Teil I

Python-Geist, Anfänger-Geist

Kapitel 1

Was ist Python?

Schön ist besser als hässlich. *Tim Peters*[1]

Das erste Kapitel dieses Buchs führt in Python ein: Sie erfahren, was Python ist, lernen die Geschichte dieser Sprache kennen und erfahren, wann Python nützlich und wann Python nicht nützlich ist. Sie werden nicht einmal einen Computer für diesen ersten Teil benötigen, es sei denn, Sie machen die Übungen, bei denen Sie gebeten werden, ein Dokument auf einer Internet-Seite zu lesen.

1.1 Warum programmieren? Warum in Python programmieren?

Sofern Sie mit einem Computer lediglich Ihre Bilanz erstellen möchten, brauchen Sie nicht programmieren zu können. Es gibt bessere Werkzeuge – z.B. Bleistift, Papier und Taschenrechner. Und falls Sie Ihren Computer ausschließlich für Textverarbeitung und Seitenlayout nutzen wollen, brauchen Sie ebenfalls nicht Programmieren zu lernen; es gibt auf dem Markt Unmengen von Programmen, die das, was Sie vorhaben, äußerst gut machen.

1 Anm. d. Übers.: Falls Sie sich fragen, wer dieser Tim Peters ist (der noch für ein paar weitere Zitate gut ist), sei Ihnen Folgendes verraten: Tim Peters beantwortet in der Python-Newsgruppe mehr Fragen als irgendjemand sonst auf kompetente und weise Weise. Da ihn jedoch noch niemand außer Guido persönlich getroffen hat, ist es unklar, ob er wirklich ein Mensch ist oder lediglich ein besonders raffiniertes Python-Programm ...

Aber falls es für Ihre Anwendung keine Software gibt oder wenn die bereits existierende Sie nicht zufrieden stellt, bleibt als einzige Antwort, dass Sie Ihre eigene Software machen. Dieses einfache Prinzip hat vermutlich zu mehr Durchbrüchen in der Programmierung geführt als jedes andere. Linux ist das perfekte Beispiel: Linus Torvalds war unglücklich mit den bestehenden Implementierungen von Unix für PCs und beschloss daher, seine eigene Version zu schreiben. Heute ist Linux populär genug, um Bill Gates und Microsoft zu beunruhigen.

Unix wurde in den frühen Siebzigern im AT&T-Labor in Murray Hill, New Jersey, entwickelt. Als mächtiges Mehrbenutzer-Betriebssystem war es das geistige Produkt von Dennis Ritchie, Brian Kernighan und Ken Thompson, die anscheinend über zuviel Zeit und einen Computer verfügten, den niemand sonst benutzen mochte. Selbst dieser Computer war bis zum Ende der achtziger Jahre viel zu teuer für Heimanwender. Diejenigen, die Unix bei der alltäglichen Arbeit benutzten, schauten lächelnd auf die schwächlichen Betriebssysteme herab, die es für PCs gab. Wir waren durch unsere »großen Kisten« verdorben. 1987 jedoch entwickelte Andrew Tanenbaum ein sehr kleines Unix-artiges Betriebssystem, das auf Heim-PCs laufen würde. Er nannte es *Minix*. Linus Torvalds entwickelte später eine portablere und nützlichere Version davon und nannte sie Linux; sie verfügt über mindestens die gleichen Fähigkeiten wie kommerzielle Versionen und läuft auch auf preiswerten Heim-PCs. Der wesentliche Unterschied zwischen Minix und Linux bestand in den frühen Tagen darin, dass die Lizenzierung für Minix strenger war als die für Linux. Heute zeigt sich der Unterschied darin, dass es Tausende von Linux-Begeisterten gibt und dass nahezu alles, was man sich wünscht, bereits von jemand anderem ins Rollen gebracht wurde.

Besonders Wissenschaftler benötigen häufig Software, die es noch nicht gibt, und schreiben oft selbst welche, um ihre Forschungsgebiete voranzubringen. Obwohl ich kein Wissenschaftler bin, verfüge ich über ein Forschungsgebiet: Ich bin vom Kalender der Maya fasziniert und habe mich Jahre lang damit beschäftigt, C-Programme zu schreiben, die mir helfen sollten, diesem Interesse nachzugehen. Als ich Python fand, gab ich den Programmcode, den ich bereits geschrieben hatte, rasch auf und implementierte von nun an in Python alles erneut. Die Programme und Bibliotheken, die dabei herauskamen, sind aufgeräumter, einfacher, kleiner und wesentlich mächtiger und ich war in der Lage, alles, was ich benötigte, in weit weniger Zeit zu erstellen als es mir mit dem Originalcode möglich gewesen wäre.

Viele andere Leute sind gezwungen, zumindest ein wenig Programmieren zu lernen, um wiederkehrende, langweilige Aufgaben zu automatisieren. Ein

Beispiel wären kleinere Programme, welche die wöchentlichen Berichte einer Arbeitsgruppe aus einem speziellen Verzeichnis oder Ordner heraussuchen, sicherstellen, dass jeder seinen Bericht auf den neuesten Stand gebracht hat, einige einfache Operationen ausführen, um die einzelnen Berichte zu einem zusammenzufassen und das Ergebnis auszudrucken oder dem Gruppenleiter E-Mails zu schicken. Ich hatte am Anfang meiner Laufbahn als Programmierer Derartiges zu tun und war dabei erfolgreich, wenn auch mit einer großen Portion Hackerei unter Verwendung etlicher verschiedener Skript-Sprachen. Hätte es damals bereits Python gegeben, hätte ich alles in kürzerer Zeit mit weniger Zeilen und in einer einzigen Programmiersprache erledigen können.

In der folgenden Liste finden sich solche wiederkehrenden Aufgaben, mit denen ich im Laufe der Jahre zu tun hatte; viele davon können nun in Python zufriedenstellend gelöst werden:

- ✘ Berichte einsammeln und zu einem umfassenderen Bericht zusammenfassen
- ✘ URL (Uniform Resource Locator, Internet-Adresse) in einem Internet-Dokument auf ihre Gültigkeit überprüfen,
- ✘ regelmäßig automatische Sicherungskopien wichtiger Dateien und Ordner erstellen,
- ✘ einen automatischen Bericht per E-Mail versenden, um Ihrem Chef vorzutäuschen, dass Sie wirklich etwas leisten,
- ✘ automatisch PERT-Diagramme (im Projektmanagement benutzte Diagramme) mit wesentlich einfacheren Eingaben erzeugen
- ✘ eine Liste aller Dokumente in einem bestimmten Verzeichnisbaum anfertigen und diese je nach Endung oder Erweiterung weiterverarbeiten,
- ✘ Dateilisten in einer bestimmten Reihenfolge erstellen, die in anderen Programmen Verwendung finden können,
- ✘ Ihre Videosammlung im Griff behalten.

Um eigene Software zu erstellen, musste man in der Vergangenheit zunächst eine gehörige Portion komplizierter und geheimnisvoller Syntax erlernen, bevor selbst die einfachsten Programme geschrieben werden konnten. FORTRAN, eine frühe, aber noch immer beliebte Sprache, ist sehr gut für wissenschaftliches Programmieren geeignet, weil sie über viele mathematische Funktionen verfügt, aber ihre Syntax ist – nun, nicht einleuchtend. C, eine andere Sprache, von der Sie vermutlich schon gehört haben, hat wegen der immensen Möglichkeiten, die sie dem Programmierer bietet,

viele Anhänger, aber es ist allzu leicht, damit knifflige, nahezu unlesbare Programme zu schreiben. C-Programmierer räumen ein, dass die Sprache zu schlechten Programmiergewohnheiten verleitet, doch »FORTRAN verstärkt sie«. Die Benutzung von Python dagegen führt zu guten Programmiergewohnheiten (oder ermutigt wenigstens dazu) und versucht, die Lernkurve dergestalt zu verkürzen, dass die lernend verbrachte Zeit auf ein Minimum reduziert wird.

Die folgenden Beispiele sind einige kleine, ausführbare Programme in FORTRAN, C und Python; sie tun nicht viel, aber das traditionelle Anfänger-Programm jeder Sprache gibt eben die Phrase »Hallo, Welt« auf dem Bildschirm aus.

FORTRAN:

```
PROGRAM
    PRINT *, "Hallo, Welt"
END PROGRAM
```

C:

```
#include <stdio.h>
main() {
    printf("Hallo, Welt\n");
}
```

Python:

```
print "Hallo, Welt"
```

Ich halte Python für die nahezu perfekte Einsteigerprogrammiersprache; die Syntax ist relativ einfach und schlicht und anstelle »vieler Möglichkeiten, etwas zu tun« gibt es in der Regel einen offensichtlich besten Weg – oder ein paar gute Wege. Anstelle von Erweiterungspaketen, die ausfindig gemacht werden müssen und zur Installation einiges Fachwissen erfordern, sind wesentliche Merkmale bereits im Standardumfang von Python enthalten, von denen einige beim wissenschaftlichen Programmieren äußerst dienlich sind. Für ausgefallenere Programmieraufgaben wird ein Erweiterungspaket mit dem Namen Tcl/Tk benötigt, das vom Windows-Installationspaket automatisch für Sie eingerichtet wird. Trotz ihrer grundsätzlichen Einfachheit gestattet es die Sprache, komplexe und anspruchsvolle Ideen auf intuitive Weise auszudrücken, denn sie verwendet systematisch und konsequent das Konzept der so genannten *objektorientierten Programmierung* (OOP).

In den folgenden Kapiteln werde ich versuchen, Ihnen ein solides Verständnis der elementaren Programmierung in Python zu vermitteln. Das erste Drittel des Buchs umfasst die grundlegenden Elemente in Python. Das zweite Drittel behandelt Objekte von Grund auf, denn Objekte sind von fundamentaler Bedeutung, wenn man die Möglichkeiten der Sprache voll aus-

schöpfen will. Das letzte Drittel des Buchs beinhaltet Pythons plattformunabhängige, grafische Benutzerschnittstelle *tkinter*. Wir werden dort ebenfalls ein wenig die Programmierung mit dem *Common Gateway Interface* (CGI) im Internet streifen, um Ihnen anzudeuten, wie nützlich Kenntnisse darüber sein können.

In diesem Buch gehe ich davon aus, dass Sie nichts über das Programmieren wissen. Ohne die vorgefassten Meinungen und unnötigen Informationen aus anderen Programmiersprachen werden Sie entschieden im Vorteil sein und ich hoffe, dass sich herausstellen wird, dass die Programmierung von Computern oftmals viel leichter ist, als man meint. Ich gehe allerdings davon aus, dass Sie Computer bedienen können und über grundlegendes Wissen im Hinblick auf Pogramme wie Textverarbeitungen, Texteditoren und die Befehlszeilen-Schnittstelle Ihrer speziellen Plattform (DOS oder die von Ihnen bevorzugte Shell wie *ksh*, *csh* oder *bash* unter Unix) verfügen.

Obwohl ich zahlreiche vernünftige Gründe für das Programmieren angesprochen habe, zählt meines Erachtens letztendlich, dass die Entwicklung funktionierender Programme Spaß macht. Das prickelnde Gefühl, wenn man in der Lage ist, einige grundlegende Anweisungen zusammenzufügen, und sieht, dass das Ergebnis exakt so ausfällt, wie man es sich auf einem Computer – einer Maschine, die manche Menschen als widerspenstig und frustrierend ansehen – vorgestellt hat, ist mit nichts zu vergleichen. Für mich ist das Programmieren in Python niemals nur ein Job gewesen. Ich hoffe, dass Sie nach dem Durcharbeiten dieses Buchs in der Lage sein werden, ebensoviel Spaß beim Programmieren zu haben wie ich.

1.2 Die Geschichte von Python

Python wurde gegen Ende 1989 von Guido van Rossum während eines Weihnachtsurlaubs entwickelt, als sein Forschungslabor geschlossen war und er nichts weiter vorhatte. Er nahm Eigenschaften aus vielen Programmiersprachen wie *ABC*, *Modula-3*, *C* (zumindest die am wenigsten umstrittenen Merkmale) und einigen anderen. Er sah sich besonders gern *Monty Python's Flying Circus* im Fernsehen an und als es daran ging, die Sprache zu benennen, wählte er Python. Nach der Erprobung innerhalb einer kleinen Gruppe von Freunden und Kollegen wurde Python 1991 als Public Domain veröffentlicht. Im Gegensatz zu einigen anderen Sprachen ist Python nicht nur absolut kostenlos, sondern es gibt auch keinerlei Einschränkungen des Gebrauchs. Es wird nicht erwartet, dass in dieser Sprache entwickelte Programme als Public Domain veröffentlicht werden, es wird von Programmierern nicht verlangt, dass sie Guido über Veränderungen in

Kenntnis setzen, und Programme in Python können ohne Lizenzgebühren von und an jedermann verkauft werden.

Neben der Syntax der Sprache selbst war die Entscheidung, Python als Public Domain zu veröffentlichen, ein wesentlicher Faktor seiner weltweiten Akzeptanz. Andere Sprachen mögen eine größere Zahl von Anwendern haben, aber nur wenige Sprachen können sich einer derart leidenschaftlichen Nutzergemeinde rühmen wie Python. Python mag eine junge Programmiersprache sein, aber seine passionierten Anwender versammeln sich mindestens einmal jährlich, mitunter auch häufiger, zu internationalen Konferenzen.

Die Nutzergemeinde hat eine informelle Organisation geschaffen, die sich der Unterstützung und Verbreitung des Gebrauchs von Python widmet: die *Python Software Activity* (PSA), welche etwa 300 natürliche Personen und 30 Körperschaften umfasst. Einzelmitglieder zahlen gutes Geld zur Unterstützung von Python; Körperschaften bezahlen noch mehr. Die Mitgliedschaft ist nicht erforderlich, sondern streng freiwillig; es wird von niemandem jemals verlangt, auch nur einen Pfennig für den Gebrauch von Python zu welchem Zweck auch immer zu bezahlen. Daher ist es bemerkenswert, dass so viele zum Gedeihen dieser Sprache beigetragen haben.

In den frühen Jahren lautete eine häufig gestellte Frage: »Was geschieht, wenn Guido von einem Bus überfahren wird?« Die Gesellschaft sorgte sich, dass im Falle von Guidos Tod auch Python stürbe. Im Jahre 1998 wurde das *Python Consortium* gegründet, um das Überleben und Wachstum von Python zu sichern. Körperschaftliche Mitglieder des Konsortiums zahlen eine beträchtliche Summe Geld an Guido, damit er an Python (und an nichts anderem) arbeiten kann; sie stellen der Python-Gemeinde weitere nützliche Dienste zur Verfügung und ernennen einen Nachfolger für die Zeit, wenn er nicht mehr willens oder fähig sein sollte, den künftigen Kurs von Python vorzugeben.

Es erscheint sicher, dass Python nicht einfach nur überleben, sondern mit Nachdruck ins 21. Jahrhundert übergeht. Guido beschreibt sich selbst als »konservativen Programmierer«, dessen Bestimmung es ist, dass sich Python ausschließlich in Richtungen, die er für nötig erachtet, weiterentwickelt. Ein grundlegender Vorteil dieser Aussicht besteht darin, dass Programme, die mit frühen Versionen von Python erstellt wurden, auch in künftigen Versionen größtenteils weiterhin unverändert laufen werden. Ich begann damit, Python zu verwenden, als es Version 1.3 gab, und man ist jetzt bei Version 1.5.2; sämtlicher Programmcode, den ich schrieb, läuft in der neuesten Version noch immer ohne Veränderung. Die Version 1.5.2 bildet die Grundlage

für die Beispiele und Code-Ausschnitte in diesem Buch; mit der nächsten Version, 1.6, deren Erscheinen im Laufe des Jahres 2000 fällig wird, sollte der gesamte, in diesem Buch enthaltene Programmcode unverändert laufen. Irgendwann im Jahr 2000 oder 2001 wird die Version 2.0 in den Beta-Test gehen und der gesamte Programmcode dieses Buchs sollte im Wesentlichen noch immer einwandfrei funktionieren. Obwohl der Code hier unter Verwendung von 1.5.2 erstellt wurde, ist er, soweit es möglich war, mit früheren Versionen getestet worden. Falls Sie bezüglich der Versionsunterschiede auf irgendwelche Probleme stoßen sollten, schauen Sie auf der Internet-Seite zu diesem Buch nach, ob dort bereits Korrekturen festgehalten sind. Falls Sie keine Korrektur oder Revision auf der Internet-Seite finden, schicken Sie dem Autor eine vollständige Beschreibung des Problems.

1.3 Das schädliche goto

Im Jahre 1968 schrieb Edsger W. Dijkstra, eine Persönlichkeit in der Programmierung, einen Brief an den Herausgeber der *Communications of the ACM* mit der Behauptung, dass sich die `goto`-Anweisung, ein Element nahezu jeder Programmiersprache jener Zeit, nachteilig auf die Denkweise von Programmierern auswirke.

Ein `goto` (gehe zu) ist eine Anweisung in der Computersprache, die dem Computer sagt, dass er an eine andere Stelle im Programm springen und die dortigen Anweisungen ausführen soll. Sobald solche Anweisungen abgearbeitet sind, muss sich der Programmierer überlegen, wohin der Computer zurückspringen soll. Die Programmierung ohne `goto`-Anweisungen heißt *strukturierte Programmierung*, da dem Programmierer wie den Lesern von Programmen in der Regel recht klar ist, was das Programm zu welchem gegebenen Zeitpunkt macht. Der Gebrauch der `goto`-Anweisung in einem Programm bedeutet dagegen, dass die einzelnen Schritte des Programms gewissenhaft verfolgt werden müssen, was üblicherweise sehr fehleranfällig ist.

Im Jahre 1968 wurde die überwiegende Mehrheit der Programme in so genanntem *Spagetti-Code*, einem von zahlreichen `goto`-Anweisungen gekennzeichneten Stil, verfasst, in dem es keine Modularisierung und nur wenige Unterprogramme gibt. Wir werden Unterprogramme beziehungsweise Prozeduren und Funktionen ebenso wie die Modularisierung in späteren Lektionen besprechen.

Im Jahre 1968 schrieb ich Programme in COBOL. Wie jeder andere schrieb ich Spagetti-Code; dieser war selbst für seinen Autor verdammt schwer zu lesen und schien nahezu unlesbar, wenn man ihn nicht selbst programmiert hatte. Die wenigen strukturierten Programmkonstrukte waren von begrenztem Nutzen; sie waren schwer zu handhaben und es war schwierig, an Dokumentationsmaterial heranzukommen. Die einzige Möglichkeit, die Programmlogik zu verstehen, bestand oftmals darin, sich ein so genanntes Flussdiagramm anzuschauen, d.h. ein besonderes Diagramm mit speziellen Symbolen für Eingang, Ausgang und Entscheidungen. Fertigte ein Programmautor kein Flussdiagramm an, musste man ein spezielles Programm benutzen; dieses las das erste Programm ein und zeichnete das Diagramm automatisch. Nur zu wenigen Programmen wurden im Voraus Flussdiagramme gezeichnet – was man als den korrekten Weg, ein Programm zu schreiben, angesehen hätte. Aus meiner persönlichen Erfahrung kann ich wahrheitsgemäß sagen, dass die `goto`-Anweisung in der Tat dazu ermuntert, ein Programm in Unordnung zu bringen.

COBOL ist eine der ältesten Programmiersprachen und half seinerzeit, in der Ära der Großrechner wie IBM 360, den Gebrauch von Computern populär zu machen. Sie wurde von Admiral Grace Hopper erfunden, die für die Redensart »es ist leichter, Vergebung als Erlaubnis zu erlangen« Bekanntheit erlangte. COBOL war daher beachtenswert, weil sie auf für den Menschen lesbaren, verständlichen Worten basierte und nicht auf speziellen Zahlen, deren Sinn sich nur den Computern erschloss. Heute gilt sie als hoffnungslos wortlastig. Ich überlegte, damit ein »Hallo, Welt«-Programm zu schreiben, um zu demonstrieren, wie wortlastig die Sprache ist, aber es war zu lang.

Sowohl Benjamin Whorf als auch Noam Chomsky glauben, dass Sprache das Denken strukturiert; man kann nur denken, was sich in Worten ausdrücken lässt. Wenn es in Ihrer Sprache keine Zukunftsform gibt, werden Sie es schwer (wenn nicht gar unmöglich) finden, über Ereignisse nicht in Vergangenheit oder Gegenwart nachzudenken. Computer-Programmiersprachen sind ein guter Beweis für diesen Standpunkt; sofern es keine »Worte« oder Methoden in einer Computersprache gibt, mit denen sich das, was Sie vorhaben, ausdrücken lässt, ist es schwierig, über die Dinge nachzudenken, die Ihnen nicht möglich sind. Anfängliche Programmiersprachen waren monolithische Konstrukte, eher verbrochen als erschaffen, um die Eigentumsabsichten der Firmen, die sie erfunden hatten, voranzutreiben. Es war nahezu ausgeschlossen, den Sprachen neue Merkmale hinzuzufügen, da man keinen Zugang zum Quellcode der Sprache oder ihren Entwicklern

hatte. Darüber hinaus war es Programmierern nicht möglich, der Sprache irgendetwas hinzuzufügen, woran die Entwickler nicht gedacht hatten.

Erweiterbarkeit wurde zu einem Ziel für die Sprachentwickler; als Guido Python entwarf und entwickelte, machte er es sehr leicht, der Sprache Eigenschaften und Module hinzuzufügen. Er ließ auch die `goto`-Anweisung aus, bot stattdessen viele nützlich strukturierte Programmiertechniken und orientierte sich beim Entwurf der Sprache an einer Reihe neuer Prinzipien: denen des *objektorientierten Entwurfs*. Die ersten Softwareentwicklungstheoretiker behaupteten, dass die Daten, mit denen ein Programm arbeitete, das Ausschlaggebende seien; definiere, wie deine Daten strukturiert sind, dann ergeben sich die Methoden, die du anzuwenden hast, um sie zu beeinflussen, ganz von selbst. Das war ihre Aussage. Später wurden *Objekte*, Teile der Software, welche die strukturierten Daten mit den manipulativen Methoden verknüpfen, bei der Programmierung weit wichtiger. Guido erleichterte die Verwendung von Objekten in Python, viel mehr als in den meisten anderen Sprachen. Einige Sprachen pfropften Objekte in einen hauptsächlich linearen Kern, aber Python wurde von Anfang an auf Objektorientierung ausgerichtet. Sie können über Objekte in Python *nachdenken* und wir werden im zweiten Drittel des Buchs mehrere Kapitel damit verbringen, Fertigkeiten und Techniken der objektorientierten Programmierung zu entwickeln. Sie werden feststellen, dass es, sobald Sie über Objekte nachdenken können, äußerst einfach ist, sie in Python zu implementieren.

