A Python Grundlagen

A.1 Die Pythonshell

Pythonprogramme werden i. A. nicht compiliert sondern durch einen Interpreter ausgeführt. Python bietet eine interaktive „Shell“ an, mit der Pythonausdrücke und -kommandos auch direkt am Pythoninterpreter ausprobiert werden können. Diese Shell arbeitet in einer sog. *Read-Eval-Print-Loop* (kurz: REPL): Pythonausdrücke werden also interaktiv eingelesen, diese werden ausgewertet und der Ergebniswert ausgegeben (sofern er eine Stringrepräsentation besitzt). Wird dagegen ein Python-Kommando eingegeben, so wird das Kommando einfach durch Python ausgeführt. Diese interaktive Pythonshell erweist sich besonders für das Erlernen, Ausprobieren und Experimentieren mit Algorithmen als didaktisch nützlich.

Pythons Shell kann entweder von der Kommandozeile aus durch Eingabe des Kommandos „`python`“ gestartet werden – dies ist etwa unter Linux und Linux-ähnlichen Betriebssystemen üblich. Windows-Installationen bieten darüberhinaus oft die spezielle Anwendung „IDLE“ an, mit der die Pythonshell betreten werden kann. Hier ein Beispiel für das Verhalten der Pythonshell (das „>>>“ stellt hierbei die Eingabeaufforderung der Pythonshell dar):

```
>>> x = 2**12
>>> x/2
2048
```

In der ersten Zeile wurde ein Kommando (nämlich eine Zuweisung) eingegeben, das durch Python ausgeführt wurde (und keinen Rückgabewert lieferte). In der zweiten Zeile wurde ein Ausdruck eingegeben; dieser wird ausgewertet und die Stringrepräsentation auf dem Bildschirm ausgegeben.

A.2 Einfache Datentypen

A.2.1 Zahlen

Pythons wichtigste Zahlen-Typen sind Ganzzahlen (*int*), lange Ganzzahlen (*long int*), Gleitpunktzahlen (*float*). Einige einfache Beispiele für Python-Zahlen sind „12“, „3.141“, „$4.23E-5$“ (Gleitpunkt-Darstellung), „$0xFE$“ (hexadezimale Darstellung), „3/4“ (Bruchzahlen), „$12084131941312L$“ (long integers mit beliebig vielen Stellen).

A.2.2 Strings

Strings sind in Python Sequenzen einzelner Zeichen. Im Gegensatz zu Listen und Dictionaries (die wir später ausführlich behandeln) sind Strings *unveränderlich*, d. h. ist ein bestimmter String einmal definiert, so kann er nicht mehr verändert werden. Man hat die Wahl, Strings entweder in doppelte Anführungszeichen (also: `"..."`) oder in einfache Anführungszeichen (also: `'...'`) zu setzen. Die spezielle Bedeutung der Anführungszeichen kann, ganz ähnlich wie in der `bash`, mit dem Backspace (also: \) genommen werden. Syntaktisch korrekte Python-Strings wären demnach beispielsweise:
`"Hallo"`, `'Hallo'`, `'"Hallo"'`, `'\'\''`, `"Python's"`, `'Hallo Welt'`, ...

Verwendet man dreifache Anführungszeichen (also: `"""..."""` oder `'''...'''`), so kann man auch mehrzeilige Strings angeben.

Aufgabe A.1

Geben Sie mit dem Python **print**-Kommando den Text
`Strings in Python koennen entweder mit "double ticks"`
`oder mit 'einfachen ticks' umschlossen werden.`

A.2.3 Variablen

Variablen sind, genau wie in anderen Programmiersprachen auch, (veränderliche) Platzhalter für bestimmte Werte. Variablennamen müssen mit einem Buchstaben oder mit dem Zeichen „_“ beginnen und dürfen keine Leerzeichen oder Sonderzeichen (außer eben dem Zeichen „_“) enthalten. Korrekte Variablennamen sind beispielsweise „i“, „$\_i$“, „*Kaese*“ oder „*kaese*“; die Zeichenketten „2*dinge*“ oder „*leer zeichen*“ wären beispielsweise keine korrekten Variablennamen.

A.2.4 Typisierung

Python ist, im Gegensatz zu vielen gängigen Programmiersprachen, nicht statisch getypt; d. h. der Typ einer Variablen muss nicht vor Ausführung eines Programms festgelegt sein, sondern er wird dynamisch – also während der Programmausführung – bestimmt. Das hat den Vorteil, dass Variablen nicht deklariert werden müssen; man muss Ihnen einfach einen Wert zuweisen, wie etwa in folgendem Beispiel:

>>> $x = 2.01$

Der Python-Interpreter leitet dann einfach den Typ der Variablen aus der ersten Zuweisung ab.

Die Verwendung von Variablen kann grundsätzlich flexibler erfolgen als bei statisch getypten Programmiersprachen. Ein Beispiel (das die **if**-Anweisung verwendet, die im nächsten Abschnitt eingeführt wird):

if *gespraechig*:
  $x =$ `"Guten Morgen"`

else:
  $x = 12**12$
print x

Der Typ der Variablen x ist vor Programmausführung nicht bestimmt. Ob s vom Typ *str* oder vom Typ *long int* sein wird, hängt vom Inhalt der Variablen *gespraechig* ab.

A.2.5 Operatoren

Die folgende Tabelle zeigt eine Auswahl an Operatoren, die Python anbietet, um Ausdrücke zu verknüpfen.

$X + Y$, $X - Y$	Plus/Konkatenation, Minus

Beispiele:	>>> 2+3
	5
	>>> '2'+'3'
	'23'
	>>> [1,2,3]+[10]
	[1,2,3,10]

$X * Y$, $X ** Y$	Multiplikation, Potenzierung

Beispiele:	>>> 2*6
	12
	>>> '2'*6
	'222222'
	>>> [0,1]*3
	[0,1,0,1,0,1]

X / Y, $X // Y$ $X \% Y$	Division, restlose Division Rest (bei der Division)

Beispiele:	>>> 2.0/3
	0.66666666
	>>> 2/3
	0
	>>> 17%7
	3

$X < Y$, $X \leq Y$ $X > Y$, $X \geq Y$	kleiner, kleinergleich (lexikographisch bei Sequenzen) größer, größergleich (lexikographisch bei Sequenzen)

Beispiele:	>>> 4<2
	False
	>>> 'big'<'small'
	True
	>>> [1,100]<[2,1]
	True

$X == Y$, $X != Y$ X **is** Y, X **is not** Y X & Y, $X \mid Y$, $X \hat{} Y$ $\tilde{} X$ $X << Y$, $X >> Y$	Gleichheit, Ungleichheit (Werte) Objektgleichheit, Objektungleichheit Bitweises „Und“, bitweises „Oder“, bitweises exkl. „Oder“ Bitweise Negation Schiebe X nach links, rechts um Y Bits

Beispiele:	>>> 9 & 10
	8
	>>> 10 | 6
	14
	>>> 3 << 4
	48

X **and** Y X **or** Y **not** X X **in** S	Wenn X falsch, dann X, andernfalls Y Wenn X falsch, dann Y, andernfalls X Wenn X falsch, dann *True*, andernfalls *False* Test auf Enthaltensein eines Elements X in einer Kollektion S von Werten.

Beispiele:	>>> *True* **and** *False*
	False
	>>> 'al' **in** 'hallo'
	True
	>>> 4 **in** [1,2,3]
	False

Einige der Operatoren sind *polymorph*, d. h sie sind auf unterschiedliche Typen anwendbar. Die hier wirkende Art der Polymorphie nennt man auch *Überladung*. Ein überladener Operator verwendet i. A. für verschiedene Typen auch verschiedene Algorithmen. Ein typisches Beispiel stellt der Python-Operator + dar: Er kann sowohl auf Strings oder auf Listen, als auch auf Ganzzahlwerte, auf Fließkommawerte oder auf komplexe Zahlen angewendet werden; während der +-Operator Strings und Listen konkateniert (d. h. zusammenfügt) führt er auf Zahlenwerten eine klassische Addition durch.

A.3 Grundlegende Konzepte

A.3.1 Kontrollfluss

Einrücktiefe. Die *Einrücktiefe* von Python-Kommandos spielt – im Gegensatz zu vielen anderen Programmiersprachen – eine Rolle. Damit haben die Leerzeichen am Zeilenanfang eine Bedeutung und sind Teil der Syntax der Programmiersprache. Die Einrücktiefe dient dazu Anweisungsblöcke zu spezifizieren: Anweisungen, die dem gleichen Anweisungsblock angehören, müssen die gleiche Einrücktiefe haben. Der Anweisungsblock, der einer **if**-Anweisung oder einer **while**-Anweisung folgt, wird also nicht explizit eingeklammert, sondern die Anweisungen werden durch den Python-Interpreter dadurch als zugehörig erkannt, dass sie dieselbe Einrücktiefe haben.

Steuerung des Kontrollflusses. Wie beschrieben im Folgenden die drei wichtigsten Kommandos zur Steuerung des Kontrollflusses, d. h. zur Steuerung des Ablaufs der Python-Kommandos: Die **if**-Anweisung, die **while**-Anweisung und die **for**-Anweisung. Für die Syntaxbeschreibungen dieser (und weiterer) Kommandos werden die folgenden Formalismen verwendet:

- In *eckigen Klammern* eingeschlossene Teile (also: [...]) sind optionale Teile, d. h. diese Teil der Syntax können auch weggelassen werden.
- Ist der in eckigen Klammern eingeschlossene Teil von einem Stern gefolgt (also: [...]*), so kann der entsprechende Teil beliebig oft (auch 0-mal) wiederholt werden. Beispielsweise kann der **elif**-Teil der **if**-Anweisung beliebig oft (und eben auch 0-mal) hintereinander verwendet werden.

Die **for**-Schleife:

if ⟨*test*⟩ : ⟨*Anweisungsfolge*⟩ [**elif** ⟨*test*⟩ : ⟨*Anweisungsfolge*⟩]* [**else** : ⟨*Anweisungsfolge*⟩]	Die **if**-Anweisung wählt eine aus mehreren Anweisungsfolgen aus. Ausgewählt wird diejenige Anweisungsfolge, die zum ersten ⟨*test*⟩ mit wahrem Ergebnis gehört.

Beispiel:

if $a < b$:
  $x = [a,b]$
elif $a > b$:
  $x = [b,a]$
else:
  $x = a$
	Dieses Beispiel implementiert eine Fallunterscheidung: Je nachdem, ob $a < b$, ob $a > b$ oder ob keiner der beiden Fälle gilt, wird der Variablen x ein anderer Wert zugewiesen.

