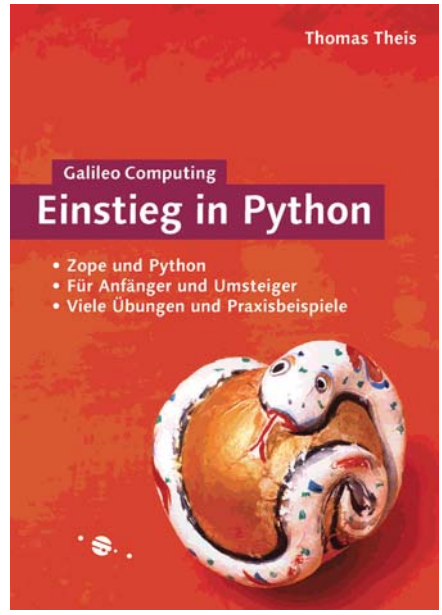
Leseprobe - Kapitel 1:

Thomas Theis

# Einstieg in Python




Galileo Computing

# Inhalt

# A Einführung

*In diesem Kapitel werden die beiden Themen dieses Buches, Python und Zope, kurz vorgestellt. Der Leser wird zur eigenen Programmierung in Python angeleitet. Die Installation von Python unter Windows wird beschrieben.*

## A.1 Dieses Buch

Im vorliegenden Buch wird dem Leser eine Einführung in zwei Themen vermittelt: Python und Zope.

- Python ist eine sehr einfach zu erlernende Programmiersprache und ideal als Einstieg in die Welt der Programmierung geeignet. Trotz ihrer Einfachheit bietet diese Sprache auch die Möglichkeit, komplexe Programme für vielfältige Anwendungsgebiete zu schreiben.
- Zope dient als Werkzeug zur Herstellung dynamischer Websites. Alle Komponenten, die dazu benötigt werden, sind in Zope enthalten. Es ist mit Hilfe von Python entstanden und kann in Zusammenarbeit mit Python-Programmen eingesetzt werden.

Die Kapitel B bis H dieses Buches widmen sich den verschiedenen Aspekten der Programmierung mit Python. In den Kapiteln I und J wird die Entwicklung von Websites mit Zope, auch im Hinblick auf Datenbanken, vorgestellt. Kapitel L behandelt die Zusammenarbeit zwischen Python und MySQL-Datenbanken. Die Kapitel K und M frischen die Grundlagen zu den Themen HTML und MySQL auf.

Es wird besonderer Wert darauf gelegt, dass der Leser selber praktisch mit Python und Zope arbeitet. Er wird von Anfang an dazu aufgefordert, dem logischen Faden von Erklärungen und Beispielen zu folgen.

## A.2 Python-Programmierkurs

**Übungsaufgaben**

Im ersten Kapitel, dem Programmierkurs, werden Übungsaufgaben gestellt, die unmittelbar gelöst werden sollten. Auf diese Weise kann der Leser seine Kenntnisse prüfen, bevor er zum nächsten Thema übergeht. Die Lösungen der Übungsaufgaben finden sich in Kapitel O am Ende des Buches. Dabei ist Folgendes zu beachten:

- Es gibt für jedes Problem viele richtige Lösungen. Der Leser sollte sich nicht davon beunruhigen lassen, dass seine Lösung eventuell nicht

genau so aussieht wie die angegebene. Die angegebene Lösung ist als Anregung zu betrachten, was man anders und gegebenenfalls besser machen kann.

- Bei der eigenen Lösung der Aufgaben wird sicherlich der eine oder andere Fehler auftreten. Man sollte sich dadurch nicht entmutigen lassen.
- Nur aus Fehlern kann man lernen. Auf die vorgeschlagene Art und Weise wird man eine Programmiersprache wirklich erlernen – nicht allein durch das Lesen von Programmierregeln.

## A.3 Schnelle Anwendungs-Entwicklung

Python bietet besonders gute Möglichkeiten zur schnellen Entwicklung umfangreicher Anwendungen. Diese Technik ist unter dem Stichwort RAD (=Rapid Application Development) bekannt geworden. Python vereint zu diesem Zweck folgende Vorteile:

**einfach**

- Einfache, eindeutige Syntax: Python ist eine ideale Programmiersprache für Einsteiger. Sie beschränkt sich auf einfache, klare Anweisungen. In anderen Programmiersprachen werden vielfältige Lösungswege für das gleiche Problem angeboten, so dass der Entwickler weniger geleitet als verunsichert wird. Python beschränkt sich häufig auf einen einzigen möglichen Lösungsweg. Dieser prägt sich ein und wird dem Entwickler vertraut.

**klar**

- Klare Strukturen: Ein Entwickler wird in Python gezwungen, in einer gut lesbaren Struktur zu schreiben. Die Anordnung der Programmzeilen ergibt gleichzeitig die logische Struktur des Programms.

**wiederverwendbar**

- Wiederverwendung von Code: Die Modularisierung, d.h. die Zerlegung eines Problems in Teil-Probleme und die anschliessende Zusammenführung der Teil-Lösungen zu einer Gesamt-Lösung, wird in Python sehr leicht gemacht. Die vorhandenen Teil-Lösungen können sehr unkompliziert für weitere Aufgabenstellungen genutzt werden, so dass ein Entwickler nach einiger Zeit über einen umfangreichen Pool an Modulen verfügt.

**einheitliche Objekte**

- Objekt-Bearbeitung: In Python werden alle Daten als Objekte gespeichert. Dies führt zu einer einheitlichen Daten-Behandlung für Objekte unterschiedlichen Typs. Andererseits wird die physikalische Speicherung der Objekte von Python automatisch, ohne Eingriff des Entwick-

lers vorgenommen. Er muss sich nicht um die Reservierung bzw. Freigabe geeigneter Speicherbereiche kümmern.

- Interpreter / Compiler: Python-Programme werden unmittelbar interpretiert. Sie müssen nicht erst kompiliert und gebunden werden. Dies ermöglicht einen häufigen, schnellen Wechsel zwischen Codierungs- und Testphase. **interpretiert**

- Betriebssystem-Unabhängigkeit: Sowohl Programme, die von der Kommandozeile aus bedient werden, als auch Programme mit grafischen Benutzeroberflächen können auf unterschiedlichen Betriebssystemen (Windows, Unix, Mac OS) ohne Neu-Entwicklung und Anpassung eingesetzt werden. **unabhängig**

## A.4 Installation

Python ist eine frei verfügbare Programmiersprache, die auf verschiedenen Betriebssystem-Plattformen eingesetzt werden kann. Auf der CD zu diesem Buch findet sich die Version 2.2.1 für Windows. Die jeweils neueste Version kann man sich von der offiziellen Python-Website `http://www.python.org` aus dem Internet laden.

**Installation**

Zur Installation unter Windows ruft man die ausführbare Datei, z. B. `[CD-Rom-Laufwerk]:\Python\Python-2.2.1.exe` auf. Die Voreinstellungen des Installationsvorganges können unverändert übernommen werden. Dabei wird Python im Verzeichnis `c:\python22` installiert. Anschließend verfügt man über einen Eintrag im `Startmenü, Programme, Python 2.2` mit dem folgenden Inhalt:


**Abbildung A.1** Python, Eintrag im Windows-Startmenü

Die Installation von Zope wird im Kapitel I unmittelbar beim Einsatz dieses Werkzeuges besprochen.

# B Programmierkurs

*Ein Programmierkurs mit vielen Erläuterungen führt den Leser schrittweise in die Programmierung mit Python ein. Übungsaufgaben geben ihm die Gelegenheit, seine Kenntnisse unmittelbar zu testen. Ein Programmierprojekt verknüpft zum Abschluss viele Teilaspekte zu einem Ganzen.*

## B.1 Python als Taschenrechner

### B.1.1 Aufruf der Python Shell

Python Shell

Man ruft über `Startmenü, Programme, Python 2.2, IDLE (Python GUI)` die nachfolgend dargestellte Entwicklungsumgebung IDLE für Python auf. Sie wird auch Python Shell genannt. Sie kann sowohl zur Programmierung als auch als »Taschenrechner« genutzt werden.

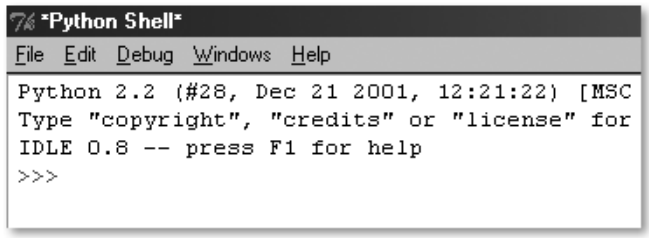



**Abbildung B.1** Entwicklungsumgebung IDLE, Python Shell

Zur Durchführung kleinerer Berechnungen müssen keine vollständigen Programme geschrieben und gestartet werden. Man kann die gewünschten Rechenoperationen direkt an der Eingabestelle, erkennbar an den Zeichen `>>>`, eingeben. Eine abschließende Betätigung der ↵-Taste führt zur unmittelbaren Berechnung und Ausgabe des Ergebnisses.

Die dabei angewendeten Rechenregeln finden auch bei der Erstellung »richtiger« Python-Programme Verwendung, es lohnt sich also eine genauere Betrachtung.

### B.1.2 Addition, Subtraktion und Multiplikation

Nachfolgend werden Eingabe, Verarbeitung und Ausgabe einiger einfacher Berechnungen dargestellt:

```
>>> 41+7.5
48.5
>>> 12-18
-6
>>> 7*3
21
```

**Listing B.1** Einige einfache Berechnungen

**Berechnungen**

Zur Erläuterung: Der Reihe nach werden die Operatoren + (Addition), - (Subtraktion) und * (Multiplikation) eingesetzt. Das Ergebnis erscheint jeweils in der Zeile darunter, nachdem man die ↵-Taste gedrückt hat. Als Dezimalzeichen gilt der Punkt, wie bei der ersten Berechnung (`41 + 7.5 = 48.5`) erkennbar.

### B.1.3 Division

Es folgen zwei Divisionsaufgaben:

```
>>> 20/8
2
>>> 20/-8
-3
```

**Listing B.2** Divisionsaufgaben

Zur Erläuterung: Es fällt auf, dass es sich nicht um die mathematisch richtigen Ergebnisse (`2.5` und `-2.5`) handelt. Dies liegt daran, dass beide Zahlen als ganze Zahlen eingegeben wurden. Bei Python wird das Ergebnis einer solchen Division berechnet, indem:

- das mathematisch richtige Ergebnis ermittelt wird
- die nächstkleinere ganze Zahl ermittelt wird (von `2.5` aus ist dies `2`, von `-2.5` aus ist dies `-3`)

In einigen wenigen Fällen mag dieses Ergebnis erwünscht sein. Falls man bei einer Division allerdings das richtige Ergebnis haben möchte, so sollte man zumindest bei einer der beiden Zahlen angeben, dass Stellen hinter dem Komma gewünscht sind. Dies wird nachfolgend dargestellt:

```
>>> 20/8.0
2.5
>>> 20.0/8
2.5
```

```
>>> 20/-8.0
-2.5
>>> 20.0/-8
-2.5
```

**Listing B.3** Ergebnisse mit Nachkommastellen

Zur Erläuterung:

- Das richtige Ergebnis erscheint, sobald entweder die Zahl `20` oder die Zahl `8` als `20.0` bzw. `8.0` eingegeben werden.

### B.1.4 Rangfolge der Rechenoperatoren, Klammern

**Rangfolge**

Es gilt, wie in der Mathematik, Punkt- vor Strichrechnung. Multiplikation und Division haben also Vorrang vor Addition und Subtraktion. Das Setzen von Klammern führt dazu, dass die Ausdrücke innerhalb der Klammern als Erstes berechnet werden. Beides wird an den folgenden Beispielen gezeigt:

```
>>> 7+2*3
13
>>> (7+2)*3
27
```

**Listing B.4** Rangfolge der Operatoren

Zur Erläuterung: Bei der ersten Rechnung werden zunächst `2` und `3` multipliziert, anschließend wird `7` addiert. Bei der zweiten Rechnung werden zunächst `7` und `2` addiert, anschließend wird mit `3` multipliziert.