1.4 Stärken und Schwächen von Python

Python ist eine ausgezeichnete Sprache für etliche Anwendungsbereiche. Python-Programme können Shell-Scripts ersetzen und die zur Ausführung einer Aufgabe erforderliche Anzahl von Zeilen deutlich senken. Eine ganze Reihe von C++-Programmierern verwendet Python zum Erstellen von *Prototypen*, was bedeutet, dass sie, anstelle die Spezifikationen eines Programms mühselig bis ins letzte Detail auszuarbeiten, einen Prototypen in Python bauen, der eine *grafische Benutzungsoberfläche* (GUI, Graphical User Interface) enthalten kann. Weil Python so ausgeklügelt ist, nimmt die Erstellung von Prototypen damit weniger Zeit (mitunter auch drastisch viel weniger Zeit) in Anspruch als das Niederschreiben oder Zeichnen einer kompletten Spezifikation. Programmierer, die sich mit diesen Dingen befassen, sagen sogar, dass sie am Ende ein besseres Produkt erzielen, da Python zu klarem und elegantem Denken anregt. Selbst wenn die endgültige Version eines Produkts in C++ verfasst sein muss, sagen sie, dass der vorherige

Gebrauch der Sprache Python schließlich zu viel kleineren, besser konstruierten Programmen führt.

Nichtsdestotrotz bleiben einige wenige Bereiche, in denen Python nicht glänzt. Die Bearbeitung sehr langer Textdateien, in denen komplizierte *regular expressions* (RE, Reguläre Ausdrücke) vorkommen, dauert in Python in der Regel wesentlich länger als zum Beispiel in Perl. Obwohl die Unterschiede zwischen den beiden Sprachen in der Behandlung von Regulären Ausdrücken oftmals durch den Einsatz einiger simpler Optimierungstechniken minimiert werden können, eignet sich Perl für manche Aufgaben unbestritten besser als Python. Grundsätzlich aber gleicht die Geschwindigkeit, mit der ein Programmierer ein weiteres Werkzeug oder einen zusätzlichen Prototyp erzeugen kann, den Geschwindigkeitsunterschied in der Ausführung allemal aus. Die Zeit zwischen Idee und Implementierung ist in den meisten Sprachen maßgeblich für die Höhe der Kosten eines Programms verantwortlich, doch mit Python kann diese teure Lücke so weit überbrückt werden, dass Programmierer nach den Worten von Frank Stajano »ausführbare Ideen« haben können. Die Niederschrift Ihrer Ideen in Python verschafft Klarheit und wird Ihnen oftmals innerhalb kürzester Zeit zu einem marktreifen Programm verhelfen.

Programme, die in Echtzeit ausgeführt werden müssen, sind möglicherweise nicht zur kompletten Implementierung in Python geeignet. Interpretierte Sprachen sind generell zu langsam für die Art unmittelbarer Reaktion, die von derartigen Server-Programmen erwartet wird. Beispielsweise wäre Python eine schlechte Wahl für ein Voicemail-Programm, dass gleichzeitig einige hundert Telefonanschlüsse bedienen muss. Dennoch gibt es verschiedene Möglichkeiten zur Verbesserung der Reaktionszeit eines solchen Programms. Zunächst einmal können einige kritische Bereiche des Programmcodes in einer maschinennahen Sprache wie C verfasst werden. In Python werden solche Teile in C genauso behandelt wie die eingebauten Befehle. Das String-Modul wurde in C geschrieben; deshalb sind Operationen mit Strings in Python extrem schnell. Aus diesem Grund ist Python *erweiterbar*, wobei man die Geschwindigkeit maschinennaher Sprachen in Python nutzen kann. Zweitens können auch maschinennahe Sprachen Python *einbetten*, was es erleichtert, Python-Funktionen von C (oder ähnlichen Sprachen) aus aufzurufen. Diese Einbettbarkeit macht die Stärken von Python maschinennahen Sprachen zugänglich. Diese beiden Eigenschaften machen Python zu einem äußerst praktischen *Klebstoff*, der bestehenden Teilen die Zusammenarbeit als ein vereinigtes Ganzes ermöglicht.

1.5 Zusammenfassung

In diesem ersten Kapitel haben Sie erfahren, was Programmieren ist, warum Sie das Programmieren möglicherweise erlernen möchten und warum Python für Sie eine gute Wahl als Einsteigerprogrammiersprache darstellt. Sie haben einiges über die Geschichte dieser Sprache und etwas über die Geschichte von Programmiersprachen im Allgemeinen erfahren und Sie haben eine der einflussreichsten Ideen der Informatik kennen gelernt: dass die von Ihnen gewählte Programmiersprache beeinflusst, was Sie überhaupt programmieren können und wie es strukturiert ist. Abschließend haben wir einige Bereiche gestreift, in denen Python nicht unbedingt die beste Wahl ist, und Sie lernten einige Arten kennen, wie sich Pythons Schwächen in diesen Bereichen ausgleichen lassen.

1.6 Workshop

1.6.1 F&A

F: Ist Python portabel?

A: Python lässt sich auf nahezu jeder Plattform ausführen; bereits übersetzte Binärdateien sind für die meisten Betriebssysteme erhältlich. Das einzige größere Betriebssystem, das von Python nicht unterstützt wird, ist NetWare, doch zumindest eine Person hat öffentlich zugegeben, daran zu arbeiten.

F: Kann ich Python als CGI-Sprache (Common Gateway Interface) verwenden?

A: Ja. Die einzige Voraussetzung dazu besteht darin, dass der Internet-Server Python-Programme unterstützen und deren Ausführung zulassen muss. Sie können allerdings auf Ablehnung seitens der Internet-Server-Administratoren stoßen, die nicht noch eine weitere CGI-Sprache auf ihrem System haben wollen.

F: Ist Python zur CGI-Programmierung sicher genug?

A: Python ist diesbezüglich sicherer als Perl, aber weniger sicher als Java.

F: Verhilft Python zu Ruhm und Ehre?

A: Nicht unbedingt, aber Sie werden keinen Schaden nehmen, wenn Sie Python lernen.

F: Wird Python von irgendwelchen führenden Firmen zur Programmentwicklung genutzt?

A: Ja, sicher. Zu den Firmen, die sich auf Python verlassen, zählen die NASA, Yahoo, Red Hat, Infoseek und Industrial Light and Magic. Wir wissen auch, dass einige andere große Fische in der Computerindustrie Python benutzen; sie geben dies aber nur ungern zu, weil sie spüren, dass ihnen Python einen wirklichen Wettbewerbsvorteil verschafft. Dinge schneller als andere Firmen zu erstellen, macht sie ihren Kunden gegenüber aufgeschlossen und verspricht Folgegeschäfte.

1.6.2 Quiz

1. Python ist als Einsteigerprogrammiersprache besonders geeignet, denn Python ist:

   a) leistungsfähig, schnell, monolithisch.

   b) flexibel, erweiterbar, einbettbar.

   c) elegant, klar, einfach.

   d) echtzeitsfähig, mit leistungsfähigen regulären Ausdrücken ausgestattet, ähnlich wie C.

2. Wer stellte zuerst die Behauptung auf, dass `goto`-Anweisungen zu unstrukturierten Programmen führen?

   a) Nicklaus Wirth

   b) Edsger Dijkstra

   c) Benjamin Wolf

   d) Charles E. Thompson, Jr.

3. Wer hat Python erfunden?

   a) Tim Peters

   b) Ivan Van Laningham

   c) Guido van Rossum

   d) Edsger Dijkstra

1.6.3 Antworten

1. b, c. Gute Gründe für die Wahl von Python sind b und c; Python ist flexibel, erweiterbar, einbettbar, elegant, klar und einfach.

2. b. Edsger Dijkstra hielt die `goto`-Anweisung für schädlich, da sie es den Programmieren gestattete, »ihre Programme in Unordnung zu bringen«.

3. c. Guido van Rossum ist der Schöpfer von Python, egal, was Tim Peters behaupten oder ich mir wünschen mag. Wie dem auch sei, Tim hat (abgesehen davon, dass er ein sehr lustiger Vogel ist) maßgeblich zur Entwicklung von Python beigetragen, wogegen ich mich glücklich schätze, Python einfach benutzen zu können.

1.6.4 Übungen

✘ Besuchen Sie `http://www.python.org/`, den Ausgangspunkt für alle Python betreffenden Dinge. Als dieses Buch entstand, gewann eine weitere äußerst wertvolle Online-Quelle an Bedeutung, *The Vaults of Parnassus*, die unter `http://www.vex.net/parnassus/` zu finden ist. Dies ist eine umfangreiche Website mit Volltextsuche und vielen Links zu Software, die auf der ganzen Welt in Python geschrieben wurden.

✘ Schließen Sie sich der Python-Mailinglist oder der Python-Tutoren-Mailinglist an. Unter `http://www.python.org/psa/MailingLists.html` können Sie mehr erfahren und sich anmelden.

✘ Lesen Sie Edsger Dijkstras kurzen Brief über `goto`-Anweisungen, der unter `http://www.acm.org/classics/oct95/` auf der Internet-Seite der ACM (Association for Computing Machinery, Vereinigung von Informatikern in den USA, älteste und mit mehr als 80.000 Mitgliedern größte Informatikgesellschaft der Welt) gefunden werden kann.

  Das ist kein leichter Stoff; Edward Halls ausgezeichnetes Buch über Whorfs Theorien *The Silent Language* bietet Ihnen ein wenig Abwechslung.

✘ Erstellen Sie Ihre eigene Liste sich wiederholender Aufgaben, die Sie ungern erledigen und deren Ausführung Sie lieber einem Computer übertragen würden. Nehmen Sie diese Liste bei der Lektüre des Buchs von Zeit zu Zeit zur Hand, um zu sehen, ob Ihnen Lösungsideen kommen. Versuchen Sie, zumindest einen dieser Zeitsparer anzufertigen, bevor Sie am Ende des Buchs ankommen. Falls Sie nur ein einziges Programm schreiben und es Ihnen damit gelingt, zumindest einige Minuten in der Woche einzusparen, hat sich dieses Buch bereits bezahlt gemacht.

- Sollten Sie zur fest entschlossenen, auf Vollständigkeit bedachten Sorte Menschen zählen, könnten Sie Shunryu Suzukis *Zen-Geist, Anfänger-Geist* lesen. Eine andere Spaß-Lektüre (d.h., ich denke, sie macht Spaß!) ist Richard Wexelblats *History of Programming Languages.* Vollständige bibliographische Angaben zu allen Büchern, auf die im Text hingewiesen wird, sind in Anhang A, »Bibliographie«, zu finden. Wenn Sie fest entschlossen und auf Vollständigkeit bedacht sind, sollten Sie auch überlegen, ob Sie nicht selbst ein Buch schreiben.

Kapitel 2

Der Python-Interpreter

Calliopus gelang dann ein selterner Erwerb von Krokodilen direkt aus Ägypten, die jedoch während der Reise sehr litten und sich in der Arena als unbefriedigend heraus stellten ... Er hatte eine verirrte Python akzeptiert, die auf einem Markt von den Nachtwachen gefangen worden war.

Lindsay Davis in »Two for the Lions«

Im Verlauf dieses zweiten Kapitels werden wir erfahren, was der Python-Interpreter ist, was den Unterschied zwischen übersetzten und interpretierten Sprachen ausmacht, was das Programm IDLE ist und wie es funktioniert, und wir werden unser allererstes Python-Programm schreiben und ausführen.

2.1 Der Interpreter: eine minimalistische Einführung

Man unterscheidet zwei Hauptkategorien von Programmiersprachen: übersetzte und interpretierte. Der erste Typ ist für gewöhnlich um einiges schneller als der zweite, aber interpretierte Sprachen sind in der Regel leichter im Gebrauch und bei der Fehlersuche. *Übersetzte* (kompilierte) Sprachen sind solche, bei denen der Programmierer bedeutsame Befehle und Angaben eingibt und das Ergebnis als Dokument sichert; anschließend lässt der Programmierer ein spezielles, Compiler (Übersetzer) genanntes, Programm über die gespeicherte Datei laufen. Der Compiler liest den Quellcode (Quellcode ist das, was man in einem Editor eingibt und als Datei sichert) und übersetzt die Anweisungen, die die Programmierer lesen und

schreiben können, in Maschinensprache, die ausschließlich dazu gedacht ist, dass ein Computer sie versteht. Nachdem der Compiler die Übersetzung abgeschlossen hat, werden die Maschinenbefehle in eine besondere Datei übertragen. Bei DOS- und Windows-Rechnern muss der Name dieser Spezialdatei auf `.exe` enden, was dem Betriebssystem mitteilt, dass es sich um eine ausführbare Datei handelt. Bei Unix-Systemen bietet das Betriebssystem dem Programmierer die Möglichkeit, die Maschinensprache-Datei als ausführbare Datei zu kennzeichnen. Bei beiden Systemen kann der Programmierer oder Anwender einfach den Namen des Programms (bei DOS und Windows unter Auslassung des Suffix `.exe`, falls die Datei über ein solches verfügt) in ein Befehlszeilenfenster oder eine Befehlszeilenschnittstelle eintippen, um das Programm auszuführen. Die meisten großen Programme wie z.B. Textverarbeitungsprogramme, Internet-Browser und Datenbankanwendungen sind so erstellt.

Ein Editor ist ein Programm, mit dem man Anweisungen oder Textzeilen in ein Fenster eingeben und den Text anschließend in einer Datei abspeichern kann. Beispiele solcher Editoren sind vi unter Unix und der Windows-Editor (Notepad) unter Windows.

Die zweite Variante, interpretierte Sprachen, macht den Compiler überflüssig. Programme in einer *interpretierten Sprache* können normalerweise auf zweierlei Arten ausgeführt werden: interaktiv oder im *Stapelbetrieb*. Programme interaktiv auszuführen, ist oftmals nicht besonders bequem, denn der Programmierer startet ein Interpreter genanntes Programm (in Python heißt dieses Programm `python` oder `python.exe`) und gibt Befehle in ein Eingabefenster ein. Der Nachteil bei dieser Methode besteht darin, dass auf diese Weise getippte Programme beim Beenden des Interpreters aus dem Dasein scheiden. Im *Stapelbetrieb* gibt der Programmierer das Programm in eine Datei ein (genauso wie bei übersetzten Sprachen) und weist den Interpreter daraufhin an, die Datei auszuführen. Ein Beispiel sowohl für Unix als auch für DOS ist `python hello.py`; diese Befehlszeile befiehlt dem Betriebssystem, das Programm `python` zu starten. Sobald Python läuft, wird `hello.py` in den Speicher eingelesen und es werden die Befehle ausgeführt.

Interpreter lesen eine Datei und anstatt nun die gesamte Datei in Maschinensprache zu übersetzen, führen sie jede einzelne Zeile der Datei aus, sobald sie eingelesen wird. Dies erübrigt nicht nur die Übersetzung, sondern der Interpreter informiert den Programmierer auch unmittelbar, sobald ein Fehler auftritt; der Programmierer kann ihn dann unverzüglich korrigieren und die Ausführung des Programms fortsetzen. Dies ist ein Vorteil, da die Übersetzung und Ausführung eines Programms in der Regel weitaus länger

dauern als die interpretierte Ausführung, und wenn ein Fehler gefunden wird, können sofort Korrekturen vorgenommen werden.

Der interaktive Modus ist am nützlichsten, wenn man eine neue Technik oder einen neu gelernten Trick ausprobieren bzw. den Interpreter anweisen möchte, einige einfache Dinge zu tun, um zu sehen, wie die Sprache funktioniert. In diesen ersten Kapiteln werden Sie sich hauptsächlich mit dem interaktiven Modus beschäftigen, da Sie sich auf die Grundlagen der untersten Ebene von Python konzentrieren müssen. Sie werden daran üben und die Grundlagen erlernen, auf denen Sie später nützliche Programme aufbauen werden. Die meisten dieser Übungen werden nicht zu Resultaten für die Ewigkeit führen.

Bevor Sie den Python-Interpreter starten können, müssen Sie ihn zunächst besorgen und installieren, sofern er nicht bereits auf Ihrem Computer installiert ist. Red Hat Linux beispielsweise installiert Python direkt bei der Installation des Linux-Betriebssystems, sodass Sie Python einfach starten können. Unter Windows und anderen Linux- oder Unix-Distributionen müssen Sie Python zunächst herunterladen; vollständige Anweisungen dazu finden Sie unter `http://www.python.org/`, wo sie auch Hinweise zur gegebenenfalls nötigen Kompilierung finden. Sollte Python bereits auf Ihrem System installiert sein, stellen Sie sicher, dass IDLE ebenfalls installiert ist. IDLE ist ein spezieller, von Guido van Rossum komplett in Python geschriebener Editor; wir werden im Rahmen dieses Buchs IDLE oftmals zum Editieren von Programmen verwenden. Sollten Sie IDLE noch nicht installiert haben, sehen Sie auf der Python-Website nach. Das Windows-Installationsprogramm installiert auch IDLE, sodass Sie sich in diesem Falle keine Sorgen machen müssen.

Falls Sie die Windows-Version von Python installieren, würde ich Ihnen dringend empfehlen, den vom Installationsprogramm vorgeschlagenen Ordner zu ändern, sodass Python in einem Ordner mit dem Namen Python installiert wird. Die Installationsvoreinstellung ist ein Ordner mit dem Namen `c:\Programme\Python`, aber ich halte `c:\Python` für besser. Leerzeichen in Verzeichnisnamen können Probleme mit einigen anderen Programmen, darunter auch Python-Programmen, verursachen, wenn diese nicht mit Leerzeichen in Ordnernamen rechnen.[1] Wie auch immer, sofern Sie den voreingestellten Pfad verwenden, wird Ihnen kein größeres Unglück widerfahren und andere nützliche Pakete werden bereits wissen, wo sie sich zu befinden haben, wenn Sie das Installationsprogramm aufrufen.

1 Anm. d. Übers.: In der amerikanischen Originalversion von Windows heißt der Ordner für Programme `c:\Program Files`.

Egal, ob Sie Windows, Macintosh oder Unix/Linux benutzen: Sie müssen unbedingt das Tcl/Tk-Paket installieren. Unter Windows ist das einfach, da Sie lediglich mit »Ja« antworten müssen, wenn Sie vom Installationsprogramm gefragt werden, ob das Tcl/Tk-Paket installiert werden soll. Unter Unix ist das Ganze etwas aufwendiger, aber von Unix-Anwendern wird generell erwartet (oder verlangt), mehr über ihre Computer zu wissen als Windows- oder Macintosh-Benutzer; es könnte nötig sein, einen Guru aufzuspüren und ihn zu befragen. Verweise auf Internet-Seiten, von denen Tcl/Tk für Unix heruntergeladen werden kann, sind unter `http://www.python.org/download/` zu finden und binäre Installationspakete für Linux sind unter `http://www.andrich.net/python/` erhältlich. Bei der Lektüre dieses Buchs werden Sie auf das Tcl/Tk-Paket nicht verzichten können, da es im dritten Teil benötigt wird.

Python 1.5.2 benötigt Tcl/Tk 8.0 und wird nicht (zumindest nicht ohne eine Menge Arbeit, die Sie wahrscheinlich nicht erledigen möchten) mit Tcl/Tk 8.1 laufen. Das Windows-Installationsprogramm installiert die korrekte Version, sofern Sie der Installation von Tcl/Tk zustimmen, aber unter Unix könnte es sein, dass Sie es selbst erstellen und installieren müssen. Dieses Thema würde jedoch den Rahmen dieses Buchs sprengen.

Ich empfehle Ihnen, für Windows zwei zusätzliche Pakete herunterzuladen und zu installieren. Sie werden sie für dieses Buch nicht benötigen, aber für künftige Arbeiten werden Ihnen diese Pakete sehr gute Dienste erweisen, wenn Sie auf weitere Anwendungen und Aufgaben für Ihre Fertigkeiten in Python stoßen. Dies sind die Win32Api-Erweiterung und das PythonWin-Paket. Win32Api erlaubt den Zugriff auf dieselben Teile des Windows-Betriebssystems, mit denen auch die Programmierer übersetzter Sprachen wie C arbeiten. Das PythonWin-Paket enthält einen weiteren Editor und eine andere Entwicklungsumgebung. Manche ziehen diese IDLE vor, aber ich werde hier nicht weiter darauf eingehen.

2.2 Interaktivität und Umgebung

Nachdem Python auf Ihrem Rechner installiert ist, sind zunächst noch einige andere Schritte erforderlich, um Python lauffähig zu machen. Die meisten Installationsprogramme erledigen dies automatisch für Sie. Die Windows-Installation zum Beispiel verändert Ihre Registrierung (*Registry*), um Einstellungen hinzuzufügen, die Python benötigt, um Bibliotheken mit Methoden und Funktionen zu finden. Sollte Ihr Python bereits unter Linux vorinstalliert sein, sind die Informationen, die Python zur Ausführung und

zum Auffinden der Bibliotheken benötigt, in einer speziellen Datei mit Namen `site.py` enthalten; in diesem Fall brauchen Sie selbst keine Änderungen vorzunehmen.

In früheren Versionen von Python hätten Sie vermutlich die `PYTHONPATH`-Umgebungsvariable setzen müssen, aber wenn Sie mit Python 1.5.2 beginnen, ist dies nicht mehr nötig. Wenn Python korrekt installiert ist, werden Module (worüber Sie später mehr erfahren werden) von Python gefunden und anstandslos geladen. Sie werden `PYTHONPATH` nur unter besonderen Umständen setzen müssen, wie z.B. beim Laden von Modulen, von denen niemand sonst auf auf Ihrem Computer Kenntnis haben soll. Die meiste Arbeit, die Sie bei einem Betriebssystem haben dürften, fällt bei Windows NT an. Sie werden dem Pfad vermutlich die Verzeichnisse von Python und Tcl hinzufügen müssen, was Sie unter START/EINSTELLUNGEN/SYSTEMSTEUERUNG und Doppelklick auf das Icon SYSTEM tun können. Klicken Sie nach dem Öffnen der Systemsteuerung auf das Register UMGEBUNG. Die Registerkarte mit den Eigenschaften sollte ungefähr wie die in Abbildung 2.1 aussehen (so sieht sie auf meinem NT-System zu Hause aus).

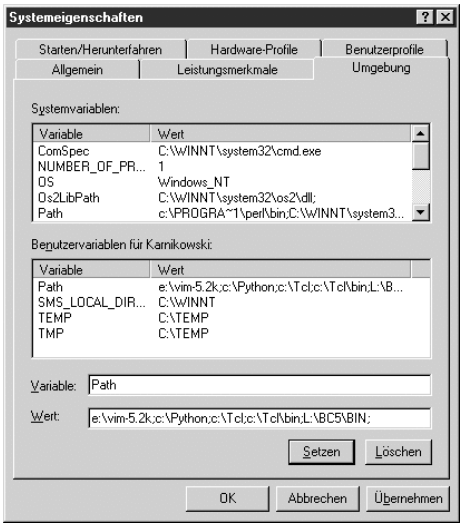


Abb. 2.1: Umgebungsvariablen setzen unter Windows NT

Beachten Sie, dass ich im Eingabefeld WERT das Verzeichnis des Python-Interpreters und der Tcl-Bibliotheken als `Tcl\bin` angegeben habe. Wenn Sie diese Ordner auf der Registerkarte UMGEBUNG hinzufügen und die Schaltfläche ÜBERNEHMEN anklicken, sollten Sie in der Lage sein, Python ohne Probleme zu starten. Damit die Änderungen wirksam werden, müssen Sie jedoch zunächst alle offenen Befehlsfenster schließen und wieder öffnen.

