Einführung ins Programmieren mit Python

Thomas Grischott
KSS

18. Dezember 2005

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Über dieses Tutorium

Möglicherweise hast du noch nie programmiert. Dieses Tutorium möchte dir dabei helfen, es zu lernen. Allerdings gibt es eigentlich nur einen Weg, programmieren zu lernen. *Du* musst Programme lesen, und *du* musst Programme schreiben. In diesen Unterlagen findest du verschiedenste Beispiele von Pythonskripten. (Man spricht eher von Pythonskripten als von Pythonprogrammen.) Du solltest sie eintippen und beobachten, was passiert. Spiele mit dem Programmtext, mache Änderungen. Das Allerschlimmste, was passieren kann, ist, dass dein Skript nicht mehr funktioniert.

Programmtext wird hier immer so dargestellt:

```
# Python ist leicht
print "Hallo Welt!"
```

Auf diese Art kann man ihn leicht vom übrigen Text unterscheiden. Zur allgemeinen Verwirrung werden aber auch Ausgaben vom Computer in der gleichen Schrift dargestellt.

Auf zu wichtigeren Dingen! Um in Python programmieren zu können, braucht man die richtige Software. Falls Python auf deinem Computer bereits installiert ist, kannst du den nächsten Abschnitt überspringen.

1.2 Python installieren

Python kann von der Webseite http://www.python.org/download für verschiedenste Plattformen heruntergeladen werden. Die aktuelle Version ist 2.3.3. Die Version für den Macintosh findet man unter http://homepages.cwi.nl/~jack/macpython/download.html.

Die Installationsdatei kann einfach durch Doppelklick geöffnet werden. Nach der erfolgreichen Installation startest du die Entwicklungsumgebung IDLE ebenfalls durch Doppelklick.

(IDLE steht übrigens für „Integrated Development Environment“, und das L erinnert an Eric Idle von der englischen Komikergruppe „Monty Python“.)

1.3 Python interaktiv

In der Entwicklungsumgebung IDLE – manche sagen auch „Python GUI“ – findest du ein Fenster mit etwa dem folgenden Text:

```
Python 2.3 (#1, Sep 13 2003, 00:49:11)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1495)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

Mit dem Zeichen `>>>` zeigt Python den interaktiven Modus an. Was im interaktiven Modus eingetippt wird, führt der Pythoninterpreter sofort aus. Gib versuchsweise `1+1` ein. Python antwortet mit `2`. Der interaktive Modus dient zum schnellen und einfachen Testen von einzelnen Befehlen und kurzen Programmsequenzen. Falls dir die Wirkung eines neuen Befehls einmal nicht klar ist, probiere ihn am besten im interaktiven Modus aus.

1.4 Pythonskripte schreiben und ausführen

Wechsle in die IDLE. Wähle im Menu `File` den Eintrag `New Window`. Gib in diesem Fenster zum Beispiel folgendes ein:

```
print "Hallo Welt!"
```

Speichere das Prögrämmchen via Menu `File`, Eintrag `Save`. Wähle als Name hallo.py. (Wer will kann auch noch das Verzeichnis ändern.) Erst jetzt, nachdem das Skript gespeichert ist, kann es ausgeführt werden.

Rufe im Menu `Edit` den Eintrag `Run script` auf. Es erscheint die Ausgabe `Hallo Welt!` im `*Python Shell*`-Fenster.

Eine kurze Einführung in die Benutzung der IDLE und Hilfe bei allfälligen Unklarheiten bietet auch die Webseite http://hkn.eecs.berkeley.edu/~dyoo/python/idle_intro/index_ger.html.

1.5 Python via Kommandozeile

Mit der Eingabe `python` (ohne Argumente) startet man aus der Konsole (Shell-Fenster, DOS-Fenster) heraus den interaktiven Modus. Um ein Pythonskript via Kommandozeile auszuführen, wird es in einem beliebigen Texteditor geschrieben und dann mit dem Kommando `python programmname` gestartet.

Wer die Konsole meiden will, kann das, und verwendet stattdessen einfach die IDLE.

Kapitel 2

Objektorientiertes Programmieren

2.1 Objekte, Eigenschaften und Methoden

Aus der Sichtweise vieler moderner Programmiersprachen besteht die Welt aus Objekten. Ein Fernsehapparat ist ebenso ein Objekt wie ein Fenster auf einem Computerbildschirm oder eine ganz gewöhnliche Zahl.

Allen diesen Objekten ist gemeinsam, dass sie einen gewissen Zustand und ein bestimmtes Verhalten besitzen.

Der Zustand eines Fernsehapparates kann zum Beispiel durch das gewählte Programm oder durch die eingestellte Lautstärke beschrieben werden oder auch durch unveränderbare Merkmale wie Farbe, Länge der Bildschirmdiagonale, Anzahl Programmspeicherplätze oder die Fähigkeit, Teletextseiten darstellen zu können (oder eben nicht). Beispiele für Verhaltensweisen, die ein Fernsehapparat an den Tag legen kann, wären etwa das Wechseln des aktuellen Programms, das Erhöhen des Bildschirmkontrasts oder ganz einfach das Ein- und Ausschalten.

Ein Fenster auf einem Computerbildschirm besitzt ebenfalls verschiedenste Merkmale: Länge, Breite, momentane Position auf dem Bildschirm, Sichtbarkeit (Fenster können sich hinter anderen Fenstern „verstecken“ oder gar ausgeblendet sein), Fenstertitel usw. Ein Fenster kann sich öffnen, verschieben, verkleinern, ausblenden und viele weitere Aktionen ausführen.

Sogar eine Zahl besitzt Merkmale, sogar so offensichtliche, dass man sie fast übersieht. Der Wert der Zahl ist so ein Merkmal, ihr Typ (ganzzahlig, negativ, rational, ...) oder – im Fall von Dezimalbrüchen – die Anzahl Nachkommastellen. Schliesslich kann eine Zahl sich mit einer anderen multiplizieren oder die Anzahl ihrer Dezimalen ändern.

Statt von Merkmalen und Verhaltensweisen spricht man in objektorientierten Kreisen von *Eigenschaften* (oder *Attributen*) und von *Methoden*.

2.2 Nachrichten

Ein Objekt gibt in der Regel seinen Zustand nicht von sich aus preis, sondern speichert seine Merkmalsausprägungen geheimnistuerisch in sogenannten *Zustandsvariablen* (oder *internen Variablen*). Die Anzahl Programmspeicherplätze eines Fernsehapparates ist zum Beispiel nicht offensichtlich, und auch die auf Pixel genaue Höhe eines Fensters auf einem Computerbildschirm ist nicht ohne weiteres ersichtlich.

Wer etwas über den Zustand eines Objekts wissen will, muss dieses danach fragen! Und wenn ein Objekt seinen Zustand ändern soll, muss es dazu aufgefordert werden! (Im Beispiel des Fernsehapparates passieren die meisten dieser Aufforderungen in der Regel über die Fernbedienung.)

So eine Frage oder Aufforderung geschieht in Form einer *Nachricht* an das Objekt. Ein Objekt verfügt für jede Nachricht, die es versteht, über eine Methode, in welcher festgelegt („programmiert“) ist, wie das Objekt auf die betreffende Nachricht reagiert. Auf die via Fernbedienung erhaltene Nachricht „schalte dich aus!“ reagiert der Fernsehapparat (hoffentlich) mit der Ausführung seiner Methode „Ausschalten“.

Normalerweise reagiert ein Objekt auf den Empfang einer Nachricht durch das Versenden einer weiteren Nachricht an ein anderes Objekt. Ein Bildschirmfenster, das die Nachricht „vergrössere dich!“ erhalten hat, ändert seine inneren Variablen entsprechend und benachrichtigt dann die anderen Fenster über seinen neuen „grösseren“ Zustand. Diese reagieren wiederum durch Zustandsänderungen (ein jetzt überdecktes Fenster ändert zum Beispiel die Ausprägung seines Merkmals „Sichtbarkeit“ auf „unsichtbar“) oder durch Versenden weiterer Nachrichten.

Objektorientieres Programmieren bedeutet also, Softwareobjekte zu entwickeln, die den Eigenschaften und Verhaltensweisen realer Objekte nahe kommen. Ein Programm ist dann eine geordnete, zielgerichtete Abfolge von Nachrichten, die an geeignete Objekte dieser Modellwelt gesendet werden. Das Verhalten der Objekte als Reaktion auf den Empfang von Nachrichten kann in einer Zustandsänderung der Objekte oder auch im Versenden weiterer Nachrichten an andere Objekte bestehen.

2.3 Erste Fingerübungen

(Die folgenden Beispiele können direkt im interaktiven Modus ausprobiert werden.)

Zahlen sind – wie gesagt – Objekte. Wir senden die Nachricht „verrate deinen Typ!“ an verschiedene Zahlobjekte:

```
>>> type(2)
<type 'int'>
>>> type(2.0)
<type 'float'>
```

Ein anderes Merkmal jedes Objektes ist seine Identität. In Python wird jedes Objekt durch eine eindeutige ganze Zahl identifiziert, ähnlich der Nummer einer Identitätskarte:

```
>>> id(2)
25190864
>>> id(2.0)
25361940
```

Die Zahlen 2 und 2.0 sind offenbar verschiedene Objekte!

Schliesslich senden wir einer Zahl noch die Aufforderung, eine andere Zahl zu addieren, zu multiplizieren und sich mit einer anderen Zahl zu vergleichen:

```
>>> 1.0.__add__(2.0)
3.0
>>> 1.5.__mul__(2.5.__add__(3.5))
9.0
>>> 3.0.__cmp__(1.0)
1
>>> 2.0.__cmp__(2.0)
0
>>> 2.0.__cmp__(5.0)
-1
```

Die letzten fünf Befehle zeigen auch, wie in Python (und übrigens in vielen anderen objektorientierten Sprachen) eine Methode eines Objekts aufgerufen wird. Zuerst wird der Name des Objekts angegeben, danach folgt durch einen Punkt abgetrennt der Name der Methode:

```
objektname.methodenname(parameter)
```

Allerdings sind für die meisten von uns wohl die folgenden mathematischen Ausdrücke verständlicher, die Python sozusagen als Entgegenkommen und in Abweichung vom strikt objektorientierten Prinzip des Methodenaufrufs auch erlaubt:

```
>>> 1.0+2.0
3.0
>>> 1.5*(2.5+3.5)
9.0
>>> 3.0>1.0
True
>>> 2.0>5.0
False
```

In den folgenden Kapiteln werden die soweit erläuterten Konzepte des objektorientierten Programmierens vorerst etwas in den Hintergrund treten. Zuerst müssen einige grundlegende Befehle, Kontroll- und Datenstrukturen eingeführt werden. Im Kapitel „Klassen“ (Kapitel 9) werden wir dann den Faden wieder aufnehmen.

Kapitel 3

Ausgabe und Eingabe

3.1 Schreiben, Rechnen, Lesen

Den Befehl `print` hast du bereits im Abschnitt 1.4 kennengelernt, um das „Hallo Welt!“-Skript zu schreiben, mit dem seit Anbeginn der Zeit jede Programmiereinführung beginnt. Der `print`-Befehl schreibt Ausgaben auf den Bildschirm. Man kann ihn auch dazu verwenden, einen 2000-Franken-Computer in einen 5-Franken-Taschenrechner zu verwandeln:

```
>>> print "2+2 ist", 2+2
2+2 ist 4
>>> print "3*4 ist", 3*4
3*4 ist 12
>>> print 100-1, "= 100-1"
99 = 100-1
>>> print "(33+2)/5+11.5 =", (33+2)/5+11.5
(33+2)/5+11.5 = 18.5
```

Zeichenketten (also Objekte vom Typ „String“) müssen zwischen einfache oder doppelte Anführungszeichen gesetzt werden. Bei Zahlen ist dies nicht nötig. Durch Komma getrennte Ausdrücke werden auf der gleichen Zeile nacheinander ausgegeben.

Python kennt übrigens die folgenden Rechenoperationen:

Operation	Symbol	Operation	Symbol	Operation	Symbol
Addition	`+`	Multiplikation	`*`	Potenzieren	`**`
Subtraktion	`-`	Division	`/`		
		Rest	`%`		

Beachte vor allem, wie sich der Divisionsoperator verhält:

```
>>> print "14/3 =", 14/3
14/3 = 4
```

```
>>> print "14.0/3 =", 14.0/3
14.0/3 = 4.66666666667
>>> print "14/3. =", 14/3.
14/3. = 4.66666666667
>>> print "5+14%3 =", 5+14%3
5+14%3 = 7
```

Python antwortet unterschiedlich, je nachdem, ob Nachkommastellen angegeben werden! Ausserdem wird die übliche Hierarchie der Operatoren berücksichtigt.

Mit den Befehlen `input` und `raw_input` kann der Benutzer aufgefordert werden, eine Zahl bzw. eine Zeichenkette einzugeben:

```
>>> input("Gib eine Zahl ein: ")
Gib eine Zahl ein: 5
5
>>> input("Gib eine Zahl ein: ")
Gib eine Zahl ein: sieben
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<string>", line 0, in ?
NameError: name 'sieben' is not defined
>>> raw_input("Gib einen Text ein: ")
Gib einen Text ein: bliblablo
'bliblablo'
```

Der Befehl `input` akzeptiert offensichtlich nur die Eingabe einer Zahl. Auf die Eingabe einer Zeichenkette reagiert er mit einer Fehlermeldung.

3.2 Variablen

Natürlich ist es nicht sehr spannend, wenn Python die Eingabe einfach wieder auf den Bildschirm schreibt. Stattdessen wird man die Eingabe normalerweise einer Variablen zuweisen:

```
>>> name = raw_input("Wie lautet dein Name? ")
Wie lautet dein Name? John
>>> name
'John'
```

Das System legt eine Variable `name` an, welche für die Zeichenkette `'John'` steht. Eine Variable stellt man sich am besten als (Anfang von einem) Pfeil vor, der auf ein bestimmtes Objekt zeigt (hier eben auf `'John'`). Würde nun der Befehl `name = "Eric"` eingegeben, dann würde die Pfeilspitze vom Objekt `'John'` gelöst und neu auf das Objekt `'Eric'` gesetzt. Das Objekt `'John'` würde – da kein Pfeil mehr darauf zeigt – überflüssig und von der systemeigenen „Müllabfuhr“ gelöscht.

Damit ein Skript möglichst einfach les- und verstehbar ist, sollten möglichst aussagekräftige Variablennamen gewählt werden. Üblicherweise schreibt man sie klein.

3.3 Beispiele

Mit Hilfe von Variablen können wir bereits kompliziertere Dialoge mit dem Computer führen. Die folgenden Beispielskripte sollten nicht mehr im interaktiven Modus ausprobiert sondern in einem Texteditor geschrieben und als Dateien (Endung `*.py`) ausgeführt werden (siehe Abschnitt 1.4). Zeilen, die mit einem `#` beginnen, dienen dabei als erklärende Kommentare und werden vom Pythoninterpreter überlesen.