#### Übung B01

Es wird vorausgesetzt, dass Python wie angegeben installiert wurde. Rufen Sie die Entwicklungsumgebung IDLE auf. Ermitteln Sie die mathematisch richtigen Ergebnisse der folgenden vier Aufgaben:

```
13 - 5 * 2 + 12 / 6
7 / 2 - 5 / 4
(12 - 5 * 2) / 4
(1 / 2 - 1 / 4 + (4 + 3) / 8) * 2
```

Die Lösungen aller Übungsaufgaben finden Sie in Kapitel O am Ende des Buches.

### B.1.5 Variablen, Zuweisung

**Variablen zuweisen**

Bisher wurde nur mit Zahlen gerechnet. Python kann, wie jede andere Programmiersprache auch, Zahlen in Variablen speichern, falls sie im Verlauf einer Berechnung mehrmals benötigt werden. Dies soll nachfolgend am Beispiel der Umrechnung von Euro in DM gezeigt werden.

```
>>> ed=1.95583
>>> 2*ed
3.9116599999999999
>>> 5*ed
9.7791499999999996
>>> 22.5*ed
44.006174999999999
>>> 2.35*ed
4.5962005000000001
```

**Listing B.5** Zuweisung an Variablen

Zur Erläuterung: Bekanntlich entspricht 1 Euro 1.95583 DM. Dieser Umrechnungsfaktor wird zunächst in der Variablen `ed` gespeichert. Dadurch wird die vereinfachte Durchführung von mehreren Euro-Umrechnungen nacheinander erleichtert. Anschließend werden die DM-Beträge für 2 Euro, 5 Euro, 22.50 Euro und 2.35 Euro berechnet und ausgegeben.

Die Speicherung des Umrechnungsfaktors geschieht mit einer Zuweisung (`ed = 1.95583`). Das heißt, die Variable `ed` bekommt den Wert, der rechts vom Gleichheitszeichen steht (`1.95583`).

In Python-Variablen können ganze Zahlen, Zahlen mit Stellen hinter dem Komma, Zeichenketten (also Texte) und andere Objekte eines Programms mit Hilfe von Zuweisungen gespeichert werden. Eine Python-Variable hat keinen festgelegten Typ für den gesamten Verlauf eines Programms. Sie bekommt ihren Typ durch die Zuweisung eines Wertes.

Der Name einer Variablen kann frei gewählt werden. Es gelten die folgenden Regeln:

- Er kann aus den Buchstaben a bis z, A bis Z, Ziffern oder dem Zeichen _ (=Unterstrich) bestehen.
- Er darf nicht mit einer Ziffer beginnen.

- Er darf keinem reservierten Wort der Programmiersprache Python entsprechen (eine Liste der reservierten Worte findet sich im Anhang).
- Die Groß- und Kleinschreibung ist zu beachten: Namen, Anweisungen usw. sollten immer genau so geschrieben werden wie vorgegeben.

Es wäre natürlich sinnvoll, den DM-Betrag so auszugeben, dass auf zwei Stellen hinter dem Komma gerundet wird. Auf eine solche formatierte Ausgabe wird später noch eingegangen.

#### Übung B02

1 Euro entspricht 6.55957 französischen Franc. Berechnen Sie für die folgenden Euro-Beträge den Franc-Betrag: 5 Euro, 20 Euro, 9.95 Euro und 92.70 Euro. Vereinfachen Sie Ihre Berechnungen, indem Sie den Umrechnungsfaktor zunächst in einer Variablen speichern.

## B.2 Ein erstes Programm

**Eingabe, Verarbeitung, Ausgabe**

Es soll nun mit Hilfe von Python ermöglicht werden, einen beliebigen Euro-Betrag, den ein Anwender eingibt und der Ihnen als Entwickler nicht bekannt ist, in DM umzurechnen. Dazu ist es notwendig, ein kleines Python-Programm einzugeben, abzuspeichern und aufzurufen. Dieser Vorgang wird nachfolgend ausführlich erklärt. Die einzelnen Schritte sind später bei jedem Python-Programm auszuführen.

Dabei wird bereits das Prinzip der klassischen Programmierung verdeutlicht:

- **E**ingabe (der Anwender gibt Daten über die Tastatur ein)
- **V**erarbeitung (das Programm verarbeitet diese und andere Daten)
- **A**usgabe (das Programm gibt das Ergebnis der Verarbeitung auf dem Bildschirm aus)

### B.2.1 Eingabe eines Programms

Zunächst wird in der Entwicklungsumgebung IDLE im Menü `File` der Menübefehl `New Window` aufgerufen. Es öffnet sich ein neues Fenster mit dem Titel `Untitled`. Das Hauptfenster mit dem Titel `Python Shell` rückt in den Hintergrund. In dem neuen Fenster wird das Programm wie nachfolgend eingegeben:

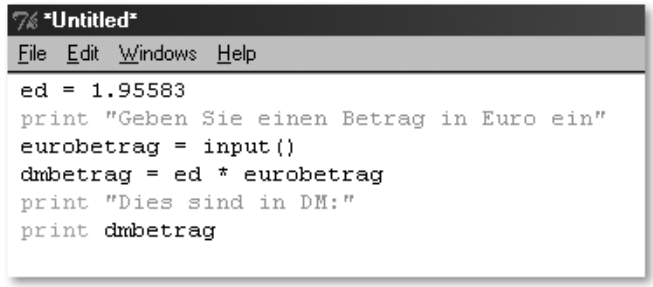



**Abbildung B.2** Eingabe eines Programms

**print, input()** Die Anweisung `print` gibt Text oder Werte von Variablen auf dem Bildschirm aus. Die Funktion `input()` veranlasst den Anwender des Programms, etwas einzugeben. Der eingegebene Wert sollte einer Variablen zugewiesen werden.

Zur Erläuterung der einzelnen Programmzeilen:

- In der ersten Zeile wird der Umrechnungsfaktor in der Variablen `ed` gespeichert.
- In der zweiten Zeile wird mit Hilfe der Anweisung `print` ein Informationstext für den Anwender auf den Bildschirm geschrieben, damit er weiß, was er machen soll.
- In der dritten Zeile wird die Funktion `input()` aufgerufen, d.h., das Programm bleibt stehen und wartet auf eine Eingabe des Anwenders über die Tastatur. An dieser Stelle sollte der Anwender eine Zahl eingeben (mit oder ohne Stellen hinter dem Dezimalpunkt) und die ↵-Taste betätigen. Die eingegebene Zahl wird in der Variablen `eurobetrag` gespeichert.
- In der vierten Zeile wird mit Hilfe des eingegebenen Betrags und des Umrechnungsfaktors der DM-Betrag berechnet und gespeichert. Die Speicherung des Umrechnungsfaktors geschieht mit der Zuweisung `dmbetrag = ed * eurobetrag`, d.h., die Variable `dmbetrag` bekommt den Wert des Rechenergebnisses `ed * eurobetrag`.
- In der fünften Zeile wird mit einem kurzen Text die nachfolgende Ergebnisausgabe eingeleitet.
- In der sechsten und letzten Zeile wird das Ergebnis, d.h. der berechnete DM-Betrag auf dem Bildschirm, ausgegeben.

Dieses Programm wird bei einem Aufruf durch den Anwender Zeile für Zeile abgearbeitet.

- Falls der Anwender etwas Falsches eingibt, z.B. keine Zahl, sondern Zeichen, wird das Programm mit einer Fehlermeldung beendet. Zunächst soll aber vereinfacht davon ausgegangen werden, dass der Anwender keine falschen Eingaben vornimmt. Die Vermeidung bzw. das Abfangen von Fehlern wird im Abschnitt B.5, Fehler und Ausnahmen, beschrieben.

### B.2.2 Speicherung eines Programms

**Speichern**

Nach der Eingabe muss das Programm gespeichert werden. Dazu wird im aktuellen Fenster im Menü `File` der Menübefehl `Save` aufgerufen. Das Programm wird in der Datei mit dem Namen `b03.py` gespeichert, wie nachfolgend gezeigt:

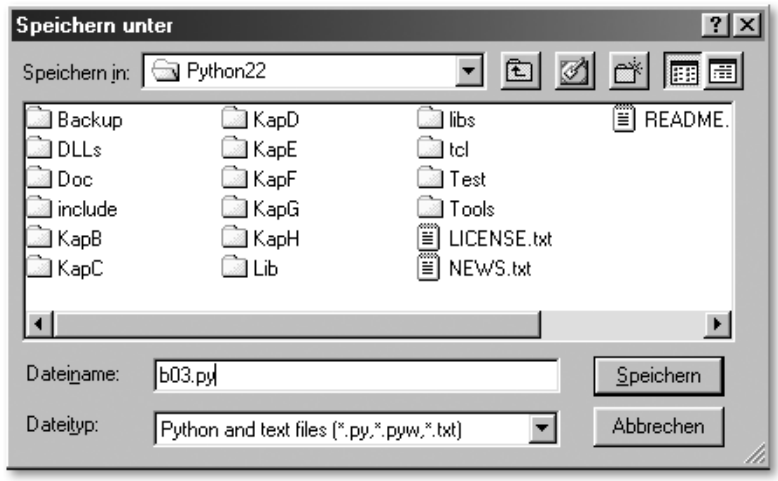



**Abbildung B.3** Speicherung des Programms

Nach Betätigung der Taste »Speichern« ist die Speicherung vollzogen. Der Name, hier `b03`, kann frei gewählt werden. Die Endung `py` ist für Python-Programme allerdings zwingend vorgeschrieben.

Das Fenster mit dem Programm hat nun das nachfolgend dargestellte Aussehen. In der Titelzeile ist der Dateiname bzw. der vollständige Pfadname erkennbar.

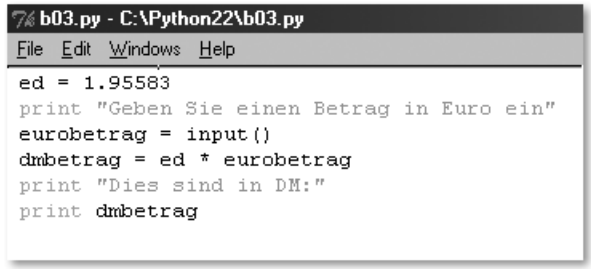



**Abbildung B.4** Gespeichertes Programm

### B.2.3 Aufruf des Programms über die Python Shell

**Aufrufen**

Zum Aufruf des Programms wird im Menü `Edit` der Menübefehl `Run Script` ausgeführt (Tastenkombination: Strg + F5). Das Hauptfenster der Entwicklungsumgebung (die Python Shell) rückt in den Vordergrund, der Informationstext für den Anwender erscheint und das Programm wartet auf eine Eingabe:

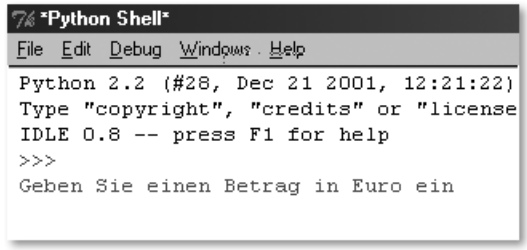



**Abbildung B.5** Eingabeposition

Nach der Eingabe einer Zahl (hier 45.2) und der Betätigung der ↵-Taste erscheint das Ergebnis. Damit endet das Programm:

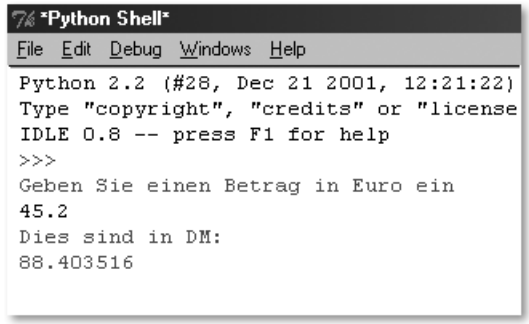



**Abbildung B.6** Ergebnis des Programms

Über die Task-Leiste kann man das Programmfenster wieder in den Vordergrund rücken.