Wenn Sie `python` eingeben und Ihr Betriebssystem die Fehlermeldung ausgibt, dass Python nicht gefunden werden kann, muss die Umgebungsvariable `PATH` geändert werden. Zuallererst müssen Sie wissen, wo sich Python auf Ihrem System befindet, und dazu entweder Ihr System gut genug kennen, um das herauszufinden (bzw. sich daran erinnern, wo Sie das Programm installiert haben) oder Ihren System- oder Netzwerkadministrator fragen. Nehmen wir nun einmal an, Sie hätten Python gefunden und zwar in einem Verzeichnis wie z.B. `/usr/local/bin`; überprüfen Sie auch in diesem Fall, ob das entsprechende Verzeichnis in der Variable `PATH` eingetragen ist (sollte das der Fall sein, liegt der Fehler an anderer Stelle und Sie werden Hilfe benötigen, um ihn zu beheben).

Um dieses Verzeichnis auf Unix/Linux-Systemen zu Ihrem Pfad hinzuzufügen, müssen Sie an Ihrem Shell-Prompt etwas wie `PATH=$PATH:/usr/local/bin; export PATH` eingeben oder es am besten gleich Ihrer Shell-Startup-Datei hinzufügen. Diese Datei sollte etwa `.profile`, `.bashrc` oder `.cshrc` heißen und sich in Ihrem Home-Verzeichnis befinden. Bearbeiten Sie einfach diese Startdatei und ergänzen Sie die `PATH`-Anweisung um die entsprechenden Verzeichnisse. Falls Sie diese nicht ausfindig machen können, benötigen Sie möglicherweise Hilfe, aber es sollte aus dem Zusammenhang hervorgehen, was Sie ändern müssen.

Falls Sie Python unter `c:\python\python.exe` finden, müssen Sie `PATH %PATH%;c:\python;c:\Tcl\bin` in einem MS-DOS-Eingabeaufforderungsfenster eingeben. Sollten Sie Windows 95/98 verwenden, fügen Sie die obige Zeile am Ende der Datei `AUTOEXEC.BAT` ein und starten Sie Ihren Rechner neu. Nach dem Neustart wird Windows wissen, wo sich Python befindet. Abbildung 2.2 zeigt den Windows-Editor mit einer korrekt angepassten `AUTOEXEC.BAT`.

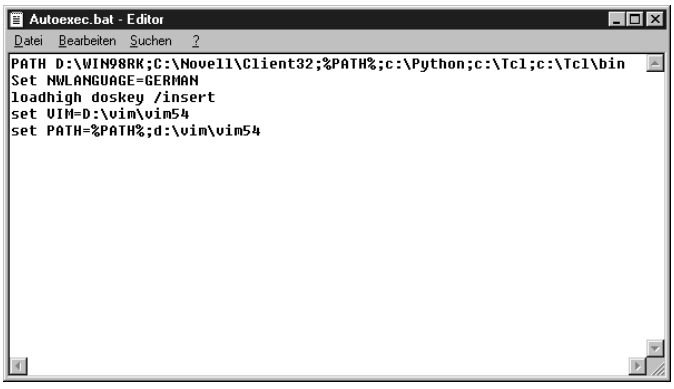


Abb. 2.2: Anpassung der Datei `AUTOEXEC.BAT`

Abschließend sollten Sie an Ihrer Windows- oder Unix-Eingabeaufforderung `python` eingeben und eine Ausgabe wie in Abbildung 2.3 sehen können.

Abb. 2.3: Was passiert, wenn Sie `python` eintippen?

Das Zeichen `>>>` heißt *Python-Prompt* – eine Zeichenfolge, die Ihnen mitteilt, dass der Python-Interpreter bereit ist, auf Ihre Befehle zu antworten. Sie können – falls gewünscht – sehen, was geschieht, wenn Sie Befehle eintippen; geben Sie `"Hallo, Welt"` ein (vergessen Sie die Anführungszeichen nicht) und drücken Sie ↵.

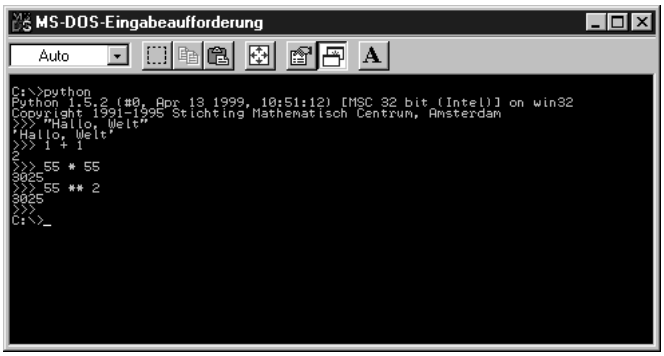


Abb. 2.4: Eingaben in den Interpreter

Um den Interpreter zu verlassen, drücken Sie unter Windows Strg+Z (möglicherweise trägt die Taste Strg auf Ihrer Tastatur die Bezeichnung Ctrl). Unter Unix und in IDLE lautet die Tastenkombination Strg+D. In beiden Fällen geben Sie diese Befehle zum Beenden ein, indem Sie die Taste Strg gedrückt halten und gleichzeitig die Taste Z bzw. D auf Ihrer Tastatur drücken. Bevor man sich mit anderem beschäftigt, schadet es nie, zu wissen, wie man ein Programm verlässt.

2.3 Die Ausführung von Skripts

Obwohl das Eintippen von Programmanweisungen Spaß machen und dabei einfach und lehrreich sein kann, ist es wenig geeignet, wenn man seine Arbeit auch speichern möchte. Im normalen Interpreter, der von einer Befehlszeile (einer DOS-Box beim PC, einem Terminalemulator-Fenster unter Unix usw.) aus ausgeführt wird, gibt es keine Möglichkeit, die Arbeit zu sichern. Interaktives Eingeben von Befehlen ist etwa so, als würde man Geld in einen Brunnen werfen: Man mag sich danach richtig gut fühlen, kann es aber nicht zurückholen.

Sie können mit einem Texteditor (wie bereits erwähnt, können Sie unter Windows immer mit dem Windows-Editor arbeiten; falls Sie jedoch einen Lieblingstexteditor haben, benutzen Sie einfach diesen) Programme eingeben und den Text in einer Datei mit der Endung bzw. Erweiterung `.py` speichern. Es ist für Python nicht nötig, die Endung `.py` zu verwenden, aber ich rate es Ihnen dringend, vor allem wenn Sie unter Windows arbeiten. Das wird das Leben leichter machen, das verspreche ich Ihnen.

Unabhängig von der verwendeten Plattform werden die Skripts stets auf dieselbe Weise ausgeführt; Sie brauchen lediglich

```
python script.py
```

an der Befehlszeile (mit welcher auch immer Sie arbeiten) einzugeben. Und genau damit beschäftigen wir uns im nächsten Abschnitt.

2.4 Hallo, Python!

Obwohl ich zum Schreiben meiner Python-Programme gern mit vi arbeite, sollten Sie einen vertrauten Editor benutzen. Der mit Windows ausgelieferte Editor ist für eine Weile genau richtig, später werden Sie allerdings die vielen nützlichen Eigenschaftenvon ausgefeilteren Editoren, wie z.B. automatische Einrückung, zu schätzen wissen.

Zeilen, die von Menschen lesbare Anweisungen in einer speziellen Programmiersprache enthalten, werden *Code* genannt.

Öffnen Sie Ihren Editor und geben Sie den folgenden Code ein:

```
print "Hallo, Welt!"
print "Ade, du schoene Welt!"
```

(Achten Sie darauf, dass die beiden Zeilen am linken Rand Ihres Textdokuments beginnen; es sollte kein Leerzeichen vor dem Schlüsselwort `print` stehen.) Speichern Sie diese Zeilen in einer Textdatei mit Namen `hello-`

`world.py`, öffnen Sie eine DOS-Box (oder was immer Sie verwenden) in demselben Verzeichnis, in dem Sie die Datei gespeichert haben, und schon können Sie Ihr erstes echtes Python-Programm starten.

Whitespace sind Tabulatoren, Leerzeichen und Wagenrückläufe (auch als *Zeilenendschaltung* bekannt). Beim Umgang mit Python müssen Sie besonders auf den Whitespace achten, denn die Leer- und Tabulatorzeichen am Zeilenanfang bestimmen die Einrückungstiefe der Zeile und alle Zeilen mit gleicher Einrückungstiefe gehören zusammen. Andere Sprachen verwenden besondere Zeichen oder spezielle Schlüsselwörter, um zusammengehörige Zeilen zu kennzeichnen, aber Leerzeichen oder Tabulatoren auf gleicher Ebene sind besser lesbar. Andere Sprachen verwenden ebenfalls besondere Zeichen, um das Ende der Anweisung zu kennzeichnen, aber Python verwendet den üblichen Zeilenvorschub.

Ein *Schlüsselwort* ist ein Wort innerhalb einer Programmiersprache, welches die Sprache anweist, etwas zu tun; `print` ist ein Schlüsselwort in Python. Sie können Schlüsselwörter in Ihren Programmen nicht als Variablennamen oder Funktionsnamen verwenden (später in diesem Buch erfahren Sie mehr über Variablen und Funktionen).

Geben Sie

```
python helloworld.py
```

in Ihr Befehlsfenster ein und schauen Sie, was passiert.

Sie sollten eine Ausgabe wie in Abbildung 2.5 erhalten.

Abb. 2.5: Hallo, Python!

Sie haben gerade Ihr allererstes Python-Programm ausgeführt – das traditionelle »Hallo, Welt«-Programm. Es wird allerdings für eine Weile das letzte Programm sein, das Sie starten; im nächsten Kapitel werden wir uns darauf konzentrieren, die mathematischen Grundlagen mit Python kennen zu lernen, und wir werden dazu die interaktiven Fähigkeiten des Interpreters einsetzen. Dazu werden wir dieses Befehlsfenster allerdings nicht weiter benutzen, da es etwas Besseres mit Namen IDLE gibt.

IDLE steht für Integrated DeveLopment Environment (Integrierte Entwicklungsumgebung) und wurde von Pythons Schöpfer, Guido van Rossum, unter Verwendung von Pythons grafischer Benutzerschnittstellen-Komponente *tkinter*, vollständig in Python geschrieben. Wir werden im letzten Drittel dieses Buchs mehr über *tkinter* erfahren, aber die Verwendung von IDLE wird Ihnen einen Vorgeschmack darauf geben, welche Art von Projekten Sie mit tkinter in Angriff nehmen können.

Sollten Sie Ihren Pfad so verändert haben, dass er sowohl das Python- als auch das Tcl-Verzeichnis enthält, empfiehlt es sich, eine Desktop-Verknüpfung anzulegen, damit IDLE immer verfügbar ist. Dies ist unter Windows sehr einfach und unter Linux nicht allzu schwer. Wie es unter Windows geht, werde ich kurz beschreiben; wenn Sie Linux oder Unix benutzen, ziehen Sie am besten Ihre Systemdokumentation zu Rate. Notfalls müssen Sie in einem Terminalemulator-Fenster `idle` eingeben, um IDLE zu starten.

Um herausfinden, wo sich IDLE befindet, müssen Sie das Verzeichnis von Python kennen; da das Python-Verzeichnis in Ihrem Pfad stehen sollte, dürfte das kein Problem darstellen. Auf meinen Windows-Systemen befindet sich Python immer in `c:\Python`. Öffnen Sie ein Windows-Explorer-Fenster für das Python-Verzeichnis. Innerhalb des Python-Verzeichnisses sollte auch ein Ordner namens TOOLS zu finden sein; öffnen Sie diesen Ordner, in dem sich ein weiterer Ordner mit Namen IDLE befinden sollte. In diesem IDLE-Ordner sollten Sie ein Icon für `idle.py` und direkt daneben eines für `idle.pyw` finden und genau das Letzte benötigen wir. Klicken Sie mit der rechten Maustaste auf das Icon `idle.pyw` und wählen Sie im Kontextmenü VERKNÜPFUNG ERSTELLEN; Windows sollte jetzt ein Icon mit dem Namen VERKNÜPFUNG MIT `idle.pyw` erstellen. Markieren Sie dieses Icon und ziehen Sie es auf den Desktop; falls Sie möchten, können Sie den Namen und auch das Icon ändern (obwohl mir persönlich das niedliche, kleine, grüne Python-Icon gut gefällt). Sie können eine Desktop-Verknüpfung auch noch auf eine andere Art erstellen: Wählen Sie dazu im Menü START den Menüpunkt EINSTELLUNGEN und klicken Sie anschließend auf TASKLEISTE & STARTMENÜ... Klicken Sie auf die Registerkarte PROGRAMME IM MENÜ "START" und auf die Schaltfläche ERWEITERT... Im Verzeichnisbaum von PROGRAMME sollten Sie einen Eintrag für Python 1.5 sehen; wählen (oder doppelklicken) Sie diesen und die rechte Seite des Explorer-Fensters müsste vier Icons enthalten, von denen eines IDLE (Python GUI) heißt. Klicken Sie mit der rechten Maustaste auf das Icon und wählen Sie VERKNÜPFUNG ERSTELLEN. Ziehen Sie die erstellte Verknüpfung auf Ihren Desktop und ändern Sie ihren Namen bzw. passen Sie sie Ihren Bedürfnissen an. Wenn Sie mit der Position auf dem Desktop und dem Erscheinungsbild Ihrer IDLE-Verknüpfung zufrieden sind, starten Sie IDLE mit einem Doppelklick auf das Icon. Das Hauptfenster sollte erscheinen und aussehen wie in Abbildung 2.6.

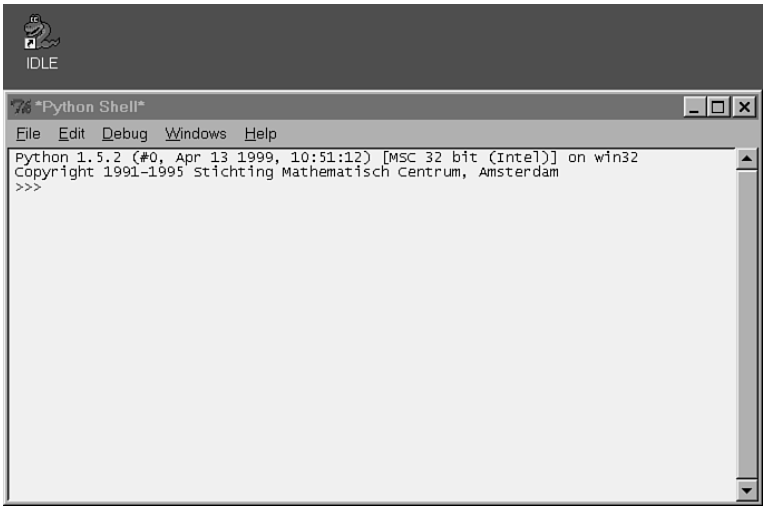


Abb. 2.6: Das IDLE-Hauptfenster

Sie können IDLE auf dreierlei Arten verlassen: indem Sie FILE/EXIT aus der Menüleiste wählen, im Hauptfenster Strg+D eingeben oder Strg+Q in irgendein IDLE-Fenster eingeben, das gerade den Fokus besitzt.[2] Versuchen Sie, das Programm auf eine dieser drei Arten zu verlassen, und starten Sie IDLE neu, um die anderen Möglichkeiten auszuprobieren. Verwenden Sie die Methode, die Ihnen am besten gefällt. Jetzt können wir uns fortgeschritteneren Aufgaben zuwenden; im nächsten Kapitel werden wir damit beginnen, uns mit einfacher Mathematik in Python unter Benutzung von IDLE zu beschäftigen.

2.5 Zusammenfassung

In diesem zweiten Kapitel haben Sie den Unterschied zwischen übersetzten und interpretierten Programmiersprachen kennen gelernt. Sie wissen, wie Python und Tcl installiert werden, wie der Pfad gesetzt werden muss, damit Python gefunden und auf einfach Weise in einer DOS-Box gestartet oder in einem Terminalemulator-Fenster ausgeführt werden kann. Sie haben erfahren, wie man einfache Skripts ausführt, wie eine Verknüpfung zu IDLE angelegt und wie IDLE gestartet wird. Und natürlich haben Sie auch erfahren, wie man IDLE beendet, was immer eine gute Sache ist.

2 Anm. d. Übers.: Unter Windows haben Sie auch noch die Möglichkeit, IDLE mit der Standard-Tastenkombination Alt+F4 zu beenden.

2.6 Workshop

2.6.1 F&A

F: Auf welchen Plattformen läuft IDLE?

A: Unix, Windows und Macintosh, denn nur diese Plattformen unterstützen *tkinter*.

F: Was mache ich, wenn IDLE nicht läuft?

A: Für dieses Problem gibt es verschiedene Ursachen, sodass sich eine Antwort schwer geben lässt. Versuchen Sie, Hilfe von jemandem zu bekommen, der mehr über Computer weiß als Sie. Falls Ihnen das nicht gelingt, besuchen Sie am besten die Python-Homepage unter `http://www.python.org/`. Suchen Sie nach Verweisen auf die Mailing-Liste und tragen Sie sich in die Python-Tutoren-Mailing-Liste ein. Diese Liste ist speziell dazu gedacht, Anfängern wie Ihnen über die ersten Hürden in der Python-Programmierung zu helfen.

F: Kann ich Python-Programme ausführen, ohne python script.py eingeben zu müssen?

A: Ja. Unter Unix können Sie *chmod* verwenden, um Ihre Skript-Dateien direkt ausführbar zu machen. Geben Sie `chmod +x script.py` ein; damit teilen Sie Unix (oder Linux) mit, dass die Datei wie ein Programm behandelt werden soll. Sie werden Ihrem Programm allerdings eine spezielle erste Zeile hinzufügen müssen, die z.B. `#!/usr/bin/env python` lauten könnte. Um die korrekte Syntax herauszufinden, schlagen Sie in der Manual Page für `env` nach. Danach sollten Sie in der Lage sein, an der Befehlszeile einfach `script.py` einzugeben. Auch unter Windows NT gibt es dafür einen Weg, der allerdings viel komplizierter ist; auf der Python-Homepage unter `http://www.python.org/` können Sie entsprechende Dokumente finden, die genaue Anweisungen enthalten.

2.6.2 Quiz

1. Was ist IDLE[3]?

   a) Ein südindischer Kloß.

   b) Eine integrierte Interpreter-Umgebung, die komplett in Python erstellt wurde.

---

3 engl. *idle*, dt. untätig, müßig, unbeschäftigt, arbeitslos, außer Betrieb, stillstehend, im Leerlauf, unproduktiv, brachliegend, tot, ruhig, still, ungenutzt, faul, träge, ...

c) Ihr Zustand am Sonntagmorgen.

d) Was Ihre Katzen ständig sind.

2. Was ist los, wenn Sie `python` eingeben und folgende Nachricht erhalten: »`The name specified is not recognized as an internal or external command, operable program, or batch file`«?

   a) Sie haben Ihren Pfad nicht korrekt gesetzt.

   b) Jemand hat Ihr Python gestohlen.

   c) Der Computer ist einfach stur; probieren Sie den Befehl erneut.

   d) Sie benutzen Windows und Bill Gates möchte nicht, dass Sie Python starten.

3. Was bedeutet das Zeichen `>>>`?

   a) Voodoo.

   b) Das »Dreimal spucken«-Zeichen.

   c) Der Python-Prompt, der Ihnen mitteilt, dass Python Ihre Befehle eifrig erwartet.

   d) Schneller Vorlauf.

4. Was geschieht, wenn Sie einen Befehl in den Python-Interpreter eingeben, den dieser nicht kennt?

   a) Nichts.

   b) Ihr Computer wird neu gestartet.

   c) Es wird eine bestimmte Traceback-Nachricht ausgegeben, die mit `NameError:` beginnt und auf die das Wort folgt, das Sie eingegeben haben und das nicht erkannt worden ist. Wenn Sie zwei oder mehr Wörter eingeben, führt dies zu einem `SyntaxError:`.

   d) Sie müssen den Befehl zur Strafe 500 Mal korrekt eingeben.

2.6.3 Antworten

1. b. IDLE ist eine integrierte Interpreter-Umgebung, die vollständig in Python geschrieben wurde.[4]

2. a. Sie haben Ihren Pfad nicht korrekt gesetzt, wenn sich die DOS-Eingabeaufforderung darüber beschwert, dass sie den von Ihnen eingegebenen Namen nicht kennt.

---

4 Anm. d. Übers.: Eric *Idle* war übrigens Mitglied von Monty Python's Flying Circus.

3. c. Das Zeichen >>> heißt Python-Prompt; achten Sie darauf, dass Sie diesen von einer normalen Eingabeaufforderung (wie z.B. dem DOS- oder Linux-Prompt) unterscheiden.

4. c. Wenn Sie einen unbekannten Befehl in den Python-Interpreter eingeben, führt dies zu einer Traceback-Fehlermeldung. Diese Meldungen sind normalerweise, aber nicht immer, ziemlich informativ. Zum Glück führt Python keinen Neustart Ihres Rechners durch und stürzt auch nicht ab. Ich arbeitete einmal an einem Rechner, auf dem regelmäßige Diagnoseprogramme ausgeführt werden mussten, und um in den speziellen Diagnosemodus zu gelangen, musste ich ungefähr 15 Befehle exakt richtig eingeben. Nachdem das geschafft war, hatten all diese Diagnoseprogramme eine äußerst wählerische Syntax, die etwa 30 durch Kommata getrennte Befehlszeilen-Parameter erforderte. Falls ein Parameter nicht gebraucht wurde, musste man das Komma trotzdem eingeben, nur um der Maschine mitzuteilen, dass der Parameter nicht gebraucht wurde. Die meisten Parameter wurden nicht gebraucht. Wenn man ein Komma zuviel oder eines zuwenig eingab, wurde der Diagnosemodus beendet und der Rechner hing und musste neu gestartet werden. Es war kein sehr beliebter Rechner.

2.7 Übungen

- ✘ Spielen Sie mit dem Interpreter, indem Sie ihn in einer DOS-Box ausführen; probieren Sie IDLE aus. Schauen Sie, ob Sie Unterschiede im Verhalten der beiden feststellen können.

- ✘ Besuchen Sie die Python-Homepage, `http://www.python.org/`, und suchen Sie die Mailing-Listen. Tragen Sie sich in die Tutoren-Mailing-Liste ein. Ich bin Mitglied dieser Liste und viele weitere erfahrene Leute sind es auch. Wir wollen Ihre Fragen beantworten und Ihnen über die Stolpersteine beim Lernen von Python hinweghelfen. Wenn Sie uns dann den Edelstein aus der Hand schnappen können, wird es Zeit für Sie zu gehen, Grashüpfer. Oder nehmen Sie dann zumindest einen Platz unter den Lehrern ein.

Kapitel 3

Einfache Arithmetik mit Python

Einfach ist besser als komplex. *Tim Peters*

In diesem Kapitel werden wir Python und IDLE benutzen, um uns mit einfacher Arithmetik zu beschäftigen. Das sind sehr einfache Ideen, aber sie bilden die Grundlage für die vielen, komplexeren Konzepte, die folgen werden. Obwohl es nicht im Mindesten nötig ist, ein Mathe-Zauberer zu sein, um zu programmieren, wäre es gut, wenn Sie sich noch an einige Grundlagen der Schulmathematik erinnern könnten. Auch wenn Sie sich nicht mehr so genau erinnern können, werden Sie keine Schwierigkeiten haben, falls Sie wissen, wie Sie mit einfachen Formeln umgehen.

