

Programming Python

Teil I: Der Pythonkern

Heiko Schröder

Version 0.5

Vorwort zur Version 0.4

In dieser Version wurden im Vergleich zur Version 0.3 vor allem in den Kapiteln über Funktionen und Klassen etliche Ergänzungen vorgenommen. Von einer geplanten Referenz wurde schliesslich doch abgesehen, da sie der Intention dieser Schrift ein wenig widerspricht. Es geht darum, die Sprache so weit zu erlernen, dass eine Referenz *verstanden* werden kann. Daher ist allen Programmierern dringend die Referenz »Python-Kurz und Gut« aus dem O'Reilly-Verlag von Mark LUTZ zu empfehlen, deren einziger kleiner Nachteil darin besteht, dass sie kein Register besitzt, was aber vielleicht bei einem Umfang von 68 Seiten verschmerzt werden kann.

Mit der Version 0.4 wird der Text dieser Schrift quasi »eingefroren« und in den folgenden Versionen nur noch auf Fehler, Stil und Abbildungsqualität überprüft. Der Umfang wird sich jetzt also nicht mehr wesentlich verändern.

Vorwort zur Version 0.3

Dieses kleine Buch ist ein Versuch, ein kleines Lehrbuch für den Informatikunterricht zu ersetzen. Vorgestellt wird die Struktur der Programmiersprache Python, die für den Informatikunterricht geradezu ideal erscheint. Das Buch ist weder eine Anleitung für professionelle Programmierer, noch eine Referenz. Es geht darum, die Struktur der Sprache kennenzulernen, *damit* es leichter ist, eine Referenz zu befragen. Als Ergänzung zu diesem Buch ist ein zweiter Teil mit dem Titel »Programming Python - Tkinter« geplant, sowie eine »Kurzreferenz« als dritten und letzten Teil.

Diese Version des Buches ist noch nicht auf dem endgültigen Stand. Sie wird auf ihre Tauglichkeit hin von den Mädchen des 11. Jahrgangs Informatik am Emil-von-Behring-Gymnasium in Grosshansdorf getestet. Die offizielle Version wird natürlich 1.0 sein.

Wenn Sie bereits Erfahrungen mit anderen Programmiersprachen haben, können Sie sich im Kapitel 1 einen ersten Überblick verschaffen. Der Text ist in diesem Abschnitt allerdings sehr knapp gehalten. Vieles wird durch Abbildungen ergänzt. Die Abschnitte, die mit einem Sternchen * versehen sind, brauchen Sie beim ersten Lesen noch nicht zu berücksichtigen.

Die Kapitel 2 bis 6 stellen den Hauptteil des Buches dar. Hier finden Sie zahllose, aber kurze Beispiele, die Sie unbedingt alle durchführen sollten! Wenn Sie nicht wissen, wie der Python-Interpreter aufzurufen ist, schauen Sie bitte in den Anhängen B.1 und B.2 nach, wie dies geht. In Python kann jeder Code sofort überprüft und ausgeführt werden. Aus diesem Grund wird die Sprache für das Entwickeln grosser Projekte herangezogen, da bei Compilersprachen wie C++ und Java ein Testen einzelner Module sehr viel schwieriger ist. Wenn Sie genau wissen wollen, wie Python im Gegensatz zu C++ arbeitet, ist für Sie der Abschnitt A wichtig, der Ihnen aber keine Programmierbeispiele zeigt.

Wenn Sie wissen was objektorientierte Programmierung ist (OOP), so werden Sie feststellen, dass sich mit Python sich die Konzepte des objektorientierten Programmierens so leicht erlernen lassen wie mit keiner anderen Sprache. Wer Python beherrscht, kann ohne weiteres auf Java und C++ oder Object Pascal bei Bedarf umschwenken – und wird dann wahrscheinlich feststellen, wie unbeholfen und aufgesetzt die objektorientierte Programmierung bei C++ und erst recht bei Object Pascal ist. Wenn Sie mehr über die Vorzüge der Sprache erfahren wollen, oder wenn Sie sich gefragt haben, was der Name *Python* im Zusammenhang mit dieser Sprache überhaupt soll, schauen Sie bitte in das Kapitel B.3.

Wie gesagt: es geht darum, sich mit diesem Buch den »Pythonkern« zu erarbeiten, zu verstehen, wie die Sprache funktioniert. Sie werden feststellen, dass es diese Sprache geschafft hat, trotz exakter Programmierung einen geringstmöglichen Ballast an Syntax zu erfordern. Viel Freude an Python wünscht dem Leser der Autor dieser Schrift

Heiko Schröder

Inhaltsverzeichnis

1	Objekte und Namen	1
1.1	Datenobjekte und Funktionen	2
1.2	Klassen	2
1.3	Namen	3
1.3.1	Namenszuweisung für Funktionen*	5
1.3.2	Vereinbarung von Klassen*	6
1.4	Vergleich mit anderen Programmiersprachen*	9
1.5	Module	11
1.6	Zusammenfassung	15
2	Datenobjekte	16
2.1	Zahlen	16
2.2	Strings	18
2.3	Listen	20
2.4	Tupel	22
2.5	Dictionaries	23
2.6	Dateien	26
3	Anweisungen	28
3.1	Vergleiche	29
3.2	Enthaltensein	29

<i>INHALTSVERZEICHNIS</i>	<i>VI</i>
3.3 Schleifen	30
3.3.1 Die endliche Schleife: For-Schleife	30
3.3.2 Die bedingte Schleife: While-Schleife	31
3.4 Ergänzungen zu Schleifen	32
3.5 Die Verzweigung	32
3.6 Namen in Anweisungsblöcken	33
4 Funktionen	34
4.1 Wertevorbelegung	36
4.2 Beliebige Anzahl von Argumenten	37
4.3 Kommentare	38
4.4 Einlesen von der Tastatur	39
4.5 Ausgabe	39
4.6 lambda-Ausdrücke	40
5 Klassen	42
5.1 Attribute, Methoden, Instanzen, Klassen	45
5.2 Vererbung	46
5.3 Noch einmal: Dateien	47
5.4 Was ist OOP?	48
6 Module	52
7 Ausnahmen	54
A Wie werden Programme erstellt?	55
A.1 Über Bits, Bytes und Assembler	56
A.1.1 Zahlencode	56
A.1.2 Zeichencode	57
A.1.3 Befehlscode	57

A.1.4	Zusammenfassung	58
A.1.5	Programmieren auf unterster Ebene	58
A.1.6	Was ist ein Assembler?	59
A.2	Die echten Programmiersprachen	60
A.2.1	Wie werden Übersetzer entwickelt?	61
A.2.2	Was ist ein Compiler? Compilersprachen	64
A.2.3	Was ist ein Interpreter? Interpretersprachen	65
A.2.4	Wie arbeitet Python	65
B	Die ersten Schritte	67
B.1	Aufrufen des PYTHON-Interpreters	67
B.2	Aufrufen von IDLE	68
B.2.1	Wo befindet sich IDLE?	69
B.3	Woher kommt der Name Python?	69
B.4	Hilfe bei Problemen	73

Abbildungsverzeichnis

1.1	Datenobjekte	4
1.2	Funktionsobjekte	5
1.3	Klassen	8
1.4	Hartes und weiches Linken	10
1.5	Module sind Namenscontainer	13
1.6	Das Mainmodul der Python-Umgebung	14
2.1	Dictionaries im Vergleich mit Strings, Listen und Tupel	25
5.1	Prozedurales Programmieren	49
5.2	Objektorientiertes Programmieren	50
5.3	Mehrfachvererbung zwischen Klassen	50
A.1	Hierarchie von Programmiersprachen	62

Kapitel 1

Objekte und Namen

Dieses Kapitel gibt Ihnen einen ersten Eindruck von der Programmiersprache. Es ist durchaus möglich, dass Ihnen das Ganze zu schnell geht. Aber *alle* angesprochenen Dinge werden ab Kapitel 2 detaillierter behandelt. An dieser Stelle soll der Aufbau der Sprache, ihre logische Struktur, deutlich werden. für wen ist daher dieser Abschnitt gedacht?

1. Für diejenigen unter Ihnen, die einen groben Überblick über die Sprache haben wollen, und sich nicht davon erschüttern lassen, wenn sie nicht gleich alles verstehen.
2. Für diejenigen unter Ihnen, die schon Erfahrungen mit anderen Programmiersprachen haben, und es nicht erwarten können, gleich loszulegen.
3. Für diejenigen, die schon die daran anschliessenden Abschnitte durchgearbeitet haben, und sich einen umfassenden Überblick verschaffen wollen.
4. Für diejenigen, die nur mal schnell nachschlagen wollen, nach dem Motto »Wie war das noch...?«

Prüfen Sie die Beispiele unbedingt nach! Sie werden so erheblich schneller lernen! Rufen Sie den Python-Interpreter auf, am besten sogar IDLE. Wie das geht, sehen Sie im Anhang. Verlassen Sie möglichst IDLE nicht innerhalb eines Kapitels.

1.1 Datenobjekte und Funktionen

In der Programmiersprache Python haben wir es, genau wie im gewöhnlichen Alltag mit Dingen zu tun,

- die irgendetwas enthalten oder
- die irgendeine Aufgabe ausführen

können. Solche Dinge heissen in der Informatik *Objekte*. Ein Objekt enthält im einfachsten Fall einen *Datensatz*; in der gleichen Art und Weise wie ein Glas einen Apfelsaft als Inhalt haben kann. In diesem Falle handelt es sich dabei um ein reines *Datenobjekt*.

Um den Apfelsaft zu trinken, bedarf es einer gewissen Methode, wie Sie sicherlich zugeben werden. Man kann das Glas sehr gesittet zum Mund führen und womöglich dabei auch noch den Finger abspreizen. Man kann sich natürlich aus den Saft über das Gesicht giessen, was gewiss keine besonders brauchbare Methode ist. Statt diese Methode selbst auszuführen, können wir uns auch eine Maschine denken, die uns automatisch mehr oder weniger brauchbar den Apfelsaft in den Mund giesst. Solche Maschinen sind natürlich ebenfalls Objekte, die aber im Gegensatz zum Glas mit Apfelsaft *keine* Daten enthalten, sondern »funktionalen Code«. Diese Objekte nennen wir *Funktionsobjekte* oder einfach nur *Funktionen*.

1.2 Klassen

Nun wäre es gewiss wünschenswert, für das Entleeren eines Apfelsaftglases genau die »richtigen« Funktionen zur Verfügung zu haben, die nur für diese Art von Objekten gedacht sind. So ist beispielsweise das regelrechte Ausschütten eher für das Bewässern eines Gartens als für den Trinkvorgang geeignet. Das Abspreizen des Fingers ist – wenn es denn überhaupt einen Sinn dafür gibt – für das eigentliche Ziel völlig wirkungslos. Um zu verhindern, dass unbrauchbare oder gar falsche Funktionen auf Datenobjekte wirken, für die sie nicht gedacht sind, gibt es die sogenannte Klasse. In einer solchen Klasse werden Datenobjekte und ihre zugehörigen Funktionen zu einer Einheit verbunden und bilden ein hybrides Gebilde, was weder ein reines Datenobjekt noch eine reine Funktion ist. Auf diese Klassen *wirken* keine Funktionen mehr. Die in den Klassen enthaltenen Datensätze werden nur

noch durch die *klasseneigenen* Funktionen manipuliert. Statt von Funktionen wird aus diesem Grunde von *Methoden*. Der eigentliche Arbeitsvorgang passiert also *innerhalb* der Klasse und kann von aussen nicht gesteuert werden. Untereinander tauschen die Klassen nur noch *Nachrichten* aus, also Datensätze, die von Methoden einer Klasse an die Methoden einer anderen Klasse weitergereicht werden. Klassen sind demnach Objekte, die miteinander *kommunizieren* können, während reine Datenobjekte Gebilde sind, die kein Eigenleben haben, sondern reine Befehlsempfänger sind.

1.3 Namen

Wie nun wird auf Objekte *zugegriffen*? Es nützt Ihnen nichts, wenn Sie im Restaurant ein Glas Apfelsaft bestellen und Ihnen der Kellner antwortet: »Jawohl, das Glas steht bereit«, ohne Ihnen zu sagen, *wo* es steht. Sie müssen also gewissermassen die *Adresse* des Glases kennen. Die »Anrede« eines Objektes geschieht durch Aufrufen eines *Namens*, genauso wie wir es unter Menschen kennen. In der Tat kann unser *Name* gewissermassen als Adresse fungieren, wenn wir ihn rufen. Der Transport der Information geschieht durch Schallwellen in der Luft. Beim Computer übernimmt diesen Transport die CPU (Mikroprozessor), sofern eine eindeutige *Zuweisung* des Namens auf das Objekt *vor dem ersten Aufruf* vorgenommen wurde.

Eine solche Zuweisung nehmen Sie über den Python-Interpreter durch einen *Zuweisungsoperator* vor, der die Form des mathematischen Gleichheitszeichens besitzt. Wenn ein Datenobjekt, zum Beispiel die unvermeidliche Zeichenkette 'Hello world', den Namen »anna« erhalten soll, geschieht dies durch die Anweisung

```
>>>anna='Hello world'
```

Jetzt wird Ihnen der Interpreter bei jedem Aufruf von *anna* das Objekt *Hello world* auf dem Bildschirm als Antwort ausgeben¹. Geben Sie also einfach *anna* ein und Sie erhalten auf dem Bildschirm den folgenden Kommunikationsablauf:

```
>>>anna
```

¹Denken Sie bitte daran, unbedingt die Anführungszeichen zu setzen. Warum das notwendig ist, erfahren Sie im Abschnitt 2.2, in dem es um Textketten (Strings) geht.

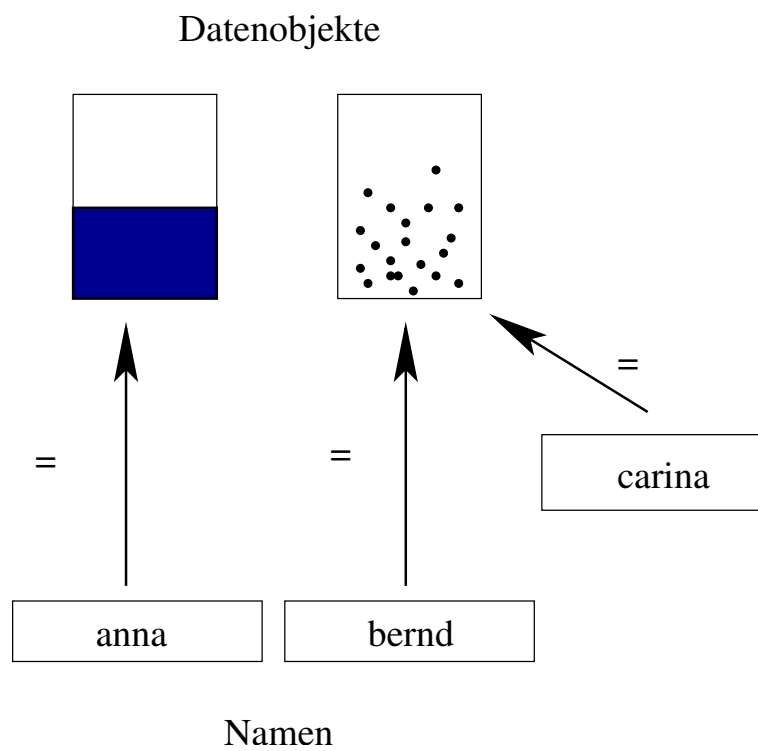


Abbildung 1.1: Datenobjekte

```
'Hello world'
>>>
```

Die Zuweisung eines Namens an ein Datenobjekt geschieht also immer auf dem Wege

$$name = objekt$$

In der Abbildung 1.1 sind die Datenobjekte als Rechtecke mit Inhalt gezeichnet. Auf das zweite Objekt zeigen *zwei* verschiedene Namen. Wenn sich der Verweis von *bernd* ändert, zeigt somit *carina* immer noch auf das alte Objekt!

1.3.1 Namenszuweisung für Funktionen*

Bei Funktionen funktioniert die Zuweisung ganz ähnlich. Da jedoch auch Werte an die Funktion zur Weiterverarbeitung übergeben werden können, wird hier ein anderer Operator verwendet, der *def* heisst, denn bei einer Funktion ist es mit einer reinen Namenszuweisung nicht getan. Es muss ja noch vorgegeben werden, was die Funktion machen soll. Sie sehen hier ein kleines Beispiel, das Sie vielleicht noch nicht hundertprozentig verstehen können. Alles wird Ihnen im Abschnitt 4 sehr genau erklärt.

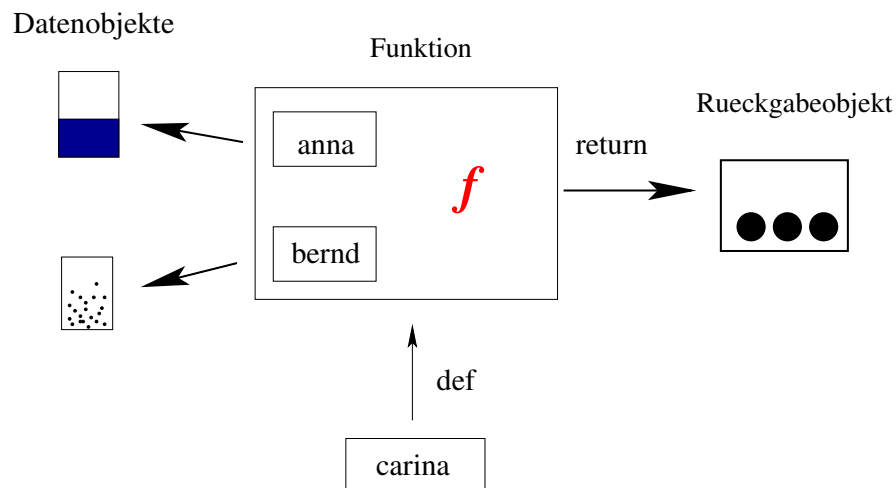


Abbildung 1.2: Funktionsobjekte

```
>>>def fakultaet(n):
...     j=1
...     for i in range(n):
...         j=j*i
...     return j
```

Die Funktion wird durch den folgenden Anweisungscode *definiert*. Daher der Name des Operators. Der Name der Funktion lautet natürlich *fakultaet*. Der Parameter *n* ist ein Name, der auf eine Zuweisung wartet. In der Tat wird die folgende Kommunikation ablaufen können:

```
>>>fakultaet(4)
24
>>>
```

Beim Aufruf wird der Name *n* an das Objekt *4* gebunden. Daraufhin wird der Anweisungscode abgearbeitet². Am Ende wird durch die *return*-Anweisung ein Objekt zurückgegeben. Soll mit dem Ergebnisobjekt weitergearbeitet werden, muss diesem ebenfalls ein Name, z.B. *ergebnis*, in der Form

```
>>>ergebnis=fakultaet(4)
```

zugewiesen werden.