Die **while**-Schleife:

while ⟨*test*⟩ :
  ⟨*Anweisungsfolge*⟩
[**else** :
  ⟨*Anweisungsfolge*⟩]
	Die **while**-Anweisung stellt die allgemeinste Schleife dar. Die erste ⟨*Anweisungsfolge*⟩ wird solange ausgeführt, wie ⟨*test*⟩ wahr ergibt. Die zweite ⟨*Anweisungsfolge*⟩ wird ausgeführt, sobald die Schleife normal (d. h. ohne Verwendung der **break**-Anweisung) verlassen wird.

Beispiel:

$z = 42$; *geraten* = *False*
while not *geraten*:
  r=*int*(*raw_input*(`'Zahl?'`))
  if $r < z$:
  print `'Hoeher!'`
  elif $r > z$:
  print `'Niedriger!'`
  else: *geraten* = *True*
	Diese **while**-Schleife implementiert ein einfaches Ratespiel. Mittels der Funktion *raw_input* wird von Standardeingabe ein String eingelesen und in mittels der Funktion *int* in eine Zahl konvertiert. Ist der eingelesene Wert ungleich z, so wird eine entsprechende Meldung ausgegeben. Hat der Benutzer richtig geraten, wird die Variable *geraten* auf den Wert „*True*“ gesetzt. Daraufhin bricht die **while**-Schleife ab, da ihre Bedingung „**not** *geraten*“ nicht mehr gilt.

Die **for**-Schleife:

for ⟨*ziel*⟩ **in** ⟨*sequenz*⟩ :
  ⟨*Anweisungsfolge*⟩
[**else** :
  ⟨*Anweisungsfolge*⟩]
	Die **for**-Schleife ist eine Schleife über Sequenzen (also Listen, Tupel, ...). Die Variable ⟨*ziel*⟩ nimmt hierbei für jeden Schleifendurchlauf einen Wert der Sequenz ⟨*sequenz*⟩ an.

Beispiel:

s=0 **for** *c* **in** `'12345'`: *s*+=*int*(*c*)	Die **for**-Schleife durchläuft den String `'12345'` zeichenweise; es wird also fünftmal die Zuweisung *s*+=*int*(*c*) ausgeführt, wobei die Variable *c* immer jeweils eines der Zeichen in `'12345'` enthält. Die Variable *s* enthält also nach Ausführung der Schleife den Wert $\sum_{i=1}^{5} i = 15$.

for *i* **in** *range*(10,20):
  print `'i ist jetzt'`,*i*
	Die Funktion *range* erzeugt eine Liste der Zahlen von 10 bis ausschließlich 20. Dieses Programm gibt die Zahlen 10 bis (ausschließlich) 20 in der folgenden Form aus:
	`i ist jetzt 10`
	`i ist jetzt 11`
	...
	`i ist jetzt 19`

Im letzten Programmbeispiel wird die Pythonfunktion *range* verwendet. Diese gibt eine Liste ganzer Zahlen im angegebenen Bereich zurück; $range(a,b)$ liefert alle ganzen Zahlen zwischen (einschließlich) a und (ausschließlich) b zurück. Es gilt also:

$$range(a,b) == [a, a+1, \ldots, b-2, b-1]$$

Optional kann man auch als drittes Argument eine Schrittweite angeben. Beispielsweise liefert $range(1,9,2)$ als Ergebnis die Liste [1,3,5,7] zurück. Es gilt also

$$range(a,b,c) == [a, a+c, a+2c, \ldots, b-2c, b-c]$$

übergibt man *range* nur ein einziges Argument, so beginnt die Ergebnisliste bei 0. Es gilt also

$$range(a) == [0, 1, \ldots, a-2, a-1]$$

Aufgabe A.2

(a) Erweitern Sie das als Beispiel einer **while**-Schleife dienende Ratespiel so, dass eine Ausgabe erfolgt, die informiert, wie oft geraten wurde (etwa „`Sie haben 6 Rate-Versuche gebraucht.`“).

(b) Erweitern Sie das Programm so, dass das Ratespiel vier mal mit vier unterschiedlichen Zahlen abläuft; am Ende sollen Sie über den besten Rate-Lauf und den schlechtesten Rate-Lauf informiert werden, etwa so:
`Ihr schlechtester Lauf: 8 Versuche; ihr bester Lauf: 3 Versuche.`

Aufgabe A.3

(a) Schreiben Sie ein Pythonskript, das die Summe aller Quadratzahlen zwischen 1 und 100 ausgibt.

(b) Schreiben Sie ein Pythonskript, das eine Zahl n von der Tastatur einliest und den Wert $\sum_{i=0}^{n} i^3$ zurückliefert.

(c) Schreiben Sie ein Pythonskript, das zwei Zahlen n und m von der Tastatur einliest und den Wert $\sum_{i=n}^{m} i^3$ zurückliefert.

Aufgabe A.4

Schreiben Sie ein Pythonskript, das Ihnen die vier kleinsten perfekten Zahlen ausgibt. Eine natürliche Zahl heißt perfekt, wenn sie genauso groß ist, wie die Summe Ihrer positiven echten Teiler (d. h Teiler außer sich selbst). Beispielsweise ist 6 eine perfekte Zahl, da es Summe seiner Teiler ist, also $6 = 1 + 2 + 3$.

A.3.2 Schleifenabbruch

Die beiden im Folgenden vorgestellten Kommandos, **break** und **continue** geben dem Programmierer mehr Flexibilität im Umgang mit Schleifen; man sollte diese aber spar-

sam verwenden, denn sie können Programme schwerer verständlich und damit auch schwerer wartbar[1] werden lassen.

Mit der **break**-Anweisung kann man vorzeitig aus einer Schleife aussteigen; auch ein möglicherweise vorhandener **else**-Zweig wird dabei nicht mehr gegangen. Folgendes Beispiel liest vom Benutzer solange Zahlen ein, bis eine „0“ eingegeben wurde.

```
while True:
  i = int(raw_input('Bitte eine Zahl eingeben: '))
  if i == 0: break
print 'Fertig'
```

Mit der **continue**-Anweisung kann man die restlichen Anweisungen im aktuellen Schleifendurchlauf überspringen und sofort zum Schleifen„kopf“ springen, d. h. zum zur Prüfanweisung einer **while**-Schleife bzw. zum Kopf einer **for**-Schleife, der der Schleifenvariablen das nächste Element der Sequenz zuordnet.

A.3.3 Anweisungen vs. Ausdrücke

Gerade für den Programmieranfänger ist es wichtig, sich des Unterschieds bewusst zu sein zwischen ...

- ... einer *Anweisung*, die etwas „tut“, d. h. eigentlich einen Rechner- oder Programminternen Zustand verändert, wie etwa das Ausführen einer Variablenzuweisung, das Verändern des Speicherinhalts, das Ausführen einer Bildschirmausgabe) und
- ... einem *Ausdruck*, der einen bestimmten Wert repräsentiert.

Beispiele. Der Python-Code x=5+3 stellt eine Anweisung dar, nämlich die, der Variablen x einen Wert zuzuweisen. Die rechte Seite dieser Zuweisung, nämlich 5+3, ist dagegen ein Ausdruck, der für den Wert 8 steht. Man beachte in diesem Zusammenhang den Unterschied zwischen „=“, das immer Teil einer Zuweisung (also: eines Kommandos) ist und „==“, das einen Vergleich darstellt (also einen Wahrheitswert zurückliefert) und folglich immer Teil eines Ausdrucks ist: Der Python-Code 5==3 ist also ein Ausdruck, der für den Wert *False* steht.

Aufgabe A.5

Viele Anweisungen enthalten Ausdrücke als Komponenten. Gibt es auch Ausdrücke, die Anweisungen als Komponenten enthalten?

In der interaktiven Pythonshell kann der Programmierer sowohl Anweisungen als auch Ausdrücke eingeben. Die Pythonshell geht aber jeweils unterschiedlich mit diesen um: Wird ein Kommando eingegeben, so führt die Pythonshell das Kommando aus. Wird dagegen ein Ausdruck eingegeben, so wird der Ausdruck zunächst (falls nötig) ausgewertet und anschließend die String-Repräsentation des Ausdrucks ausgegeben.

[1] Spricht man in der Softwaretechnik von Wartbarkeit, so an meint man damit i. A. die Einfachheit ein Programm im nachhinein anzupassen oder zu erweitern. Je übersichtlicher und besser strukturiert ein Programm bzw. Softwaresystem ist, desto besser wartbar ist es.

if-Ausdrücke. Neben der in Abschnitt A.3.1 vorgestellten **if**-Anweisung bietet Python auch die Möglichkeit Ausdrücke mit **if** zu strukturieren:

⟨*expr*$_1$⟩ **if** ⟨*condition*⟩ **else** ⟨*expr*$_2$⟩	Dieser Ausdruck steht für den Wert des Ausdrucks ⟨*expr*$_1$⟩ falls ⟨*condition*⟩ wahr ist, andernfalls steht dieser **if**-Ausdruck für den Wert des Ausdrucks ⟨*expr*$_2$⟩

Beispiele:

>>> x=3 ; y=4
>>> `'a'` **if** x+1==y **else** `'b'`
a
	Da x+1==y wahr ist, steht der **if**-Ausdruck in der zweiten Zeile für den Wert `'a'`.

>>> x=3 ; y=4
>>> `'Hallo Welt'`[7 **if** x==y **else** 4]
o
	Der als String-Index verwendete **if**-Ausdruck steht – da $x \neq y$ – für den Wert 4; der gesamte Ausdruck ergibt also als Wert das (von Null an gezählte) vierte Zeichen des Strings `'Hallo Welt'`, also `'o'`.

Aufgabe A.6

Welchen Wert haben die folgenden Python-Ausdrücke:

(a) `'Hallo'`[4 **if** (4 **if** 4==2 **else** 3)==3 **else** 5]

(b) `'Hallo'` +`'welt'` **if** *str*(2 -1)==*str*(1) **else** `'Welt'`

(c) [0 **if** i%3==0 **else** 1 **for** i **in** *range*(1,15)]

A.3.4 Funktionen

Komplexere Programme sollte man in kleinere Programmeinheiten aufteilen und diese dann zusammenfügen. Die gängigste Möglichkeit, ein Programm in einfachere Teile aufzuteilen, sind Funktionen; jede Funktion löst eine einfache Teilaufgabe und am Ende werden die Funktionen dann entsprechend kombiniert (beispielsweise durch Hintereinanderausführung).