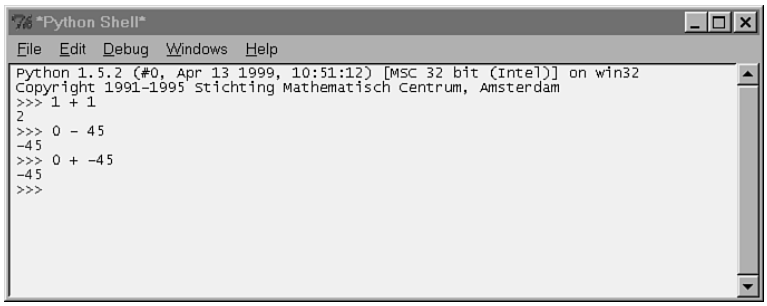
Diejenigen unter Ihnen, die nicht gut in Mathematik waren, sollten Folgendes zur Kenntnis nehmen: Ich hatte Algebra in der High School und wäre beinahe durchgerasselt (ich hatte einen schlechten Lehrer, aber was noch wichtiger war: ich habe nicht aufgepasst). Es packte mich erst viel später in einem Einführungskurs in Elektronik bei der Armee. Zwei Jahre nach diesem Kurs habe ich dann selbst unterrichtet.

3.1 Addition und Subtraktion

Jeder beginnt mit Addition und Subtraktion, denn dies sind die Grundsteine der Mathematik. Ich bin sicher, dass Sie die Regeln kennen, und es gibt keine wirklichen Unterschiede zwischen den Regeln, die Sie kennen, und den mathematischen Regeln in Python (oder irgendeiner anderen Programmiersprache).

Wir werden ein paar Additionen und Subtraktionen ausprobieren. Starten Sie den Python-Interpreter, indem Sie `python` in die Befehlszeile eingeben und geben Sie dann `1 + 1` in die Befehlszeile ein. Falls Python nicht mit `2` reagiert, haben Sie ein wirkliches Problem! Jetzt probieren Sie einige kompliziertere Probleme wie die Addition einiger großer Zahlen und die Addition positiver und negativer Zahlen. Es hilft Ihnen, wenn Sie sich die Gesamtheit der Zahlen (Integer) auf einer unendlichen Linie mit Null im Ursprung vorstellen, bei der die Zahlen Punkten entsprechen, die Sie durch Vor- oder Rückwärtsbewegung erreichen können, indem Sie die Addition und Subtraktion anwenden. Nehmen Sie zum Beispiel an, dass Sie sich an Position 0 befinden und zur Position -45 gehen möchten. Offensichtlich müssen Sie sich um 45 Einheiten rückwärts bewegen, aber Sie können auf zwei Wegen dorthin gelangen: Sie können eine negative Zahl addieren oder Sie können eine positive Zahl subtrahieren. Probieren Sie es aus. Sie sollten das Folgende im Interpreter-Fenster sehen (siehe Abbildung 3.1).

Abb. 3.1: 1 + 1



All das ist recht einfach, aber was passiert, wenn Sie wirklich große Zahlen addieren und subtrahieren? Mit Papier und Bleistift ist das kein Problem. Egal, wie groß die Zahlen sind, mit denen Sie arbeiten, mit etwas Geduld werden Sie letztlich zur korrekten Antwort kommen. Aber wenn Sie einen modernen wissenschaftlichen Taschenrechner oder einen Chinesischen oder Japanischen Abakus verwenden (siehe Abbildung 3.2), werden Ihnen die Stellen ausgehen, sobald Sie immer größere Zahlen addieren.

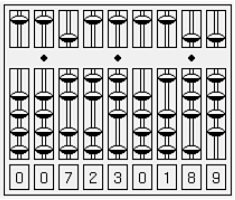


Abb. 3.2: Ein (computerisierter) japanischer Abakus

Beachten Sie, dass »Stellen« auf einem Abakus wirklich leicht zu erkennen sind. In einem Computer sind sie ein wenig schwieriger zu sehen, denn es gibt dort keine Eins-zu-Eins-Entsprechung zwischen der internen Darstellung von Zahlen, die der Computer benutzt, und der Form, wie Computer Ihnen diese Zahlen üblicherweise zeigen. Die meisten Computer verwenden 32 »Bit«, um Zahlen intern darzustellen, und eines dieser Bits dient üblicherweise dazu, das Vorzeichen – Plus oder Minus für eine positive bzw. negative Zahl – zu speichern; somit bleiben nur 31 Bit für den Zahlenwert übrig. Daher lautet die größtmögliche Integer-Zahl in Python 2.147.483.647 (zwei Milliarden plus Kleingeld).

Sie werden den Ausdruck bereits zuvor gehört haben, aber wir wollen ihn wiederholen, um ganz sicherzugehen. Ein *Bit* ist eine einzelne Ein-Aus-Zahl; sie kann entweder die Zahl 0 oder die Zahl 1 repräsentieren. Nichts weiter. Um größere Zahlen darstellen zu können, ist es erforderlich, diese Bits zu verketten, und das machen moderne Computer. Sie verketten ihre Bits zu *Bytes* (8 Bit) und *Words* (üblicherweise 32 Bit), um sie leichter handhabbar und verständlich zu machen.

Abbildung 3.3 zeigt, was geschieht, wenn Sie versuchen, Zahlen in der Nähe dieser 32-Bit-Grenze auszugeben.

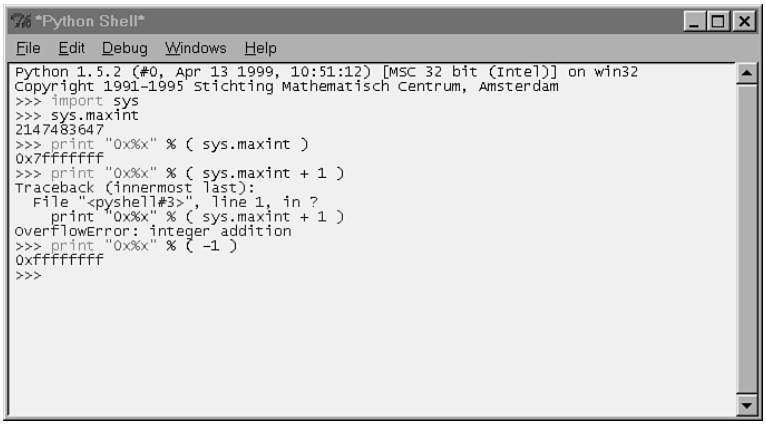


Abb. 3.3: 32-Bit-Zahlen

Wie Sie anhand dieser Abbildung erkennen können, ist es nur dann möglich, alle 32 Bit einer normalen Integer-Zahl vorzeichenlos zu nutzen, wenn man die Werte *hexadezimal* ausgibt; die Regeln zur Ausgabe hexadezimaler Zahlen legen fest, dass eine derartige Darstellung *vorzeichenlos* ist, sodass alle Bits zu sehen sind. Beachten Sie auch, dass Python einen besonderen Namen für die größte Zahl mit Vorzeichen hat: *maxint*. Dieser Wert befindet sich in einem speziellen Modul mit Namen `sys` und um von Ihren Programmen auf `maxint` zugreifen zu können, muss das Modul `sys` importiert werden. Sie werden später mehr über das *Importieren* lernen, aber für diesen Moment können Sie sich einfach vorstellen, dass Sie alles in Ihr Programm übernehmen, was es in dem Modul gibt.

Das Hexadezimal-System ist eine Methode, Bits so zu gruppieren, dass Menschen sie sich leichter bildlich vorstellen können. Die Binärzahl 1111 entspricht vier Bit, aber Programmierer schreiben stattdessen für gewöhnlich »F«. Hexadezimalsystem bedeutet »Sechzehnersystem«, weshalb hexadezimale Ziffern von 0-F reichen – im Dezimalsystem ausgedrückt von 0–15. Wenn Sie bei Freunden und Ihrer Gattin oder Ihrem Gatten als absoluter Computerfreak gelten möchten, nummerieren Sie Ihre Videokassetten hexadezimal und beginnen Sie bei Kassette 0.

Nun gut, lassen Sie uns sehen, was in Python passiert, wenn man eine wirklich große Zahl zu einer kleinen addieren möchte. Was passiert, wenn wir beispielsweise 9999999999 zu 1 addieren, zeigt Abbildung 3.4.

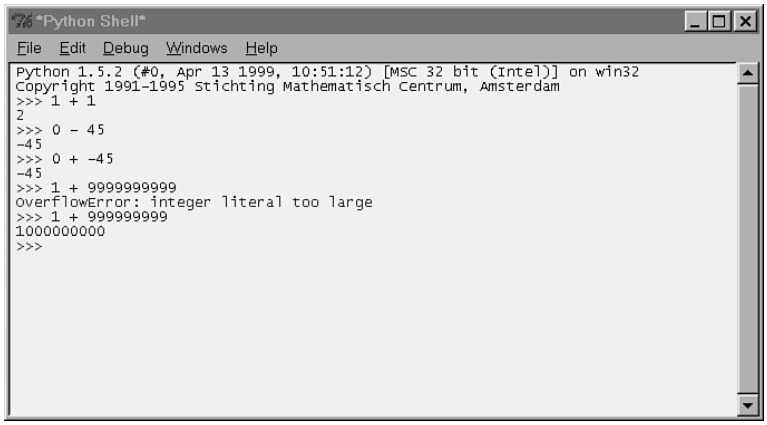


Abb. 3.4: 1 + ganz viel

Hmmm. Was hat das zu bedeuten? Einfach das Folgende: Die von uns addierte Zahl war zu groß, als dass Python mit ihr umgehen könnte. Python hat versucht, 9999999999 in die interne Darstellung umzurechnen und die Stellen reichten dazu nicht aus. Wir erhalten dasselbe Ergebnis, wenn wir versuchen, von einer großen Zahl zu subtrahieren. Aber was machen wir jetzt? Nun, es gibt eine sehr einfache Lösung: Wir müssen Python lediglich sagen, dass eine große Zahl wie eine große Zahl behandelt werden soll. Wir tun dies, indem wir Python einen Tipp geben: Hängen Sie einfach ein *L* an jede lange Zahl an, falls Sie denken, sie führe zu einem Überlauf (womit das Ausgehen von Stellen gemeint ist). Abbildung 3.5 zeigt ein Beispiel.

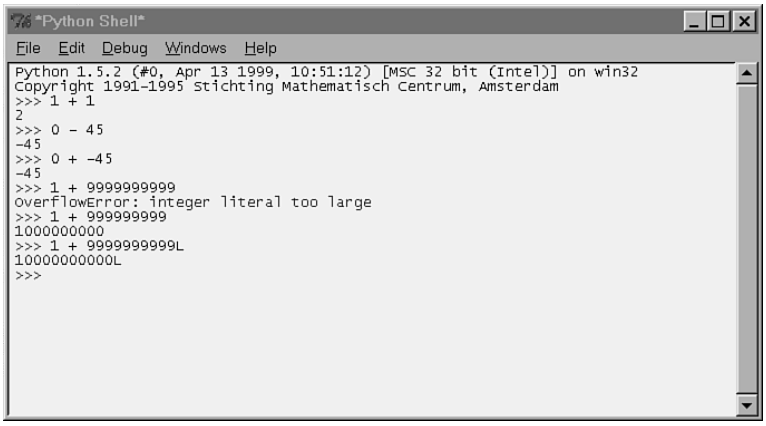


Abb. 3.5: Nochmal 1 + ganz viel

Das Anhängen eines *L* an jede große Zahl sagt Python »diese Zahl ist *lang*«. Eine *lange* Zahl kann von jeder erforderlichen Länge sein; die Stellen, oder die Stäbe auf dem Abakus, sind allein durch die Größe des Speichers in Ihrem Computer und Ihre Geduld begrenzt. Wir werden jetzt eine wirklich große Zahl ausprobieren: Kopieren Sie die nächste Zeile und fügen Sie sie ins Hauptfenster von IDLE ein (unter `http://na-tzul.pauahtun.org/TYPython/Chapter3/googol.txt` finden Sie eine Textdatei mit der Zahl, Sie können aber auch einfach 1 mit 100 Nullen eingeben …):

```
1000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000L
```

Setzen Sie dahinter ein Pluszeichen (+) ein und fügen Sie dieselbe Zahl erneut ein. Drücken Sie ↵. Das Ergebnis ist in Abbildung 3.6 dargestellt.

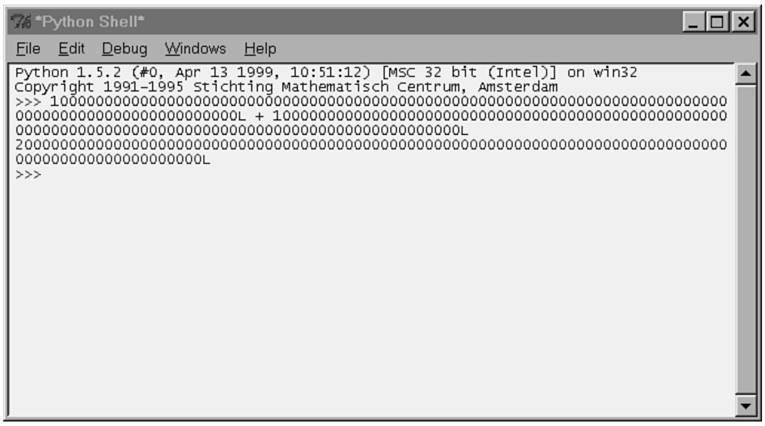


Abb. 3.6: Addition zweier wirklich großer Zahlen

Eine Eins gefolgt von 100 Nullen heißt *Googol*. Sie wurde vom Amerikanischen Mathematiker Edward Kasner erfunden und von dessen neunjährigem Neffen Milton Sirotta auf diesen Namen getauft.

Sie können reguläre Integer und lange Integer problemlos addieren. Pythons Regeln sagen dazu, dass, wann immer Sie arithmetische Funktionen mit zwei Operanden ausführen, der kleinere Operand zum Typen des größeren »erweitert« wird. Daher wird Python, wenn Sie 1 + 1000000000000000000000000L addieren, 1 automatisch in 1L umwandeln.

HINWEIS

Operatoren wie »+« und »-« arbeiten mit *Operanden*. So bedeutet der Ausdruck »1 + 4« z.B. »wende den Plus-Operator auf die Operanden 1 und 4 an«.

Sie werden sich ebenfalls mit einem dritten Zahlentyp zu beschäftigen haben: *Gleitkommazahlen*. Das sind Zahlen mit einem Dezimalkomma[1] wie zum Beispiel 1,1 und 3,14159265359. Die Addition und Subtraktion dieser Zahlen liefert Ihnen häufig Ergebnisse, die Sie vielleicht nicht erwartet hätten; in Abbildung 3.7 zum Beispiel lautet die Antwort, mit der Sie vielleicht rechnen würden 0,00001.

1 Anm. d. Übers.: Da im englischen Sprachraum bei Dezimalzahlen nicht das Komma, sondern der Punkt verwendet wird, müssen Sie bei der Eingabe in Python stattdessen einen Dezimalpunkt verwenden (wie auch in allen anderen Programmiersprachen – das ist übrigens etwas, das Sie beim Erlernen von Python nicht vergessen dürfen).

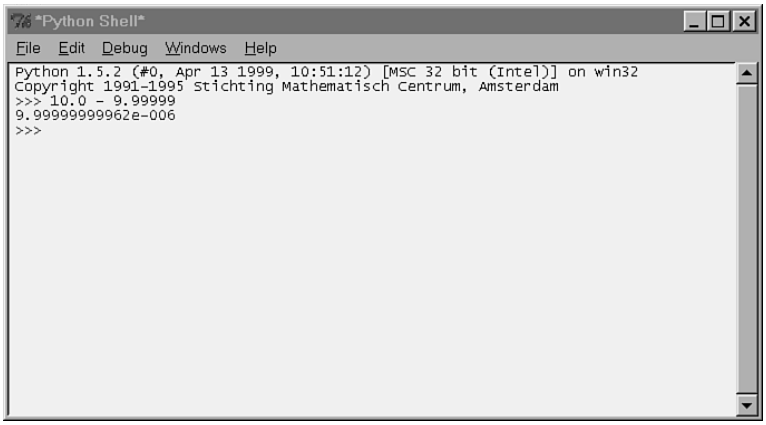


Abb. 3.7: Subtraktion von Gleitkommazahlen

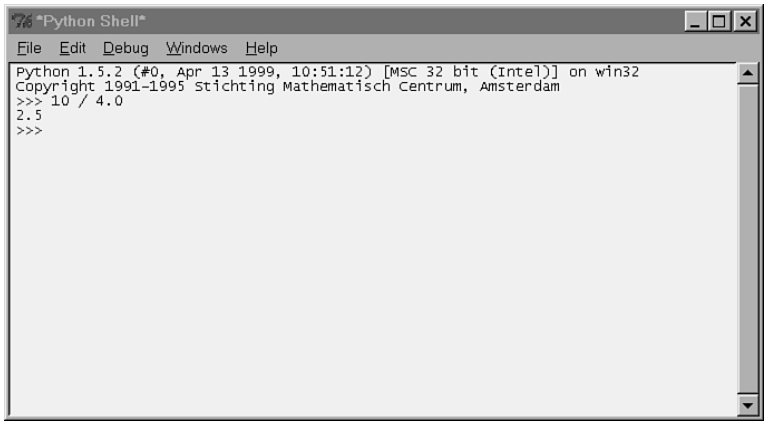
Was Sie stattdessen erhalten, wird als »wissenschaftliche Schreibweise« bezeichnet, die wir in einem späteren Abschnitt dieses Kapitels ausführlicher besprechen werden. In der Zwischenzeit werden wir uns mit anderen Themen beschäftigen.

3.2 Multiplikation, Division und Modulo

Im vorhergehenden Abschnitt haben Sie die Addition und Subtraktion mit normalen Integer-Zahlen, langen Integer-Zahlen und Gleitkommazahlen kennen gelernt. In diesem Abschnitt werden Sie unter Benutzung derselben Zahlentypen Multiplikation, Division und etwas, wovon Sie vielleicht noch nichts gehört haben – *Modulo*, kennen lernen.

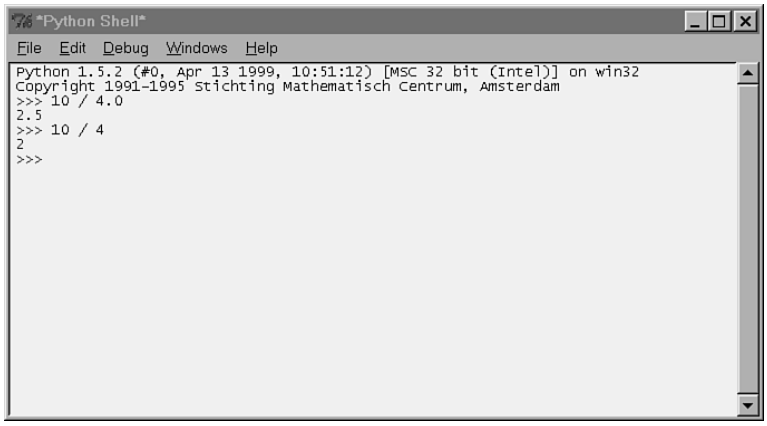
Die Regeln für diese Operationen unterscheiden sich in Python nicht wesentlich von denen, die Sie bereits kennen, aber Sie müssen sich im Klaren darüber sein, dass der Operanden-Typ starken Einfluss darauf hat, wie Python die Zahl behandelt. Zum Beispiel führt 10 geteilt durch 4,0, wie in Abbildung 3.8 gezeigt, zum erwarteten Ergebnis 2,5.

Abb. 3.8: Division



Aber in Abbildung 3.9 erhalten wir etwas anderes.

Abb. 3.9: Eine weitere Division



Der Unterschied besteht darin, dass wir Python im ersten Fall durch Benutzung des Dezimalpunkts beim Operanden 4,0 mitgeteilt haben, dass uns die Dezimalstellen wichtig sind. Im zweiten Fall haben wir Python durch Auslassung des Dezimalkommas ausdrücklich mitgeteilt, dass sie uns nicht wichtig sind. Und Python ließ die 0,5, die wir nicht benötigen, entgegenkommend aus. Auf diese Weise findet Python wie bei Addition und Subtraktion auf Grundlage der Informationen, die Sie ihm über die Operanden geben, das Richtige heraus. Dies bedeutet auch, dass wir bezüglich der Operandentypen vorsichtig sein müssen; es kann leicht passieren, dass man an einer wichtigen Stelle ein Dezimalkomma vergisst, was dann jedoch zu unerwarteten Ergebnissen führt.

Die Leichtigkeit, mit der man unerwartete oder unbeabsichtigte Ergebnisse erhalten kann, war schon immer ein ständiger Diskussionspunkt in der Python-Mailing-Liste. Manche befürworteten einen speziellen Operator, der sich vom gegenwärtig für die Division verwendeten »/« unterscheidet, um Python ausdrücklich anzugeben, dass eine reine Integer-Division (in Abbildung 3.9 dargestellt) oder eine reine Gleitkommadivision (Abbildung 3.8) durchgeführt werden soll. Zwei Vorschläge waren »/.« für reine Gleitkommadivision und »//« für reine Integer-Division. Keiner dieser Vorschläge, und ich lasse eine große Menge anderer aus, fand Anklang bei Guido, aber er sagt, dass er im Hinblick auf zukünftige Versionen von Python aufgeschlossen bleibt.

Ebenso wie bei der Addition und Subtraktion sind lange Integer-Zahlen leicht zu multiplizieren und zu dividieren (siehe Abbildung 3.10).

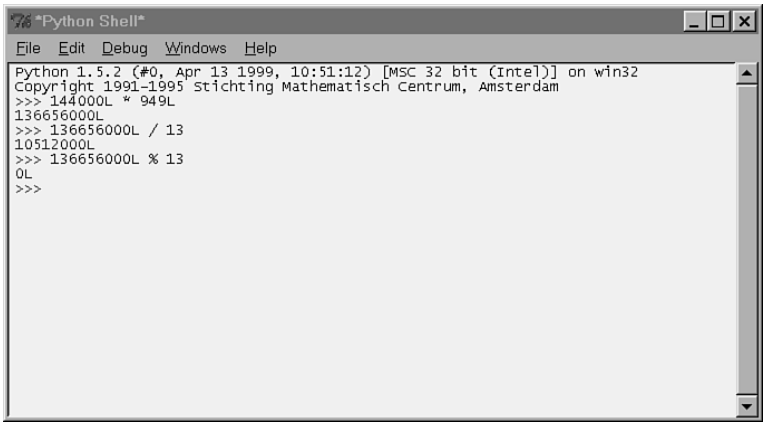


Abb. 3.10: Division und Multiplikation mit langen Integer-Zahlen

Beachten Sie, dass bei der dritten Eingabe in der obigen Abbildung ein neuer Operator auftaucht: der *Modulo*-Operator % (Restoperator). Dieser Operator führt eine Ganzzahldivision mit normalen oder langen Integer-Zahlen durch und verwirft das übliche Ergebnis. Die vom Modulo-Operator zurückgegebene Antwort ist der Rest, der bei einer Ganzzahldivision übrig bleibt und sich im vorausgegangenen Fall, 136656000L % 13, auf Null beläuft (denn 13 teilt 136.656.000 glatt und ohne Rest). Der Modulo-Operator ist sehr praktisch, vor allen Dingen bei kalendarischen Berechnungen; Schaltjahrrregeln gibt es in praktisch jedem Kalender und solche ohne Schaltjahr basieren in der Regel noch stärker auf Modulo-Arithmetik. Der Maya-Kalender ist ein erstklassiges Beispiel für Letzteres, aber auch unser eigener Gregorianischer Kalender hat natürlich Schaltjahrregeln. Der Julianische Kalender, aus dem sich der Gregorianische ableitet, besitzt eine sehr

einfache Schaltjahrregelung: Jedes durch 4 teilbare Jahr ist ein Schaltjahr. Die Gregorianische Regel modifiziert dies, indem sie sagt, dass Jahrhunderte (1700, 1900 usw.) nur dann Schaltjahre sind, wenn sie sich ohne Rest durch 400 teilen lassen. Abbildung 3.11 zeigt, wie man herausfindet, ob 2000 ein Schaltjahr ist.

