group(4) ⇝ 'cb'
```

Die Capture Group an Position 3, zeigt an, auf welchen Teilstring der reguläre Ausdruck beim letzten Matching der Wiederholung gepasst hat.

Als etwas komplexeres Beispiel wollen wir in einem Text ein Kennzeichen suchen. Kennzeichehn können wir mit dem folgenden regulären Ausdruck definieren.

```
regexp_kennzeichen = '[A–Z][A–Z]?[A–Z]?(-|␣)[A–Z][A–Z]?␣[1-9][0-9]*'
```

Dann könenn wir z.B. einen Text aus einer Datei einlesen und alle vorkommenen Auto-Kennzeichen ausgeben:

```
import re

h = open('kenz.txt')
text = h.read()
h.close()

regexp_kennzeichen = '[A–Z][A–Z]?[A–Z]?(-|␣)[A–Z][A–Z]?␣[1-9][0-9]*'

m = re.search(regexp_kennzeichen,text)

while m!=None:
  print(m.group())
  text = text[m.end():]
  m = re.search(regexp_kennzeichen,text)
```

Mit der Funktion *sub* ist es außerdem möglich, alle (maximalen) Matchings eines regulären Ausdrucks in einem String durch einen String zu ersetzen:

Bsp.:

```
re.sub('(bc)+','x','abcbcabcbcbc') ⇝ 'axax'
```

Beachte, dass hierbei das Ergebnis nicht wieder gematcht wird:

```
re.sub('(bc)+', 'bc', 'abcbcabcbcbc') ⇝ 'abcabc'
```

Möchte man nur eine bestimmte Anzahl von Vorkommen ersetzen, kann man mit einem zusätzlichen Parameter die maximale Anzahl von Ersetzungen angeben:

```
re.sub('(bc)+','x','abcbcabcbcbc',1) ⇝ 'axabcbcbc'
```

Die Funktion sub ist nicht mutierend. Der übergene String wird nicht verändert (wie es in Python bei Strings ja nie der Fall ist). Stattdessen wir ein neuer String zurückgegeben.

Reguläre Ausdrücke werden z.B. auch in Textverarbeitungssystemen eingesetzt. In der Vorlesung schauen wir uns hierzu den Suchen und Ersetzen Mechanismus von Atom genauer an.

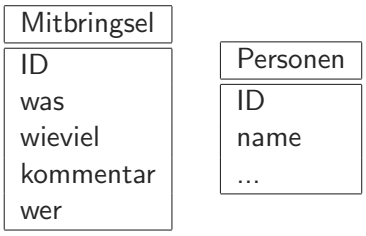
4 Anwendungen

4.1 Datenbanken

Bisher haben wir die Strukturierung von Daten innerhalb einer Programmiersprache kennen gelernt. Unsere letzte Anwendung hat einen Eindruck gegeben, wie man dies mit Hilfe von Tabellen (csv) machen kann. Für komplexere Anwendungen reichen einzelne Tabellen aber nicht aus. Datenbanken bieten einem die Möglichkeit mehrere Tabellen zu kombinieren und effizient auf die gespeicherten Daten zuzugreifen. Ausgehend von unserer csv-Anwendung wollen wir uns das Thema Datenbanken erarbeiten. Hierbei wollen wir nun die (vereinfachte) Datenbank einer Mitbring-Anwendung, wie sie z.B. auf einem entsprechenden Server im Internet vorgehalten wird modellieren.

In der Mitbringlisten-Anwendung mit csv-Dateien, haben wir die einzelnen Informationen zu einem Mitbringsel durch Strings identifiziert. Zu einer Person können wir uns aber auch vorstellen, dass es weitere Inforamtionen, wie seinen Namen, eine Telefonnummer und ggf. ein Bild und Status gibt. Diese Werte möchten wir natürlich nicht bei jedem einzelnen Mitbringsel wieder mit abspeichern. Außerdem wird es dann schwierig, diese Daten zu ändern. Man müsste alle diese Spalten bei allen Vorkommen der Person ändern. Deshalb legt man für die Personen der Mitbring-Anwendung eine separate Tabelle an, in welcher man die speziellen Attribute der Personen speichert und dann entsprechend nur hier ändern muss.

Bei der Angabe beim Mitbringsel, können wir dann einfach auf die entsprechenden Personen verweisen. Hierzu müssen diese natürlich eindeutig identifiziert werden könnnen, wozu man in der Datenbank einen speziellen Identifier (ID) verwendet. Der Name eignet sich in der Regel nicht als Schlüssel, da er nicht eindeutig ist. Bei Studierenden könnten wir die Matriklenummer verwenden. Bei beliebigen Personen eignet sich aber besser eine künstliche ID, welche nur innerhalb der Datenbank verwendet wird. Diese Ids sind meist Integer-Werte, welche den Datensatz eindeutig identifizieren. Solche eindeutigen Identifier einer Tabelle nennt man auch *P*rimärschlüssel. Primärschlüssel müssen eindeutig sein und keine zwei Spalten dürfen denselben Primärschlüssel besitzen.



Konkrete Tabellen könnten dann wie folgt aussehen:

Mitbringsel:

ID	was	wieviel	kommentar	wer
1	Brot	2	eins mit Zwiebeln, eins ohne	2
2	Rotwein	3 Flaschen	extra vegan	3
3	Reissalat	große Schüssel	vegan, glutenfrei	1
4	Frikadellen	17	mit Schweinefleisch	2

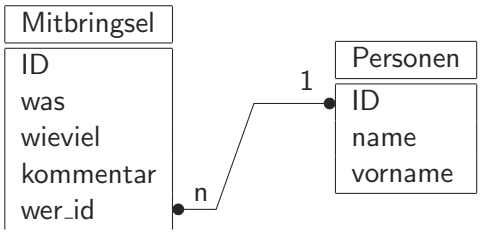
Personen:

ID	name	vorname
1	Huch	Frank
2	Smyth	Steven
3	Esquivel	Sandro

Die IDs bei den Mitbringseln sind im Moment noch nicht notwendig, schaden aber auch nicht.

Beachte, dass wir die IDs nur eindeutig für die jeweilige Tabelle verwenden. Dies reicht aus, da wir in der wer-Spalte der Mitbringsel-Tabelle nur Personen-IDs verwenden. Es ist klar, zu welcher Tabelle diese ID gehört.

Im allgemeinen besteht eine Datenbank, ähnlich wie eine Tabellenkalkulation, aus einer Reihe von Tabellen, welche über IDs zusätzlich Verweise auf anderen Tabelleneinträge enthalten. Zur Modellierung solcher Datenbanken vewendet man gerne Datenbank-Schemata, in welchen Tabellen und Beziehungen zwischen den Tabellen graphisch ansprechend dargestellt werden können. Für unsere bisherige Anwendung ergibt sich das folgende Datenbank-Schema:

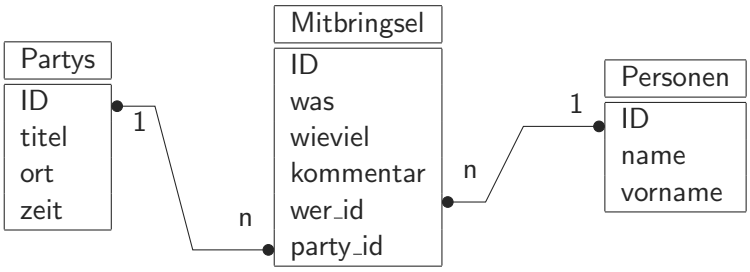


Jede Zeile entspricht einem Eintrag in die Tabelle. Die Reihenfolge ist eigentlich irrelevant, wichtig ist nur, dass wir die Zeilen unterscheiden können[1]. Hierfür eignen sich Namen in der Regel nicht, da es durchaus Menschen mit gleichen Namen gibt. Als Unteschеidungsmerkmal führt man deshalb in der Regel *Schlüssel* ein, welche es uns ermöglichen unserer Datensätze in jedem Fall zu unterscheiden. Solche Schlüssel kennen wir auch aus anderen Bereichen, wie z.B. die Matrikelnummer oder unsere Personalausweisnummer. Hier haben wir der Einfachheithalber eine Zahl hinzugefügt, welche nicht doppelt verwendet werden darf. Dass einzelne Zahlen fehlen stellt kein Problem dar und kann z.B. dadurch entstanden sein, dass eine Person, welche zuvor mit dem Schlüssel 2 existierte gelöscht wurde. Solche IDs werden von Datenbanksystemen in der Regel automatisch generiert. In unserer csv-Anwendung, mussten wir uns hierum noch selber kümmern.

Unter Verwendung des Primärschlüssels id für eine Person, können wir in der Mitbringseltabelle mit der Spalte wer_id einen Verweis auf die Personen-Tabelle anlegen. Wird ein Schlüssel in einer anderen Tabelle als verweis verwendet, nennt man diesen Schlüssel an dieser Stelle eine *Fremdschlüssel*. Für die Tabelle Mitbringsel ist dieser Fremdschlüssel kein Schlüssel, da er in mehreren Mitbringseln vorkommen kann. Es handelt sich also um eine 1:n-Bezihung und wir notieren sie im Datenbank-Schema mit Hilfe der eingefügten Kanten.

[1] In unserer bishrigen Implementierung mit csv-Dateien, spielt die Reihenfolge natürlich sehr wohl eine Rolle. Beim Übergang zu Datenbanken wird sie aber unwichtig werden.

Um unsere Anwendung zu vervollständigen, ist es sinnvoll auch noch eine Tabelle für die Partys, die verwaltet werden anzulegen. Als Atribute könnten hier ein Titel, ein Ort und eine Zeit sinnvoll sein. In unserer csv-Anwendung haben wir diese Tabelle dadurch realisiert, dass wir Mitbringlisten in unterschiedlichen Dateien gespeichert haben. Dies geht in einer Datenbank nur über eine weitere Tabelle. Auch zwischen der Tabelle Party und der Mitbringseltabelle besteht eine 1:n-Beziehung, welche wir wieder durch ein Attribut party_id in der Mitbringsel-Tabelle realisieren können.



Im folgenden werden wir ein einfaches Datenbanksystem kennenlernen. Wir verwenden SQLite (genauer sqlite3) und realisieren als Beispiel dieses Datenbankschema. Gesteuert werden Datenbanken in der Regel durch *Structured Query Language (SQL)*. Die wichtigsten Konstrukte wollen wir schrittweise in SQLite kennen lernen.

Zunächst starten wir SQLite, wobei wir als Parameter den Namen der Datenbank angeben, mit der wir arbeiten wollen, mittels `sqlite3 chat`. Danach können wir interaktiv mit der Datenbank arbeiten und zunächst eine Tabelle für Personen anlegen:

```
sqlite> CREATE TABLE personen (id INTEGER PRIMARY KEY,
                               name TEXT, vorname TEXT);
```

Die Schlüsselworte schreibt man in SQL in der Regel mit Großbuchstaben. SQLite erlaubt hier aber auch Kleinbuchstaben.

Mit diesem ersten SQL-Befehl generieren wir eine Tabelle mit dem Namen personen, welche drei Spalten (id, name und vorname) hat. Als id verwedet man in der Regel Zahlen, was wir mit dem Typ INTEGER für die erste Spalte festlegen. Außerdem legen wir fest, dass die id ein Primärschlüssel ist und vom Datenbanksystem automatisch inkrementiert wird. In die beiden weiteren Spalten können Strings (Typ TEXT) geschrieben werden.

Nun können wir einen ersten Datensatz in diese Tabelle eintragen:

```
sqlite> INSERT INTO personen (id, name, vorname) VALUES
        (1,'Huch', 'Frank');
```

Mit Hilfe der Spaltenbeschriftung zwischen dem Tabellennamen und dem Schlüsselwort VALUES, können wir die Reihenfolge der Werte bestimmen, welche wir eintragen wollen. Außerdem können wir einzelne Spalten leer lassen, was dazu führt, dass entweder Default-Werte oder NULL-Werte eingetragen werden. Werden alle Werte in der Reihenfolge, wie sie beim Anlegen der Tabelle definiert wurden, eingetragen, kann des Tupel mit den Spaltennamen auch weggelassen werden:

```
sqlite> INSERT INTO personen VALUES (2,'Smyth', 'Steven');
sqlite> INSERT INTO personen VALUES (3,'Esquivel', 'Sandro');
```

Versucht man einen Wert mit einem bereits vergebenen Schlüssel einzutragen, führt dies auf Grund der Primärschlüssel-Eigenschaft des id-Attributes zu einer Fehlermeldung:

```
sqlite> INSERT INTO personen (id, name, vorname) VALUES
        (2,'Merkel', 'Angela');
Error: UNIQUE constraint failed: personen.id
```

Auf Grund der Angabe PRIMARY KEY bei der Definition der id-Spalte, haben wir aber auch die Möglichkeit, das System eine geeignete id generieren zu lassen, wenn wir keinen passenden Wert angeben:

```
sqlite> INSERT INTO personen (name, vorname) VALUES
        ('Merkel', 'Angela');
```

Nun können wir unsere erste Anfrage an die Datenbank stellen und alle Spalten der Tabelle personen ausgeben:

```
sqlite> SELECT * FROM personen;
1|Huch|Frank
2|Smyth|Steven
3|Esquivel|Sandro
4|Merkel|Angela
```

Die Verwendung des * bedeutet hierbei, dass alle Spalten in der Reihenfolge, wie sie bei der Generierung der Tabelle angegeben wurden, ausgegeben werden. Wir können uns aber auch auf die Ausgabe einzelner Spalten einschränken und auch deren Reihenfolge verändern:

```
sqlite> SELECT vorname,name FROM personen;
Huch|Frank
Smyth|Steven
Esquivel|Sandro
Merkel|Angela
```

Nun können wir eine zweite Tabelle für die Partys anlegen:

```
sqlite> CREATE TABLE partys
            (id INTEGER PRIMARY KEY AUTOINCREMENT,
             titel TEXT, ort TEXT, zeit DATETIME);
sqlite> INSERT INTO partys (titel, ort, zeit)
        VALUES ('Weihnachtsfeier','Ú2','2017-12-14 18:00:00');
sqlite> SELECT * FROM partys;
1|Weihnachtsfeier|Ú2|2017-12-14 18:00:00
```

Hier haben wir zusätzlich noch angegeben, dass das Datenbanksystem den Primärschlüssel id automatisch hochzählen soll und hierbei, immer einen neuen Schlüssel generiert, der größer als alle vorher generierten Schlüssel ist. Da dies Datenbankspezifisch ist, gehen wir hierauf nicht näher ein.

Entsprechend lagen wir auch eine Tabelle für die Mitbringsel an:

```
sqlite> CREATE TABLE mitbringsel
            (id INTEGER PRIMARY KEY AUTOINCREMENT,
             was TEXT, wie_viel TEXT, kommentar TEXT,
             wer_id INTEGER, party_id INTEGER);
sqlite> INSERT INTO mitbringsel (was, wie_viel, kommentar,
                                 wer_id, party_id)
        VALUES ('Brot','2','eins mit Zwiebeln, eins ohne',2,1)
```

```
sqlite> INSERT INTO mitbringsel (was, wie_viel, kommentar,
                                 wer_id, party_id)
        VALUES ('Rotwein','3 Flaschen','extra vegan',3,1)
sqlite> INSERT INTO mitbringsel (was, wie_viel, kommentar,
                                 wer_id, party_id)
        VALUES ('Reissalat','ganz viel','vegan, glutenfrei',1,1)
sqlite> INSERT INTO mitbringsel (was, wie_viel, kommentar,
                                 wer_id, party_id)
        VALUES ('Frikadellen','17','mit Schweinefleisch',2,1)
sqlite> SELECT * FROM mitbringsel
1|Brot|2|eins mit Zwiebeln, eins ohne|2|1
2|Rotwein|3 Flaschen|extra vegan|3|1
3|Reissalat|ganz viel|vegan, glutenfrei|1|1
4|Frikadellen|17|mit Schweinefleisch|2|1
```

Im nächsten Schritt wollen wir nur die Mitbringsel der Person mit der id 2 ausgeben. Hierzu ist es möglich in der SELECT-Anweisung mittels einer WHERE-Klausel Einschränkungen zu definieren:

```
sqlite> SELECT was, wie_viel,kommentar FROM mitbringsel
        WHERE wer_id=2;
Brot|2|eins mit Zwiebeln, eins ohne
Frikadellen|17|mit Schweinefleisch
```

In der WHERE-Klausel können auch Ungleichungen (!=,<,>,<=,>=), sowie boolesche Kombinationen dieser (AND bzw. OR als Infixoperatoren) verwendet werden.

In einer anderen Anfrage möchten wir für die Mitbringsel auch noch die Person ausgeben, die etwas mitgebracht hat. Hierzu wählen wir zunächst nicht nur Spalten aus einer Tabelle, sondern direkt aus zwei Tabellen aus:

```
sqlite> SELECT * FROM mitbringsel, personen;
1|Brot|2|eins mit Zwiebeln, eins ohne|2|1|1|Huch|Frank
1|Brot|2|eins mit Zwiebeln, eins ohne|2|1|2|Smyth|Steven
1|Brot|2|eins mit Zwiebeln, eins ohne|2|1|3|Esquivel|Sandro
1|Brot|2|eins mit Zwiebeln, eins ohne|2|1|4|Merkel|Angela
2|Rotwein|3 Flaschen|extra vegan|3|1|1|Huch|Frank
2|Rotwein|3 Flaschen|extra vegan|3|1|2|Smyth|Steven
2|Rotwein|3 Flaschen|extra vegan|3|1|3|Esquivel|Sandro
2|Rotwein|3 Flaschen|extra vegan|3|1|4|Merkel|Angela
3|Reissalat|ganz viel|vegan, glutenfrei|1|1|1|Huch|Frank
3|Reissalat|ganz viel|vegan, glutenfrei|1|1|2|Smyth|Steven
3|Reissalat|ganz viel|vegan, glutenfrei|1|1|3|Esquivel|Sandro
3|Reissalat|ganz viel|vegan, glutenfrei|1|1|4|Merkel|Angela
4|Frikadellen|17|mit Schweinefleisch|2|1|1|Huch|Frank
4|Frikadellen|17|mit Schweinefleisch|2|1|2|Smyth|Steven
4|Frikadellen|17|mit Schweinefleisch|2|1|3|Esquivel|Sandro
4|Frikadellen|17|mit Schweinefleisch|2|1|4|Merkel|Angela
```

Wir erhalten also alle möglichen Kombinationen von Einträgen der einen und der anderen Tabelle (kartesisches Produkt der beiden Tabellen). Diese entstehende Tabelle können wir nun auf die Einträge einschränken, bei denen die empfaenger_id mit der id der Tabelle personen übereinstimmt:

```
sqlite> SELECT * FROM mitbringsel, personen WHERE wer_id == personen.id;
1|Brot|2|eins mit Zwiebeln, eins ohne|2|1|2|Smyth|Steven
2|Rotwein|3 Flaschen|extra vegan|3|1|3|Esquivel|Sandro
3|Reissalat|ganz viel|vegan, glutenfrei|1|1|1|Huch|Frank
4|Frikadellen|17|mit Schweinefleisch|2|1|2|Smyth|Steven
```

Schränken wir uns auf die interessanten Spalten ein und ordnen sie zusätzlich nach dem Namen des Mitbringenden, erhalten wir:

```
sqlite> SELECT was,wie_viel,kommentar,name,vorname
        FROM mitbringsel, personen
        WHERE wer_id == personen.id
        ORDER BY name;
Rotwein|3 Flaschen|extra vegan|Esquivel|Sandro
Reissalat|ganz viel|vegan, glutenfrei|Huch|Frank
Brot|2|eins mit Zwiebeln, eins ohne|Smyth|Steven
Frikadellen|17|mit Schweinefleisch|Smyth|Steven
```

Ein Nachteil des Selektierens aus zwei Tabellen erkennen wir an der Anfrage

```
SELECT * FROM mitbringsel, personen;
```

Es werden jeweils alle Einträge beider Tabellen mit einander kombiniert, was zu einer sehr großen Zwischentabelle führt und bei großen Tabellen entsprechend ineffizient werden kann. Eine Alternative stellen sogenannte Joins dar, bei denen zwei Tabellen gemäß eines Kriteriums kombiniert werden können. In unserem Beispiel kann eine effizientere Abfrage wie folgt aussehen:

```
sqlite> SELECT was,wie_viel,kommentar,name,vorname
        FROM mitbringsel INNER JOIN personen
        ON wer_id == personen.id
        ORDER BY name;
```

In vielen Fällen ist diese Optimierung aber nicht notwendig und kann meist automatisch vom Datenbanksystem aus der ersten Anfrage mit dem kartesischen Produkt hergeleitet werden. Allerdings sind viele SQL-Programmierer der Meinung, dass die Formulierung mit Hilfe eines Joins klarer ist. Deshalb werden viele Anfragen auch mit Hilfe von Joins definiert und dieses Konstrukt soll hier der Vollständigkeit halber präsentiert werden.

Als letztes SQL-Konstrukt lernen wir noch die DELETE-Anweisung kennen, mit welcher wir auch Tabelleneinträge löschen können. Als Beispiel wollen wir Frau Merkel aus der Personen-Tabelle löschen:

```
DELETE FROM personen WHERE name='Merkel';
```

Danach kommt Frau Merkel zwar nicht mehr in der Personen-Tabelle vor, aber es kann natürlich noch Mitbringsel geben, welche auf sie verweisen. Diese können wir aber genauso einfach löschen:

```
DELETE FROM mitbringsel WHERE wer_id=4;
```

Um mit Datenbanken auch praktisch arbeiten zu können, verwenden wir eine SQLite3-Anbindungung, welche als Bibliothek verfügbar ist. Nähere Infos hierzu finden sich auf der iLearn-Seite zur Vorlesung.

Die Bibliothek bindet man mittels

```
import sqlite3    # Anbindungsmodul für sqlite3.
```

in sein Python Programm ein. Danach 'öffnet' man eine spezielle Datenbank mittels

```
db = sqlite3.connect('mitbringliste.sq3')
```

und erhält einen sogenannten Handle (was man sich, wie einen Zeiger vorstellen kann) auf die Datenbank. Unter Verwendung dieses Handels, welchen wir uns in der Variablen db gemerkt haben, können wir nun einen sogenennten Cursor erstellen, über welchen einzelne SQL-Befehle ausgeführt werden können und in welchem das Ergebnis eines 'SELECT'-Befehls gespeichert wird:

```
cur = db.cursor()
```

Wir speichern den Cursor in der Variablen cur ab und können nun SQL-Befehle mit der execute-Methode als String an die Datenbank schicken:

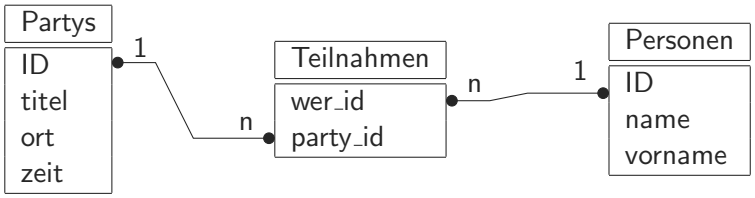
```
cur.execute('CREATE TABLE personen '+
            '(id INTEGER PRIMARY KEY AUTOINCREMENT, '+
            'name TEXT, vorname TEXT)')
```

Liefert ein SQL-Befehl (wie z.B. der SELECT-Befehl) eine Tabelle als Ergebnis, wird diese im Cursor-Objekt gespeichert. Mit Hilfe der Methode fetchall der Cursor-Klasse können wir die Tabelle in Form einer zweidimensionalen Listen darstellen. Die Verwendung sollte mit dem folgenden Beispielprogramm klar werden:

```
import sqlite3

# Datenbankverbindung erstellen
db = sqlite3.connect("mitbringliste.sq3")

# Cursor für Datenbankzugriff erstellen
cur = db.cursor()

# SQL-Befehle ausführen
cur.execute('CREATE TABLE personen '+
            '(id INTEGER PRIMARY KEY AUTOINCREMENT, '+
            'name TEXT, vorname TEXT)')
...
cur.execute('INSERT INTO personen (id, name, vorname) '+
            'VALUES (1, 'Huch', 'Frank')')
...
cur.execute('SELECT * FROM personen')

# Ergebnis des SELECT-Befehls ausgeben
personen = cur.fetchall()
print(personen)  # Gibt die Tabelle der aktuellen
                 # Personen als Liste aus
```

In der Vorlesung werden wir die Mitbringlisten-Anwendung von csv auf SQL umstellen. Hierbei ist es notwendig, die relevanten Benutzereingaben in die SQL-Anfragen zu integrieren. Das bedeutet, dass die SQL-Anfragen dynamisch zusammengesetzt werden und als Teilstrings Benutzereingaben enthalten.

Abschließend können wir noch einmal einen anderen Blick auf die von uns designte Datenbank werfen. Wir hätten auch mit den beiden Tabellen Mitbringsel und Personen beginnen können und uns überlegen können, dass z.B. die Teilnahme an einer Party auch eine Relation zwischen den beiden Tabellen darstellt. Da eine Person zu mehreren Partys gehen kann und zu einer Party auch mehrere Personen kommen können, handelt es sich nicht um eine 1:n-Beziehung, sondern um eine n:m-Beziehung. Datenbanken erlauben in der Regel nicht die direkte Realisierung von n:m-Beziehungen und man realisiert solch eine Beziehung im Datenbankentwurf durch eine Hilfstabelle, welche z.B. Teilnahmen heißen könnte und die Teilnahme einer Person an einer Party darstellt:



In der Tabelle Teilnahmen verwenden wir nun zwei Fremschlüssel und realisieren so die n:m-Beziehung mit Hilfe von zwei 1:n-Beziehungen. Der Tabelle Teilnahmen geben wir in diesem Fall keinen Schlüssel separaten Primärschlüssel, sondern können die beiden Fremsdschlüssel als soganannten *Verbundschlüssel* verwenden, was bedeutet, dass die Kombination der beiden Fremdschlüssel eine Schlüssel darstellen muss, also einen Eintrag in der Tabelle eindeutig identifieziert. Mit anderen Worten darf es keine zwei Einträge mit derselben wer_id und party_id geben. Dies ist aber sinnvoll, da man ja auch nicht zweimal zur selben Party gehen kann. In SQLITE kann ein solcher Verbundschlüssel wie folgt angelegt werden:

```
CREATE TABLE teilnahmen (wer_id, party_id,
                         PRIMARY KEY (wer_id, party_id))
```

Versuchen wir dann zweimal dieselbe Teilnahme einzutragen erhalten wir folgenden Fehler:

```
sqlite> INSERT INTO teilnahmen VALUES (1,2)
sqlite> INSERT INTO teilnahmen VALUES (1,2)
Error: UNIQUE constraint failed: teilnahmen.wer_id, teilnahmen.party_id
```

Vergleicht man nun die Tabelle Teilnahme mit der Tabelle Mitbringsel von oben, so erkennt man sehr viele Gemeinsamkeiten und auch das Mitbringen eines Mitbringsels ist eigentlich eine n:m-Beziehung zwischen den beiden Tabellen Partys und Personen, der wir allerdings noch weitere Spalten hinzugefügt haben. Im Gegensatz zur Teilnehme ist es bei den Mitbringseln aber sinnvoll einen separaten Primärschlüssel zu verwenden, da man so eben auch modellieren kann, dass eine Person mehrere Dinge zu derselben Party mitbringt.

In unterschiedlichen SQL-Dialekten (auch in SQLITE) ist es möglich, die Fremdschlüsselbeziehungen bei der Anlage einer neuen Tabelle auch explizit zu spezifizieren. Das Datenbanksystem überprüft bei Manipulationen der Datenbank dann später, ob diese Beziehungen auch eingehalten werden. Im Rahmen dieser Vorlesung wollen wir hierauf aber nicht weiter eingehen. Der interessierte Teilnehmer kann weitere Inforamtionen hierzu z.B. in der Dokumentation von SQLITE im Internet finden. In manchen Fällen können solche Konsitenzüberprüfungen auch unpraktisch sein, z.B. wenn man mehrere Einträge (mit Schlüssel und zugehörigem Fremdschlüssel) schrittweise zur Datenbank hinzufügen oder löschen möchte.

4.2 HTML

Bevor wir uns weitere mit der Programmierung und Python auseinandersetzen, machen wir einen kleinen Exkurs in die Welt des Internets.

HTML ist eigentlich die Sprache des WWWs.

Sie bietet einfache Strukturierungsmöglichkeiten für Textdokumente, sowie die Einbindung von Bildern zur Textgestaltung.

4.2.1 Genereller Aufbau von HTML-Seiten

Der generelle Aufbau von HTML-Seiten:

```
<! DOCTYPE HTML PUBLIC ''-//W3C//DTD HTML 4.01 Transitional//EN''>
```

↑ HTML-Typ

```
<html>
```

Tags, zu denen meist passende schließende Tags gehören (Groß-Kleinschreibung egal)

```
  <head>
    <title> Hello </title>
  </head>
```

Kopf, in der Regel hier nur Titel der Seite ⇝ Fensterleiste

```
  <body>
    ... eigentliches Dokument ...
  </body>
</html>
```

Rumpf

Ein paar Strukturierungselemente:

<h1>, <h2>, <h3> für Überschriften (werden größer/fetter gesetzt)

Hier kommt ein Zeilenumbruch
danach geht es hier weiter.

Sonstige Zeilenumbrüche oder übermäßige Whitespaces haben keine Strukturierungseffekte, sie werden als ein Leerzeichen dargestellt.

Mit Was ich schon immer mal sagen wollte wird ein Paragraph definiert, welcher im Text entsprechend abgesetzt wird.

Viele HTML-Tags können mit zusätzlichen Attributen versehen werden, die zusätzliche Eigenschaften festlegen.

Bsp.: <h1 align='center'> Überschrift </h1>

(weitere Attribute werden noch folgen)

4.2.2 Sonderzeichen

Sonderzeichen, wie z.B. Umlaute, können z.T. nicht richtig dargestellt werden (browserabhängig). Sicher ist die Verwendung von

&auml	statt ä
&uuml	statt ü
&ouml	statt ö
&Auml	statt 'A
⋮	
&szlig	statt ß

4.2.3 Listen

Aufzählungslisten:

```
<ul>
  <li> Item 1 </li>
  <li> Item 2 </li>
  <li> <ul>
         <li> Item 3a </li>
         <li> Item 3b </li>
       </ul>
</ul>
```

 statt <ul> liefert nummerierte Listen.

Attribute können Aufzählungsmarkierung verändern:

Bsp.:

```
<ul type=square'>
```

⇝ □ statt ·

```
<ol type='I' start='3'>
```

⇝ III ...

4.2.4 Links

Als wichtige Strukturierungsmöglichkeit bietet HTML die Möglichkeit unterschiedliche Dokumente (auch Bilder) miteinander zu verlinken.

Link: Hier finden Sie Infos zu Python . ⇝ Hier finden Sie Infos zu Python. Wird keine komplette URL angegeben, so wird die Datei relativ zu aufrufenden Datei gesucht.

Mit Attribut target kann noch das Ziel definiert werden, z.B. target=_blank ⇝ in neuem Fenster.

4.2.5 Grafiken

Mit <img src='bild.jpg'> können Bilder eingebunden werden. Viele verschiedene Bildformate sind möglich. Wieder können URLs oder lokale Adressen für die Bilder angegeben werden.

Grafiken können auch in Text eingebaut werden, sind aber meist größer als Buchstaben:

abc  def

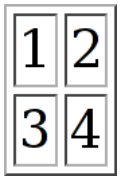
Mittels align='right' oder align='left' kann Text um die Bilder 'herumfließen' (Bilder werden am Seitenrand dargestellt).

Bilder können auch als Links verwendet werden:
<img src='www.Python-lang.org/images/logo.gif'>

4.2.6 Tabellen

Als weiteres wichtiges Strukturierungsmittel können Tabellen eingesetzt werden:

```
<table border='1'>
  <tr>
    <td> 1 </td>
    <td> 2 </td>
  </tr>
  <tr>
    <td> 3 </td>
    <td> 4 </td>
  </tr>
</table>
```



Hierbei wird innerhalb der tr-Tags eine Zeile und innerhalb eines td-Tags eine Spalter innerhalb einer Zeile definiert.

border=0 (default) liefert keinen Rand.

Einige Eigenschaften können durch Attribute verändert werden. Mehr ⇝ `de.selfhtml.org`

Wie können wir nun unsere Python/HTML-Kenntnisse sinnvoll kombinieren?

Wir können Python dazu einsetzen, HTML-Dokumente zu generieren. Wir betrachten wieder das Beispiel der Multiplikationstabelle.