Funktionsdefinitionen. In Python leitet man eine Funktionsdefinition mit dem Schlüsselwort **def** ein:

def ⟨*bez*⟩(⟨*p1*⟩, ⟨*p2*⟩, ...): ⟨*kommando1*⟩ ⟨*kommando2*⟩ ... [**return** ⟨*ausdruck*⟩]	Definiert eine Funktion mit Namen ⟨*bez*⟩, die mit den Paramtern ⟨*p1*⟩, ⟨*p2*⟩, ... aufgerufen wird. Ein Funktionsaufruf führt dann die im Funktions„körper“ stehenden Kommandos ⟨*kommando1*⟩, ⟨*kommando2*⟩, ... aus. Mit dem **return**-Kommando wird die Funktion verlassen und der auf **return** folgende Ausdruck als Wert der Funktion zurückgeliefert. Enthält der Funktionskörper kein **return**-Kommando, so liefert die Funktion den Wert „*None*“ zurück.

Beispiele:

def *getMax*(*a*,*b*): **if** *a* > *b*: **return** *a* **else**: **return** *b* >>> *getMax*('hallo','welt') 'welt'	Die Funktion *getMax* erwartet zwei Parameter *a* und *b* und liefert mittels **return** den größeren der beiden Werte zurück. Die letzten beiden Zeilen zeigen eine Anwendung der Funktion *getMax* in Pythons interaktiver Shell.

Es gibt eine weitere Möglichkeit der Parameterübergabe über sog. *benannte Parameter* . Die Übergabe eines benannten Parameters erfolgt nicht (wie bei Standard-Parametern) über eine festgelegte Position in der Parameterliste, sondern über einen Namen. Bei der Funktionsdefinition muss immer ein default-Wert für einen benannten Parameter spezifiziert werden. Die im Folgenden definierte Funktion *incr* erwartet einen Parameter *x* und optional einen benannten Parameter *increment*, der – falls nicht explizit spezifizert – den Wert „1“ besitzt.

```
>>> def incr(x,increment=1):
...     return x+ increment
...
>>> incr(4)
5
>>> incr(4,increment=10)
14
```

Übrigens müssen benannte Parameter immer rechts der Standardparameter aufgeführt sein; ein Aufruf *incr*(*increment*= -2,4) wäre also syntaktisch nicht korrekt.

Lokale Variablen. Alle in einer Funktion verwendeten Variablen sind *lokal*, d. h außerhalb der Funktion weder sichtbar noch verwendbar und nur innerhalb der Funktion gültig. Weist man einer bestimmten Variablen, die es im Hauptprogramm bzw. aufrufenden Programm schon gibt, einen Wert zu, so wird die Hauptprogramm-Variable dadurch weder gelöscht noch verändert; in der Funktion arbeitet man auf einer Kopie, die von der Variablen des Hauptprogramms entkoppelt ist. Lässt man beispielsweise den Code auf der linken Seite durch Python ausführen, so ergibt sich die auf der rechten

Seite gezeigte Ausgabe:

```
>>>   print 'f: x ist',x
>>>   x=2
>>>
print 'f: lokales x ist',x      erzeugt      f: x ist 50
>>>                          ⟹               f: lokales x ist 2
>>> x=50                                     x ist noch 50
>>> f(x)
>>> print 'x ist noch', x
```

Solange x kein neuer Wert zugewiesen wurde, wird das x aus dem Hauptprogramm verwendet; erst nach der Zuweisung wird ein „neues“ lokales x in der Funktion verwendet, die vom x des Hauptprogramms abgekoppelt ist; außerdem wird sichergestellt, dass das x des Hauptprogramms nicht überschrieben wird und nach dem Funktionsaufruf wieder verfügbar ist.

A.3.5 Referenzen

Eine Zuweisung wie

$x = y$

bewirkt im Allgemeinen nicht, dass eine neue Kopie eines Objektes y angelegt wird, sondern nur, dass x auf den Teil des Hauptspeichers zeigt, an dem sich y befindet. Normalerweise braucht sich der Programmierer darüber keine Gedanken zu machen; ist man sich dieser Tatsache jedoch nicht bewusst, kann es zu Überraschungen kommen. Ein einfaches Beispiel:

```
>>> a = [1,2,3]
>>> b = a
>>> a.append(5)
>>> b
[1,2,3,5]
```

Dass a und b tatsächlich auf den gleichen Speicherbereich zeigen, zeigt sich durch Verwendung der Funktion *id*: $id(x)$ liefert die Hauptspeicheradresse des Objektes x zurück. Für obiges Beispiel gilt:

```
>>> id(a) == id(b)
True
```

Will man, dass `b` eine tatsächliche Kopie der Liste `a` enthält und nicht nur, wie oben, einen weiteren Zeiger auf die gleiche Liste, dann kann man dies folgendermaßen angeben:

```
>>> b = a[:]
```

Dabei ist in obigem Fall $a[:]$ genau dasselbe wie $a[0:2]$ und bewirkt eine Kopie der Liste.

Aufgabe A.7

Was ist der Wert der Variablen a, b und c nach der Eingabe der folgenden Kommandos in den Python-Interpreter:

```
>>> a = ['a','ab','abc']
>>> b = a
>>> b.append('abcd')
>>> c = b[:]
>>> c[0] = '0'
```

A.4 Zusammengesetzte Datentypen

Python besitzt mehrere zusammengesetzte Datentypen, darunter Strings (*str*), Listen (*list*), Tupel (*tuple*), Mengen (*set*) und sog. *Dictionaries* (*dict*), das sind Mengen von Schlüssel-Wert-Paaren, die einen schnellen Zugriff auf die Werte über die entsprechenden Schlüssel erlauben. Strings, Listen, Tupel, Mengen und sogar Dictionaries sind *iterierbar*, d. h. man kann sie etwa mittels **for**-Schleifen durchlaufen.

Mittels der Funktionen *list*(*s*), *tuple*(*s*) und *set*(*s*) kann eine beliebige Sequenz *s* in eine Sequenz vom Typ „Liste“, „Tupel“ bzw. „Set“ überführt werden. Im Folgenden einige Beispiele:

```
>>> list ((1,2,3))
[1, 2, 3]
```

```
>>> tuple(range(10,15))
(10, 11, 12, 13, 14)
```

```
>>> set(range(5))
set([0, 1, 2, 3, 4])
```

A.4.1 Listen

Python-Listen sind Sequenzen von durch Kommata getrennten Werten, eingeschlossen in eckigen Klammern. Listen können Werte verschiedener Typen enthalten, unter Anderem können Listen wiederum Listen enthalten; Listen können also beliebig geschachtelt werden. Folgende Python-Werte sind beispielsweise Listen:

[] (die leere Liste), [5,3,10,23], ['spam', [1,2,3], 3.14, [[1],[[2]]]]

Listenmethoden. Folgende Auflistung zeigt eine Auswahl der wichtigsten Methoden zur Manipulation von Listen. Alle hier gezeigten Methoden – (mit Ausnahme von *count*()) – manipulieren eine Liste destruktiv und erzeugen keinen Rückgabewert.

l.*append*(*x*)	Fügt *x* am Ende der Liste *l* ein. Man beachte, dass *append* ein reines Kommando darstellt, keinen Rückgabewert liefert, sondern lediglich die Liste *l* verändert.

Beispiel:

$l = range(3)$ $l.append($`'last'`$)$	Die Liste l hat nach Ausführung dieser beiden Kommandos den Wert [0,1,2, `'last'`]

$l.sort()$	Sortiert $\langle liste \rangle$ aufsteigend. Auch *sort* ist ein reines Kommando, liefert also keinen Rückgabewert sondern verändert lediglich die Liste l.

Beispiel:

$l = [4,10,3,14,22]$ $l.sort()$	Die Liste l hat nach Ausführung dieser beiden Kommandos den Wert [3, 4, 10, 14, 22].

$l.reverse()$	Dreht die Reihenfolge der Listenelemente um. Auch *reverse* ist ein reines Kommando und liefert keinen Rückgabewert.

Beispiel:

$l = list($`'hallo'`$)$ $l.reverse()$	Die Liste l hat nach Ausführung dieser beiden Kommandos den Wert [`'o'`, `'l'`, `'l'`, `'a'`, `'h'`]

$l.insert(i,x)$	Fügt ein neues Element x an Stelle i in der Liste l ein. Die Zuweisung $l[i:i] = [x]$ hätte übrigens genau den selben Effekt.

Beispiel:

$l = range(6)$ $l.insert(2,$`'neu'`$)$	Die Liste l hat nach Ausführung dieser beiden Kommandos den Wert [0, 1, `'neu'`, 2, 3, 4, 5]

$l.count(x)$ $l.remove()$	Gibt die Anzahl der Vorkommen von x in $\langle l \rangle$ zurück. Löscht das erste Auftreten von x in der Liste l.

Beispiel:

$l = range(3)$ `+ \` $\quad range(3)[::-1]$ $l.remove(1)$	Die Liste l hat nach Ausführung dieser beiden Kommandos den Wert [0, 2, 2, 1, 0].

Man kann sich alle Methoden des Datentyps *list* mit Hilfe der Pythonfunktion *dir* ausgeben lassen. Der Aufruf

```
>>> dir( list )
[ ... , 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', ...]
```

liefert eine Stringliste aller Methodennamen zurück, die für den Datentyp *list* definiert sind.

Aufgabe A.8

Geben Sie in der Python-Shell den Ausdruck

$$[1,2,3].remove(1)$$

ein. Was wird zurückgeliefert? Erklären Sie das Ergebnis!

Aufgabe A.9

Geben Sie ein möglichst kurzes Pythonkommando / Pythonskript an, das ...

(a) ... die Anzahl der für den Datentyp *dict* definierten Operationen ausgibt.

(b) ... die Anzahl der für den Datentyp *list* definierten Operationen ausgibt, die mit `'c'` beginnen.

(c) ... die Länge des längsten Operationsnamens der auf dem Datentyp *list* definierten Operationen ausgibt. Hinweis: für diese Aufgabe wäre die Pythonfunktion *map* gut geeignet, die wir zwar noch nicht behandelt haben, über die Sie sich aber mittels *help*(*map*) informieren können.