Falls man Veränderungen am Programm vornehmen möchte, so ist anschließend eine erneute Speicherung vorzunehmen. Dabei wird nicht mehr nach dem Dateinamen gefragt, da dieser bereits bekannt ist.

Falls ein anderes Programm entwickelt werden soll, so sollte das Programmfenster geschlossen und ein neues Programmfenster über das Menü `File`, `New Window` geöffnet werden.

### B.2.4 Aufruf des Programms aus der Kommandozeile

**Kommandozeile**

Das Programm in der Datei `b03.py` ist ein Kommandozeilen-Programm, d.h., es erwartet eine Eingabe über die Tastatur und erzeugt eine einfache Ausgabe auf dem Bildschirm ohne Erzeugung einer eigenen Benutzeroberfläche. Im Kapitel H, Programmierung von Benutzeroberflächen, wird gezeigt, wie man mit Hilfe von Python Programme mit grafischen Benutzeroberflächen erzeugen kann.

Für einen Aufruf des Programms aus der Kommandozeile muss unter Windows zunächst auf den DOS-Bildschirm gewechselt werden. Dazu hat man die beiden folgenden Möglichkeiten:

- `Startmenü, Programme, MS-DOS-Eingabeaufforderung`
- `Startmenü, Ausführen`, Eingabe und Bestätigung (mit OK) des Befehls `command`, wie nachfolgend dargestellt:

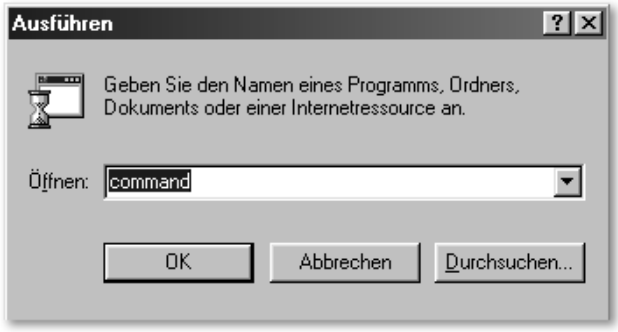



**Abbildung B.7** Wechsel zur Kommandozeile

Es erscheint die MS-DOS-Eingabeaufforderung, hier für MS Windows 98:


**Abbildung B.8** MS-DOS-Eingabeaufforderung

Mit Eingabe des Befehls `cd\python22` wechselt man in das Verzeichnis `c:\python22`, in dem Python installiert und in dem das Programm `b03.py` abgelegt wurde:


**Abbildung B.9** Wechsel in das Python-Installationsverzeichnis

**Aufrufen**

Anschließend veranlasst man die Ausführung des Programms mit der Anweisung `python b03.py`. Der Informationstext für den Anwender erscheint und das Programm wartet auf eine Eingabe:


**Abbildung B.10** Eingabeposition

Nach der Eingabe einer Zahl (hier 45.2) und der Betätigung der ↵-Taste erscheint das Ergebnis. Damit endet das Programm:


**Abbildung B.11** Programm beendet

Dieses DOS-Fenster kann man für spätere Aufrufe von Python-Programmen geöffnet lassen oder mit Eingabe des Befehls `exit` wieder schließen.

**Kein Doppelklick**

Python-Programme können auch mit einem einfachen Doppelklick auf den Dateinamen im Windows-Explorer gestartet werden. Es öffnet sich ebenfalls ein DOS-Fenster, das sich allerdings nach erfolgter Eingabe unmittelbar wieder schließt, sodass das Ergebnis der Berechnung nicht zu sehen ist. Abhilfe könnte durch eine zusätzliche Eingabeaufforderung am Ende des Programms geschaffen werden. Diese hätte nur die Aufgabe, das Fenster sichtbar zu lassen. Allerdings ist diese Methode nicht empfehlenswert.

### Übung B04

Schreiben Sie ein Programm zur Eingabe und Umrechnung von beliebigen Euro-Beträgen in französische Franc. Speichern Sie das Programm in der Datei `b04.py`. Rufen Sie das Programm sowohl unter der Entwicklungsumgebung IDLE als auch unter DOS auf.

### Übung B05

Schreiben Sie ein Programm zur Berechnung des monatlich zu zahlenden Steuerbetrags. Der Anwender soll zunächst dazu aufgefordert werden, sein monatliches Bruttogehalt einzugeben. Anschließend werden (hier sehr vereinfacht) 18% dieses Betrags berechnet und ausgegeben. Speichern Sie das Programm in der Datei `b05.py`.

### B.2.5 Kommentare

**Kommentieren**

Bei umfangreicheren Programmen erweist es sich als sehr nützlich, Kommentare zur Erläuterung in den Programmtext einzufügen. Kommentare werden durch das Zeichen # (Doppelkreuz) eingeleitet und gehen bis zum Zeilenende. Sie werden nicht als Programmschritte betrachtet und gelangen nicht zur Ausführung. Nachfolgend wurde das Programm in der Datei `b03.py` um Kommentare erweitert.

```
ed = 1.95583    # Umrechnungsfaktor

# Eingabe
print "Geben Sie einen Betrag in Euro ein"
eurobetrag = input()

# Verarbeitung
dmbetrag = ed * eurobetrag

# Ausgabe
print "Dies sind in DM:"
print dmbetrag
```

**Listing B.6** Datei b03.py

### B.2.6 Verkettung von Ausgaben

**Mehrere Ausgaben**

Mit Hilfe der Anweisung `print` können auch mehrere Ausgaben in einer Zeile ausgegeben werden. Die einzelnen Teile der Ausgabe werden durch Kommata voneinander getrennt. Nachfolgend wurde die Ausgabe des Programms in der Datei `b03.py` geändert (man beachte besonders die letzte Zeile).

```
ed = 1.95583    # Umrechnungsfaktor

# Eingabe
print "Geben Sie einen Betrag in Euro ein"
eurobetrag = input()

# Verarbeitung
dmbetrag = ed * eurobetrag

# Ausgabe
print "Dies sind in DM:", dmbetrag
```

**Listing B.7** Datei b03.py (inklusive Verkettung)

Der Ausgabetext wird dann in einer Zeile dargestellt:

```
>>>
Geben Sie einen Betrag in Euro ein
45.2
Dies sind in DM: 88.403516
```

## B.3 Verzweigungen

**Bedingte Ausführung**

In den bisherigen Programmen wurden alle Anweisungen der Reihe nach ausgeführt. Zur Steuerung eines Programmablaufs werden allerdings häufig Verzweigungen benötigt. Innerhalb des Programms wird dann aufgrund einer Bedingung entschieden, welcher Zweig des Programms ausgeführt werden soll.

### B.3.1 Vergleichsoperatoren

Bedingungen werden mit Hilfe von Vergleichsoperatoren formuliert. Es folgt eine Tabelle der Vergleichsoperatoren und ihrer Bedeutung:

| Operator | Bedeutung |
|---|---|
| > | größer als |
| < | kleiner als |
| >= | größer als oder gleich |
| <= | kleiner als oder gleich |
| == | gleich |
| != | ungleich |

**Tabelle B.1** Vergleichsoperatoren

### B.3.2 Einfache Verzweigung

**if-else**

Im folgenden Beispiel wird untersucht, ob eine Variable größer als 0 ist. Wenn dies der Fall ist, so wird ausgegeben: »Diese Zahl ist größer als 0«, ansonsten wird ausgegeben: »Diese Zahl ist kleiner oder gleich 0«. Es wird also nur eine der beiden Anweisungen ausgeführt.

```
x = 12
if x > 0:
    print "Diese Zahl ist größer als 0"
```

```
else:
    print "Diese Zahl ist kleiner oder gleich 0"
```

**Listing B.8** Datei bo6.py

Zur Erläuterung:

- In der ersten Zeile bekommt die Variable `x` den Wert `12` zugewiesen.
- In der zweiten Zeile wird mit Hilfe der Anweisung `if` eine Verzweigung eingeleitet. Danach wird eine Bedingung formuliert (hier `x>0`), die entweder wahr oder falsch ergibt. Anschließend folgt das Zeichen `:` (Doppelpunkt).

**Einrücken**

- Es folgen eine oder mehrere Anweisungen, die nur ausgeführt werden, falls die Bedingung wahr ergibt. Es ist erforderlich, die Anweisungen innerhalb des so genannten `if`-Zweiges mit Hilfe der ⇥-Taste einzurücken. Ansonsten kann Python die Zugehörigkeit zur Verzweigung nicht erkennen. Gleichzeitig wird das Programm dadurch übersichtlicher.
- In der vierten Zeile wird mit Hilfe der Anweisung `else` der alternative Teil der Verzweigung eingeleitet. Es folgt ebenfalls ein Doppelpunkt.
- Es folgen eine oder mehrere Anweisungen, die nur ausgeführt werden, falls die Bedingung falsch ergibt. Auch diese Anweisungen müssen eingerückt werden.

Die Ausgabe des Programms:

```
>>>
Diese Zahl ist größer als 0
```

Ein weiteres Beispiel zur einfachen Verzweigung:

Es soll wiederum eine Umrechnung eines Betrags von Euro in eine andere Währung erfolgen. Der Anwender entscheidet, ob dies DM oder französische Franc (FF) sein sollen.

```
print "Geben Sie einen Betrag in Euro ein"
eurobetrag = input()

print "Umrechnung in DM (1) oder in Franc (2)?"
antwort = input()

if antwort == 1:
    ed = 1.95583
    ubetrag = ed * eurobetrag
```

```
else:
    ef = 6.55957
    ubetrag = ef * eurobetrag

print "Ergebnis:"
print ubetrag
```

**Listing B.9** Datei b07.py

Zur Erläuterung: Es werden zwei Zahleneingaben gefordert, zum einen der Betrag in Euro, zum anderen soll eine Zahl eingegeben werden, mit deren Hilfe das Programm entscheiden kann, in welche Währung umgerechnet werden soll.

- Falls der Anwender eine 1 eingibt, so wird der Umrechnungsfaktor für DM zugewiesen und in DM umgerechnet.
- Falls der Anwender eine 2 (oder eine andere Zahl) eingibt, so wird der Umrechnungsfaktor für FF zugewiesen und in FF umgerechnet.

Nach der Verzweigung wird das Ergebnis ausgegeben, unabhängig davon, auf welche Art und Weise es berechnet wurde. Die zusätzlichen Leerzeilen im Programm sind nicht unbedingt erforderlich. Sie dienen nur der besseren Übersicht.

Eine mögliche Ausgabe des Programms:

```
>>>
Geben Sie einen Betrag in Euro ein
24.5
Umrechnung in DM (1) oder in Franc (2)?
1
Ergebnis:
47.917835
```

### Übung B08

Das Programm zur Berechnung des monatlich zu zahlenden Steuerbetrags soll verändert werden. Der Anwender soll zunächst dazu aufgefordert werden, sein monatliches Bruttogehalt einzugeben. Falls das Gehalt über 2.500 Euro liegt, so sind 22% Steuer zu zahlen, ansonsten 18% Steuer. Speichern Sie das Programm in der Datei `b08.py`.

Zur Erläuterung: Es ist nur *eine* Eingabe erforderlich. Innerhalb des Programms muss aufgrund des Gehaltes entschieden werden, welcher Steuersatz genommen wird.

### B.3.3 Mehrfache Verzweigung

In vielen Anwendungsfällen gibt es mehr als zwei Möglichkeiten, zwischen denen zu entscheiden ist. Dazu wird eine mehrfache Verzweigung benötigt.

