

mitp

4., aktualisierte Auflage

Michael Weigend

inklusive CD-ROM



**Objektorientierte
Programmierung mit**

Python 3

Einstieg, Praxis, professionelle Anwendung

**Klassen, Objekte und Vererbung praktisch
angewendet**

**Datenbanken, grafische Benutzungsoberflächen
und Internet-Programmierung**

**Übungen mit Musterlösungen
zu jedem Kapitel**

Python im interaktiven Modus

In diesem Kapitel beginnt die praktische Arbeit mit Python. Zunächst wird erklärt, wie man die Software installiert, die man zum Programmieren benötigt. Im interaktiven Modus führen Sie einzelne Python-Anweisungen aus und gewinnen einen Eindruck, wie man mit Python Probleme lösen kann.

2.1 Python installieren

Bevor Sie mit der praktischen Programmierung beginnen, müssen Sie Python auf Ihrem Rechner installieren. Sie können sämtliche Software, die Sie für die Arbeit mit Python benötigen, von der Python-Homepage <http://www.python.org/download> herunterladen. Wählen Sie die Version, die dort empfohlen wird. Das ist nicht unbedingt die aktuellste Version. Dieses Buch bezieht sich auf Version 3.1.1, die am 17. August 2009 veröffentlicht wurde. Falls Sie eine neuere Version installieren, müssten aber dennoch alle Programme, die in diesem Buch beschrieben werden, funktionieren, da bei der Weiterentwicklung von Python 3 auf Abwärtskompatibilität Wert gelegt wird. Python ist völlig kostenlos und kompatibel mit der GNU General Public License (GPL).

Python ist für Microsoft Windows, Unix und Mac OS verfügbar. Die offizielle Distribution der Python Software Foundation umfasst vier Komponenten:

- Den Python-Interpreter
- Die Entwicklungsumgebung IDLE
- Module, die vom Python-Interpreter eingebunden werden können
- Eine umfangreiche Dokumentation

Auf Unix-Systemen ist Python in der Regel bereits installiert. Das gilt insbesondere für Linux-Distributionen. Prüfen Sie, welche Version vorliegt, indem Sie in einem Konsolenfenster auf der Kommandozeile den Befehl `python -V` eingeben. Sofern Python installiert ist, meldet Ihnen das System die Version. Beispiel:

```
$ python -V
Python 3.1.1
```

Wenn Sie keine Version von Python 3 vorfinden, müssen Sie nachinstallieren. Die Installation bei Unix-Systemen ist Aufgabe des Systemadministrators (*root*). Folgende Schritte sind notwendig:

- Das Datenarchiv (Unix *Tarball*) mit allen Python-Dateien wird von der Download-Seite <http://www.python.org/download> heruntergeladen. Man findet dort zwei Versionen, von denen man sich eine aussuchen kann. Sie unterscheiden sich nur im angewendeten

Kompressionsverfahren: `Python-3.1.1.tgz` und `Python-3.1.1.tar.bz2`. Sie finden diese Dateien auch auf der beiliegenden CD im Ordner *System*. Das Archiv wird im Verzeichnis `/usr/src` gespeichert.

- Die Archivdatei wird auspackt.
- Das Datenarchiv enthält C-Quelltexte, die erst noch kompiliert werden müssen. Auf dem Rechner muss also ein C-Compiler installiert sein, was auf Unix-Systemen in der Regel der Fall ist.
- Schließlich erfolgt mit `make`, `configure` und `install` die eigentliche Installation.

Eine typische Befehlsfolge ist:

```
$ tar xzf Python-3.1.1.tgz
$ cd Python 3.1
$ ./configure
$ make
$ make install
```

Unter Windows ist die Installation sehr einfach. Auf der Download-Seite <http://www.python.org/download> werden zwei unterschiedliche Installationsdateien angeboten:

- Python 3.1.1 Windows x86 MSI Installer
- Python 3.1.1 Windows AMD64 MSI Installer

Beide Dateien finden Sie auch auf der beiliegenden CD im Ordner *System/Windows*. Das erste Installationspaket ist für Systeme mit 32-Bit-Architektur und das zweite für Systeme mit 64-Bit-Architektur. Wenn Sie nicht wissen, welche Architektur bei Ihrem Rechner vorliegt, haben Sie wahrscheinlich ein 32-Bit-System. Dennoch können Sie es zuerst mit der Installation des AMD64 MSI Installers versuchen. Führen Sie einfach das Programm aus (Programmicon anklicken) und folgen Sie den Anweisungen. Wenn das Installationspaket nicht zu Ihrem System passt, erhalten Sie eine entsprechende Meldung und die Installation wird abgebrochen. Dann versuchen Sie es halt mit der 32-Bit-Variante. Es kann also nichts schiefgehen.

Danach muss der Systempfad richtig gesetzt werden. Dazu müssen Sie die Umgebungsvariable `Path` bearbeiten. Unter MS Vista geht das so: Klicken Sie auf den Start-Button und wählen Sie `SYSTEMSTEUERUNG`. Es öffnet sich ein Fenster mit Systemprogrammen. Klicken Sie auf das Icon `SYSTEM`. Es öffnet sich ein neues Fenster. Wählen Sie `ERWEITERTE SYSTEMEINSTELLUNGEN`. Es öffnet sich ein Fenster mit dem Titel `Systemeigenschaften`. Wählen Sie den Reiter `ERWEITERT` und klicken Sie auf die Schaltfläche `UMGEBUNGSVARIABLEN`. Suchen Sie im Feld `SYSTEMVARIABLEN` die Variable `Path` und klicken Sie sie an, um sie auszuwählen. Drücken Sie dann auf die Schaltfläche `BEARBEITEN`. Es erscheint ein weiteres kleines Eingabefenster (siehe Abbildung 2.1 rechts oben). Im unteren Eingabefeld können Sie den aktuellen Wert der Umgebungsvariablen `Path`. Setzen Sie den Cursor an das Ende der Zeile und hängen Sie folgenden Text an:

```
;C:\python31
```