A.4.2 Sequenzen

Listen, Tupel und Strings sind sog. Sequenz-Typen, d. h. die enthaltenen Werte besitzen eine feste Anordnung. Dies ist sowohl beim *set*-Typ als auch bei Dictionaries nicht der Fall: In welcher Reihenfolge sich die Elemente einer Menge befinden wird nicht gespeichert; ebenso ist die Anordnung der in einem Dictionary enthaltenen Schlüssel-Wert-Paare nicht relevant.

Slicing. Sei S eine Variable, die ein Sequenz-Objekt enthält – also etwa einen String, eine Liste oder ein Tupel. Dann sind die folgenden Zugriffsoperationen auf S anwendbar.

$S[i]$	**Indizierung** Selektiert Einträge an einer bestimmten Position. Negative Indizes zählen dabei vom Ende her.

Beispiele:

$S[0]$	liefert das erste Element der Sequenz S
$S[-2]$	liefert das zweitletzte Element der Sequenz S
`['ab','xy'][-1][0]`	liefert `'x'` zurück.

	Slicing (**Teilbereichsbildung**)
$S[i:j]$	Selektiert einen zusammenhängenden Bereich einer Sequenz; die Selektion erfolgt von einschließlich Index i bis ausschließlich Index j.
$S[:j]$	die Selektion erfolgt vom ersten Element der Sequenz bis ausschließlich Index j
$S[i:]$	die Selektion erfolgt vom einschließlich Index i bis zum letzten Element der Sequenz.

Beispiele:

$S[1:5]$	selektiert den zusammenhängenden Bereich aller Elemente ab einschließlich Index 1 bis ausschließlich Index 5
$S[3:]$	selektiert alle Elemente von S ab Index 3
$S[:-1]$	selektiert alle Elemente von S, bis auf das letzte
$S[:]$	selektiert alles, vom ersten bis zum letzten Element

	Extended Slicing
$S[i:j:k]$	Durch k kann eine Schrittweite vorgegeben werden.

Beispiele:

$S[::2]$	selektiert jedes zweite Element
$S[::-1]$	selektiert alle Elemente von S in umgekehrter Reihenfolge
$S[4:1:-1]$	selektiert die Elemente von rechts nach links ab Position 4 bis ausschließlich 1.
`'Welt'[::-1]`	ergibt `'tleW'`
`'hallo welt'[-2::-2]`	ergibt `'lwolh'`
$range(51)[::-10]$	ergibt [50, 40, 30, 20, 10, 0]

Handelt es sich bei der Sequenz um eine Liste, so kann – da Listen ja veränderliche Objekte sind – auch eine Zuweisung über Slicing erfolgen. Es folgen zwei Beispiele, wie Teile von Listen mittels Zuweisungen verändert werden können.

```
>>> l = range(7)
>>> l[2:5] = ['x']*3
>>> l
[0, 1, 'x', 'x', 'x', 5, 6]
```

```
>>> l = ['x']*6
>>> l[::2]=[0]*3
>>> l
[0, 'x', 0, 'x', 0, 'x']
```

```
>>> l = range(7)
>>> l[-3::-1]=range(5)
>>> l
[4, 3, 2, 1, 0, 5, 6]
```

Funktionen auf Sequenzen. Folgende Funktionen sind auf alle Sequenzen anwendbar; die meisten der hier aufgeführten Funktionen liefern Rückgabewerte zurück.

len(*S*)	Liefert die Länge der Sequenz *S* zurück.

Beispiele:

len(`'hallo'`)	Liefert die Länge des String zurück, nämlich 5.
len([1, [2,3]])	Liefert die Länge der Liste zurück, nämlich 2.

min(*S*) *max*(*S*)	Liefert das minimale Element der Sequenz *S* zurück. Liefert das maximale Element der Sequenz *S* zurück.

Beispiele:

max(`'hallo'`)	Liefert die maximale Element des Strings, nämlich `'o'` zurück.
max([101,123,99])	Liefert die Zahl 123 zurück.

sum(*S*)	Liefert die Summe der Elemente der Sequenz *S* zurück.

Beispiele:

sum(*range*((100))	Berechnet $\sum_{i=0}^{99}$ und liefert entsprechend 4950 zurück.

del *S*[*i*] **del** *S*[*i* :*j* :*k*]	Löscht einen Eintrag einer Sequenz. **del** kann auch mit Slicing und Extended Slicing verwendet werden. **del** kann man nur auf veränderliche Sequenzen anwenden.

Beispiele:

l = *range*(10) **del** *l* [::2]	Löscht jedes zweite Element der Liste; *l* hat also nach Ausführung der beiden Kommandos den Wert [1, 3, 5, 7, 9].

Aufgabe A.10

Bestimmen Sie den Wert der folgenden Ausdrücke:

(a) $range(1,100)[1], range(1,100)[2]$

(b) $[range(1,10),\ range(10,20)][1][2]$

(c) `['Hello',2,'World'][0][2] + ['Hello',2,'World'][0]`

(d) $len(range(1,100))$

(e) $len(range(100,200)[0:50:2])$

Hinweis: Versuchen Sie zumächst die Lösung ohne die Hilfe des Pythoninterpreters zu bestimmen.

Aufgabe A.11

Wie können Sie in folgendem Ausdruck (der eine verschachtelte Liste darstellt)

$$[[x],[[[y]]]]$$

auf den Wert von y zugreifen?

Aufgabe A.12

Lösen sie die folgenden Aufgaben durch einen Python-Einzeiler:

(a) Erzeugen Sie die Liste aller geraden Zahlen zwischen 1 und 20.

(b) Erzeugen Sie die Liste aller durch 5 teilbarer Zahlen zwischen 0 und 100.

(c) Erzeugen Sie die Liste aller durch 7 teilbarer Zahlen zwischen 0 und 100; die Liste soll dabei umgekehrt sortiert sein, d. h. die größten Elemente sollen am Listenanfang und die kleinsten Elemente am Listenende stehen.

A.4.3 Tupel

Tupel sind Listen ähnlich, jedoch sind Tupel – wie auch Strings – unveränderlich. Tupel werden in normalen runden Klammern notiert. Tupel können genauso wie andere Sequenzen auch indiziert werden. Es folgen einige Beispiele:

```
>>> x = ('Das', 'ist', 'ein', 'Tupel')
>>> x[1]
'ist'
>>> x[2][0]
'e'
>>> x[0] = 'Hier'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  TypeError: 'tuple' object does not support item assignment
```

Die letzte Zuweisung ist aufgrund der Unveränderlichkeit von Tupeln verboten. Will man in der Variablen x ein Tupelobjekt speichern, dessen erste Position den Wert `'Hier'` enthält und das ansonsten mit dem „alten“ x identisch ist, so muss man wie folgt vorgehen:

```
>>> x = ('Hier',) + x[1:]
>>> x
('Hier', 'ist', 'ein', 'Tupel')
```

Man beachte: Durch die Zuweisung in der ersten Zeile wurde kein Tupel-Objekt verändert, sondern ein neues Tupel-Objekt erzeugt, durch Konkatenation des ein-elementigen Tupels `('Hier',)` mit dem drei-elementigen Tupel $x[1:]$.

A.4.4 Dictionaries

Ein Dictionary-Objekt stellt eine effiziente Repräsentation einer Zuordnung von Schlüsseln auf Werte dar. Ein Anwendungsbeispiel ist ein Adressbuch, das bestimmte Namen (die Schlüssel) auf Adressen (die Werte) abbildet. Ein Dictionary-Objekt sollte die folgenden drei Operationen effizient unterstützen: **1.** Das Einfügen eines neuen Wertes v mit dem Schlüssel k. **2.** Das Finden eines bestimmten Wertes v anhand seines Schlüssels k. **3.** Das Löschen eines Schlüssels k zusammen mit dem zugehörigen Wert v.

Aufgrund der Tatsache, dass der Informatiker eine effiziente Unterstützung der Dictionary-Operationen häufig benötigt, bietet Python einen eigenen internen Typ *dict* an, der diese Operationen effizient unterstützt. Während Listen in eckigen Klammern und Tupel in runden Klammern notiert werden, werden Dictionaries in geschweiften Klammern geschrieben:

$$\{\ \langle schlüssel1\rangle\ :\langle wert1\rangle,\ \langle schlüssel2\rangle\ :\langle wert2\rangle,\ \ldots\}$$

Ein einfaches Beispiel:

```
>>> ab = { 'Carlo'  : 'carlo@web.de',
           'Hannes' : 'hannes@gmail.de',
           'Matilda' : 'matilda@gmx.de' }
```

Die Operationen „Einfügen“ und „Suchen“ werden über den Indizierungsoperator [...] angesprochen, so dass sich die Verwendung eines Dictionary-Objektes z. T. „anfühlt“ wie ein Listen- oder Tupelobjekt. Beispiele:

```
>>> ab['Hannes']
'hannes@gmail.de'
>>> ab['Hannes']='hannes@gmx.de'
>>> ab['Hannes']
'hannes@gmx.de'
```

Die Löschfunktion ist über die Funktion **del** implementiert.

```
>>> del ab['Matilda']
>>> print 'Es gibt',len(ab),'Eintraege in ab'
'Es gibt 2 Eintraege in ab'
```

Man kann also, genau wie bei anderen veränderbaren Sequenzen, auf einzelne Elemente zugreifen, löschen und alle für Sequenzen definierte Funktionen anwenden. Wichtig zu wissen ist, dass man nur unveränderliche Werte als Schlüssel verwenden kann – also insbesondere *keine* Listen!

Aufgabe A.13

Erklären Sie, was das Problem wäre, wenn man auch veränderliche Werte (wie beispielsweise Listen) als Schlüssel in Dictionaries zulassen würde.

Die Schlüssel müssen nicht alle den gleichen Typ haben:

```
>>> ab[(1,2,3)] = 123
>>> ab[1] = 100
>>> ab[(1,2,3)]-ab[1]
23
```

Methoden auf Dictionaries. Die folgenden Methoden auf Dictionaries werden von einigen der vorgestellten Algorithmen verwendet:

d.values()	Liefert eine Liste aller in *d* enthaltenen Werte zurück.
d.keys()	Liefert eine Liste aller in *d* enthaltenen Schlüssel zurück.
d.items()	Liefert alle in *d* enthaltenen Schlüssel-Werte-Paare als Tupel-Liste zurück.