**if-elif-else**

Im folgenden Beispiel wird untersucht, ob eine Variable größer als 0, kleiner als 0 oder gleich 0 ist. Es wird eine entsprechende Meldung ausgegeben:

```
x = -12

if x > 0:
    print "Diese Zahl ist größer als 0"
elif x < 0:
    print "Diese Zahl ist kleiner als 0"
else:
    print "Diese Zahl ist gleich 0"
```

**Listing B.10** Datei b09.py

Zur Erläuterung:

- Mit der Anweisung `if` wird die Verzweigung eingeleitet. Falls `x` größer als 0 ist, werden die nachfolgenden, eingerückten Anweisungen ausgeführt.
- Nach der Anweisung `elif` wird eine weitere Bedingung formuliert. Diese wird nur untersucht, falls die erste Bedingung (nach dem `if`) nicht zutraf. Falls `x` kleiner als 0 ist, werden die nachfolgenden, eingerückten Anweisungen ausgeführt.
- Die Anweisungen nach der Zeile mit der `else`-Anweisung werden nur durchgeführt, falls keine der beiden vorherigen Bedingungen zutraf (hinter dem `if` und hinter dem `elif`). In diesem Fall bedeutet das, dass `x` gleich 0 ist, da es nicht größer und nicht kleiner als 0 ist.

Die Ausgabe des Programms:

```
>>>
Diese Zahl ist kleiner als 0
```

Innerhalb einer Verzweigung können mehrere `elif`-Anweisungen vorkommen. Diese werden der Reihe nach untersucht, bis das Programm zu der Bedingung kommt, die zutrifft. Die weiteren `elif`-Anweisungen oder eine `else`-Anweisung werden dann nicht mehr beachtet.

Es ist nicht zwingend notwendig, dass eine `else`-Anweisung bei einer einfachen Verzweigung oder bei einer mehrfachen Verzweigung vorkommt.

### Übung B10

Das Programm zur Berechnung des monatlich zu zahlenden Steuerbetrags soll weiter verändert werden (Datei `b10.py`). Der Anwender soll zunächst dazu aufgefordert werden, sein monatliches Bruttogehalt einzugeben. Anschließend soll das Gehalt nach der folgenden Steuertabelle berechnet werden:

| Gehalt | Steuersatz |
|---|---|
| mehr als 4.000 Euro | 26% |
| weniger als 2.500 Euro | 18% |
| dazwischen | 22% |

**Tabelle B.2** Steuertabelle für die Übung

## B.3.4 Logische Operatoren

**Logische Verknüpfung**

Mit Hilfe der logischen Operatoren (`and`, `or` und `not`) können mehrere Bedingungen miteinander verknüpft werden.

- Eine Bedingung, die aus einer oder mehreren Einzelbedingungen besteht, die mit dem Operator `and` (=und) verknüpft sind, ergibt wahr, wenn *jede* der Einzelbedingungen wahr ergibt.
- Eine Bedingung, die aus einer oder mehreren Einzelbedingungen besteht, die mit dem Operator `or` (=oder) verknüpft sind, ergibt wahr, wenn *eine* der Einzelbedingungen wahr ergibt.
- Der Operator `not` (=nicht) kehrt den Wahrheitswert einer Bedingung um, d.h., eine falsche Bedingung wird wahr, eine wahre Bedingung wird falsch.

Der gleiche Zusammenhang nachfolgend in einer Wahrheitswerttabelle dargestellt:

| Bedingung 1 | Operator | Bedingung 2 | gesamte Bedingung |
|---|---|---|---|
| wahr | `and` | wahr | wahr |
| wahr | `and` | falsch | falsch |
| falsch | `and` | wahr | falsch |
| falsch | `and` | falsch | falsch |
| wahr | `or` | wahr | wahr |
| wahr | `or` | falsch | wahr |
| falsch | `or` | wahr | wahr |
| falsch | `or` | falsch | falsch |
| | `not` | wahr | falsch |
| | `not` | falsch | wahr |

**Tabelle B.3** Ergebnis beim Einsatz von logischen Operatoren

Ein Beispiel:

```
x = 12
y = 15
z = 18

# Bedingung 1
if x<y and x<z:
    print "x ist die kleinste Zahl"
else:
    print "x ist nicht die kleinste Zahl"

# Bedingung 2
if y>x or y>z:
    print "y ist nicht die kleinste Zahl"

# Bedingung 3
if not z>y:
    print "z ist nicht größer als y"
else:
    print "z ist größer als y"
```

**Listing B.11** Datei b11.py

Zur Erläuterung:

- Die Bedingung 1 ergibt wahr, wenn `x` kleiner als `y` *und* kleiner als `z` ist. Dies trifft bei den gegebenen Anfangswerten zu.
- Die Bedingung 2 ergibt wahr, wenn `y` größer als `x` *oder* `y` größer als `z` ist. Die erste Bedingung trifft zu, also ist die gesamte Bedingung wahr.
- Die Bedingung 3 ergibt wahr, wenn `z` *nicht* größer als `y` ist. Dies trifft nicht zu, denn `z` ist größer als `y`, also wird die Anweisung im `else`-Zweig ausgeführt.

Die Ausgabe des Programms:

```
>>>
x ist die kleinste Zahl
y ist nicht die kleinste Zahl
z ist größer als y
```

### Übung B12

Das Programm zur Berechnung des monatlich zu zahlenden Steuerbetrags soll wiederum verändert werden. Die Steuertabelle sieht nun wie folgt aus:

| Gehalt | Familienstand | Steuersatz |
|---|---|---|
| > 4.000 Euro | ledig | 26% |
| > 4.000 Euro | verheiratet | 22% |
| <= 4.000 Euro | ledig | 22% |
| <= 4.000 Euro | verheiratet | 18% |

**Tabelle B.4** Steuertabelle für die Übung

Der Anwender soll neben seinem Gehalt auch seinen Familienstand nennen. Dabei soll z.B. der Familienstand »ledig« mit 1 und der Familienstand »verheiratet« mit 2 angegeben werden. Verwenden Sie logische Operatoren zur Zusammenfassung von Bedingungen innerhalb einer Zeile der Steuertabelle bzw. zur Zusammenfassung der Bedingungen aus mehreren Zeilen der Steuertabelle.

Diese Aufgabe ist (wie alle Programmieraufgaben) auf mehrere Arten lösbar. Sie finden in Kapitel O, Lösungen, zwei alternative Möglichkeiten (Dateien `b12.py` und `b13.py`). Betrachten Sie diese Lösungen als Ansätze zur Programmoptimierung.

### B.3.5 Geschachtelte Verzweigung

**Verschachtelung**

Verzweigungen können auch geschachtelt werden. Dies bedeutet, dass eine Verzweigung eine weitere Unterverzweigung beinhaltet. Ein Beispiel:

```
x = -6
if x > 0:                # äußeres if
    print "Diese Zahl ist größer als 0"
else:                    # äußeres else
    if x < 0:            # inneres if
        print "Diese Zahl ist kleiner als 0"
    else:                # inneres else
        print "Diese Zahl ist gleich 0"
```

**Listing B.12** Datei b14.py

Zur Erläuterung:

**Äußeres if**

- Mit der äußeren `if`-Anweisung wird die geschachtelte Verzweigung eingeleitet. Falls `x` größer als 0 ist, werden die nachfolgenden, einfach eingerückten Anweisungen ausgeführt.

**Inneres if**

- Falls die Bedingung hinter dem äußeren `if` nicht zutrifft, so wird die innere `if`-Anweisung hinter dem äußeren `else` untersucht.
- Trifft sie zu, so werden die nachfolgenden, doppelt eingerückten Anweisungen ausgeführt.
- Trifft die innere `if`-Anweisung nicht zu, dann werden die dem inneren `else` nachfolgenden, doppelt eingerückten Anweisungen ausgeführt.

Zu beachten ist besonders die doppelte Einrückung, damit die richtige Verzweigungstiefe von Python erkannt werden kann.

Die Ausgabe des Programms:

```
>>>
Diese Zahl ist kleiner als 0
```

#### Übung B15

Lösen Sie das Problem aus Übung B12 mit einer geschachtelten Verzweigung.

### B.3.6 Rangfolge der bisher genutzten Operatoren

In vielen Ausdrücken treten mehrere Operatoren auf. Bisher sind dies Rechenoperatoren, Vergleichsoperatoren und logische Operatoren. Damit Python weiß, in welcher Reihenfolge die einzelnen Teilschritte bearbeitet werden sollen, gibt es eine festgelegte Rangfolge der Operatoren. Die Teilschritte, bei denen Operatoren mit einem höheren Rang beteiligt sind, werden als erste ausgeführt.

**Rangfolge der Operatoren**

Nachfolgend die Rangfolge der bisher verwendeten Operatoren in Python, beginnend mit den Operatoren, die den höchsten Rang haben. Operatoren, die in der gleichen Reihe stehen, haben den gleichen Rang. Teilschritte, in denen mehrere dieser Operatoren gleichen Ranges stehen, werden von links nach rechts ausgeführt.

| Operator | Bedeutung |
|---|---|
| + - | positives Vorzeichen einer Zahl, negatives Vorzeichen einer Zahl |
| * / | Multiplikation, Division |
| + - | Addition, Subtraktion |
| < <= > >= == != | kleiner, kleiner oder gleich, größer, größer oder gleich, gleich, ungleich |
| not | logische Verneinung |
| and | logisches und |
| or | logisches oder |

**Tabelle B.5** Rangfolge der bisher genutzten Operatoren

Es gibt noch eine Reihe weiterer Operatoren. Im Anhang findet sich eine vollständige Rangfolge der Operatoren in Python.

## B.4 Schleifen

Neben der Verzweigung gibt es noch eine weitere wichtige Struktur zur Steuerung von Programmen: die Schleife. Mit Hilfe einer Schleife kann man die wiederholte Ausführung eines Programmschrittes ermöglichen.

**Wiederholung**

Es muss zwischen zwei Typen von Schleifen unterschieden werden, der `for`-Schleife und der `while`-Schleife. Als einfaches Unterscheidungsmerkmal sollen die folgenden Regeln dienen:

**for** ▶ Man verwendet eine `for`-Schleife, falls ein Programmschritt für eine regelmäßige, zu diesem Zeitpunkt bekannte Folge von Werten wiederholt ausgeführt werden soll.

**while** ▶ Man verwendet eine `while`-Schleife, falls sich erst durch Eingaben des Anwenders ergibt, ob und wie oft ein Programmschritt wiederholt ausgeführt werden soll.

Eine `for`-Schleife wird auch Zähl-Schleife genannt, eine `while`-Schleife bedingungsgesteuerte Schleife.

### B.4.1 for-Schleife

Die `for`-Schleife wird anhand des folgenden Anwendungsbeispieles erläutert:

```
for i in 2, -6, 7.5, 22:
    print "Zahl:", i
    print "Quadrat:", i*i
```

**Listing B.13** Datei b16.py

**Liste** Die erste Zeile muss wie folgt gelesen werden: Führe die nachfolgenden, eingerückten Anweisungen für jede Zahl aus der Liste `2`, `-6`, `7.5` und `22` aus. Nenne diese Zahl in diesen Anweisungen `i`. Natürlich kann auch eine andere Variable (statt `i`) als so genannte Schleifen-Variable genutzt werden.