In der Abbildung 4 sehen Sie eine Funktion, der der Name *carina* zugewiesen wurde. Als *Wertobjekte* werden zwei Datenobjekte so übergeben, dass die Namen *anna* und *bernd* auf sie zeigen. Wesentlich ist, dass sowohl *anna* als auch *bernd* von aussen nicht zugänglich sind, sondern Namen sind, die zur Funktion gehören. Welche Namen *innerhalb* der Funktion zugänglich sind³, entzieht sich ebenfalls dem Blick von aussen. Es kann⁴ ein Objekt nach aussen zurückgegeben werden.

1.3.2 Vereinbarung von Klassen*

Klassen werden durch den *class*-Operator vereinbart. Das *Prinzip* sieht wie folgt aus:

²Die Einrückungen sind sehr notwendig, denn sie stellen so einen Anweisungsblock dar. Die Einrückung ersetzt ein »begin« und »end« in anderen Sprachen.

³wie zum Beispiel der Name *j* in der Funktion *fakultaet*

⁴aber muss nicht

```
class Klassenname:

    name1=datenobjekt1
    name2=datenobjekt2

    def funktion1(parameter1,...):
        Anweisungscode

    def funktion2(parameter1,...)
        Anweisungscode
```

Also: zuerst werden die Datensätze vereinbart. Danach die zugehörigen Funktionen der Klasse, die ja Methoden genannt werden. Am besten zeigen wir ein Beispiel:

```
>>>class mathe:
...     self.j=1
...     def fakultaet(self,n)
...         for i in range(n):
...             self.j=self.j*i
...     return self.j
```

Warum der Parameter *self* notwendig ist, erfahren Sie genauer im Abschnitt 5. Probieren wir das Beispiel aus:

```
>>>anna=mathe()
>>>anna.fakultaet(4)
24
>>>
```

In der ersten Zeile wird ein Objekt vom Typ *mathe* eingerichtet⁵. Der Name *anna* wird von der Klasse im Parameter *self* gespeichert. Wunderbar. Aber was soll das?

Nun, das alles wird Ihnen wie gesagt, genauer im Abschnitt 5 erklärt. Es ging hier nur darum, zu zeigen, was Klassen *sind*. Als Motivation, dies genauer zu lernen: *alle*, aber auch wirklich *alle* Dinge beim grafischen Programmieren

⁵Ja, die runden Klammern sind wirklich notwendig!

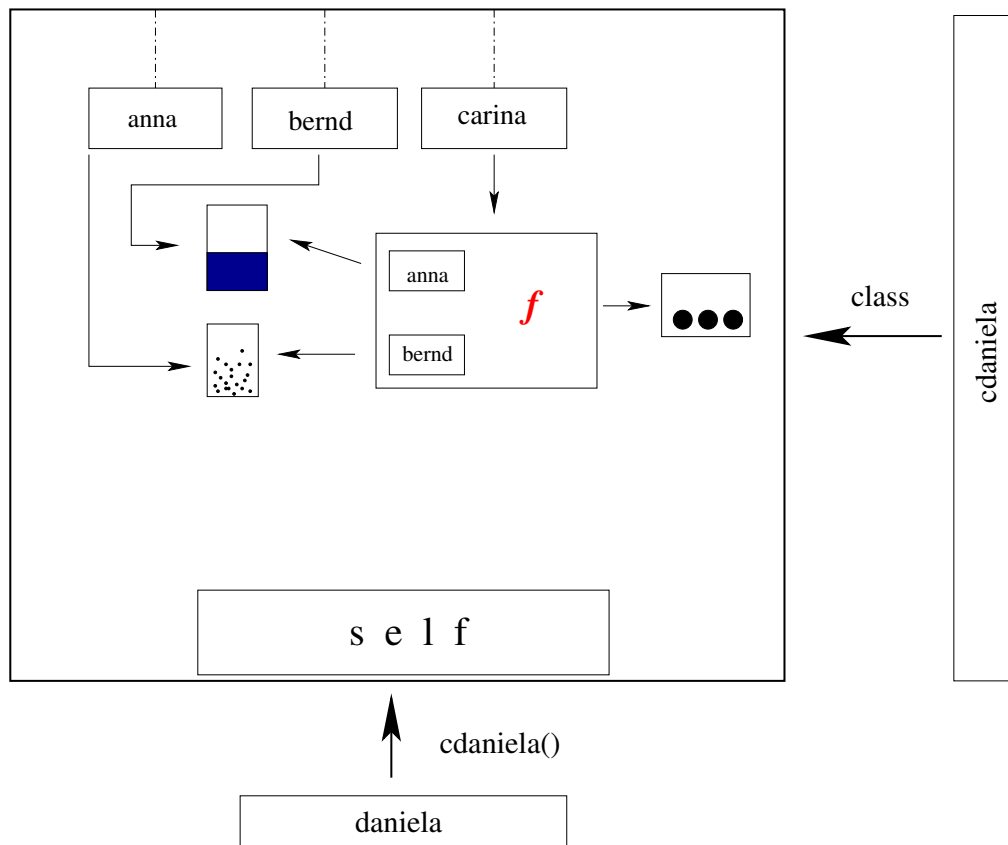


Abbildung 1.3: Klassen

sind solche Klassen. Ein *button* ist eine Klasse mit ganz bestimmten Methoden, eine Menüliste ist wiederum eine andere Klasse mit anderen Methoden usw.

Die Abbildung 5 zeigt noch einmal die Vorgänge innerhalb einer Klasse. Der einzige von aussen zugängliche, aber nicht verfügbare Name ist *self*. Diesem wird der Name des Objektes übergeben, das von diesem Klassentyp ist. In diesem Fall heisst das Objekt *daniela*. Der Objektname ist streng vom Klassennamen *cdaniela* zu unterscheiden. Der Name *cdaniela* zeigt *auf die Klasse als Objekt*, während *self* eine Verbindung eines eingerichteten Objektes vom Typ *cdaniela* mit den internen Namen *anna*, *bernd* und *carina* vornimmt. Wie wir später sehen werden, ist *self* sozusagen die Kommunikationszentrale *zwischen* den einzelnen Objekten.

1.4 Vergleich mit anderen Programmiersprachen*

Es ist *völlig egal*, von welcher Art das Objekt ist, dem Sie einen Namen zuweise. Aufrufe geschehen grundsätzlich *nur* über Namen. Wenn Sie schon Erfahrungen mit anderen Programmiersprachen, wie z.B. C , C++ , Java oder Pascal haben, werden Sie sich fragen, warum man sich keine Gedanken über die Grösse des einzurichtenden Objektes machen muss, weshalb also *keine Typendeklaration* erforderlich ist.

Nun, Sie haben vielleicht aber auch schon festgestellt, dass das was wir hier als *Namen* bezeichnen, die *Pointer* in diesen Sprachen entsprechen⁶. Ein Pointer ist durch eine *weiche Bindung* (soft link) an das Objekt gebunden, auf das er zeigt. Bei Python gibt es *nur* die weichen Bindungen! In den genannten Programmiersprachen gibt es aber zusätzlich noch die *harte Bindung* (hard link). Wenn Sie beispielsweise in C *anna=4* eingeben, so zeigt der Name *anna* nicht auf das Objekt *4* , sondern *ist* identisch mit der Adresse des Objektes *4* .

Wie bitte? Noch einmal: bei einem weichen Link ist der Name ein Container, der die Adresse des angebundenen Objektes *enthält*. Bei einem harten Link stellt dieser Name die Adresse selbst dar. Ist der Name *anna* wie bei Pascal durch *anna=4* hart mit dem Speicherplatz verbunden, so würde eine folgende Zuweisung *anna=5* keinen neuen Speicherplatz einrichten, sondern

⁶Genauer gesagt: sie sind ein typenloser Pointer.

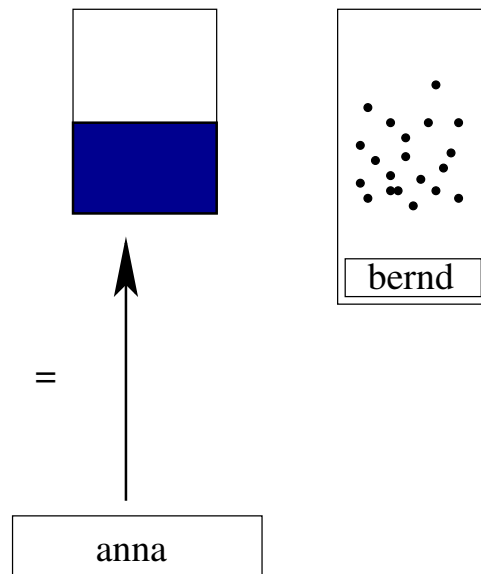


Abbildung 1.4: Hartes und weiches Linken

lediglich den *Inhalt* des Speicherplatzes *anna* verändern. Wenn Sie in diesen Sprachen versuchen, eine Zuweisung *anna*='Hello world' abzulassen, so hätte dies einen Fehler zur Folge, wenn *anna* vorher nur Zahlen gespeichert hat. Ein Text erfordert sehr viel mehr Arbeitsspeicher⁷ als eine Zahl. Also muss bei einem harten Link *vorher* gesagt werden, wie gross der Speicherplatz sein muss, d.h. eine Typendeklaration des einzurichtenden Namens ist erforderlich. Diese Sachverhalte sind in der Abbildung 1.4 dargestellt. Der Name *bernd* ist also direkt mit dem Objekt verbunden.

Bei einem weichen Link wie bei Python wird erst das Objekt erstellt und dann lediglich die Adresse an den Namen übergeben, der auf dieses Objekt zeigen soll. Die Grösse des Speicherplatzes kann dem Namen egal sein, da dies durch den Python-Interpreter selbst erledigt wird.

Wird der Name eines hart gelinktes Objekt gelöscht, so ist das Objekt selbst weg. Bei einem weichen Link wird dagegen nur die Verbindung zu dem Objekt aufgehoben. Es existiert dann im Arbeitsspeicher als *Speicherleiche*, die dann durch den automatischen *garbage collector* (wie auch in Java) automatisch entfernt wird.

⁷Im einfachsten Fall je ein Byte für jedes Zeichen (siehe Anhang).

Wir merken uns also: bei Python ist eine Typendeklaration *nicht* erforderlich! Es gibt nur Namen.

1.5 Module

Ein *Modul* ist nun nichts weiter als der Ort, an dem Namen gespeichert werden. Dieses ist lediglich eine Datei, in der Sie den Code speichern. Wir werden sehr bald verstehen, dass ein solches Modul sozusagen ein *Programm* darstellt. Statt also alles am Interpreter direkt auszuprobieren, benutzen Sie den Editor von IDLE und erstellen Sie eine Datei mit der Dateiendung *.py*, zum Beispiel *test.py*. In diese Datei schreiben Sie dann ihren Code genauso ein, als ob Sie ihn direkt am Interpreter-Prompt eingeben würden.

Dann rufen Sie den Interpreter auf und laden das Modul mit dem Befehl *import test*⁸. Sodann läuft der Code automatisch ab. Der Witz ist der, dass Sie mit den durch das Modul eingerichteten Namen nach »Ablauf« des Programms weiterarbeiten können. Ein Beispiel: schreiben Sie im Editor den Text

```
#Modul test
anna=4
3+5*anna
print 'Das war ein erstes Beispiel'
```

Speichern Sie dieses Modul als *test.py* ab. Rufen Sie dann im Interpreter das Modul wie folgt auf

```
>>>import test
Das war ein erstes Beispiel
```

Geben Sie danach am Interpreterprompt lediglich

```
>>>anna
```

⁸Jawohl, *ohne* die Endung *.py*. Besser ist es, danach noch einen Aufruf *from test import ** abzusetzen. Warum, erfahren Sie im Abschnitt 6.

ein, um zu überprüfen, ob *anna* tatsächlich noch verfügbar ist. Sie werden sehen, dass dies entweder nicht der Fall ist, oder irgendein anderes Objekt von früher, z.B. *'Hello world'* ausgegeben wird, aber jedenfalls nicht *4*, was ja im Modul vereinbart wurde. In der Tat, der Name muss wie folgt aufgerufen werden:

```
>>>test.anna
4
```

Da haben wir es! Wenn wir in Zukunft bei den durch das Modul eingerichteten Namen nicht immer *test.* davorschreiben wollen, muss der Aufruf

```
>>>from test import *
```

angeschlossen werden. Jetzt funktioniert die Eingabe *anna* ohne die Spezifizierung *test.* wie gewünscht.

Ändern Sie jetzt einmal das Modul so, dass im Text die Zuweisung *anna=7* erfolgt. Importieren Sie wieder das Modul und überprüfen Sie, worauf *anna* jetzt zeigt.

Sie werden feststellen, dass dieses keineswegs die *7* ist. Warum? Python stellt fest, dass bereits der Name *anna* schon existiert! Ein neuer Import wird daher nicht vorgenommen. Ja, *anna* würde in der alten Form sogar dann noch verfügbar sein, wenn der Name im veränderten Modul sogar gelöscht wurde!

Das wirkt auf den ersten Blick ärgerlich, ist aber von sehr grossem Vorteil, wie Sie später einsehen werden. Um ein verändertes Modul nun vollständig, also auch mit den Veränderungen neu zu laden, muss

```
>>>reload (test)
```

eingegeben werden. Jetzt aber steht das neue *anna* wieder nur als *test.anna* zur Verfügung. Ein neuerliches *from test import ** ist also notwendig.

In der Abbildung 1.5 wird gezeigt, wie das Ganze zu verstehen ist. Module enthalten eigentlich nur Namen, in diesem Falle *cdaniela* und *cerica*, die ausserhalb der definierten Klassen sichtbar sind. Wenn ein Modul importiert wird, landet es zunächst als *Ganzes* in dem Hauptmodul mit dem Namen

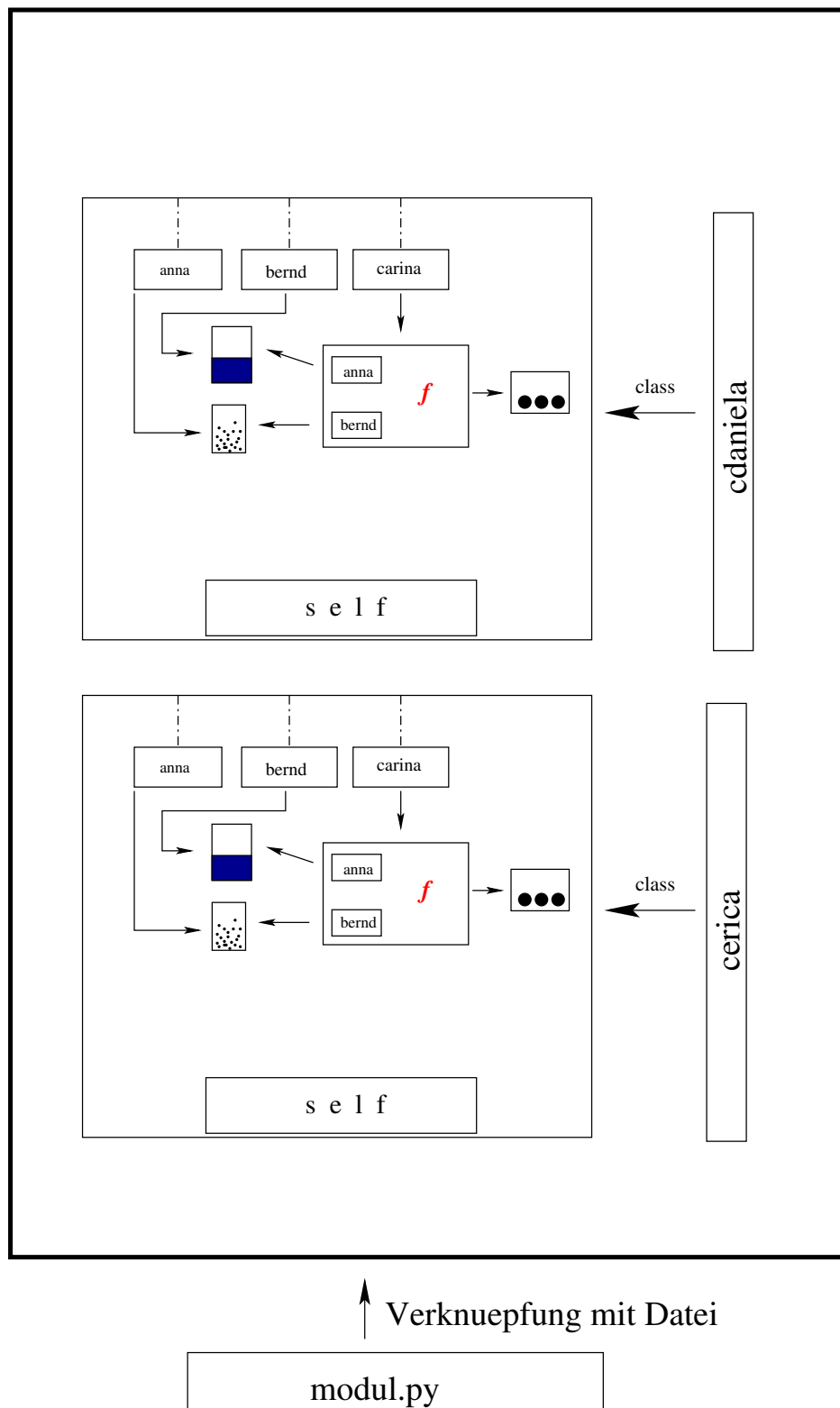


Abbildung 1.5: Module sind Namenscontainer

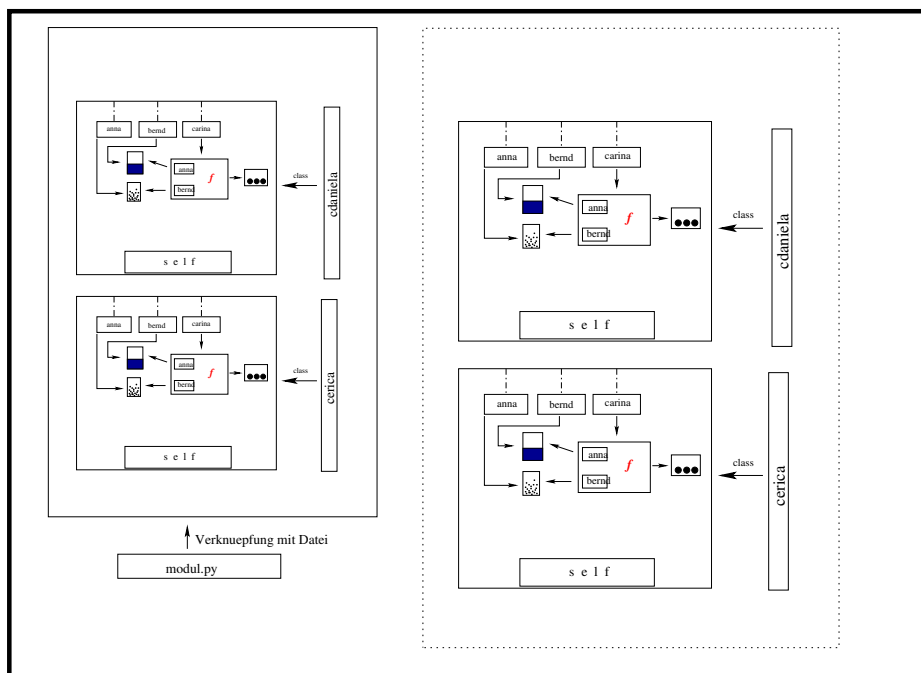


Abbildung 1.6: Das Mainmodul der Python-Umgebung

`__main__` (jeweils zwei Unterstriche am Anfang und Ende) der Python-Umgebung. In diesem Mainmodul ist nur der Modulname selbst sichtbar. Auf die Namen *innerhalb* des Moduls muss also mit Hilfe des Selektors `».` zugegriffen werden.