Als Beispiele nehmen wir an, ein Dictionary *d* sei folgendermaßen definiert:

```
>>> d = {1:'hallo', 'welt':[1,2,3], ('x','y'):10, '20':'30', 2:{1:[], 2:[2]}, 3:[]}
```

Dann gilt beispielsweise:

```
>>> d.keys()
>>> [1, 2, 3, '20',
    ('x', 'y'), 'welt']
```

```
>>> d.values()
>>> ['hallo', {1: [], 2: [2]}, [],
    '30', 10, [1, 2, 3]]
```

```
>>> d[2].keys()
[1, 2]
```

A.4.5 Strings (Fortsetzung)

Häufig gebraucht, sowohl für große Programmierprojekte als auch für viele kleine nützliche Skripts, sind Funktionen auf Strings.

Strings sind – ebenso wie Listen und Tupel – Sequenzen und entsprechend sind alle im vorigen Abschnitt beschriebenen Sequenzoperationen anwendbar. Strings sind, ebenso wie Tupel, unveränderlich, d. h. ein einmal definierter String kann nicht verändert werden. Man kann also weder einzelne Zeichen aus einem einmal erstellten String herauslöschen, noch kann man an einen einmal definierten String Zeichen anfügen.

Es folgt eine Liste der wichtigsten String-Operationen:

Suchen

s. find(*s1*)	Liefert den Offset des ersten Vorkommens von *s1* in *s* zurück.
s. replace(*s1*,*s2*)	Liefert einen String zurück, in dem alle Vorkommen von *s1* durch *s2* ersetzt sind.
s. startswith(*s1*)	Liefert *True* zurück, falls *s* mit *s1* beginnt.
s. endswith(*s1*)	Liefert *True* zurück, falls *s* mit *s1* endet.

Als Beispiel nehmen wir an, ein String *s* sei folgendermaßen definiert:

```
>>> s = 'Hallo Welt, dies, genau dies, ist ein Teststring'
```

```
>>> s.find('s,')
15
```

```
>>> s.replace('dies','das')
'Hallo Welt, das, genau
 das, ist ein Teststring'
```

```
>>> s.startswith('Ha')
True
```

Aufteilen, Zusammenfügen

s. split (*s1*)	Gibt eine Liste von Wörtern von *s* zurück, mit *s1* als Trenner.
s. partition (*sep*)	Sucht nach dem Trenner *sep* in *s* und liefert ein 3-Tupel (*head*,*sep*, *tail*) zurück, wobei *head* der Teil vor *sep* und *tail* der Teil nach *sep* ist.
s. join(*l*)	Verkettet die Stringliste *l* zu einem einzigen String mit *s* als Trenner.

Beispiele:

```
>>> 'Hi hi you foo'.split()
['Hi', 'hi', 'you', 'foo']
```

```
>>> '1. Zwei. 3.'.split('.')
['1', ' Zwei', ' 3', '']
```

```
>>> ','.join([
...  'a','b','c'])
'a,b,c'
```

Aufgabe A.14

Schreiben Sie eine Pythonfunktion *zipString*, die zwei Strings als Argumente übergeben bekommt und einen String zurückliefert, der eine „verschränkte“ Kombination der beiden übergebenen Strings ist.

Beispielanwendungen:

```
>>> zipString('Hello','World')
'HWeolrllod'
>>> zipString('Bla','123')
'B1l2a3'
```

A.4.6 Mengen: Der *set*-Typ

Einige Algorithmen benötigen duplikatfreie Sammlungen von Werten. Hier bietet sich Pythons *set*-Datentyp an. Etwa der in Abschnitt 6.4 beschriebene LR-Parsergenerator verwendet *set*-Objekte zur Repräsentaton und Berechnung von FIRST- und FOLLOW-Mengen.

set-Objekte können aus Sequenzen (wie Listen, Tupel oder Strings) mittels der Konstruktor-Funktion *set*() erzeugt werden. Beispielsweise erzeugt folgende Anweisung

$s = set(range(3))$

eine Menge, die die Zahlen „0“, „1“ und „2“ enthält.

Es folgt eine Liste der wichtigsten Methoden auf Mengen:

Einfügen, Löschen

$s.add(x)$	Fügt ein Element x in eine Menge s ein. Befindet sich der Wert x bereits in der Menge s, so bleibt s durch dieses Kommando unverändert. Die Methode *add* ist ein reines Kommando und liefert keinen Rückgabewert.
$s.remove(x)$	Löscht ein Element x aus der Menge s. Das Element x muss in der Menge s enthalten sein – anderfalls entsteht ein `KeyError`. Auch die Methode *remove* ist ein reines Kommando und liefert keinen Rückgabewert.

Beispiele (wir gehen davon aus, die Menge *s* sei jeweils durch *s=set(range(3))* definiert):

```
>>> s.add(10)
>>> s
set([0, 1, 2, 10])
```

```
>>> s.add(2)
>>> s
set([0, 1, 2])
```

```
>>> s.remove(0)
>>> s
set([1, 2])
```

```
>>> s.remove(6)
KeyError: 6
```

Vereinigung, Schnitt

s.union(s1)	Liefert die Vereinigung „$s \cup s1$“ zurück. Es wird also ein *set*-Objekt zurückgeliefert, das alle Elemente enthält, die sich entweder in *s* oder in *s1* befinden. Die *union*-Methode ist rein funktional und lässt sowohl *s* als auch *s1* unverändert.
s. intersection (s1)	Liefert den Schnitt „$s \cap s1$“ zurück. Es wird also ein *set*-Objekt zurückgeliefert, das alle Elemente enthält, die sich sowohl in *s* als auch in *s1* befinden. auch die *intersection*-Methode verändert die Parameter nicht.
s. difference (s1)	Liefert die Mengendifferenz „$s \backslash s1$“ zurück. Es wird also ein *set*-Objekt zurückgeliefert, das alle Elemente aus *s* enthält, die nicht in *s1* enthalten sind. Auch die *difference*-Methode verändert die Parameter nicht.

Wir geben einige Beispiele an und gehen dabei davon aus, dass die folgenden beiden Definitionen

```
>>> s=set('hallo welt')
>>> s1=set('hello world')
```

voranstehen:

```
>>> s.union(s1)
set(['a',' ','e','d','h',
     'l','o','r','t','w'])
```

```
>>> s. intersection (s1)
set([' ','e','h','l','o','w'])
```

```
>>> s. difference (s1)
set(['a', 't'])
```

A.5 Funktionale Programmierung

Das Paradigma der *Funktionalen Programmierung* unterscheidet sich vom Paradigma der imperativen Programmierung vor allem dadurch, dass imperativen Programme überwiegend *Anweisungen* verwenden. Eine Anweisung „tut“ etwas, d. h. die verändert den Zustand des Programms bzw. des Speichers bzw. den Zustand von Peripheriegeräten (wie etwa des Bildschirms). Auch **for**- oder **while**-Schleifen sind typische Anweisungen: In jedem Schleifendurchlauf verändert sich i. A. der Zustand der Schleifenvariablen.

Funktionale Programme verwenden nur oder überwiegend Ausdrücke, die strenggenommen nichts „tun“, sondern lediglich für einen bestimmten Wert stehen und kei-

ne Zustände verändern. Viele Programmierfehler entstehen, da der Programmierer den Überblick über die durch das Programm erzeugten Zustände verloren hat. Programmiert man mehr mit Ausdrücken, so schließt man zumindest diese Fehlerquelle aus. Beispielsweise lohnt es sich immer in Erwägung zu ziehen, eine „imperative“ Schleife durch eine Listenkomprehension, eine *map*-Anweisung oder eine *filter* -Anweisung zu ersetzen.

A.5.1 Listenkomprehensionen

Listenkomprehensionen sind Ausdrücke, keine Kommandos – sie stehen also für einen bestimmten Wert. Man kann Listenkomprehensionen als das funktionale Pendant zur imperativen Schleife betrachten. Sie sind insbesondere für Mathematiker interessant und leicht verständlich aufgrund ihrer an die mathematische Mengenkomprehension angelehnte Notation. Die Menge

$$\{2 \cdot x \mid x \in \{1, \ldots 20\},\ x \text{ durch 3 teilbar }\}$$

entspricht hierbei der Python-Liste(nkomprehension)

$$[\ 2{*}x\ \textbf{for}\ x\ \textbf{in}\ \mathit{range}(1{,}21)\ \ \textbf{if}\ x\%3{==}0\ ]$$

Jede Listenkomprehension besteht mindestens aus einem in eckigen Klammern [...] eingeschlossenen Ausdruck, gefolgt von einer oder mehreren sogenannten **for**-Klauseln. Jede **for**-Klausel kann optional durch eine **if**-Klausel eingeschränkt werden.

$$\begin{array}{l}[\langle \mathit{ausdr}\rangle\ \textbf{for}\ \langle \mathit{ausdr1}\rangle\ \textbf{in}\ \langle \mathit{sequenz1}\rangle\ [\textbf{if}\ \langle \mathit{bedingung1}\rangle] \\ \qquad\qquad \textbf{for}\ \langle \mathit{ausdr1}\rangle\ \textbf{in}\ \langle \mathit{sequenz2}\rangle\ [\textbf{if}\ \langle \mathit{bedingung2}\rangle]\ \ldots\ ]\end{array}$$

Der Bedingungsausdruck dieser **if**-Klauseln hängt i. A. ab von einer (oder mehrerer) durch vorangegangene **for**-Klauseln gebundenen Variablen. Dieser Bedingungsausdruck „filtert“ all diejenigen Ausdrücke der jeweiligen Sequenz aus für die er den Wahrheitswert „*False*“ liefert.