Die genannten Zahlen werden zusammen mit einem kurzen Informationstext ausgegeben bzw. quadriert und ausgegeben. Das Ergebnis:

```
>>>
Zahl: 2
Quadrat: 4
Zahl: -6
Quadrat: 36
Zahl: 7.5
Quadrat: 56.25
Zahl: 22
Quadrat: 484
```

**range()** Im dargestellten Fall handelt es sich um eine unregelmäßige Liste von Zahlen. Meist werden Schleifen allerdings für regelmäßige Listen von Zahlen genutzt. Dabei erweist sich der Einsatz der Funktion `range()` als sehr nützlich. Ein Beispiel:

```
for i in range(3,11,2):
    print "Zahl:", i
    print "Quadrat:", i*i
```

**Listing B.14** Datei b17.py

Range heißt übersetzt Bereich. Innerhalb der Klammern hinter `range` können bis zu drei ganze Zahlen, durch Komma getrennt, eingetragen werden:

- Die erste ganze Zahl (hier `3`) gibt den Beginn des Bereiches an, für den die nachfolgenden Anweisungen ausgeführt werden sollen.
- Die zweite ganze Zahl (hier `11`) kennzeichnet das Ende des Bereiches. Es ist die erste Zahl, für die die Anweisungen *nicht* mehr ausgeführt werden.
- Die dritte ganze Zahl (hier `2`) gibt die Schrittweite für die Schleife an. Die Zahlen, für die die Anweisungen ausgeführt werden, haben also jeweils einen Abstand von +2 zueinander.

Die Funktion `range()` ergibt somit eine regelmäßige Liste von ganzen Zahlen (3, 5, 7, 9). Das Ergebnis:

```
>>>
Zahl: 3
Quadrat: 9
Zahl: 5
Quadrat: 25
Zahl: 7
Quadrat: 49
Zahl: 9
Quadrat: 81
```

Einige allgemeine Hinweise zu `for`-Schleifen:

- Eine oder mehrere der drei Zahlen können auch negativ sein.
- Man sollte darauf achten, dass man sinnvolle Zahlenkombinationen angibt. Die Angabe `range(3,-11,2)` ist nicht sinnvoll, da man nicht von der Zahl +3 in Schritten von +2 zur Zahl -11 gelangt. Python fängt solche Schleifen ab und lässt sie nicht ausführen.
- Ähnlich wie bei der `if`-Anweisung ist darauf zu achten, dass ein : (Doppelpunkt) nach der Liste von Zahlen bzw. Bereichsangaben folgt.

- Ebenfalls ähnlich wie bei der `if`-Anweisung müssen die Anweisungen, die mehrmals ausgeführt werden sollen, eingerückt geschrieben werden. Ansonsten kann Python die Zugehörigkeit zur Schleife nicht erkennen. Gleichzeitig wird das Programm dadurch übersichtlicher.

### Übung B18

Ermitteln Sie durch Überlegung (nicht durch einen einfachen Aufruf) die Ausgabe des folgenden Programms.

```
print "Schleife 1"
for i in 2, 3, 6.5, -7:
    print i

print "Schleife 2"
for i in range(3,11,3):
    print i

print "Schleife 3"
for i in range(-3,14,4):
    print i

print "Schleife 4"
for i in range(3,-11,-3):
    print i
```

**Listing B.15** Datei b18.py

### Übung B19

Schreiben Sie ein Programm, das die folgende Ausgabe erzeugt. Beachten Sie, dass für dieses Programm keine Eingabe durch den Anwender notwendig ist. Es handelt sich um eine regelmäßige Liste von Zahlen, für die der DM-Betrag durch Umrechnung mit dem Faktor 1.95583 ermittelt werden soll.

```
>>>
10 Euro =  19.5583 DM
20 Euro =  39.1166 DM
30 Euro =  58.6749 DM
40 Euro =  78.2332 DM
50 Euro =  97.7915 DM
60 Euro =  117.3498 DM
70 Euro =  136.9081 DM
```

```
80 Euro =  156.4664 DM
90 Euro =  176.0247 DM
100 Euro =  195.583 DM
```

### B.4.2 Exkurs: Formatierte Ausgabe

Formatierung

Schleifen unterstützen u.a. die Durchführung von Berechnungen für eine Liste von Werten. Bei der Ausgabe steht man häufig vor dem Problem, diese Werte regelmäßig, in Tabellenform, anzuordnen, dabei hilft die formatierte Ausgabe.

Die Anweisung `print` bietet die Möglichkeit:

- die Mindest-Ausgabebreite der Zahlen zu bestimmen
- die Anzahl der Nachkommastellen zu bestimmen

Das nachfolgende Programm soll dies an einigen Beispielen verdeutlichen:

```
print "ganze Zahlen:"
for zahl in range(2,13,2):
    quadrat = zahl * zahl
    print "%3i %4i" % (zahl, quadrat)

print
print "Zahlen mit Nachkommastellen:"
for zahl in range(2,13,2):
    zehntel = zahl/10.0
    quadrat = zehntel * zehntel
    print "%8.3f %8.3f" % (zehntel, quadrat)
```

**Listing B.16** Datei b20.py

Zur Erläuterung:

Innerhalb des ersten Blockes wird eine Liste von ganzen Zahlen quadriert und ausgegeben. Die Zeile, in der die Zahlen ausgegeben werden, ist wie folgt zu lesen:

- Es werden zwei ganze Zahlen ausgegeben, die erste mit der Mindestbreite 3, die zweite mit der Mindestbreite 4. Dabei steht `%i` als Platzhalter für eine ganze Zahl, dazwischen steht die gewünschte Breite. Hinter dem einzelnen Prozentzeichen stehen die Variablen bzw. Werte, die ausgegeben werden sollen, in Klammern.

Innerhalb des zweiten Blockes wird eine Liste von Zahlen mit Nachkommastellen quadriert und ausgegeben. Die Zeile, in der die Zahlen ausgegeben werden, ist wie folgt zu lesen:

- Es werden zwei Zahlen mit Nachkommastellen ausgegeben, beide mit der Mindestbreite 8, davon 3 Stellen hinter dem Komma. Bei der Gesamtstellenzahl ist der Dezimalpunkt mitzurechnen, also verbleiben in diesem Fall noch 4 Stellen vor dem Komma. Hinter dem einzelnen Prozentzeichen stehen wiederum die Variablen bzw. Werte, die ausgegeben werden sollen, in Klammern.

Falls man bei einer Zahl mit Nachkommastellen die Formatierung `%.3f` angibt, so wird nur die Anzahl der Stellen hinter dem Komma festgelegt, nicht die Mindestbreite.

Die Ausgabe des Programms:

```
>>>
ganze Zahlen:
  2    4
  4   16
  6   36
  8   64
 10  100
 12  144

Zahlen mit Nachkommastellen:
   0.200    0.040
   0.400    0.160
   0.600    0.360
   0.800    0.640
   1.000    1.000
   1.200    1.440
```

### Übung B21

Schreiben Sie das Programm aus der vorherigen Übung so um, dass die nachfolgende Ausgabe (mit Überschrift) erzeugt wird. Die Beträge für Euro und DM sollen jeweils mit zwei Nachkommastellen angezeigt und am Dezimalpunkt ausgerichtet werden.

```
>>>
    Euro      DM
   10.00   19.56
```

```
  20.00   39.12
  30.00   58.67
  40.00   78.23
  50.00   97.79
  60.00  117.35
  70.00  136.91
  80.00  156.47
  90.00  176.02
 100.00  195.58
```

### B.4.3 while-Schleife

Die `while`-Schleife wird anhand der folgenden Problemstellung erläutert: **Bedingte Schleife**

Mit Hilfe des Programms sollen die Zahlen, die der Anwender eingibt, permanent addiert und ausgegeben werden. Solange die Summe der Zahlen kleiner als 50 ist, wird der Vorgang wiederholt. Trifft dies nicht mehr zu, so wird das Programm beendet.

```
summe = 0

while summe < 50:
    print "Bitte eine Zahl eingeben:"
    eingabe = input()
    summe = summe + eingabe
    print "Zwischensumme:", summe

print "Ende"
```

**Listing B.17** Datei b22.py

Zur Erläuterung:

- Zunächst wird die Variable für die Summe der Zahlen auf 0 gesetzt.
- Die `while`-Anweisung leitet die Schleife ein. Die wörtliche Übersetzung der Zeile lautet: »solange die Summe kleiner als 50 ist«. Dies bezieht sich auf die folgenden, eingerückten Anweisungen. Hinter dem Wort `while` steht (wie bei einer `if`-Anweisung) eine Bedingung, die mit Hilfe von Vergleichsoperatoren erstellt wird. Auch hier darf der : (Doppelpunkt) am Ende der Zeile (wie bei `if` und `for`) nicht vergessen werden. **Einrücken**
- Der Anwender wird aufgefordert, eine Zahl einzugeben.
- Die eingegebene Zahl wird in der Variablen `eingabe` gespeichert.

- Sie wird zur bisherigen Summe hinzuaddiert. Die neue Summe errechnet sich also aus der alten Summe plus der eingegebenen Zahl.
- Die neue Summe wird ausgegeben.
- Die Anweisung zur Ausgabe des Textes `Ende` wird erst erreicht, wenn die Summe den Wert 50 erreicht oder überschritten hat.

Eine mögliche Ausgabe des Programms:

```
>>>
Bitte eine Zahl eingeben:
12
Zwischensumme: 12
Bitte eine Zahl eingeben:
6
Zwischensumme: 18
Bitte eine Zahl eingeben:
24
Zwischensumme: 42
Bitte eine Zahl eingeben:
14
Zwischensumme: 56
Ende
```

#### Übung B23

Schreiben Sie ein Programm (Datei `b23.py`), das den Anwender wiederholt dazu auffordert, einen Euro-Betrag einzugeben. Der eingegebene Betrag soll anschließend in DM umgerechnet und ausgegeben werden. Das Programm soll beendet werden, falls der Anwender den Betrag 0 eingibt.

Bei einer `while`-Schleife wird immer angegeben, unter welcher Bedingung wiederholt werden soll und nicht, unter welcher Bedingung beendet werden soll. Daher ist für dieses Programm die umgekehrte Bedingung zu formulieren: »solange die Eingabe ungleich 0 ist«.

## B.5 Fehler und Ausnahmen

**Fehler**

Falls der Anwender nach einer Eingabeaufforderung etwas Falsches eingibt, z. B. keine Zahl, sondern einen Text, wird das Programm mit einer Fehlermeldung beendet. Bisher sollte vereinfacht davon ausgegangen

werden, dass der Anwender richtige Eingaben vornimmt. Die Vermeidung bzw. das Abfangen von Fehlern wird in diesem Abschnitt beschrieben.

### B.5.1 Basisprogramm mit Fehlern

Durch das Abfangen von falschen Eingaben wird die Benutzung eines Programms für den Anwender deutlich komfortabler. Nachfolgend soll ein kleines Programm, in dem eine Zahl eingegeben werden soll, Schritt für Schritt dahingehend verbessert werden:

```
print "Bitte geben Sie eine Zahl ein"
eingabe = input()
print "Sie haben die Zahl", eingabe, "richtig eingegeben"
print "Ende des Programms"
```

**Listing B.18** Datei b24.py (Version 1)

Falls der Anwender eine Zahl (z. B. `45.2`) eingibt, so läuft das Programm fehlerfrei bis zum Ende und erzeugt die folgende Ausgabe:

```
>>>
Bitte geben Sie eine Zahl ein
45.2
Sie haben die Zahl 45.2 richtig eingegeben
Ende des Programms
```

Falls der Anwender dagegen z. B. `45.a2` eingibt, so bricht das Programm vorzeitig ab und erzeugt die folgende Ausgabe:

```
>>>
Bitte geben Sie eine Zahl ein
45.a2
Traceback (most recent call last):
  File "C:\Python22\b24.py", line 4, in ?
    eingabe = input()
  File "<string>", line 1
    45.a2
       ^
SyntaxError: unexpected EOF while parsing
```

Diese Meldungen sind ein Hinweis darauf, an welcher Stelle im Programm der Fehler bemerkt wurde (Datei `b24.py`, Zeile 4). Außerdem wird die Art des Fehlers mitgeteilt (`SyntaxError`).