```
# quadervolumen.py
# Berechnung des Volumens eines Quaders
laenge = input("Laenge des Quaders in cm: ")
breite = input("Breite des Quaders in cm: ")
hoehe = input("Hoehe des Quaders in cm: ")
volumen = laenge*breite*hoehe
print "Das Volumen des Quaders betraegt", volumen, "cm^3."
```

Bei meinem ersten Versuch, das Skript zu starten, reklamierte der Interpreter übrigens mit der Meldung

```
  File "quadervolumen.py", line 7
    print "Das Volumen des Quaders betraegt" volumen, "cm^3."
                                              ^
SyntaxError: invalid syntax
```

Ich hatte das erste Komma in der letzten Zeile vergessen! Mit dem `^` gibt der Interpreter jeweils an, wo er das Problem (ungefähr) vermutet.

Achte beim nächsten Beispiel auf die Wirkung von `+` und `*` in Verbindung mit Zeichenketten:

```
# gutenmorgen.py
# ...tut was?...
name = raw_input("Dein Name, bitte? ")
gruss = "Guten Morgen, "+name+"! "      # Konkatenation
gruesse =3*gruss                        # Vervielfältigung
print                                   # Leerzeile
print gruss
print gruesse
```

3.4 Aufgaben

1. Schreibe ein Pythonskript, welches nach der Eingabe von Grösse (in m) und Gewicht (in kg) gemäss der Formel bmi = Masse/Grösse^2 den Body-Mass-Index eines Mannes (oder einer Frau) berechnet.

2. Schreibe ein Pythonskript, das zunächst eine Person interviewt und dann mit Hilfe der gesammelten Informtionen eine Kurzgeschichte schreibt, in der die Person vorkommt. Nimm zur Vereinfachung an, dass die Person weiblich (oder männlich) ist, oder schreibe für jedes Geschlecht eine eigene Version.

   Ein Stück eines möglichen Programmablaufs:

   ```
   Dein Vorname? Thomas
   Dein Wohnort? Andeer
   Deine Haarfarbe? braun?
   ...
   Lehrer Lämpel stand vor der Andeerer Post und schaute ungeduldig auf
   die Uhr. Er wartete seit geraumer Zeit auf seinen Freund Thomas. Sie
   hatten sich zu ihrem wöchentlichen Pythonkränzchen verabredet.
   Endlich tauchte in der Ferne ein brauner Haarschopf auf. Lämpel konnte
   aufatmen.
   ...
   ```

Kapitel 4

Kontrollstrukturen

Kaum ein grösseres Skript wird derart stur Zeile für Zeile abgearbeitet, wie das in unseren bisherigen Beispielen der Fall war. In jedem Programm, das etwas auf sich hält, kommen Verzweigungen und Wiederholungen vor.

4.1 Verzweigungen (`if`)

Wir beginnen mit einer klitzekleinen Tippübung:

```
# betrag.py
x = input("Zahl: ")
if x < 0:
    x = -x
print "Betrag:", x
```

Falls die Bedingung `x < 0` (man sagt auch „logischer Ausdruck“) erfüllt (wahr) ist, wird der eingerückte Anweisungsblock ausgeführt. Andernfalls wird die Anweisung `x = -x` übersprungen und direkt der `print`-Befehl verarbeitet.

Hier ist eine kleine Liste aller Vergleiche, die eine logischer Ausdruck enthalten kann:

Operator	Erklärung
`<`	kleiner als
`<=`	kleiner oder gleich
`>`	grösser als
`>=`	grösser oder gleich
`==`	gleich
`!=`	ungleich
`<>`	ungleich (eine andere Schreibweise für `!=`)

Mehrere Vergleiche können auch mit `and`, `or` oder `not` zu komplizierteren Bedingungen zusammengesetzt werden. (Achte im folgenden Beispiel auch auf die erste Klammer nach dem `if`. Sowas kann nur Python!)

```
# alter.py
alter = input("Dein Alter? ")
if (6 <= alter < 16) or (alter > 65):
    print "Du faehrst zum halben Preis."
```

Nach dem `if`-Block kann ein `else`-Block folgen, der nur dann ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Das folgende kleine Skriptlein fragt nach einem Passwort. Je nachdem, ob das eingegebene Passwort richtig oder falsch ist, reagiert das Programm unterschiedlich:

```
# passwort.py
passwort = raw_input("Passwort eingeben: ")
if passwort != "sesam":
    print "Passwort ist ungueltig!"
    print "Zutritt verweigert!"
else:
    print "Willkommen!"
```

An dieser Stelle muss auf eine ganz besondere Besonderheit von Python hingewiesen werden. Für den Interpreter muss klar sein, welche Befehle zum Beispiel zum `if`-Block gehören. (Sonst weiss er ja nicht, wie viele Zeilen er überspringen muss, wenn die Bedingung nicht erfüllt ist.) In den allermeisten Programmiersprachen werden deshalb die betreffenden Befehle mit speziellen Klammern zu einem Block zusammengefasst. Einrückungen im Programmtext sind dann nur Lesehilfen aber für die korrekte Programmausführung unwesentlich. Nicht so bei Python!! Hier ist es unerlässlich, dass alle Befehle, die einen Block bilden, gleich weit eingerückt werden! Wie weit eingerückt wird, ist dabei unwesentlich, wichtig ist aber, *dass* eingerückt wird und dass alle Befehle eines Blocks *genau gleich weit* eingerückt werden! Versuche einmal, im Passwortskript die Zeile `print "Zutritt verweigert!"` etwas weniger (oder mehr) einzurücken!

Auch Fallunterscheidungen mit mehr als drei Fällen sind kein Problem:

```
# platzkategorie.py
kategorie = raw_input("Platzkategorie (Loge, Balkon, Parkett, Rang)? ")
if kategorie == "Loge":
    preis = 50
elif kategorie == "Balkon":
    preis = 37
elif kategorie == "Parkett":
    preis = 29
elif kategorie == "Rang":
    preis = 20
else:
    preis = 0
if preis > 0:
    print "Deine Eintrittskarte kostet", preis, "Franken."
else:
    print "Kategorie existiert nicht!"
```

Die Bedingung muss nicht notwendigerweise ein Vergleich sein. Probiere zum Beispiel folgendes im interaktiven Modus aus:

```
>>> 'P' in "Python"
True
>>> "thon" in "Python"
True
>>> "ton" in "Python"
False
```

Jedes Objekt besitzt in Python einen Wahrheitswert! Numerische Objekte mit dem Wert Null haben den Wahrheitswert `False`, Zahlen ungleich Null sind `True`. Sequenzen, also zum Beispiel Zeichenketten oder Listen (siehe Abschnitt 5.2.2) besitzen den Wahrheitswert `False`, wenn sie leer sind, und `True`, wenn sie nicht leer sind. Probiere es aus (wieder interaktiv):

```
>>> a = 2
>>> if a:
        print "a ist ungleich Null."

a ist ungleich Null.
>>> wort = "Hallo"
>>> if wort:
        print "Du hast etwas gesagt."

Du hast etwas gesagt.
>>> wort = ""
>>> if not wort:
        print "Sag etwas!"

Sag etwas!
```

4.2 Bedingte Wiederholungen (`while`)

Hast du etwa gedacht, schlimmer könne es nicht mehr werden, nachdem wir deinen Computer als 5-Franken-Taschenrechner missbraucht haben? Hier ist ein ähnlich anspruchsvolles Programm:

```
# quadratzahlen.py
n = 1
while n <= 10:
    print n**2
    n = n+1
```

Der eingerückte Block, das Schleifeninnere, wird so lange wiederholt, wie die Bedingung nach dem Schlüsselwort `while` erfüllt ist. Wiederum ist wichtig, dass alle Anweisungen innerhalb des Schleifenblocks gleich weit eingerückt werden!

Was tut wohl das nächste Beispiel?

```
# summe.py
a = -1
summe = 0
print "Gib mehrere Zahlen ein! (Weiter mit 0)"
while a != 0:
    print "Zwischensumme:", summe
    a = input("Zahl? ")
    summe += a  # Kurzform fuer summe = summe + a
print "Die Summe aller Zahlen ist", summe, "."
```

Die Passwortabfrage aus dem letzten Abschnitt kann noch verbessert werden:

```
# passwort2.py
passwort = ""
while passwort != "sesam":
    passwort = raw_input("Passwort, bitte: ")
print "Willkommen!"
```

Beachte, dass die Variable `passwort` bereits existieren muss, sobald sie mit dem wahren Passwort `sesam` verglichen wird. Darum wird ihr in der ersten Zeile eine leere Zeichenkette zugewiesen.

Wir sind schon fast am Ende des Abschnitts angekommen. Versuche zum Schluss folgende Anweisung im interaktiven Modus:

```
>>> while True:
        print "Hilfe, ich stecke fest!"
```

So etwas nennt man eine Endlosschleife. Man kann sie (wie jedes laufende Programm) mit der Tastenkombination Ctrl-c brutal beenden. Endlosschleifen können aber – in Verbindung mit einer `break`-Anweisung – auch als „Programmiertrick“ bewusst eingesetzt werden, wie das letzte Beispiel in diesem Abschnitt zeigen soll:

```
# eingabe.py
print "Gib eine Zahl zwischen 1 und 10 ein!"
while True:
    zahl = input("Zahl: ")
    if 1 <= zahl <= 10:
        break   # beendet die Ausfuehrung der Schleife
    else:
        print "Zahl passt nicht."
print "Danke schoen!"
```

4.3 Iterationen (`for`)

Die Ausgabe unseres Einstiegsbeispiels für diesem Abschnitt sollte dir seltsam vertraut vorkommen:

```
# quadratzahlen2.py
for n in range(1,11):
    print n**2
```

Um das Skript besser zu verstehen, solltest du zuerst die `range()`-Funktion interaktiv kennen lernen. Experimentiere, bis du die Wirkung der (bis zu) drei Parameter verstehst.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(10,0,-2)
[10, 8, 6, 4, 2]
```

Beachte, dass der Endwert jeweils gerade nicht mehr in der Liste vorkommt! (Mehr über Listen gibt's im Abschnitt 5.2.2.) Im obigen Beispiel nimmt `n` also nacheinander alle Werte aus der Liste `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` an. Man sagt auch, `n` durchlaufe die Liste. In jedem Schritt wird das Quadrat von `n` ausgegeben.

Das nächste Beispiel berechnet die ersten zwanzig Fibonaccizahlen. Die Kurzschreibweisen für mehrere Zuweisungen sind eine elegante Spezialität von Python!

```
# fibonacci.py
a = b = 1   # Abkuerzung fuer a = 1; b = 1
for i in range (1,21):
    print a,
    a, b = b, a+b   # Abkuerzung fuer a = b; b = a+b
```

Die Sequenz, welche die Variable durchläuft, muss nicht unbedingt eine Liste von Zahlen sein. Im nächsten Beispiel wird über die einzelnen Buchstaben einer Zeichenkette iteriert:

```
# besprache.py
text = raw_input("Bitte Text eingeben: ")
betext = ""
for buchstabe in text:
    betext = betext + buchstabe
    if buchstabe in "AEIOUaeiou":
        betext = betext + 'b' + buchstabe.lower()
print betext
```

Die Methode `lower()` einer Zeichenkette liefert übrigens eine Kopie der Zeichenkette aus lauter Kleinbuchstaben. Lass das `.lower()` in der zweitletzten Zeile doch versuchsweise einmal weg, und gib einen Text ein, der auch Grossbuchstaben enthält.

Alles, was `for`-Schleifen können, kann man grundsätzlich auch mit `while`-Schleifen erreichen. `for`-Schleifen sind aber eine einfache Alternative, wenn die Anzahl Iterationen zum vornherein feststeht.