Wenn mit *from* der *Inhalt* eines Moduls importiert wird, wie bei dem rechten Modul, bei dem der Rahmen gestrichelt gezeichnet wurde, so stehen diese Namen alle so zur Verfügung, als hätte man sie direkt im Interpreter definiert. Wenn diese Namen aber schon durch einen vorherigen Import existieren, muss das gesamte Modul nachgeladen und ein erneutes *from* durchgeführt werden.

1.6 Zusammenfassung

Fassen wir noch einmal zusammen.

1. Die kleinste Form eines Objektes ist das Datenobjekt, das irgendetwas enthalten kann, wie ein Glas Apfelsaft seinen Inhalt. Es gibt nur sechs Datenobjekte.
2. Über Anweisungen lassen sich die Inhalte von Datenobjekten verändern. Es gibt nur drei wesentliche Anweisungen, der Rest sind sogenannte Operatoren, die eine Zuordnung oder einen Vergleich zwischen Objekten vornehmen⁹.
3. Werden Anweisungen zu einer Einheit miteinander verbunden, erhalten wir ein *funktionales Objekt* in Form einer Maschine. Solche Objekte nennen wir *Funktionen*.
4. Um zu verhindern, dass die falschen Funktionen auf Datenobjekte wirken, werden durch *Klassen* Funktionen mit Datenobjekten gekoppelt, wobei eine neue Art von Objekten entsteht, die gewissermassen hybrid sind. Sie sind sowohl Datenobjekte als auch Funktionen.
5. Ein *Modul* ist nichts weiter als eine Ansammlung von Namen, die auf vereinbarte Objekte zeigen. Die Vereinbarungen selbst müssen natürlich im Modul erfolgen.

⁹Das bereits schon angesprochene Gleichheitszeichen `»=«` ist ein solcher Operator, der keine mathematische Feststellung bedeutet, sondern einen Namen mit einem Objekt verbindet.

Kapitel 2

Datenobjekte

Zu diesem Kapitel siehe Abbildung 1.1.

Wie wir bereits sagten, gibt es nur sechs Datenobjekte. Hier sind sie:

2.1 Zahlen

Dies sind die kleinsten Objekte, die es gibt. Man unterscheidet ganzzahlige Zahlobjekte (*Integer*), Gleitkommazahlen (*Floats*), sowie die Ihnen sicherlich nicht in jedem Fall bekannten »komplexen Zahlen« (*Complex*¹). Das Einzige, was Sie beachten müssen ist, dass eine Zahl um so mehr Speicherplatz im Arbeitsspeicher benötigt, je komplizierter sie aufgebaut ist. In diesem Sinne sind Ganzzahlobjekte die kleinsten. Probieren Sie einmal der Reihe nach aus:

```
>>>2+3
>>>2*3
>>>2+3.0
>>>3.0/2
>>>3/2
>>>3.0/2.0
>>>8%2
>>>7%2
>>>4**2
```

¹Wenn Sie wissen was das ist, wissen Sie auch, wozu es nützlich ist, mit diesen Zahlen umzugehen. Wenn nicht, schadet es an dieser Stelle nichts.

```
>>>j*j
>>>(1-j)*2
>>>(1+j)*j
```

Wie Sie sehen, kann der Interpreter als »Taschenrechner« arbeiten. Die letzten beiden Fälle beziehen sich auf komplexe Zahlen, wobei die sogenannte »imaginäre Einheit« das j bedeutet². Die Zahl » j « ist so etwas wie » $\sqrt{-1}$ «, wobei diese Schreibweise mit grösster Vorsicht zu geniessen ist. Die Zahl j hat die Eigenschaft $j \cdot j = -1$. Sie hat auch eine anschauliche Bedeutung, aber ich will nicht wortbrüchig werden und nun doch komplexe Zahlen genauer erklären.

Was die Informatik betrifft, leisten die Zeichen $+$, $-$, $*$, $/$, $**$ und $\%$ wesentlich mehr als nur die Ausführung einer Rechenoperation. Diese Zeichen, die aus diesem Grunde auch *Operatoren* heissen, generieren aus zwei Zahlenobjekten ein drittes, das Ihnen als »Ergebnis« angezeigt wird. Nach der Abarbeitung der *Anweisung* $2*3$ sind sowohl die »2« als auch die »3« und das Ergebnis »6« als Objekte verloren. Man kann auf sie nicht mehr zugreifen. Wozu auch? Nun, soll mit dem Ergebnis weitergerechnet werden, ist die Anbindung an einen Namen notwendig.

```
>>>Ergebnis=2*3
>>>10**Ergebnis
```

zeigt, dass jetzt der Name *Ergebnis* auf das Ergebnisobjekt der Operation » $2*3$ « zeigt. Das Beispiel ist hinreichend dämlich. Daher gehen wir möglichst schnell zu einem etwas interessanteren Datenobjekt gleich über. Bevor wir dieses tun nur noch ein Hinweis. Probieren Sie aus:

```
>>>2/10E-10
>>>2/0.0000000001
```

Sie sehen ein paar kleine Überraschungen. $10E-10$ bedeutet natürlich 10^{-10} . Sie sehen, dass bei derart grossen Zahlen die Rechengenauigkeit keine Begeisterungswelle auslöst. Tatsächlich ist Python als Taschenrechner mit Gleitkommazahlen zunächst nur für kleinere Rechnungen sehr gut zu gebrauchen. Es gibt ein separates Modul oder sogar eine mathematische Erweiterung von

²Haben Sie eigentlich herausgefunden, was der Operator $\%$ durchführt? Es ist eine Restklassenbestimmung (*modulo*).

Python, die jeden Wunsch erfüllt. Wir wollen uns hier noch nicht damit abgeben. Sie sehen, dass die zweite Rechnung *genauer* ist.

Sehr merkwürdig ist aber, dass

```
>>>2*10000000000
```

am Ende ein »L« an die Zahl gehängt wird. Dies ist ein sehr grosser Vorteil gegenüber vielen anderen Sprachen. Python kennt beliebig grosse Ganzzahl-objekte! Normalerweise gibt es je nach Art der Abspeicherung eine untere und obere Grenze (siehe Anhang). Durch Anhängen eines »L« wird diese Beschränkung aufgehoben. Rechnen Sie daher möglichst *immer* mit Integerzahlen. Notfalls hängen Sie ein L an das Ende. Natürlich dauert die Rechnung dann etwas länger, da Long-Integer sehr viel mehr Speicherplatz benötigen. Aber es ist immer noch im allgemeinen besser als das Rechnen mit Gleitkommazahlen.

Wenn wir wieder an unser Apfelsaftglas denken, so ähnelt dieser sehr elementare Datentyp sehr viel weniger diesem Glas als vielmehr dem Apfelsaft selbst. Kommen wir nun also zu den Apfelsaft*gläsern*.

2.2 Strings

Strings sind Zeichenketten, die durch Hochkommata eingeschlossen werden. Dabei ist es egal, ob Sie zwei oder ein Hochkomma setzen. Verwenden Sie gar *drei* Hochkommata, kann der Text beliebig lang werden. Die Anweisung

```
>>>beispiel='Dies ist ein Beispiel.'
```

weist dem Objekt »'Dies ist ein Beispiel.'« den Namen »beispiel« zu und kann daher einfach ausgedruckt werden. Bedenken Sie, dass Namen *keine* Zeichenketten sind und *niemals* durch Hochkommata eingeschlossen werden dürfen. Die Bezeichnung *beispiel* ist Name für ein Objekt. Im Gegensatz dazu ist »'beispiel'« kein *Name* für ein Objekt, sondern das Objekt selbst, das den Inhalt 'beispiel' besitzt. Genauso wie ein alleinstehendes Zahlenobjekt steht der Aufruf

```
>>>'beispiel'
```

in der Luft und kann nach der Einrichtung nicht wieder aufgerufen werden, da eine Zuweisung zum Namen fehlt.

Jetzt kommt etwas Interessantes. Führen Sie aus

```
>>>'Bete '+'und '+'arbete.'
```

Wie Sie sehen, hat der Additionsoperator hier die Funktion einer Verkettung. Er generiert allerdings einen *neuen* String, also ein *neues* Objekt, das einem Namen zugewiesen werden kann, wie im folgenden Beispiel

```
>>>ergebnis='Bete '+'und '+'arbete.'  
>>>ergebnis
```

Genau wie im Falle eines Gemüsetopfes, aus dem Sie sich etwas auf den Teller füllen, können Sie bei Strings auf Teile desselben zugreifen. Dieses Herausziehen von Teilen mit Hilfe eines Indexes nennt man *slicing*. Allerdings ist dieser Vergleich nicht so besonders gut, denn es bleibt noch genauso viel im Topf wie vorher. Es wird lediglich ein *neuer* Topf eingerichtet, der den herausgegriffenen Teil als Kopie enthält.

```
>>>ergebnis[0]
```

liefert Ihnen den ersten Buchstaben. Jawohl, bei Python fängt man immer mit dem Zählen bei 0 und nicht bei 1 an!

```
>>>ergebnis[2:6]
```

liefert Ihnen als Ergebnisstring »'te u'«, also den String vom dritten bis zum siebten Buchstaben³. Genausogut können auch eine Grenze fortgelassen werden

```
>>>ergebnis[:6]
```

oder sogar beide

³Sagten wir schon, dass wir bei 0 mit dem Zählen beginnen? ;-)

```
>>>ergebnis[:]
```

Der letzte Fall scheint etwas sinnlos zu sein. Aber er ist es nicht, wenn Sie bedenken, dass »ergebnis[:]« *keine Namenszuweisung* ist (!!), sondern ein ganz neuer String, der mangels Namens in der Luft hängt. Eine Zuordnung

```
>>>neuerString=ergebnis[:6]
```

»rettet« gewissermassen den Inhalt »Bete u«, indem er dem Objekt »ergebnis[:6]« den Namen »neuerString« gibt.

Jetzt wissen wir das Wichtigste über Zeichenketten⁴. Noch einmal: wo ist der Unterschied zwischen den Objekten 2 und '2'? Nun, im zweiten Fall ist es *keine* Zahl, sondern ein Zeichenobjekt. Sie wissen bereits, was

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

bedeutet. Aber was gibt

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

am Ende? Die *Zahl* »23«? Nun, versuchen Sie einmal mit dem Ergebnis eine Division durchzuführen. Sie werden hoffentlich *keine* Überraschung erleben.

Strings ähneln sehr stark den Apfelsaftgläsern, denn sie enthalten »Datenmaterial«, das man teilweise ausschütten kann wie wir gesehen haben. Nun lernen wir die Regale kennen, in die wir die Apfelsaftgläser stellen können.

2.3 Listen

Listen sind mit Abstand der wichtigste Datentyp. Genauso wie Strings sind Listen zusammengesetzte Objekte. Bei ihnen kann jedoch das was an den einzelnen Stellen steht, ein beliebiges Objekt sein. Listen sind Aufzählungen, die durch eckige Klammern eingeschachtelt sind. So sind zum Beispiel

⁴C-Programmierer einmal hergehört: es gibt keine Objekte vom Typ CHAR. Und das Besondere ist: das soll auch bitte sehr so bleiben, denn diese Objekte sind nicht notwendig ;-).

```
>>>drei=['Bete ','und']  
>>>['Eins',2,drei]
```

solche Listen. Wie sie sehen, enthält die zweite (keinem Namen zugewiesene) Liste die Liste »drei« an dritter Stelle. Geben Sie dieser Liste doch am besten einen Namen, etwa

```
>>>aufDiePlaetze=['Eins',2,drei]
```

Genauso wie bei Strings gibt es hier ein *Slicing*. Geben Sie ein:

```
>>>aufDiePlaetze[2]
```

Alle Aktionen, die wir bei Strings gesehen haben, lassen sich übertragen. Aber es gibt einen kleinen, jedoch *sehr* feinen Unterschied.

```
>>>drei=drei+['arbeite.']  
>>>drei  
>>>aufDiePlaetze
```

Was ist jetzt los? In der ersten Zeile wurde dem Namen »drei« ein Objekt zugewiesen, das aus dem alten Objekt, auf das der Name vorher zeigte (also die Liste ['Bete ', 'und ']) offenbar eine neue Liste macht, die um 'arbeite' länger ist. Nun, genau das bestätigt anscheinend die nächste Eingabe, jedoch nicht die dritte. Die Liste »aufDiePlätze« sieht noch genau so aus wie vorher. Der Liste »aufDiePlätze« wurde bei der Zuweisung der Name »drei« übergeben, der auf ein bestimmtes Objekt mit einem bestimmten Inhalt zeigte. Dieser alte Inhalt ist bei Veränderung des Objektes, auf das »drei« zeigt, immer noch da! Durch den Additionsoperator wurde ein neues Objekt erstellt, auf das nun der Name »drei« zeigt. Davon wird die Liste »aufDiePlätze« nicht berührt.

Wird nun aber die Liste »drei« an Ort und Stelle verändert, indem wir dem ersten Element einen anderen Wert zuweisen,

```
>>>drei[0]='Arbete '
```

so zeigen die Aufrufe

```
>>>drei
>>>aufDiePlaetze
```

die Auswirkungen. Sowohl die Liste »drei« als auch die Liste »aufDiePlaetze« ist von der Veränderung betroffen, denn durch die Zuweisung »drei[0]='Arbete'« wird *kein* neues Objekt erstellt, sondern das alte lediglich verändert! Der Name »drei« zeigt also immer noch auf dasselbe Objekt!! Bedenken Sie, dass im Falle eines Strings der Form

```
>>>drei='Bete und arbete.'
```

zwar ein Zugriff auf »drei[0]« möglich ist, aber *keine Zuweisung* der Form »drei[0]='Arbete'«. Strings sind *unveränderliche Sequenzen*, während Listen *veränderliche Sequenzen* darstellen. Bei Listen ist *listennam[i]* immer ein *Name*, der auf ein Objekt zeigt, während bei Strings dieses eine *Anweisung* ist, um ein neues Objekt zu erstellen, der lediglich das Zeichen an der Stelle *i* enthält.

Listen sind also in der Tat Regale, aus denen man bestimmte Objekte verändern kann. Ja, es ist auch möglich, wie wir später sehen werden, Objekte aus diesen Regalen zu entfernen oder hinzuzufügen. Listen sind die dynamischsten Objekte, die es in Python gibt. Daher werden sie auch so häufig verwendet⁵.

Im folgenden Abschnitt lernen wir nun einen wieder etwas weniger dynamischen Datentyp kennen, der viel eher der herkömmlichen Vorstellung einer Liste entspricht.

2.4 Tupel

Tupel unterscheiden sich von Listen lediglich in den folgenden Aspekten:

1. Tupel werden verwendet statt eckigen Klammern, runde Klammern, z.B. (*eins*, 2, *drei*).

⁵Die Idee stammt übrigens von sogenannten *praedikativen Programmiersprachen* wie Lisp (=list processing) oder Prolog, die Probleme auf logischem Wege durch eigenes »Dazulernen« (=Listenverlängerung) lösen können. Diese Sprachen haben eine gewisse Bedeutung in der sogenannten künstlichen Intelligenz.

2. Tupel sind im Gegensatz zu Listen *unveränderliche Sequenzen* wie Strings auch.

Es ist also nicht möglich, in einem Tupel irgendwelche Veränderungen vorzunehmen. Haben wir

```
>>>a='Arbete ','und ','arbete.'
```

dem Interpreter mitgeteilt⁶, so gibt eine Zuweisung der Form

```
>>>a[2]='Feierabend.'
```

eine Fehleranzeige. Es muss also 'weitergearbeitet' werden, da Tupel unveränderlich sind, wie jene Regale in einer Wohnung, die per Vorgabe durch den Eigentümer immer dasselbe enthalten sollen, wie schon seit Urgrossvaters Zeiten.

Wozu sind diese »engstirnigen« Datenobjekte dann gut? Wie wir noch sehen werden, spielen sie bei Funktionen eine ganz entscheidende Rolle!

2.5 Dictionaries

Dictionaries entsprechen nun Listen, die quasi ungeordnet sind. Statt über einen Index wird auf einzelne Teile des Dictionaries über Schlüsselobjekte zugegriffen. Dictionaries werden in *Mengenklammern* eingeschlossen, denn wie bei einer mathematischen Menge ist die Reihenfolge egal, da es Schlüsselobjekte gibt. Ein Beispiel, das Sie unbedingt ausprobieren sollten:

```
>>>lexikon={'eins': [1,2], 2: ('8',9)}
>>>lexikon['eins']
[1,2]
>>>lexikon[2]
('8',9)
>>>
```

⁶Jawohl, hier liegt *kein* Fehler vor! Sie dürfen auch die runden Klammern weglassen.

Links vom Trenner »:« steht das Schlüsselobjekt und rechts das diesem Schlüssel zugeordnete Objekt. Sehr wichtig ist der Hinweis, dass links neben dem Trenner : ein Objekt oder eine Name stehen muss, der auf ein Objekt zeigt. Der Trenner : bedeutet *keine* Zuweisung. Der Aufruf *lexikon[2]* funktioniert also nur deswegen, weil 2 ein Objekt ist. Fügen Sie in der ersten Zeile im Dictionary noch den Eintrag *null: 0* hinzu, so dass also

```
>>>lexikon={'eins': [1,2], 2: ('8',9),null=0}
```

eingegeben wird, so erhalten Sie eine Fehlermeldung, da *null* nicht auf das Objekt 0, sondern ins Leere zeigt.

Während in Listen den einzelnen Objekten, die die Listenkomponenten darstellen, ein Name, nämlich *listenname[i]*, zugewiesen wird, ist dies bei den Dictionaries anders. Das Dictionary stellt keine Zuordnung zwischen Namen und Objekten, sondern zwischen *Objekten und Objekten* her. Sie »bilden« also Objekte auf andere Objekte ab. Daher werden Dictionaries auch als *Abbildungen* bezeichnet.