Falls der Anwender z. B. `ab45` eingibt, so bricht das Programm ebenfalls vorzeitig ab und erzeugt die folgende Ausgabe:

```
>>>
Bitte geben Sie eine Zahl ein
ab45
Traceback (most recent call last):
  File "C:\Python22\b24.py", line 4, in ?
    eingabe = input()
  File "<string>", line 0, in ?
NameError: name 'ab45' is not defined
```

In diesem Fall wurde der Fehler an der gleichen Stelle bemerkt (Datei `b24.py`, Zeile 4). Es handelt sich allerdings um eine andere Fehlerart (`NameError`).

### B.5.2 Programmabbruch vermeiden

Der Fehler soll zunächst einmal abgefangen werden, damit ein Abbruch des Programms vermieden werden kann. Dazu ist es wichtig, die Stelle, an der der Fehler auftrat, zu kennen. An dieser Stelle muss das Programm verbessert werden. In beiden Fehlerfällen handelte es sich um die Zeile, in der der Anwender zur Eingabe aufgefordert wird.

**try-except**

Die Art des Fehlers ist zunächst nicht so wichtig, denn das Programm soll zukünftig an dieser Stelle alle Arten von Fehlern abfangen, die Python von selber erkennen kann. Dies wird in einem ersten Schritt durch folgende Verbesserung erreicht:

```
print "Bitte geben Sie eine Zahl ein"

try:
    eingabe = input()
    print "Sie haben die Zahl", eingabe, \
          "richtig eingegeben"
except:
    print "Falsche Eingabe"

print "Ende des Programms"
```

**Listing B.19** Datei b24.py (Version 2)

Zur Erläuterung:

- Die Anweisung `try` leitet eine Ausnahmebehandlung ein. Ähnlich wie bei der `if`-Anweisung gibt es verschiedene Zweige, die das Programm durchlaufen kann. Es wird versucht (engl.: try) die Anweisungen, die eingerückt hinter `try` stehen, durchzuführen.
- Falls die Eingabe erfolgreich ist, so wird der `except`-Zweig nicht ausgeführt, ähnlich wie bei dem `else`-Zweig der `if`-Anweisung.
- Falls die Eingabe dagegen nicht erfolgreich ist und es sich um einen Fehler handelt, der Python bekannt ist, so wird der Fehler bzw. die Ausnahme (engl.: exception) mit der Anweisung `except` abgefangen. Es werden dann alle eingerückten Anweisungen in diesem Zweig durchgeführt, die Fehlermeldung erscheint.
- Anschließend läuft das Programm ohne Abbruch zu Ende, da der Fehler zwar auftrat, aber abgefangen wurde.
- Nur die »kritische« Zeile wurde in die Ausnahmebehandlung eingebettet. Der Programmierer muss sich also Gedanken darüber machen, welche Stellen seines Programms fehlerträchtig sind. Eine Eingabeaufforderung ist solch eine typische Stelle. Andere Fehlermöglichkeiten sind z. B. die Bearbeitung einer Datei (die eventuell nicht existiert) oder die Ausgabe auf einen Drucker (der eventuell nicht eingeschaltet ist).

- Man hat die Möglichkeit, unterschiedliche Fehlerarten in spezifischen `except`-Zweigen abzufangen. Vorteil: Die Fehlermeldung kann präzise formuliert werden. Nachteil: Es könnte ein Fehler auftreten, zu dem es keinen `except`-Zweig gibt. Auf diese Möglichkeit wird in Kapitel E eingegangen.

Falls der Anwender in dem angegebenen Programm eine richtige Zahl eingibt, so erscheint die bereits bekannte Bestätigung:

```
>>>
Bitte geben Sie eine Zahl ein
45.2
Sie haben die Zahl 45.2 richtig eingegeben
Ende des Programms
```

Falls der Anwender eine falsche Eingabe macht, z. B. `ab45`, so erscheint die nachfolgende Meldung. Das Programm bricht aber nicht ab, sondern läuft bis zum Ende.

```
>>>
Bitte geben Sie eine Zahl ein
45.a2
Falsche Eingabe
Ende des Programms
```

### B.5.3 Fehler abfangen

Fehler abfangen

In einem zweiten Schritt soll dafür gesorgt werden, dass der Anwender nach einer falschen Eingabe erneut eingeben kann. Der gesamte Eingabevorgang einschließlich der Ausnahmebehandlung wird so lange wiederholt, bis er erfolgreich war. Dazu das folgende Programm:

```
fehler = 1

while fehler == 1:
    try:
        print "Bitte geben Sie eine Zahl ein"
        eingabe = input()
        fehler = 0
    except:
        print "Falsche Eingabe, bitte wiederholen"

print "Sie haben die Zahl", eingabe, "richtig eingegeben"
print "Ende des Programms"
```

**Listing B.20** Datei b25.py

Zur Erläuterung:

- Es wird zunächst die Variable `fehler` auf den Wert `1` gesetzt. Diese Variable ist notwendig, um die Eingabe gegebenenfalls wiederholen zu können.
- Mit der `while`-Anweisung wird eine Schleife formuliert, in der der Eingabevorgang einschließlich der Ausnahmebehandlung eingebettet ist. Die Schleife wird wiederholt, solange die Variable `fehler` den Wert `1` hat.
- Falls die Eingabe erfolgreich ist, so wird `fehler` auf `0` gesetzt. Dies führt dazu, dass die Schleife beendet wird und das Programm regulär fortfahren kann.
- Falls die Eingabe nicht erfolgreich ist, so hat `fehler` nach wie vor den Wert `1`. Dies führt dazu, dass der Eingabevorgang wiederholt wird.

- Die Eingabeaufforderung ist hier innerhalb des `try`-Zweiges untergebracht worden, da sie bei Auftreten eines Fehlers wiederum ausgegeben werden sollte.
- Zu beachten ist die doppelte Einrückung, ähnlich wie bei einer geschachtelten Verzweigung.

Nachfolgend eine mögliche Eingabe, zunächst zweimal mit Fehlern, anschließend richtig:

```
>>>
Bitte geben Sie eine Zahl ein
45.a2
Falsche Eingabe, bitte wiederholen
Bitte geben Sie eine Zahl ein
ab45
Falsche Eingabe, bitte wiederholen
Bitte geben Sie eine Zahl ein
45.2
Sie haben die Zahl 45.2 richtig eingegeben
Ende des Programms
```

### Übung B26

Verbessern Sie das Programm zur Eingabe und Umrechnung von beliebigen Euro-Beträgen in französische Franc aus Übung B04. Eingabefehler des Anwenders sollen abgefangen werden. Das Programm soll den Anwender so lange zur Eingabe auffordern, bis sie erfolgreich war (Datei `b26.py`).

### Übung B27

Verbessern Sie das Programm zur Berechnung des monatlich zu zahlenden Steuerbetrags aus Übung B12. Eingabefehler des Anwenders (sowohl bei Gehalt als auch bei Familienstand) sollen abgefangen werden. Das Programm soll den Anwender so lange zur Eingabe auffordern, bis sie erfolgreich war (Datei `b27.py`).

Es geht hier um das Abfangen vollständig falscher Eingaben (Zeichen statt Zahlen). Falls der Anwender beim Familienstand weder eine 1 noch eine 2 eingibt, soll dies noch nicht behandelt werden.

### B.5.4 Fehler erzeugen

**Fehler erzeugen**

»Wieso sollte man Fehler erzeugen?«, wird man sich angesichts dieser Überschrift fragen. Es gibt, besonders im Zusammenhang mit der Eingabe von Daten durch einen Anwender, sinnvolle Gründe dafür.

Im folgenden Beispiel wird der Anwender wie in Programm `b25.py` dazu aufgefordert, Zahlen einzugeben. Diese Zahlen sollen allerdings weder größer als 1000 noch kleiner als 0 sein. Diese Einschränkung kann mit Hilfe der Anweisung `raise` bearbeitet werden, wie im nachfolgenden Programm dargestellt:

```
fehler = 1
while fehler == 1:
    try:
        print "Bitte geben Sie eine Zahl", \
              "zwischen 0 und 1000 ein"
        eingabe = input()

        # Eingabe zu gross oder zu klein
        if eingabe > 1000 or eingabe < 0:
            raise

        fehler = 0
    except:
        print "Falsche Eingabe, bitte wiederholen"

print "Sie haben die Zahl", eingabe, "richtig eingegeben"
print "Ende des Programms"
```

**Listing B.21** Datei b28.py

Falls die eingegebene Zahl größer als 1000 oder kleiner als 0 ist, so wird die Anweisung `raise` ausgeführt. Dadurch wird ein Fehler erzeugt, als ob der Anwender Zeichen statt Zahlen eingegeben hätte. Das Programm verzweigt unmittelbar zur Anweisung `except` und führt die dort angegebenen Anweisungen aus.

In diesem Fall handelt es sich zwar nur um einen logischen Eingabefehler, aber er wird genauso behandelt. Der Anwender wird somit veranlasst, Zahlen nur innerhalb des gewünschten Bereiches einzugeben.

Nachfolgend eine mögliche Eingabe, zunächst zweimal mit Fehlern, anschließend richtig:

```
>>>
Bitte geben Sie eine Zahl zwischen 0 und 1000 ein
1100
Falsche Eingabe, bitte wiederholen
Bitte geben Sie eine Zahl zwischen 0 und 1000 ein
-5
Falsche Eingabe, bitte wiederholen
Bitte geben Sie eine Zahl zwischen 0 und 1000 ein
23
Sie haben die Zahl 23 richtig eingegeben
Ende des Programms
```

Mit Hilfe der Anweisungen `try`, `raise` und `except` lassen sich also auch logische Eingabefehler des Anwenders abfangen und behandeln. Nachteil der vorgeführten Methode: Alle Fehler werden gleich behandelt, die Informationen für den Anwender im Fehlerfall sind noch nicht sehr genau. Dies soll in einem späteren Abschnitt verbessert werden.

### Übung B29

Verbessern Sie das Programm zur Berechnung des monatlich zu zahlenden Steuerbetrags aus Übung B12 bzw. B27 weiter. Zusätzlich zum Abfangen der Eingabefehler des Anwenders soll darauf geachtet werden:

- dass als Gehälter nur positive Zahlen eingegeben werden
- dass als Familienstand nur eine 1 oder eine 2 eingegeben wird

Das Programm soll den Anwender so lange zur Eingabe auffordern, bis sie erfolgreich war (Datei `b29.py`).

## B.6 Funktionen und Module

**Modularisierung**

Theoretisch könnte ein Programmierer auf die Modularisierung, d.h. die Zerlegung seines Programms in selbst geschriebene Funktionen, verzichten. Besonders bei der Entwicklung von größeren Programmen bieten Funktionen allerdings unübersehbare Vorteile:

- Umfangreiche Programme können in übersichtliche Teile zerlegt werden.
- Pflege und Wartung von Programmen wird erleichtert.
- Der eigene Programmcode wird von einem anderen Programmierer oder vom Programmierer selbst (zu einem späteren Zeitpunkt) leichter verstanden.

- Programmteile, die mehrmals benötigt werden, müssen nur einmal definiert werden.
- Nützliche Programmteile können in mehreren Programmen verwendet werden.

**Vordefinierte Funktionen**

Neben den selbst geschriebenen Funktionen gibt es bei Python, wie bei jeder anderen Programmiersprache auch, eine große Menge an nützlichen, vordefinierten Funktionen, die dem Entwickler bereits viel Arbeit abnehmen können. Diese Funktionen sind entweder fest eingebaut oder über die Einbindung spezieller Module verfügbar.

Eine der fest eingebauten Funktionen wurde bereits eingesetzt: `input()`

- Jede Funktion hat eine spezielle Aufgabe. Im Fall der Funktion `input()` ist dies: das Programm anzuhalten und eine Eingabe entgegenzunehmen.
- Wie viele, aber nicht alle Funktionen, hat `input()` einen so genannten Rückgabewert, d.h., sie liefert ein Ergebnis an die Stelle des Programms zurück, von der sie aufgerufen wurde: den eingegebenen Zahlenwert.

Nachfolgend wird die Entwicklung selbst geschriebener Funktionen erläutert.