4.4 Beispiele

Das erste Beispiel prüft, ob eine Zahl gerade, ungerade oder sehr seltsam ist. Teste das Skript mit `2`, `3` und mit `3.1415926535`.

```
# geradeungerade.py
zahl = input("Sag' mir eine Zahl: ")
if zahl%2 == 0:
    print zahl, "ist gerade."
elif zahl%2 == 1:
    print zahl, "ist ungerade."
else:
    print zahl, "ist sehr seltsam."
```

Das zweite Beispiel ist etwas für Wirtschaftler. Es berechnet die Rückzahlung eines Kredits aufgrund der Höhe des Kredits, des Zinsfusses und des Betrags, der jährlich für Zinsen und Tilgung aufgewendet werden soll.

```
# rueckzahlung.py
print "Rueckzahlungsplan fuer einen Kredit"
print
schuld = input("Kreditsumme (in Franken): ")
zinssatz = input("Zinssatz (Prozent pro Jahr): ")
rueckzahlung = input("Rueckzahlung (Franken pro Jahr): ")
print
jahr = 2003
while schuld >= rueckzahlung:
    zinsen = schuld*zinssatz/100
    tilgung = rueckzahlung-zinsen
    schuld = schuld-tilgung
    jahr += 1
    print jahr,
    print " Zinsen:", zinsen, "Franken,",
    print "Tilgung:", tilgung, "Franken,",
    print "Restschuld:", schuld, "Franken"
print jahr+1,
print " Restschuld:", schuld, "Franken"
```

Das dritte Beispiel zeigt schliesslich, wie mit einer `while`-Schleife ein einfaches Menu realisiert werden kann.

```
# flaechen.py
print "Flaechenberechnungen"
wahl = ''
while wahl != 'b':
    print
    print "Waehle:"
    print " 'r' fuer Rechteck"
    print " 'q' fuer Quadrat"
    print " 'k' fuer Kreis"
    print " 'b' fuer Beenden"
    wahl = raw_input("Auswahl: ")
    print
    if wahl == 'r':
        laenge = input("Laenge des Rechtecks: ")
        breite = input("Breite des Rechtecks: ")
        print "Rechtecksflaeche: ", laenge*breite
    elif wahl == 'q':
        seite = input("Quadratseite: ")
        print "Quadratflaeche: ", seite**2
    elif wahl == 'k':
        radius = input("Kreisradius: ")
        print "Kreisflaeche: ", 3.14*radius**2
```

4.5 Aufgaben

1. Im Jahre 1582 wurde von Papst Gregor XII. eine Kalenderreform angeordnet, die auch heute noch gültig ist. Darin wurde festgelegt, dass ein Jahr als Schaltjahr gilt, wenn die Jahreszahl ohne Rest duch 400 teilbar ist oder wenn die Jahreszahl ohne Rest durch 4 aber nicht durch 100 teilbar ist. (Zum Beispiel sind 2000 und 2004 Schaltjahre, nicht aber 2003 und 1900.)

   Schreibe ein Pythonprogramm, das nach der Eingabe einer Jahreszahl feststellt, ob es sich um ein Schaltjahr handelt. Verwende den `%`-Operator.

2. Schreibe ein Skript, das so lange Zahlen einliest, bis der Benutzer 0 eingibt, und dann den Mittelwert aller eingegebenen Zahlen (ohne die 0) berechnet.

3. Schreibe ein Skript zur Umrechnung von Temperaturen von Grad Celsius nach Fahrenheit ($f = \frac{9.0}{5.0} \cdot c + 32.0$) und umgekehrt. Das Skript soll ein Menu mit je einer Option für die beiden Konvertierungsrichtungen und einer weiteren Option zum Beenden des Programms besitzen.

Kapitel 5

Datenstrukturen

5.1 Übersicht

Python kann mit verschiedenen Arten von Daten umgehen. Zahlen und Zeichenketten kennen wir bereits. Eine Übersicht über die wichtigsten Datentypen (mit jeweils einem Beispiel) liefert die folgende Zusammenstellung.

```
>>> type(123)
<type 'int'>
>>> type(123456789L)
<type 'long'>
>>> type(12.345)
<type 'float'>
>>> type(1+2j)
<type 'complex'>
>>> type(True)
<type 'bool'>
>>> type("Wort")
<type 'str'>
>>> type((1, 'a', [2]))
<type 'tuple'>
>>> type([1, 'a', [2]])
<type 'list'>
>>> type({'A':65, 'B':66})
<type 'dict'>
>>> type(None)
<type 'NoneType'>
```

Keine Angst, wenn du hier noch nicht alles verstehst! Das Kapitel hat ja gerade erst angefangen! Am Ende des Kapitels wirst du (zwar nicht ganz alle aber) die meisten dieser Datentypen kennen.

5.2 Sequenzen

Eine Sequenz entsteht durch Aneinanderreihen von mehreren Objekten. Beispielsweise ist eine Zeichenkette eine Folge von einzelnen Buchstaben.

Die einzelnen Objekte einer Sequenz sind durchnumeriert. Die Nummer eines Objekts heisst „Index“. Das erste Objekt trägt immer den Index 0, das zweite den Index 1, das dritte den Index 2 usw. Über den Index kann man auf die einzelnen Elemente der Sequenz zugreifen:

```
>>> s = "Kaulquappe"
>>> s[0]
'K'
>>> s[1]
'a'
>>> s[-1]   # das letzte Objekt der Sequenz
'e'
>>> s[-2]   # das zweitletzte Objekt
'p'
>>> s[2:6]  # Ausschnitt ("Slice") aus der Sequenz von Index 2 bis Index 5
'ulqu'
>>> s[4:-4]
'qu'
>>> s[4:]   # Slice von Index 4 bis zum Ende der Sequenz
'quappe'
>>> s[:4]   # Slice vom Anfang der Sequenz bis zu Index 3
'Kaul'
>>> s[:]    # komplette Kopie der Sequenz
'Kaulquappe'
```

Zu den Sequenzen zählen neben den Zeichenketten auch Tupel (siehe Abschnitt 5.2.1) und Listen (Abschnitt 5.2.2).

5.2.1 Tupel

Mehrere durch Kommas getrennte und in runde Klammern eingeschlossene Objekte bilden ein Tupel. In der Regel verwendet man diesen Datentyp zur Beschreibung von Objekten, die aus logisch zusammengehörigen Komponenten bestehen. Im ersten Beispiel werden Punkte in einem Koordinatensystem als Tupel dargestellt:

```
>>> punkt = (x, y, z) = (2, 0, -5)
>>> punkt[0]
2
>>> y, z  # Klammern kann man auch weglassen
(0, -5)
>>> max(punkt)
2
```

Und das zweite Beispiel folgt sogleich:

```
>>> name = ("John", "Cleese")
>>> name[-2]
'John'
>>> len(name)
2
>>> min(name)
'Cleese'
```

Die Methoden `len()`, `min()` und `max()` liefern übrigens die Länge, das kleinste bzw. das grösste Element einer beliebigen Sequenz. Bei Zeichenketten gilt dabei die alphabetische Reihenfolge, wobei Grossbuchstaben „kleiner sind“ als Kleinbuchstaben:

```
>>> min('a', 'Z')
'Z'
>>> 'Z' < 'a'
True
```

5.2.2 Listen

Eine Liste entsteht, wenn mehrere Objekte durch Kommas getrennt zwischen eckige Klammern geschrieben werden. Während bei Tupeln die einzelnen Objekte meistens Teile eines Ganzen sind, enthalten Listen in der Regel mehrere unabhängige Objekte gleichen Typs, zum Beispiel die verschiedenen Wochentage:

```
# wochentage.py
wochentage = ["Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag",\
      "Samstag", "Sonntag"]
tag = 0
while not 1 <= tag <= 7:
    tag = input("Welcher Wochentag (1-7)? ")
print "Du hast den", wochentage[tag-1], "gewaehlt."
```

Hast du bemerkt, wie mit `\` eine lange Zeile umgebrochen werden kann? Und ist dir klar, warum in der letzten Zeile `wochentage[tag-1]` und nicht `wochentage[tag]` steht? Genau, die Numerierung der Listenobjekte beginnt ja mit dem Index 0! Für den ersten Tag lautet die Zugriffsanweisung also `wochentage[0]`, für den zweiten Tag `wochentage[1]` usw.

Ein Listeneintrag kann selbst wiederum eine Liste sein:

```
>>> liste = [[1, 2], [3, 4]]
>>> liste[0]
[1, 2]
>>> liste[0][1]
2
```

Listen sind – im Gegensatz zu Zeichenketten und Tupeln – veränderbare Sequenzen. Anweisungen wie `s = "Haus"; s[0] = "M"` oder `t = (0, 1, 2); t[1] = 3` führen zu Fehlermeldungen (ausprobieren!); hingegen ist folgendes erlaubt:

```
>>> liste = ["Spam", "Spam", "Egg", "and", "Spam"]
>>> liste[2] = "Sausage"
>>> liste
['Spam', 'Spam', 'Sausage', 'and', 'Spam']
```

Daher verfügen Listen neben den Operationen, welche auf alle Sequenzen angewendet werden können, über zusätzliche Methoden. Es folgt eine kleine Aufstellung:

Operation	Erklärung
`x in l`	liefert `True`, falls `x` ein Element der Liste `l` ist
`l.count(x)`	gibt an, wie oft `x` in der Liste `l` vorkommt
`l.index(x)`	liefert den Index des ersten Auftretens von `x`
`l.insert(i, x)`	fügt `x` vor dem Element mit dem Index `i` ein
`l.append(x)`	hängt `x` als neues Element am Ende der Liste an
`l.extend(s)`	verlängert die Liste um alle Elemente der Sequenz `s`
`del l[i]`	löscht das Listenelement mit dem Index `i`
`l.pop()`	liefert das letzte Listenelement und löscht es gleichzeitig
`l.reverse()`	kehrt die Reihenfolge der Elemente um
`l.sort()`	sortiert die Liste aufsteigend

Im folgenden, breits recht komplizierten Skript werden verschiedene dieser Operationen verwendet, um über ein einfaches Menu eine Liste zu verwalten:

```
# liste.py
liste = []
wahl = 0
while wahl != 5:
    print 22*'-'
    print "1. Liste anzeigen"
    print "2. Eintrag hinzufuegen"
    print "3. Eintrag entfernen"
    print "4. Eintrag aendern"
    print "5. Beenden"
    print 22*'-'
    wahl = input("Auswahl: ")
    print 22*'-'
    if wahl == 1:
        if len(liste) > 0:
            for i in range(len(liste)):
                print str(i+1)+".", liste[i]  # Bem. 1
        else:
```

```
            print "Liste ist leer"
    elif wahl == 2:
        item = raw_input("Neuer Eintrag: ")
        liste.append(item)
    elif wahl == 3:
        item = raw_input("Zu loeschender Eintrag: ")
        if item in liste:
            index = liste.index(item)       # Bem. 2, 3
            del liste[index]                # Bem.    3
        else:
            print "Eintrag nicht gefunden"
    elif wahl == 4:
        altesItem = raw_input("Zu aendernder Eintrag: ")
        if altesItem in liste:
            neuesItem = raw_input("Neu: ")
            index = liste.index(altesItem)  # Bem. 2
            liste[index] = neuesItem
        else:
            print "Eintrag nicht gefunden"
print "Auf Wiedersehen!"
```

Einige Bemerkungen zum obigen Skript:

Bem. 1: Mit dem Befehl `str()` wird eine Zahl in eine Zeichenkette umgewandelt. An diese wird dann mittels `+"."` ohne Abstand ein Punkt angehängt. Würde die Zeile stattdessen `print i+1, ".", liste[i]` lauten, dann würde zwischen der Nummer und dem Punkt ein (unschöner) Leerschlag erscheinen.

Bem. 2: Wer bereits etwas Programmiererfahrung in einer anderen Sprache wie Pascal oder Modula besitzt, lässt sich diese Zeilen auf der Zunge zergehen. In Python ist es nicht nötig, die Liste mit Hilfe einer Schleife mühsam nach einem bestimmten Eintrag zu durchsuchen.

Bem. 3: Mit diesen beiden Zeilen wird jeweils nur der erste von möglicherweise mehreren gleichen Einträgen entfernt. Sollen alle gleichen Einträge gelöscht werden, fügt man vor diesen beiden Zeilen ein `while item in liste:` ein. (Einrücken der beiden folgenden Zeilen nicht vergessen!)

Nach diesem längeren Programm erholen wir uns wieder mit einigen entspannenden Tippübungen im interaktiven Modus.

Studiere zuerst dieses Beispiel mit mehrfach geschachtelten Listen:

```
>>> liste = []
>>> for i in [1, 2, 3, 4, 5]:
        liste = [i, liste]
```

```
>>> liste
[5, [4, [3, [2, [1, []]]]]]
>>> liste[0]
5
>>> liste[1]
[4, [3, [2, [1, []]]]]
>>> liste[1][0]
4
>>> liste[1][1]
[3, [2, [1, []]]]
>>> liste[1][1][0]
3
>>> liste[1][1][1]
[2, [1, []]]
>>> liste[1][1][1][1]
[1, []]
>>> liste[1][1][1][1][1]
[]
```

Die nächsten Beispiele zeigen, wie aus einer vorhandenen Liste eine neue Liste gewonnen werden kann. Aufmerksames Lesen lohnt sich, denn hier erlebst du Python „at it's best“:

```
>>> [i**2 for i in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]
>>> [i**2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [i for i in range(50) if i%7 == 0]
[0, 7, 14, 21, 28, 35, 42, 49]
>>> a = [1, 2, 3, 4]
>>> b = [2, 3, 4, 5]
>>> [i for i in a if i in b]  # Durchschnitt von a und b
[2, 3, 4]
>>> c = ['A', 'B', 'C']
>>> d = [1, 2]
>>> [(i, j) for i in c for j in d]  # karthesisches Produkt von c und d
[('A', 1), ('A', 2), ('B', 1), ('B', 2), ('C', 1), ('C', 2)]
```

Zum Schluss dieses Abschnittes möchte ich wiederum auf eine Besonderheit Pythons (und ähnlicher sogenannter Skriptsprachen) hinweisen. Es geht um sogenannte „flache“ und „tiefe“ Kopien von Listen. Da die Sache etwas anspruchsvoll ist, und nachdem bereits einiges im Abschnitt über Listen „nicht ganz ohne“ war, darfst du den Rest dieses Abschnitts beim erstmaligen Lesen überspringen und direkt auf Seite 28 weiterlesen.