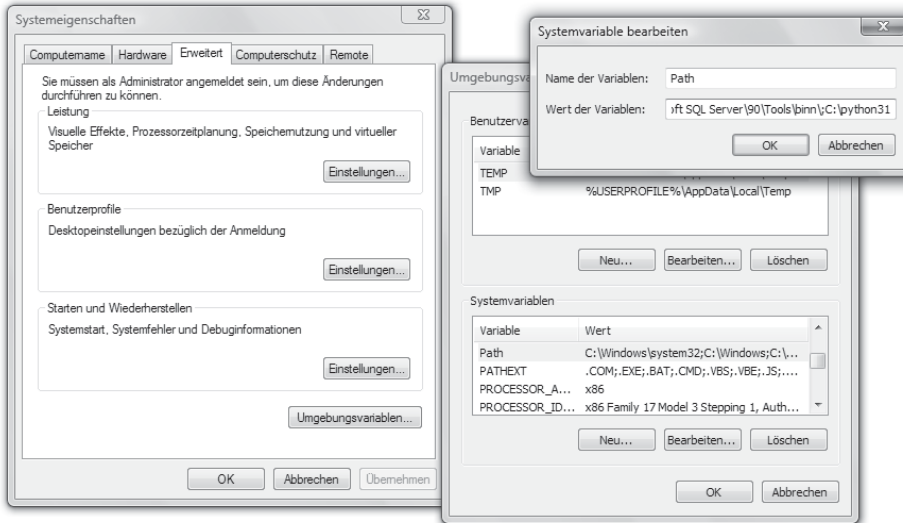


Abb. 2.1: Erweitern der Umgebungsvariablen Path unter MS Windows Vista

Für Windows wird noch ein Zusatzpaket von Mark Hammond mit Windows-spezifischen Erweiterungen empfohlen. Es enthält unter anderem die Programmentwicklungsumgebung Pythonwin. Man findet es unter der Adresse <http://sourceforge.net/projects/pywin32/files/>. Mit Pythonwin kann man auch Programme testen, die Kommandozeilenargumente verwenden (mehr dazu in Abschnitt 9.6).

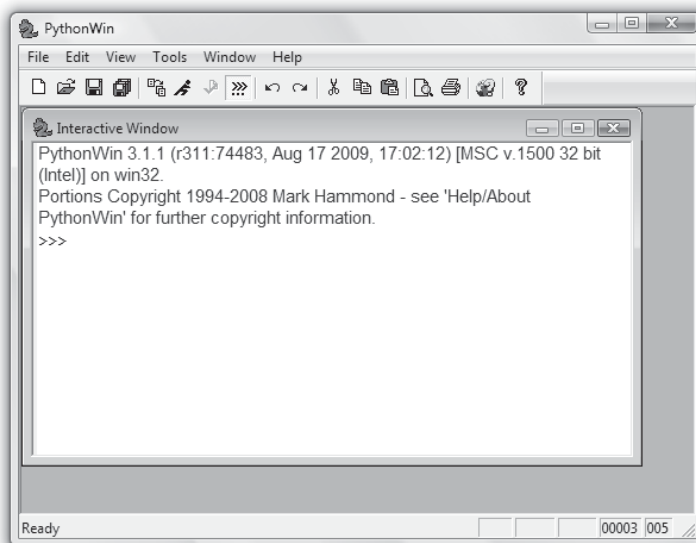


Abb. 2.2: Die kostenlose Entwicklungsumgebung PythonWin von Mark Hammond

2.2 Python im interaktiven Modus

Der Python-Interpreter kann in einem interaktiven Modus aufgerufen werden, in dem man einzelne Zeilen Programmtext eingeben und die Wirkung sofort beobachten kann. Im interaktiven Modus kann man mit Python experimentieren, etwa um sich in neue Programmier Techniken einzuarbeiten oder logische Fehler in einem Programm, das man gerade bearbeitet, aufzuspüren.

Der interaktive Python-Interpreter – die Python-Shell – kann auf verschiedene Weise gestartet werden.

2.2.1 Start des Python-Interpreters in einem Konsole-Fenster

Geben Sie in einem Betriebssystemfenster (z.B. Eingabeaufforderung bei Windows-Systemen) das Kommando `python` ein. Bei Windows kann man auch einfach das Icon des Python-Interpreters anklicken. Man findet es z.B. im Start-Menü unter: `START|ALLE PROGRAMME|PYTHON 3.1|PYTHON (COMMAND LINE)`. Abbildung 2.3 zeigt, wie sich der Interpreter in einem Eingabeaufforderung-Fenster meldet.

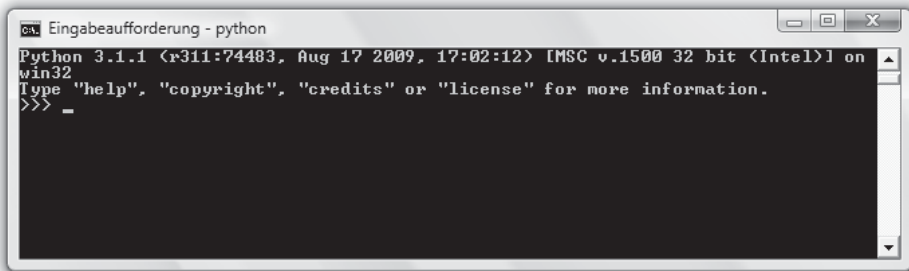


Abb. 2.3: Python nach dem Start in einem Kommando-Fenster

2.2.2 Die Python-Shell von IDLE

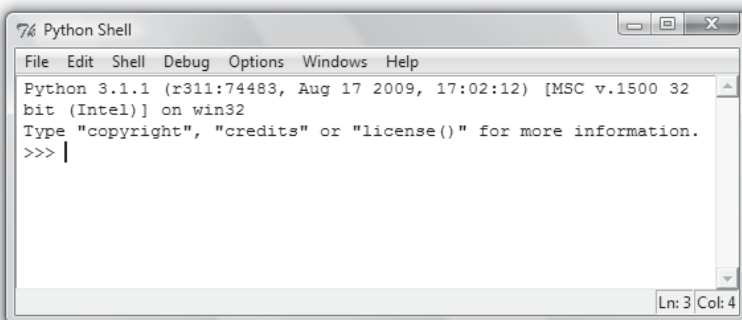


Abb. 2.4: Die Python-Shell der integrierten Entwicklungsumgebung IDLE

Zur Standarddistribution von Python gehört die integrierte Entwicklungsumgebung IDLE. Der Name erinnert an Eric Idle, einem Mitglied der Monty-Python-Gruppe. Wenn Sie IDLE starten, öffnet sich das Shell-Fenster, in dem Sie einen Dialog mit dem interaktiven Interpreter führen können.