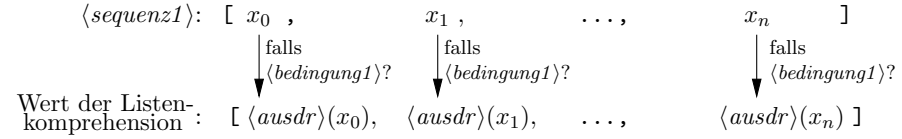


Abb. A.1: *Funktionsweise einer Listenkomprehension mit einer* ***for****-Schleife und einer* ***if****-Bedingung. Die Ausdrücke ⟨sequenz1⟩,⟨bedingung1⟩ und ⟨ausdr⟩ beziehen sich hier auf die entsprechenden Platzhalter, die in obiger Syntaxbeschreibung verwendet wurden. Wie man sieht, ist der Wert der Listenkomprehension* immer *eine Liste, deren Elemente durch Anwendung von ⟨ausdr⟩ auf die einzelnen Elemente der Liste ⟨sequenz1⟩ entstehen.*

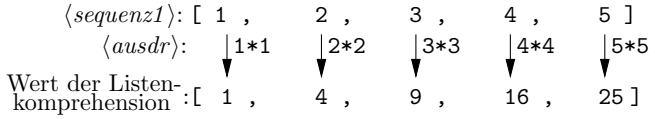
Beispiele

Wir gehen in vielen der präsentierten Beispiel darauf ein, welchen Wert die einzelnen Platzhalter der obigen Syntaxbeschreibung haben, d. h. wir geben oft der Klarheit halber an, was der jeweilige „Wert“ der Platzhalter ⟨*ausdr*⟩, ⟨*ausdr1*⟩, ⟨*sequenz1*⟩, ⟨*bedingung1*⟩, usw. ist.

i) Die Liste aller Quadratzahlen von 1^2 bis 5^2:

```
>>> [x*x for x in range(1,6) ]
[1, 4, 9, 16, 25]
```

⟨*ausdr*⟩ entspricht hier dem Ausdruck $x*x$; ⟨*sequenz1*⟩ entspricht *range*(1,6). Für jeden Wert in *range*(1,6), also für jeden Wert in [1,2,3,4,5], wird ein Listeneintrag der Ergebnisliste durch Auswertung des Ausdrucks $x*x$ erzeugt. Ergebnis ist also [1*1, 2*2, ...]. Die folgende Abbildung veranschaulicht dies nochmals:



ii) Die Liste aller durch 3 oder durch 7 teilbarer Zahlen zwischen 1 und 20:

```
>>> [x for x in range(1,20)
...          if x%3==0 or x%7==0 ]
[3, 6, 7, 9, 12, 14, 15, 18]
```

⟨*ausdr*⟩ entspricht hier dem nur aus einer Variablen bestehenden Ausdruck x; ⟨*sequenz1*⟩ entspricht *range*(1,20); ⟨*bedingung1*⟩ entspricht $x\%3==0$ **or** $x\%7==0$. Hier wird also eine Liste erzeugt die aus allen x in *range*(1,20) besteht für die die **if**-Bedingung *True* ergibt.

Aufgabe A.15

(a) Schreiben Sie eine Pythonfunktion *teiler*(*n*), die die Liste aller Teiler einer als Parameter übergebenen Zahl *n* zurückliefert. Tipp: Am leichtesten mit Verwendung einer Listenkomprehension. Beispielanwendung:

```
>>> teiler(45)
>>> [1, 3, 5, 9, 15]
```

(b) Geben Sie – mit Verwendung der eben geschriebenen Funktion *teiler* – einen Python-Ausdruck (kein Kommando!) an, der eine Liste aller Zahlen zwischen 1 und 1000 ermittelt, die genau 5 Teiler besitzen.

(c) Geben Sie – mit Verwendung der eben geschriebenen Funktion *teiler* – einen Python-Ausdruck an, der die Zahl zwischen 1 und 1000 ermittelt, die die meisten Teiler besitzt.

iii) Die Liste aller möglichen Tupel von Zahlen aus 1 bis 10.

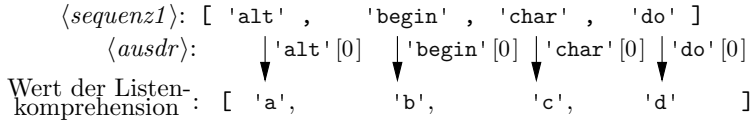
```
>>> [ (x,y) for x in range(1,10)
...          for y in range(1,10)]
[(1, 1), (1, 2), ... ,(1,9), (2,1), (2,2), ...
(9, 9)]
```

Der Platzhalter ⟨*ausdr*⟩ entspricht in diesem Fall dem Tupel (x, y), der Platzhalter ⟨*sequenz1*⟩ entspricht *range*(1,10) und der Platzhalter ⟨*sequenz2*⟩ entspricht *range*(1,10). Man sieht: Es können beliebig viele **for**-Klauseln hintereinander stehen, was einer Schachtelung von **for**-Schleifen entspricht. Im ersten Durchlauf hat x den Wert 1 und y durchläuft die Zahlen von 1 bis (ausschließlich) 10; im zweiten Durchlauf hat x den Wert 2 und y durchläuft wiederum die Zahlen von 1 bis ausschließlich 10, usw. Jede dieser beiden **for**-Klauseln könnte (auch wenn dies in obigem Beispiel nicht geschieht) ein **if**-Statement verwenden, das die Werte für x bzw. y, die durchgelassen werden, einschränkt.

iv) Die jeweils ersten Zeichen von in einer Liste befindlichen Strings.

```
>>> [x[0] for x in ['alt','begin','char','do']]
['a','b','c','d']
```

Der Platzhalter ⟨*ausdr*⟩ entspricht hier dem Ausdruck $x[0]$ und der Platzhalter ⟨*sequenz1*⟩ entspricht der Stringliste ['alt','begin',...]. Die Schleifenvariable x durchläuft nacheinander die Strings 'alt', 'begin', usw. In jedem Durchlauf wird das erste Zeichen des jeweiligen Strings in die Ergebnisliste eingefügt. Die folgende Abbildung veranschaulicht dies nochmals:



Aufgabe A.16

Gegeben sei ein (evtl. langer) String, der '\n'-Zeichen (also Newline-Zeichen, oder Zeilentrenner-Zeichen) enthält. Geben Sie – evtl. unter Verwendung einer Listenkomprehension – einen Ausdruck an, der ...

(a) ... die Anzahl der Zeilen zurückliefert, die dieser String enthält.
(b) ... alle Zeilen zurückliefert, die weniger als 5 Zeichen enthalten.
(c) ... alle Zeilen zurückliefert, die das Wort 'Gruffelo' enthalten.

A.5.2 Lambda-Ausdrücke

Mittels des Schlüsselworts **lambda** ist es möglich „anonyme“ Funktionen zu definieren – Funktionen also, die keinen festgelegten Namen besitzen, über den sie wiederholt

aufgerufen werden können. Oft werden solche namenslose Funktionen in Funktionen höherer Ordnung – wie etwa *map*, *reduce* oder *filter* – verwendet. Folgende Tabelle beschreibt die Syntax eines Lambda-Ausdrucks.

lambda *x1,x2,...* : *e*	Dieser Lambda-Ausdruck repräsentiert eine Funktion, die die Argumente *x1*, *x2*, ... erwartet und den Ausdruck *e* (der üblicherweise von den Argumenten abhängt) zurückliefert.

Die folgenden beiden Definitionen ergeben genau dieselbe Funktion *add3*:

```
>>> def add3(x,y,z): return x+y+z
```

```
>>> add3 = lambda x,y,z : x+y+z
```

Beide der obigen Definitionen erlauben einen Aufruf wie in folgendem Beispiel gezeigt:

```
>>> add3(1,2,3)
6
```

Das durch den Lambda-Ausdruck erzeugte Funktionsobjekt kann auch sofort ausgewertet werden wie etwa in folgendem Beispiel:

```
>>> (lambda x,y: x*(y-x))(2,5)
6
```

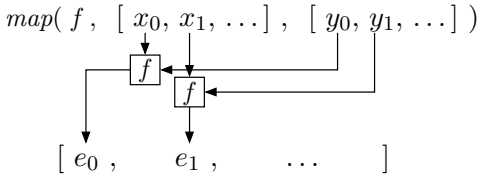
A.5.3 Die *map*-Funktion

Die *map*-Funktion verknüpft mehrere Listen elementweise mit einer als Parameter übergebenen Funktion:

map(*f* , *l1* , *l2*, ...)

Die *map*-Funktion liefert als Ergebnis immer eine Liste zurück. Die *map*-Funktion ruft die Funktion *f* zunächst auf alle ersten Elemente der Listen *l1* , *l2*, ... , auf, anschließend für alle zweiten Elemente, usw. Die Menge der so erhaltenen Werte wird als Liste zurückgeliefert.

Folgendes Beispiel zeigt die Anwendung der *map*-Funktion auf eine zweistellige Funktion *f*; es werden zwei Listen $[x_0, x_1, \ldots]$ und $[y_0, y_1, \ldots]$ elementweise verknüpft und daraus eine neue Liste $[e_0, e_1, \ldots]$ erzeugt:



```
>>> def add(x,y): return x+y
>>> map(add, [1,3,5], [10,100,1000])
[11, 102, 1003]
```

Häufig wird ein Lambda-Ausdruck verwendet, um das als ersten Parameter erwartete Funktionsobjekt zu erzeugen – dies zeigen die folgenden beiden Beispiele:

```
>>> map(lambda x,y:x+y,
...         'Hallo','Welt!')
['HW', 'ae', 'll', 'lt', 'o!']
```

```
>>> map(lambda x,y,z: (x+y)*z,
...      [1,2,3], [4,5,6], range(10,13))
[50, 77, 108]
```

Aufgabe A.17

Verwenden Sie die *map*-Funktion, um einer (String-)Liste von Zeilen Zeilennummern hinzuzufügen. Der Ausdruck:

```
['Erste Zeile', 'Zweite Zeile', 'Und die dritte Zeile']
```

sollte also umgewandelt werden in folgenden Ausdruck:

```
['1. Erste Zeile', '2. Zweite Zeile', '3. Und die dritte Zeile']
```

A.5.4 Die *all*- und die *any*-Funktion

Die *all*-Funktion und die *any*-Funktion verknüpfen eine Menge von Wahrheitswerten mittels einer logischen Und-Verknüpfung bzw. mittels einer logischen Oder-Verknüpfung.:

all(*l*)	Liefert genau dann „*True*“ zurück, wenn alle Elemente des iterierbaren Objektes *l* den Wahrheitswert „*True*“ besitzen.
any(*l*)	Liefert genau dann „*True*“ zurück, wenn mindestens ein Element des iterierbaren Objektes *l* den Wahrheitswert „*True*“ besitzt.

Beispiele:

```
>>> all( [x<10 for x in range(9)])
True
```

```
>>> any(map(str.isdigit ,'124'))
True
```

A.5.5 Die *enumerate*-Funktion

Die *enumerate*-Funktion ist nützlich, wenn man sich nicht nur für die einzelnen Elemente einer Sequenz interessiert, sondern auch für deren Index in der Sequenz.

enumerate(*iter*)	Die *enumerate*-Funktion erhält als Argument eine iterierbares Objekt *iter* und erzeugt daraus als Ergebnis wiederum einen Iterator. Dieser enthält Paare bestehend aus einem Zähler und aus den einzelnen Elementen des als Argument übergebenen Objekts.