### B.6.1 Einfache Funktionen

Einfache Funktionen führen bei ihrem Aufruf immer genau das Gleiche aus. Im folgenden Beispiel führt der Aufruf der Funktion `stern()` jedes Mal dazu, dass eine Reihe von 20 Sternen als optische Trennlinie auf dem Bildschirm ausgegeben wird:

```
# Definition der Funktion
```

**Definition**

```
def stern():
    for i in range(1,21,1):
        print "*",
    print

# Hauptprogramm
x = 12
y = 5
stern()                          # 1. Aufruf
print "x = %i, y = %i" % (x,y)
stern()                          # 2. Aufruf
```

```
print "x + y =", x + y
stern()                         # 3. Aufruf
print "x - y =", x - y
stern()                         # 4. Aufruf
print "x * y =", x * y
stern()                         # 5. Aufruf
```

**Listing B.22** Datei b30.py

Zur Erläuterung:

**Funktionsdefinition**

Im oberen Teil des Programms wird die Funktion `stern()` definiert:

- Nach der Anweisung `def` folgt der Name der Funktion (hier `stern`), anschließend runde Klammern und der bereits bekannte Doppelpunkt. Innerhalb der Klammern könnten Werte an die Funktion übergeben werden (dazu später).
- Die nachfolgenden, eingerückten Anweisungen werden jedes Mal durchgeführt, wenn die Funktion aufgerufen, d. h. benutzt wird.
- Es wird eine `for`-Schleife 20-mal durchgeführt. Innerhalb der `for`-Schleife wird jedes Mal ein Stern auf den Bildschirm geschrieben. Das anschließende Komma sorgt nur dafür, dass nach der Ausgabe nicht in die nächste Zeile gewechselt wird.

- Die einzelne `print`-Anweisung am Ende der Funktionsdefinition sorgt für einen Zeilenwechsel nach den 20 Sternen.

Eine Funktion wird zunächst nur definiert und nicht durchgeführt. Sie steht sozusagen zum späteren Gebrauch bereit.

Im unteren Teil beginnt das eigentliche Programm:

- Es werden verschiedene Rechenoperationen mit zwei Variablen durchgeführt.
- Jede Ausgabezeile wird mit Hilfe der Funktion `stern()` in eine Reihe von Sternen eingebettet.

**Funktionsaufruf**

- Insgesamt findet der Aufruf der Funktion `stern()` 5-mal statt.
- Das Programm fährt nach Bearbeitung der Funktion `stern()` jedes Mal mit der Anweisung fort, die dem Aufruf der Funktion folgt.

Eine Funktion wird aufgerufen, indem ihr Name, gefolgt von den runden Klammern, notiert wird. Falls Informationen an die Funktion geliefert werden sollen, so werden diese innerhalb der runden Klammern angegeben (dazu später).

Der Name einer Funktion kann frei gewählt werden. Es gelten die gleichen Regeln wie bei den Namen von Variablen:

- Er kann aus den Buchstaben a bis z, A bis Z, Ziffern oder dem Zeichen _ (=Unterstrich) bestehen.
- Er darf nicht mit einer Ziffer beginnen.
- Er darf keinem reservierten Wort der Programmiersprache Python entsprechen (eine Liste der reservierten Worte findet sich im Anhang).

Das Ergebnis des Programms:

```
>>>
* * * * * * * * * * * * * * * * * * * * *
x = 12, y = 5
* * * * * * * * * * * * * * * * * * * * *
x + y = 17
* * * * * * * * * * * * * * * * * * * * *
x - y = 7
* * * * * * * * * * * * * * * * * * * * *
x * y = 60
* * * * * * * * * * * * * * * * * * * * *
```

### Übung B31

Erstellen Sie ein Programm (Datei `b31.py`), mit dessen Hilfe eine beliebige eingegebene Zahl quadriert wird (zur Vereinfachung ohne das Abfangen von Fehleingaben). Oberhalb und unterhalb des Programms soll zur optischen Verdeutlichung der Name des Programms erscheinen, wie in der nachfolgenden Ausgabe dargestellt. Schreiben Sie zu diesem Zweck eine Funktion `titel()`, die insgesamt zweimal aufgerufen werden soll.

```
>>>
*****************************
*      Programm Quadrat     *
*****************************
Bitte geben Sie eine Zahl ein:
3.6
Das Quadrat von 3.6 ist 12.96
*****************************
*      Programm Quadrat     *
*****************************
```

### B.6.2 Funktionen mit einem Parameter

**Parameter**

Bei einem Aufruf können auch Informationen an Funktionen übermittelt werden, so genannte Parameter. Dies führt im Allgemeinen dazu, dass diese Informationen innerhalb der Funktion ausgewertet werden und bei jedem Aufruf zu unterschiedlichen Ergebnissen führen.

Im folgenden Programm wird eine Funktion definiert, an die ein Betrag in Euro übermittelt wird. Innerhalb der Funktion wird dieser Betrag in DM umgerechnet und ausgegeben. Die Funktion wird mehrmals aufgerufen, jedes Mal mit einem anderen Euro-Betrag. Zur Vereinfachung wurde auf das Abfangen von Eingabefehlern verzichtet.

```
# Definition der Funktion
def dmaus(eurobetrag):
    ed = 1.95583
    dmbetrag = ed * eurobetrag
    print "%.2f Euro ergeben %.2f DM" % \
        (eurobetrag, dmbetrag)

# Hauptprogramm
print "Umrechnung von Euro in DM:"
dmaus(24)           # 1. Aufruf
dmaus(12.5)         # 2. Aufruf
dmaus(3*5.5)        # 3. Aufruf

print "Bitte einen Betrag in Euro:"
eingabe = input()
dmaus(eingabe)      # 4. Aufruf
```

**Listing B.23** Datei b32.py

Zur Erläuterung:

Die Definition der Funktion `dmaus()` beinhaltet in den Klammern eine Variable. Dies bedeutet, dass beim Aufruf der Funktion ein Wert übermittelt und dieser Variablen zugewiesen wird. Im vorliegenden Programm sind dies:

1. eine ganze Zahl, `eurobetrag` wird 24 zugewiesen
2. eine Zahl mit Nachkommastellen, `eurobetrag` wird 12.5 zugewiesen
3. das Ergebnis einer Berechnung, `eurobetrag` wird das Ergebnis von $3 \times 5.5$ zugewiesen
4. eine Variable, `eurobetrag` wird der Wert der Variablen `eingabe` zugewiesen

Die Funktion erwartet genau einen Wert, da sie nur sinnvoll arbeiten kann, wenn ihr ein Wert übermittelt wird. Sie darf also nicht ohne einen Wert oder mit mehr als einem Wert aufgerufen werden.

Eine mögliche Ausgabe des Programms:

```
>>>
Umrechnung von Euro in DM:
24.00 Euro ergeben 46.94 DM
12.50 Euro ergeben 24.45 DM
16.50 Euro ergeben 32.27 DM
Bitte einen Betrag in Euro:
9.95
9.95 Euro ergeben 19.46 DM
```

### Übung B33

Es soll für verschiedene Bruttogehälter der Steuerbetrag berechnet werden (Datei `b33.py`). Falls das Gehalt über 2.500 Euro liegt, so sind 22% Steuer zu zahlen, ansonsten 18% Steuer (wie Übung B08). Die Berechnung und die Ausgabe des Steuerbetrags sollen innerhalb einer Funktion mit dem Namen `steuer()` stattfinden. Die Funktion soll für die folgenden Gehälter aufgerufen werden: 1.800 Euro, 2.200 Euro, 2.500 Euro, 2.900 Euro.

## B.6.3 Funktionen mit mehreren Parametern

**Mehrere Parameter**

Eine Funktion kann noch vielseitiger werden, falls man ihr mehrere Parameter übermittelt. Dabei ist auf die übereinstimmende Anzahl und Reihenfolge der Parameter zu achten.

Im folgenden Beispiel wird eine Funktion `mw()` definiert. Diese Funktion erwartet drei Zahlen als Parameter und berechnet den Mittelwert dieser drei Zahlen. Falls mehrere Parameter verwendet werden, so werden diese sowohl bei Definition als auch bei Aufruf durch Kommata voneinander getrennt.

```
# Definition der Funktion
def mw(z1, z2, z3):
    summe = z1 + z2 + z3
    mittelwert = summe / 3.0
    print "Der Mittelwert von",
    print z1, ",", z2, "und", z3, "ist", mittelwert
```

```
# Hauptprogramm
mw(3,5,10)          # 1. Aufruf
mw(-2,55,12)        # 2. Aufruf
mw(2.4,2.6,5)       # 3. Aufruf
```

**Listing B.24** Datei b34.py

Zur Erläuterung:

Es werden genau drei Parameter erwartet, bei jedem der drei Aufrufe werden auch genau drei Werte übermittelt, z. B. wird beim 1. Aufruf:

- die Zahl 3 der Variablen `z1` zugewiesen
- die Zahl 5 der Variablen `z2` zugewiesen
- die Zahl 10 der Variablen `z3` zugewiesen

Die Ausgabe des Programms:

```
>>>
Der Mittelwert von 3 , 5 und 10 ist 6.0
Der Mittelwert von -2 , 55 und 12 ist 21.6666666667
Der Mittelwert von 2.4 , 2.6 und 5 ist 3.33333333333
```

## Übung B35

Das Programm zur Berechnung des monatlich zu zahlenden Steuerbetrags soll wiederum verändert werden. Die Steuertabelle sieht wie folgt aus:

| Gehalt | Familienstand | Steuersatz |
|---|---|---|
| > 4.000Euro | ledig | 26% |
| > 4.000 Euro | verheiratet | 22% |
| <= 4.000 Euro | ledig | 22% |
| <= 4.000 Euro | verheiratet | 18% |

**Tabelle B.6** Steuertabelle für diese Übung

Es soll eine Funktion `steuer()` geschrieben werden, die zwei Parameter erwartet: Gehalt und Familienstand. Dabei soll z. B. der Familienstand »ledig« mit 1 und der Familienstand »verheiratet« mit 2 angegeben werden. Die Funktion soll für die folgenden Kombinationen aufgerufen werden:

- Gehalt: 2.800 Euro, Familienstand ledig
- Gehalt: 2.900 Euro, Familienstand verheiratet
- Gehalt: 4.000 Euro, Familienstand ledig
- Gehalt: 4.500 Euro, Familienstand verheiratet
- Gehalt: 5.200 Euro, Familienstand ledig

### B.6.4 Funktionen mit Rückgabewert

Funktionen werden häufig nur zur Berechnung, aber nicht zur Ausgabe der berechneten Ergebnisse eingesetzt. Die Ausgabe bzw. weitere Verwertung der Rechenergebnisse soll dem Hauptprogramm bzw. dem aufrufenden Programm überlassen werden. Zu diesem Zweck können Funktionen ihre Ergebnisse als so genannte Rückgabewerte zurückliefern.

**Rückgabewert**

Im Unterschied zu vielen anderen Programmiersprachen können Funktionen in Python mehr als einen Rückgabewert liefern. In diesem Abschnitt sollen allerdings zunächst nur Funktionen betrachtet werden, die genau einen Rückgabewert zur Verfügung stellen.

Im folgenden Beispiel wird eine Funktion, die einen Rückgabewert liefert, auf zwei verschiedene Arten im Hauptprogramm eingesetzt und aufgerufen.

```
# Definition der Funktion
def dm(eurobetrag):
    ed = 1.95583
    dm = ed * eurobetrag
    return dm

# 1. Aufruf
euro = 24
erg = dm(euro)
print "%.2f Euro ergeben %.2f DM" % (24, erg)

# 2. Aufruf
print "12.5 Euro ergeben", dm(12.5), "DM"
```

**Listing B.25** Datei b36.py

Zur Erläuterung:

- Bei der Definition der Funktion wird der DM-Betrag zunächst berechnet. Anschließend wird er mit Hilfe der Anweisung `return` an die auf-

rufende Stelle zurückgeliefert. Die Anweisung `return` beendet außerdem unmittelbar den Ablauf der Funktion.