Allerdings solltest du zu gegebener Zeit unbedingt auf die nächsten zwei Seiten zurückkommen!

Aus der Tatsache, dass Listen veränderbare Sequenzen sind, und aus der Vorstellung, die wir uns von Variablen gemacht haben (Pfeile, die auf Objekte zeigen; siehe Abschnitt 3.2), ergibt sich nämlich eine vielleicht überraschende Konsequenz:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b[2] = 5
>>> b
[1, 2, 5]
```

So weit, so gut! Wie sieht nun aber `a` aus? Hättest du folgendes erwartet?

```
>>> a
[1, 2, 5]
```

Die Änderung, die wir an der Liste `b` vorgenommen haben, betrifft auch die Liste `a`! Das liegt daran, dass die beiden „Pfeile“ `a` und `b` auf ein und dasselbe Objekt, nämlich auf die Liste `[1, 2, 3]` zeigen. (Prüfe das mit `id(a)` und `id(b)`!) Eine Änderung an diesem Objekt zeigt sich daher in beiden Variablen.

Soll `b` nicht einfach ein neuer Name für dasselbe Objekt sein, sondern wirklich eine neue Liste bezeichnen, dann kann man mit Hilfe des „Slice“-Operators eine Kopie der Ausgangsliste herstellen:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> id(a)
5249584
>>> id(b)
5248272
```

Und weiter:

```
>>> b[2] = 5
>>> a
[1, 2, 3]
>>> b
[1, 2, 5]
```

Die Listen `a` und `b` sind offensichtlich wirklich zwei verschiedenen Objekte, und eine Änderung am einen Objekt hat keinen Einfluss auf das andere!

Leider wird die Sache noch komplizierter, wenn geschachtelte Listen im Spiel sind. Die “Unterlisten“ einer Liste sind eigenständige Objekte, und die „Haupliste“ enthält nur Verweise („Pfeile“) auf diese Objekte. Wird mit dem „Slice“-Operator eine sogenannte „flache“ Kopie einer geschachtelten Liste erzeugt, dann wird nur die äussere Hauptliste kopiert, nicht aber die Unterlisten.

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> c = [a, b]
>>> c
[[1, 2], [3, 4]]
>>> d = c[:]     # flache Kopie
>>> id(c) == id(d)
False
>>> id(c[0]) == id(d[0])
True
>>> id(c[1]) == id(d[1])
True
```

Die Hauptlisten `c` und `d` sind offenbar verschiedene Objekte, während die Unterlisten `c[0]` und `d[0]` bzw. `c[1]` und `d[1]` identisch sind. Es geht weiter:

```
>>> c[0] = b     # Eine Aenderung an der Hauptliste ...
>>> c
[[3, 4], [3, 4]]
>>> d            # zeigt sich nicht in der flachen Kopie,
[[1, 2], [3, 4]]
>>> c[1][1] = 5  # eine Aenderung an einer Unterliste ...
>>> c
[[3, 5], [3, 5]]
>>> d            # zeigt sich auch in der flachen Kopie.
[[1, 2], [3, 5]]
```

Möchte man eine sogenannte „tiefe Kopie“, also eine von der Ausgangsliste völlig unabhängige Kopie herstellen, die auch in ihren Unterlisten keinen Zusammenhang zum Original mehr hat, müsste man ausdrücklich auch Kopien von den Unterlisten herstellen. Glücklicherweise enthält Python in einem Modul (siehe Kapitel 7) namens `copy` bereits die Methode `deepcopy()`, welche uns genau diese Arbeit abnimmt:

```
>>> import copy
>>> a = [[1, 2], [3, 4]]
>>> b = a[:]              # flache Kopie
>>> c = copy.deepcopy(a)  # tiefe Kopie
>>> a[0] = 0              # Aenderung der aeusseren Hauptliste
>>> a[1][1] = 5           # Aenderung einer inneren Unterliste
>>> a
[0, [3, 5]]
>>> b
[[1, 2], [3, 5]]
>>> c
[[1, 2], [3, 4]]
```

5.3 Dictionaries

Ein Dictionary ist eine Art „Luxusausgabe“ einer Liste. Statt über Indizes greift man bei einem Dictionary aber über beliebige Schlüssel auf die Werte zu:

```
>>> woerterbuch = {"sun":"Sonne", "light":"Licht"}
>>> woerterbuch["sun"]
'Sonne'
```

Die Werte eines Dictionarys können auch Listen (oder sogar wiederum Dictionaries) sein:

```
>>> woerterbuch["light"] = ["Licht", "leicht"]
>>> woerterbuch
{'sun': 'Sonne', 'light': ['Licht', 'leicht']}
```

Im folgenden Skript werden Namen samt zugehörigen Telefonnummern in einem Dictionary gespeichert. Das Programm gleicht stark dem Listenverwaltungsskript auf Seite 23, so dass ausführliche Erklärungen hoffentlich nicht nötig sind. Nur auf zwei Kleinigkeiten soll kurz hingewiesen werden: `\t` steht für einen horizontalen Tabulator, und mit dem Methodenaufruf `telefonbuch.has_key(name)` wird geprüft, ob das Objekt `name` unter den Schlüsseln des Dictionarys `telefonbuch` vorkommt.

Here we go:

```
# telefonbuch.py
telefonbuch = {}
wahl = 0
while wahl != 5:
    print 31*'-'
    print "1. Telefonbuch anzeigen"
    print "2. Eintrag hinzufuegen/aendern"
    print "3. Eintrag loeschen"
    print "4. Nummer nachschlagen"
    print "5. Beenden"
    print 31*'-'
    wahl = input("Auswahl: ")
    print 31*'-'
    if wahl == 1:
        for name in telefonbuch.keys():
            print "Name:", name, "\tNummer:", telefonbuch[name]
    elif wahl == 2:
        name = raw_input("Name: ")
        nummer = raw_input("Nummer: ")
        telefonbuch[name] = nummer
    elif wahl == 3:
        name = raw_input("Name: ")
```

```
        if telefonbuch.has_key(name):
            del telefonbuch[name]
        else:
            print "Eintrag nicht gefunden"
    elif wahl == 4:
        name = raw_input("Name: ")
        if telefonbuch.has_key(name):
            print "Nummer:", telefonbuch[name]
        else:
            print "Eintrag nicht gefunden"
print "Auf Wiedersehen!"
```

Dem besonders aufmerksamen Leser und Ausprobierer sind sicher zwei Dinge aufgefallen: Erstens erscheinen bei der Ausgabe des Telefonbuchs die Einträge in einer anderen Reihnefolge als bei der Eingabe, und zweitens kann ein Schlüssel nur einmal im Dictionary vorkommen. Die Anweisung `telefonbuch[name] = nummer` überschreibt die zu `name` gehörige Telefonnummer, falls der Schlüssel `name` bereits vorkommt.

Zum Schluss dieses Abschnitts findest du hier noch eine Zusammenstellung einiger wichtiger Operationen für Dictionarys. Du wirst nicht alle sofort brauchen, aber vielleicht probierst du trotzdem einige im interaktiven Modus aus?

Operation	Erklärung
`d.has_key(k)`	prüft, ob das Dictionary `d` den Schlüssel `k` enthält
`k in d`	wie `d.has_key(k)`
`d.get(k, x)`	liefert `d[k]`, falls `k in d`, sonst wird `x` zurückgegeben
`d.copy()`	erzeugt eine flache Kopie (s. Seite 26) des Dictionarys
`del d[k]`	löscht den Eintrag zum Schlüssel `k`
`d.clear()`	löscht alle Einträge, zurück bleibt ein leeres Dictionary `{}`
`d.keys()`	liefert eine Kopie der Liste aller Schlüssel von `d`
`d.values()`	liefert eine Kopie der Werteliste von `d`
`d.items()`	kopiert das Dictionary in Form einer Liste von Paaren
`d1.update(d2)`	aktualisiert das Dictionary `d1` mit den Einträgen von `d2`

5.4 Beispiele

Im ersten Beispiel trennt die Methode `s.split()` eine Zeichenkette `s` in eine Liste von einzelnen Wörtern auf. Aus dieser Liste wird die Liste der Wortlängen berechnet, deren Maximum dann zurückgegeben wird:

```
# wortlaengen.py
text = raw_input("Bitte gib einige Woerter ein: ")
laengen = [len(wort) for wort in text.split()]
maxLaenge = max(laengen)
print "Das laengste Wort enthaelt", maxLaenge, "Buchstaben."
```

Im zweiten Beispiel benutzen wir die Methode `s.count(c)`, welche zählt, wie oft das Zeichen `c` in der Zeichenkette `s` vorkommt. Das Skript erzeugt ein Dictionary, in welches für jeden Buchstaben eines eingegebenen Worts (versuche „Erdbeere“!) diese Häufikeit eingetragen wird:

```
# haeufigkeiten.py
wort = raw_input("Bitte gib ein Wort ein: ")
haeufigkeiten = {}
for buchstabe in wort:
    haeufigkeiten[buchstabe] = wort.count(buchstabe)
print haeufigkeiten
```

Das dritte Beispiel realisiert ein englisch-deutsches Wörterbuch, das zu einem englischen Wort mehrere deutsche Übersetzungen (in Form einer Liste) enthalten kann. Die Eingabe von englischen Wörtern und auch von mehreren deutschen Übersetzungen kann mit der Eingabetaste abgebrochen werden. In diesem Fall enthalten die Variablen `englisch` bzw. `deutsch` leere Zeichenketten mit dem Wahrheitswert `False`.

```
# woerterbuch.py
woerterbuch = {}
englisch = raw_input("Englisches Wort: ")
while englisch:
    woerterbuch[englisch] = []
    deutsch = raw_input("Deutsche Uebersetzung: ")
    while not deutsch:              # erzwingt eine erste Uebersetzung
        deutsch = raw_input("Deutsche Uebersetzung: ")
    while deutsch:                  # fuer weitere Uebersetzungen
        woerterbuch[englisch] += [deutsch]
        deutsch = raw_input("Weitere deutsche Uebersetzung: ")
    print
    englisch = raw_input("Englisches Wort: ")
print woerterbuch
```

5.5 Aufgaben

1. Mache dir den Unterschied zwischen `l.append(x)` und `l.extend(s)` klar. Erzeuge dazu die Liste `l = ["John", "Eric", "Terry"]`, und sende ihr die beiden Nachrichten `l.append("Michael")` und `l.extend("Michael")`.

2. Erzeuge auf kurze und elegante Art eine Liste aller bisherigen Schaltjahre. Beginne mit `[i for i in range(1582, 2005) ... ]`, und verwende den `%`-Operator.

3. Ändere das Listenverwaltungsskript auf Seite 23 so ab, dass jeweils *alle* passenden Einträge gelöscht oder geändert werden. Füge ausserdem einen weiteren Menupunkt hinzu, der die Liste sortiert.

4. Was leistet das folgende Skript?

```
# ???.py
l = ["Akkordeon", "Alphorn", "Balalaika",\
     "Bassgeige", "Blockfloete", "Cello", "Dudelsack"]
d = {}
for b in "ABCD":
    d[b] = [w for w in l if w[0] == b]
print d
```

5. Baue das Skript `woerterbuch.py` von Seite 30 zu einem Vokabeltrainer aus. Füge dazu eine Abfrage hinzu, welche der Reihe nach alle englischen Wörter ausgibt und den Benutzer zur Eingabe einer Übersetzung auffordert. Falls diese in der Liste der richtigen Übersetzungen vorkommt, soll das Skript mit „Bravo, richtig!“ antworten. Andernfalls gibt es die Liste der richtigen Übersetzungen aus.

Kapitel 6

Funktionen

6.1 Eigene Funktionen

Wir beginnen ausnahmsweise nicht mit der üblichen Tippübung sondern mit einem Beispiel für etwas, das wir zwar tun könnten aber nicht tun sollten (also für einmal nicht eintippen):

```
# kreisflaechen1.py
radius = 5
flaeche = 3.14*radius**2
print "Flaecheninhalt des ersten Kreises:", flaeche
radius = 10
flaeche = 3.14*radius**2
print "Flaecheninhalt des zweiten Kreises:", flaeche
```

Ein grosser Teil des Skripts wiederholt sich. Programmierer hassen es aber, Dinge wiederholen zu müssen. (Dafür gibt es doch Computer, oder?) Glücklicherweise erlaubt uns Python, die mehrfach verwendeten Teile des Skript in eine eigene Funktion zu verpacken. Hier ist das verbesserte Skript:

```
# kreisflaechen2.py
def kreisflaeche(radius):
    flaeche = 3.14*radius**2
    return flaeche

# Hauptprogramm
flaeche = kreisflaeche(5)
print "Flaecheninhalt des ersten Kreises:", flaeche
flaeche = kreisflaeche(10)
print "Flaecheninhalt des zweiten Kreises:", flaeche
```

Die Ausführung des Skripts startet nicht wie gewohnt mit der ersten Zeile! Stattdessen überspringt der Pythoninterpreter die ersten sechs Zeilen und beginnt direkt mit der Anweisung

`flaeche = kreisflaeche(5)`. Hier passiert nun folgendes: Die Variable `radius`, die in der Funktionsdefinition vorkommt, übernimmt den Wert 5. (Ihre Pfeilspitze wird also auf das Objekt 5 gesetzt.) Nach der Berechnung der Kreisfläche sorgt der `return`-Befehl dafür, dass das Ergebnis der Flächenberechnung an `kreisflaeche(5)` zurückgegeben wird. (Der Pfeil von `kreisflaeche(5)` zeigt dann auf das Ergebnisobjekt.) Die Variable `flaeche` übernimmt schliesslich dieses Ergebnis, und weiter geht's im Hauptprogramm.

Die Variable `radius` aus der Funktionsdefinition nennt man – etwas hochgestochen – auch einen „formalen Parameter". Solche Gesellen können durchaus auch im Rudel auftreten:

```
>>> def oberflaeche(laenge, breite, hoehe):
        flaeche = 2*laenge*breite
        flaeche += 2*laenge*hoehe
        flaeche += 2*breite*hoehe
        return flaeche

>>> oberflaeche(1, 2, 3)
22
```

Andererseits sind auch Funktionen ganz ohne formale Parameter denkbar. Beispiel gefällig?