Es dürfen auch Namen Schlüssel sein, sofern sie auf ein Objekt zeigen. Geben Sie bitte ein:

```
>>>anna=2
>>>lexikon={'eins': [1,2], anna: ('8',9)}
>>>lexikon[anna]
('8',9)
>>>
```

Der Aufruf *lexikon[anna]* funktioniert also, da *anna* vorher einem Objekt zugeordnet wurde. Wäre die Anweisung *anna=2* vorher nicht erfolgt, würde der Name *anna* im Dictionary ins Leere zeigen würde und es gäbe eine Fehlermeldung.

Genauso wie Listen, sind Dictionaries an Ort und Stelle *veränderlich*!

In der Abbildung 2.1 werden noch einmal Dictionaries mit Strings, Listen und Tupeln verglichen. Oben in dieser Abbildung sehen wir einen String, auf den wir zwar durch Indizierung *stringname[i]* zugreifen können. Es ist aber nicht möglich, den Inhalt der Stelle mit der Nummer *i* zu verändern. Um sich zu merken, dass die Indizierung stets bei 0 und nicht bei 1 beginnt, denken Sie sich an Stelle der einzelnen Stellen die *Schnitte* zwischen den

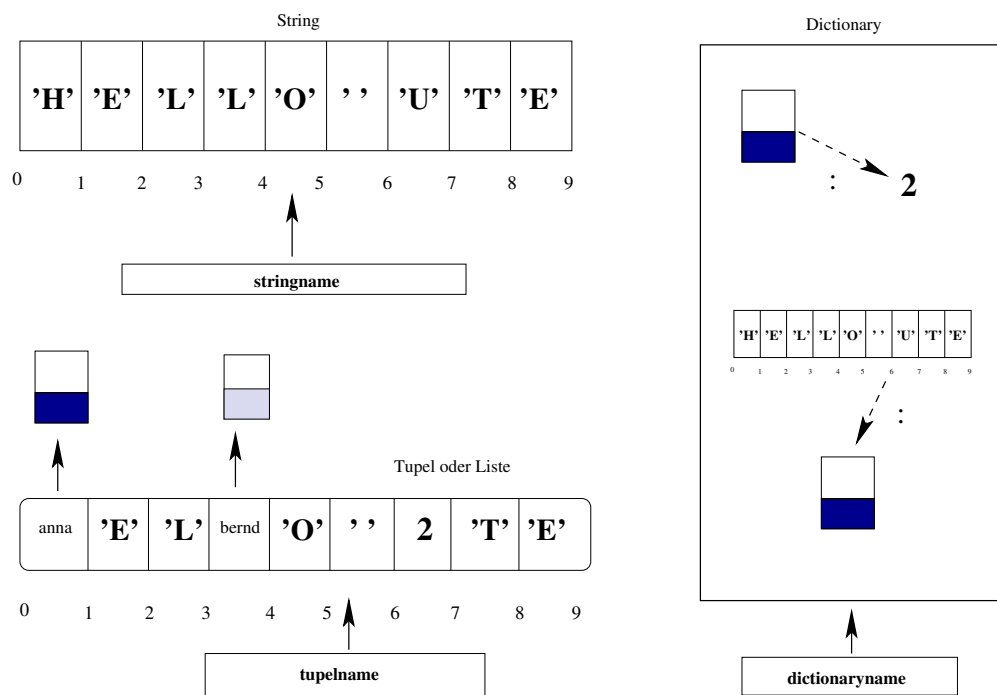


Abbildung 2.1: Dictionaries im Vergleich mit Strings, Listen und Tupel

Stellen numeriert. Die Stelle *rechts* von dem Schnitt ist dann diejenige, die zu der jeweiligen Nummer gehört.

Ein Tupel und eine Liste unterscheiden sich vom String dadurch, dass sie verschiedene Objekte als Inhalt haben können. Dabei können die Objekte sehr wohl durch ihre Namen ersetzt werden, wenn ihnen denn solche zugeordnet sind. Ein Tupel ist dabei wieder ein unveränderliches Gebilde, während bei einer Liste die Inhalte der einzelnen Stellen ausgetauscht werden können. Die Liste selbst kann sogar schrumpfen und wachsen.

Wie nun ein Dictionary arbeitet sehen Sie rechts. Hier werden Objekte anderen Objekten zugeordnet, aber nicht zugewiesen. Ein Aufruf eines Schlüsselobjektes (Objekt, von dem der Pfeil ausgeht) geschieht durch *dictionaryname[schlüsselobjekt]*. Ausgegeben wird dann das Objekt, dem *schlüsselobjekt* zugewiesen wurde. Sie sehen, dass durch diese Zuordnung keine auf eine »Reihenfolge« verzichtet werden kann. Werden als Schlüsselobjekte ausschließlich *Zahlen* verwendet, so ersetzt das Dictionary quasi eine Liste. Auf diese Weise lassen sich Listen realisieren, deren Indizierung z.B. nur durch gerade Zahlen erfolgen soll, während Indizierungen durch ungerade Zahlen aus irgendwelchen Gründen nicht vorkommen sollen. Ein Dictionary ist also eine noch flexiblere »Liste« als die Liste selbst.

2.6 Dateien

...sind Datenobjekte, die einer Datei auf der Festplatte zugeordnet sind. Damit haben wir einen ähnlichen Fall wie bei einem Dictionary, weil einem Objekt ein anderes zugeordnet wird. Nehmen wir an, es gäbe auf unserer Festplatte eine Datei mit dem Namen *test.py*⁷, so wird eine Zuordnung des Namens *meineDatei* auf *test.py* wie folgt vorgenommen:

```
>>>meineDatei=open('test.py','w')
```

Der Parameter *'w'* gibt an, dass in die Datei geschrieben werden soll. Wollen Sie aus einer Datei lesen, muss dieser Parameter *'r'* heissen. Mit

```
>>>meineDatei.write('Hello world\n')
```

⁷Die Datei, auf die verwiesen wird, kann eine beliebige Textdatei sein.

schreiben Sie eine Zeile Text in die Datei⁸. Sie sehen, dass Sie hier über eine Spezifikation mit dem Selektor `».` auf die Datei zugreifen müssen. Es können ja schliesslich mehrere Dateien geöffnet sein. So wird die Anweisung `anna.write()` von der Anweisung `meineDatei.write()` unterschieden. Diese Art des Aufrufes hat bereits sehr viel mit Klassen zu tun, so dass diese Details im Kapitel 5.3 noch einmal wiederholt werden. Schliessen Sie mit

```
>>>meineDatei.close()
```

die Datei wieder und schauen Sie nach, ob Sie in Ihrer Datei wirklich die eingegebene Textzeile finden. Sie lesen übrigens aus einer Datei mit der Methode `meineDatei.readline()`.

Grundsätzlich können zunächst nur Textdateien bearbeitet werden.

⁸`\n` bedeutet einen Zeilensprung.

Kapitel 3

Anweisungen

Anweisungen führen mit dem Inhalt von Objekten irgendetwas aus. Wir wollen nun alle folgenden Ausführungen an einem Beispiel demonstrieren, dass wir sukzessive von Abschnitt zu Abschnitt weiterentwickeln wollen. Wir führen eine sehr merkwürdige Liste ein, die wir »feld« nennen wollen:

```
>>>feld=[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
```

Hier haben wir schon eine Anweisung benutzt, die wir bereits intensiver kennengelernt haben: die Zuweisung eines Namens zu einem Objekt durch den Zuweisungsoperator »=«. Wir führen für *dasselbe* Objekt einen zweiten Namen ein, den wir »sfeld« nennen wollen¹.

```
>>>sfeld=feld
```

Noch einmal: *jetzt* bedeutet die Angabe »feld« auf der rechten Seite des Operators, das Objekt, auf das Feld zeigt. Diesem wird der *zweite* Name »sfeld« zugewiesen.

Wenn Sie der Unterschied zwischen den Namenszuweisungen in Python im Vergleich zu anderen Programmiersprachen interessiert, finden Sie eine detaillierte Erläuterung im Abschnitt 1.4. Dort wird Ihnen der Unterschied zwischen hartem und weichem Linken erklärt. Da es das sogenannte harte Linken in Python nicht gibt², ersparen wir uns die Wiederholung dieser Details. Aber es lohnt sich dennoch, den Unterschied zwischen den beiden Namensanbindungen zu kennen.

¹ Ja, es handelt sich um ein ganz kleines Spiel!

² was eine Typdeklaration überflüssig macht!

3.1 Vergleiche

Probieren Sie nacheinander die folgenden Zeilen aus:

```
>>>sfeld==feld
>>>sfeld<feld
```

Im ersten Fall wird die »Antwort« 1, im zweiten Fall die Antwort 0 ausgegeben. Die 1 steht für »richtig«, die 0 für »falsch«. Im ersten Fall wird überprüft, ob die beiden Namen auf dasselbe Objekt zeigen. Der Operator `==` ist also die Frage nach der Gleichheit, ganz im Gegensatz zu dem Zuweisungsoperator `=`, der nichts mit einer Gleichheitsüberprüfung zu tun hat.

Übrigens ist jedes Objekt, dass leer oder die 0 ist, als »falsch« zu betrachten. Das werden wir noch sehen. Der Operator für »ungleich« lautet `!=`, für »grösser gleich« `>=`.

3.2 Enthaltensein

Die Anweisung

```
>>>range(3)
```

erzeugt eine Liste (!) mit drei Zahlobjekten als Eintrag und zwar von 0 bis 2. Diese Liste dient wunderbar als Zählliste für sogenannte *Schleifen*, wie wir noch sehen werden. Jetzt fragen wir danach, ob ein bestimmtes Objekt in dieser Liste enthalten ist.

```
>>>i=2
>>>a=range(3)
>>>i in a
```

Die Antwort ist offenbar richtig, denn das Objekt 2, auf das der Name `i` zeigt, ist in der Liste `a` enthalten. Nun etwas Merkwürdiges:

```
>>>a=[2,1]
>>>b=[2,1]
>>>a==b
>>>a is b
```

Sehr wohl ist `a` mit `b` identisch, aber es handelt sich *nicht* um dieselben Objekte. `==` prüft also auf gleichen Inhalt, aber `is` untersucht, ob die Objekte identisch sind. Das ist der feine Unterschied. Zu aller Verwirrung zeigt aber

```
>>>a=2
>>>b=2
>>>a is b
```

sehr wohl, dass in diesem Fall offenbar die Namen auf dasselbe Objekt zeigen. Aber das *muss* nicht so sein. Erst bei der Zuweisung `b=a` bedeuten beide Namen *wirklich* dasselbe Objekt.

3.3 Schleifen

Sobald ein Vorgang immer wiederholt werden soll, handelt es sich um eine Anweisung, die *besonders* für den Computer geeignet ist. Wir wollen uns die folgende Aufgabe stellen. Unser Objekt »feld« soll in jedem der Viererlisten `[0,0,0,0]` die Zahlenreihe `[1,2,3,4]` zu stehen bekommen. Da wir vier solcher Listen haben, muss *derselbe Vorgang* viermal durchgeführt werden.

Der ersten Zeile kann natürlich via `feld[0]=[1,2,3,4]` der entsprechende Inhalt zugewiesen werden. Damit können wir nun entsprechend für die anderen drei Zeilen fortfahren und eine entsprechende Anweisung formulieren. Das aber ist der schlechteste Programmierstil, den es überhaupt nur geben kann, da nun die eigentliche Arbeit vom Programmierer selbst durchgeführt wird. Daher, extrem wichtig:

Wenn ein analoger Vorgang mehrere Male wiederholt wird: Schleife, nichts sonst!

Es gibt bei Python *zwei* Möglichkeiten, eine Schleife zu realisieren:

3.3.1 Die endliche Schleife: For-Schleife

```
>>>for i in range(4):
>>>    feld[i]=[1,2,3,4]
```

Dies ist die eleganteste Lösung. Bedenken Sie, dass die Einrückung in der zweiten Zeile *extrem* wichtig ist. Auf diese Weise wird Python angezeigt,

was wiederholt werden soll. Es können auch *mehrere* Anweisungen sein, die dann mit derselben Einrückung untereinander geschrieben werden (bei IDLE erfolgt die Einrückung automatisch, aber nur dann, wenn Sie nicht den Doppelpunkt vergessen! Der ist *sehr* wichtig, denn er leitet einen sogenannten »Anweisungsblock« ein).

Schauen Sie sich danach das Objekt »feld« einmal an. Es ist alles so wie vorgesehen! Sie müssen im Interpreter allerdings erst einmal so lange RETURN eingeben, bis wieder der Python-Prompt »> > >« erscheint. Erst dann können Sie sich »feld« ansehen.

Hat sich auch »sfeld« geändert? Natürlich ist das der Fall, da beide Namen ja auf dasselbe Objekt zeigten. Eine *noch* elegantere Lösung wäre mit dem Code

```
>>>for i in range(4):
>>> for j in range(4):
...   feld[i][j]=j+1
```

möglich gewesen. »feld[2][1]« bedeutet zum Beispiel in dem 3. Element, der Liste *feld*, das ja *selbst* wieder eine Liste ist, auf die ja der Name *feld[2]* zeigt, ist das Objekt zu holen, auf das der Name *feld[2][1]* zeigt, und das ist natürlich das 2. Element des 3. Elementes von *feld*.

3.3.2 Die bedingte Schleife: While-Schleife

Eine andere, aber in diesem Falle etwas unübliche Möglichkeit wäre es gewesen, die Zuweisung in der folgenden Form durchzuführen:

```
>>>while i in range (4):
...   feld[i]=[1,2,3,4]
...   i=i+1
```

Warum funktioniert das Ganze so nicht? Ja, worauf zeigt denn der Name *i*? Während die »For« tatsächlich eine *Zuweisung* ist, bietet »while« lediglich nur einen Vergleich an. Im Gegensatz zu »for«, setzt »while« den Index »i« nicht automatisch bei jedem Durchlauf um »eins hoch«. Daher muss innerhalb der Schleife dieses zusätzlich erfolgen. Aber, bevor die Schleife beginnt, muss auch ein Startwert angegeben werden. Es fehlt daher vor dem Beginn

der Schleife die Zuweisung » $i=0$ «. Bitte bedenken Sie, dass die letzte Zeile » $i=i+1$ « auf jeden Fall zum Anweisungsblock gehören muss! Daher muss sie genauso eingerückt werden wie die vorletzte Zeile. Wenn Sie das nicht tun, wird i erst ausserhalb der Schleife verändert, was eine Endlosschleife zur Folge hat.

While-Schleifen sind wichtig, aber benutzen Sie die For-Schleife sooft es geht, denn diese kann niemals zu einer Endlosschleife werden. Die entsprechende »elegante« Form der Doppelschleife würde in diesem Fall so aussehen:

```
>>>i=0
>>>while i in range (4):
...     j=0
...     while j in range (4):
...         feld[i][j]=j+1
...         j=j+1
...     i=i+1
```

Achten Sie unbedingt auf die richtigen Einrückungen.

3.4 Ergänzungen zu Schleifen

Mit *break* können Sie eine Schleife beenden und durch die Anweisung *continue* springen Sie jeweils zum Schleifenkopf. Die Anweisung *pass* macht schliesslich gar nichts.

3.5 Die Verzweigung

Als dritte und letzte elementare Anweisung gibt es die logische Verzweigung, die, je nach Fall, das Programm auf die eine oder andere Weise weiterlaufen lässt. Wir wollen nun unser Objekt *feld* so verändern, dass an allen Stellen, an denen wir eine gerade Zahl finden, wieder eine 0 setzen, an den Stellen, wo eine 1 steht wird diese in die Zeichenkette 'eins' verändert, die 4 dagegen in die Zeichenkette 'vier'. Das sieht dann wie folgt aus:

```
>>>for i in range(4):
...     for j in range(4):
```

```
... if feld[i][j]%2==0:
...     feld[i][j]=0
... elif feld[i][j]==1:
...     feld[i][j]='eins'
... else:
...     feld[i][j]='vier'
```

Wie Sie sehen, funktioniert »if« genauso wie eine Schleife. Durch den Operator »:« wird ein *Anweisungsblock* eingeleitet. Das Wort »elif« bedeutet natürlich »wenn sonst« und ist eine Zusammenziehung aus »else« und »if«. Das reine »else« bedeutet »sonst, in allen anderen Fällen«.

3.6 Namen in Anweisungsblöcken

Wie wir gesehen haben, sind Schleifen und Verzweigungen identisch aufgebaut. Auf die Kopfzeile erfolgt ein Block eingerückter Anweisungen.

```
>>>Kopfzeile (Bedingung)
...     Anweisung1
...     Anweisung2
...     :
>>>Weitere Anweisung, die nicht mehr zum Block gehört
```

Namen, die *innerhalb* eines Anweisungsblockes eingeführt werden, leben nur so lange wie dieser Block abgearbeitet wird. Ein Anweisungsblock kann nur auf Namen zugreifen, die zu einem Anweisungsblock gehören, in den er eingebettet ist. Aus diesem Grunde war es möglich, in der inneren Schleife auch auf das »i« der umgebenden Schleife zuzugreifen. Ausserhalb des Anweisungsblocks existiert das »i« nicht mehr, oder es zeigt auf das Objekt, was *vor* dem Eintritt in die Schleife an das »i« gebunden war. Es ist also in einem Anweisungsblock möglich, *lokal* einen Namen für ein anderes Objekt zu verwenden, ohne *nach* Beendigung der Schleife die Anbindung an das alte Objekt verloren zu haben.

Kapitel 4

Funktionen

Zu diesem Kapitel siehe Abbildung 1.2.

Zu den Funktionen ist es nun nur noch ein denkbar kleiner Schritt. Wir haben eben gesehen was Anweisungsblöcke sind. Wir erinnern uns: Funktionen sind Objekte, die *funktionalen* Code enthalten. Wir haben uns also inzwischen von den Apfelsaftgläsern entfernt und denken bereits über einen Roboter nach, der uns diese Apfelsaftgläser in den Mund giessen kann.

Um nun eine solche Maschine zu bauen, müssen wir ihr lediglich einen Namen geben. Wir wollen nun eine solche Maschine mit dem Namen *startposition* verwenden, um unser *feld* wieder *überall* auf 0 zu setzen. Den Anweisungsblock in Form einer Doppelschleife können wir schon schreiben. Aber wie geben wir dem Ganzen einen Namen? Dieses geschieht mit dem Operator *def*, der nichts anderes als = durchführt. Der Unterschied besteht darin, dass = Namen von *Datenobjekten* zuweist, während *def* Namen von *Funktionen*, also Anweisungsblöcken zuweist.

```
>>>def start():
...     for i in range(4):
...         for j in range(4):
...             feld[i][j]=0
```

Nun können wir durch Eingabe von »start()« am Python-Prompt die gewünschte Aktion ausführen. Schauen wir hinterher nach, sehen wir, dass das *feld* wieder überall auf 0 gesetzt wurde.