2.2.3 Die ersten Python-Befehle ausprobieren

Die Python-Shell meldet sich immer mit einer kurzen Information über die Version und einigen weiteren Hinweisen. Dann folgen drei spitzen Klammern `>>>`. Diese Zeichenfolge nennt man Promptstring oder einfach Prompt. Sie stellt eine Aufforderung zur Eingabe dar. Hinter dem Prompt kann man eine *Anweisung* eingeben und durch `[Enter]` beenden. Der Python-Interpreter bearbeitet die Anweisung sofort. In den nächsten Zeilen kommt entweder eine Fehlermeldung, ein Funktionsergebnis oder (z.B. bei Zuweisungen) *keine* Systemantwort. Python-Anweisungen können arithmetische Ausdrücke (Terme) sein:

```
>>> 2+2
4
>>> (2+3)*4
20
```

Hier wird jeweils der mathematische Term ausgewertet und das Rechenergebnis ausgegeben. Nun geben wir einen ungültigen Term ein, bei dem eine Klammer fehlt:

```
>>> (2+3)*1+2)
SyntaxError: invalid syntax
```

In diesem Fall reagiert das System mit einer Fehlermeldung. Es handelt sich hier um einen Syntaxfehler, d.h. einen Verstoß gegen die Regeln, die den Aufbau einer Anweisung aus Zeichen definieren.

Bei manchen Anweisungen gibt das System überhaupt keine Rückmeldung. Beispiel:

```
>>> zahl = 1
>>>
```

In diesem Beispiel wird dem Namen `zahl` das Objekt mit dem Wert `1` zugewiesen. Man kann es sich auch so vorstellen, dass der Wert `1` unter dem Namen `zahl` gespeichert wird. Er kann wieder abgerufen werden, indem man den Namen der Variablen eingibt:

```
>>> zahl
1
```

2.2.4 Hotkeys

Es gibt einige nützliche Tastenkombinationen (Hotkeys, Shortcuts), die die Arbeit mit der IDLE-Shell im interaktiven Modus erheblich beschleunigen:

Mit `Alt+p` und `Alt+n` können Sie in der Folge der zuletzt eingegebenen Kommandos (History) vor- und zurückgehen. Geben Sie zunächst zwei beliebige Befehle ein:

```
>>> 1 + (2 * 3) + 4
11
>>> 1234 * 56789
70077626
>>>
```

Wenn Sie *einmal* die Tastenkombination `Alt+p` betätigen, erscheint hinter dem letzten Prompt das vorige Kommando (*previous*):

```
>>> 1234 * 56789
```

Bei nochmaliger Eingabe dieses Hotkeys erscheint die vorvorige Zeile:

```
>>> 1 + (2 * 3) + 4
```

Mit der Tastenkombination `Strg+c` brechen Sie die Ausführung eines gerade laufenden Programms ab. Man macht dies, wenn die Ausführung zu lange dauert oder ein Programm auf Grund eines Programmierfehlers überhaupt nicht zu einem Ende kommt. Zum Ausprobieren können Sie einfach einen Term eingeben, für dessen Auswertung der Python-Interpreter viel Zeit braucht. Die Potenz 12345^{12345} ist eine Zahl mit etwa fünfzigtausend Ziffern und ist entsprechend mühevoll zu berechnen.

```
>>> 12345**12345
```

Wenn Sie nun die Tastenkombination `Strg+c` zum Abbruch eingeben, erhalten Sie nach einigen Sekunden (bitte Geduld!) folgende Meldung:

```
Traceback (most recent call last):
  File "<pyshe11#6>", line 1, in <module>
    12345**12345
KeyboardInterrupt
```

2.3 Objekte

Python ist in einem sehr umfassenden Sinne eine objektorientierte Programmiersprache. Daten, Funktionen und andere Sprachelemente werden durch Objekte repräsentiert. Wenn man in der Mathematik von der Zahl 123 spricht, denkt man zunächst an einen numerischen Wert. *Werte* von Objekten werden durch *Literale* repräsentiert. Das sind Zeichenfolgen, die nach bestimmten Regeln aufgebaut sind. Der Wert der natürlichen Zahl 123 kann durch die Zeichenfolgen 123 (Dezimalzahl) oder 0173 (Oktalzahl) repräsentiert werden – zwei verschiedene Literale für den gleichen numerischen Wert.

Der Wert ist nur *ein* Aspekt eines Objektes. Für das Objekt mit dem Wert 123 ist auch von Bedeutung, dass es sich um eine ganze Zahl handelt. Das Objekt gehört zu einem bestimmten *Typ*, nämlich dem Typ »ganze Zahl« (engl. *integer*). Steht die Ziffernfolge in Hochkomma oder Anführungsstrichen, handelt es sich um eine Zeichenkette (engl. *string*). Die Zeichenkette '123' ist etwas anderes als die ganze Zahl 123. Der Typ eines Objektes wird dann wichtig, wenn es in einem Programm verarbeitet werden soll. Mit Zahlen kann man arithmetische Operationen durchführen, mit Zeichenketten nicht. Probieren Sie aus:

```
>>> 123-1
122
>>> '123'-'1'
Traceback (most recent call last):
  File "<pysHELL#27>", line 1, in ?
    '123'-'1'
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Offenbar ist die Subtraktion für Objekte vom Typ Zeichenkette nicht erlaubt.

Der Typ eines Objektes kann mit der Standardfunktion `type()` ermittelt werden. Beispiel:

```
>>> type(123)
<class 'int'>
>>> type('123')
<class 'str'>
```

Ein drittes Merkmal aller Objekte ist, dass sie eine *Identität* besitzen. Bei Python wird die Identität eines Objektes durch eine (einmalige) ganze Zahl repräsentiert, die mit der Standardfunktion `id()` abgefragt werden kann.

```
>>> id(123)
8004856
>>> id('123')
19427280
```

Die Identität dient der Identifizierung eines Objektes. Es ist mehr als nur eine verschlüsselte Form des Wertes. Es kann sein, dass zwei Objekte den gleichen Wert, aber unterschiedliche Identität besitzen. Diese Objekte sind dann *gleich*, aber nicht *identisch*. In der realen Welt können z.B. auch Atome völlig gleich sein, ohne identisch zu sein.

Halten wir also fest: Alle Objekte besitzen einen Wert, einen Typ und eine Identität (siehe Abbildung 2.5).

Zeichenketten und ganze Zahlen sind Beispiele für Standard-Typen von Python, die in der Programmiersprache vorgegeben sind (*built-in types*). Es ist auch möglich, eigene Datentypen zu definieren.

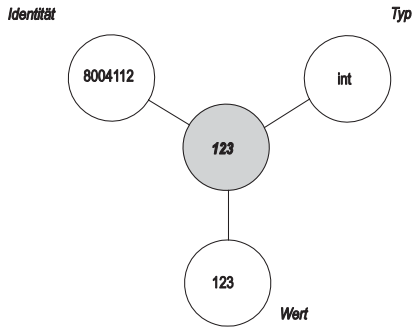


Abb. 2.5: Repräsentation der Zahl 123 durch ein Objekt mit Identität, Wert und Typ

2.4 Namen

Objekte können anonym sein oder einen Namen besitzen, über den sie angesprochen werden können. Wenn Sie in der Python-Shell den Ausdruck