```
h = File.open('multi.html','w')

h.print('<!DOCTYPE html PUBLIC \'-//W3C//DTD HTML 4.01 Transitional//EN\'>

h.print('<html>')
h.print('<head><title>Multiplikationstabelle</title></head>')
h.print('<body>')
h.print('<table>')
for i in 1..10:
  h.print('<tr>')
  for j in 1..10:
    h.print('<td>')
    h.print((i*j).to_s)
    h.print('</td>')
  end
  h.print('</tr>')
end
```

```
h.print('</table>')
h.print('</body>')
h.print('</html>')

h.close
```

Schöner wäre die Lösung sicherlich unter Wiederverwendung der Multiplikationstabelle aus dem csv-Beispiel. Die Umwandlung beliebiger zweidimensionaler Tabellen in HTML-Tabellen kann als Übung definiert werden.

4.3 Internet und WWW

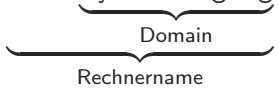
Das Internet ist ein weltweites Netzwerk zum elektronischen Austausch von Daten zwischen Rechnern. Hierbei werden die unterschiedlichsten Dienste angeboten, wie Email, WWW, Internettelefonie, Livestreams, usw.

Die Knoten des Internets werden über IP-Adressen identifiziert (IPv4), z.B.:

221.186.184.68 (Server: www.Python-lang.org)

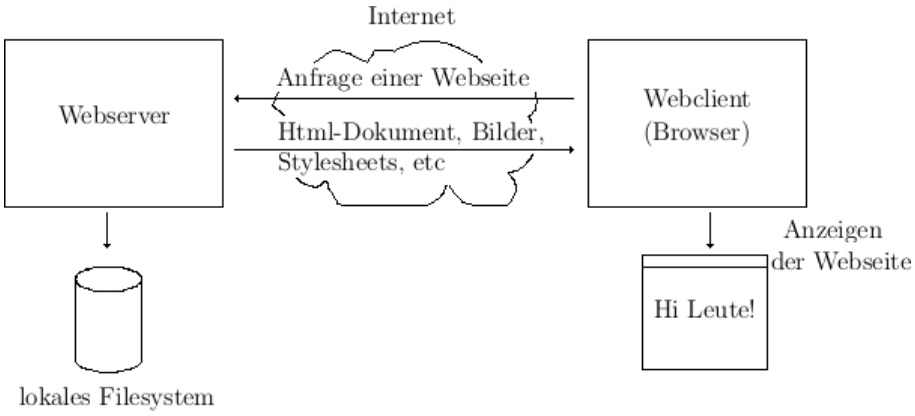
Nachrichten zwischen diesen Rechnern werden über das Internet Protocol (IP) versandt. IP-Pakete sind Pakete, die bei allen Internetanwendungen verwendet werden. Hierauf bauen weitere Datenaustauschprotokolle auf, welche zusätzliche Eigenschaften zusichern. Z.B. das Transmission Control Protocol (TCP), welches 'Verbindungen' und verlustfreie Datenübertragung ermöglicht.

Ein weiterer Dienst ist das Domain Name System (DNS), welches die Verwendung von Namen statt IP-Adressen ermöglicht, z.B. www.Python-lang.org. Lokal gesehn wird of auch



nur www als Rechnername bezeichnet. Zur eindeutigen Identifikation im Internet benötigt aber jeder Rechner auch noch eine Domain.

Das WWW ist ein spezieller Dienst zum Dokumentenaustausch im Internet. Hierbei werden die Dokumente mit Hilfe der HyperText Markup Language (HTML) strukturiert. Das zugehörige Protokoll ist das HyperText Transfer Protocol (HTTP):



Hierbei werden durch den Web-Server die Uniform Resource Locator (URL) auf das lokale Filesystem auf dem Web-Server umgebogen.

Bsp.: Der Webserver des Instituts für Informatik stellt die Dateien im public_html-Verzeichnis der User (z.B. fhu) unter www.informatik.uni-kiel.de/∼ fhu/... zur Verfügung (wichtig: Leserechte auch für das Home-Verzeichnis für alle geben).

Wegen der Beliebtheit und Standardisierung des WWW mit seinen komfortablen Browsern werden zusehens auch andere Internetdienste über ein WWW-Interface verfügbar gemacht (z.B. Email).

5 Laufzeitanalyse und Algorithmen

5.1 Arrays

Bevor wir uns genauer mit der Laufzeitanalyse und dem Sortieren beschäftigen, schauen wir uns die möglichen Datenstrukturen in Python an, welche man zum Sortieren vieler Werte vewenden können.

Wir kennen bereits Listen, mit welchen wir beliebig viele Werte zusammen fassen kann. Sie sind insbesondere auch dynaisch in ihrer Größe,so dass man auch noch nachträglich WErte hinzunehmen kann. Diese Dynamik hat ihren Preis und führt dazu, dass es bei häufiger Verwendung dazu kommen kann, dass komplette Listen kopiert werden müssen. In einzelnen Fällen kann deshalb die Laufzeit des Hinzufügens eines Wertes nicht konstant sein. Bleibt aber die Länge der Liste unverändert, ist es möglich in konstanter Zeit auf ein Elemtment der Liste zuzugreifen und es ist ebefalls möglich ein Element in konstanter Zeit zu verändern. Ein vergleichbare Datenstruktur bieten viele Programmiersrpachen in Form eines Arrays (oder Feldes)an, bei denen es aber in der Regel nicht möglich ist, die Länge des Arrays zu verändern.

Auch in Python gibt es Arrays, welche es ermöglichen effizient mit großen Datenmengen zu arbeiten. Hierzu verwendet man die Array-Bibliothek NumPy, welche Arrays direkt als C-Datenstruktur repräsentiert und viele Operationen auf Arrays besonders effizient (in C implementiert) anbietet. Wir werden die relevanten Methoden von NumPy nach un nach einführen. Prinzipiell können alle Algorithmen zum Sortieren aber genau so auch mit Listen realisiert werden. Die implementierten Algorithmen unterscheiden sich nicht einmal und können wowohl Listen als auch NumPy-Arrays sortieren. In der Vorlesung vergleichen wir auch einmal die Laufzeiten von Listen und Arrays, allerdings mit einem ernüchternden Ergebnis: Verwendet man wenige vordefinierte Algorthmen, sind unsere Algorithmen auf Listen um den Faktor 2-3 schneller.

NumPy erlaubt es in Python Arrays (auch mehrdimensional) zu definieren. Ein Array unterscheidet sich nur wenig von Listen. Der wesentliche Unterschied ist, dass Arrays nicht vergrößert oder verkleinert werden können. Sie sind also weniger dynamisch. Der Zugriff auf einzelne Array-Komponenten ist hierdurch in konstanter Zeit möglich (bei Listen auch), aber ist sehr viel schneller. Kleine Arrays können auf Basis einer Liste konstruiert werden. Große Arrays können mit Hilfe der Funktion numpy.arange konstruiert werden:

```
import numpy

a = numpy.array([1,2,3,4,5])

print(a)    # liefert Ausgabe: [1 2 3 4 5]

a = numpy.arange(3,7)

print(a)    # liefert Ausgabe: [3 4 5 6]
```

```
a = numpy.zeros(5)

print(a)   # liefert Ausgabe: [0. 0. 0. 0. 0.]

a = numpy.zeros(5, dtype=int)

print(a)   # liefert Ausgabe: [0 0 0 0 0]
```

Im Gegensatz zu Listen verwendet Python bei Arrays ein Leerzeichen anstelle eines Kommas. Floatingpointzahlen ohne Nachkommastelle werden nur mit einem Punkt anstelle von .0 dargestellt.

Der Zugriff auf Arrays ist meist über dieselben Funktionen, wie bei Listen möglich. Slices können auch selektiert werden, aber nicht durch Arrays anderer Länge, als die Slice-Länge ersetzt werden.

Wenn wir uns im folgenden mit Sortieralgorithmen beschäftigen kann es sinnvoll sein, ein zufälliges Array einer gewissen Größe zu konstruieren:

```
import numpy
import random

def rand_array(size) :

  a = numpy.arange(size)

  for i in range(size) :
    a[i] = random.randrange(300000)

  return a


print(rand_array(5))  # unterschiedliche Ausgaben von Arrays der Länge 5 m
```

Entsprechend ist dies natürlich auch für Listen möglich:

```
def rand_list(size) :

  l = []

  for i in range(size) :
    l.append(random.randrange(300000))

  return l


print(rand_list(5))  # unterschiedliche Ausgaben von Arrays der Länge 5 mö
```

5.2 Sortieren

Ein interessantes Problem ist es eine große Anzahl von Werten zu sortieren. Funktionen bzw. Methoden zum Sortieren stellen alle gängigen Programmiersprachen zur Verfügung.

Dennoch ist es sinnvoll sich einige Sortieralgroithmen genauer an zu schauen und hierbei auch deren Laufzeitverhalten zu untersuchen.

5.2.1 Sortieren durch Auswählen

1. Lösung Minsort (oder Selectionsort)

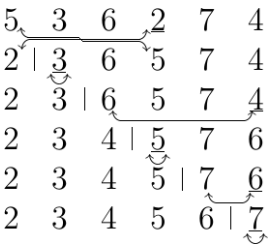
Idee: Zunächst wandeln wir einen Algorithmus zur Bestimmung des Minimums einer Liste so ab, dass nicht das Minimum, sondern seine Position zurückgeliefert wird. Zusätzlich wird das Minimum nur ab einer vorgegebenen Position gesucht:

```
def min_pos_from(l,pos) :
  min_pos = pos
  for i in (range(pos+1,len(l)) :
    if l[i] < l[min_pos] :
      min_pos = i

  return min_pos
```

Dann: min_pos_from ([2,7,5,4,8],1) $\rightsquigarrow 3$

Wie können wir hiermit jetzt sortieren?

Bsp.:



D.h. wir laufen von links nach rechts durch die Liste und vertauschen jeweils das aktuelle Element mit dem Minimum der Restliste. Als erstes ist es günstig eine Hilfsprozedur zum Vertauschen zweier Listenelemente zu definieren:

```
def swap(l,i,j) : # swap! verändert die Liste
  dummy = l[i]
  l[i] = l[j]
  l[j] = dummy

def min_sort(l) :
  for i in range(len(l)-1) : #da einelementige Liste immer sortiert
    pos = min_pos_from(l,i)
    swap(l,i,pos)

  return l
```

Dann: p(min_sort(numpy.array ([5,3,6,2,7,4]))) ⇝[2 3 4 5 6 7]

Wie soll man einen Sortieralgorithmus beurteilen?

Laufzeiten vergleichen! Aber: was sind gute Testfälle?

Zunächst scheint es sinnvoll, die Laufzeit in Abhängigkeit der Eingabegröße zu untersuchen. Es zeigt sich folgendes Verhalten:

Größe des Arrays	Laufzeit	
1000	0.308	
* 2 ↳		↴ * 4.16
2000	1.284	
* 2 ↳		↴ * 4.02
4000	5.156	
* 2 ↳		↴ * 3.82
8000	19.713	

Der Algorithmus scheint also quadratische Laufzeit in der Größe der Listen zu haben. Hierbei scheinen die konkreten Werte, welche in der Liste vorkommen, fast völlig unwichtig zu sein. Die Laufzeiten verändern sich nicht, ob man sortierte oder unsortierte Listen betrachtet.

Dies liegt daran, dass das Programm zwei verschachtelte Schleifen verwendet. Bei beiden Schleifen handelt es sich um for-Schleifen, so dass die Anzahl der Schleifendurchläufe nicht von den Werten abhängt.

```
for i in range(len(l)-1) :
  for j in range(i+1, len(l)) :
    ...
```

Als einziger Unterschied fallen ein paar Zuweisungen in minPosFrom bei einer sortierten Liste weg.

Welche Werte werden also für i und j durchlaufen?

i	j
0	1
⋮	2
	3
	⋮
	len(l)
1	2
⋮	3
	⋮
	len(l)
2	3
⋮	⋮
	len(l)
len(l)-3	len(l)-2
	len(l)
len(l)-2	len(l)

D.h. $(*)$ wird so oft durchlaufen:

$$\begin{aligned} len(l) + len(l) - 2 + len(l) - 3 + \ldots + 1 &= \sum_{n=1}^{len(l)} n \\ &= \frac{(len(l) - 1) \cdot len(l)}{2} \\ &= \frac{len(l)^2 - len(l)}{2} \\ &= \frac{1}{2} len(l)^2 - \frac{1}{2} len(l) \end{aligned}$$

D.h. bis auf ein paar $\frac{1}{2}len(l)$ viele Werte werden $\frac{1}{2}len(l)^2$ viele Werte durchlaufen. Die genaue Zahl der Durchläufe ist nicht so wichtig. Wichtiger ist, dass ihre Anzahl quadratisch mit der Listengröße wächst, also für doppelte Listengröße die 4-fache Laufzeit benötigt wird!

Geht sowas auch besser?

5.2.2 Sortieren durch Einfügen

Unschön an Min-Sort ist insbesondere, dass er auch für bereits sortierte Listen quadratische Laufzeit hat. Besser ist hier Insertion-Sort:

Idee: Sortiere wie einen Stapel Spielkarten. Ziehe Karte und füge diese in bereits sortierte Hand ein.

Bsp.:

5	3	6	2	7	4
5	|	3	6	2	7	4
3	5	|	6	2	7	4
3	5	6	|	2	7	4
3	5	2	|	6	7	4
3	2	5	|	6	7	4
2	3	5	|	6	7	4
2	3	5	6	|	7	4
2	3	5	6	7	|	4
2	3	5	6	4	|	7
2	3	5	4	6	|	7
2	3	4	5	6	|	7
2	3	4	5	6	7	|

Insertion-Sort können wir wie folgt in Python implementieren:

```
def ins_sort(l) :
  for i in range(len(l)) :
    j = i
    while j>0 and l[j-1] > l[j] :
      swap(l,j-1,j)                    # Wiederverwendung von swap
      j = j - 1

  return l
```

Dann: print(ins_sort (numpy.array ([5,3,6,2,7,4]))) $\rightsquigarrow$ [2 3 4 5 6 7]

Beachte: Bei sortierten Feldern wird fast nichts gemacht:

```
1 | 2   3   4   5
1   2 | 3   4   5
1   2   3 | 4   5
1   2   3   4 | 5
1   2   3   4   5 |
```

Der Algorithmus hat also im besten Fall lineare Laufzeit in der Größe der zu sortierenden Liste.

Aber warum ist die absolute Laufzeit schlechter als bei Min-Sort? Das Problem ist, dass wir die Elemente so lange tauschen bis wir ein Element an der richtiger Stelle eingefügt habe. Hierdurch ist die Operation, welche im Schleifenrumpf ausgeführt wird aufwendiger als bei Min-Sort.

D.h. bis auf ein paar $\frac{1}{2}len(l)$ viele werden $\frac{1}{2}len(l)^2$ viele Werte durchlaufen und dabei getauscht. Dies kann dadurch optimiert werden, dass man sich das einzufügende Element merkt und die anderen nach hinten schiebt.

```
2  4  6  8  10 12 |  5
2  4  6  8  10 12    12    (5)
2  4  6  8  10 10    12    (5)
2  4  6  8  8  10    12    (5)
2  4  6  6  8  10    12    (5)
2  4  5  6  8  10    12 |
```

In der Implementierung:

```
def ins_sort(l) :
  for i in range(len(l)) :
    j = i
    ins = l[i]
    while j>0 and ins < l[j-1] :
      l[j] = l[j-1]
      j = j - 1

    l[j] = ins

  return l
```

Diese Implementierung ist jetzt ungefähr genau so schnell wie Selection-Sort für unsortierte Listen und viel schneller für sortierte Listen.

Wir erhalten folgende Komplexitäten:

	Selection-Sort	Insertion-Sort
Best-Case (sortierte Liste)	quadratisch in Größe der Liste	linear in Größe der Liste
Worst-Case (unsortiere Liste)	quadratisch in Größe der Liste	quadratisch in Größe der Liste

5.3 Die Fibonacci-Funktion

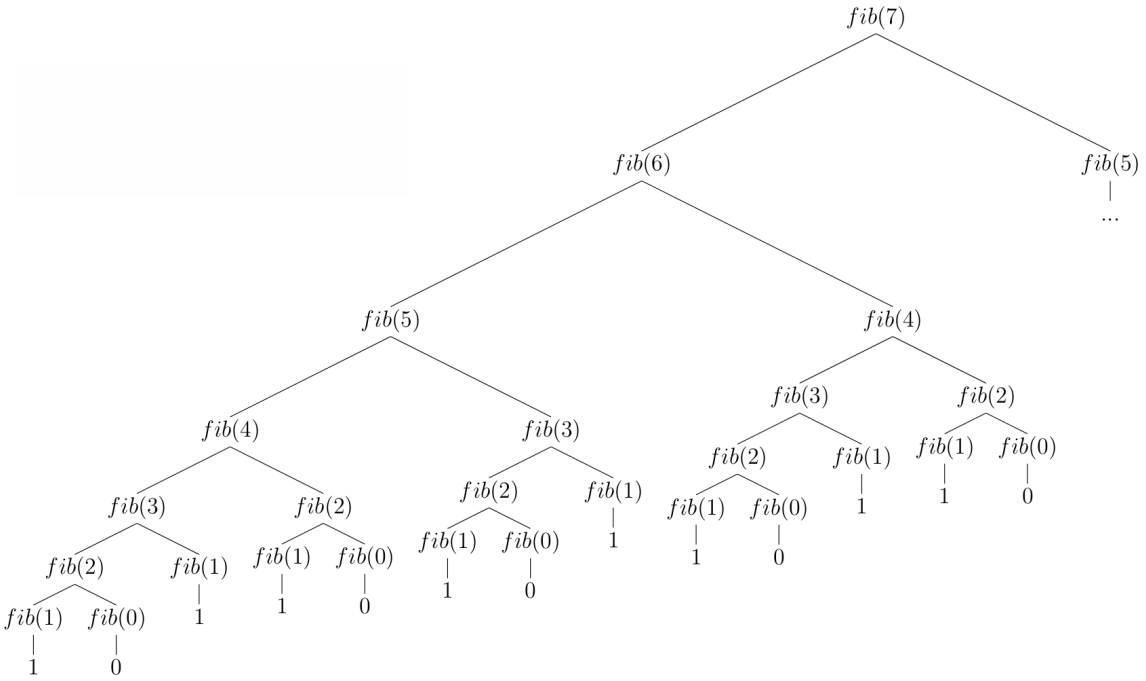
Im Folgenden werden wir unterschiedliche Algorithmen bzgl. ihres Laufzeitverhaltens analysieren.

Zunächst betrachten wir noch einmal die rekursive Implementierung der Fibonacci-Funktion:

```
def fib(n) :
  if n == 0 :
     return 0
  else if n == 1 :
     return 1
  else :
     return fib(n-1) + fib(n-2)
```

Praktische Tests: Oha!

Woran liegt dies?



Beachte: In jeder Rekursion werden zwei rekursive Aufrufe gemacht. Hierdurch ergibt sich eine **exponentielle Laufzeit**.

Wie schlimm ist dies?

$$
\begin{aligned}
fib(2) &\leadsto 2^2 = 4\text{Schritte} \\
fib(10) &\leadsto 2^{10} = 1024\text{Schritte} \\
fib(20) &\leadsto 2^{20} = 1048576\text{Schritte} \\
fib(30) &\leadsto 2^{30} = 1073741824\text{Schritte} \\
fib(50) &\leadsto 2^{50} = 1,125 \cdot 10^{15}\text{Schritte} \\
fib(100) &\leadsto 2^{100} = 1,268 \cdot 10^{30}\text{Schritte}
\end{aligned}
$$

Betrachte 1000000 Schritte/Sekunde

Dann

$$\begin{aligned} fib(100) &= \frac{1,268 \cdot 10^{30}}{1000000} \text{ s} \\ &= 1,268 \cdot 10^{24} \text{ s} \\ &= 4,021 \cdot 10^{16} \text{ Jahre} \end{aligned}$$

Aber zum Glück ist eine effizientere Implementierung möglich.

Es gibt aber auch Probleme, für die keine effizienteren Lösungen als exponentielle Lösungen bekannt sind. Solche Probleme werden wir später in der Vorlesung noch kennen lernen.

5.4 $\mathcal{O}$-Notation

Bisher haben wir Algorithmen mit unterschiedlichen Laufzeitverhalten kennengelernt: konstant, linear, quadratisch, exponentiell, usw. Hierbei ist es natürlich (außer bei konstant) wichtig zu sagen, worin der Algorithmus diese Komplexität hat. Als prägnantere Schreibweise schlug der Zahlentheoretiker Landau die Schreibweise der sogenannten *Groß-O-Notation* vor. Wir schreiben hierbei ein Algorithmus ist in $\mathcal{O}(1)$ falls er konstante Laufzeit besitzt. Lineare Algorithmen in einer Variablen n (was beispielsweise die Länge eines Strings oder die Größe eines Parameters sein kann) notieren wir mit der Laufzeit $\mathcal{O}(n)$, quadratische mit $\mathcal{O}(n^2)$ und exponentielle mit $\mathcal{O}(2^n)$. Hierbei müssen wir immer noch angeben, worin wir dieses Laufzeitverhalten angeben, d.h., was n ist.

Hinter der $\mathcal{O}$-Notation steckt eine formale mathematische Theorie, welche wir hier nicht weiter behandeln wollen. Es ist nur wichtig festzuhalten, dass $\mathcal{O}(1) = \mathcal{O}(2) = \mathcal{O}(42)$, $\mathcal{O}(n) = \mathcal{O}(42 \cdot n + 42)$ und $\mathcal{O}(n^2) = \mathcal{O}(\frac{1}{3}n^2 + 42n + 5)$. Konstante Faktoren, die multipliziert oder addiert werden und Faktoren mit kleinerem Exponenten sind also für die Laufzeitanalyse nicht relevant.

5.5 Suchen von Elementen

Aufgabe: Gegeben ein Liste von Zahlen l und eine Zahl n. Überprüfe, ob n in l vorkommt.

Einzige mögliche Lösung: Vergleiche n mit allen Elementen:

```
def elem(n,l) :
  i = 0
  found = False
  while not(found) and i < len(l) :
    if l[i] == n :
      found = True
    else :
      i = i + 1

  return found

print(elem(3,[1,2,3,4,5]))   ↝ True
```

Welche Laufzeit hat $elem$? Im besten Fall benötigt die Suche nur einen Schleifendurchlauf, wenn $n = a[0]$ gilt. Man sagt: die Best-Case-Laufzeit ist *konstant*.

Im schlechtesten Fall findet sie das Element nicht, dann durchläuft sie die Schleife genau $len(l)$-mal. Im Worst-Case ist sie also *linear in der Größe von* l.

Für alle vorkommenden Werte wird die Schleife im Durchschnitt $len(l)/2$-mal durchlaufen. Also $\frac{1}{2} \cdot len(l)$ viel Durchläufe. Durch den Faktor $\frac{1}{2}$ wird die Laufzeit zwar verbessert, sie ist aber immer noch linear zu $len(l)$, d.h. auch im Durchschnitt (*Average-Case*) ist $elem$ linear in der Größe von l.

Können wir l besser strukturieren/anordnen, damit $elem$ effizienter werden kann?

Hinweis: Telefonbuch

Idee: Sortiere die Elemente

Dann: Suche nur solange, wie n kleiner als aktuelles Element:

```
def elem(n,l) :
  i = 0
  found = False
  while not(found) and i < len(l) and l[i] <= n :
    if l[i]==n :
      found = True
    else:
      i = i + 1

  return found
```

Dann wird auch bei nicht vorkommenden Elementen die Schleife durchschnittlich nur $\frac{1}{2} \cdot len(l)$-mal durchlaufen. Das ist zwar besser, aber immer noch lineare in der Größe von l.

Das es auch noch effizienter geht verdeutlichen wir uns mit Hilfe eines kleinen Spiels: **Zahl raten**:

- Ein Spieler denkt sich eine Zahl aus, der andere muss diese Erraten. Hat der zweite Spieler die Zahl nicht erraten, so sagt der erste Spieler dem zweiten Spieler als Hilfe, ob die geratene Zahl kleiner oder größer als die gesuchte Zahl ist. Der zweite Spieler soll die Zahl möglichst schnell finden.
- Um immer direkt die Hälfte der Zahlen ausschließen zu können ist es natürlich sinnvoll, immer die Zahl in der Mitte des in Frage kommenden Bereichs zu nennen. So kann in einem Schritt die Hälfte aller Werte ausgeschlossen werden.
- Mögliche Rateschritte wären also für eine Zahl zwischen 1 und 64:

  32 - zu klein $\rightsquigarrow$ 48 - zu groß $\rightsquigarrow$ 40 - zu klein $\rightsquigarrow$ 44 - zu groß $\rightsquigarrow$ 42 $\rightsquigarrow$ gefunden

Das hier verwendete Prinzip, den Suchraum in (ungefähr) gleich große Teile aufzuteieln und dann schrittweise zu entscheiden, mit welchem Teil weitergemacht wird, nennt man *Teile und Herrsche (devide and conquer)*. Diese Idee können wir auch auf die Elementsuche übertragen:

5.5.1 Binäre Suche

Idee: Vergleiche n mit mittlerem Element von l:

n ist gleich	$\rightsquigarrow$	gefunden
n ist kleiner	$\rightsquigarrow$	suche in linker Hälfte
n ist größer	$\rightsquigarrow$	suche in rechter Hälfte

Danach erfolgt die Suche innerhalb der Hälfte nach der gleichen Idee. Als Beispiel suchen wir die 8 in einer Liste:

Bsp.:

```
8 in [2, 4, 6, 8, 9, 11, 14, 15, 16, 17, 19, 22, 23]
                              x
                  x
                        x
                     x
```

Es werden also nur 4 Vergleichsschritte benötigt um das Element zu finden. Auch im Fall, dass das Element nicht iin der Liste vorkommt, kann man schnell terminieren, wenn nur noch eine Bereich der Länge eins in Frage kommt. Eine rekursive Implementierung kann einfach angegeben werden.

```
def bin_search(l,n,start,end) :
  if start >= end :
    return False
  else :
    pos = (start + end) // 2
    if n == l[pos] :
      return True
    elif n < l[pos] :
      return bin_search(l,n,start,pos)
    else :
      return bin_search(l,n,pos+1,end)
```

Dann:

```
def elem(n,l) :
  return bin_search(l,n,0,len(l))
```

Bsp.:

$$
\begin{aligned}
& elem(14, \overbrace{[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]}^{=l}) \\
\rightsquigarrow\ & bin\_search(l, 14, 0, 10) \\
\rightsquigarrow\ & pos = (0 + 10)//2 = 5 \\
& a[5] = 16 \neq 16 (> 14) \\
\rightsquigarrow\ & bin\_search(l, 14, 0, 5) \\
\rightsquigarrow\ & pos = (0 + 5)//2 = 2 \\
& a[2] = 13 \neq 14 (< 14) \\
\rightsquigarrow\ & bin\_search(l, 14, 3, 5) \\
\rightsquigarrow\ & pos = (3 + 5)//2 = 4 \\
& a[4] = 14 == 14 \\
\rightsquigarrow\ & return\ True
\end{aligned}
$$

Wie viele Schritte (rekursive Aufrufe) benötigt das Programm nun?

In jedem Schritt wird die eine Hälfte der Werte 'weggeschmissen', d.h. nicht weiter betrachtet und nur noch die andere Hälfte weiter untersucht. Führt man dies rekursiv immer wieder aus, benötigt man log_2 viele Schritte in der Größe des Listen, der Algorithmus ist *logarithmisch in der Größe des Listen*, also $\mathcal{O}(\log n)$ mit n Anzahl der zu sortiertenden Zahlen.

Kann dieser Algorithmus auch iterativ implementiert werden? Ja, mit der gleichen Idee:

```
def bin_search(n,l) :
  left = 0
  right = len(l)
  pos = (left+right) // 2
  while left < right and l[pos] != n :
    if l[pos] < n :
      left = pos+1
      right = pos-1

    pos = (left + right) // 2

  return l[pos] == n
```

Bemerkung: $(left + right)/2 = left + (right - left)/2$

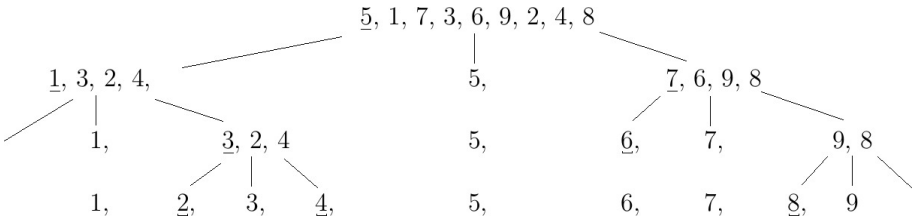
5.6 Effizientes Sortieren

Es stellt sich die Frage ob es auch Sortierverfahren gibt, welche die Aufgabe schneller als in quadratischer Laufzeit lösen können. Wirklich effiziente Lösungen basieren auf der teile und herrsche Idee, welche wir uns am Beispiel des *Quicksort*-Algorithmus anschauen wollen.

Beim Quicksort-Algorithmus wählt man zunächst ein beliebiges Element (meistens das Erste) des Listen (*Pivot*-Element genannt) aus und vergleicht es nach und nach mit allen anderen Elementen des Listen. Es ist klar, dass alle kleineren Elemente links von diesem

einsortiert werden müssen und alle anderen rechts von ihm. Teile also alle anderen Elemente so auf, dass alle kleineren Elemente vor dem Pivotelement stehen und alle größeren Elemente hinter dem Pivotelement. Mit den Kleineren und dem Größeren verfahre rekursiv so weiter, bis nur noch einelementige Listen übrig bleiben, die natürlich sortiert sind.

Bsp.:



Als erste Implementierung, verwenden wir jeweils ein neues Liste, in das wir die kleineren Werte von vorne und die größeren Werte nach hinten packen können. Danach wird rekursiv für die kleineren und größeren Elemente mittels Quicksort sortiert. Wir verwenden jeweils das erste Element des zu sortierenden Bereichs (a[l]) als Pivot-Element.