Etwas merkwürdig erscheint sicherlich die Angabe der runden Klammern, die unbedingt erfolgen muss. Runde Klammern? Das hat doch etwas mit Tupeln zu tun? Richtig! Sehen wir uns an, wozu das Ganze gut ist. Wir wollen nun eine Funktion mit dem Namen *input* schreiben, die nun beliebig durch Vorgabe von vier Zahlen, die nun nicht unbedingt 1, 2, 3 und 4 sein müssen, bei Bedarf das *feld* wieder so belegen wie vorhin. Das Ganze sieht dann so aus:

```
>>>def input(a,b,c,d)
...   for i in range (4):
...     feld[i]=[a,b,c,d]
```

Der Aufruf erfolgt nun auf die Weise

```
>>>input(2,4,6,8)
```

Es werden jetzt demzufolge gewissermassen *von aussen* Werte an die Funktion übergeben. Die Funktion kann also wenigstens von der Aussenwelt Informationen empfangen. Was tatsächlich übergeben wird, ist ein Tupel! Das Tupel enthält die Zahlenwerte als Objekte, auf die nun intern die Namen a, b, c und d verweisen. Jetzt also ist die Funktion flexibel geworden. Sie reagiert auf die Aussenwelt. Ist sie eine »stumme« Maschine, wie zum Beispiel die Funktion *start*, so müssen dennoch die Klammern angegeben werden. Auch wenn sie leer sind. Bei Datenobjekten gibt es keine Empfangstupel, bei Funktionen dagegen immer.

Aber zur Kommunikation gehört natürlich auch die Notwendigkeit, dass die Funktion etwas an die Aussenwelt zurückgibt. Wir wollen nun unsere Funktion *input* so abändern, dass sie die mit der Zeilennummer multiplizierten Summen der letzten beiden Zeilen an die Aussenwelt wieder zurückgibt. Diese Rückgabe erfolgt mit dem Übergabeoperator *return*. Und zwar wie folgt:

```
>>>def input(a,b,c,d)
...   suma=sumb=0
...   for i in range (4):
...     feld[i]=[a,b,c,d]
...     if i==2:
...       for j in range(4):
...         suma=suma+i*feld[i][j]
```

```

...     elif i==3:
...         for j in range(4):
...             sumb=sumb+i*feld[i][j]
...     return (suma,sumb)

```

Tatsächlich gibt jetzt diese Funktion beim Aufruf

```
>>>input(1,2,3,4)
```

das Tupel (20,30) zurück, wie es sein soll. Dieses Objekt kann jetzt natürlich einem Namen wieder zugewiesen werden, so dass wir nach der Zuweisung

```
>>>ergebnis=input(1,2,3,4)
```

mit diesem Objekt »ergebnis« weiterarbeiten können.

Eine Funktion ist also quasi eine Maschine, in die wir etwas über ein Tupel hineintun können und aus der in Form irgendeines Objektes etwas herauskommt. Für das Einlesen muss immer ein Tupel herhalten. Für das Auslesen kann *irgendein* Objekttyp dienen.

Die einzelnen Argumente werden nach der *Stellung* ausgewertet, d.h. es ist klar, dass der Name *a* dem Objekt *1* zugewiesen wird. Sehr wohl kann die Zuweisung aber auch so erfolgen, dass ein Aufruf wie

```
>>>input(a=2, c=5, b=6, d=7)
```

erfolgt. In diesem Falle wird intern ein Dictionary erzeugt, in dem die *Schlüsselworte* a,b, c und d den Zahlobjekten zugewiesen werden.

4.1 Wertevorbelegung

Sehr wohl können einige der Parameter einer Funktion bereits mit Werten vorbelegt werden. Ein Beispiel:

```

>>>def func(monty, python, who='I am ', what='a lumberjack ', how='and I am OK
...     print monty+python+who+what+how

```

```
>>>func ('Michael', 'Palin: ')\n'Michael Palin: I am a lumberjack and I am OK'\n>>>func ('Michael', 'Palin: ', 'You are ')\n'Michael Palin: You are a lumberjack and I am OK'
```

Wie Sie sehen, ist es auf diese Weise möglich, Funktionen mit weniger Argumenten aufzurufen als wirklich vorhanden sind. Allerdings *müssen* zwei Argumente übergeben werden, weil dies der Anzahl der noch nicht zugewiesenen Namen entspricht.

4.2 Beliebige Anzahl von Argumenten

Ist bei einer Funktion nicht klar, wie viele Argumente sie besitzen muss, so wird durch die Angabe eines Parameters der Form **name* beim Aufruf ein beliebiges Tupel erstellt.

```
>>>def pruefe(*args):\n...     if 2 in args: print 'Yes'\n...     else: print 'No'\n... \n>>>pruefe (1,2,6)\n'Yes'
```

Bedenken Sie, dass bei einer Festlegung der Form *def name (a, *b)*: der erste Parameter ein beliebiges Objekt erwartet, aber der zweite Parameter **b* den ganzen Rest (!) als Tupel übergibt.

Wenn eine Funktion nicht durch Stellung der Argumente, sondern durch Schlüsselworte aufgerufen wird, so können überzählige Zuweisungen via ***name* in ein Dictionary übernommen werden. Zum Beispiel:

```
>>>def zeige(a,b,**c):\n...     print c\n... \n>>>zeige (a=0, b=1, d=7, e=8)\n{d:7, e:8}
```

4.3 Kommentare

Wie wir bereits sehen, verlieren wir bei etwas längerem Code schnell die Übersicht, was die Funktion macht. Daher sollten wir *unbedingt* den Funktionen einen Text zufügen, der nicht vom Interpreter verarbeitet wird. Solch ein Text wird durch ein #-Zeichen eingeleitet und darf sich nur bis zum Zeilenende erstrecken. Wenn mehrere Zeilen nötig sind, muss erneut das Kreuz gesetzt werden.

Eine gute Kommentierung beschreibt die einlesenden Namen und das was zurückgegeben wird und natürlich die Aufgabe der Funktion. Grundsätzlich sollte eine Funktion immer nur eine einzige Aufgabe ausführen und nicht mehrere auf einmal! Eine gute Kommentierung sieht damit so aus:

```
>>>def input(a,b,c,d)
... #Weist jeder Zeile die Werte a,b,c,d in dieser Reihenfolge zu.
... #Gibt die mit dem Zeilenindex multiplizierten Summen
... #der letzten beiden Zeilen zurück
...
... suma=sumb=0
... for i in range (4):
...     feld[i]=[a,b,c,d]
...     if i==2:
...         for j in range(4):
...             suma=suma+i*feld[i][j]
...     elif i==3:
...         def input(a,b,c,d)
...         #Weist jeder Zeile die Werte a,b,c,d in dieser Reihenfolge zu.
...         #Gibt die mit dem Zeilenindex multiplizierten Summen
...         #der letzten beiden Zeilen zurück
...
...         suma=sumb=0
...         for i in range (4):
...             feld[i]=[a,b,c,d]
...             if i==2:
...                 for j in range(4):
...                     suma=suma+i*feld[i][j]
...             elif i==3:
...                 for j in range(4):
...                     sumb=sumb+i*feld[i][j]
...         return (suma,sumb)
```

```
...     for j in range(4):
...         sumb=sumb+i*feld[i][j]
...     return (suma,sumb)
```

Natürlich hat am Eingabeprompt ein Kommentar keine Wirkung. Daher wird in naher Zukunft auch der Pythonprompt weggelassen, da der Code inzwischen so anwächst, dass Module schon fast zwingend sind. Im Alauf des Verstehens sind Module noch nicht an der Reihe, aber vielleicht lesen Sie doch schon einmal das sehr kurze Kapitel 6, um sich die doch allmählich aufwändige Arbeit am Eingabeprompt zu ersparen.

4.4 Einlesen von der Tastatur

Eine sehr wichtige Funktion ist *raw_input()*. Diese Funktion macht es möglich, während des Laufens eines Programms über die Tastatur einem Namen einen String zu übergeben. Probieren Sie aus:

```
>>>name=raw_input()
```

und geben Sie anschliessend »name« aus. Die Funktion *raw_input* liest vom Standardeingabekanal und weist dem bereitgestellten Namen den eingegebenen String zu. Dieser muss dann intern mit Hilfe von Stringoperationen in andere Objekte zerlegt werden. Das sehen wir im nächsten Kapitel.

4.5 Ausgabe

Sehr viel flexibler ist *print*, das im Gegensatz zu *raw_input()* eine Anweisung ist und daher nicht mit runden Klammern als Tupel arbeitet. *print* kann *jedes* Objekt ausgeben und eignet sich als Ausgabe innerhalb einer Funktion. Schreiben wir uns nun eine Funktion *output()*, die unser *feld* so ausgibt, dass alle Elemente *untereinander* stehen. Der zugehörige Code lautet:

```
>>>def output():
...     for in in range[i]:
...         print feld[i]
...     print '\n'
```


Als letzte Zeile wird mit dem Steuerstring `'\n'` eine Leerzeile ausgegeben. Sehen Sie nach, was passiert, wenn Sie `output()` aufrufen.

Wir haben lediglich deswegen noch nicht über *print* gesprochen, weil die Ausgabe durch Aufruf des Objektes gegeben war. Die Anweisung *print* erlaubt allerdings eine »formatierte« Ausgabe!

4.6 lambda-Ausdrücke

Wir denken uns eine Funktion *otto*, die nichts weiter tun soll, als eine Summe von drei eingegebenen Werten zu berechnen.

```
>>>def otto(x,y,z): return x+y+z
...
>>>otto(1,2,3)
6
```

So weit so gut. Die Funktion besteht nur aus einer einzigen Anweisung. Daher haben wir sowohl die *def*-Anweisung, als auch die Funktionsanweisung in eine Zeile geschrieben. Solche *einzeiligen* Funktionen können nun auch *ohne* Namensgebung erstellt werden, und zwar in der Form.

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

Damit lässt sich allerdings nichts mehr anfangen, da anschliessend auf das Objekt nicht mehr zugegriffen werden kann. Eine Namenszuweisung wie

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

ist natürlich möglich und der Aufruf

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

funktioniert genauso wie unser *otto*. Natürlich kann das Ganze auch mit vorbelegten Parametern ausgenutzt werden:

```
>>>x=(lambda x='arbeite ', y='und ', z='arbeite': x+y+z)
>>>x('bete ')
'bete und arbeite'
```

lambda-Ausdrücke erzeugen also ein anonymes Funktionsobjekt, dem zunächst kein Name zugewiesen wird. Der Vorteil ist der, dass solche Ausdrücke als *Argumente* an andere Funktionen weitergegeben werden können, was mit einer *def*-Anweisung nicht möglich ist. Wirklich merken müssen Sie sich allerdings, dass lambda-Ausdrücke *immer* nur aus einer Funktionsanweisung bestehen dürfen. Ein *if*-Verzweigung in einem lambda-Ausdruck ist also nicht möglich.

Kapitel 5

Klassen

Zu diesem Kapitel siehe Abbildung 1.3.

In diesem Abschnitt geben wir die Schreibweise mit dem Pythonprompt auf und setzen voraus, dass bereits Module geschrieben werden, die aber erst im folgenden Kapitel beschrieben werden, da dies in den Aufbau der Sprache besser passt. Dennoch können Sie im Vorgriff das kurze Kapitel über Module lesen.

Eine *Klasse* ist nun lediglich nichts anderes mehr als die Möglichkeit, einen neuen Objekttyp bereitzustellen, bei dem die zu bearbeitenden Datenobjekte mit den »richtigen« Funktionen gekoppelt werden. Ohne die Arbeit mit *Klassen* gibt es zwischen den Objekten ein heilloses Durcheinander! Wir werden nun unser *feld* von jetzt an als eine solche Klasse einführen und alles was wir bisher geschrieben haben, entsprechend umsetzen.

Zunächst wird mit dem Schlüsselwort *class* der Name der Klasse festgelegt und dem Klassenobjekt zugewiesen. Da wir später unser Objekt wieder *feld* nennen wollen, darf die Klasse selbst nicht *feld* heißen, denn dieser Klassenname steht ja sozusagen für einen Objekttyp. Setzen wir also an:

```
class kfeld:
```

Gut. Dann könnte man den Datensatz aus den ganzen Nullen einsetzen, der später bereitgestellt werden soll. Anstelle den Datensatz `[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]` jetzt einfach hinzuschreiben und ihn mit einem Namen zu versehen, gehen wir einen *sehr* viel besseren Weg. Wir erinnern uns daran, dass eine Namenszuweisung zu einem *Datenobjekt* mit dem Operator `=` erfolgt, eine entsprechende Zuweisung für eine *Funktion* dagegen mit dem Operator *def*. Wenn

wir den Operator `=` nun auch als Zuweisung für unsere gerade im Aufbau befindlichen neuen Hybridobjekte gut funktionieren soll, muss er sozusagen in seiner Wirkungsweise überschrieben werden¹.

Dieses Überschreiben erfolgt mit einer Funktion², oder *Methode*, wie wir in Zukunft sagen werden, die den Namen `__init__` tragen muss. Der Name wird mit *zwei* Unterstrichen begonnen und endet auch mit zwei Unterstrichen.

Es gibt nun noch eine letzte Sache zu beachten. Wenn wir später das Objekt *feld* vom Typ *kfeld* generieren wollen, so soll es den Datensatz *data*, der die schon bekannte verschachtelte Liste enthält, sowie alle bisher beschriebenen Funktionen *input*, *output* und *start* zur Verfügung stellen. Es soll natürlich aber auch möglich sein, noch weitere Objekte vom Type *kfeld* zu generieren, die ihr eigenes Innenleben haben. Daher wird sich im allgemeinen die Funktion *feld.output()* im Ergebnis von der Funktion *sfeld.output()* unterscheiden, wenn *sfeld* ein zweites *kfeld*-Objekt ist, das sein Eigenleben führt. Daher muss der *Name* des Objektes irgendwo im Objekt selbst gespeichert werden, damit keine Verwechslungen möglich sind. Jener Name, der *innerhalb* des Klassenobjektes auf den eigentlichen Namen des Klassenobjektes zeigt, heisst *self*.

Langer Rede kurzer Sinn, hier ist der Code:

```
class kfeld:
    #Erstellt ein 4x4 Spielfeld

    def __init__(self):
        self.data[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]

    def start(self):
        for i in range(4):
            for j in range(4):
                self.data[i][j]=0

    def input(self,a,b,c,d)
        #Weist jeder Zeile die Werte a,b,c,d in dieser Reihenfolge zu.
        #Gibt die mit dem Zeilenindex multiplizierten Summen
```

¹Obwohl es in sehr einfachen Fällen, wie diesen, auch ohne geht. Aber es ist gewiss nicht die reine Lehre!

²In Klassen werden Funktionen als *Methoden* bezeichnet.

```

#der letzten beiden Zeilen zurück

suma=sumb=0
for i in range (4):
    self.data[i]=[a,b,c,d]
    if i==2:
        for j in range(4):
            suma=suma+i*self.data[i][j]
    elif i==3:
        for j in range(4):
            sumb=sumb+i*self.data[i][j]
return (suma,sumb)

def output(self):
    for in in range[i]:
        print self.data[i]
    print '\n'

```

Es ist sehr wichtig, dass *jede* Funktion an der ersten Stelle den Parameter *self* enthält. Nun können wir nach Belieben Objekte vom Typ *kfeld* einrichten. Zum Beispiel richten wir mit

```
feld=kfeld()
```

ein solches Objekt ein, dass unser gutes altes *feld* ist. Wir können aber auch ein zweites Objekt dieser Art einrichten, das zum Beispiel *sfeld* heissen soll. Die Anweisung dafür lautet jetzt

```
sfeld=kfeld()
```

Bei diesen Aufrufen wird jeweils ein eigenes Objekt eingerichtet (eine sogenannte *Instanz* gebildet), das über den »Konstruktor« `__init__` erfolgt und ausserdem auch noch den jeweiligen Objektnamen dem Parameter *self* zuordnet. Beim Objekt *feld* zeigt der Parameter *self* also auf den Namen *feld*, beim anderen Objekt ist es der Name *sfeld*. Da jedes Objekt sein eigenes *self* besitzt, kann es dabei nicht zu Zweideutigkeiten kommen.

Der Aufruf der *input*-Methode erfolgt jetzt über

```
feld.input(1,2,3,4)
```

Der Name des Objektes wird also durch den »Selektor« eindeutig auf die richtige Funktion *input* bezogen, die eben zu *feld* gehört. Natürlich lautet der Code bei dem Objekt *sfeld* genauso. Aber es gibt selbstverständlich ganz andere Möglichkeiten, sich Klassen zusammenzubauen als es in *kfeld* geschehen ist. Und in diesen Klassen könnte ebenfalls eine Funktion mit dem Namen *input* existieren, die ganz anders arbeitet und womöglich auf den *kfeld* Objekten gar nicht funktioniert.

Mit

```
sfeld.input(2,2,2,2)
```

wird das Objekt *sfeld* auf eine ganz andere Weise verändert als *feld*. Folglich liefern *feld.output()* und *sfeld.output()* verschiedene Ergebnisse.

5.1 Attribute, Methoden, Instanzen, Klassen

Um den vielfältigen Begriffswirrwarr ein wenig zu ordnen, noch ein paar Erklärungen in einer Übersicht:

1. *Attribute* heissen die Datenobjekte einer Klasse oder einer Instanz.
2. *Methoden* heissen die Funktionen einer Instanz oder einer Klasse.
3. *Instanzen* sind Abbilder einer Klasse, also »echte« Objekte, die Attribute und Methoden enthalten. Sowohl Attribute als auch Methoden müssen *qualifiziert* aufgerufen werden, das heisst in der Form *instanzname.attributname* bzw. *instanzname.methodenname*.
4. *Klassen* sind quasi die Schablonen für Instanzen. Stellen Sie sich eine Klasse beispielsweise als einen Stempel vor, so sind Instanzen die mit Hilfe von Stempelfarbe erzeugten Abbilder des Stempels. Bedenken Sie aber, dass auch eine Klasse ein Objekt darstellt. Der Klassenname darf nicht mit den Namen irgendwelcher Instanzen verwechselt werden. Das heisst, es können auch die Klassen selbst mit ihrem Namen aufgerufen werden, wenn es denn notwendig ist.

Bei den Methoden wird zwischen *gebundenen* und *ungebundenen* unterschieden. Ein Beispiel:

```
class Spam:
    def drucke(self,message):
        print message
```

liefert unter dem Namen *Spam* eine Klasse, also den »Stempel«. Mit

```
object=Spam()
```

wird eine Instanz erzeugt. Die Methode

```
object.drucke('Hallo')
```

ist an diese Instanz gebunden. Obwohl die Druckmethode eigentlich zwei Parameter besitzt, muss nur *ein* Attribut übergeben werden, da *self* bereits belegt ist.