```
>>> 2+3
```

eingeben, werden anonyme Objekte mit den Werten 2 und 3 verarbeitet und in der nächsten Zeile die Summe der beiden Werte ausgegeben. Die Objekte 2 und 3 besitzen zwar einen Wert, einen Typ und eine Identität, aber keinen Namen. Mit folgenden Anweisungen werden den Zahlen 2 und 3 die Namen *zah11* und *zah12* zugeordnet:

```
>>> zah11 = 2
>>> zah12 = 3
```

Man sagt auch: *zah11* und *zah12* sind die *Namen* von zwei Variablen, denen die Werte 2 bzw. 3 zugewiesen werden. Die Namen *zah11* und *zah12* bestehen aus Buchstaben und Ziffern, beginnen aber mit einem Buchstaben. Sie sind nicht von Anführungszeichen umschlossen.

Solche Namen kann man für arithmetische Operationen verwenden:

```
>>> zah11 + zah12
5
```

2.4.1 Syntax-Regeln für Bezeichner

Bezeichner (*identifiers*) sind Zeichenketten, die für Namen verwendet werden dürfen. Im letzten Abschnitt haben wir die Bezeichner *zah11* und *zah12* verwendet. Probieren Sie folgende Anweisung aus:

```
>>> zah1$1 = 100
SyntaxError: invalid syntax
```

Das System meldet einen Syntaxfehler. Die Zeichenkette `zahl$1` ist in der Python-Syntax kein gültiger Bezeichner. Deshalb kann die obige Anweisung nicht ausgeführt werden. Die Syntax einer Programmiersprache wird ganz zweifelsfrei und eindeutig durch eine *Grammatik* definiert. Grammatiken kennt man auch aus dem Sprachunterricht. Die Grammatik einer Programmiersprache unterscheidet sich von einer solchen Grammatik eigentlich nur darin, dass sie mathematisch exakt formuliert ist. Sie enthält *Regeln*, die genau festlegen, wie gültiger Programmtext auszusehen hat. Die Grammatik ist also ein sehr wichtiges Dokument. In diesem Buch werden an einigen Stellen Grammatik-Regeln verwendet, die dann aber zusätzlich noch anschaulich erklärt werden. Im Anhang finden Sie noch weitere Informationen zum Thema Grammatik.

Der Aufbau eines Bezeichners wird durch folgende (etwas vereinfachten) Regeln beschrieben:

```
identifizier ::= id_start id_continue*
id_start    ::= <Buchstaben, Unterstrich und einige
                andere Zeichen>
id_continue ::= <alle Zeichen in id_start plus Ziffern
                und einige andere Zeichen>
```

Die erste Regel besagt Folgendes: Ein Bezeichner (*identifizier*) muss mit genau einem Zeichen aus `id_start` beginnen. Danach können beliebig viele Zeichen aus `id_continue` folgen. Das »beliebig viele« wird mit dem Stern-Operator `*`, der hinter `id_continue` steht, zum Ausdruck gebracht. »Beliebig viele« bedeutet übrigens auch, dass unter Umständen gar kein Zeichen aus `id_continue` folgt. Dann besteht der Bezeichner nur aus einem einzigen Buchstaben.

Nun müsste man bloß noch wissen, welche Zeichen zu `id_start` und `id_continue` gehören. Das wird in den letzten beiden Regeln festgelegt. Sie sind hier etwas vereinfacht formuliert. Am besten merken Sie sich Folgendes:

Hinweis

Ein Bezeichner beginnt mit einem Buchstaben oder Unterstrich. Danach folgen beliebig viele Buchstaben, Unterstriche oder Dezimalziffern.

Zulässige Bezeichner sind `__init__`, `wort_liste`, `zahl1`, `farbeNummer`, `lärm`.

Nicht erlaubt sind `1_Klasse` (Ziffer am Anfang) oder `zahl-1` (ungültiges Zeichen -).

Die Originalregeln sind etwas komplizierter und listen alle Unicode-Kategorien auf, die für Bezeichner erlaubt sind. Sie können Sie hier nachlesen: <http://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html>.

2.4.2 Schlüsselwörter

Allerdings gibt es einige reservierte Wörter, die zwar den obigen Regeln entsprechen, aber dennoch nicht als Namen verwendet werden dürfen. Man nennt sie auch Schlüsselwörter (*keywords*) und sie sind bereits mit einer bestimmten Bedeutung belegt:

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

2.5 Anweisungen

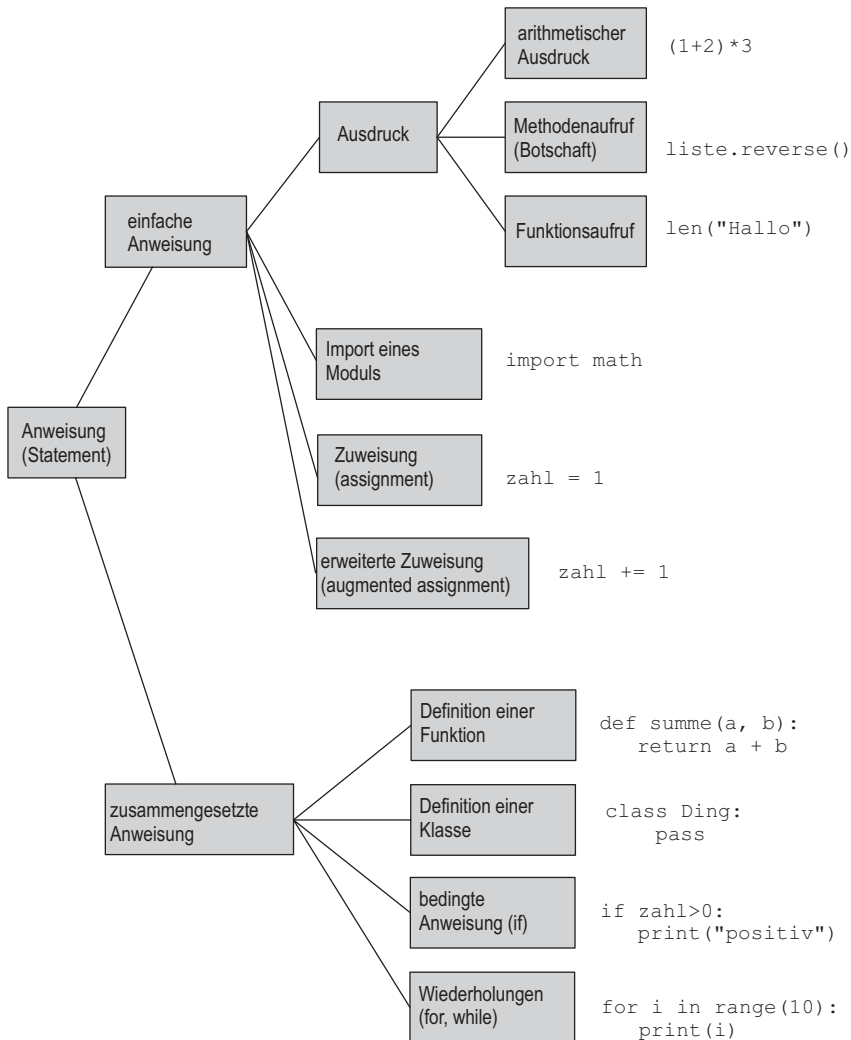


Abb. 2.6: Wichtige Python-Anweisungen im Überblick

Anweisungen (engl. *statements*) sind die Grundbausteine eines Python-Programms. Abbildung 2.6 gibt einen Überblick über die wichtigsten Typen von Python-Anweisungen. Man kann sie grob in einfache und zusammengesetzte Anweisungen einteilen. Eine zusammengesetzte Anweisung enthält als Bestandteile weitere Anweisungen und kann sehr kompliziert aufgebaut sein. An dieser Stelle gehen wir zunächst nur auf einige grundlegende einfache Anweisungen ein. Alle anderen werden später in verschiedenen Kapiteln eingeführt.

2.5.1 Ausdruckenweisungen

Arithmetische und logische Ausdrücke

Die einfachste Form einer Anweisung besteht aus einem Ausdruck. Bereits eine einzelne Zahl oder eine Zeichenkette ist ein Ausdruck und ergibt eine Anweisung, die freilich nichts bewirkt. Der eingegebene Wert wird vom interaktiven Interpreter so, wie er ist, wieder ausgegeben:

```
>>> 12
12
>>> 'Hallo'
'Hallo'
```

Mit Hilfe von Operatoren und runden Klammern können wie in der Mathematik komplexe arithmetische Ausdrücke aufgebaut werden. Gibt man sie als Anweisung ein, werden sie vom Python-Interpreter ausgewertet und das Ergebnis in der nächsten Zeile ausgegeben:

```
>>> 1000*1000
1000000
>>> (1+2)*(3-4)
-3
```

Vergleiche gehören ebenfalls zu den Ausdrücken. Ist ein Vergleich WAHR, liefert der Interpreter den Wert True und sonst False.

```
>>> 'Tag' == 'Nacht'
False
>>> 2 > 1
True
```

Vergleiche kann man mit logischen Operatoren zu komplexen logischen Ausdrücken verknüpfen:

```
>>> (2 > 0) and (2 > -1)
True
>>> not(1 < 0) or (1 < 0)
True
```

Funktionsaufruf

Funktionen sind aufrufbare Objekte (*callable objects*), die eine bestimmte Teilaufgabe lösen können. Wenn eine Funktion aufgerufen wird, übernimmt sie gewisse Werte als Eingabe, verarbeitet diese und liefert einen Wert als Ausgabe zurück (siehe Abbildung 2.7). Die Werte, die man einer Funktion übergibt, nennt man *Argumente* oder *aktuelle Parameter*. Im interaktiven Modus kann man Funktionen aufrufen und erhält dann in der nächsten Zeile den zurückgegebenen Wert. Beispiele:

```
>>> len ("Wort")
4
```

Hier ist `len` der Name der Funktion und die Zeichenkette "Wort" das Argument. Zurückgegeben wird die Länge der Zeichenkette, d.h. die Anzahl der Zeichen.

Die Funktion `min()` akzeptiert eine unterschiedliche Anzahl von Argumenten und gibt das kleinste zurück:

```
>>> min(1, 2)
1
>>> min(10, 2, 45, 5)
2
```

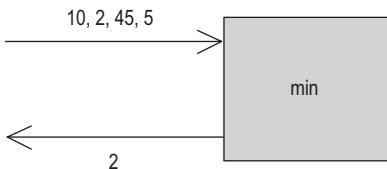
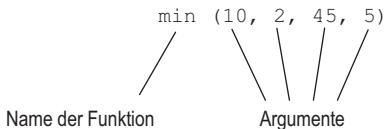
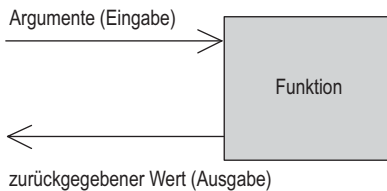


Abb. 2.7: Aufruf einer Funktion

Funktionen, die keinen Wert zurückgeben – auch das gibt es –, nennt man Prozeduren. Beispiel:

```
>>> help()
Welcome to Python 3.1! This is the online help utility.
```

Die Argumente eines Funktionsaufrufs müssen nicht unbedingt explizite Werte sein, es können auch Ausdrücke als Argumente verwendet werden. Sie werden dann vom Python-Interpreter zunächst ausgewertet und der resultierende Wert der Funktion übergeben.

```
>>> min(4+2, 4-2, 4*2)
2
```

In den Ausdrücken, die bei einem Funktionsaufruf als Argument eingesetzt werden, können auch wieder Funktionsaufrufe vorkommen. Derart verschachtelte Funktionsaufrufe werden von innen nach außen ausgewertet. Im folgenden Beispiel wird zuerst die Funktion `len()` aufgerufen. Die zurückgegebenen Werte (9 und 12) werden an die Funktion `min()` als Argumente übergeben:

```
>>> min(len("Wochenend"), len("Sonnenschein"))
9
```

Methodenaufrufe – Botschaften an Objekte

Objektorientierte Programme enthalten Botschaften an Objekte, in denen diese »aufgefordert« werden, »etwas zu tun«. Eine solche Botschaft ist auch eine Anweisung. Eine Botschaft ist der Aufruf einer Methode eines Objektes. Methoden sind Funktionen, die an ein Objekt gekoppelt sind. Ein Methodenaufruf beginnt mit dem Namen des Objektes, dahinter kommt ein Punkt, dann der Name der Methode und schließlich in Klammern die Argumente. Beispiel:

```
>>> liste = [4, 9, 3, 1, 5]
>>> liste.reverse()
>>> liste
[5, 1, 3, 9, 4]
```

In der ersten Anweisung wird ein Objekt mit dem Namen `liste` erzeugt. Es enthält als Wert eine Folge von fünf Zahlen. An dieses Objekt wird in der zweiten Anweisung eine Botschaft geschickt. Die Methode `reverse()` wird aufgerufen. Sie bewirkt, dass die Reihenfolge der Zahlen in der Liste umgekehrt wird. Doch sie gibt keinen Wert zurück. Das Objekt `liste` hat sich einfach nur verändert. Um den geänderten Wert von `liste` sichtbar zu machen, wurde in der vorletzten Zeile der Name des Objektes eingegeben.

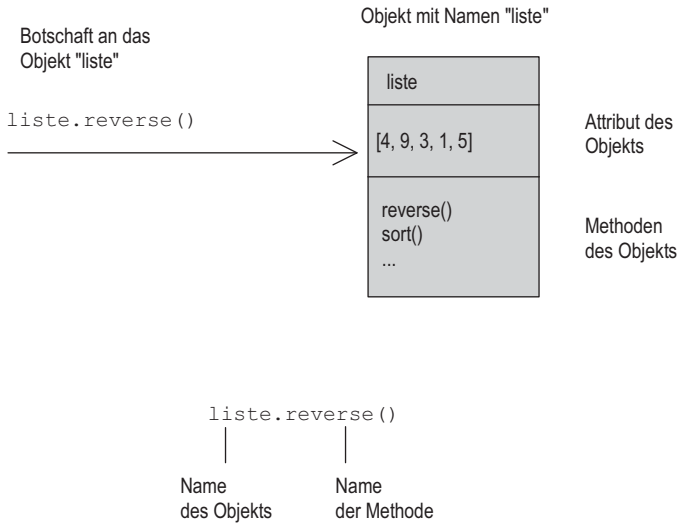


Abb. 2.8: Botschaften sind Methodenaufrufe.

Ausgabe von Werten – die print()-Funktion

Mit der `print()`-Funktion können Werte von Objekten auf dem Bildschirm ausgegeben werden. Früher (bei Python 2) gab es für die Bildschirmausgabe einen eigenen Anweisungstyp, jetzt (bei Python 3) ist es einfach nur ein Funktionsaufruf. Beispiel:

```
>>> print("Hallo!")
Hallo!
```

Nun kann man im interaktiven Modus von Python auch einfach nur die Zeichenkette "Hallo!" eingeben und erhält dann ebenfalls eine Rückmeldung:

```
>>> "Hallo!"
'Hallo!'
```

Der Unterschied ist, dass in der zweiten Zeile jetzt die Zeichenkette `Hallo` in Hochkommata steht. Die Hochkommata gehören bei Python zur Repräsentation von Zeichenketten dazu. Eine Besonderheit der `print()`-Funktion ist also, dass sie nicht die interne Repräsentation eines Wertes, sondern eine für Menschen möglichst gut lesbare Darstellung ausgibt.

In die Klammern hinter dem Funktionsnamen `print` kann auch ein komplexer Ausdruck (z.B. ein mathematischer Term) geschrieben werden. Der Ausdruck wird dann zunächst ausgewertet und das Ergebnis ausgegeben. Beispiele:

```
>>> print(2 + 3)
5
>>> print((2 + 3) * (10 - 8))
```

```
10
>>> print(min(2, 4, 1) + 1)
2

>>> a = 2
>>> b = 3
>>> print(a + b)
5
```

Es können auch mehrere Werte in einem einzigen Aufruf der `print()`-Funktion ausgegeben werden. Dazu werden in den Klammern hinter `print` mehrere Ausdrücke, durch Kommata getrennt, aufgeführt. Vor jedes ausgegebene Objekt schreibt das System ein Leerzeichen, es sei denn, es handelt sich um das erste Wort in einer Zeile.

```
>>> print(1, 2, 3, 4)
1 2 3 4
>>> print("2 mal 2 ist", 2*2)
2 mal 2 ist 4
```

2.5.2 Import-Anweisungen

Python enthält eine Reihe von Standardfunktionen, die fester Bestandteil der Programmiersprache sind (*built-in functions*). Sie sind immer verfügbar. Wir haben im vorigen Abschnitt die Standardfunktionen `min()` und `len()` verwendet.

Speziellere Funktionen befinden sich in Modulen, die erst importiert werden müssen. Beispielsweise enthält das Modul `time` Funktionen, die mit der Ermittlung und Verarbeitung von Uhrzeit oder Kalenderdaten zu tun haben. Die Funktion `asctime()` liefert eine Zeichenkette, die das aktuelle Datum und die Uhrzeit beschreibt. Bevor man diese Funktion nutzen kann, muss jedoch das Modul `time` importiert werden. Das kann durch verschiedene Varianten einer `import`-Anweisung bewerkstelligt werden:

Mit einer Anweisung des Formats

```
from modulname import funktionsname
```

wird gezielt der Name der benötigten Funktion in den lokalen Namensraum importiert. Die Funktion kann dann genauso wie eine Standardfunktion aufgerufen werden:

```
>>> from time import asctime
>>> asctime()
'Sun Sep 06 11:28:26 2009'
```

Alternativ können Sie bei der `from...import`-Anweisung anstelle des Funktionsnamens auch einen Stern `*` einsetzen. Dann werden *alle* Objekte des Moduls in den Namensraum importiert. Beispiel:

```
>>> from time import *
>>> asctime()
'Sun Sep 06 11:28:26 2009'
>>> localtime()
time.struct_time(tm_year=2009, tm_mon=9, tm_mday=6, tm_hour=10,
tm_min=52, tm_sec=13, tm_wday=6, tm_yday=249, tm_isdst=1)
```

Mit der Anweisung

```
import modulname
```

importieren Sie das ganze Modul. Um eine Funktion des importierten Moduls aufzurufen, müssen Sie dem Funktionsnamen den Namen des Moduls und einen Punkt voranstellen. Der Aufruf hat dann das folgende Format:

```
modulname.funktionsname(argumentliste)
```

Beispiel:

```
>>> import time
>>> time.asctime()
'Sun Sep 06 11:28:26 2009'
```

2.5.3 Zuweisungen

Wir kommen nun zu einem ganz zentralen Konzept der imperativen Programmierung, nämlich der Zuweisung (*assignment*). Zuweisungen sind wahrscheinlich die häufigsten Anweisungen in Programmtexten.

Zuweisung eines Wertes

Die einfachste Form der Zuweisung besteht aus einem Namen, gefolgt von einem Gleichheitszeichen und einem Wert, sie hat also die Form:

```
name = wert
```

Der Zuweisungsoperator ist das Gleichheitszeichen. (Beachten Sie, dass man für den Vergleich zweier Objekte das doppelte Gleichheitszeichen `==` verwendet.) Beispiel:

```
>>> x = 1
```

In diesem Beispiel ist `x` ein Name und `1` ein Wert. Man bezeichnet `x` auch als Variable, der man einen Wert zugewiesen hat. Anschaulich kann man sich Variablen als Behälter für Daten vorstellen. Der Variablenname ist sozusagen die Aufschrift des Behälters.

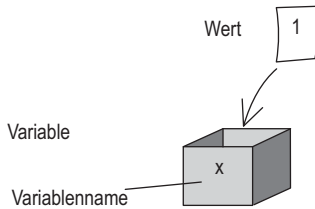


Abb. 2.9: Veranschaulichung einer Zuweisung – Variable als Behälter für Daten

Über den Namen der Variablen kann man auf ihren Inhalt zugreifen. Gibt man im interaktiven Modus den Namen ein, so liefert der Interpreter den Inhalt zurück:

```
>>> x
1
```

Bei einer weiteren Zuweisung wird der alte Wert der Variablen durch einen neuen Wert überschrieben:

```
>>> x = 100
>>> x
100
```

Übertragen von Variableninhalten

Werte können von einer Variablen auf eine andere übertragen werden. Das allgemeine Format einer solchen Art der Zuweisung ist

```
name1 = name2
```

Beispiel: Nach den folgenden Zuweisungen haben die Variablen x und y den gleichen Wert:

```
>>> x = 'Wort'
>>> y = x
>>> x
'Wort'
>>> y
'Wort'
```

Zuweisungen für mehrere Variablen

In einer einzigen Zuweisung kann man bei Python mehreren Variablen gleichzeitig einen (gemeinsamen) Wert zuordnen:

```
>>> x = y = 1
>>> x
```

```
1
>>> y
1
```

Statt eines einzelnen Namens kann links vom Zuweisungsoperator (Gleichheitszeichen) auch eine Folge von Namen stehen (durch Kommata voneinander getrennt). In diesem Fall muss auf der rechten Seite eine ebenso lange Folge von Werten bzw. Ausdrücken stehen. Das heißt, man kann in einer einzigen Zuweisung mehreren Variablen Werte zuordnen:

```
>>> x, y = 1, 2
>>> x
1
>>> y
2
```

Es ist möglich, in einer einzigen Zuweisung die Werte zweier Variablen zu vertauschen.

```
>>> x, y = 1, 2
>>> x, y = y, x
>>> x, y
(2, 1)
```

Die objektorientierte Sichtweise: Einem Objekt einen Namen geben

Die Vorstellung von Variablen als Behälter für Werte ist sehr anschaulich. Sie ist aber nur dann anwendbar, wenn es um einfache Daten-Objekte wie Zahlen oder Zeichenketten geht. Die Behältermetapher widerspricht dem objektorientierten Paradigma. Hier werden alle Daten durch Objekte repräsentiert. Aus Sicht der OOP wird bei einer Zuweisung einem Objekt ein Name gegeben. Zum Beispiel in der Zuweisung

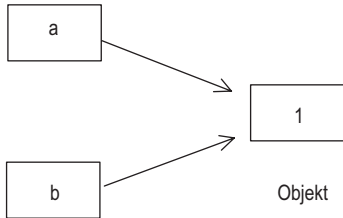
```
>>> a = 1
```

erhält das Objekt mit dem Wert 1 den Namen a. In einer anschließenden Zuweisung

```
>>> b = 1
```

wird demselben Objekt 1 ein weiterer, zusätzlicher Name zugeordnet. Das Objekt 1 hat jetzt zwei Namen, nämlich a und b. Abbildung 2.10 illustriert diesen Gedanken.

Diese Art der Darstellung ist zwar korrekter, aber auch viel abstrakter. Bei arithmetischen Operationen z.B. denkt man meist im Variablenkonzept. Testen Sie in folgendem Experiment, welche Beschreibung Ihnen verständlicher erscheint.



Namen

Abb. 2.10: Zwei Namen für das Objekt 1

Experiment

Geben Sie zu folgenden Darstellungen die zugehörige Python-Anweisung an. Die Lösung finden Sie am Ende des Kapitels.

1 Variablen mit Inhalt

»Der Inhalt der Variablen mit dem Namen a wird mit dem Wert 3 multipliziert und das Ergebnis in der Variablen mit dem Namen b abgespeichert.«

2 Objekte mit Namen

»Zum Wert des Objektes mit dem Namen x wird der Wert 10 addiert. Dem Objekt, das die Summe der beiden Werte repräsentiert, wird als neuer Name y zugeordnet.«

In manchen Fällen ist eine Zuweisung wirklich nur als Namensvergabe zu verstehen. Zum Beispiel kann man auch einer Funktion einen neuen (zusätzlichen) Namen geben:

```
>>> länge = len
>>> länge ('Hallo')
5
```

Hier erhält die Funktion mit dem Namen `len` den neuen Namen `länge`. Nach dieser Zuweisung kommt es einem so vor, als gäbe es eine neue Funktion namens `länge()`, die das Gleiche leistet wie `len()`. In Wirklichkeit hat man dem Funktions-Objekt `len` nur einen neuen (zusätzlichen) Namen gegeben.

2.5.4 Erweiterte Zuweisungen

Eine erweiterte Zuweisung (*augmented assignment*) ist eine Kombination aus einer Zuweisung und einer binären Operation. Beispiele:

```
>>> zahl_1 += 1
>>> zahl_2 *= zahl_1 + 2
```

Sie haben die gleiche Wirkung wie:

```
>>> zahl_1 = zahl_1 + 1
>>> zahl_2 = zahl_2 * (zahl_1 + 2)
```

Sie sehen an diesem Beispiel, dass die Verwendung erweiterter Zuweisungen Schreibarbeit erspart und zu kürzeren Programmtexten führt. Bei einer erweiterten Zuweisung wird der aktuelle Wert der Variablen als erster Operand gewählt. Der zweite Operand ist der Wert des Ausdrucks, der hinter dem Gleichheitszeichen steht. Auf beide Werte wird der Operator (erster Teil des Zuweisungsoperators) angewendet und das Ergebnis dem Ziel zugewiesen.

2.6 Aufgaben

Aufgabe 1

Welche Ergebnisse liefern folgende – zum Teil etwas ungewöhnlichen – Ausdrücke?