```
def qsort_h1(l,start,end) : # Sortiert den Bereich von l bis r
  l1 = list(l)
  if start >= end :
      return l1
  else :
      j = start
      k = end - 1
      for i in range(start+1,end) :
        if l[i] < l[start] : # Vergleich mit Pivot-Element
            l1[j] = l[i]
            j = j+1
        else :
            l1[k] = l[i]
            k = k-1

      l1[j] = l[start]  # Verschiebe Pivot-Element in die Mitte
      l2 = qsort_h1(l1,start,j)
      return qsort_h1(l2,j+1,end)

def qsort1(l) :
  return qsort_h1(l,0,len(l))

print(qsort1([2,6,8,4,6,2,3,8,1]))
```

Dieser Algorithmus arbeitet aber nicht in Place, d.h. er benötigt weiteren Speicher. Das Liste wird immer wieder kopiert. Hierdurch bleibt das UrsprungsListe zwar erhalten, aber auch für alle ZwischenListen, wird separater Speicher verwendet, welcher später aber wieder frei gegeben wird. Eine effizientere Implementierung basiert auf der Idee, dass man, wenn man eine Vertauschungsoperation verwendet auch durch geschicktes Vertauschen innerhalb des Listen die Elemente an Hand des Pivotelementes aufteilen kann:

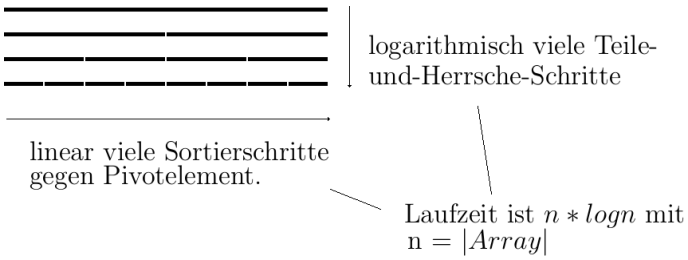
l								
5	1	7	3	6	9	2	4	8
m	i							
5	1	7	3	6	9	2	4	8
l	m	i	i					
5	1	3	7	6	9	2	4	8
l		m		i	i	i		
5	1	3	2	6	9	7	4	8
l			m				i	
5	1	3	2	4	9	7	6	8
				m				i
4	1	3	2	5	9	7	6	8
l				m				i

Dies kann wie folgt in Python umgesetzt werden:

```
def qsort_h(l,start,end) : # Sortiert den Bereich von l bis r
  if start >= end :
    return l
  else :
    m = start
    for i in range(start+1,end) :
      if l[i] < l[start] :
        m = m + 1
        swap(i,m,l)  # swap! siehe Insertion-Sort

    swap(start,m,l)
    qsort_h(l,start,m-1)
    return qsort_h(l,m+1,end)

def qsort(l) :
  return qsort_h(l,0,len(l))
```

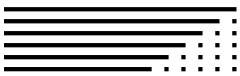
Alternative Variante (Übung)

Laufzeitanalyse: Im Durchschnitt wird das Pivot-Element die Liste in zwei gleich große Hälften teilen welche dann beide jeweils wieder sortiert werden müssen:



Die Laufzeitanalyse klingt, als ob wir eine Best-Case-Analyse gemacht hätten. Nur bei guten Pivot-Elementen wird gleichmäßig aufgeteilt. Der Fall tritt allerdings fast immer auf, weshalb man als Average-Case Laufzeit $\mathcal{O}(n \cdot \log n)$ mit n Größe des zu sortierenden Listen erhält.

Allerdings gibt es immer noch einen schlechten Fall, in dem Quicksort eine schlechtere Laufzeit hat:



linear viele Vergleiche,
wenn Array sortiert oder
falsch herum sortiert ist.

Die Worst-Case Laufzeit beträgt also immer noch $\mathcal{O}(n^2)$, mit n Länge der Liste.

Man verhindert dies dadurch, dass ein zufälliges Pivotelement gewählt wird. Hierzu kann man zu Beginn des else-Falles das erste Element gegen ein anderes zufällig gewähltes Element getauscht werden:

```
import random

def qsort_h(l,start,end) : # Sortiert den Bereich von l bis r
  if lstart >= end :
    return a
  else :
      swap!(l,start,random.randrange(start,end))  # create random number n
      m = start                                   # with n >= start and n
      ...
```

Dann ist der Worst-Case sehr unwahrscheinlich $\Rightarrow$ in der Praxis immer Laufzeit $\mathcal{O}(n \cdot \log n)$ mit $n = len(l)$.

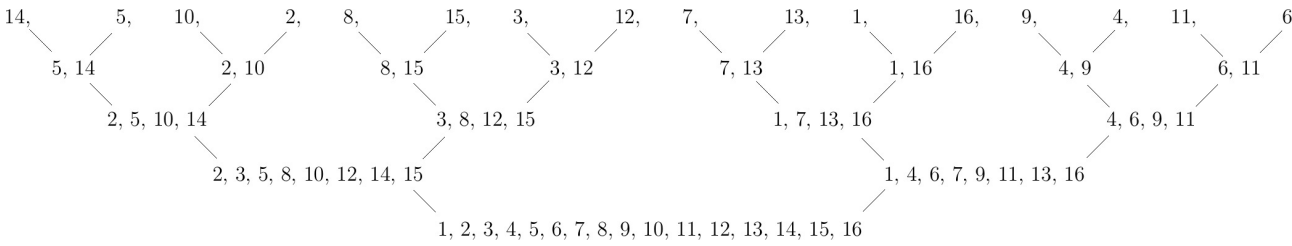
5.6.1 Andere effiziente Sortieralgorithmen

Merge-Sort Teile Liste in zwei gleich große Hälften, sortiere die Hälften rekursiv und mische die beiden sortierten Ergebnisse zusammen. Nachteil gegenüber Quicksort: nicht einfach in Place möglich: beim Mischen wird zweite Liste benötigt.

	1	3	5	7	2	4	6	8	Wie innerhalb des
$\rightsquigarrow$	1	2	3	4	5	6	7	8	Listen machbar?

Aber: Algorithmus hat auch Worst-Case-Komplexität $n \cdot \log n$ mit $n = \#$ zu sortierende Werte, im Gegensatz zu Quicksort, bei dem dies nur randomisiert erreicht werden kann.

Bsp.:



Allerdings ist Quicksort in der Praxis oft schneller (insbesondere randomisiert).

5.7 Schwere Probleme

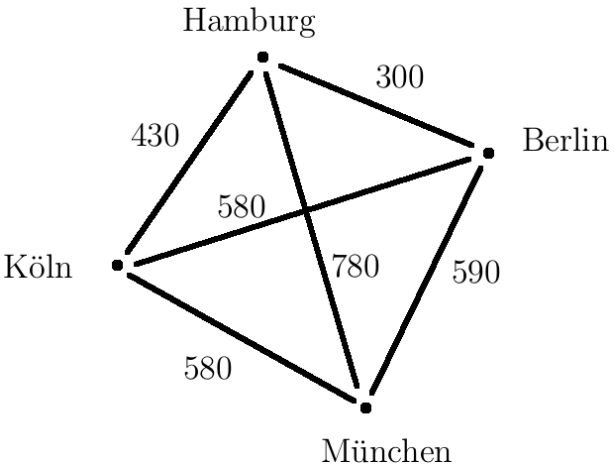
Beim Sortieren empfanden wir quadratische Laufzeit als schlecht. Es gibt aber auch Probleme, bei denen man sich quadratische, kubische oder allgemeiner polynomielle Laufzeiten wünschen würde. Die besten bekannten Lösungen sind hier aber mindestens exponentiell.

Bsp.: Planung einer Rundreise (Handlungsreisenden-Problem, Travelling Salesman-Problem) TSP.

Gegeben: n zu besuchende Städte und die Entfernungen zwischen je zwei dieser Städte.

Gesucht: kürzeste Rundtour über alle Städte.

Beispiel:



430	300
580	430
590	580
780	590
2380	1900

Im allgemeinen schwierig, nur exponentielle Algorithmen oder eff. Näherungsalgorithmen bekannt. Als weiteres Beispiel für solche Probleme betrachten wir folgendes Problem:

SAT – Erfüllbarkeit (satisfyability):

Gegeben: Aussagenlogische Formel φ mit n booleschen Variablen $(b_1, \ldots, b_n)$

Frage: Gibt es boolesche Belegung für b_1 bis b_n, so dass φ unter dieser Belegung zu True auswertet?

Bsp.: $(b_1\ and\ b_2)\ or\ !(b_3\ and\ b_2)$

Wähle: $b_1 = True$, $b_2 = True$, $b_3 = False \rightsquigarrow True$

$(b_1\ and\ b_2)\ and\ !(b_3\ or\ b_2)$ ist aber nicht erfüllbar, da $b_2 = True$ wegen and, aber $b_2 = False$ wegen zweitem or gelten muss.

Um SAT zu lösen muss man im Allgemeinen alle möglichen Belegungen durchtesten, was aber exponentiell viele (2^n) Belegungen sind.

Man kann aber einen einfachen nicht-deterministischen Algorithmus angeben, der das Problem löst: Wähle nicht-deterministisch einfach eine (richtige) Belegung aus und überprüfe, ob diese zu True auswertet. D.h. es gibt einen nicht-deterministisch linearen Algorithmus in der Größe der Formel.

Entsprechend kann man auch für das TSP einen nichtdeterministisch polynomiellen Algorithmus finden. Deterministisch sind bis jetzt aber nur exponentielle Algorithmen bekannt. Eine offene Frage der Informatik ist, ob es generell möglich ist, für solche Algorithmen auch deterministische Algorithmen mit polynomieller Laufzeit zu finden. Hierzu fasst man alle

Probleme, für welche es nichtdeteministische polynomielle Algorithmen gibt, in der Klasse NP und alle Probleme, für die deterministische polynomielle Algorithmen existieren in der Klasse P zusammen. Dann ist eine offene Frage der Informatik, ob

$$NP = P \quad \text{oder} \quad \underbrace{NP \neq P}?$$

Es wird allgemein die Ungleichheit der beiden Klassen vermutet, was aber sehr schwer zu zeigen ist, da über alle möglichen Algortihmen argumentiert werden muss.

Beachte: Bei Komplexitätsbetrachtungen ist nicht nur die Laufzeit wichtig, sondern oft auch Speicherbedarf. Quadratischer Speicherverbrauch ist oft kritischer als quadratische Laufzeit.

Es gibt aber auch Probleme, für die nicht einmal exponentielle Lösungen bekannt sind. Z.B. Laufzeit $\underbrace{2^{2^{2^{2^{\cdots}}}}}_{\text{n-mal}}$!

$\rightsquigarrow$ noch schlimmer, sogar **unlösbare Probleme:**

5.7.1 Halteproblem

Nachdem wir nun ja schon einige Programme geschrieben haben, stellt sich vielleicht folgende (scheinbar) einfache Aufgabe:

Gesucht: Programm, welches ein Python-Programm P einliest und True ausgibt, falls P terminiert, und False, falls P nicht terminiert.

Zeige Unlösbarkeit: Angenommen, wir könnten folgendes Programmfragment/Funktionsdefinition definieren:

```
def terminiert(prog_name) :
  f = open(prog_name)
  text = f.read()
  f.close()
  ....
  return b # mit True,  falls Programm in text terminiert
           # und False, sonst
```

Heirbei würde die Funktion $\mathrm{terminiert(prog\_name)}$ das Ergebnis True liefern, falls das übergebene Programm $\mathrm{prog\_name}$ (in Form des Dateinamens) bei seiner Ausführung terminiert und False sonst. Da Programme ja nicht als Datum übergeben werden können, übergeben wir den Dateinamen und lesen als erste Anweisungen in der Definition von terminiert das Programm ein. Danach haben wir es als String in der Variablen $text$ vorliegen, welchen wir dann irgendwie zur weiteren Analyse verarbeiten könnten.

Dann könnten wir folgendes Testprogramm schreiben und unter dem Dateinamen test.py abspeichern:

```
def terminiert(prog_name) :
  f = open(prog_name)
```

```
    text = f.read()
    f.close()
    ....
    return b # mit True,  falls Programm in text terminiert
             # und False, sonst

if terminiert('test.py') :
    while True:
        print('I will not terminate')
else :
    print('Fertig')
```

Würde man dann

```
python3 test.py
```

aufrufen, so würde dieser Aufruf terminieren, falls test.py nicht terminiert und es würde nicht terminieren, falls test.py terminiert.

Dies ist ein Widerspruch, woraus direkt folgt, dass $terminiert$ nicht definiert werden kann.

5.8 Besonderheiten von Python

Im folgenden wollen wir noch eine paar besondere Strukturen von Python betrachten. In vielen anderen Programiersprachen finden sich ähnliche Konzepte, aber eben nicht in allen. Dennoch kann e sinnvoll sein, solche Konstrukte zu verwenden, da sie es dem Programmierer möglich machen, viele Algorithmen abstrakter und damit auch kompakter auszudrücken.

for-Schleifen

Bisher haben wir for-Schleifen immer verwendet um über Zahlen zu iterieren und hierfür die Funktion range verwendet. In Kombination mit Listen sah das z.B. wie folgt aus:

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

sum = 0
for i in range(l):
    sum = sum + l[i]

print(sum)
```

Hierbei ist man eigentlich gar nicht an dem Index interessiert, sondern nur an allen Elementen des Arrays. Dennoch muss man die Indizes explitiz aufzählen. Deshalb ist es in vielen Programmiersprachen (außer z.B. in C) alternativ auch möglich, eine Schleife direkt über alle Elemente einer Aufzählbaren Struktur laufen zu lassen, also z.B. über eine Liste. Hierdurch vereinfacht sich unser Programm wie folgt:

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

sum = 0
for elem in l:
```

```
    sum = sum + elem

print(sum)
```

Die Variable elem wird nun der Reihe nach an alle Elemente der Liste gebunden. Tatsächlich ist auch die range, welche wir bisher verwendet ein Object (wie eine Liste), welche alle seine Werte nach und nach verfügbar macht (auch wie eine Liste). Solche Strukturen nennt man meist Iteratoren oder iterierbare Strukturen. Es ist auch möglich, für eigene Klassen solche Iterator-Eigenschaften zu definieren, was wir hier aber nicht weiter betrachten wollen.

Für Iteratoren kann man das Schlüsselwort in auch verwenden, um zu überprüfen, ob eine Wert in der Sturktur enthalten ist.

```
print(3 in [1,2,3,4,5,6])   # True
print(3 in range(1,6))      # True
print(3 in range(11,16))    # False
```

5.8.1 List-Comprehensions

Listen können in Python sehr elegant mit Hilfe von List-Comprehensions definiert werden. Diese sind sehr nah an der in der Mathematik üblichen Notation von Mengen. Der Aufbau ähnelt der for-Schleife, nur dass das die Schlüsselwörter in den Rumpf einer Liste wandern. Als eifnaches Beispiel definieren wir zunächst die Liste aller geraden Zahlen kleiner 10:

```
[ n for n in range(20) if n % 2 == 0] # liefert: [0,2,4,6,8]
```

Die Zunächst legt man mit dem ersten Ausdruck fest, welche Werte in die Liste aufgenommen werden. Dann generiert man möglich Kandidaten mit Hilfe eines for-in-Ausdrucks und fügt anshcließend mit if noch einschränkungen hinzu. Hierbei ist es auch möglich, mehrere Iteratoren zu schachteln, wir das folgende Beispiel zeigt:

```
[ (n,m) for n in range(3) for m in range(4,6) if n != m ]
# liefert [(0, 4), (0, 5), (1, 4), (1, 5), (2, 4), (2, 5)]
```

Hier wurden die Ergebnisse als Paare zusammengefasst. Wir könne sie aber z.B. auch addieren oder mit anderen Operatoren/Funktonen verknüpfen:

```
[ n + m for n in range(3) for m in range(4,6) if n != m ]
# liefert [4, 5, 5, 6, 6, 7]
```

5.8.2 Dictionaries als Verallgemeinerung von Listen

Wenn wir bisher mehrere Werte zu einem Wert zusammenfassen wollten, haben wir entweder Klassen oder Listen verwendet. Listen sind besonders gut geeignet, eine beliebige Anzahl von Werten abzuspeichern. Hierbei ist später der Zugriff aber immer über einen Index zwischen 0 und der Listegröße-1 notwendig. Wenn man keine gute Zuordnung zwischen solchen Indizes und den gespeicherten Werten kennt, kann die Liste die falsche Datenstruktur sein. In manchen Fällen hätte man lieber einen Schlüssel, unter dem man (ähnlich, wie in einer Datenbank) Werte abspeichert und später wieder nachschlagen kann.

Als Bespiel betrachten wir ein Telefonbuch. Wenn man die Telefonnummer einer Person finden will, so wird man möglichst nicht das ganze Telefonbuch nach der Person durchsuchen, sondern direkt auf den Namen des gesuchten Teilnehmers zugreifen wollen. Unter Verwendung von binärer Suche haben wir ein Verfahren kennen gelernt, wie wir dies effizient realisieren können. Es gibt aber auch noch andere Ansätze, welche direkt über einen Schlüssel gehen und in Python als Dictionary zur Verfügung stehen.

Ein Dictionary stellt eine Schlüssel-Wert-Struktur zur Verfügung, bei der unter einem eindeutigen Schlüssel genau ein Wert abgespeichert werden kann. Ein mögliches Telefonbuch sähe als Dictionary wie folgt aus:

```
telefonbook = { 'Frank': 7277, 'Steven': 7264, 'Sandro': 4843 }
print(telefonbook['Frank'])
```

Der Zugriffs auf die gespeicherten Werte im Dictionary erfolgt genau wie bei Listen, nur dass man den Schlüssel anstelle des Indexes verwendet. So könnne wir z.B. die Telefonnummer von Steven wie folgt korrigieren:

```
telefonbook['Steven'] = 7263
```

Dictionaryes können auch erweitert werden:

```
telefonbook['Notruf'] = 112
```

fügt einen neues Schlüssel-Wert-Paar zu unserem Telefonbuch hinzu, welches danach genau wie die anderen Schlüssel nachgeschlagen werden kann.

Im Gegensatz zu Listen können Dictionaries nicht mit + zusammengefügt werden. Möchte man einen Dictionary erweitern, müssen alle Werte einzeln hinzu gefügt werden. Ähnlich wie bei Listen kann man die Länge eines Dictionaries mittels len abfragen. In unserem Beispiel hätte das telefonbook nach hinzufügen der Notrufnummer die Länge 4.

Für die Implementierung eines Dictionaries gibt es unterschiedliche Verfahren. Zum einen kann man eine Funktion definieren, welche die Schlüssel auf die Indizes eines gegebenen Liste fester Größe abbildet. Eine gute Funktion versucht hierbei möglichst alle Schlüssel mit der gleichen Wahscheinlichkeit zu erreichen. Andere Implementierungen basieren auf (höhenbalancierten) Suchbäumen, welche ähnlich wie bei der binären Suche eine Auffinden eines Schlüssels mit logarithmischer Laufzeit ermöglichen.

Es ist auch möglich über die Einträge eines Dictionaries zu iterieren, wie die folgenden Beispiele zeigen:

```
en_de = {"red" : "rot", "green" : "gruen", "blue" : "blau", "yellow":"gelb"}

for k in en_de :
    print(k)      # liefert green blue red yellow (untereinander)

for v in en_de.values() :
    print(v)      # liefert gruen blau rot gelb (untereinander)

for k,v in en.de.items() :
    print(k,v)    # liefert: green grún
                  #          blue blau
                  #          red rot
                  #          yellow gelb
```

Man beachte, dass hier in der for-Schleife zwei Variablen belegt werden. Die erste mit dem Schlüssel und die zweite mit dem zugehörigen Wert. Solche Zuweisungen sind in Python über Paare möglich, wie das folgende Beispiel zweigt:

```
x,y = (4, 'Hallo')

print(len(y)+x)  # Gibt 9 aus.
```

5.8.3 Higher-Order

In modernen Programmiersprachen ist es auch möglich Funktionen als Werte zu behandeln, das heißt als Parameter zu übergeben, als Rückgabewerte zurück zu geben oder Funtkionen in Datenstrukturen zu speichern.

Wir betrachten hierzu das folgende Beispiel:

```
def inc(n) :
  return n+1

def dec(n) :
  return n-1

def apply(fun,arg) :
  return fun(arg)

print(apply(inc,41))  # 42
print(apply(dec,74))  # 73
```

Die Funktion apply erwartet zwei Argumente: eine Funktion als ersten Parameter und einen Wert als zweiten Parameter. Das Ergebnis ist dann die Anwendung der Funktion auf den Parameter. Im Hauptprogramm verwenden wir dann apply einmal mit der Funktion inc und einmal mit der Funktion dec.

Sieht man bei seinen Funktionen andere Funktionen als Parameter vor, ermöglicht man es einem Nutzer der Funktion auf elegante Weise Einfluss auf das konkrete Verhalten zu nehmen. Als Beispiel definieren wir eine Funktion map, welche eine übergebene Funktion auf alle Elemente einer Liste anwendet:

```
def map(f,l) :
  res = []
  for e in l :
    res.append(f(e))
  return res

print(map(inc,[1,2,3,4]))  # [2,3,4,5]
print(map(dec,[1,2,3,4]))  # [0,1,2,3]
```

Hier haben wir map nicht-mutierend definiert. Eine mutierende Variante wäre natürlich auch möglich.

Manchmal kann es praktisch sein, Funktionen direkt als Werte zu definieren, also ohne eine Funktionsdefinition mit def und return anzugeben. Hierzu verwendet man das Schlüsselwort lambda, wie folgende Anwednungen von map zeigen:

```
l =  [1,2,3,4,5]

print(map(lambda x: x + 1,l) # [2,3,4,5,6]
print(map(lambda x: x * 2,l) # [2,4,6,8,10]
```