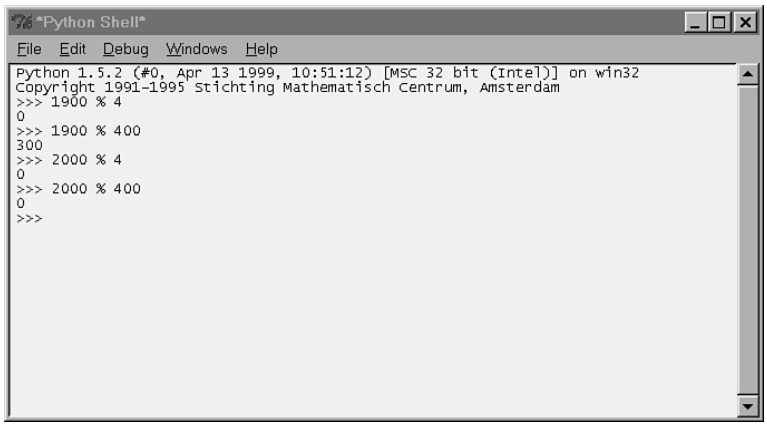


Abb. 3.11: Schaltjahre und Jahrhunderte

Es gibt eine besondere Funktion, die Sie verwenden können, wenn Sie sowohl das Ergebnis als auch den Rest einer Division benötigen: `divmod(x,y)` gibt sowohl `x / y` als auch `x % y` aus. Sie können dies in einem IDLE-Fenster ausprobieren, indem Sie `print divmod(53,13)` eingeben und sich das Ergebnis anschauen. Die Schreibweise `(4, 1)` dient speziell für Gruppen, die wir später im ersten Teil dieses Buchs kennen lernen werden.

3.3 Rundung, floor() und ceil()

Die Themen, die wir in diesem Abschnitt besprechen, betreffen ausschließlich Gleitkommazahlen. Wann immer Sie eine mathematische Operation mit Gleitkommazahlen ausführen, müssen Sie die Grenzen der zugrunde liegenden Computerdarstellung dieser Zahlen im Hinterkopf haben. Computer verarbeiten Zahlen nur *binär,* d.h. im Zweiersystem. Im Zweiersystem gibt es nur zwei Ziffern: 0 und 1. Dies ist ein ideales Zahlensystem für Computer, denn 0 und 1 können durch eine einfache Ein/Aus-Schaltung dargestellt werden: Glühbirnen, Kippschalter, Vakuumröhren und Transistoren sind hierfür allesamt Beispiele. Die Umrechnung von Ganzzahlen oder Integerzahlen von einem Stellenwertsystem in ein anderes ist einfach und Com-

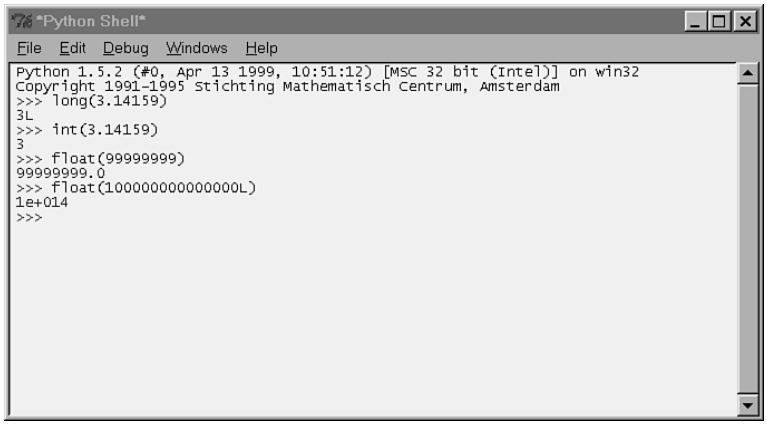
puter machen das bereits seit Jahren. Und darin sind sie auch sehr gut. Wenn Integer von einem Stellenwertsystem in ein anderes umgerechnet werden, geht keine Information verloren. Die Umrechnung von einem Stellenwertsystem in ein anderes kann immer eins zu eins durchgeführt werden; dabei ist alles eindeutig. Die Ziffern im Sechzehnersystem (hexadezimal) umfassen den Bereich von 0 bis F und diese werden im Zehnersystem (dem Dezimalsystem) direkt in die Zahlen 0 bis 15 übertragen. Informatiker sagen, dass bei der Bearbeitung von Integern kein *Genauigkeitsverlust* auftritt.

All das ändert sich allerdings, sobald man Dezimalkommata ins Spiel bringt. Historisch gesehen, waren die Arten der Gleitkommazahl-Darstellung in Computern verschiedenartig, komplex und verdreht. Erst in jüngster Vergangenheit entwickelte sich eine Art Standard zur Darstellung dieser Zahlen und viele sind der Meinung, dass auch dieser Standard, IEEE-64, voller Fallgruben und Unzulänglichkeiten steckt. Leider würde selbst der Platz des gesamten Buchs nicht ausreichen, um dieses faszinierende Thema zu behandeln. Und vermutlich würden sich auch nicht mehr als drei Leute dafür interessierten, wenn ich näher darauf einginge. Daher beschränke ich mich darauf zu sagen, dass die Probleme vielfältig und die Themenbereiche komplex sind.

An dieser Stelle müssen wir nur berücksichtigen, dass der Genauigkeitsverlust Probleme im Umgang mit Gleitkommazahlen mit sich bringt. Überraschenderweise ist es dabei so, dass Multiplikation und Division unter solchen Umständen zwar einem gewissen Fehlermaß unterliegen, sich der Fehler bei wiederholten Operationen jedoch nicht signifikant auswirkt, wogegen der Fehler sich bei wiederholter Addition und Subtraktion wesentlich vergrößert. Die Moral ist die, dass man Gleitkommaaddition und -subtraktion nur mit Vorsicht verwenden sollte und wo immer möglich versuchen sollte, entsprechende Operationen durch Multiplikation und Division auszugleichen.

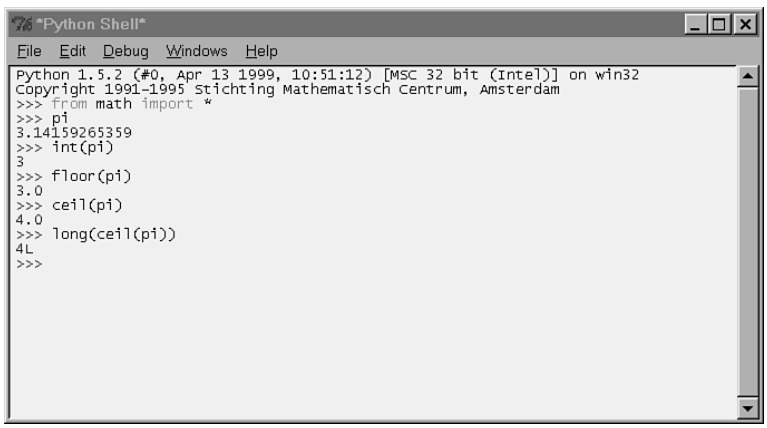
Wenn Sie Gleitkomma-Mathematik benötigen, werden Sie früher oder später einige Ergebnisse von Gleitkommaberechnungen in Integer, bisweilen auch in lange Integer, umwandeln wollen. In Python gibt es dabei einen einfachen Weg, die eingebauten Funktionen zur *Typkonvertierung* zu nutzen. Einige Beispiele sind in Abbildung 3.12 dargestellt.

Abb. 3.12: Einige Typkonvertierungen



Früher oder später werden Sie sich jedenfalls in Situationen befinden, in denen Sie diese Konvertierungen in die eine oder andere Richtung gehen lassen wollen; das heißt, man benötigt manchmal aufgerundete und manchmal abgerundete Ergebnisse. Der zweckmäßige Weg, sicherzustellen, dass Ergebnisse so gerundet werden, wie Sie sie benötigen, besteht darin, die mathematischen Funktionen `floor()` und `ceil()` zu verwenden. Erinnern Sie sich daran, wie wir zuvor das Modul `sys` importiert haben? Und so etwas muss auch immer dann geschehen, wenn Sie spezielle mathematische Funktionen benötigen: Sie importieren das Modul `math`. Wie Sie in Abbildung 3.13 sehen können, benutzen wir dieses Mal eine andere Form der Import-Anweisung.

Abb. 3.13: `floor()` und `ceil()`



Würden wir diese spezielle Form nicht benutzen, müssten wir `math.pi` eingeben, um den Zahlenwert von *Pi* zu erhalten; in ähnlicher Weise müssten

wir `math.floor()` und `math.ceil()` schreiben, um auf `floor()` und `ceil()` zugreifen zu können. Viele Leute vermeiden die Form `from x import *` und behaupten, sie sei nicht immer sicher. Es kann sein, dass Namen in einem importierten Modul mit den zukünftigen kollidieren können, die Programmierer in ihren Programmen verwenden möchten, aber die Wahrscheinlichkeit ist gering und für gewöhnlich merkt man, wenn ein Problem auftaucht. Wenn ich allerdings den Programmcode eines Moduls, das ich importieren möchte, nicht gelesen habe, werde ich die Form `import *` so gut wie nie verwenden. Gut dokumentierte (oder zumindest wohlbekannte) Module wie *tkinter* bilden da die Ausnahme.

Aus der obigen Abbildung können Sie schon fast alles ersehen, was Sie über diese mathematischen Funktionen wissen müssen – passen Sie vor allem auf, dass die Klammern immer paarweise auftreten. Was Sie noch sehen können, ist, dass eine Funktion, `long()`, als Eingabewert den Ausgabewert einer anderen Funktion, hier `ceil()`, verwenden kann. Es ist gut, das zu wissen; statt Zahlen als Eingabewerte in Funktionen zu verwenden, können Sie dafür Funktionen aufrufen, die Ihnen einen Eingabewert berechnen.

Ein weiterer Aspekt beim Runden von Gleitkommazahlen ist die Frage, »was bei negativen Werten geschieht.« Diese Frage werden wir etwas später in diesem Kapitel behandeln.

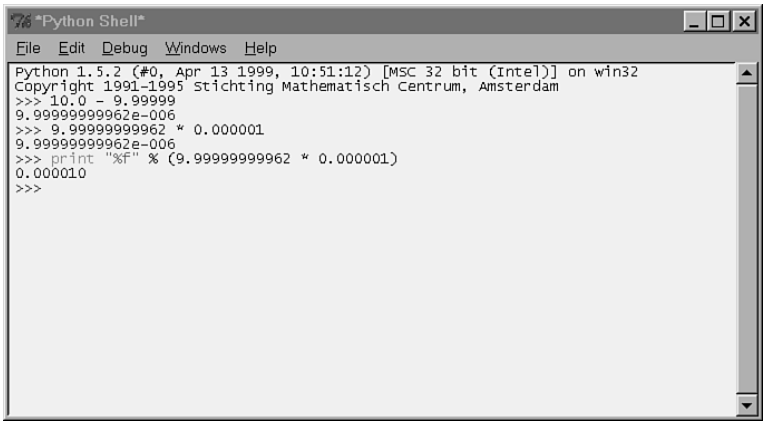
3.4 Exponentialrechnung

Exponentialrechnung oder Potenzierung ist etwas, woran Sie sich möglicherweise aus der Schulmathematik erinnern. Es bedeutet einfach, dass man eine Zahl nimmt und diese einige Male mit sich selbst multipliziert. 2^2 bedeutet zum Beispiel, 2 zweimal zu multiplizieren oder zu *quadrieren*, was zum Ergebnis 4 führt; entsprechend bedeutet 2^3, 2 mal 2 mal 2 zu rechnen (*kubieren* genannt), was zum Ergebnis 8 führt.

Wissenschaftliche Schreibweise oder genauer gesagt *Gleitkommaschreibweise* verwendet Exponenten, um sich das Schreiben sehr großer Zahlen mit vielen wiederkehrenden Nullen zu ersparen. Weiter oben (Abbildung 3.7) sahen Sie ein Beispiel, in dem wir 10,0 – 9,99999 subtrahierten und die überraschende Antwort 9.99999999962e-006 erhielten, eigentlich aber die Antwort 0,00001 erwartet hätten. Die wissenschaftliche Schreibweise umfasst zwei Teile: den *Exponentialteil* und den *Fraktionsteil* (der fälschlicherweise oft als *Mantisse* bezeichnet wird). Das *e* steht für den Exponenten und der Teil -006 liefert den tatsächlich verwendeten Exponenten. Zahlen wie diese werden für gewöhnlich als »9.99999999962 mal 10

hoch minus 6« bezeichnet – falls Sie sich die Mühe machen, alle Neunen hinter dem Dezimalkomma auszusprechen. Negative Exponenten sagen Ihnen, dass die Zahl so viele Male durch die Basis (d.h. 10) geteilt werden soll. 10^{-1} zum Beispiel bedeutet dasselbe wie 1/10 oder 0,1. Unser Exponent minus 6 bedeutet hier, dass 1 sechsmal durch 10 geteilt werden soll bzw. 1/1000000. Das ist eine kleine Zahl und ihre dezimale Entsprechung ist 0,000001. Abbildung 3.14 zeigt dies.

Abb. 3.14: Negative Exponenten



Beachten Sie, dass das ausgegebene Ergebnis 0,00001 ist und somit sehr nahe an dem liegt, was wir ursprünglich erwartet haben. Erinnern Sie sich noch, dass ich zuvor sagte, dass Rundungsfehler bei der Addition und Subtraktion dazu neigen, sich wesentlich zu vergrößern? Wenn Sie mit sehr kleinen Zahlen wie 10^{-6} operieren, summieren sich diese Fehler.

Die wissenschaftliche Schreibweise ist in der Programmierung und in wissenschaftlichen Kreisen weit verbreitet; es ist daher von Nutzen, sich zumindest ein wenig damit vertraut zu machen. Ein paar übliche Werte, denen Sie häufig begegnen können, sind 10^3, 1000; 10^6, eine Million; 10^9, eine Milliarde; 10^{12}, eine Billion. Und erinnern Sie sich an die von 100 Nullen gefolgte 1 – die Googol? Nun, es gibt in der wissenschaftlichen Schreibweise zwei Arten, Googol zu schreiben: 10^{100} ist die eine und $\#_{10}10^{10}$[#10 hoch 10 hoch 10] die andere; gleichwohl haben beide dieselbe Bedeutung. Nachdem Kasner die Googol erfunden hatte, erfand ein anderer Mathematiker die so genannte *Googolplex*, die als 10^{googol} definiert ist und die voll auszuschreiben ich nicht wagen würde. Für die Neugierigen gibt es auf der Internet-Seite zu diesem Buch ein Python-Programm zur Ausgabe von Googols und Googolplexen. Es wird laufen, bis Ihnen die Geduld oder Ihrem Computer der Speicher oder der Platz auf der Festplatte ausgeht, oder bis zum Hitzetod des

Universums – je nachdem, was zuerst eintritt. Abbildung 3.15 zeigt einen schnellen Weg, eine Googol in Python auszugeben.

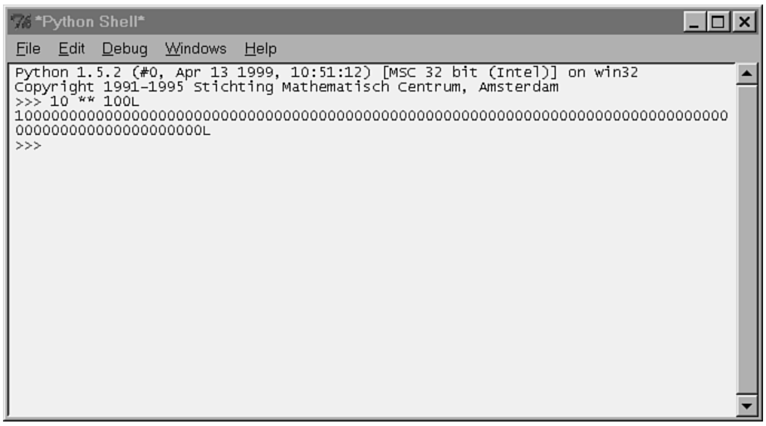


Abb. 3.15: Eine Googol

3.5 Klammerung

Nachdem Sie sich mit der wissenschaftlichen Schreibweise angefreundet haben, werden Sie vielleicht herausfinden, dass eine Menge wirklich nützlicher Informationen überall im Internet (und manchmal auch in richtigen Büchern) in mathematischen Formeln verschlossen ist. Solche Formeln, zumindest aber alle außer den allereinfachsten, haben zwei Zeichen gemeinsam: Klammern. Dafür gibt es einen Grund. Es gibt Klammern, um die *Reihenfolge von Operationen* anzuzeigen. So kann man eine Formel anschauen und sofort wissen, was als Erstes zu bearbeiten ist, selbst, wenn man nicht begreift, worum es geht oder was die Formel leisten soll.

Python hat mit vielen anderen Programmiersprachen ein ausgefeiltes System von *Vorrangregeln* gemein[2]; sie gestatten es dem Programmierer, zahlreiche Klammern auszulassen und somit die Lesbarkeit enorm zu verbessern. Sie müssen sich lediglich 20 oder 30 komplexe Regeln merken und Ihre Gleichungen entsprechend umschreiben und schon können auch Sie in einer der leserlichsten Programmiersprachen (natürlich Python) unlesbaren Programmcode schreiben.

Ich bin kein Freund von Operator-Vorrangregeln, wie Sie sich vielleicht vorstellen können. Ich mag die Dinge klar und deutlich ausgedrückt und bevor-

2 Anm. d. Übers.: Und auch hier gibt es wieder etwas, das Sie beim Erlernen von Python nicht vergessen dürfen ...

zuge es, Computern zu sagen, was sie zu tun haben, und ich lasse es mir nicht von ihnen vorschreiben. Daher habe ich meine eigenen Regeln, die um einiges einfacher sind:

Regel 1: Sofern es sich bei allen mathematischen Operationen um Additionen oder Subtraktionen handelt, vergessen Sie die Klammern.

Regel 2: Sobald Sie irgendeine andere Art von Operatoren einsetzen, verklammern Sie alles.

Am Anfang, als ich Programmieren lernte, wurden mir die Vorrangregeln mehrfach zum Verhängnis.

Sie unterscheiden sich von dem, was ich erwartet hatte. Wenn eine Formel »400 + 1 / 3« lautet, hätte ich erwartet, dass sie 401 durch 3 teilt. Wie Sie in Abbildung 3.16 sehen können, passiert das jedoch nicht.

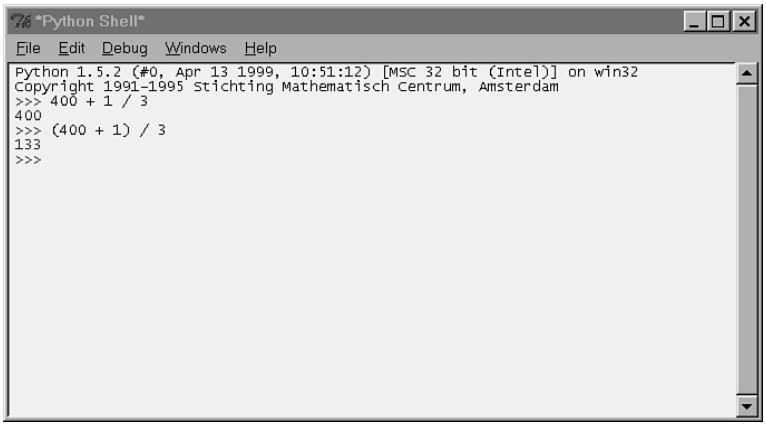


Abb. 3.16: Die Gefahren der Vorrangregeln

Falls Sie meine Regeln befolgen und alles verklammern und Ihr Programmcode anfängt, vor Klammern zu ersticken, sind Sie im Begriff, zuviel in eine Zeile hineinzuzwängen. Teilen Sie diese in mehrere Zeilen auf. Das verbessert die Lesbarkeit und ist daher eine sehr gute Sache. Bedenken Sie, dass jemand Ihren Programmcode warten muss; gelegentlich werden Sie dieser Jemand sein, manchmal aber auch nicht. Schauen Sie sich jede Zeile kritisch an und fragen Sie sich: »Werde ich mich in sechs Wochen noch daran erinnern, was diese Zeile tut?« Wenn die Antwort nein lautet, sollten Sie ernsthaft erwägen, die Zeile zu ändern.

3.6 »Gotchas« und Gemischtes

Das wichtigste »*Gotcha*« (*Gotchas* sind Fallstricke) bei Python im Umgang mit Zahlen ist die Art und Weise, wie Python `floor()` und `ceil()` auf negative Zahlen anwendet. Python unterscheidet sich darin von C und C++, sodass Sie, sofern Sie Erfahrungen mit einer dieser Sprachen gesammelt haben, auf mögliche Fehler gefasst sein müssen. Abbildung 3.17 verdeutlicht das Problem.

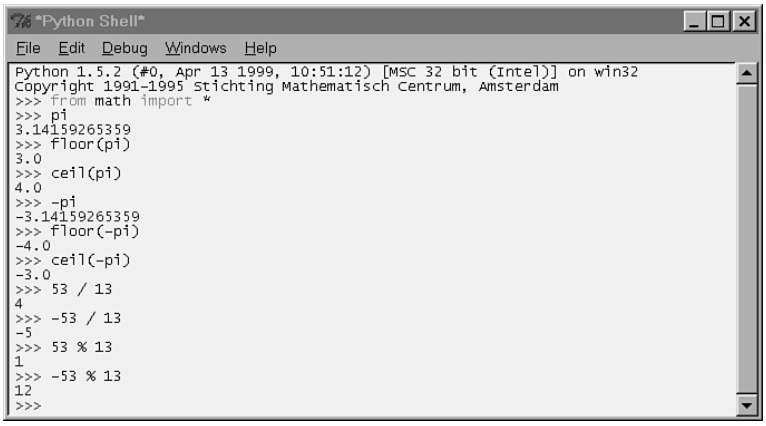


Abb. 3.17: Die Konvertierung negativer Zahlen

In C liefert Ihnen die Anwendung von `floor()` auf pi die Zahl 3, aber die Anwendung auf -pi ergibt -3 und nicht -4 wie in Python. Wenn Sie `ceil()` anwenden, liefert Python 4 bzw. -3. Warum das? Ferner ist `floor()` als »abwärts rundend« definiert; sollte das nicht bedeuten, dass sich beide Sprachen identisch verhalten? Nun, das ist nicht der Fall. Der Grund für den Unterschied liegt darin, dass im Fall von Python abwärts in der Bedeutung von kleiner genommen wird, und zwar in derselben Weise wie 0,0001 < 0 < 10000 und 0,0000000000001 kleiner ist als 0,0001. In C hingegen bedeutet kleiner näher an Null. Daher ist -4 in Python kleiner als -3,14159265359; die Integer-Division verhält sich in der gleichen Weise wie `floor()` und `ceil()`, indem sie zur kleineren Zahl rundet statt näher zur Null.