```
type(id('a'))
type(3/2)
type(2.0/2)
min("Abend", "Aa1", "Ba11")
-2 ** -3
not(2.0 > 2)
type(len('123'))
(1 < 2) + (1 == 1)
type (1+2 < 2)
```

Aufgabe 2

Tragen Sie in jeder Zeile der folgenden Tabelle die Werte der Variablen ein, die sie nach Ausführung der Anweisung in der ersten Spalte tragen.

Anweisung	x	y	z
x = y = 1	1	1	-
x = 2	2	1	-
z = x			
z *= 3			
x, y = y, 3			
y = y/2			
x, y, z = x, x, x			
x = 'y'			

Tabelle 2.1: Wirkung von Anweisungen auf Variablen

Anweisung	x	y	z
$y = 2 > z$			
$y = \min(2, z, 5)$			

Tabelle 2.1: Wirkung von Anweisungen auf Variablen (Forts.)

Aufgabe 3

Schreiben Sie zu den folgenden umgangssprachlichen Beschreibungen passende Python-Anweisungen auf.

Beschreibung	Anweisung
Das Objekt "Elena" erhält den Namen person.	<code>person = "Elena"</code>
Der Variablen zahl wird der Wert 10 zugewiesen.	
Der Inhalt der Variablen zahl wird um 5 erhöht.	
Der Inhalt der Variablen zahl wird auf dem Bildschirm ausgegeben.	
Der Inhalt der Variablen x wird mit dem Inhalt der Variablen y multipliziert und das Ergebnis der Variablen mit dem Namen produkt zugewiesen.	
Dem Objekt mit dem Wert ['rot', 'gelb', 'grün'] wird der Name s zugeordnet. An das Objekt s wird die Botschaft geschickt, es möge die Reihenfolge seiner Listenelemente umkehren. Anschließend soll der geänderte Wert ausgegeben werden.	

Tabelle 2.2: Formulierung von Anweisungen

2.7 Lösungen

Lösung 1

Ausdruck	Ergebnis	Erklärung
<code>type(id('a'))</code>	<code><class 'int'></code>	Die Identität eines Objektes ist immer eine Nummer, d.h. eine positive ganze Zahl.
<code>type(3/2)</code>	<code><class 'float'></code>	Obwohl 3 und 2 ganze Zahlen sind, wird eine exakte Division durchgeführt. Das Ergebnis ist eine Gleitkommazahl (float).
<code>type(2.0/2)</code>	<code><class 'float'></code>	Da 2.0 eine Gleitkommazahl ist, ist das Ergebnis auch eine Gleitkommazahl (float).

Tabelle 2.3: Ungewöhnliche Ausdrücke

Ausdruck	Ergebnis	Erklärung
<code>min("Abend", "Aa1", "Ball")</code>	<code>'Aa1'</code>	Zeichenketten werden nach der lexikografischen Ordnung sortiert. Die Zeichenkette <code>'Aa1'</code> kommt in der Reihenfolge zuerst und ist somit das Minimum.
<code>-2** -3</code>	<code>-0.125</code>	Berechnet wird <code>(-2)</code> hoch <code>(-3)</code> .
<code>not (2.0 > 2)</code>	<code>True</code>	<code>2.0 > 2</code> hat den Wahrheitswert <code>FALSCH</code> , also ist <code>not (2.0 > 2)</code> <code>WAHR</code> .
<code>type(len('123'))</code>	<code><class 'int'></code>	Die Funktion <code>len()</code> liefert die Länge eines Strings als ganze Zahl (Typ <code>int</code>).
<code>(1 < 2) + (1 == 1)</code>	<code>2</code>	Zwei boolesche Werte können auch addiert werden. <code>True</code> wird dann als <code>1</code> interpretiert.
<code>type (1 + 2 < 2)</code>	<code><class 'bool'></code>	Der Vergleich <code>1 + 2 < 2</code> liefert den Wahrheitswert <code>False</code> .

Tabelle 2.3: Ungewöhnliche Ausdrücke (Forts.)

Lösung 2

Anweisung	x	y	z	Erläuterung
<code>x = y = 1</code>	1	1	-	x und y erhalten den Wert 1.
<code>x = 2</code>	2	1	-	x wird der Wert 2 zugewiesen.
<code>z = x</code>	2	1	2	z erhält den gleichen Wert wie x.
<code>z *= 3</code>	2	1	6	Der Inhalt von z wird mit 3 multipliziert und das Ergebnis wieder z zugewiesen.
<code>x, y = y, 3</code>	1	3	6	x erhält den Wert von y und y erhält den Wert 3.
<code>y = y/2</code>	1	1.5	6	y erhält das Ergebnis einer exakten Division, eine Gleitkommazahl, zugewiesen.
<code>x, y, z = x, x, x</code>	1	1	1	x, y und z wird als Wert der Inhalt von x zugewiesen.
<code>x = 'y'</code>	<code>'y'</code>	1	1	Der Variablen x wird als Wert die Zeichenkette <code>'y'</code> zugewiesen.
<code>y = 2 > z</code>	<code>'y'</code>	<code>True</code>	1	Der Vergleich <code>2 > z</code> ist wahr, deshalb wird y der Wert <code>True</code> zugewiesen.
<code>y = min (2, z, 5)</code>	<code>'y'</code>	1	1	Die Funktion liefert das Minimum der Zahlen 2, 1 (Inhalt von z) und 5, also 1. Dieser Wert wird y zugewiesen.

Tabelle 2.4: Die Wirkung von Anweisungen auf Variablen

Lösung 3

Beschreibung	Anweisung
Das Objekt "Elena" erhält den Namen person.	<code>person = "Elena"</code>
Der Variablen zahl wird der Wert 10 zugewiesen.	<code>zahl = 10</code>
Der Inhalt der Variablen zahl wird um 5 erhöht.	<code>zahl += 5</code> oder <code>zahl = zahl + 5</code>
Der Inhalt der Variablen zahl wird auf dem Bildschirm ausgegeben.	<code>print(zahl)</code>
Der Inhalt der Variablen x wird mit dem Inhalt der Variablen y multipliziert und das Ergebnis der Variablen mit dem Namen produkt zugewiesen.	<code>produkt = x * y</code>
Dem Objekt mit dem Wert ['rot', 'gelb', 'grün'] wird der Name s zugeordnet. An das Objekt s wird die Botschaft geschickt, es möge die Reihenfolge seiner Listenelemente umkehren. Anschließend soll der geänderte Wert ausgegeben werden.	<code>s = ['rot', 'gelb', 'grün']</code> <code>s.reverse()</code> <code>print(s)</code>

Tabelle 2.5: Formulierung von Anweisungen**Lösung zum Experiment »Zuweisung« (Abschnitt 2.5.3)****1 Variablen mit Inhalt**

```
>>> b = a * 3
```

2 Objekte mit Namen

```
>>> y = x + 10
```