```
>>> def begruessung():
        print "Hallo, wie geht's?"

>>> begruessung()
Hallo, wie geht's?
```

Auch beim Aufruf einer Funktion ohne Parameter müssen die runden Klammern angegeben werden! Sonst führt Python die Funktion nicht aus, sondern meldet stattdessen einfach, dass so eine Funktion als Objekt irgendwo in seinem Speicher existiert:

```
>>> begruessung
<function begruessung at 0x5078f0>
```

Das letzte Beispiel hat gezeigt, dass eine Funktion nicht unbedingt eine `return`-Anweisung enthalten muss. Funktionen ohne `return`, die also keinen Wert zurückliefern, heissen auch „Prozeduren". Prozeduren werden oft verwendet, um Skripte übersichtlicher zu gliedern.

```
# caesar.py
def eingabe():
    print "Dieses Skript verschluesselt geheime Botschaften,"
    print "so wie es schon Caesar gemacht hat..."
    klartext = raw_input("Gib ein Wort im Klartext ein: ")
    return klartext.upper()
```

```
def verarbeitung():
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZABC"
    geheimtext = ""
    for buchstabe in klartext:
        nummer = alphabet.index(buchstabe)
        nummer += 3
        geheimtext += alphabet[nummer]
    return geheimtext

def ausgabe():
    print "Der Geheimtext lautet:", geheimtext

# Hauptprogramm
klartext = eingabe()
geheimtext = verarbeitung()
ausgabe()
```

Manche Probleme lösen sich fast wie von Zauberhand, wenn man eine Funktion verwendet, die sich selbst aufruft. Solche Funktionen nennt man „rekursive Funktionen“ Das Typische an ihnen ist, dass sie eigentlich nur den einfachsten Fall des Problems lösen können. Füttert man sie mit einem komplizierteren Fall, so vereinfachen sie das Problem lediglich ein wenig. Dann rufen sie zur Lösung des vereinfachten Problems sich selbst ein weiteres Mal auf. (Und dieser Aufruf macht's unter Umständen genau gleich, und der nächste ebenso...!)

```
>>> def fakultaet(n):
        if n == 1:
            return 1
        else:
            return n*fakultaet(n-1)

>>> fakultaet(123)
121463043670253296757662432418812958554542170884833823153289181618292358923621676688311569606126402021707358352212940477825910915704116514721860295199062616467307339074198149529600000000000000000000000000000L
```

Die obige Funktion kann eigentlich nur den kinderleichten Fall $1! = 1$ selbst berechnen. Soll sie die Fakultät einer grösseren Zahl berechnen, dann will sie zuerst „von sich selbst“ wissen, was denn die Fakultät der um eins kleineren Zahl sei. Geschickt, nicht wahr?

6.2 Globale und lokale Variablen

Probiere im interaktiven Modus folgende Zeilen aus, und staune. Die beiden völlig identischen Anweisungen `print "x =", x` geben zwei verschiedene Werte aus!?!

```
>>> def f():
        x = 2
        print "x =", x

>>> x = 1
>>> f()
x = 2
>>> print "x =", x
x = 1
```

Offenbar gibt es ausserhalb und innerhalb der Funktion zwei verschiedene Variablen mit gleichem Namen aber unterschiedlichen Werten. Man nennt das `x`, welches ausserhalb der Funktion den Wert 1 erhält, eine „globale Variable“ und das `x`, dem innerhalb der Funktion der Wert 2 zugewiesen wird, eine „lokale Variable“. Innerhalb der Funktion verdeckt die lokale Variable die globale Variable. Sobald die Ausführung der Funktion beendet ist, hört auch die lokale Variable auf zu existieren, und die globale Variable wird wieder sichtbar.

Neben den Parametern einer Funktion sind alle Variablen lokal, die innerhalb der Funktion auf der linken Seite einer Zuweisung stehen. Deshalb erzeugt die Funktion im nächsten Beispiel lokale Variablen `b` und `d`. Der Variablen `c` wird innerhalb der Funktion jedoch nichts zugewiesen, also bleibt sie global.

Damit du das – zugegebenermassen etwas künstliche – Beispiel nicht eintippen musst, gebe ich anschliessend einen Probelauf wieder. Vergleiche das Skript mit der Ausgabe! Achte insbesondere auf die Fehlermeldung am Schluss. Weil `d` als lokale Variable nur innerhalb der Funktion existiert, ist sie dem Hauptprogramm unbekannt!

```
# variablen1.py
def funktion(a):
    b = 20
    d = c*10
    print "Innerhalb der Funktion hat der formale Parameter a den Wert", a
    print "Innerhalb der Funktion hat die lokale Variable b den Wert", b
    print "Die globale Variable c hat auch innerhalb der Funktion",
    print "den Wert", c
    print "Innerhalb der Funktion hat die lokale Variable d den Wert", d
    return a+1

# Hauptprogramm
a, b, c = 1, 2, 3
print "Vor dem Funktionsaufruf hat die globale Variable a den Wert", a
print "Vor dem Funktionsaufruf hat die globale Variable b den Wert", b
print "Vor dem Funktionsaufruf hat die globale Variable c den Wert", c
print
c = funktion(10)
print
```

```
print "Nach dem Funktionsaufruf hat die globale Variable a den Wert", a
print "Nach dem Funktionsaufruf hat die globale Variable b den Wert", b
print "Nach dem Funktionsaufruf hat die globale Variable c den Wert", c
print "Im Hauptprogramm existiert die lokale Variable d nicht", d
```

Programmablauf:

```
Vor dem Funktionsaufruf hat die globale Variable a den Wert 1
Vor dem Funktionsaufruf hat die globale Variable b den Wert 2
Vor dem Funktionsaufruf hat die globale Variable c den Wert 3

Innerhalb der Funktion hat der formale Parameter a den Wert 10
Innerhalb der Funktion hat die lokale Variable b den Wert 20
Die globale Variable c hat auch innerhalb der Funktion den Wert 3
Innerhalb der Funktion hat die lokale Variable d den Wert 30

Nach dem Funktionsaufruf hat die globale Variable a den Wert 1
Nach dem Funktionsaufruf hat die globale Variable b den Wert 2
Nach dem Funktionsaufruf hat die globale Variable c den Wert 11
Im Hauptprogramm existiert die lokale Variable d nicht

Traceback (most recent call last):
  File "/Users/thomas/Desktop/untitled3.py", line 23, in -toplevel-
    print "Im Hauptprogramm existiert die lokale Variable d nicht", d
NameError: name 'd' is not defined
```

Das Hauptprogramm und alle Funktionen führen selbständig Buch über alle ihnen bekannten globalen und lokalen Variablen und deren aktuelle Werte. Für die Buchhaltung werden ganz normale Dictionaries verwendet. Diese sogenannten „Namensräume“ kann man sich mit den (eingebauten) Funktionen `globals()` bzw. `locals()` anzeigen lassen. Statt der vielen `print`-Anweisungen im obigen Skript hätten wir kürzer schreiben können:

```
# variablen2.py
def funktion(a):
    b = 20
    d = c*10
    print "Lokale Variablen:"
    print locals()
    print
    return a+1

# Hauptprogramm
a, b, c = 1, 2, 3
print "Globale Variablen vor dem Funktionsaufruf:"
print globals()
```

```
print
c = funktion(10)
print "Globale Variablen nach dem Funktionsaufruf:"
print globals()
```

Die Ausgabe des Skripts hätte dann etwa so ausgesehen:

```
Globale Variablen vor dem Funktionsaufruf:
{'a': 1, 'c': 3, 'b': 2, '__builtins__': <module '__builtin__' (built-in)>,
'funktion': <function funktion at 0x4eefb0>, '__name__': '__main__',
'__doc__': None}

Lokale Variablen:
{'a': 10, 'b': 20, 'd': 30}

Globale Variablen nach dem Funktionsaufruf:
{'a': 1, 'c': 11, 'b': 2, '__builtins__': <module '__builtin__' (built-in)>,
'funktion': <function funktion at 0x4eefb0>, '__name__': '__main__',
'__doc__': None}
```

(Uns interessieren nur die Variablen `a`, `b`, `c` und `d`.)

Wir haben bereits gesehen, dass das Hauptprogramm nicht auf lokale Variablen zugreifen kann. Wir wissen auch, dass man umgekehrt aus einer Funktion heraus eine globale Variable lesen kann, sofern es keine lokale Variable gleichen Namens gibt. Allerdings ist es nicht ohne weiteres möglich, innerhalb einer Funktion eine globale Variable zu ändern. Man kann aber mit der Anweisung `global` verhindern, dass eine Variable in den lokalen Namensraum eingetragen wird. Die Funktion bearbeitet dann die entsprechende Variable des Hauptprogramms:

```
>>> def quadriere():
        global x
        x **= 2

>>> x = 5
>>> quadriere()
>>> x
25
```

Ohne die Anweisung `global x` würde man eine Fehlermeldung erhalten, da `x` dann als lokale Variable behandelt würde. (`x` steht ja auf der linken Seite der Zuweisung `x **= 2`.) Der lokalen Variablen `x` ist aber vor dem Quadrieren nie ein Wert zugewiesen worden. Folglich kann auch nichts quadriert werden. Probier's aus!

6.3 Beispiele

Im ersten Beispiel wird untersucht, ob eine einzugebende Zahl eine Primzahl ist. Die verwendete Methode ist recht „billig“. Vielleicht gelingt es dir ja, sie noch zu verbessern?

```
# primzahltest.py
def primzahl(zahl):
    if zahl <= 2:
        prim = True
    else:
        for i in range(2,zahl):
            if zahl%i == 0: # Teiler gefunden
                prim = False
                break
        else:
            prim = True
    return prim

# Hauptprogramm
kandidat = input("Gib eine natuerliche Zahl ein! ")
print kandidat, "ist",
if primzahl(kandidat):
    print "eine",
else:
    print "keine",
print "Primzahl."
```

Bemerkenswert an diesem Beispiel ist, dass auch eine `for`-Schleife einen `else`-Block enthalten kann. Dieser wird nur ausgeführt, wenn die Schleife nicht mit `break` abgebrochen worden ist.

Das zweite Beispiel ist ein Höhepunkt jeder Informatikvorlesung: QuickSort! Wunderhübsch, aber nicht ganz einfach zu verstehen! Wenn dir meine Erklärung nach dem folgenden Skript nicht genügt, dann hilft vielleicht dein Mathematik- oder Informatiklehrer weiter.

```
>>> def quicksort(liste):
        if len(liste) <= 1:
            return liste
        else:
            return quicksort([x for x in liste[1:] if x < liste[0]])\
                   +[liste[0]]\
                   +quicksort([x for x in liste[1:] if x >= liste[0]])

>>> liste = [2,5,7,1,5,10,34,1]
>>> quicksort(liste)
[1, 1, 2, 5, 5, 7, 10, 34]
```

Jede längere Liste wird zuerst in zwei kürzere und damit einfacher zu sortierende Listen zerlegt. Die eine dieser Teillisten enthält alle Elemente, die kleiner sind als das erste Element der Originalliste. Die andere Teilliste enthält (ausser dem ersten Element) alle übrigen Elemente der Originalliste. Beide Teillisten werden (rekursiv) „quicksortiert“ und wieder zusammengefügt.

6.4 Aufgaben

1. Schreibe ein Skript `flaechen.py`, welches je eine Funktion zur Berechnung von Rechtecksflächen, Quadratflächen, Dreiecksflächen und Kreisflächen enthält. Das Skript soll ein Menu zur Auswahl der gewünschten Berechnung besitzen.

2. Gliedere das Skript zur Umrechnung von Temperaturen aus Abschnitt 4.5 mit Hilfe zweier Funktionen (`c_nach_f(c_temp)` und `f_nach_c(f_temp)`) und einer Prozedur (`zeige_optionen()`).

3. Schreibe eine rekursive Funktion `fibo(n)` zur Berechnung der n-ten Fibonaccizahl a_n. (Für die Fibonaccizahlen gelten $a_1 = a_2 = 1$ und $a_n = a_{n-1} + a_{n-2}$, falls $n > 2$.)

4. Kennst du das Spiel „Die Türme von Hanoi“? Das Spiel besteht aus drei Stäben, die auf ein Brett montiert sind. Auf dem ersten Stab befindet sich ein Turm von n Lochscheiben, deren Durchmesser nach oben hin abnimmt. Die Aufgabe besteht darin, die Scheiben vom ersten auf den dritten Stab umzuschichten. Dabei darf jeweils nur die oberste Scheibe eines Turms bewegt werden, und grössere Scheiben dürfen niemals auf kleineren liegen.

   Schreibe eine rekursive Funktion `hanoi(n, von, via, nach)`, welche die einzelnen Spielzüge berechnet. Ein Aufruf der Funktion könnte etwa so aussehen:

   ```
   >>> hanoi(3, 1, 2, 3)
   Scheibe von 1 nach 3
   Scheibe von 1 nach 2
   Scheibe von 3 nach 2
   Scheibe von 1 nach 3
   Scheibe von 2 nach 1
   Scheibe von 2 nach 3
   Scheibe von 1 nach 3
   ```

   Hinweis: Überlege dir, wie du die Aufgabe lösen würdest, wenn du ganze Stapel von jeweils $n - 1$ Scheiben miteinander bewegen dürftest.

Kapitel 7

Module

7.1 Allgemeines

Die interaktive Aufwärmübung dieses Kapitels liefert bei minimalem Aufwand einen maximalen Ertrag:

```
>>> import calendar
>>> calendar.prcal(2004)
```

In Python muss das Rad eben nicht dauernd neu erfunden werden! Viele nützliche Funktionen (oder Klassen; siehe Kapitel 9) hält Python bereits in sogenannten „Modulen“ bereit. Ein Modul wird mit dem `import`-Befehl verfügbar gemacht, und die im Modul enthaltenen Funktionen können dann mit `modulname.funktion(parameter)` aufgerufen werden.

Alternativ kann man auch einzelne Funktionen importieren, welche dann ohne die Angabe des Modulnamens verwendet werden können:

```
>>> from calendar import prcal
>>> prcal(2004)
```

Die folgende Tabelle enthält eine Auswahl wichtiger Module. Eine vollständige Übersicht über alle Module findest du unter http://www.python.org/doc/current/lib/lib.html. Informationen zu einem bestimmten Modul liefert der Befehl `help("modulname")`.

Modul	Erklärung
`calendar`	Kalenderfunktionen
`copy`	Kopieren von Listen und Dictionaries (siehe Seite 27)
`cPickle`	schnelle (in C programmierte) Version von `pickle`
`math`	mathematische Funktionen und Konstanten
`pickle`	Laden und Speichern verschiedenster Datentypen (siehe Seite 49)
`random`	Erzeugen von Zufallszahlen
`time`	zeitbezogene Funktionen
`Tkinter`	Programmieren von graphischen Benutzeroberflächen (GUIs)
`turtle`	Turtlegraphik

7.2 random

Die Funktion `random()` liefert eine zufällige Gleitkommazahl aus dem Intervall $0 \leq x < 1$. Mit `randint(a,b)` können ganzzahlige Zufallszahlen im Bereich $a \leq x \leq b$ erzeugt werden. Mit der Prozedur `shuffle(l)` kann eine Liste `l` „in place“ gemischt werden.

```
>>> from random import random, randint, shuffle
>>> random()
0.56040878668865002
>>> for i in range(10):
        print randint(1,6),

6 2 3 5 6 6 4 4 6 1
>>> farben = ["Herz", "Ecken", "Schaufel", "Kreuz"]
>>> shuffle(farben)
>>> farben
['Schaufel', 'Herz', 'Kreuz', 'Ecken']
```

7.3 time

In diesem Modul sind Funktionen enthalten, die für Zeitberechnungen benötigt werden. Die Funktion `ctime()` gibt beispielsweise die aktuelle Zeit in Form einer Zeichenkette aus:

```
>>> from time import ctime
>>> ctime()
'Wed Mar 31 15:34:51 2004'
```

Daraus kann man sich eine kleine Uhr zusammenbasteln:

```
# uhr.py
from time import ctime

letzteZeit = ""
while True:
    neueZeit = ctime()
    if neueZeit != letzteZeit:
        print "Es ist jetzt", neueZeit[11:19], "Uhr."
        letzteZeit = neueZeit
```

Weitere nützliche Funktionen aus diesem Modul sind `sleep(t)` zur Unterbrechung der Programmausführung für `t` Sekunden und `time()`, womit man sich die seit dem 1. Januar 1970 abgelaufene Zeit in Sekunden angeben kann.

7.4 turtle

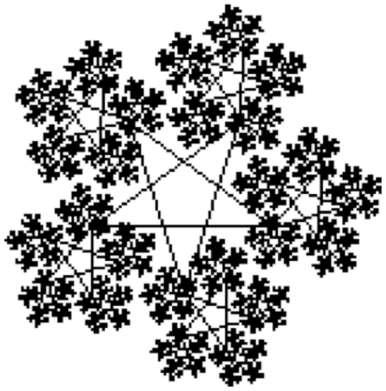
Turtlegraphik („Schildkrötengraphik“) ist ein aus der Programmiersprache LOGO entlehntes Konzept. Die Idee ist bereits 30 Jahre alt, aber man kann damit nach wie vor auf einfache und schnelle Art ansprechende Bilder produzieren.

Eine Turtlegraphik entsteht durch eine Schildkröte, die mit Befehlen wie `forward(distanz)`, `backward(distanz)`, `left(winkel)` oder `right(winkel)` über die Zeichenfläche gesteuert wird. Dabei hinterlässt die Schildkröte eine Spur, es sei denn, man befiehlt ihr mit `up()`, den Stift vom Papier zu heben. (Nach `down()` senkt sie den Stift wieder.)

```
# stern.py
from turtle import *

def stern(groesse):
    for i in range(5):
        forward(groesse)
        left(16)
        if groesse > 3:
            stern(groesse/2.5)
        left(128)

reset()
stern(65)
```



Mit dem Befehl `from turtle import *` werden alle Funktionen, Klassen und Konstanten des Moduls auf einmal importiert. Das ist zwar bequem, aber manchmal auch etwas gefährlich, da die importierten Funktionen unter Umständen gleichnamige Funktionen überschreiben, die du selbst eben geschrieben hast (oder umgekehrt).

Und falls dir die Schildkröte zu langsam kriecht: Mit

```
>>> reset()
>>> tracer(0)
>>> stern(65)
>>> tracer(1)
```

geht’s schneller!

7.5 Eigene Module

Module kann man auch selbst schreiben! Das ist sehr nützlich, wenn man eigene Funktionen (oder Klassen) entwickelt hat, die man in verschiedenen Skripten immer wieder brauchen möchte.

Angenommen, du hast eine ausgeklügelte Funktion `int_input()` geschrieben, die zur Eingabe einer ganzen Zahl zwischen gewissen Grenzen auffordert. Diese Funktion hast du als Datei `werkzeug.py` abgespeichert:

```
# werkzeug.py
def int_input(aufforderung,
              von = -2147483647,   # Bem. 1
              bis =  2147483647):  # Bem. 1
    ok = False
    while not ok:
        try:                       # Bem. 2
            eingabe = raw_input(aufforderung)
            eingabe = int(eingabe)
            ok = (von <= eingabe <= bis)
            if not ok:
                print "Eingabe liegt nicht im verlangten Bereich!"
        except:                    # Bem. 2
            print "Eingabe ist keine ganze Zahl!"
    return eingabe
```

Einige Erklärungen:

Bem. 1: Hier werden die beiden Parameter `von` und `bis` auf eine sehr kleine bzw. sehr grosse ganze Zahl „voreingestellt“. Man nennt solche Parameter auch „Schlüsselwortargumente“. Die Defaultwerte gelten immer dann, wenn beim Aufruf der Funktion nur ein Argument angegeben wird. (Beachte die beiden Funktionsaufrufe weiter unten.)

Bem. 2: Die Funktion `int()` dient hier dazu, die Zeichenkette `eingabe` in eine ganze Zahl zu verwandeln. Diese Umwandlung kann aber nur mit vernünftigen Eingaben funktionieren. Probiere im interaktiven Modus `int("3.14")` und `int("Trallalla!")` aus! Damit in solchen Fällen nicht das ganze Programm mit einer Fehlermeldung abgebrochen wird, kann man die kritischen Zeilen in einen `try`-Block verpacken und im `except`-Block angeben, was im Falle eines Fehlers stattdessen passieren soll.

Wird das Skript `werkzeug.py` ausgeführt, passiert – scheinbar gar nichts! Um herauszufinden, ob die Funktion funktioniert wie sie funktionieren soll, müssen wir sie natürlich zuerst testhalber aufrufen, zum Beispiel im interaktiven Modus. Besser ist es aber, man ergänzt `werkzeug.py` um ein kleines Hauptprogramm, welches möglichst alle denkbaren Situationen durchspielt. Diese „Testumgebung“ kann später wiederverwendet werden, wenn man die Funktion abändert oder nochmals nach Fehlern suchen muss:

```
# Hauptprogramm, Testumgebung
n = int_input("Ganze Zahl eingeben: ")
m = int_input("Ganze Zahl im Bereich 1..100 eingeben: ", 1, 100)
print n, m
```

Nun soll aber `werkzeug.py` in Zukunft als Modul von anderen Skripten importiert werden, und dabei würde es ziemlich stören, wenn bei jedem Importieren diese Testfunktionsaufrufe ausgeführt würden.

Füge darum zu Beginn der Testumgebung noch die Zeile `if __name__ == "__main__":` ein, und rücke die letzten drei Zeilen als Anweisungsblock ein. Die `if`-Anweisung bewirkt, dass das Hauptprogramm nur ausgeführt wird, wenn `werkzeug.py` „direkt“ aufgerufen wurde. Wird `werkzeug.py` jedoch von einem anderen Skript importiert, bleiben diese Zeilen ohne Wirkung, denn die Variable `__name__` enthält dann den Modulnamen `werkzeug`.

Wenn sich das aufrufende Skript im gleichen Verzeichnis befindet wie das neue Modul, dann kann man wie gewohnt mit `import werkzeug` oder mit `from werkzeug import int_input` das ganze Modul oder die einzelne Funktion importieren. Um das neue Modul auch im interaktiven Modus verwenden zu können, muss es in einem der Verzeichnisse abgelegt werden, in welchen Python nach Modulen sucht. Welche das sind, gibt die Variable `path` im Modul `sys` an:

```
>>> import sys
>>> sys.path
```

Eine gute Idee ist auch, ein eigenes Verzeichnis mit eigenen Modulen anzulegen. Dann fügt man den Pfad zu diesem Ordner mit

```
>>> import sys
>>> sys.path.append("Pfadname")
```

zu den Pfaden hinzu, welche Python nach Modulen durchsucht. (Pfadname muss dabei durch den Pfad zum gewählten Verzeichnis ersetzt werden. Da das Format von Dateipfaden unter verschiedenen Betriebssystemen unterschiedlich ist, orientiert man sich am besten am Inhalt der `path`-Variablen.)

Dasselbe ist natürlich auch innerhalb eines Skripts möglich. Das folgende Beispiel erwartet das Modul `werkzeug.py` im Verzeichnis EigeneModule und importiert daraus die Funktion `int_input`:

```
# werkzeugimport.py
from sys import path
path.append("/Applications/MacPython-2.3/Extras/Tools/EigeneModule")
from werkzeug import int_input

n = int_input("Ganze Zahl eingeben: ")
m = int_input("Ganze Zahl im Bereich 1..100 eingeben: ", 1, 100)
print n, m
```

7.6 Aufgaben

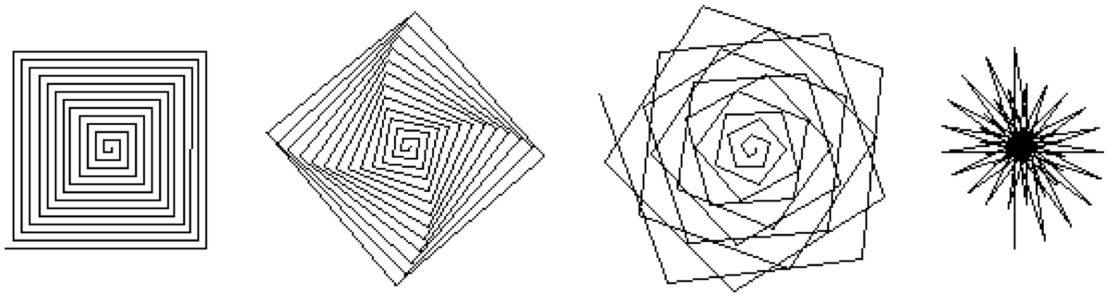
1. Programmiere dieses kleine Ratespiel:

```
Welche Zahl denke ich mir? 46
Zu hoch!
Welche Zahl denke ich mir? 18
Zu tief!
Welche Zahl denke ich mir? 37
Zu hoch!
Welche Zahl denke ich mir? 28
Richtig!
```

   Die vom Computer „gedachte“ Zahl soll zufällig im Bereich 1 bis 100 gewählt werden. Verwende für die Eingaben die Funktion `int_input()`, die du aus dem selbstgeschriebenen Modul `werkzeug.py` importierst.

   Zusatzaufgabe:
   Überlege dir eine Ratestrategie, welche möglichst wenige Versuche erfordert.

2. Schreibe eine Funktion `spirale(winkel)`, die mit Hilfe von Turtlegraphik die abgebildeten Spiralen erzeugen kann.



3. Erzeuge mit dem Befehl `range()` unterschiedlich lange Zahlenlisten, mische sie mit der Prozedur `shuffle()` aus dem Modul `random`, sortiere sie mit der Funktion `quicksort()` von Seite 38 und miss jeweils mit Hilfe der Funktion `time()` aus dem Modul `time` die für das Sortieren benötigte Zeit.

Kapitel 8

Dateien

8.1 Zeichenketten speichern und laden

Die folgenden drei Zeilen schreiben eine tiefe Lebensweisheit in eine Datei Brian.txt:

```
>>> datei = file("Brian.txt", 'w')
>>> datei.write("Always look on the bright side of life!\nMonty Python")
>>> datei.close()
```

Suche die Datei, öffne sie und staune!

Auch Dateien sind in Python Objekte! Mit der Funktion `file()` kann so ein Dateiobjekt erzeugt werden. Als Argumente müssen dabei der Dateiname und ein Modus – zum Beispiel `'w'` für schreiben oder `'r'` für lesen – angegeben werden. Dateiobjekte verfügen (unter anderem) über die Methoden `write()` zum Speichern einer Zeichenkette und `close()` zum Abschliessen des Schreibvorgangs.

Das Lesen aus einer Datei funktioniert ganz analog:

```
>>> datei = file("Brian.txt", 'r')
>>> text = datei.read()
>>> datei.close()
>>> print text
Always look on the bright side of life!
Monty Python
```

Mit dem neuen Wissen verbessern wir unser Telefonbuch aus Abschnitt 5.3. Zwei neue Menueinträge erlauben das Speichern bzw. Laden der Telefonbuchdaten. Beim Speichern werden die Telefonbucheinträge zeilenweise in eine Datei geschrieben. Jede Zeile enthält zuerst einen Namen, dann ein Komma mit anschliessendem Leerschlag als Trennsymbol, dann die zugehörige Telefonnummer und schliesslich ein Zeilenumbruchsymbol. Beim Laden werden die Zeilen der Datei mit der Dateiobjektmethode `readline()` eine nach der anderen gelesen, jeweils vom abschliessenden Zeilenumbruch befreit und mit der Zeichenkettenmethode `split()` beim Trennsymbol in Name und Telefonnummer aufgespaltet:

```
# telefonbuch2.py
def anzeigen():
    for name in telefonbuch.keys():
        print "Name:", name, "\tNummer:", telefonbuch[name]

def hinzufuegen():
    name = raw_input("Name: ")
    nummer = raw_input("Nummer: ")
    telefonbuch[name] = nummer

def loeschen():
    name = raw_input("Name: ")
    if telefonbuch.has_key(name):
        del telefonbuch[name]
    else:
        print "Eintrag nicht gefunden"

def nachschlagen():
    name = raw_input("Name: ")
    if telefonbuch.has_key(name):
        print "Nummer:", telefonbuch[name]
    else:
        print "Eintrag nicht gefunden"

def laden():
    dateiname = raw_input("Laden von Datei: ")
    datei = open(dateiname, 'r')
    while True:
        zeile = datei.readline()
        if zeile == "":
            break
        zeile = zeile[:-1]
        [name, nummer] = zeile.split(', ')
        telefonbuch[name] = nummer
    datei.close()

def speichern():
    dateiname = raw_input("Speichern in Datei: ")
    datei = open(dateiname, 'w')
    for name in telefonbuch.keys():
        datei.write(name+', '+telefonbuch[name]+'\n')
    datei.close()
```

```
def menu():
    print 31*'-'
    print "1. Telefonbuch anzeigen"
    print "2. Eintrag hinzufuegen/aendern"
    print "3. Eintrag loeschen"
    print "4. Nummer nachschlagen"
    print "5. Telefonbuch (hinzu)laden"
    print "6. Telefonbuch speichern"
    print "7. Beenden"
    print 31*'-'

# Hauptprogramm
telefonbuch = {}
wahl = 0
while wahl != 7:
    menu()
    wahl = input("Auswahl: ")
    if wahl == 1:
        anzeigen()
    elif wahl == 2:
        hinzufuegen()
    elif wahl == 3:
        loeschen()
    elif wahl == 4:
        nachschlagen()
    elif wahl == 5:
        laden()
    elif wahl == 6:
        speichern()
print "Auf Wiedersehen!"
```

Gibt man beim Laden der Telefonbuchdatei einen nicht existierenden Dateinamen an (zum Beispiel wegen eines Tippfehlers), dann wird das Programm mit einer Fehlermeldung abrupt beendet. Um das zu verhindern, setzt man beim Öffnen von Dateien in der Regel `try`-Anweisungen ein (siehe Seite 43). Die Funktion `laden()` im obigen Skript wird verbessert:

```
def laden():
    dateiname = raw_input("Laden von Datei: ")
    try:
        datei = open(dateiname, "r")
        while True:
            zeile = datei.readline()
            if zeile == "":
                break
            zeile = zeile[:-1]
```

```
            [name, nummer] = zeile.split(', ')
            telefonbuch[name] = nummer
        datei.close()
    except:
        print "Datei nicht gefunden"
```

8.2 Für andere Datentypen: `pickle`

Auf die im letzten Abschnitt beschriebene Art lassen sich nur Zeichenketten speichern. Um Daten eines anderen Typs zu sichern, müssen diese zuerst in eine Zeichenkette umgewandelt werden. (Im Beispiel `telefonbuch2.py` haben wir deshalb ein Dictionary in eine mehrzeilige Zeichenkette verwandelt.)

Beim Laden von als Zeichenketten gespeicherten Daten muss man sich allerdings merken, von welchem Typ die Daten ursprünglich waren. Das ist nicht immer ganz einfach. War beispielsweise die Zeichenkette `123.45` ursprünglich eine Gleitkommazahl oder immer schon eine Zeichenkette?

Glücklicherweise gibt es in Python das Modul `pickle`, welches erlaubt, fast beliebige Objekte zu speichern:

```
>>> woerterbuch = {"sun":"Sonne", "light":"Licht"}
>>> import pickle
>>> datei = file("Woerter.dmp", 'w')
>>> pickle.dump(woerterbuch, datei)
>>> datei.close()
```

... und wieder zu laden:

```
>>> datei = file("Woerter.dmp", 'r')
>>> woerterbuch = pickle.load(datei)
>>> datei.close()
```

Schliesslich sei hier noch das (in C programmierte) Modul `cPickle` erwähnt, welches genau dasselbe tut wie `pickle`, nur etwa 1000 Mal schneller. Zum Speichern umfangreicher Datenmengen verwendet man mit Vorteil `cPickle`.

8.3 Aufgaben

1. Erweitere den Vokabeltrainer `woerterbuch.py` von Seite 30 und Aufgabe 5 im gleichen Kapitel um Funktionen zum Speichern und Laden der Wortliste. Verwende dabei das Modul `pickle` (oder `cPickle`).

Kapitel 9

Klassen

9.1 Unsere erste Klasse

In diesem Abschnitt werden wir eigene Objekte programmieren. Vielleicht möchtest du vorher nochmals die Grundideen des objektorientierten Programmierens studieren. Lies dazu die Abschnitte 2.1 und 2.2.

Um ein Objekt zu erzeugen, geht man in Python (wie in anderen objektorientieren Programmiersprachen) so vor: Zuerst muss eine Konstruktionsanleitung, eine Art Bauplan des Objekts geschrieben werden. Darin wird festgelegt, welche Merkmale (Attribute) das Objekt besitzen und über welche Methoden es verfügen soll. Einen solchen Bauplan nennt man eine *Klassendefinition* oder etwas salopp auch einfach eine *Klasse*.

Jede Klasse besitzt (bereits von Haus aus) eine *Konstruktormethode*, kurz *Konstruktor*. (In der Regel muss die Konstruktormethode noch den eigenen Bedürfnissen angepasst werden.) Durch den Aufruf dieser Methode können Objekte gemäss den Vorgaben der Klassendefinition erzeugt (man sagt auch *instanziert*) werden. Der Konstruktor heisst in Python immer gleich wie die zugehörige Klasse.

Dazu braucht's sofort ein Beispiel. Erinnere dich an die Schildkröten aus dem Abschnitt über Turtlegraphik (7.4). So eine Schildkröte ist in Tat und Wahrheit ein Objekt der Klasse `Pen` aus dem Modul `turtle`. Mit dem Konstuktor `Pen()` können mehrere Schildkröten ins Leben gerufen werden:

```
>>> from turtle import Pen
>>> john = Pen()
>>> john.forward(50)
>>> eric = Pen()
>>> eric.left(150)
>>> eric.forward(100)
```

In diesem Beispiel habe wir einfach eine bereits bestehende Klasse aus einem Modul importiert. Im nächsten Beispiel schreiben wir die Klassendefinition selbst. Vorhang auf für unsere

allererste eigene Klasse! Die Objekte unserer ersten Klasse sind Würfel mit einer beliebigen Anzahl Seiten. Ein solcher „Polywürfel“ soll die Attribute `seiten` (die Anzahl seiner Seiten) und `wert` (die gerade oben liegende Zahl) sowie eine Methode `wuerfeln()` besitzen:

```
# polywuerfel.py
from random import randrange

class Polywuerfel:
    def __init__(self, seiten):
        self.seiten = seiten
        self.wert = 1

    def wuerfeln(self):
        self.wert = randrange(1, self.seiten+1)
```

Nun sind einige Erklärungen fällig!

An erster Stelle innerhalb der Klassendefinition steht üblicherweise die Definition der Konstruktormethode `__init__()`. Das ist diejenige Methode, die beim Erzeugen eines neuen Objekts ausgeführt wird. Was innerhalb der Klassendefinition `__init__()` heisst, wird also von ausserhalb als `Polywuerfel()` aufgerufen. Die Konstruktormethode sorgt dafür, dass die Attribute des Würfels mit geeigneten Anfangswerten belegt werden.

Sicher ist dir aufgefallen, dass sowohl der Konstruktor als auch die anderen Methoden mit einem seltsamen ersten Argument `self` definiert werden. Wozu dient wohl dieses `self`? Ganz einfach: `self` steht für ein Objekt, dass (später) gemäss der Klassendefinition erzeugt werden soll. Konkreter am Beispiel: Die Konstruktormethode richtet jeden neu erzeugten Würfel so ein, dass die anfänglich oben liegende Zahl (also die Eigenschaft `wert` des neuen Würfels) die 1 sein soll. Es muss also bereits in der Klassendefinition auf den erst noch zu erzeugenden Würfel mit seinen Attributen zugegriffen werden können. Genau das ermöglicht `self`.

Beachte, dass beim Aufruf von Objektmethoden dieses erste Argument fehlt:

```
>>> w = Polywuerfel(12)
>>> w.wert
1
>>> w.wuerfeln()
>>> w.wert
6
```

Der Konstruktoraufruf `Polywuerfel(12)` übergibt sein Argument `12` also an den formalen Parameter `seiten` an der zweiten Position in der Definition der Konstruktormethode. Die Methode `wuerfeln()` wird ganz ohne Argumente aufgerufen.

PS: Hast du übrigens gemerkt, dass Klassennamen in Python gross geschrieben werden?

9.2 Sichtbarkeit von Attributen

Leider kann man mit einem Polywürfel auch dumme Dinge anstellen. Nichts hindert uns daran, einen 12-seitigen Würfel mit einer oben liegenden Augenzahl 13.5 herzustellen:

```
>>> w = Polywuerfel(12)
>>> w.wert = 13.5
>>> w.seiten
12
>>> w.wert
13.5
>>> w.seiten = 15
>>> w.seiten
15
```

So etwas sollte nicht sein, zumal wir uns einen 12-seitigen Würfel wohl mit den Augenzahlen 1 bis 12 bedruckt vorstellen! Und ein 12-seitiger Würfel mit 15 Seiten ist erst recht suspekt.

Besser wäre es, wenn eine direkte Beeinflussung „von aussen“ gar nicht möglich wäre. Die Änderung der oben liegenden Zahl sollte nur möglich sein, indem man dem Würfel die Nachricht „bitte ändere die oben liegende Zahl“ sendet und dann das Objekt selbst die Änderung ausführen lässt. Die Aufforderung, eine sinnlose oben liegende Augenzahl einzustellen, könnte das Objekt dann verweigern. Und die Änderung der Seitenzahl soll der Würfel – nachdem er einmal erzeugt wurde – überhaupt nicht erlauben.

Python hat wie immer eine Lösung parat: Durch das Voranstellen von zwei Unterstrichen vor seinen Namen kann man ein Attribut vor dem Zugriff „von aussen“ schützen. Man nennt so ein Attribut *privat* (im Gegensatz zu den bisherigen *öffentlichen* Attributen).