Es existiert aber wegen der vorangegangenen Definition der Klasse auch die Methode *Spam.drucke*. Bedenken Sie, dass hier beim qualifizierten Aufruf der Klassenname, und nicht der Instanzname, angegeben wird. Die Methode ist in dem Sinne *ungebunden*, weil noch nicht bekannt ist, auf welches Objekt sie wirkt. Daher muss bei einem Aufruf wie

```
Spam.drucke(object,'Hallo')
```

der *self*-Parameter durch den Namen einer existierenden Instanz belegt werden.

5.2 Vererbung

Und jetzt kommt wirklich ein Knaller. Auch ohne, dass wir in den Code von *kfeld* eingreifen müssen, können wir uns eine neue Klasse verschaffen, die nur an diejenigen Stellen anders ist, die man haben will und ansonsten *alles* von *kfeld* erbt. Nennen wir die neue Klasse *nfeld*, so wird die Vererbung via

```
class nfeld(kfeld):
```

angezeigt. Wird ein Objekt

```
neu=nfeld()
```

deklariert, enthält es sämtliche Datensätze, die auch *kfeld*-Objekte besitzen. Merhfachvererbung ist ebenfalls möglich, wenn mehrere Objekte durch Kommata getrennt in den runden Klammern erscheinen. Nun wollen wir etwas ändern. Die *nfeld*-Objekte sollen je nach Angabe statt Nullen, das Feld mit der vorgegebenen Zahl belegen. Wir müssen den Konstruktor `__init__()` daher *überschreiben*.

Das Ganze sieht jetzt so aus:

```
class nfeld(kfeld):

    def __init__(self,w):
        self.data=[[w,w,w,w],[w,w,w,w],[w,w,w,w],[w,w,w,w]]
```

Jetzt muss bei der Instanzbildung ein Wert übergeben werden, zum Beispiel:

```
neu=nfeld(2)
```

Der Zuweisungsoperator `=` weist, da `__init__` ja angibt, dass dies so zu tun ist (`__init__` »überlädt« ja den Operator `=`), den Namen *self* dem Objekt *neu* zu. Damit ist aber erst *ein* Parameter zugewiesen. `__init__` wartet noch auf einen zweiten. Dieser muss explizit in den runden Klammern angegeben werden. Sind (ausser *self*) noch weitere Parameter erforderlich, geschieht dies wie bei der gewöhnlichen Tupelübergabe von Funktionen.

So einfach ist das. Die Objekte kommunizieren jetzt nur noch über ihre Methoden miteinander. Auf gar keinen Fall sollte man direkt auf den Datensatz *feld.data* von aussen zugreifen. Es *ist* möglich, aber extrem schlechter Programmierstil.

5.3 Noch einmal: Dateien

Sie erinnern sich, dass wir die *Dateien* als grundlegende Datenobjekte bisher nur wenig angesprochen haben. Das hat seinen Grund, denn Dateien sind gerade solche Klassenobjekte, wie wir sie gerade behandelt haben. Durch

```
myfile=open('mydatei.txt','w')
```


wird auf der Festplatte eine Datei mit dem Namen *mydatei.txt* dem Namen *myfile* zugewiesen, und zwar so, dass in die Datei auf der Festplatte geschrieben werden kann; daher das *w* als zweiter Parameter. Für das »Nur-Lesen« muss ein *'r'* übergeben werden.

```
myfile.write('feld')
```

schreibt den String *'feld'* in die Datei. Zunächst einmal können wir nur Strings schreiben und lesen. Durch die entsprechende Methode *read* und *readline()* wird entsprechend aus einer Datei gelesen und ein String zurückgegeben.

Die Methode

```
myfile.close()
```

schliesst die Datei, indem die Verbindung des Namens »myfile« auf »mydatei« aufgehoben wird.

Damit haben wir die wesentlichen Eigenschaften von *Dateien* nachgeholt. *Listen* und *Strings* sind Objekttypen, die ebenfalls über ihre eigenen Methoden verfügen. So hängt beispielsweise die Methode *append()* an eine vorliegende Liste an, *reverse()* dreht die Reihenfolge um und so weiter. Die Liste, was man mit Strings und Listen machen kann, ist zu lang, so dass Sie an dieser Stelle auf eine Referenz verwiesen werden müssen.

5.4 Was ist OOP?

OOP ist nichts weiter als eine Abkürzung für *objektorientiertes Programmieren*, das heisst das »Programmieren mit Klassen«. Das Gegenteil davon heisst *prozedurales Programmieren*.

Warum reicht das gewöhnliche, prozedurale Programmieren nicht aus? Die Abbildungen 5.1 und 5.2 zeigen Ihnen den Unterschied. Beim prozeduralen Programmieren wuseln Anweisungen und Funktionen zwischen verschiedenen Datenobjekten hin und her wie Wasserläufer auf einem See zwischen den Orten, wo sich gerade eine Beute zeigt. Schon dieses kleine Bild zeigt, dass der Programmierer dabei sehr schnell den Überblick verlieren kann. Ausserdem ist nicht gewährleistet, dass nicht versehentlich die falsche Funktion für das falsche Objekt verwendet wird.

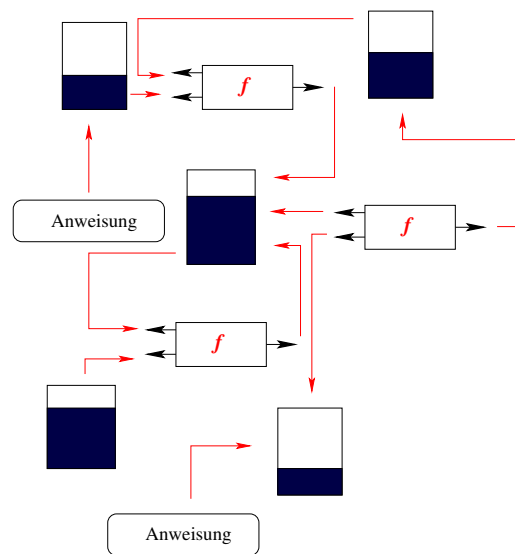


Abbildung 5.1: Prozedurales Programmieren

Beim objektorientierten Programmieren sind die Funktionen in einer Klasse eingesperrt und verrichten nur dort ihre Arbeit. Es ist durchaus Absicht, dass in der Abbildung keine Anweisungen innerhalb der Klasse eingezeichnet sind, denn *alles* sollte in einer Klasse als Funktion (Methode) vorliegen! Diese Methoden geben einem Klassenobjekt sozusagen ein Eigenleben. Zwischen den Klassenobjekten bewegen sich weder Anweisungen noch Funktionen, sondern es werden lediglich Daten als *Nachrichten* ausgetauscht. Die Klassenobjekte (Instanzen) kommunizieren über den *self*-Parameter miteinander. Dieser *self*-Parameter ist mit dem Namen der Instanz identisch³ und ist symbolisch in der Abbildung mit dem Rahmen der Klasse verbunden. Welche Inhalte der Instanz an dem Datenaustausch beteiligt sein können, zeigt Ihnen die Abbildung durch die gestrichelten Linien, die von den einzelnen Elementen zum Rahmen und damit zum *self*-Parameter führen. Fehlt diese Verbindung, so ist nimmt der entsprechende Datensatz oder die Methode an dem Nachrichtenaustausch nicht teil.

Ein weiterer unglaublicher Vorteil des OOP ist die Vererbung. Die Abbildung 5.3 zeigt sogar eine Mehrfachvererbung.

Nehmen Sie an, Sie wollen ein Programm verbessern und kennen die Art und

³Und nicht mit dem Namen der Klasse!!

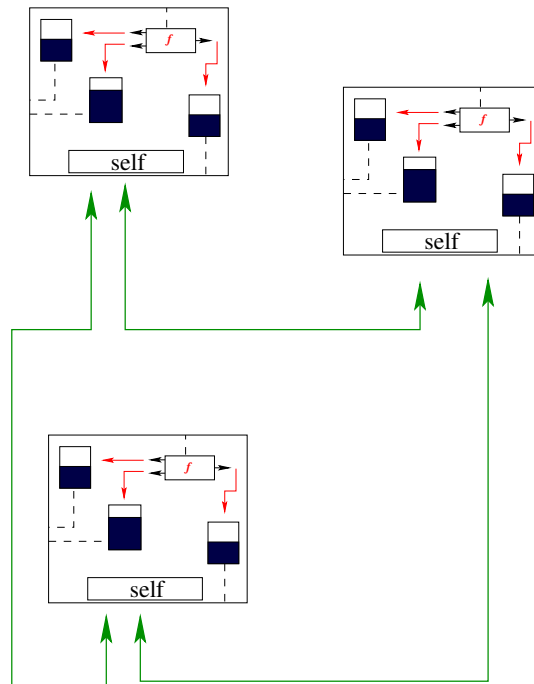


Abbildung 5.2: Objektorientiertes Programmieren

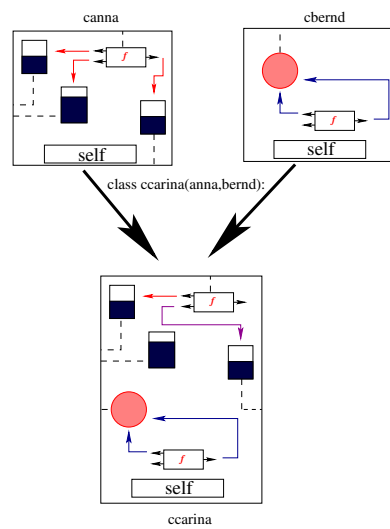


Abbildung 5.3: Mehrfachvererbung zwischen Klassen

Weise *wie* die Methoden der Klassen *canna* und *cbernd* arbeiten. Sie kennen aber *nicht* den Code! Sie sind dann in der Lage, sich eine neue Klasse *ccarina* zu erstellen, die zunächst *alle* Eigenschaften der Klassen *canna* und *cbernd* enthält. Die entsprechende Zuweisung `class ccarina(canna,cbernd):` legt diese Klasse durch Vererbung an.

Wenn Ihnen die Funktion *f* so wie sie in *canna* arbeitet, nicht gefällt, verschaffen Sie sich eine neue, in dem Sie lediglich den Code für die Funktion *f* in *ccarina* völlig neu schreiben. Die Abbildung zeigt aber auch, dass es bei Mehrfachvererbung zu Problemen kommen kann. Während der Funktionsname *f* in den Klassen *canna* und *cbernd* eindeutig war, ist dies nach der Vererbung in *ccarina* nicht mehr der Fall! Die von *cbernd* übernommene Funktion muss einen anderen Namen bekommen, und zwar schon in *cbernd*, da sonst in *ccarina* ein Namenskonflikt mit der aus *canna* übernommenen Funktion existiert. Bei Mehrfachvererbungen muss der Programmierer also höllisch aufpassen!

Neben der Überschreibung (=Überladung) von Methoden gibt es auch die Überladung von *Operatoren*, wie zum Beispiel *+* oder ***. Für diese Überladung schreiben Sie Funktionen mit speziell definierten Namen, die mit *zwei* Unterstrichen beginnen und mit zwei Unterstrichen aufhören. Für das Überladen von *+* heisst die zugehörige Funktion immer `__add__`, für das Überladen von *** dagegen `__mul__`. Sie erinnern sich sicherlich noch an `__init__`. Dies ist die Funktion, mit der sie den Objektkonstruktor überladen können. Wenn Sie es von aussen steuern wollen, *wie* die über *self* ansprechbaren Datensätze belegt werden sollen, benötigen Sie immer eine `__init__`-Methode.

Nun sieht es auf den ersten Blick so aus, als ob OOP das Allheilmittel sei. Was das *grafische* Programmieren betrifft, so mag dies auch richtig sein. Aber für ein Erstellen einfacher Algorithmen, die in erster Linie mathematische Probleme lösen sollen, ist OOP in der Regel eine Kanone, die auf Spatzen gerichtet ist. In diesem Fall ist prozedurales Programmieren vorzuziehen. Es ist das Schöne, dass man in Python nicht zu OOP gezwungen wird (wie zum Beispiel bei Java), sondern sehr wohl auch prozedural programmieren kann. Man sollte es aber auf jeden Fall vermeiden, eine Mischung beider Programmierarten zu akzeptieren!

OOP erfordert ein sehr viel sorgfältigeres Überlegen und Designen, *bevor* die erste Programmierzeile geschrieben wird.

Kapitel 6

Module

Zu diesem Kapitel siehe die Abbildungen 1.5 und 1.6.

Module sind nichts weiter als Container für Namen. In der Regel sind diese Namen ausschliesslich *Klassen* zugeordnet. Schreiben Sie beliebigen Code (vor allen Dingen Klassen!) immer in ein Modul. Ein Modul ist nun eine Datei, die sie über einen Editor auf der Festplatte ablegen. Diese Datei muss in jedem Falle die Endung *.py* besitzen!

Nennen wir unser Modul, in das wir den im vorigen Kapitel erstellten Klassentext einfügen, einfach *modul1.py*. Wie gesagt, das Modul ist ein *Namensraum*. Wenn jetzt unsere Objektklasse *kfeld* dem Interpreter verfügbar gemacht werden soll, muss das Modul *modul1* importiert werden, und zwar »ohne« die Erweiterung *.py*! Dies geschieht durch den Aufruf

```
import modul1
```

Jetzt stehen *alle* Namen des Moduls, so zur Verfügung, dass sie ebenfalls über den Punktselektor angesprochen werden müssen. So heisst unser Klassentyp zunächst *modul1.kfeld*. Wenn Sie dies verhindern wollen, können Sie durch die *from*-Anweisung auch nur bestimmte Namen des Moduls holen, die Sie einzeln auflisten. Wenn Sie *alle* Namen ohne Punktselektor verfügbar machen wollen, geben Sie lediglich

```
from modul1 import *
```

ein, und schon können wir bei der Verwendung des Namens *kfeld* den Vorsatz *modul1* fortlassen.

Wenn Sie nun ein Modul geändert haben, so nützt ein neuerlicher *import* nichts, es sei denn, es wurden neue Namen hinzugefügt. Daher muss mit

```
reload(modul1) #Bedenken Sie, dass das eine Funktion ist!
```

das Modul als Kopie nachgeladen werden. Anschliessend ist bei Bedarf wieder ein *from* notwendig.

Etwas aufwändig, aber man kann sich daran gewöhnen.

Es gibt für Python eine Unzahl an Modulen für *jede* Art von Programmierung, ob es Netzwerke sind, ob es Mathematik ist, Datenbankprogrammierung, und, und. Das mit Abstand wichtigste Modul ist für uns *Tkinter*, das eine Anbindung der extrem einfachen grafischen Bibliothek *Tk* ermöglicht. Dieses wird ausführlicher im zweiten Band von »Programming Python« beschrieben.

Kapitel 7

Ausnahmen

Ausnahmen dienen dazu, Unsinn abzufangen. Schauen Sie sich das folgende Beispiel einmal an:

```
def kaboom(list,n):  
    print list[n]
```

Die Anweisung

```
kaboom([1,2,3],3)
```

ergibt einen Fehler, da das dritte Element ja nicht existiert (die Listenindices beginnen bei Null!!). Um auf diese Weise den Code nicht zum Absturz zu bringen, wird dieser Fehler durch eine Ausnahmebehandlung abgefangen, die wie folgt aussieht:

```
try:  
    kaboom([1,2,3],3)  
except IndexError:  
    print 'Das ging daneben'
```

Der von dem Standard-Fehlerkanal des Rechners zurückgegebene Wert muss in einem Objekt aufgefangen werden, das hie *IndexError* heisst. Falls dieser »richtig« ist, also sich von Null unterscheidet, wird der Code ausgeführt, der in dem *except*-Anweisungsblock steht. Wenn das Ganze für Sie wie eine Fallunterscheidung via *if* aussieht, so haben Sie recht. Genau das ist so. Aber *if* kann keine Fehler abfangen, die Systemmeldungen sind. Daher muss die Angelegenheit mit *try* und *except* erfolgen.

Anhang A

Wie werden Programme erstellt?

Eine Programmiersprache ist nichts anderes als ein Bindeglied zwischen dem Programmierer und dem Prozessor (CPU¹). Der Rechner selbst versteht nur eine sogenannte *Maschinensprache*, die aber für Menschen sehr schwierig zu handhaben ist. Es ist zwar für den Anfang nicht unbedingt notwendig, dass Sie die genaueren Details verstehen, aber da der Abschnitt A.1 recht kurz ist, empfehle ich Ihnen, diesen wenigstens kurz zu überfliegen.

Ein Programm, das in einer bestimmten *Programmiersprache* verfasst ist, kann demnach nicht unmittelbar vom Prozessor verarbeitet werden. Das Programm muss daher entweder von einem *Compiler* oder einem *Interpreter* übersetzt werden. Was das ist, zeigt Ihnen der Abschnitt A.2.

Die Programmiersprache PYTHON ist ein Mittelding zwischen einem Interpreter und einem Compiler; dem Programmierer erscheint sie allerdings vorwiegend als Interpreter mit all seinen Vorteilen. Der Abschnitt B.1 zeigt den Umgang mit dem eigentlichen Python-Interpreter.

Der Interpreter selbst besitzt allerdings keinen *Editor*, um Programmtext zu schreiben. Jede eingegebene Zeile wird sofort ausgeführt; sie ist beim Verlassen des Interpreters verloren. Das Programm IDLE fasst *Interpreter*, *Editor* und einen sogenannten *Debugger* zu einer Einheit zusammen. Solch eine Einheit heisst *Integrierte Entwicklungsumgebung*. Von IDLE handelt der letzte Teil B.2 dieses Kapitels.

¹Diese Abkürzung bedeutet: *central processing unit* und ist die Bezeichnung für den Mikroprozessor des Rechners.

A.1 Über Bits, Bytes und Assembler

Der Prozessor eines Computers versteht nur *Maschinensprache*. Diese Sprache besteht aus einzelnen Zeichen und Befehlen. Ein einzelnes Zeichen ist ein *Byte*. Ein solches *Byte* stellt auch auf den Datenträgern die kleinste adressierbare Einheit dar: eine Speicherzelle.

Das Byte selbst besteht wieder aus acht »Unterteilungen«, die einzeln nicht adressierbar sind. Diese Unterteilungen heissen *Bits*. Ein einzelnes Bit kann nur zwei Zustände einnehmen: 0 und 1, was zugleich »Strom aus« und »Strom an« entspricht. Jedes Speichermedium arbeitet allerdings auf magnetischer Basis, so dass Sie sich zum Beispiel eine Festplatte mit kleinsten Magneten vollgefüllt vorstellen sollte, wobei die Magnete in ihrer kleineren Umgebung quasi parallel liegen und jeder dieser Magnete nur die beiden Zustände »Nordpol oben« und »Nordpol unten« kennt. Jeweils acht Magnete sind zu einer Einheit zusammengefasst, die eine Speicherstelle darstellt: ein »Byte«.