Beispiele:

```
>>> enumerate('Hallo')
<enumerate object at ...>
```

```
>>> [x for x in enumerate('Hallo')]
[(0, 'H'), (1, 'a'), (2, 'l'), (3, 'l'), (4, 'o')]
```

A.5.6 Die *reduce*-Funktion

reduce(f, l)	Verknüpft die Elemente einer Liste (bzw. einer Sequenz) nacheinander mit einer zwei-stelligen Funktion. Die Verknüpfung erfolgt von links nach rechts.

Der Aufruf ($\oplus$ stehe hierbei für einen beliebigen binären Operator)

reduce(**lambda** $x,y{:}x \oplus y$, $[x_0, x_1, x_2, \ldots, x_n]$)

liefert also den Wert des Ausdrucks

$$(\cdots(((x_0 \oplus x_1) \oplus x_2) \oplus \ldots) \oplus x_n)$$

zurück.

Wir verwenden die *reduce*-Funktion für die Implementierung von Hashfunktionen in Abschnitt 3.4 und für die Implementierung eines rollenden Hashs in Abschnitt 7.5.

Beispiele. Die folgende Aufzählung gibt einige Anwendungsbeispiele für die Verwendung der *reduce*-Funktion:

i) Aufsummieren aller ungeraden Zahlen von 1 bis 1000.

```
>>> reduce(lambda x,y: x+ y, range(1,1000,2))
250000
```

Berechnet die Summe $(\cdots((1+3)+5)+\ldots+999)$. Die gleiche Berechnung kann man auch mit *sum*(*range*(1,1000,2)) durchführen.

ii) Verknüpfen einer Menge von Strings zu einem String der aus einer Menge von Zeilen besteht.

```
>>> reduce( lambda x,y: x+'\n'+y,
        ['Erste Zeile', 'Zweite Zeile', 'Dritte Zeile'])
'Erste Zeile\nZweite Zeile\nDritte Zeile'
```

Die als erster Parameter übergebene Funktion verkettet zwei Strings mit dem Newline-Zeichen `'\n'` als Trenner. Die *reduce*-Funktion verkettet ebentsprechend alle Strings in der Liste und fügt jeweils ein `'\n'`-Zeichen zwischen zwei Strings ein.

iii) Umwandeln einer als String repräsentierten Hexadezimal-Zahl in einen Python Integerwert unter Verwendung des *Horner-Schemas*:

Angenommen, die hexadezimale Zahl $h_0h_1h_2h_3h_4$ sei gegeben. Will man daraus die entsprechende Dezimalzahl über

$$h_0 * 16^4 + h_1 * 16^3 + h_2 * 16^2 + h_3 * 16^1 + h_4 * 16^0$$

berechnen, so ist dies wenig effizient. Es werden zur Berechnung der Potenzen sehr viele (nämlich 4+3+2) Multiplikationen durchgeführt – und Multiplikationen sind meist sehr rechenintensiv. Die gleiche Berechnung kann folgendermaßen mit weniger Multiplikationen durchgeführt werden:

$$(((h_0 * 16 + h_1) * 16 + h_2) * 16 + h_3) * 16 + h4$$

Dieses Berechnungs-Schema ist das sog. *Horner-Schema*. Eine Implementierung kann elegant mit Hilfe der *reduce*-Funktion erfolgen:

```
>>> hexNum = '12fb3a'
>>> reduce(lambda x,y: 16*x+ y,
            [c2h(h) for h in hexNum])
1243962
```

Wir nehmen an, *c2h* wandelt eine als String repräsentierte hexadezimale Ziffer in einen Zahlenwert um. Die Listenkomprehension [*c2h*(*h*) **for** *h* **in** *hexNum*] erzeugt zunächst eine Liste der Integerwerte, die den einzelnen Ziffern in *hexNum* entsprechen – hier wäre das die Liste [1,2,15,11,3,10]. Die *reduce*-Funktion verknüpft dann die Elemente der Liste mit als Lambda-Ausdruck spezifizierten Funktion und verwendet so das Horner-Schema um die Dezimalrepräsentation der Hexadezimalzahl `'12fb3a'` zu berechnen.

Aufgabe A.18

Verwenden Sie die *reduce*-Funktion, um eine Funktion *max*(*lst*) zu definieren, die das maximale in *lst* befindliche Element zurückliefert.

Aufgabe A.19

Verwenden Sie die *reduce*-Funktion, um eine Liste von Tupeln „flachzuklopfen“ und in eine einfache Liste umzuwandeln. Beispiel: Die Liste [(1,10), ('a','b'), ([1], [2])] sollte etwa in die Liste [1,10,'a','b',[1],[2]] umgewandelt werden.

Aufgabe A.20

Implementieren Sie die Funktionen *any* und *all* mittels der *reduce*-Funktion.

A.6 Vergleichen und Sortieren

Zum Einen beschreibt Abschnitt 2 Sortieralgorithmen, zum Anderen verwenden viele in diesem Buch vorgestellten Algorithmen Sortierfunktionen – etwa einige Heuristiken zur Lösung des Travelling-Salesman-Problems (etwa der in Abschnitt 8.5.3 vorgestellte genetische Algorithmus und der in Abschnitt 8.6 vorgestellte Ameisen-Algorithmus).

A.6.1 Vergleichen

Für viele in diesem Buch vorgestellten Algorithmen ist es wichtig genau zu verstehen, wie Werte in Python verglichen werden. Während intuitiv klar sein dürfte, dass Zahlenwerte einfach ihrer Größe nach verglichen werden, bedarf es einer kurzen Erläuterung was Vergleiche von Werten zusammengesetzter Typen oder Vergleiche von Werten unterschiedlicher Typen betrifft.

Vergleiche mit *None*. Der Wert *None* wird von Python immer als kleiner klassifiziert als jeder andere Wert. Beispiele:

```
>>> None < 0
True
```

```
>>> None < -float('inf')
True
```

```
>>> None < False
True
```

```
>>> None < None
False
```

Anmerkung: Der Python-Wert *float*(`'inf'`) steht für den mathematischen Wert ∞ („Unendlich“), Der Python-Wert `-`*float*(`'inf'`) steht entsprechend für den mathematischen Wert $-\infty$ („Minus Unendlich“).

Vergleiche mit booleschen Werten. Bei Vergleichen mit Booleschen Werten muss man sich lediglich darüber im Klaren sein, dass in Python der boolesche Wert „*False*“ der Zahl „0“ und der boolesche Wert „*True*“ der Zahl „1“ entspricht:

```
>>> False == 0
True
```

```
>>> True == 1
True
```

Vergleiche zwischen booleschen Werten und Zahlen ergeben dementsprechende Ergebnisse. Beispiele:

```
>>> False < True
True
```

```
>>> False < -1
False
```

```
>>> True < 10
True
```

Vergleiche von Sequenzen. Sequenzen sind in Python lexikographisch geordnet, d. h. zwei Sequenzen werden von links nach rechts verglichen; die erste Stelle, die sie unterscheidet, entscheidet darüber, welche der Sequenzen kleiner bzw. größer ist. Dies entspricht genau der Art und Weise, wie Namen in einem Telefonbuch angeordnet sind: Die Namen werden zunächst nach dem linkesten Buchstaben sortiert; besitzen zwei Namen denselben linkesten Buchstaben, so entscheidet der nächste Buchstabe über deren Anordnung, usw.

Beispielsweise gilt

```
>>> 'aachen' < 'aalen'
True
```

da sich die ersten beiden Stellen nicht unterscheiden und `'c' < 'l'` gilt.

Außerdem werden kürzere Sequenzen – bei identischem Präfix – als kleiner klassifiziert. Einige weitere Beispiele für Vergleiche von Sequenzen:

```
>>> [2,100] < [3,1]
True
```

```
>>> [0] < [1]
True
```

```
>>> [0] < [0,0,0]
True
```

```
>>> [] < [0]
True
```

Zahlen werden in Python immer als kleiner klassifiziert als Werte zusammengesetzter Typen. Einige Beispiele:

```
>>> 0 < [0]
True
```

```
>>> [0] < [[0]]
True
```

```
>>> 100 < []
True
```

A.6.2 Sortieren

Python bietet eine destruktive Sortierfunktion *sort* (die keinen Rückgabewert liefert) und eine nicht-destruktive Sortierfunktion *sorted* (die die sortierte Version der Sequenz als Rückgabewert liefert) an. Die Funktion *sort* sortiert in-place, ist also speichereffizienter und schneller als die Funktion *sorted*, die zunächst eine neue Kopie der Sequenz anlegen muss.

Ein Beispiel für die unterschiedliche Funktionsweise von *sort* und *sorted*; in beiden Fällen sei definiert:

$$l = list(\texttt{'Python'})$$

```
>>> sorted(l)
['P', 'h', 'n', 'o', 't', 'y']
```

```
>>> l.sort()
>>> l
['P', 'h', 'n', 'o', 't', 'y']
```

Sortieren nach bestimmten Eigenschaften. Häufig möchte man eine Sequenz von Werten nicht nach der üblichen (i. A. lexikographischen) Ordnung, sondern stattdessen nach einer selbst bestimmten Eigenschaften sortieren. Möchten man etwa eine Liste von Strings (anstatt lexikographisch) der Länge der Strings nach sortieren, so könnte man wie folgt vorgehen: Zunächst „dekoriert“ man die Strings mit der Information, die für die gewünschte Sortierung relevant ist – in diesem Fall würde man also jeden String mit seiner Länge dekorieren und eine Liste von Tupeln der Form $(len(s), s)$ erzeugen. Ein Sortierung dieser Tupelliste bringt das gewünschte Ergebnis: Die Tupel werden nach ihrer erste Komponente (d. h. ihrer Länge nach) sortiert; besitzen zwei Tupel dieselbe erste Komponente (d. h. besitzen die entsprechenden Strings dieselbe Länge), so werden diese nach ihrer zweiten Komponente geordnet, also lexikographisch nach ihrem

Namen. Anschließend müsste man die für die Sortierung relevante „Dekoration“ wieder entfernen. In dieser Weise könnte man etwa folgendermaßen Pythons Stringmethoden ihrer Länge nach sortieren:

```
1 >>> methods = [(len(s),s) for s in dir(str)]
2 >>> methods.sort()
3 >>> methods = [s for l,s in methods]
4 >>> methods
5 ['find', 'join', 'count', 'index', 'ljust', 'lower', 'rfind', 'rjust', ...]
```

(Wir erinnern uns: *dir*(*str*) erzeugt die Liste aller Methoden des Typs *str*.)

Die Dekoration erfolgt durch die Listenkomprehension in Zeile 1, das Entfernen der Dekoration erfolgt durch die Listenkomprehension in Zeile 3.