```
# polywuerfel2.py
from random import randrange

class Polywuerfel:
    def __init__(self, n):
        self.__seiten = n
        self.__wert = 1

    def wuerfeln(self):
        self.__wert = randrange(1, self.__seiten+1)
        return self.__wert
```

Machen wir den Test! Wir versuchen, die Anzahl der Seiten des Würfels abzurufen:

```
>>> w = Polywuerfel(12)
>>> w.__seiten
```

```
Traceback (most recent call last):
  File "<pyshell#93>", line 1, in -toplevel-
    w.__seiten
AttributeError: Polywuerfel instance has no attribute '__seiten'
```

Python tut nun einfach so, als hätte der Würfel gar kein Attribut `__seiten`. Was ist aber, wenn wir – vergesslich wie wir halt sind – doch einmal wissen wollen, wieviele Seiten unser Würfel hat? Wir bitten ihn per Nachricht um Auskunft! Die Klasse `Polywuerfel` muss also eine neue Methode zur Abfrage der Seitenzahl erhalten. Und wenn wir gerade dabei sind, spendieren wir dem Polywürfel auch gleich eine Methode zur Abfrage der oben liegenden Zahl und eine weitere Methode zum Ändern der oben liegenden Augenzahl. In dieser letzten Methode achten wir darauf, dass die Augenzahl erstens ganz ist und zweitens zwischen 1 und der Seitenzahl liegt:

```
# polywuerfel3.py
from random import randrange

class Polywuerfel:
    def __init__(self, n):
        self.__seiten = n
        self.__wert = 1

    def wuerfeln(self):
        self.__wert = randrange(1, self.__seiten+1)
        return self.__wert

    def getSeiten(self):
        return self.__seiten

    def getWert(self):
        return self.__wert

    def setWert(self, wert):
        if not wert%1 and 1 <= wert <= self.__seiten:
            self.__wert = wert
        else:
            print "Ungueltige Augenzahl"
```

Methoden zum Lesen von Attributen beginnen in Python oft mit `get`, Methoden zum Ändern von Attributwerten mit `set`. Besonders das Schreiben der `set`-Methode verlangt Sorgfalt, da darauf zu achten ist, dass das Attribut keine sinnlosen Werte annimmt, die zu Fehlern im Programmablauf führen könnten. Lässt man diese Kontrolle weg, dann könnte man eigentlich auch gleich auf private Attribute verzichten...

Zum Schluss dieses Abschnitts probieren wir die neuen Methoden natürlich noch aus:

```
>>> w = Polywuerfel(12)
>>> w.getSeiten()
12
>>> w.setWert(7)
>>> w.getWert()
7
>>> w.setWert(4.5)
Ungueltige Augenzahl
>>> w.setWert(15)
Ungueltige Augenzahl
>>> w.wuerfeln()
9
```

9.3 Überladen von Methoden (Polymorphie)

In Python gibt es Operationen, die auf verschiedene Objekttypen angewendet werden können und dabei unterschiedlich wirken. Man spricht dann von „überladenen“ (oder „polymorphen“) Operationen. Ein Beispiel, das du schon längst kennst, ist der `+`-Operator. Auf Ganzzahlen angewendet, liefert er als Resultat wiederum eine ganze Zahl:

```
>>> 5+3
8
```

Zeichenketten hingegen werden durch ein `+` natürlich nicht addiert sondern einfach aneinandergefügt:

```
>>> "Guten"+" "+"Morgen!"
'Guten Morgen!'
```

Ein anderes Beispiel ist der Befehl `print`, der eine – möglichst lesbare – Beschreibung des Objekts ausgibt, auf welches er angewendet wurde. Bei Zahlobjekten ist das einfach der Wert der Zahl, bei Zeichenketten die Zeichenkette ohne die Anführungszeichen. Beim Würfelobjekt `w` aus dem letzten Abschnitt sieht die Beschreibung folgendermassen aus:

```
>>> print w
<__main__.Polywuerfel instance at 0x4c0b48>
```

Aus dieser Beschreibung können wir entnehmen, dass `w` eine Instanz (also ein Objekt) der Klasse `Polywuerfel` ist, und wo dieses Objekt im Speicher abgelegt wurde. Möglicherweise interessiert uns die Angabe über den Speicherort aber nicht so sehr; stattdessen würden wir lieber wissen, wie viele Seiten unser Würfel besitzt. Dazu müssen wir der `print`-Anweisung eine speziell auf `Polywuerfel`-Objekte zugeschnittene, neue Bedeutung geben. Wir überladen den Befehl also ein weiteres Mal. Und das geht so:

Wir definieren innerhalb der Klassendefinition eine Methode mit dem vorgegebenen Namen `__str__(self)`. Befehlszeilen der Art `print wuerfelobjekt` werden vom Pythoninterpreter intern in einen Aufruf dieser Methode, also `wuerfelobjekt.__str__()`, umgewandelt.

```
# polywuerfel4.py
from random import randrange

class Polywuerfel:
    def __init__(self, n):
        self.__seiten = n
        self.__wert = 1

    def wuerfeln(self):
        self.__wert = randrange(1, self.__seiten+1)
        return self.__wert

    def getSeiten(self):
        return self.__seiten

    def getWert(self):
        return self.__wert

    def setWert(self, wert):
        if not wert%1 and 1 <= wert <= self.__seiten:
            self.__wert = wert
        else:
            print "Ungueltige Augenzahl"

    def __str__(self):
        return "Polywuerfel mit "+str(self.__seiten)+" Seiten"

# Hauptprogramm zum Testen
if __name__ == "__main__":
    w = Polywuerfel(20)
    print w
```

(An diesem Beispiel siehst du auch gleich nochmals, wie man dafür sorgt, dass die Anweisungen der Testumgebung wirklich nur beim Testen ausgeführt werden und nicht, wenn die Klasse importiert wird.)

Alle Methoden, welche überladbare Operationen oder Funktionen repräsentieren, besitzten reservierte Namen, die mit doppelten Unterstrichen beginnen und enden. Die folgende Tabelle enthält eine klitzekleine Auswahl.

zu überladende Operation	entsprechende Methode
`+`	`__add__(self, zweitobjekt)`
`+=`	`__iadd__(self, zweitobjekt)`
`<`, `<=`, `==`, `>`, `>=`, `!=`	`__cmp__(self, vergleichsobjekt)`
`print`	`__str__(self)`

Um zwei Würfel (hinsichtlich der oben liegenden Zahlen) zu vergleichen, müssten also die Vergleichsoperatoren `<`, `<=`, `==`, `>`, `>=` und `!=` überladen werden. Dazu muss die Klassendefinition nur um die eine einzige Methode `__cmp__(self, vergleichsobjekt)` ergänzt werden. Der Rückgabewert dieser Methode muss negativ, 0 oder positiv sein, je nach dem, ob das Objekt kleiner, gleich gross oder grösser als das Vergleichsobjekt ist. Ergänze die Klasse `Polywuerfel` um die folgende Definition:

```
def __cmp__(self, vergleichswuerfel):
        a = self.__wert
        b = vergleichswuerfel.__wert
        if a < b: return -1
        elif a == b: return 0
        else: return 1
```

Und hier folgt der Testlauf:

```
>>> w1 = Polywuerfel(6)
>>> w2 = Polywuerfel(15)
>>> w1.wuerfeln()
6
>>> w2.wuerfeln()
3
>>> w1 < w2
False
>>> w1 != w2
True
```

9.4 Unterklassen, Vererbung

Nimm an, du müsstest ein Pythonskript schreiben, das Aufgaben aus der räumlichen Geometrie bearbeiten soll. Du hast dazu eine Klasse `Koerper` zur Modellierung von beliebigen dreidimensionalen Objekten verfasst. Als einziges Attribut besitzen solche Körper ihr jeweiliges Volumen, und ihre einzige Methode dient dazu, zwei Körper zu einem neuen, dritten Körper zusammenzulegen:

```
# koerper.py

class Koerper:
    def __init__(self, volumen):
        if volumen > 0:
            self.__volumen = volumen
        else:
            print "Ungueltiges Volumen!"
```

```
    def __add__(self, andererKoerper):
        summe = Koerper(self.__volumen + andererKoerper.__volumen)
        return summe

    def getVolumen(self):
        return self.__volumen
```

Wir probieren die Klasse für einmal wieder im interaktiven Modus aus:

```
>>> k1 = Koerper(10)
>>> k2 = Koerper(20)
>>> k1.getVolumen()
10
>>> k2.getVolumen()
20
>>> k3 = k1 + k2
>>> k3.getVolumen()
30
```

(Statt der letzten Anweisung hättest du übrigens auch einfach `(k1+k2).getVolumen()` schreiben können. Hübsch, nicht?)

Weiter im Text: In deinen raumgeometrischen Aufgaben kommen immer wieder Kugeln vor, und du möchtest daher auch den Objekttyp `Kugel` in dein Pythonprogramm einführen. Nun sind Kugeln ja spezielle Körper, und alles, was dein Pythonskript mit (beliebigen) Körpern anrichten kann, sollte es auch mit den (spezielleren) Kugeln tun können. Objektorientierte Programmiersprachen machen so etwas möglich, ohne dass man eine komplett neue Klasse `Kugel` schreiben und darin alle Methoden der Klasse `Koerper` von neuem definieren müsste!

Durch die Klassendefinition `class Subklasse(Basisklasse)` wird die neue Klasse als speziellere, genauer beschriebene Unterklasse eingerichtet. Bei uns sind Kugeln die spezielleren Objekte, die zum Beispiel durch Angabe ihrer Radien genauer beschrieben werden können als die sehr allgemeinen Objekte der Klasse `Koerper`. Unterklassen erben die Attribute und Methoden der Oberklasse und stellen im allgemeinen zusätzliche Attribute zur Verfügung oder verfeinern durch zusätzliche Methoden (oder durch Überschreiben einer Methode) das Verhalten ihrer Oberklassen.

Ergänze also das `Koerper`-Skript um die folgenden Zeilen:

```
class Kugel(Koerper):
    def __init__(self, radius):                # init-Methode der Unterklasse
        from math import pi
        Koerper.__init__(self, 4/3*radius**3*pi)     # ... und der Oberklasse
        self.__radius = radius

    def getRadius(self):
        return self.__radius
```

Ganz besonders bemerkenswert sind zwei Sachen:

Erstens stellst du fest, dass die Konstruktormethode der Subklasse `Kugel` den Konstruktor der Basisklasse aufruft. Die Basisklasse ist also die eigentliche Geburtshelferin für Instanzen der Subklasse!

Zweitens kanns du aus dem folgenden interaktiven Testlauf ersehen, dass die Kugeln tatsächlich über die Methoden `getVolumen()` und `__add__()` der Oberklasse verfügen, umgekehrt die Körper aber die Methode `getRadius()` der Unterklasse nicht kennen:

```
>>> ko = Koerper(100)
>>> ku = Kugel(10)
>>> ku.getRadius()
10
>>> ku.getVolumen()
3141.5926535897929
>>> k = ko + ku
>>> k.getVolumen()
3241.5926535897929
>>> k.getRadius()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    k.getRadius()
AttributeError: Koerper instance has no attribute 'getRadius'
```

Die Klasse `Koerper` hat ihre Basisklasse nur um das Attribut `__radius` erweitert. Die nächste Klasse – wiederum eine Spezialisierung der Klasse `Koerper` – enthält auch eine neue Methode. Mit ihr können die Objekte der neuen Subklasse aufeinander gestapelt werden:

```
class Quader(Koerper):
    def __init__(self, laenge, breite, hoehe):
        volumen = laenge * breite * hoehe
        Koerper.__init__(self, volumen)         # init-Methode der Oberklasse
        self.__laenge = laenge
        self.__breite = breite
        self.__hoehe = hoehe

    def stapeln(self, anzahl):
        neueHoehe = anzahl * self.__hoehe
        stapel = Quader(self.__laenge, self.__breite, neueHoehe)
        return stapel

    def getLaenge(self):
        return self.__laenge
```

```
    def getBreite(self):
        return self.__breite

    def getHoehe(self):
        return self.__hoehe
```

Der Testlauf berechnet das Gesamtvolumen eines nicht näher beschriebenen Körpers, einer Kugel und einem Turm aus 25 aufeinander gestapelten Quadern:

```
>>> ko = Koerper(100)
>>> ku = Kugel(10)
>>> qu = Quader(3, 4, 5)
>>> qu.getHoehe()
5
>>> turm = qu.stapeln(25)
>>> turm.getHoehe()
125
>>> k = ko + ku + turm
>>> k.getVolumen()
1914.1592653589794
```

Vielleicht versucht du ja auch noch, ein paar Kugeln zu stapeln. Was meint wohl Python dazu?

Zum Schluss musst du dir der Vollständigkeit halber noch eine Warnung anhören, welche die Sichtbarkeit von Attributen betrifft: Objekte von Unterklassen können private Attribute der Oberklasse nicht sehen! Konkret bedeutet das, dass die Verwendung des privaten Attributs `__volumen` in der Definition der abgeleiteten Klasse `Quader` zu einer Fehlermeldung führen würde. (Probier's aus!) Wenn eine Klasse also zur Mutterklasse werden soll, dann empfiehlt es sich, darin keine privaten Attribute zu verwenden.

9.5 Beispiele

Im einfacheren ersten Beispiel wird eine Lampe modelliert, die ein- und ausgeschaltet und deren Glühbirne ausgewechselt werden kann. Der Einfachheit halber werden öffentliche Attribute verwendet. Wenn du Lust auf die kleine Übung hast, dann kannst du die Attribute „privatisieren".

```
# lampe.py

class Lampe:
    def __init__(self, leistung):
        self.leistung = leistung
        self.eingeschaltet = False
```

```
    def einschalten(self):
        self.eingeschaltet = True
        print "Lampe brennt mit einer "+str(self.leistung)+"-Watt-Birne."

    def ausschalten(self):
        self.eingeschaltet = False
        print "Lampe ist ausgeschaltet."

    def wechsleBirne(self, leistung):
        if self.eingeschaltet:
            print "Schalte die Lampe zuerst aus."
        else:
            self.leistung = leistung
            print "Birne ersetzt! Neue Leistung betraegt "+\
                  str(leistung)+" Watt."
```

Im ausführlicheren zweiten Beispiel geht es ums liebe Geld. Auch hier werden – auch aus dem am Ende des vorhergehenden Abschnitts erwähnten Grund – nur private Attribute verwendet.

(folgt)

9.6 Aufgaben

1. Schreibe eine Klasse `Fahrzeug` zur Beschreibung von idealisierten, dimensionslosen Fahrzeugen, die sich in der xy-Koordinatenebene bewegen.

   Deine Fahrzeuge sollen jeweils über die Attribute `x` und `y` zur Beschreibung ihrer aktuellen Position, über zwei weitere Attribute `a` und `b` zur Beschreibung ihrer Ausrichtung und schliesslich über das Attribut `v` zur Angabe ihrer Geschwindigkeit verfügen.

   An Methoden müssen die Fahrzeuge besitzen: `lenkelinks()` und `lenkerechts()`, um die Fahrzeugausrichtung um jeweils einen festen Drehwinkel zu ändern, `beschleunige()` und `bremse()`, um ihre Geschwindigkeit um eine Einheit anzupassen, und schliesslich `fahre()`, um die neue Fahrzeugposition nach einer Zeiteinheit zu berechnen.

   Hinweis: Bei der Berechnung der neuen Fahrzeugausrichtung hilft dir vielleicht eine Formelsammlung oder sogar dein Mathematiklehrer weiter...

Kapitel 10

GUIs mit Tkinter