Dies kann Probleme verursachen, wenn Sie nicht damit rechnen. Ich hatte zum Beispiel eine Funktion in C, mit der ich die Differenz des Gregorianischen Kalenders zum Julianischen Kalender in Tagen berechnete; dieser Wert wird als *Abweichung* bezeichnet und ist über einen Zeitraum von 100 oder 200 Jahren konstant. Ich übertrug die Funktion in Python und probierte sie schnell aus; sie schien zu funktionieren, aber zwei oder drei Tage später hatte ich die Gelegenheit, sie mit negativen Daten zu testen. Sie be-

hauptete, dass es vor dem Jahr 0 (ich verwende hier die astronomische Schreibweise, in der es ein Jahr 0 gab und Jahre vor 0 mit einem Minuszeichen gekennzeichnet sind, statt vor Christi Geburt zu schreiben) keine Abweichung gab. Das stimmte einfach nicht und ich verfolgte es schließlich bis zu einer Zeile zurück, die `devn = 2 - leaper + (leaper / 4 )` lautete. Sowohl `devn` als auch `leaper` sind *Variablen*, was einfach »ein Platz ist, der einen Wert speichert«. (Sie werden Variablen im nächsten Kapitel kennen lernen). Weil `leaper` das Jahrhundert speichert, ergab die Division `leaper / 4` bei einer negativen Zahl einen kleineren Wert statt des näher an Null liegenden Wertes, den ich erwartet hatte.

Inzwischen stellt sich heraus, dass der Python-Weg für eine große Menge kalendarischer Berechnungen die bevorzugte Methode ist; diejenigen unter den Lesern, die keine anderen Programmiersprachen kennen, werden kein Problem damit haben, da sie die Funktionsweise noch nicht kennen. Ich »wusste«, wie es hätte funktionieren sollen, und ich war verärgert, als sich die Funktion nicht in der gewünschten Weise verhielt. Langer Rede kurzer Sinn: ich hatte Programmcode benutzt, der in C funktionierte, ihn auf Python übertragen und die Python-Funktionen nicht ausreichend getestet.

Die Moral ist hier ganz klar: Lernen Sie Ihre Werkzeuge kennen, testen Sie Ihren Programmcode gründlich und gehen Sie nicht immer davon aus, dass Sie bereits alles über Ihre Programmierwerkzeuge wissen. Sie könnten unangenehm überrascht werden.

Einer der wesentlichen Vorteile von Python ist die eingebaute Unterstützung langer Integer. In C und C++ mit dem Maya-Kalender zu arbeiten, kann äußerst frustrierend sein, denn wenn Sie Unterstützung für sehr große Zahlen benötigen, müssen Sie diese selbst programmieren. Python vereinfachte Berechnungen des Maya-Kalenders, aber einige Bereiche wären noch einfacher gewesen, wenn es keinen Unterschied zwischen gewöhnlichen und langen Integern gäbe oder wenn Python von sich aus lange Integer verwenden würde, wenn die Zahlen zu groß werden.

Die Unterscheidung zwischen gewöhnlichen Integern und langen Integern scheint weitgehend willkürlich zu sein. Guido überlegt, sie in einer zukünftigen Python-Version vielleicht aufzuheben, aber bis dahin wird es noch eine ganze Weile dauern. Seien Sie in der Zwischenzeit einfach vorsichtig, wenn Sie Arithmetik verwenden, die größere als 32-Bit-Integer erfordert; falls Sie zufällig mit dem Maya-Kalender zu tun haben, ist es zum Beispiel überhaupt nicht schwer, die Grenze von zwei Milliarden (plus Kleingeld) zu überschreiten.

Es gibt Mittel und Wege, Zahlen bei Bedarf automatisch von 32-Bit in lange Zahlen umzuwandeln, aber Sie werden diese Methoden vorerst nicht ken-

nen lernen. Selbst dadurch kann die Grenze zwischen diesen beiden Typen von ganzen Zahlen nicht ganz transparent gemacht werden und es ist oftmals leichter, anfangs alle mathematischen Berechnungen mit langen Zahlen zu beginnen, obwohl Sie dafür eine Geschwindigkeitseinbuße in Kauf nehmen müssen. Natürlich werden Sie nicht ausschließlich lange Zahlen benutzen wollen, wenn es für Ihr Problem nicht erforderlich ist. Für die meisten gewöhnlichen Programme reichen normale 32-Bit-Integer vollkommen aus.

3.7 Zusammenfassung

In diesem Kapitel haben wir alle einfachen arithmetischen Operatoren in Python behandelt und Sie haben den Interpreter eingesetzt, um das Verhalten des Interpreters kennen zu lernen. Sie sollten allmählich ein Gefühl für Python entwickeln, sodass Sie sich recht wohl fühlen sollten, wenn wir uns im nächsten Kapitel mit Variablen und Kontrollfluss beschäftigen. Wir haben ebenfalls Einiges zum Runden, zu Gleitkommafunktionen, der wissenschaftlichen Schreibweise, der Klammerung und zur unterschiedlichen Philosophie von Python und einigen weiteren Sprachen besprochen.

Zu Ihrer Annehmlichkeit und zur Wiederholung habe ich eine Tabelle der mathematischen Operatoren, die in Python Verwendung finden, beigefügt. Einige dieser Operatoren bedeuten etwas anderes, wenn sie auf andere Objekte als Zahlen angewendet werden, aber wir werden das im entsprechenden Zusammenhang besprechen. Ähnliche Tabellen finden Sie jeweils in der Zusammenfassung der Kapitel, in denen spezielle Symbole vorkommen. Später in diesem Buch werden Sie mehr darüber erfahren, warum sich Operatoren unterschiedlich verhalten, wenn sie auf unterschiedliche Objekte angewendet werden.

Tabelle 3.1: Mathematische Operatoren in Python

Symbol	Bedeutung
`+`	Addition/Identität
`-`	Subtraktion/Negation
`*`	Multiplikation
`/`	Division
`%`	Modulo-Operator (Restoperator)
`**`	Potenzierung
`divmod(x,y)`	Funktion, die sowohl x/y als auch x % y zurückgibt

3.8 Workshop

3.8.1 F&A

F: Was ist besser? Ein Computer oder ein Abakus?

A: Das kommt ganz darauf an. Wenn Sie Kunden in einem Geschäft bedienen und Einkäufe zusammenrechnen ist ein Abakus eindeutig das richtige Werkzeug – es sei denn, Sie wünschen automatisch ausgedruckte Quittungen. Wenn Sie dagegen Programme schreiben wollen, hat so ein Abakus entsetzlich lange Download-Zeiten.

F: Wann wurde die wissenschaftliche Schreibweise erfunden?

A: Sie wurde vor 1890 mindestens zweimal verwendet, aber das waren möglicherweise künstliche, bei der Übersetzung entstandene Begriffe. Eine frühe Originalquelle stellt Jeans´ *Theoretical Mechanics* dar, ein Text von 1907, in dem die Masse der Erde mit $6x10^{27}$ Gramm angegeben wird. (Dank an John Harper von der Victoria University, Wellington, Neuseeland, für dieses Zitat.)

3.8.2 Quiz

1. Wie teilen Sie dem Python-Interpreter mit, dass es sich bei einer Zahl um einen langen Integer handelt?

   a) long("10000000000000000000000000000000000000")

   b) 10000000000000000000000000000000L

   c) string.atol("10000000000000000000000000000000")

   d) *irgendeine Integer* / 1L

2. Was geschieht, wenn Sie eine Gleitkommazahl mit einem wirklich langen Integer multiplizieren?

   a) Sie erhalten einen OverflowError.

   b) Python verwandelt das Ergebnis in eine komplexe Zahl.

   c) Python sagt Ihnen, dass Sie das Ergebnis ohnehin nicht verstünden.

   d) Wenn die wirklich große Zahl um die 300 Ziffern hat, antwortet Python mit `1.#inf` (oder `inf` unter Linux) anstelle eines nummerischen Ergebnisses; andernfalls wird ein Ergebnis in wissenschaftlicher Schreibweise ausgegeben.

3. Wenn *binär* das 2er-System und *hexadezimal* das 16er-System bezeichnet, was ist dann *oktal* und wer verwendet es?

   a) 32er-System; von Macintosh-Programmierern verwendet.

   b) Es hat nichts mit Zahlen zu tun; *oktal* ist der Name des Zeichens #.

   c) 8er-System; von Unix-Programmierern benutzt, die Hosenträger tragen und Pfeife rauchen.

   d) 8er-System; aber niemand verwendet es.

3.8.3 Antworten

1. a, b, c, d. Alle angegebenen Methoden funktionieren.

2. d. Die Gleitkommaarithmetik bei Windows und Linux stößt bei einem Exponenten von plus oder minus 308 an ihre Grenze, was eine ziemlich große (oder sehr kleine) Zahl ist. Wenn diese Grenzen überschritten werden, geben die Gleitkommabibliotheken ein spezielles Symbol für Unendlichkeit aus.

3. c. Oktal ist in der Tat das 8er-System; und es wird hauptsächlich von Unix-Programmierern und -Programmen verwendet (nicht alle von uns tragen Hosenträger). Der bekannteste Einsatz ist das *chmod*-Programm in Unix, welches es Anwendern gestattet, die Zugriffsrechte ihrer Dateien zu ändern. Ein Spitzname für das Zeichen »#« lautet im Englischen übrigens *Octothorpe* (wegen der acht Spitzen).

3.9 Übungen

Sollten Sie zufällig herausfinden, dass Sie sich für Dinge wie die Genauigkeit von Gleitkommazahlen und den durch die Darstellung von Gleitkommazahlen in Computern verursachten Genauigkeitsverlust interessieren, konsultieren Sie auf alle Fälle Donald Knuths *The Art of Computer Programming*, besonders Band 2, *Seminumerical Algorithms*, Kapitel 4, in dem viele historische Aspekte behandelt werden.

Falls Sie an große Zahlen wie der Googol interessiert sind, können Sie mit den folgenden Websites Ihrem Forscherdrang nachgehen. Einige der dort vorgestellten Algorithmen eignen sich zur Implementierung in Python und auf der Internet-Seite dieses Buchs können Sie einige dieser Implementierungen finden.

1. Große endliche und unendliche Zahlen:
   `http://www.sci.wsu.edu/math/faculty/hudelson/moser.html`

2. Wie viel ist eine Gazillion?
   http://www.straightdope.com/mailbag/mgazilli.html
3. Wie man eine Googolplex erhält: http://www.informatik.uni-frankfurt.de/~fp/Tools/GetAGoogol.html
4. Namen kleiner und großer Zahlen:
   http://studwww.rug.ac.be/~hvernaev/FAQ/node26.html
5. Googolplex: http://www.informatik.uni-frankfurt.de/~fp/Tools/Googool.html (beste Wahl)
6. Pi und seine Freunde: http://www.go2net.com/internet/useless/useless/pi.html
7. Maya-Zeitrechnung und Zeitsymbole: http://www.pauahtun.org/calglyph.html

Finden Sie heraus, wie groß eine Zahl sein muss, um Python zum Absturz zu bringen. Wie groß muss eine Zahl sein, um Ihren Computer abstürzen zu lassen? (Sichern Sie vorher alle Ihre Dateien.)

Kapitel 4

Variablen und Kontrollfluss

Egal, was man ihnen vorsetzt, nichts hindert sie daran weiterzurennen.

Lynda Plettner, Hundeschlittenführerin,
im Gespräch über ihre Hunde während des Iditarot 1999

In diesem Kapitel werden Sie erfahren, was Variablen sind und wie sie eingesetzt werden. Wir werden uns damit beschäftigen, wie man Python veranlasst, Entscheidungen zu treffen und sich danach entsprechend zu verhalten. Da es eine der Stärken von Computern ist, dasselbe immer und immer wieder zu tun, bis eine Bedingung erfüllt bzw. nicht erfüllt ist, werden wir erfahren, wie man dies in Python macht. Wenn Sie mit diesen Konzepten vertraut sind, werden wir einige davon verwenden, um ein kleines, aber nützliches Programm zu schreiben.

4.1 Variablen

Sie werden sich vielleicht daran erinnern, was Variablen sind: all diese *x* und *y* aus der Schulmathematik, die Sie plagten als Sie viel lieber während des Unterrichts schlafen wollten. Jede Formel hat Variablen. Eine Formel bildet den Rahmen, um ein Problem zu formulieren, aber für das eigentliche Ergebnis muss man konkrete Werte in die Variablen einsetzen und die Formel ausrechnen. Diejenigen unter uns, die *tatsächlich* wärend des Unterrichts schliefen, erschauderten selbstverständlich beim Gedanken an das »Ausrechnen der Formel«.

Variablen funktionieren in Formeln und der Computerprogrammierung genau gleich; sie sind nur Platzhalter für die tatsächlichen Werte. In der

Algebra sind Variablen allerdings ausschließlich Platzhalter für nummerische Werte; dagegen können Variablen in Computerprogrammen *jegliche* Art von Information enthalten. Sie können sie so lange speichern, wie Ihr Programm läuft, oder auch nur für Millisekunden, wenn Ihnen das genügt. Sie können Variablen auch als »Schmierzettel« verwenden – einen Ort, an dem ein Zwischenwert notiert werden kann, sodass er nicht vergessen wird, während Sie an einer anderen Stelle der Formel oder des Problems arbeiten. Es wird Ihnen helfen, wenn Sie sich Variablen als »Haftnotizen« vorstellen; tun Sie dies unbedingt.

Variablen können in Python beliebige Namen haben – solange er nicht mit dem Namen eines von Python reservierten Schlüsselwortes übereinstimmt und solange er mit einem Buchstaben oder einem Unterstrich (_) beginnt. Erlaubte Namen sind `i`, `z1` und `old`. Namen, die nicht funktionieren, sind `123abc` (beginnt mit einer Zahl) und `if` (ist ein reserviertes Schlüsselwort). Wie immer ist das aber noch nicht alles. Einige Variablennamen sind besser als andere und manche Namen sind in gewissen Zusammenhängen besser als andere. Ich werde später in diesem Kapitel ein paar davon besprechen, aber eine gute allgemeine Richtschnur ist, Ihren Variablen »sprechende Namen« zu geben. Manchmal bedeutet dies, dass Sie lange Namen wählen sollten, manchmal auch kurze.

Reservierte Wörter werden auch als *Schlüsselwörter* bezeichnet, für die sich Python das Recht vorbehält, sie zu benutzen; in Python ist es nicht erlaubt, sie auf eine falsche Weise zu verwenden. Zu diese Wörter gehören `if`, `for` usw. Eine vollständige Tabelle mit all diesen Wörtern finden Sie am Ende dieses Kapitels.

Starten Sie den Interpreter, damit wir die Benutzung von Variablen ausprobieren können (siehe Abbildung 4.1).

Wenn Sie IDLE oder Python in einem Befehlszeilen-Fenster ausführen, können Sie sich den Wert einer Variablen anzeigen lassen, indem Sie einfach den Namen der Variablen eingeben und ↵ drücken. Wenn Sie Python allerdings benutzen, um damit Programme auszuführen, ist das nicht möglich; dann müssen Sie auf die `print`-Anweisung zurückgreifen. Die `print`-Anweisung kann auf verschiedene Arten benutzt werden und ich werde später näher darauf eingehen; merken Sie sich jetzt nur, dass Sie sowohl mit `print` *Variablenname* als auch mit *Variablenname* im interaktiven Modus (IDLE) den Wert einer Variable anzeigen können.

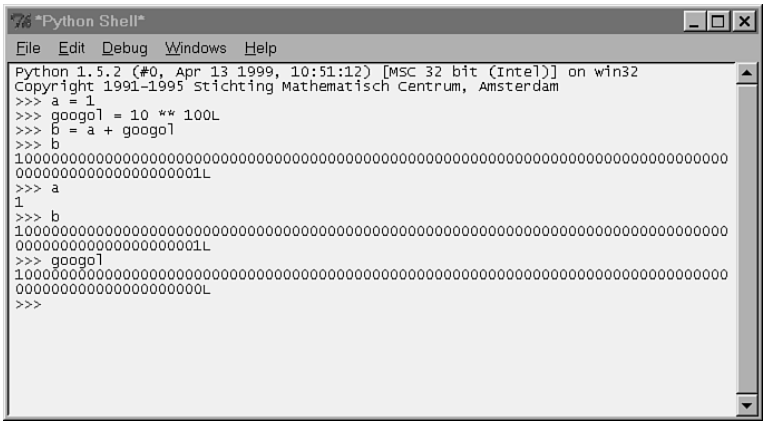


Abb. 4.1: Einige Variablen

Abbildung 4.1 zeigt ein anderes wichtiges Konzept – das der *Zuweisung*. Das Zeichen »=« bedeutet, dass der Wert auf der rechten Seite genommen und in der Variablen links davon gespeichert wird. Dieses Verfahren unterscheidet sich vom doppelten Gleichheitszeichen (»==«), das Sie später sehen, wenn wir die `if`-Anweisung besprechen werden; das einfache Gleichheitszeichen ist eine Zuweisungsanweisung und kein algebraischer Operator. Das doppelte Gleichheitszeichen ist ein Vergleichsoperator. Die beiden Zeichen sind in ihrer Bedeutung sehr unterschiedlich. In der Zeile, in der `googol = 10 ** 100L` steht, ist zu sehen, dass auf der rechten Seite der Zuweisung auch ein Ausdruck stehen kann; ein *Ausdruck* ist einfach ein Stück Programmcode, der zu einem Ergebnis führt. In diesem Fall berechnet `10 ** 100L` den Wert 10^{100} und das Zeichen »=« speichert den berechneten Wert in der Variable mit Namen `googol`. Solange bis Sie Python beenden, enthält `googol` denselben Wert (sofern Sie ihn nicht verändern).

Sie können das selbstverständlich tun. Variablen verhalten sich in Gleichungen oder Berechnungen genauso wie richtige Zahlen. Und das bedeutet, wie Abbildung 4.2 zeigt, dass Sie eine Variable wie eine Zahl benutzen können, nachdem Sie sie erstellt haben (indem Sie ihr einen Wert zuweisen).

Dies ist auch eine weitere Methode zum Gebrauch von `print`: Es lassen sich nicht nur gleichzeitig verschiedene Variablen ausgeben, indem man sie durch Kommata trennt, sondern auch konkrete Zahlen (die so genannten *Konstanten*) und sogar Strings einsetzen (siehe Abbildung 4.3).

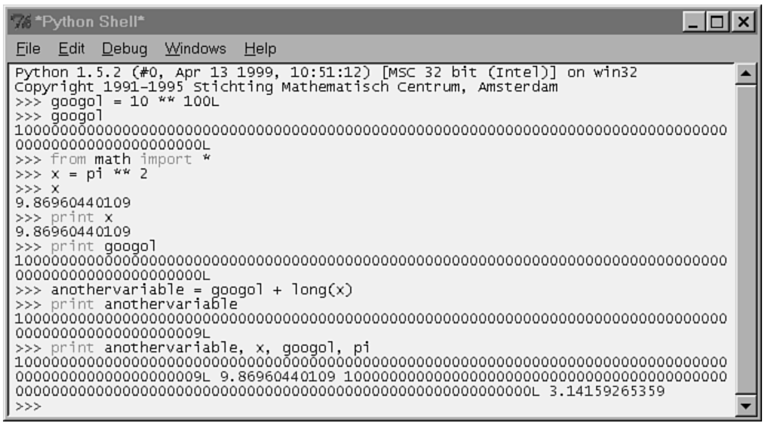


Abb. 4.2: Rechnen mit Variablen und Zuweisungen an Variablen

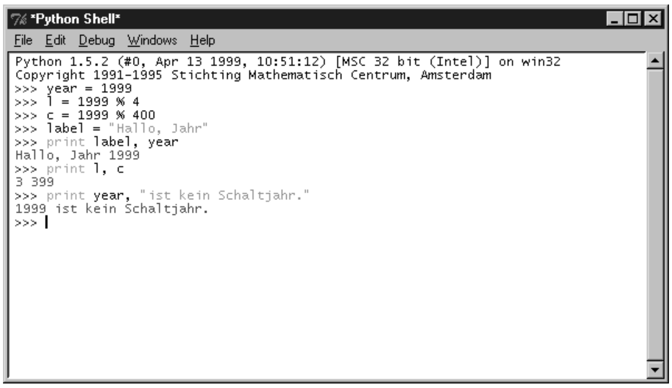


Abb. 4.3: Mehr Spaß mit `print`

Variablen sind bei der Programmierung sehr wichtig, aber um wirklich nützliche Dinge zu tun, reichen sie nicht aus. Es ist wichtig zu wissen, wie Sie den Computer anweisen, eine Entscheidung zu treffen und was er, abhängig vom Ergebnis dieser Entscheidung, als Nächstes tun soll. Entscheidungen sind das Thema des restlichen Kapitels. Wenn Sie Python mitteilen, welche Entscheidungen getroffen werden sollen, steuern Sie die Logik des Programms. Dies wird normalerweise als Kontrollfluss bezeichnet und eine Reihe grundlegender Spezialbegriffe zum Treffen von Entscheidungen in Python helfen Ihnen, den Fluss der Logik Ihres Programms zu steuern. Wir werden mit der wichtigsten (und sicherlich ältesten) Anweisung beginnen: `if`.

4.2 if, elif und else

Die `if`-Anweisung ist die grundlegendste Kontrollfluss-Anweisung beim Programmieren. Ihre Syntax in Python ist sehr einfach:

```
if test:
    tu etwas
```

Der Ausdruck *test* enthält fast immer einen *Vergleichsoperator*. Eine Tabelle aller Vergleichsoperatoren in Python finden Sie in der Zusammenfassung am Ende dieses Kapitels. Der erste Vergleichsoperator, den Sie kennen lernen sollten, ist der Test auf Gleichheit: »==«. Üblicherweise überprüft man damit, ob eine Variable einen bestimmten Wert hat, wie das folgende Beispiel zeigt:

```
if i == 1:
    tu etwas
```

Sie können den obigen Test ungefähr so ins Deutsche übersetzen: »Falls die Variable `i` *gleich* dem Wert 1 ist, dann *tu etwas*.« Der *tu-etwas*-Teil kann beliebig viele Programmanweisungen einschließlich anderer `if`-Anweisungen umfassen. In anderen Programmiersprachen kann *test* eine *Zuweisung* sein, sodass die `if`-Anweisung zweierlei oder dreierlei auf einmal erledigen kann, aber Guido hat beschlossen, das in Python nicht zuzulassen. Wenn man Programmierern gestattet, in `if`-Anweisungen Zuweisungen zu benutzen, sind Techniken wie zum Beispiel die folgende zulässig:

```
if i = 1
    tu etwas
```

Das bedeutet, dass Sie der Variablen `i` den Wert `1` zuweisen und anschließend überprüfen könnten, ob das Ergebnis zutrifft. Das kann jedoch leicht zu einem Fehler führen. Vielleicht wollten Sie auf Gleichheit prüfen und haben dabei das zweite Zeichen »=« vergessen; in diesem Beispiel wird das Ergebnis stets zutreffen, da `i` *immer* den Wert `1` haben wird. C-Programmierer verwenden maßlos viel Zeit darauf, Fehler wie diesen zu suchen, und Guido war der Ansicht, dass man die Zeit zur Fehlersuche besser mit anderen Dingen verbringen sollte.