Pythons Sortierfunktionen bieten die Möglichkeit, sich diese „Dekorationsarbeiten“ abnehmen zu lassen. Den Funktionen *sort* und *sorted* kann man mittels eines sog. *benannten Parameters „key“* eine Funktion übergeben, deren Rückgabewert für die Sortierung verwendet wird. Dadurch kann man Pythons Stringmethoden folgendermaßen ihrer Länge nach sortieren:

```
1 >>> methods = dir(str)
2 >>> methods.sort(key=len)
3 >>> methods
4 ['find', 'join', 'count', 'index', 'ljust', 'lower', 'rfind', 'rjust', ...]
```

Häufig gibt man den „*key*“-Parameter mittels eines Lambda-Ausdrucks an. Folgendermaßen könnte man etwa Pythons Stringmethoden sortiert nach der Anzahl der enthaltenen `'e'`s sortieren; die Sortierung erfolgt in diesem Beispiel übrigens absteigend, was durch den benannten Parameter „*reverse*“ festgelegt werden kann:

```
1 >>> methods=dir(str)
2 >>> methods.sort(key=lambda s: s.count('e'), reverse=True)
3 >>> methods
4 ['__reduce_ex__', '_formatter_field_name_split', '__getattribute__', ...]
```

Aufgabe A.21

Sortieren Sie die Zeilen einer Datei `test.txt` ...

(a) ... absteigend ihrer Länge nach.

(b) ... der Anzahl der enthaltenen Ziffern nach.

(c) ... der Anzahl der enthaltenen Wörter (verwenden Sie die String-Methode *split*) nach.

(d) ... der Länge des längsten Wortes der jeweiligen Zeile nach.

Hinweis: Die Zeilen der Datei `test.txt` können Sie folgendermaßen auslesen:
open(`'test.txt'`).*readlines*()

A.7 Objektorientierte Programmierung

Zentral für die objektorientierte Programmierung ist die Möglichkeit neue *Klassen* erzeugen zu können. Eine Klasse ist eigentlich nichts anderes als ein Python-Typ, genau wie *int*, *string*, *list* oder *dict*. Die Syntax zur Erzeugung einer neuen Klasse lautet:

class ⟨*name*⟩ : ⟨*kommando*$_1$⟩ ... ⟨*kommando*$_n$⟩	Erzeugt eine neue Klasse mit Namen ⟨*name*⟩. Jedesmal, wenn ein Objekt dieser Klasse erzeugt wird, werden ⟨*kommando*$_1$⟩, ...⟨*kommando*$_n$⟩ ausgeführt. Häufig befinden sich unter den Kommandos Methoden-Definitionen (d. h. relativ zur Klasse lokale Funktionen) oder Definitionen von Klassenvariablen.

Listing A.33 zeigt ein Beispiel für eine sehr einfache Klassendefinition:

```
1 class Auto:
2   typ = 'VW Golf'
3   def sagHallo( self ):
4     print 'Hallo, ich bin ein Auto'
```

Listing A.33: *Definition einer einfachen Klasse*

In Zeile 2 wird eine relativ zur Klassendefinition lokale Variable *typ* definiert; eine solche lokale Variable nennt man im Sprachjargon der Objektorientierten Programmierung als *Klassenattribut*. In Zeile 4 wird eine Funktion *sagHallo* definiert; im Sprachjargon der Objektorientierten Programmierung wird eine solche lokale Funktion als Methode bezeichnet. Jede Methode *muss* als erstes Argument den Parameter „ *self*“ übergeben bekommen; *self* enthält immer die Referenz auf das Objekt selbst; so kann innerhalb der Methode etwa auf Attribute des Objekts zugegriffen werden. Bei jedem Methodenaufruf wird self immer explizit mit übergeben.

Durch folgende Anweisung

```
>>> einAuto = Auto()
```

kann man eine *Instanz* der Klasse erzeugen, im OO-Sprachjargon üblicherweise auch als ein *Objekt* (in diesem Fall der Klasse *Auto*) bezeichnet. Auf das Attribut *typ* kann man mittels *einAuto.typ* zugreifen, und auf die Methode *sagHallo* kann man mittels *einAuto.sagHallo* zugreifen – dadurch erhält die Methode implizit als erstes Argument das Objekt *einAuto*; in der Definition von *sagHallo* wird dieses allerdings nicht verwendet.

```
>>> einAuto.typ
'VW Golf'
>>> einAuto.sagHallo()
'Hallo, ich bin ein Auto'
```

Enthält eine Klassendefinition die Methode *__init__*, so wird diese Methode bei jedem Erzeugen eines Objektes automatisch ausgeführt. Neben dem obligaten Argument *self* kann die *__init__*-Methode noch weitere Argumente enthalten; die Erzeugung von Objekten kann so abhängig von bestimmten Parametern erfolgen. Listing A.34 zeigt eine modifizierte Definition der Klasse *Auto* die bei der Objekterzeugung zwei Parameter *t* und *f* erwartet:

```
1  class Auto:
2    anzAutos = 0
3
4    def __init__(self, t, f):
5      self.typ = t
6      self.farbe = f
7      Auto.anzAutos += 1
8
9    def __del__(self):
10     Auto.anzAutos -= 1
11
12   def ueberDich(self):
13     print "Ich bin ein %ser %s; du hast momentan %d Autos" % \
14           (self.farbe, self.typ, Auto.anzAutos)
```

Listing A.34: *Definition einer komplexeren Auto-Klasse*

Bei der Erzeugung einer neuen Instanz von *Auto* wird nun immer automatisch die *__init__*-Methode ausgeführt, die neben *self* zwei weitere Argumente erwartet, die dann in Zeile 6 und 7 den (Objekt-)Attributen *typ* und *farbe* zugewiesen werden. Man kann mittels *self.typ* bzw. *self.farbe* auf die Attribute *typ* bzw. *farbe* des aktuellen Objektes zugreifen.

Die Attribute *self.typ* und *self.farbe* gehören also zu *einem* bestimmten Objekt der Klasse *Auto* und können für unterschiedliche Objekte unterschiedliche Werte annehmen.

Dagegen ist das in Zeile 2 definierte Attribut *anzAutos* ein Klassenattribut, d. h. es gehört nicht zu einer bestimmten Instanz von *Auto*, sondern ist global für alle Objekte der Klasse sichtbar; Gleiches gilt für alle Methodendeklarationen – auch sie sind global für alle Objekte der Klasse sichtbar.

Bei jeder Erzeugung einer Klasseninstanz erhöhen wir die Variable *anzAutos* um Eins. Die in Zeile 10 definierte spezielle Methode *__del__* wird immer dann automatisch aufgerufen, wenn mittels des **del**-Kommandos ein Objekt der Klasse gelöscht wird; in Zeile 11 erniedrigen wir die Variable *anzAutos* um Eins, wenn ein Objekt gelöscht wird.

In folgendem Beispiel werden drei verschiedene Variablen vom Typ *Auto* erzeugt:

```
>>> a1 = Auto("Mercedes-Benz", "gruen")
>>> a2 = Auto("BMW", "rot")
>>> a3 = Auto("VW Golf", "schwarz")
```

Nun können wir uns mittels der Methode *ueberDich* Informationen über das jeweilige Objekt ausgeben lassen:

```
>>> a1.ueberDich()
Ich bin ein gruener Mercedes-Benz; du hast momentan 3 Autos
>>> del(a1)
>>> a2.ueberDich()
Ich bin ein roter BMW; du hast momentan 2 Autos
```

Man kann auch eine neue Klasse erzeugen, die auf den Attributen und Methoden einer anderen Klasse basiert – im OO-Jargon nennt man das auch Vererbung. Falls uns das Alter eines Autos nur dann interessiert, wenn es sich um einen Oldtimer handelt, dann könnten wir eine Klasse *Oldtimer* wie folgt definieren:

```
1 class Oldtimer(Auto):
2   def __init__ ( self, t, f, a):
3     Auto. __init__ ( self ,t ,f)
4     self . alter = a
5   def ueberDich(self):
6     Auto.ueberDich(self)
7     print "Ausserdem bin ich %d Jahr alt" % self.alter
```

Wie man sieht, muss man die *__init__*-Methode der Basisklasse explizit aufrufen; Gleiches gilt auch für andere gleichlautende Methoden: die Methode *ueberDich* muss die gleichlautende Methode der Basisklasse explizit aufrufen. Wir können nun ein Objekt vom Typ *Oldtimer* folgendermaßen erzeugen und verwenden:

```
>>> o1 = Oldtimer("BMW", "grau", 50)
>>> o1.ueberDich()
Ich bin ein grauer BMW; du hast momentan 3 Autos
Ausserdem bin ich 50 Jahr alt
```

Basisklassen modellieren i. A. allgemeinere Konzepte und daraus abgeleitete Klassen modellieren entsprechend spezialisiertere Konzepte, wie es ja im Falle von *Auto* und *Oldtimer* auch der Fall ist: „Oldtimer“ ist ein Spezialfall von einem „Auto“.

Neben der *__init__*-Methode und der *__del__*-Methode gibt es in Python noch eine Reihe weiterer Methoden mit spezieller Bedeutung, unter Anderem:

- *__str__*(*self*): Diese Methode berechnet die String-Repräsentation eines bestimmten Objektes; sie wird durch Pythons interne Funktion *str*(...) und durch die **print**-Funktion aufgerufen.
- *__cmp__*(*self*, *x*): Diese Methode wird bei Verwendung von Vergleichsoperationen aufgerufen; sie sollte eine negative ganze Zahl zurückliefern, falls $self < x$; sie sollte 0 zurückliefern, falls $self == x$; sie sollte eine positive ganze Zahl zurückliefern, falls $self > x$.
- *__getitem__*(*self*, *i*): Wird bei der Auswertung des Ausdrucks *self*[*i*] ausgeführt.
- *__setitem__*(*self*, *i*, *v*): Wird bei einer Zuweisung *self*[*i*]=*v* ausgeführt.
- *__len__*(*self*): Wird bei der Ausführung der Python internen Funktion *len*(...) aufgerufen.

A.7.1 Spezielle Methoden

Python interpretiert einige Methoden, deren Namen stets mit „__“ beginnen und mit „__“ enden, in einer besonderen Weise. Ein Beispiel haben wir hierbei schon kennengelernt: die *__init__*-Methode, die immer dann aufgerufen wird, wenn eine neue Instanz einer Klasse erzeugt wird. Wir lernen im Folgenden noch einige weitere (nicht alle) solcher Methoden kennen.