- Beim ersten Aufruf wird der Rückgabewert in der Variablen `erg` zwischengespeichert. Er kann im weiteren Verlauf des Programms an beliebiger Stelle verwendet werden. Im vorliegenden Fall wird er kommentiert und formatiert ausgegeben.
- Beim zweiten Aufruf wird der Rückgabewert unmittelbar ausgegeben.

Die Ausgabe des Programms:

```
>>>
24.00 Euro ergeben 46.94 DM
12.5 Euro ergeben 24.447875 DM
```

### Übung B37

Das Programm aus Übung B35 soll umgeschrieben werden. Innerhalb der Funktion `steuer()` soll der Steuerbetrag nur berechnet und an das Hauptprogramm zurückgeliefert werden. Die Ausgabe des ermittelten Wertes soll im Hauptprogramm stattfinden.

## B.6.5 Module

In den bisher genannten Beispielen wurden Funktion und eigentliches Hauptprogramm in der gleichen Datei definiert. Falls es sich um spezifische Funktionen handelt, die für ein bestimmtes Programm zugeschnitten wurden, ist dies auch sinnvoll.

**Externe Module**

Nach einiger Zeit wird man aber feststellen, dass einige nützliche Funktionen immer wieder und von verschiedenen Programmen aus benötigt werden. Diese Funktionen sollten in eigenen Modulen gespeichert werden. Die Erstellung und Nutzung von Modulen ist in Python sehr einfach und soll am folgenden Beispiel erläutert werden.

Die gewünschte Funktion wird einfach in einer eigenen Datei gespeichert. Der Name der Datei ist gleichzeitig der Name des Moduls. An der Funktion selber ändert sich nichts:

```
# Definition der Funktion
def dm(eurobetrag):
    ed = 1.95583
    dm = ed * eurobetrag
    return dm
```

**Listing B.26** Datei b38.py

Anschließend kann die Funktion in jedem Programm genutzt werden. Voraussetzung: Sie wird aus dem betreffenden Modul importiert.

```
# Import aus dem Modul
from b38 import dm

# 1. Aufruf
euro = 24
erg = dm(euro)
print "%.2f Euro ergeben %.2f DM" % (24, erg)

# 2. Aufruf
print "12.5 Euro ergeben", dm(12.5), "DM"
```

**Listing B.27** Datei b39.py (Version 1)

Zur Erläuterung:

Die erste Anweisung (`from b38 import dm`) kann wie folgt gelesen werden: Importiere die Funktion `dm()` aus dem Modul `b38` (also aus der Datei `b38.py`).

**Import**

Falls im Modul `b38` mehrere Funktionen definiert wurden, die innerhalb eines Programms eingesetzt werden sollen, so kann man alternativ eine der beiden folgenden Anweisungen einsetzen:

- `from b38 import *`
- `import b38`

Dadurch werden jeweils alle Funktionen aus dem Modul importiert. Die zweite Version setzt allerdings voraus, dass bei den Funktionsaufrufen im importierenden Programm genau angegeben wird, aus welchem Modul sie stammen, wie in der nachfolgenden Version des Programms `b39`.

```
# Import aus dem Modul
import b38

# 1. Aufruf
euro = 24
erg = b38.dm(euro)
print "%.2f Euro ergeben %.2f DM" % (24, erg)
```

```
# 2. Aufruf
print "12.5 Euro ergeben", b38.dm(12.5), "DM"
```

**Listing B.28** Datei b39.py (Version 2)

Zur Erläuterung:

Die Funktion wird nicht nur mit ihrem Namen aufgerufen, sondern in der Schreibweise `Modulname.Funktionsname`. Dies hat auch den Vorteil der Eindeutigkeit, falls ein Funktionsname in mehreren Modulen verwendet wurde.

### Übung B40/B41

Das Programm aus Übung B37 soll umgeschrieben werden. Die Funktion `steuer()` soll in das Modul `b40` ausgelagert werden. Das Hauptprogramm in Datei b41.py soll diese Funktion aus dem Modul importieren und nutzen.

## B.7 Programmierprojekt Rechnungserstellung

In den bisherigen Abschnitten wurden alle wichtigen Elemente der klassischen Programmierung vorgestellt. In diesem Abschnitt wird ein Programm erläutert, bei dem diese Elemente im Gesamtzusammenhang eingesetzt werden.

### B.7.1 Aufgabenstellung

Es handelt sich um eine Aufgabe aus dem Bereich Warenwirtschaft. Es soll eine Rechnung, bestehend aus einzelnen Rechnungsposten erstellt werden. Dem Programmierer ist die nachfolgende Preisliste bekannt:

| Artikelnummer | Netto-Einzelpreis in Euro |
|---|---|
| 17 | 2,20 |
| 22 | 24,50 |
| 38 | 15,00 |
| 47 | 9,95 |
| 125 | 12,95 |

**Tabelle B.7** Preisliste für Programmierprojekt

Der Anwender, z.B. eine Verkaufskraft an der Kasse, gibt die einzelnen Posten der Rechnung ein. Dies sind die Artikelnummer, gefolgt von der Anzahl der gekauften Artikel mit dieser Artikelnummer. Dabei sollte darauf geachtet werden, dass nur eine der vorhandenen Artikelnummern eingegeben werden kann. Die Eingabe soll beendet werden, falls der Anwender eine 0 als Artikelnummer eingibt. Allgemein sollen Eingabefehler bzw. nicht sinnvolle Eingaben abgefangen werden und zu einer Wiederholung der jeweiligen Eingabe führen.

Über Artikelnummer, Menge und Einzelpreis soll im Programm der Wert eines einzelnen Rechnungspostens ermittelt werden. Die Posten werden addiert und führen zum Netto-Gesamtwert der Rechnung. Daraus wird die Umsatzsteuer und der Bruttowert errechnet und ausgegeben. Dieser wird zusätzlich noch (zur Information) in DM ausgegeben.

Das Programm soll mit Hilfe von Funktionen in übersichtliche Teile zerlegt werden. Die verschiedenen Eingabeüberprüfungen auf sinnvolle Inhalte sollen in ein Modul ausgelagert werden.

### B.7.2 Ablaufplan des Programms

Ein erster, grober Ablaufplan des Programms sieht wie folgt aus:

- Eingabe einer ersten Artikelnummer
- Schleife: Solange die eingegebene Artikelnummer ungleich 0 ist, sollen folgende Schritte durchgeführt werden:
  - Zur eingegebenen Artikelnummer wird der Einzelpreis ermittelt
  - Eingabe der gekauften Menge
  - Es wird der Wert eines einzelnen Rechnungspostens ermittelt
  - Dieser Wert wird zum jeweiligen Netto-Gesamtwert hinzuaddiert
  - Eingabe der nächsten Artikelnummer
- Berechnung der Gesamtrechnungssumme brutto
- Ausgabe aller Gesamtrechnungssummen

### B.7.3 Ausgelagerte Funktionen

In das Modul `b43` werden insgesamt drei Funktionen ausgelagert, die auch von anderen Programmen genutzt werden können:

- Eingabe der Artikelnummer
- Eingabe der Menge
- Ermittlung des Preises

### Eingabe der Artikelnummer

Die Funktion `eingabe_artnr()` hat keinen Parameter, aber einen Rückgabewert. Die Eingabe der Artikelnummer durch den Anwender ist in eine Ausnahmebehandlung eingebettet. Dadurch wird dafür gesorgt, dass auf jeden Fall eine reguläre Artikelnummer zurückgeliefert wird. Die Eingabe von Zeichen statt Zahlen oder die Eingabe einer ungültigen Artikelnummer führen zur Wiederholung der Eingabe.

```
# Eingabe der Artikelnummer
def eingabe_artnr():
    fehler = 1
    while fehler == 1:
        try:
            print "Artikelnummer:"
            ein = input()
            if ein != 0 and ein != 17 and ein != 22 \
               and ein != 38 and ein != 47 and ein != 125:
                raise
            fehler = 0
        except:
            print "Es ist nur erlaubt: 0 17 22 38 47 125"
            print "Bitte wiederholen"
    return ein
```

**Listing B.29** Datei b43.py, Eingabe der Artikelnummer

Das Zeichen \ am Ende der `if`-Anweisung weist darauf hin, dass diese Anweisung in der nächsten Zeile fortgesetzt wird.

### Eingabe der Menge

Die Funktion `eingabe_menge()` hat einen Parameter und einen Rückgabewert. Die Artikelnummer wird an die Funktion übermittelt. Dadurch kann der Anwender darüber informiert werden, zu welchem Artikel er eine Menge eingeben soll. Die Eingabe der Menge ist ebenfalls in eine Ausnahmebehandlung eingebettet. Dadurch wird dafür gesorgt, dass auf jeden Fall eine sinnvolle Menge zurückgeliefert wird. Die Eingabe von Zeichen statt Zahlen oder die Eingabe einer nicht sinnvollen Mengenangabe führen zur Wiederholung der Eingabe.

```
# Eingabe der Menge
def eingabe_menge(nr):
    fehler = 1
```

```
    while fehler == 1:
        try:
            print "Menge des Artikels", nr, ":"
            ein = input()
            if ein < 0:
                raise
            fehler = 0
        except:
            print "Es ist nur eine Menge >= 0 sinnvoll"
            print "Bitte wiederholen"
    return ein
```

**Listing B.30** Datei b43.py, Eingabe der Menge

### Ermittlung des Preises

Die Funktion `ermitteln_epreis()` hat ebenfalls einen Parameter und einen Rückgabewert. Auch hier wird die Artikelnummer an die Funktion übermittelt. Mit Hilfe der Artikelnummer wird in einer mehrfachen Verzweigung der zugehörige Einzelpreis ermittelt.

```
# Preisermittlung
def ermitteln_epreis(nr):
    if nr == 17:
        epreis = 2.2
    elif nr == 22:
        epreis = 24.5
    elif nr == 38:
        epreis = 15
    elif nr == 47:
        epreis = 9.95
    else:
        epreis = 12.95
    return epreis
```

**Listing B.31** Datei b43.py, Preisermittlung

### Hauptprogramm

**Hauptprogramm**

Der Ablauf des Hauptprogramms hält sich im Wesentlichen an den bereits bekannten groben Ablaufplan. Die Programmteile, die eine größere Schachtelungstiefe haben, die also mehrfache Verzweigungen und/oder Schleifen beinhalten, wurden ausgelagert. Sowohl Hauptprogramm als auch Funktionen sind dadurch übersichtlicher und besser zu warten.

Einzelne Aspekte, die diesem Programm später hinzugefügt werden können, betreffen eventuell nur eine einzelne Funktion. Der Rest des Programms kann unverändert bleiben.

```
# Module importieren
from b43 import *

# Gesamtrechnungssumme auf 0 setzen
nettosumme = 0

# 1. Eingabe einer Artikelnummer
artnr = eingabe_artnr()

# Schleife für alle Rechnungsposten
while artnr != 0:

    # Preisermittlung
    epreis = ermitteln_epreis(artnr)

    # Eingabe der Menge
    menge = eingabe_menge(artnr)

    # Gesamtpreis des Rechnungspostens
    posten = menge * epreis

    # Summierung für Gesamtrechnungssumme
    nettosumme = nettosumme + posten

    # nächste Eingabe einer Artikelnummer
    artnr = eingabe_artnr()

# Berechnung der Rechnungssumme brutto und in DM
ust = nettosumme * 0.16
bruttosumme = nettosumme + ust
bruttosumme_dm = bruttosumme * 1.95583

# Ausgabe aller Rechnungssummen
print "Netto       %7.2f Euro" % (nettosumme)
print "Ust         %7.2f Euro" % (ust)
print "Brutto      %7.2f Euro" % (bruttosumme)
print "entspricht  %7.2f DM " % (bruttosumme_dm)
```

**Listing B.32** Datei b42.py