Der *tu-etwas*-Teil der `if`-Anweisung muss eingerückt werden; in den meisten anderen Programmiersprachen ist die Einrückung freigestellt, in Python dagegen nicht. Andere Sprachen verwenden hauptsächlich spezielle Markierungen, um den Beginn und das Ende des *tu-etwas*-Teils, der als Block bezeichnet wird, anzuzeigen. Pascal zum Beispiel benutzt `BEGIN` und `END`, sodass eine `if`-Konstruktion folgendermaßen aussähe:

```
if b < 0 then
begin
    result := result + pi ;
end ;
```

Das Zeichen < bedeutet *kleiner als* und das Zeichen > bedeutet *größer als*. Derselbe Programmcode sähe in C so aus:

```
if ( b < 0 )
{
    result = result + pi ;
}
```

oder optional wie folgt:

```
if ( b < 0 ) result = result + pi ;
```

Die erforderliche Einrückung in Python zählt zu den am heftigsten diskutierten Eigenschaften dieser Sprache; Programmierer scheinen sie entweder zu lieben oder zu hassen. Einige Leute mögen die Sprache, können sich für diese Einrückung jedoch nicht recht begeistern (der technische Redakteur dieses Buchs ist so jemand); und insgesamt gesehen führt diese erforderliche Einrückung am ehesten zu einem Sprachenstreit. Ich selbst neige dazu zu denken, dass diese Einrückung zur Markierung zusammengehöriger Abschnitte des Programmcodes sehr attraktiv ist und Python zu einer der am besten lesbaren Sprachen weit und breit macht. Dies ist auch die Eigenschaft, bei der es am wenigsten wahrscheinlich ist, dass Guido sie ändern wird, sodass ich Ihnen rate, sie lieben zu lernen, falls Sie das nicht schon längst tun.

Als ich Python programmieren lernte, tauchte in Kommentaren zu Programmen oder Newsgroup-Nachrichten häufig der Begriff *iff* auf. Er verwirrte mich lange Zeit, denn er war offensichtlich nicht Teil irgendeiner Programmiersprache; schließlich stieß ich in einem obskuren Programm auf einen obskuren Programmkommentar, der lautete »*iff*: genau dann, wenn« (»*if, and only if*«). Ich hoffe, dass ich Ihnen hiermit jahrelange Verwirrung erspart habe ...

Die vollständige Syntax für die `if`-Anweisung erlaubt eine erweiterte Steuerung des logischen Flusses Ihrer Programme:

```
if test :
    tu etwas
elif test :
    tu etwas
elif test :
    tu etwas
else :
    tu etwas
```

Die *tu-etwas*-Blöcke können jede beliebige Länge annehmen und alle zulässigen Python-Programm-Anweisungen enthalten. Wir werden diese Form in IDLE testen und daran sehen, wie sie funktioniert. Erinnern Sie sich noch an die Gregorianischen und Julianischen Schaltjahrregeln, über die wir im letzten Kapitel sprachen? Starten Sie IDLE; Abbildung 4.4 zeigt eine komplette `if`-Konstruktion, die ermittelt, ob es sich bei einer Jahreszahl um ein Schaltjahr des Gregorianischen Kalenders handelt.

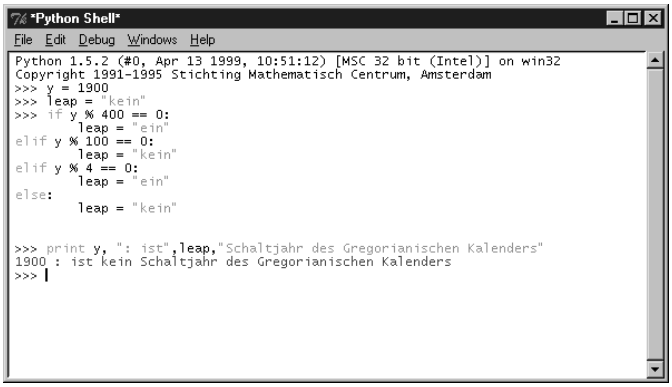


Abb. 4.4: `if` für den Gregorianischen Kalender

Beachten Sie in Abbildung 4.4, dass man IDLE das Ende eines eingefügten Programmcode-Blocks mitteilt, indem man am Ende des Blocks eine Leerzeile einfügt. Drücken Sie dazu einfach ein weiteres Mal auf die Taste ↵. Das funktioniert auch im Befehlszeilen-Interpreter genauso, aber dieser rückt im Gegensatz zu IDLE die Zeilen nicht automatisch für Sie ein. Wenn Sie im Befehlszeilen-Interpreter (oder in Dateien, die Sie mit einem Editor erstellen) die Zeilen einrücken wollen, dann können Sie dies sowohl mit Leerzeichen als auch mit Tabulatorzeichen tun. Dabei müssen Sie jedoch darauf achten, dass Sie alle zusammengehörigen Zeilen eines Blocks auf die gleiche Art und Weise (mit der gleichen Kombination von Leerzeichen und Tabulatorzeichen) einrücken. Ich empfehle Ihnen, sich entweder für Tabulatorzeichen oder Leerzeichen zu entscheiden, aber oft wird Ihnen die Entscheidung auch von Ihrem Lieblingseditor abgenommen. Sollten Sie einmal in die Verlegenheit geraten, dass Ihre Zeilen zu lang werden (mein Drucker mag das gar nicht), so können Sie am Ende der Zeile einen Backslash »\« setzen und die Anweisung in der folgenden Zeile fortsetzen. Falls Sie in die Versuchung kommen, den Backslash »\« benutzen zu wollen, sollten Sie jedoch zuerst in sich gehen und sich fragen, ob Sie diese lange Zeile in sechs Wochen wohl noch verstehen werden ... (Sie erinnern sich doch noch an meine Klammerregeln ...?)

Für den Julianischen Kalender brauchen Sie natürlich nur die Teilbarkeit durch vier zu überprüfen; das bedeutet, dass 1900 im Julianischen Kalender ein Schaltjahr war.

Wenn Sie möchten, können Sie mehrere Überprüfungen in einer Zeile kombinieren, indem Sie die speziellen logischen Operatoren `and`, `or` und `not` einsetzen. Wenn Sie in einer `if`-Anweisung zwei Tests mit `and` verknüpfen, müssen beide Bedingungen zutreffen; falls der erste Test das Ergebnis »falsch« liefert, wird dabei der zweite gar nicht erst ausgewertet. Bei `or` kann jeder der beiden Tests das Ergebnis »wahr« liefern. Falls der erste Test das Ergebnis »wahr« liefert, wird der zweite nicht ausgewertet, da das nicht nötig ist. Abbildung 4.5 zeigt, wie die Operatoren funktionieren.

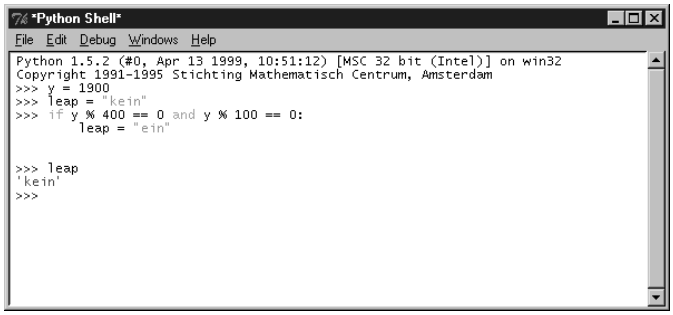


Abb. 4.5: Die Anweisungen `if` und `and`

Der erste Test – für die Teilbarkeit durch 400 – führte zum Ergebnis »falsch«; daher wurde der Test auf Teilbarkeit durch 100 gar nicht erst ausgeführt.

4.3 for und range()

Obwohl die Anweisung `if` sehr nützlich ist, kann man sie nicht in allen Fällen gebrauchen, besonders, wenn man etwas wieder und wieder tun will und dazu jedes Mal lediglich den Wert einer Variablen ändert. Für solche Fälle ist Pythons `for`-Anweisung ideal. In anderen Sprachen müssen Sie bereits vorher berechnen, wie oft Sie etwas ausführen möchten, Variablen mit Startwerten belegen und einige andere, schwer zu merkende Details berücksichtigen. Pythons `for`-Anweisung ist bestechend einfach: Sie bedeutet einfach, »hier ist eine Liste von Dingen; `for` (für) jeden Eintrag in der Liste *tu etwas*«:

```
for target in list :
    tu etwas
else:
    tu etwas, sofern keine break-Anweisung ausgeführt wurde.
```

Target steht hier einfach für einen Variablennamen; der *tu etwas*-Teil wird formal als *body* (Rumpf) der `for`-Anweisung bezeichnet. Während der Ausführung des Rumpfes kann optional eine `break`-Anweisung auftreten, die zum Abbruch der Ausführung des Rumpfes führt. Falls keine `break`-Anweisung ausgeführt wurde, dann (und nur dann[1]) wird der Rumpf des `else`-Abschnitts ausgeführt. Sie werden im Zusammenhang mit `for` tatsächlich nur selten `else`-Abschnitte sehen, weil sie nicht häufig gebraucht werden; benötigt man sie aber, sind sie Geschenke des Himmels. Wir werden mehr über die `break`-Anweisung erfahren, wenn wir uns mit `while` beschäftigen. In Abbildung 4.6 können Sie sich jedoch zuerst einmal ein Beispiel von `for` anschauen.

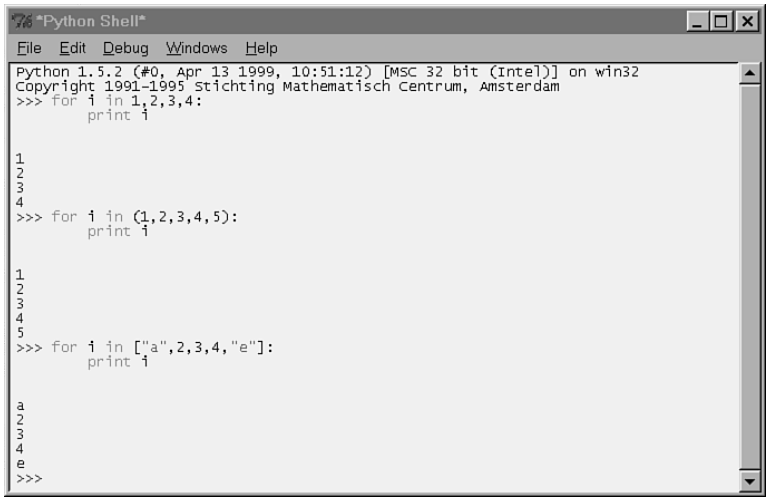


Abb. 4.6: Die Anweisung `for`

In der obigen Abbildung wird *list* durch `1,2,3,4`, durch `(1,2,3,4,5)` und durch `["a",2,3,4,"e"]` ersetzt; `for` findet vier, fünf und nochmals fünf Einträge in den Listen und weist der Variable `i` die Werte in der Reihenfolge zu, in der sie in *list* stehen. Sie können dies verfolgen, da der aktuelle Wert von `i` bei jeder Iteration ausgegeben wird. Beachten Sie auch, dass der Inhalt von *list* nicht auf Zahlen beschränkt ist; `Strings` funktionieren ebenso gut.

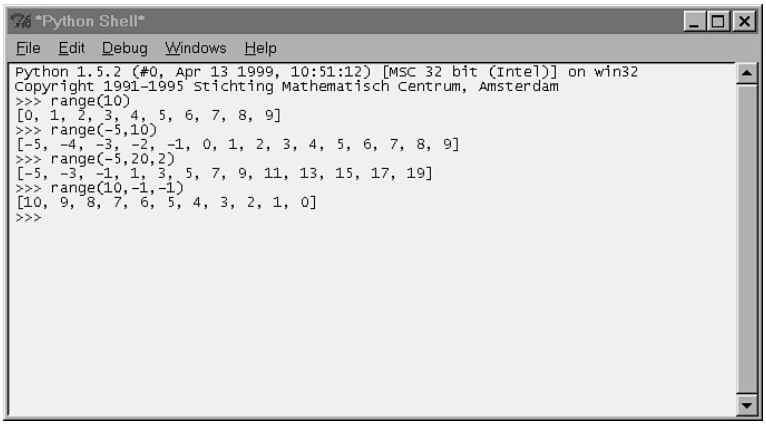
Iterieren bedeutet, etwas wieder und wieder zu tun; jeder einzelne Durchlauf ist eine Iteration. Das Schmieden eines Nagels kann als iterativer Prozess bezeichnet werden, in welchem jeder einzelne Hammerschlag eine

1 Anm. d. Übers.: Sie erinnern sich doch noch an iff …???

Iteration bildet. Wenn die Schleife komplett ausgeführt ist, haben Sie einen vollständigen Nagel. Und schmerzende Finger.

Die `for`-Anweisung wäre offensichtlich sinnvoller, falls wir uns die Eingabe aller benötigten Werte für *list* sparen könnten, und genau dabei hilft uns `range()`. Die Funktion `range()` ist eine raffinierte, eingebaute Funktion (*eingebaut*, engl. *built-in*, bedeutet, dass Python sie immer kennt; man muss nichts Besonderes tun, um sie verwenden zu können); sie liefert bei Bedarf sortierte Listen von Zahlen. Sie können sie auf vier verschiedene Arten verwenden. Die erste und einfachste ist in der ersten Eingabezeile in Abbildung 4.7 zu sehen.

Abb. 4.7: Die Funktion `range()`



Diese erste Syntax, `range(10)`, liefert eine Liste, in der Zahlen von 0 bis 9 enthalten sind; das Argument gibt dabei nicht die Größe, sondern den Grenzwert für die Zahlen in der Liste an, und die voreingestellte Startzahl ist immer 0. Wie in C beginnen nummerische Zahlenfolgen bei 0 und enden (bei positiver Richtung) eine Zahl unterhalb der Grenze; bei der Grenze 10 liegt das Ende also bei 9. Was geschieht, wenn man eine negative Zahl wählt? Was passiert bei Eingabe von `range(0)`?

Die zweite, in der Abbildung gezeigte Form, `range(-5,10)`, befiehlt `range()`, eine Liste zu erstellen, die wie oben eine Zahl vor der 10 endet, aber bei -5 beginnt. So erhält man eine Liste, die mit einer negativen Zahl beginnt.

Die dritte Form, `range(-5,20,2)`, zeigt, dass es für die Funktion drei mögliche Argumente gibt: eine optionale *Startangabe*, ein benötigtes *Ende* und eine optionale *Schrittweite*. Sie können erkennen, dass diese Form bei -5 beginnt, jeweils in Zweierschritten fortschreitet und bei 19 endet.

Abschließend demonstriert die vierte Form, dass `range()` nicht immer in positiver Richtung fortschreiten muss. Falls die *Startangabe* größer ist als das *Ende* und die *Schrittweite* negativ, zählt `range()` rückwärts.

Das Kombinieren von `for`, `range()` und `if` ist sehr gebräuchlich; in Abbildung 4.8 finden Sie ein Beispiel.

Abb. 4.8: Die Anweisungen `for`, `range()` und `if`

4.4 while

Die `while`-Anweisung befiehlt Python ebenfalls, etwas immer wieder zu tun, aber sie hat eine andere Syntax als die `for`-Anweisung. Diese Syntax ist in Abbildung 4.9 dargestellt.

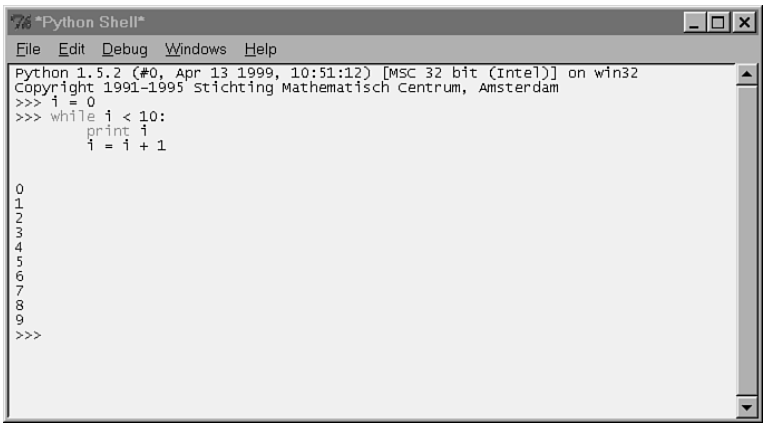
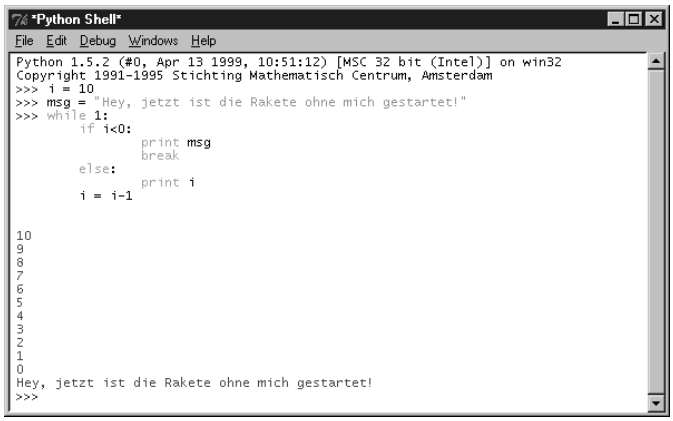


Abb. 4.9: Die Anweisung `while` mit einem Test

Natürlich ist es nicht sehr interessant, von 0 bis 9 zu zählen. Aber `while` dient nicht nur zum Zählen. Man kann damit auch einen bestimmten Ein-

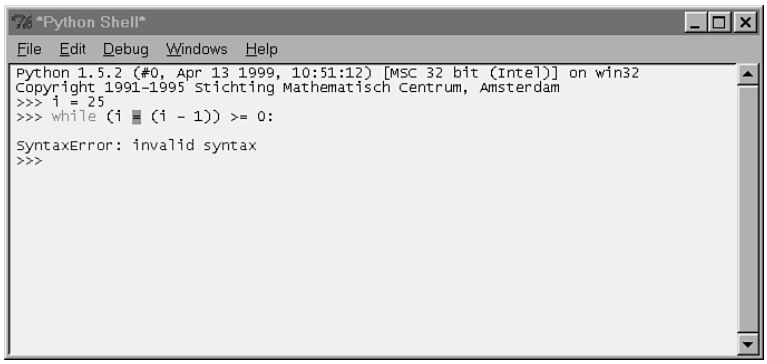
trag in einer Sequenz suchen oder überprüfen, ob eine bestimmte Bedingung eingetreten ist. Abbildung 4.10 zeigt den ersten dieser Fälle.

Abb. 4.10: Die Anweisung `while 1`



Beachten Sie, dass `while 1` stets wahr ist; das bedeutet, dass `while` unendlich oft ausgeführt wird, solange Sie, der Programmierer, nicht eingreifen. Dies wird als *Endlosschleife* bezeichnet; Endlosschleifen, die sich unendlich fortsetzen, sind zu vermeiden, aber Endlosschleifen sind sehr nützlich, sofern Sie eine *Abbruchbedingung* festlegen. Hier durchkämmen wir einige Zahlen; weil wir wie bei der `if`-Anweisung keine Werte im *test*-Teil der `while`-Anweisung zuweisen können (vergleiche Abbildung 4.11, um zu sehen, was geschieht, wenn wir es versuchen), müssen wir den Wert von `i` selbst ändern. Deswegen brauchen wir die Anweisung `i = i + 1`. Beachten Sie zum Schluss, was geschieht, wenn wir den gewünschten Wert sehen: Wir verwenden die `break`-Anweisung, um die unendliche `while`-Schleife (sinnvollerweise) zu verlassen.

Abb. 4.11: Eine ungültige `while`-Anweisung



Ebenso wie bei `for`, enthält die komplette Syntax für `while` einen `else`-Abschnitt:

```
while test:
    tu etwas
else:
    tu etwas, sofern keine break-Anweisung ausgeführt wurde.
```

Die `else`-Bedingung wird auch hier nicht häufig eingesetzt, kann aber gelegentlich recht wertvoll sein.

Die zweite Art der `while`-Anweisung, in der man sich einfach so lange im Kreis dreht, bis eine bestimmte Bedingung zutrifft, wird in Abbildung 4.12 dargestellt.

Es handelt sich dabei zugegebenermaßen um ein konstruiertes Beispiel, aber wir werden später Anwendungen für diese Art der `while`-Anweisung finden, die weniger künstlich sind.

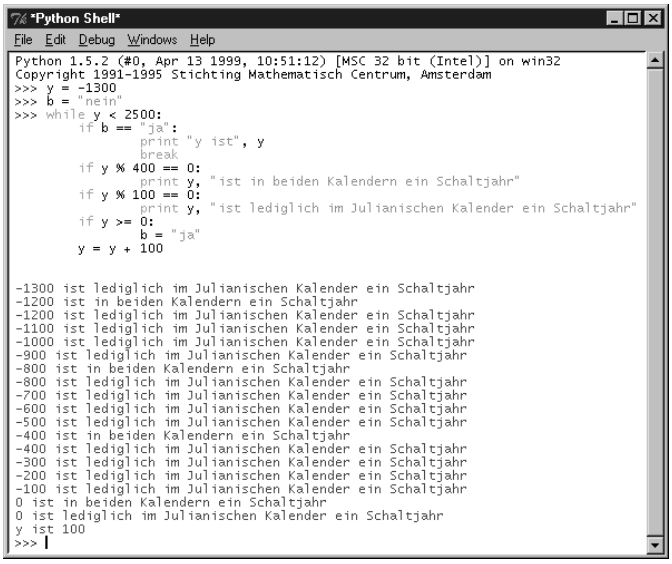


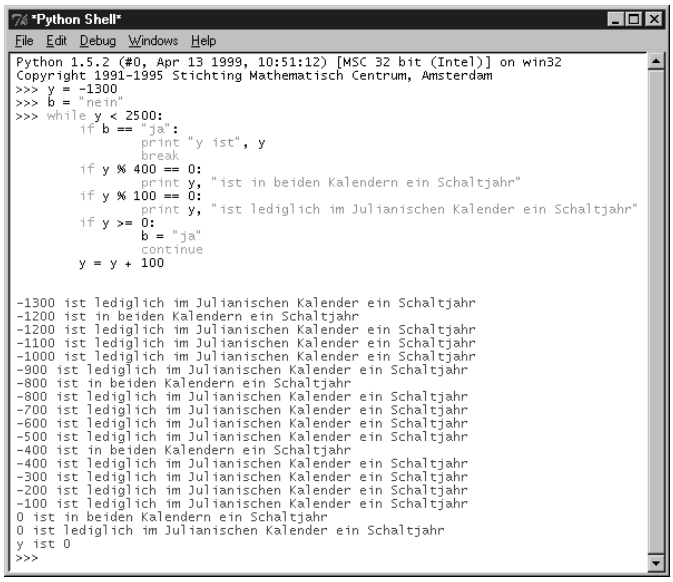
Abb. 4.12: Kombination der Anweisungen `while` und `if`

4.5 break, continue und pass

Wie Sie bereits gesehen haben, dient die `break`-Anweisung dazu, eine `for`- oder `while`-Schleife zu verlassen. In anderen Situationen werden Sie es möglicherweise sinnvoll finden, die Schleife nicht abzubrechen, sondern von vorn mit ihrer Ausführung zu beginnen. Genau das macht die `continue`-Anweisung: Sie springt unmittelbar zum Anfang der übergeordneten `for`-

oder `while`-Schleife, ohne die restlichen Anweisungen des Schleifenblocks auszuführen. Python prüft dann (in einer `while`-Schleife) erneut die Schleifenbedingung oder setzt die Laufvariable (in einer `for`-Schleife) auf den nächsten Wert. Als Beispiel können wir dem Programmcode in Abbildung 4.12 eine `continue`-Anweisung hinzufügen; Sie können dies in Abbildung 4.13 sehen.