Ein solches Byte kann zum Beispiel die Darstellung

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline & & 0 & & 0 & & & \\ \hline \end{array} \quad (\text{A.1})$$

besitzen. Welche Information steckt nun in einem solchen Byte? Dafür gibt es im wesentlichen drei Möglichkeiten. Dieser Code bedeutet entweder

1. eine Zahl oder
2. ein Zeichen oder
3. einen Befehl

A.1.1 Zahlencode

Falls es sich um eine Zahl handelt, kann diese schnell entschlüsselt werden, wenn Sie das »Zweiersystem« (Binärsystem) beherrschen. Die dargestellte Zahl lässt sich dann durch

$$1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 128 + 64 + 16 + 4 + 2 + 1 = 215 \quad (\text{A.2})$$

entschlüsseln. Wie Sie sich leicht überlegen können, lassen sich mit einem Byte nur 256 verschiedene Zahlen darstellen: von 0 bis 255 oder auch von

−127 bis +128. Was ist nun los, wenn die Zahl 300 dargestellt werden soll? Nun, dann sind für diese *eine* Zahl *zwei* Bytes notwendig. Eine Zusammenfassung von zwei Bytes ergibt ein sogenanntes *Wort*. Damit lassen sich immerhin $256 \cdot 256 = 65536$ verschiedene Zahlen darstellen. Wenn dieser Bereich überschritten werden soll, müssen entsprechend mehr Bytes herhalten. Je höher die Zahl, desto grösser der Speicherplatz. Für Gleitkommazahlen sind entsprechend viele Bytes notwendig, um auch noch das Komma und eine Zehnerpotenz als Stellenwertanzeige abzuspeichern.

A.1.2 Zeichencode

Falls es sich um ein Zeichen handelt, gehört zu dieser Zahl 215 ein ganz bestimmtes Zeichen der Tastatur. Die Zuordnung des Zahlenschlüssels zu dem Zeichen ist durch eine internationale Konvention, eine Tabelle, gegeben, die ASCII² heisst. Wenn über die Tastatur ein solches Zeichen eingelesen werden soll, wird intern über die ASCII Tabelle in den Binärcode für den Rechner umgewandelt. Falls sich dieses Zeichen nicht auf der Tastatur befindet, kann dieses (in vielen Fällen) durch Festhalten der ALT-Taste und Eingabe des Zahlencodes erzeugt werden. Aber nicht alle Editoren unterstützen diese Funktion. Wie wir eben gesehen haben, kann die ASCII-Tabelle nur 256 Zuordnungen für Zeichen vornehmen. Jedem Tastendruck entspricht schliesslich genau *ein* Byte. Dies bedeutet natürlich, dass mit einer solchen Zuordnung zwar die westlichen Alphabete und etliche Sonderzeichen abgedeckt werden, keineswegs aber auch noch das kyrillische Alphabet oder gar die chinesischen Zeichen. Wenn solche Zeichen benötigt werden, war bisher der Einsatz von besonderen Codetabellen notwendig, die eben kein ASCII sind und daher auch nicht von jedem Rechner, der eine solche Datei verarbeiten soll, gelesen werden kann. Daher wurde der Entschluss gefasst, eine umfassendere Standardtabelle zu schaffen, die eine Zuordnung von *Worten* (zwei Bytes) zu einem Zahlencode ermöglicht. Dieser *Unicode* ist bereits in Gebrauch und umfasst wie wir gesehen haben 65536 verschiedene Zeichen.

A.1.3 Befehlscode

Im Falle eines Befehls wird ein entsprechendes Kommando ausgeführt, dass diesem Zahlencode eine bestimmte Anweisung zuordnet. Welche Zuordnung dies ist, hängt vom Typ des Prozessors ab. So sieht zum Beispiel ein in

²American Standard Code for International Interchange

Maschinensprache geschriebenes Programm für einen Apple-Rechner *ganz* anders aus als ein entsprechendes Programm für einen PC-Prozessor. In den meisten Programmen werden auf eine ganz ähnliche Art und Weise Anweisungen an eine Taste gebunden, indem vorher die STRG-Taste³ gedrückt und festgehalten wird und danach eine Taste der Tastatur betätigt wird. Dabei entsteht intern eine Umwandlung in eine Maschinenanweisung für den Prozessor.

A.1.4 Zusammenfassung

Noh einmal im Zusammenhang:

- Jede Maschinenanweisung besteht aus einem Binärcode, bei dem ein einzelnes Zeichen ein Byte darstellt, das wiederum aus acht Bits besteht, die lediglich den Inhalt 0 oder 1 haben können.
- Für Zahlen ergibt sich die Zuordnung sich aus der Darstellung durch das »Zweiersystem«.
- Für Zeichen ergibt sich die Zuordnung durch die ASCII-Tabelle, die Unicode-Tabelle oder einer entsprechenden Codetabelle für das jeweilige Land.
- Die Zuordnung zu Befehlen hängt von den Vorgaben des Prozessors ab.

A.1.5 Programmieren auf unterster Ebene

Wer demnach ein Programm im Binärcode schreibt, erstellt ein Produkt, das auf dem Rechner sofort lauffähig ist. Der Vorteil besteht darin, dass das Programm sehr schnell läuft. Tatsächlich wurde früher auch so mit Hilfe von Lochstreifen und Lochkarten so programmiert. Die Nachteile allerdings überwiegen:

- Der Programmierer bewegt sich in einer sehr schwer zugänglichen Binärsprache. Umfangreichere Programme sind auf diese Weise kaum erstellbar, geschweige denn zu warten.

³Steuerungstaste, auf amerikanischen Tastaturen heisst diese Taste auch CTRL (Control).

- Das Programm ist trotz dieses sehr hohen Aufwands hardwareabhängig, das heisst, es läuft *nur* auf dem Prozessor, der in dem Rechner eingebaut ist. Für einen anderen Prozessor muss das Programm völlig umgeschrieben werden!

Immerhin kann der Programmierer allerdings *alles* programmieren, was der Prozessor beherrscht. Um diesen Vorteil wenigstens für zeitkritische Module nutzen zu können, galt es nun, den Binärcode für den Programmierer zugänglicher zu machen. Ein erster, kleinerer Schritt in diese Richtung bestand darin, statt der Schreibweise im Binärsystem, den Inhalt eines Bytes in dem »Achtersystem« (Oktalsystem) darzustellen. Der Binärcode 215 lautet in diesem System

$$3 \cdot 8^2 + 2 \cdot 8^1 + 7 = 0327 \quad (\text{A.3})$$

Eine *Oktalzahl* erkennen Sie immer an der führenden Null, die zu Beginn des eigentlichen Oktalcodes stehen muss.

Im Rahmen der Verkürzung wurde noch ein weitere Schritt vorgenommen und das »Sechzehnersystem« (*Hexagesimalsystem*) bemüht. Dieses ist insofern besser, als dass *alle* Bytes zweistellig geschrieben werden können. Die hinteren vier Bits eines Bytes heissen *low bits* und gehören zur Potenz 16^0 , die vorderen vier Bits sind die *high bits* und gehören zur Potenz 16^1 . Bedenken Sie, dass bei diesem System ausser den Ziffern 0 bis 9 auch noch die Ziffern *A* bis *F* notwendig sind um »zehn« bis »fünfzehn« darzustellen, die in diesem System ja durch eine *einstellige* Zahl beschrieben werden müssen, da 10 in diesem System bereits »sechzehn« bedeutet. Die Darstellung der 215 lautet in diesem System

$$13 \cdot 16^1 + 7 \cdot 16^0 = 0xD7 \quad (\text{A.4})$$

Ähnlich wie eine Oktalzahl steht zu Beginn die Kennung des Systems, und zwar *0x*. Erst dann erscheint der Code.

Sowohl das Oktalsystem als auch das Hexagesimalsystem sind bereits »Sprachen«, die vor der Verarbeitung durch den Prozessor in den Binärcode transformiert werden müssen. Tatsächlich sind diese Darstellungen noch keine wirklichen Sprachen, da sie nur Abkürzungen für *einen* bestimmten Binär-code bedeuten.

A.1.6 Was ist ein Assembler?

Der nächste Schritt bestand darin, als Abkürzungen für die Inhalte von Bytes kurze Namen aus drei Buchstaben zu verwenden, die unmittelbar die

Aufgabe des Befehls darstellen sollten. So bedeutet etwa *mov ax, 3* für den PC-Prozessor die Anweisung »bewege die Zahl 3 in den Register *ax* des Prozessors«. Bedenken Sie, dass sowohl *mov* als auch *ax* und 3 je ein Byte bedeuten. Wenn Sie bereits schon programmiert haben, werden Sie sehen, dass diese Anweisung schon einige Elemente einer »echten« Programmiersprache enthält. Es ist allerdings wenig mehr als eine »mnemotechnische« Sprache, bei der die einzelnen vorhandenen Befehle des Prozessors durch *ein* Wort, wie hier zum Beispiel, *mov* ersetzt werden. Jeder einzelne Befehl führt *genau* eine Aufgabe aus. Um solch eine komplizierte Prozedur wie etwa das Ausgeben auf einem Bildschirm zu ermöglichen, müssen diese Befehle zu einem noch grösseren Block »zusammengeklebt« werden.

Aus diesem Grunde heisst diese mnemotechnische Sprache, die also die Binärbefehle nur durch Worte ersetzt, *Assembler*, also »Zusammenbauer«. Natürlich können Sie sich denken, dass Programme, die die Aufgabe eines einzigen *print* Befehls in einer höheren Programmiersprache durchführen, sehr umfangreich sind und natürlich vor der Ausführung *interpretiert* werden müssen, denn der Prozessor versteht eben nichts als reine Maschinsprache. Dieser Interpreter heisst ebenfalls *Assembler*. Nicht nur die Sprache, sondern auch das »Übersetzerprogramm« trägt denselben Namen. Ja, und in welcher Sprache ist nun der Assembler selbst geschrieben? Nun, dass *musste* zunächst Maschinsprache sein; daran geht kein Weg vorbei.

A.2 Die echten Programmiersprachen

Es lohnt sich heute auf gar keinen Fall mehr, Assembler zu lernen. Das werden wir gleich sehen! Der grösste Nachteil des Assemblers besteht darin, *maschinenabhängigen* (hardwareabhängigen) Code zu erstellen. Wer also sein für einen Macintosh geschriebenen Code auf einem PC verwenden will, kann das ganze Programm noch einmal neu verfassen. Assemblercode ist *nicht portierbar*. Die Entwickler machten aus der Not eine Tugend und begannen die Programmierung von der Hardwareebene loszukoppeln und zwar in folgender Weise:

- Es wäre schön, wenn den Befehlsblöcken, die durch einen Assembler erstellt wurden, eigene Namen gegeben werden, die dann als ein *einzig*er Befehl zur Verfügung stehen (denken Sie an unser *print*-Beispiel des letzten Abschnitts).

- Dieser »Quellcode« (Source) muss dann von einem hardwarespezifischen Übersetzerprogramm in Maschinensprache übertragen werden.

Was ist damit gewonnen? Nun, die im ersten Punkt genannte »Programmiersprache« ist auf *allen* Rechnern dieselbe. Bedenken Sie, dass das für den Assembler nicht gilt. Ein Befehl wie *mov* heisst auf einem anderen Rechner unter Umständen anders. Aber der Befehl *print* wie er zum Beispiel in der Programmiersprache »C« festgelegt ist, ist eben *in* der Sprache selbst festgelegt. Das Übersetzerprogramm muss sich nun darum kümmern, wie dieser Befehl auf dem spezifischen Rechner in Maschinensprache übertragen wird. Die Hardwareabhängigkeit ist also jetzt von der Sprache losgekoppelt und lediglich auf das Übersetzerprogramm übertragen worden.

Noch einmal: der Quellcode, die sogenannte *Source*, ist eine ASCII-Datei, die auf jedem Rechner von jedem Editor gelesen und bearbeitet werden kann. Dieser Quellcode ist also von der Hardware nicht abhängig. Das Übersetzerprogramm wandelt diesen Code in einen hardwareabhängigen Binärcode um, der dann vom Rechner verarbeitet werden kann. Auf diese Art und Weise kann der Programmierer sein Programm auf jeden Rechner portieren, wenn dort ein entsprechendes Übersetzerprogramm existiert⁴.

Die Abbildung A.1 verdeutlicht das bisher Gesagte noch einmal grafisch.

A.2.1 Wie werden Übersetzer entwickelt?

Es sieht so aus, als ob auf diese Weise die eigentliche Programmierung auf diejenigen armen Entwickler übertragen wird, die sich mit der Erstellung eines entsprechenden Übersetzerprogramms herumschlagen müssen. Also müssen diejenigen Entwickler doch Assembler beherrschen? Nein. Die Entwicklung geht wie folgt:

1. Der allererste Übersetzer einer älteren Programmiersprache wurde natürlich in Assembler irgendwann einmal geschrieben. Eine derartige ältere Programmiersprache ist zum Beispiel COBOL⁵, aber auch C⁶.

⁴Und für dieses Übersetzerprogramm bezahlen Sie, falls es kostenpflichtig ist, das Geld. Nicht für die Sprache.

⁵common business oriented language

⁶Diese Sprache wurde von Brian KERNIGHAN und Dennis RITCHIE in den 70-er Jahren für das Betriebssystem UNIX geschaffen und zwar als sogenannter Assemblerersatz.

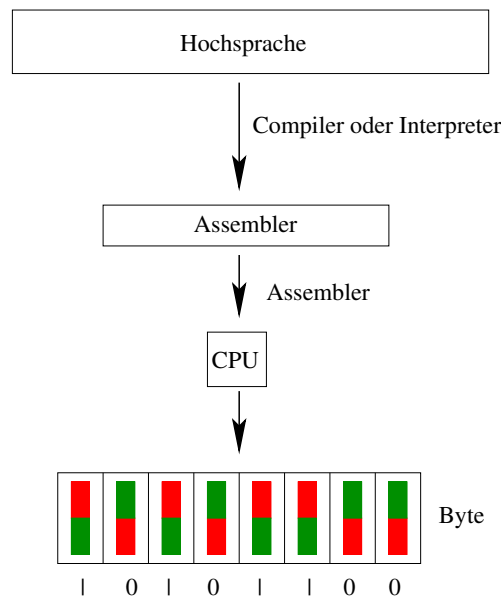


Abbildung A.1: Hierarchie von Programmiersprachen

2. Nehmen wir an, ein solcher Urübersetzer wurde aus einer Assemblerdatei erstellt, um C-Programme zu übersetzen. Geben wir diesem Ur-Übersetzer einen Namen: seine Assemblerdatei heiße *urc.asm*, die durch den Assembler in das endgültige Übersetzerprogramm *urc.exe* übertragen wird. Mit diesem Programm können nun C-Quelltexte übersetzt werden (Übersetzer der ersten Generation).
3. Jetzt schreibt der Entwickler einen neuen, sehr viel leistungsfähigeren Übersetzer (zweite Generation) in *reinem* C. Nennen wir die Quelldatei ruhig *neuc.c*⁷.
4. Nun wird die Quelldatei *neuc.c* von dem *alten* Übersetzer *urc.exe* in ein lauffähiges Programm übertragen. Nennen wir die entstehende Binärdatei ruhig *neucVorlaeufig.exe*.
5. Diese erste Version *neucVorlaeufig.exe* des neuen Übersetzers übersetzt nun seinen *eigenen* Quellcode (*Selbstübersetzung*) und erstellt auf diese Weise die viel leistungsfähigere Datei *neuc.exe*, die die endgültige

⁷Üblicherweise werden Quellcode-Dateien, die in C geschrieben wurden, mit der Endung *«.c«* versehen. Für C++ werden Endungen wie *«.C«* benutzt.

Version des neuen Übersetzers darstellt.

Selbstübersetzung hiess also der Trick. Aber irgendwann hat ja einmal alles mit Assembler angefangen. Was nun, wenn Sie auf die Idee kommen, eine *neue* Programmiersprache zu entwickeln; eine Programmiersprache mit dem Namen »EvB«, wozu auch immer diese gut sein soll. Jetzt brauchen Sie anscheinend für den *ersten* Übersetzer eines EvB-Programms Assemblerkenntnisse, oder?

Nein.

1. Schreiben Sie einfach Ihren Übersetzer *urevb* in der Programmiersprache C, die »viel« einfacher als Assembler ist. Dafür haben Sie ja mit *neuc* bereits einen Übersetzer.
2. Lassen Sie Ihre Datei *urevb.c* dann zunächst von *neuc* in eine vorläufige Version *urevb.exe* übersetzen.
3. Schreiben Sie dann die Datei *urevb* neu und zwar jetzt in der Programmiersprache EvB. Nennen wir diese Datei *evb.eur*⁸.
4. Jetzt lassen Sie *evb.eur* von *urevb* in die Datei *evbVorlaeufig.exe* übersetzen. Damit hat *urevb* seine Aufgabe erledigt und kann entsorgt werden.
5. Der im Vergleich zu *urevb* viel leistungsfähigere Übersetzer *evbVorlaeufig* übersetzt jetzt seinen *eigenen* Quellcode *evb.eur* ein weiteres Mal und der Übersetzer *evb.exe* ist fertig.

Sie haben Assembler nicht gebraucht, sondern lediglich die Kenntnis einer bereits bestehenden Sprache. Für alle Folgeversionen des Übersetzers *evb* brauchen Sie nur einmal einen Quellcode in reinem EvB schreiben. Für die erste Übersetzung können Sie auf eine bereits bestehende Version zurückgreifen. Von dieser lassen Sie Ihre neue Version in eine vorläufige Binärdatei übersetzen, die dann anschliessend ihren eigenen Quelltext noch einmal übersetzt. So geht es.

⁸Als Kennung für EvB-Code wurde *eur* vereinbart, damit EvB erkannt wird ;-).

A.2.2 Was ist ein Compiler? Compilersprachen

Der Übersetzer des vorigen Abschnitts, also ein Programm, dass aus einer Quelldatei eine lauffähige Binärdatei erstellt, heisst *Compiler*. Ein Compiler verlangt also, dass der Programmtext bereits fertig erstellt ist. Ist der Text syntaktisch fehlerfrei, wird alles auf einmal übersetzt. Der Übersetzungsvorgang läuft dabei in zwei Schritten ab.

Der eigentliche *Compiler* hat nur die Aufgabe, aus dem Quelltext eine *Objektdatei* zu erstellen. Diese Datei ist zwar Binärcode, aber im allgemeinen noch nicht lauffähig. Jeder Programmierer lagert häufig verwendete Programmblöcke in sogenannte Bibliotheken aus. So sind beispielsweise Grafikbefehle in der Regel in Bibliotheken zusammengefasst. Dies sind nichts anderes als Dateien, in denen diese Programmblöcke gesammelt vorliegen und gegebenenfalls vorübersetzt sind. Nachdem der *Compiler* seine Aufgabe erfüllt hat, setzt im zweiten Schritt der *Linker* die Verbindungen zu den benötigten Bibliotheken und fügt für jeden Programmblock, der in der Objektdatei benötigt wird, den entsprechenden Code aus der Bibliothek ein.

Heutzutage werden *Compiler* und *Linker* als eine Einheit betrachtet. Der Linker wird automatisch nach dem Compiler aufgerufen. Früher musste dies jeweils separat geschehen. Das Ergebnis des gesamten Übersetzungsvorganges ist eine ausführbare Binärdatei, die oft den Anhang *.exe* trägt⁹.

Und dann kommt das grosse Abenteuer. Läuft das Programm so wie es soll? Der Compiler entdeckt syntaktische Fehler, der Linker findet heraus, ob im Programm Befehle auftauchen, die in den einzubindenden Bibliotheken nicht existieren. Was nicht erkannt wird, sind logische Fehler. Meistens macht das Programm nicht sofort was es soll. Dann folgt die mühselige Fehlersuche. Oft gibt es dafür ein Extraprogramm, den *Debugger*¹⁰.

Compiler erzeugen schnellen Code, aber der Programmierer kann die Funktionsfähigkeit von Teilcode nicht ohne eine Gesamtübersetzung testen. Compilersprachen sind zum Beispiel C, C++, Pascal und damit auch das Object Pascal, das Delphi benutzt, sowie Java, wenn eine Applikation erstellt werden soll.

⁹Was aber zum Beispiel unter Linux nicht notwendig ist, da die entstehende Datei automatisch als ausführbar erkannt wird.

¹⁰Wörtlich »Entwanzer«.

A.2.3 Was ist ein Interpreter? Interpretersprachen

Ein *Interpreter* ist ein Programm, dass je nach Anforderung immer eine ganz bestimmte Zeile des Quelltextes übersetzt, aber niemals den gesamten Code. Auf diese Weise entsteht keine lauffähige Datei und es muss immer zuerst der Interpreter gestartet werden, bevor das eigentliche Programm geladen wird. Natürlich ist diese ältere Art der Übersetzung wesentlich langsamer als das Ergebnis eines Compilers.

Beachten Sie aber die Vorteile: ein Interpreter erlaubt es dem Programmierer, Teile seines Codes direkt am sogenannten Eingabeprompt zu testen. Dabei können Fehler sofort erkannt und beseitigt werden. Mit einem Interpreter kann so auf eine einfachere Weise eine Programmiersprache erarbeitet werden. Allerdings sollte dadurch nicht das unüberlegte Einhacken von Spaghetticode gefördert werden, wie es für so manche BASIC-Programmierer üblich ist. Und das ist, wenigstens in der Urfassung, bereits das Beispiel einer Interpretersprache. Wenn auch keiner sehr empfehlenswerter.

A.2.4 Wie arbeitet Python

Python ist nun in erster Linie ein Interpreter. Für das Starten von Programmen, muss in der Regel zunächst erst einmal der Interpreter gestartet werden. Doch halt! So ganz richtig ist das nicht. Wenn ein echtes Python-Programm, ein sogenanntes *Modul*, erstellt wurde, übersetzt Python dieses Programm in eine Art »Zwischencode« und umgeht dabei weitgehend den Geschwindigkeitsnachteil so mancher Interpretersprache. Dieser Zwischencode scheint wie eine reguläre Binärdatei eines Compilers lauffähig zu sein. In Wirklichkeit startet sie bei Eingabe des Namens sofort den Interpreter. Dazu gleich mehr.

Python arbeitet in punkto Zwischencode ähnlich wie Java bei der Erstellung von Applets (»nicht« Applikationen, wie eben beschrieben). Applets sind vorübersetzter Javacode, der in Webseiten eingebunden werden kann, und dann von einem Java-fähigen Browser interpretiert wird. Dieses Java Runtime Environment ist wie Python ein Interpreter, der aber an den verwendeten Browser gebunden ist.

Tatsächlich ist Java aber mehr eine Compilersprache und demzufolge für den Start nicht so sehr zu empfehlen (genausowenig wie C++, aber auch Pascal ist, obwohl eigentlich »Lehrsprache«, recht hölzern und für den Anfänger ungeeignet). Python ist das erste Beispiel einer Glue-Language, einer

Klebesprache, mit der nicht nur eigener Code, sondern auch Code anderer Sprachen geschrieben und getestet werden kann. Für Python existieren Interfaces zu sehr vielen bekannten Sprachen wie C, C++ und sehr guten weltweit benutzten Bibliotheken wie Tk und Qt.

Da diese Schrift aber keine Rechtfertigung für den Einsatz von Python sein soll, nur noch eine Randbemerkung: Python hat nichts mit einer Schlange zu tun, sondern geht auf die geniale Komikertruppe »Monty Python« zurück, für die der Entwickler Guido van Rossum verständlicherweise eine gewisse Begeisterung hat. Python wird ständig weiterentwickelt und kann über *www.python.org* für alle Plattformen kostenlos im Rahmen der von Linux bekannten GPL bezogen werden.

Anhang B

Die ersten Schritte

Besorgen Sie sich bitte auf jeden Fall eine Pythonversion unter der Adresse *www.python.org*. Es ist wichtig, alles auszuprobieren.

B.1 Aufrufen des PYTHON-Interpreters

Die Beschreibung in diesem Absatz bezieht sich auf Linux, da die neueren Windowsversionen keine Kommandoebene mehr kennen (sehr grosser Fehler dieses ach so fortschrittlichen Systems). Rufen Sie eine Shell auf und geben Sie am Eingabeprompt einfach *python* ein. Sie erhalten etwa das folgende Bild:

```
Python 2.2.1 (#1, Sep. 10 2002 , 17:49:17)
[GCC 3.2] on linux2
Type »help«, »copyright«, »credits« or »license« for more information
>>>
```

Die oberen beiden Zeilen geben Übersetzungsinformationen wieder. Neben der Pythonversion und dem Erstellungszeitpunkt ist auch die Version des Übersetzers *gcc* angegeben, was darauf hinweist, dass der Python-Interpreter in C programmiert wurde, denn der *gcc* ist nicht nur ein, sondern *der* Compiler unter Unix für C-Programme¹.

Die vierte Zeile ist der Eingabeprompt des Interpreters. Testen Sie einfach einmal, ob Sie die folgende Anweisung zum Laufen bringen:

¹Es gibt zwar noch andere, aber der *gcc* ist mit Abstand der am meisten verwendete.

```
>>>print 'Hallo, hier bin ich.'
```

Das Ergebnis ist wenig überraschend. Danach erscheint wieder der Eingabeprompt. Wenn Sie Lust haben, rechnen Sie einmal ein wenig. Geben Sie einfach nur Terme ein, ohne das Gleichheitszeichen zu drücken, also etwa

```
>>>3*4
```

Das ist eine Multiplikation (* steht für das Multiplikationszeichen). Zum Abschluss *etwas* Interessanteres.

```
>>>for i in range(4):  
...   print 'Bete'  
...   for j in range (4):  
...     print 'und arbete'  
...   print \n
```

Die drei Punkte müssen Sie nicht eingeben. Achten Sie aber unbedingt auf die angegebenen Einrückungen. Das ist sehr wichtig. Wenn Sie sich verschrieben haben, können Sie mit der Cursortaste aufwärts Vergangenes zurückholen. Vielleicht haben Sie Lust, ein wenig mit dem Programm zu experimentieren. Sie verlassen den Interpreter durch Eingabe von STRG-D.

B.2 Aufrufen von IDLE

Guido van ROSSUM hat sogar auch eine sogenannte *Integrierte Entwicklungsumgebung* (IDE) für Python geschaffen, die ebenfalls auf der Seite <http://www.python.org> kostenlos erhältlich ist, die er IDLE genannt hat. Ja, der Name sieht der Abkürzung IDE ähnlich, aber dieses Programm hat ebenfalls etwas mit Monty Python zu tun: es ist nach Eric IDLE, einem der bekanntesten Komiker der Truppe benannt (»Always look on the bright site of life«). IDLE hat im Gegensatz zum regulären Interpreter viele Vorteile:

1. Das Programm zeigt Syntaxhighlighting, d.h. Objekte und Schlüsselwörter der Programmiersprache werden farbig hervorgehoben, was gerade für den Anfänger *besonders* hilfreich ist.

2. IDLE rückt nach der Einleitung der sogenannten Anweisungsblöcke automatisch ein, was unbedingt erforderlich ist.
3. IDLE enthält nicht nur den Python-Interpreter, sondern auch einen Editor für das Schreiben von Modulen, sowie einen Debugger für das »Entlausen« von Programmen.

Alles in allem: Der Einsatz von IDLE ist unbedingt zu empfehlen. Es gibt übrigens auch ein Programm *eric*, das genauso wie IDLE arbeitet, aber auf die grafische Bibliothek »Qt« zugeschnitten ist, während üblicherweise unter Python die unglaublich einfache Bibliothek »Tk« verwendet wird².

B.2.1 Wo befindet sich IDLE?

Wenn Sie Windows benutzen, finden Sie dieses Programm bereits unter dem Menüpunkt »Programme« des Start-Menüs. Unter Linux findet sich das Programm in dem Python-Verzeichnis, das vom Systemadministrator eingerichtet wurde (meist `/usr/share/doc/packages/python`). Dieses Verzeichnis enthält ein Unterverzeichnis mit dem Namen *Tools* und dort findet sich ein weiteres Unterverzeichnis mit dem Namen *idle*. Setzen Sie in Ihr Heimatverzeichnis einen symbolischen Link (via `ln -s`) auf die Datei *idle.py* etwa in der Form

```
ln -s /usr/share/doc/packages/python/Tools/idle/idle.py idle
```

Danach kann *idle* direkt aufgerufen werden. Konfigurationsmöglichkeiten finden Sie in den Dateien *config.txt* und *config-unix.txt*.

B.3 Woher kommt der Name Python?

»And now to something completely different ...«

Python ist *nicht* nach der entsprechenden Schlange benannt, sondern nach der genialen, britischen Komikertruppe *Monty-Python*, für die der Erfinder der Programmiersprache Guido van ROSSUM eine sehr nachvollziehbare Begeisterung hat und die an dieser Stelle mit eine kleine Ehrung verdient.

²siehe Fortsetzung dieser Schrift: »Python-Tkinter«



Der einzige Nicht-Brite der Truppe war der inzwischen verstorbene Australier Graham CHAPMAN, der »Brian« aus dem Film »Das Leben des Brian« war wie Cleese und Gilliam ein echter Theatermann, der Autor von zahlreichen Stücken ist.



John CLEESE ist für viele der Inbegriff des britischen Komikers par excellence. Er ist noch heute tätig und taucht dann und wann auch mal kopflos auf, wie z.B. bei Harry Potter ;-).



Die meisten der für Neulinge in Sachen britischem Humors etwas gewöhnungsbedürftigen Zeichnungen des »Flying Circus« stammen von Terry GILLIAM.



»Ooolwais look on the the braaaight side of life.« Für mich die neben Cleese markanteste Figur der Truppe: der unermüdliche Eric IDLE, vor dem auch heute noch die Fangemeinde nicht sicher ist. Texter, Komponist, sprühende Einfälle. Eine der zentralen Persönlichkeiten britischen Humors überhaupt.

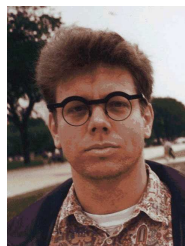


Terry JONES studierte im Gegensatz zu den meisten anderen Mitgliedern der Gruppe nicht in Cambridge, sondern in Oxford und ist wie Idle einer der Hauptdrahtzieher der Pythons. Er ist Texter, Schauspieler und Regisseur (u.a. »Das Leben des Brian«).



Und schliesslich der unverwechselbare Michael PALIN (»I'm a lumberjack and I'm OK«, »And now to something completely different«), der neben John Cleese und Eric Idle vielleicht bekanntesten Python. Texter, Komponist. Ist heute noch sehr erfolgreich tätig.

Eine sehr schöne Idee von Guido van ROSSUM, seine geniale Programmiersprache nach dieser Truppe zu benennen. Und hier nun ist der Meister selbst:



Die enorme Beliebtheit von Python hat sicherlich mehrere Gründe:

1. Die Sprache ist extrem gut lesbar aufgrund konsequenten Designs:

- (a) In Python gibt es *nur* Namen (Zeiger), während andere Sprachen wie Java und C++ sowohl Zeiger als auch Variablennamen kennen. Java unterscheidet sich von C++ nur darin, dass es die Zeigersyntax weitgehend vermeidet, sehr wohl aber den Mischmasch von Zeigern und Variablen zulässt. Python verzichtet konsequent auf Variablennamen. Damit ist eine aufwändige, fehlerträchtige Typendeklaration unnötig.
 - (b) Anweisungsblöcke werden durch eingerückten Code deutlich. Damit ist weder ein *begin* und *end* noch irgendein Klammerdesign (C++) notwendig. Während dem Programmierer bei Pascal, C++ oder Java frei gestellt ist, ob er einen Anweisungsblock einrückt oder nicht, wird er bei Python dazu gezwungen, was sich als sehr vorteilhaft erweist.
2. Erstellter Code kann sofort ausgeführt werden. Python ist zwar eine Skriptsprache, also ein Interpreter, während die schnelleren System-sprachen wie Java und C++ Compilersprachen sind, aber die Testbarkeit einzelner Module ist erheblich eingeschränkt, da immer der ganze Code übersetzt werden muss. Wenn in Python ein Modul fertiggeschrieben ist, kann es sehr wohl in C++ oder Java-Code verwandelt werden. Insofern ist Python die erste »Klebesprache« (Glue Language), die es gibt.
 3. Python hat aus anderen Programmiersprachen so ziemlich das Beste übernommen. Dies gilt vor allen Dingen für die unglaublich flexiblen Datentypen *Liste* und *Tupel*. Um zwei beliebige Datensätze *a* und *b* miteinander zu vertauschen, wie kompliziert sie auch immer sein mögen, ist zum Beispiel lediglich die Anweisung $(a, b) = (b, a)$ nötig. Und Ähnliches mehr. Die *Liste* ist ein Datentyp, der während des Programmlaufes wachsen und schrumpfen kann. Mit diesem Datentyp eine verkettete Liste herzustellen ist ein Kinderspiel.
 4. Während Java objektorientierte Programmierung erzwingt, ist in Python sowohl das prozedurale als auch das objektorientierte Programmieren möglich. Eine Mixtur ist zu vermeiden!
 5. Python besitzt Anbindungen an sehr viele grafische Bibliotheken (z.B. Qt, GTK) und Datenbanken (MySQL und PostgreSQL). Der Shooting Star unter allen grafischen Bibliotheken ist jedoch die sehr beliebte Tk, die durch das Befehlspaket *Tkinter* angesteuert wird. Während Entwicklungswerkzeuge wie Delphi den Programmierer von Anfang an

mit einer grafischen Oberfläche »belästigen«, ist dies bei Python nicht der Fall. Der Programmierer entwickelt zuerst einen lauffähigen Code, der nur den Algorithmus umsetzt, und das möglichst gut und effektiv. Dann bedarf es einer Zeit von nur wenigen Minuten, die Grafik draufzusetzen. Die Grafik ist also nahezu ein Selbstgänger! Tk wird immer wieder genannt, wenn es darum geht, im Prinzip zu verstehen, wie grafische Bibliotheken arbeiten. Warum Tk? Weil es so unglaublich einfach ist und jeden Ballast vermeidet.

6. Python sticht in Sachen Netzwerkprogrammierung sogar Perl aus.

Wer wirklich den Umgang mit Python erlernt, dürfte sich das Tor zu allen Programmiersprachen öffnen, in denen so manches Detail, was bei Python nur wenige Übungen beansprucht, sehr viel schwieriger zu lernen ist.

B.4 Hilfe bei Problemen

Eine erstklassige Hilfe bekommen Sie in jedem Fall von der Python-Gemeinde. Ob Sie nun über die Adresse *www.python.org* sich in eine Mailingliste eintragen, oder die zahlreiche Dokumentation befragen, die sich dort findet. Immer wird Ihnen sehr schnell und sehr zuvorkommend geholfen. Der Support ist wirklich erstklassig!

Bei der Linux-Version finden Sie auch in dem */doc*-Verzeichnis ihrer Python-Directory zahllose sehr gute Beispiele!

Index

- Überladung, 51
- Anweisungsblöcke, 33
- Anweisungsblock, 33
- ASCII, 57
- Assembler, 56
- Attribute, 45
- Ausgabe, 39
- Ausnahmen, 54
- Binärcode, 58
- Bit, 56
- break, 32
- Byte, 56
- class, 42
- Compiler, 55
- Complex, 16
- continue, 32
- Dateien, 47
- Datenobjekt, 2
- Datenobjekte, 16
- Debugger, 55
- def, 34
- Endlosschleife, 32
- Enthaltensein, 29
- Entwicklungsumgebung, integrier-
te, 55
- except, 54
- Floats, 16
- For-Schleife, 30
- from, 52
- Funktionen, 2
- garbage collector, 10
- IDE, 55
- IDLE, 55, 68
- import, 52
- Instanz, 44
- Instanzen, 45, 49
- Integer, 16
- Interpreter, 55
- Klasse, 2, 42
- Klassen, 45
 - Methoden, 3
 - Nachrichten, 3
- Kommentare, 38
- lambda, 40
- Link
 - harter, 9
 - weicher, 9
- Maschinensprache, 55
- Methode, 43
- Methoden, 3, 45
- MethodenSie, 49
- Modul, 11
- modulo, 17
- Monty-Python, 69
- Nachrichten, 3

- Objekte, 2
 - Dateien, 26
 - Datenobjekt, 2
 - Dictionaries, 23
 - Funktion, 2
 - Klasse, 2
 - Listen, 20
 - Strings, 18
 - Tupel, 22
 - Zahlen, 16
- Operatoren, 17
- pass, 32
- Pointer, 9
- print, 39
- Programm, 11
- Programmieren
 - objektorientiert, 48
 - prozedural, 48
- Programmiersprache, 55
- Python, Interpreter, 67
- rawinput, 39
- reload, 53
- return, 35
- Rossum, Guido van, 69
- Schleifen, 30
- self, 43
- Sequenzen, 22
- slicing, 19
- Speicherleiche, 10
- Tastatur, Einlesen, 39
- try, 54
- Typendeklaration, 9
- Unicode, 57
- Vererbung, 46
- Vergleiche, 29
- Verzweigung, 32
- While-Schleife, 31