Abb. 4.13: Die Anweisungen `while`, `break` und `continue`



Der einzige Unterschied zwischen diesem Beispiel und dem vorangegangenen liegt darin, dass *100* nicht zu *y* addiert wird, weil die `continue`-Anweisung ausgeführt wird. Der abschließende Wert ist hier *0*, während er im vorherigen Beispiel *100* war.

Eine weitere wichtige Anweisung, `pass`, wird in `for`- und `while`-Schleifen nicht häufig eingesetzt; sie kommt hauptsächlich in `class`-Anweisungen vor, die im zweiten Teil dieses Buchs behandelt werden. Die `pass`-Anweisung wird verwendet, um Python anzuzeigen, dass es überhaupt nichts tun soll; der offizielle Begriff für eine Anweisung wie diese ist *no-op*, die Kurzform für *no operation* (keine Operation). Es mag recht befremdlich erscheinen, dass Computer dazu einen speziellen Befehl benötigen, weil sie ohnehin nur Dinge tun, die man ihnen ausdrücklich befiehlt (das ist zumindest der Mythos). In den frühen Tagen der Computerspiele wurden no-ops häufig benutzt, um so genannte *Timer* (Zeitgeber) zu bauen; obwohl eine no-op-Anweisung nichts tut, benötigt sie dennoch Zeit. Daher verwendeten Programmierer `for`-Schleifen, die für eine bestimmte Anzahl von Durchläufen

nichts tun, um sicherzustellen, dass Ereignisse zur richtigen Zeit eintreten, und auf diese Weise die nötige Verzögerung zu geben, sodass ein Ereignis korrekt gestartet werden konnte. No-ops wie `pass` sind nicht mehr so wichtig, wie sie es einmal waren, aber hin und wieder braucht man sie doch. Abbildung 4.14 zeigt den korrekten Gebrauch für `pass` bei einer `if`-Anweisung. Wenn Sie versuchen, ohne einen Rumpf für die `if`-Anweisung auszukommen, werden Sie einen Syntaxfehler erhalten.

```
if test:
else:
    tu etwas
```

Die obige Syntax ist in Python nicht zulässig. Unten sehen Sie die richtige Syntax.

Abb. 4.14: Die Anweisungen `if` und `pass`

4.6 Ein vollständiges Programm

Wie versprochen, folgt ein Listing eines vollständigen Programms unter Verwendung einiger der in diesem Kapitel diskutierten Konzepte.

```
 1    #!/usr/bin/env python
 2    import sys
 3    import string
 4    if len (sys.argv) < 2:
 5        print "Aufruf:  leap.py jahr, jahr, jahr..."
 6        sys.exit (0)
 7    for i in sys.argv[1:]:
 8        try:
 9            y = string.atoi (i)
10        except:
11            print i, "ist keine Jahreszahl."
12            continue
13        leap = "kein"
14        if y % 400 == 0:
15            leap = "ein"
16        elif y % 100 == 0:
17            leap = "kein"
18        elif y % 4 == 0:
19            leap = "ein"
```

Listing 4.1: Vollständiges Programm mit Kontrollflussanweisungen

```
20        else:
21            leap = "kein"
22
23        print y, ": ist", leap, "Schaltjahr des Gregorianischen ↵
                                   Kalenders"
24
25        if y % 4 == 0:
26            leap = "ein"
27        else:
28            leap = "kein"
29
30        print y, ": ist", leap, "Schaltjahr des Julianischen ↵
                                   Kalenders"
31
32    print "Es wurde fuer ", len ( sys.argv ) - 1, "Jahre ↵
          ermittelt, ob sie Schaltjahre sind"
```

Sie können es entweder in einen Editor eingeben und es als Datei unter dem Namen `leap.py` speichern oder es von der Internet-Seite zu diesem Buch herunterladen. Starten Sie es, indem Sie `python leap.py 1900 1904 2000` in eine DOS-Box oder eine Unix-Shell eingeben. Dann probieren Sie es mit falschen Parametern wie `abcde 1900 2000` und sehen Sie, was passiert. Sie werden über das seltsam aussehende `[1:]` später etwas mehr erfahren, und zwar im Zusammenhang mit der `len()`-Funktion und den anderen Funktionen, die Sie zuvor noch nicht gesehen haben.

4.7 Zusammenfassung

Als Teil der Besprechung von Variablen haben Sie gelernt, dass bestimmte Wörter in Python speziellen Zwecken vorbehalten sind. Tabelle 4.1 listet alle von Python reservierten Schlüsselwörter auf.

Tabelle 4.1: Reservierte Schlüsselwörter in Python

and	elif	global	or
assert	else	if	pass
break	except	import	print
class	exec	in	raise
continue	finally	is	return
def	for	lambda	try
del	from	not	while

Seien Sie sich bewusst, dass Sie in Python Namen von Funktionen als Variablennamen einsetzen können; das bedeutet, dass es absolut möglich ist, `int` als Variablennamen zu verwenden, obwohl dies der Name einer Funktion ist, die `int()` geschrieben wird. Abbildung 4.15 zeigt, was geschieht, wenn Sie so verfahren.

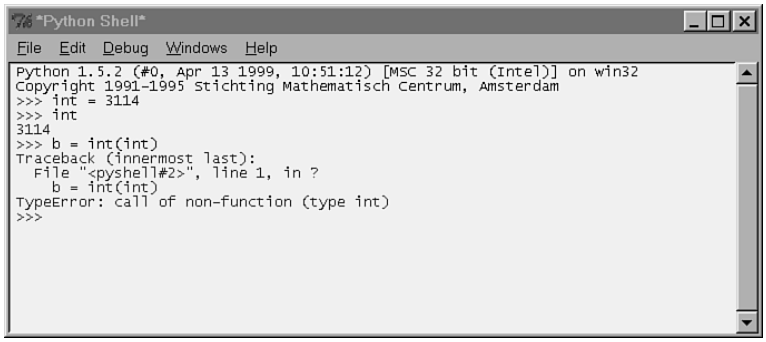


Abb. 4.15: Python ist zu nett für seine eigenen Vorzüge

Funktionsnamen sind in Python lediglich Variablen eines speziellen Typs; sie sind damit nicht vor Missbrauch geschützt.

Sie haben ebenfalls gelernt, die `if`-Anweisung zu benutzen; sie verwendet *Vergleichsoperatoren*, die alle in Tabelle 4.2 zusammengestellt sind.

`a < b`	Wahr, wenn *a* kleiner als *b* ist	`a <= b`	Wahr, wenn *a* kleiner oder gleich *b* ist
`a > b`	Wahr, wenn *a* größer als *b* ist	`a >= b`	Wahr, wenn *a* größer oder gleich *b* ist
`a == b`	Wahr, wenn *a* und *b* exakt gleich sind	`a != b`	Wahr, wenn *a* und *b* ungleich sind
`a is b`	Wahr, wenn *a* dasselbe Objekt wie *b* ist	`a is not b`	Wahr, wenn *a* und *b* nicht dasselbe Objekt sind
`a < b < c`	Wahr, wenn *a* kleiner als *b* ist UND *b* kleiner als *c* ist		Funktioniert für `>` entsprechend
`not a`	Wahr, wenn *a* den Wert `0` hat oder ein »leeres Objekt« bezeichnet (dazu gehören sowohl der spezielle »Nicht-Wert« `None` als auch leere Listen, Tupel, Dictionaries und Leerstrings `""`)	`a or b`	Wenn *a* den Wert `0` hat oder ein »leeres Objekt« bezeichnet, der Wert von *b*; andernfalls der Wert von *a*
`a and b`	Wenn *a* den Wert `0` hat oder ein »leeres Objekt« bezeichnet, der Wert von *a*; andernfalls der Wert von *b*		

Tabelle 4.2: Vergleichsoperatoren in Python

Der in der obigen Tabelle benutzte Begriff »dasselbe Objekt« bedeutet eine Variable, welche dieselbe Information enthält; das heißt, `a is b` ist wahr, wenn man sowohl `a` als auch `b` auf `"hallo"` setzt oder wenn man beiden den Wert `1` zuweist.

In diesem Kapitel haben Sie auch einige weitere grundlegende Kontrollflussanweisungen kennen gelernt: `for`, `while`, `break`, `continue` und `pass`. Sie haben ebenfalls etwas über die Funktion `range()` gelernt, die eine Hilfsfunktion für die `for`-Anweisung ist. Abschließend wurden viele dieser Konzepte in einem vollständigen Programm benutzt.

4.8 Workshop

4.8.1 F&A

F: Wie viele Kontrollflussoperatoren gibt es?

A: Sehr wenige – nur sechs, zu denen übrigens kein `goto` gehört. Mehr braucht man einfach nicht.

F: Was ist mit diesen Doppelpunkten?

A: Sie markieren das Ende einer `if`-Anweisung oder einer anderen Anweisung; es ist absolut akzeptabel, eine `if`-Anweisung wie im folgenden Beispiel zu programmieren:

```
if a < b < c:  print a, b, c
else:  print c, b, a
```

Der Gebrauch des Doppelpunkts ist die einzige Möglichkeit, die `if`-Anweisung und den Rumpf zu trennen. Obwohl ich den Stil, den ich bislang verwendet habe (und auch weiter verwenden werde) für lesbarer halte, ist er eher eine Frage der persönlichen Vorliebe. Allerdings sollte man beachten, dass man den hier genannten Stil nicht mit Einrückungen mischen kann, was insbesondere zu Problemen führt (oder zumindest auch in der lesbarsten aller Sprachen unübersichtlich wird), wenn man mehrere Kontrollflussanweisungen verschachteln möchte.

F: Es ist offensichtlich keine gute Programmierpraxis, Funktionsnamen als Variablennamen zu benutzen, weil Python nicht gut damit zurechtkommt; gibt es Sprachen, die schlau genug sind herauszufinden, was zu tun ist, wenn der Programmierer zum Beispiel int oder range als Variablennamen benutzt?

A: Es ist Ansichtssache, ob das klug ist, aber es gibt tatsächlich welche. Es gibt aber auch gute Gründe, solche Sprachen nicht zu benutzen.

F: Was geschieht, wenn man von seinem Programm möchte, dass es an irgendeine Stelle springt und niemals zurückkehrt? Könnte man in einem solchen Fall eine goto-Anweisung einsetzen?

A: Ja, man könnte, aber dann wäre es unmöglich, sich daran zu erinnern, von wo man kam.[2] Das Problem mit `goto`-Anweisungen ist nicht so sehr, wohin man geht, sondern, dass man keine Ahnung hat, wie man dorthin gekommen ist.

4.8.2 Quiz

1. Was sind Variablen?

   a) *x* und *y*.

   b) Zahlen, die sich zufällig verändern, wenn man nicht hinsieht.

   c) Namen, die für Werte stehen, die man ändern kann.

   d) Sterne, die ihre Helligkeit verändern.

2. Was ist der technische Begriff für den *tu-etwas*-Teil einer `if`-, `for`- oder `while`-Anweisung?

   a) Rumpf.

   b) Block.

   c) Rumpfblock.

   d) Fußangel.

3. Was bedeutet »a > b < c«?

   a) Es ist kein legaler Python-Ausdruck und daher bedeutungslos.

   b) Kombiniere die Werte von *a* und *c* und lege das Ergebnis in *b* ab.

   c) Vergleiche die drei Werte und falls *b* kleiner als die anderen beiden ist, so ist der Ausdruck wahr.

   d) Gebe *b* nur dann aus, wenn *a* und *c* größer sind.

---

2 Anm. d. Übers.: Eine Lösung dieses Problems könnte die erstmalig 1973 in der Zeitschrift *Datamation* von R. Lawrence Clark vorgeschlagene Anweisung `COME FROM` darstellen ... (ein Nachdruck des entsprechenden Aufsatzes findet sich in einer April-Ausgabe der *Communications of the ACM*: R. Lawrence Clark. *A Linguistic Contribution to GOTO-less programming*, in: Communications of the ACM, April 1984).

4.8.3 Antworten

1. c. Variablen sind Namen für Werte, die man verändern kann.
2. a. Der richtige Begriff für *tu-etwas* ist Rumpf; in Python haben alle Programmzeilen eines *Blocks* die gleiche Einrückung.
3. c. Vergleiche die drei Werte und falls *b* kleiner als die anderen beiden ist, so ist der Ausdruck wahr. Dabei ist man nicht auf zwei Vergleiche beschränkt. Es ist absolut zulässig, zu sagen `if a < b < c < d < …` und die Vergleichskette nach Belieben fortzusetzen.

4.9 Übungen

Überdenken Sie die komplette Schaltjahrregel in Listing 4.1 noch einmal und finden Sie eine bessere Formulierung, die `while` anstelle von `for` einsetzt. Es gibt allerdings keinen Grund, Programmcode zu ersetzen, wenn Sie darin keine Verbesserung sehen.

Suchen Sie im Internet nach Verweisen auf die allerersten Computer und finden Sie heraus, ob es irgendwelche Informationen über deren Programmiersprachen gibt. Finden Sie heraus, welche Anweisungen zur Steuerung der Logik es bei diesen frühen Sprachen und Computern gab.

Kapitel 5

Einfache Datentypen I: die nummerischen Datentypen

Wenn ihr den Sinn für Wiederholung verliert, dann wird eure Praxis recht schwierig werden.

Shunryu Suzuki, Zen-Geist, Anfänger-Geist

In den vergangenen vier Kapiteln haben Sie sich mit einigen Grundlagen beschäftigt und Variablen kennen gelernt. Aber was wird in diesen Variablen abgelegt? Sie haben festgestellt, dass Sie so ziemlich alles in einer Variable speichern können: Zahlen, Strings und sogar Funktionen. Muss es dafür nicht irgendwelche Regeln geben? Ja, sicher, es gibt tatsächlich welche und das Thema dieses und des nächsten Kapitels werden die Datentypen und die Regeln sein, nach denen Python mit solchen Typen umgeht. In diesem Kapitel werden wir die folgenden nummerischen Datentypen diskutieren:

- Integer,
- lange Integer,
- Gleitkommazahlen,
- komplexe Zahlen.

5.1 Datentypen

Computer haben es schwer herauszufinden, was Programmierer von ihnen wollen, solange sie keine Anhaltspunkte haben. Python benutzt daher die Typisierung von Daten. Das bedeutet, dass alles, was Sie in einer Variablen ablegen, einen Typ hat. Die Kategorien, in die sich Daten aufteilen lassen,

heißen *Datentypen* und dazu zählen: nummerische (numeric), Sequenz- (sequence), Funktions- (function) und benutzerdefinierte (user-defined) Datentypen. Die benutzerdefinierten Typen werden im mittleren Drittel dieses Buchs behandelt. Nummerische Typen werden in diesem, die anderen im nächsten Kapitel besprochen.

Datentypen werden in Python (wie in den meisten Programmiersprachen) in zwei große Kategorien eingeteilt: einfache (oder primitive) und benutzerdefinierte Datentypen. Die zweite Kategorie wird oftmals als Objekt bezeichnet und wir werden wiederum später in diesem Buch etwas über Objekte erfahren. Objekte sind ein sehr bedeutendes Konzept in der Programmierung, aber die Unterscheidung zwischen Objekten und einfachen Datentypen ist nicht immer klar, besonders im Falle von Funktionen, und es stimmt auch, dass nicht jedes Objekt auch ein benutzerdefinierter Datentyp ist.

In diesem und im nächsten Kapitel werden wir uns ausschließlich mit den einfachen Datentypen beschäftigen, d.h. den *nummerischen* Typen, den *Sequenz*-Typen und dem Typ *Dictionary*. Funktionen werden in Kapitel 7 besprochen und andere benutzerdefinierte Typen oder Objekte werden in Teil II dieses Buchs behandelt. Die beiden ersten einfachen Datentypen werden weiter unterteilt und wir werden uns als Nächstes mit dieser Unterteilung beschäftigen.

5.2 Nummerische Typen

Computer sind nicht schlau. Man muss in den meisten Computersprachen ausdrücklich angeben, welcher Art eine Zahl ist, die man benutzen möchte. Übersetzte (kompilierte) Sprachen müssen im Voraus informiert werden; das heißt, man muss einen Variablentyp definieren, bevor man einer Variablen einen Wert zuweisen kann; danach führt die Zuweisung eines Werts vom falschen Typ in der Regel zu einem Compiler-Fehler. Nehmen Sie einmal an, dass Sie in C eine Variable wie folgt definieren:

```
int n;
```

Das bedeutet, dass die Variable *n* Integer enthalten soll – positive oder negative ganze Zahlen. Wenn Sie später in Ihrem Programmcode Folgendes probieren:

```
n = 3.14159;
```

wird sich der C-Compiler beschweren und diese Zuweisung nicht erlauben. Python hingegen lässt Sie zum Glück den Variablentyp ohne Beschwerden ändern. Das liegt daran, dass Python mit *dynamischer Typisierung* arbeitet, während C *statische Typisierung* verwendet. Das bedeutet in einfachen

Worten, dass Sie, sobald Sie eine Variable in C als einen bestimmten Typ deklariert haben, diesen nicht mehr ändern können. In Python brauchen Sie keinen Typ zu deklarieren; wenn Sie eine Variable erstellen, indem Sie ihr einen Wert zuweisen, können Sie den Variablentyp so oft ändern, wie Sie wollen. Sie können dies in der Tat so oft tun, dass ein Programm dadurch unlesbar wird. Das machen allerdings nur wenige Leute; da Python eine ausgesprochen lesbare Sprache ist, hat niemand ein Interesse daran, ein Programm undurchsichtig zu machen.

Es ist keine gute Idee, den Typ einer Variablen jedes Mal zu ändern, aber warum ist es dann erlaubt? Wozu dient dynamische Typisierung? Warum bringt man Programmierer nicht dazu, von vornherein anzugeben, was ein Ding sein soll, und sorgt dann dafür, dass sie sich auch daran halten? Das ist es schließlich, was die meisten anderen Computersprachen tun.

Python dagegen geht von der Annahme aus, dass Sie als Programmierer wissen, was Sie tun. Statische Typen in anderen Beispielen sind ein Beispiel für die umgekehrte Annahme – dass Sie vor sich selbst in Schutz genommen werden müssen. Dynamische Typisierung kann, wenn sie nicht missbraucht wird, wesentlich zur Verringerung von Tastenanschlägen beitragen und ist daher sehr angenehm. Nehmen Sie zum Beispiel an, Sie lesen Strings, die ein Anwender eingegeben hat; Sie möchten diese Strings in Zahlen umwandeln und als Eingabewerte für Ihr Programm verwenden. Wenn es dem Anwender gestattet ist, Integer, Gleitkommazahlen und lange Integer einzugeben, müsste man in einer gebräuchlichen Programmiersprache mindestens drei Variablen unterschiedlichen Typs bereitstellen. In Python benötigen Sie dagegen nur eine Variable, die jeden der drei zulässigen Typen enthalten kann; das ist einfach, weil jede Variable jeden Typ beinhalten kann.

Wir beginnen unsere Diskussion mit einem Typ, der Ihnen bereits bekannt ist: *Integer*.

5.3 Integer

Integer sind, wie Sie aus früheren Kapiteln wissen, einfach ganze Zahlen: positive, negative und Null. Wie in anderen Programmiersprachen, leiden Integer in Python an ihrer begrenzten Länge; die meisten Sprachen implementieren Integer als Einheiten zu 32-Bit, was im Falle von Python bedeutet, dass es Integer-Werte außerhalb des Bereichs von `-maxint-1` bis `+maxint` nicht gibt. Sie können die Größe dieses Bereichs ermitteln, indem Sie Python, je nach Ihrer persönlichen Vorliebe, im Befehlszeileneditor oder mit IDLE starten, das Modul `sys` importieren und den Wert von `maxint` ab-

fragen. Listing 5.1 zeigt, womit Sie auf einem normalen PC rechnen können; auf meinem Unix-System erhalte ich dieselbe Ausgabe.

Listing 5.1: Die Addition von 1 zu `maxint`

```
1   Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit
    (Intel)] on win32
2   Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
3   >>> from sys import *
4   >>> maxint
5   2147483647
6   >>> -maxint
7   -2147483647
8   >>> maxint + 1
9   Traceback (innermost last):
10     File "<pyshell#3>", line 1, in ?
11       maxint + 1
12   OverflowError: integer addition
13   >>>
```

Es existieren einige Systeme, die 64-Bit-Integer benutzen, und ich glaube, dass es auf Cray-Rechnern 128-Bit-Integer gibt. Der Fehler, den Sie im obigen Listing erkennen können, `OverflowError`, würde auf diesen Systemen dennoch eintreten, aber der Bereich von `-maxint` bis `+maxint` wäre wesentlich größer. Ich habe keinen Zugang zu Rechnern mit diesen großen Integern, aber der Gebrauch von Pythons langem Integertyp ist ein einfacher Weg herauszufinden, wie groß diese Bereiche auf solchen Rechnern wären. Listing 5.2, `inttest.py`, sollten Sie in einer Datei abspeichern und mit Python ausführen. Sie können diesen Code wie jeden anderen in diesem Buch auch von der Internet-Seite zu diesem Buch herunterladen, anstatt ihn einzugeben.

Listing 5.2: Code zur Handhabung großer `maxints`

```
1   from sys import *
2   z = long ( maxint )
3   z1 = -z
4   print "32-Bit-Rechner:"
5   print "maxint:", z, "-maxint:", -z, "(2 ** 31L) -1:",
    (2  ** 31L) -1
6   print "-------------------------------------------------------
    ------------"
7   y = (2 ** 63L) - 1
8   print "64-Bit-Rechner:"
9   print "maxint:", y, "-maxint:", -y
10  print "-------------------------------------------------------
    ------------"
11  z = (2 ** 127L) - 1
12  print "128-Bit-Rechner:"
13  print "maxint:", z, "-maxint:", -z
```

Abbildung 5.1 zeigt, was geschieht, wenn Sie `inttest.py` ausführen.