1. Programmieren I (Einstieg)

Markus Demleitner *(msdemlei@cl.uni-heidelberg.de)*

Worum geht es?

Programmieren heißt, Probleme zu analysieren, zur formalisieren und effizient zu implementieren. Wir konzentrieren uns hier auf vor allem auf den dritten Punkt, die beiden anderen spielen aber natürlich eine große Rolle.

Technisches

Schein := (Übungen lösen)+(Kleine mündliche Prüfung)

Übungen lösen := 50% der erreichbaren Punktzahl hinkriegen.

abgeben := Lösungen rechtzeitig (Termine stehen auf den Übungsblättern) an eureN TutorIn schicken.

Tutorium

In einem Tutorium werden die Übungsblätter besprochen und Ergänzungen zur Vorlesung gegeben. Die Teilnahme am Tutorium ist in der Regel Voraussetzung für die Scheinvergabe.

Literatur

Literaturhinweise zu dieser Vorlesung gibt es online[1].

Dies ist eine Wiki-Seite, d.h. ihr könnt, dürft und sollt Ergänzungen oder Kommentare beitragen.

Ein paar Randbemerkungen

Die meisten Studierenden, die dieses Skript lesen, dürften im ersten Semester sein – ich darf also kurz folgende Punkte erwähnen:

1. Ihr seid an der Uni – und das bedeutet, dass ihr nicht wie in der Schule permanent gegängelt und dadurch zum Arbeiten gezwungen seid. Nun, jedenfalls ist das vorerst noch so. Die Unterstellung ist, dass ihr, in diesem Fall, Computerlinguistik studiert, weil es euch interessiert. Wenn es euch aber interessiert, so geht die Unterstellung weiter, solltet ihr aus eigenem Antrieb versuchen, euch das Wissen anzueignen, das ihr braucht, wenn ihr diesem Interesse nachgehen wollt. Die Vorlesung fungiert dabei mehr als Einladung, vielleicht auch als Rahmen, kann aber die eigenständige Auseinandersetzung mit dem Stoff nur ergänzen. Ob die genannte Unterstellung richtig ist oder nicht, ihr müsst damit leben, dass sie gemacht wird. Ich gebe zu, dass der Umstand, dass gerade hier im Programmierkurs anfangs nicht immer ganz klar ist, was eigentlich der Stoff mit Comptuerlinguistik zu tun hat, erschwerend hinzukommt. Seht trotzdem, dass euer Interesse wachbleibt oder, wo nötig, erwacht.
2. Daraus folgt insbesondere, dass ich (wie wohl alle anderen Dozenten auch) davon ausgehe, dass ihr euch mit dem Stoff auseinandersetzt, ohne dass etwa Hausaufgaben euch genau spezifizieren, was ihr wie zu machen habt. Um den Übergang etwas zu erleichtern, gibt es in diesem Skript viele Aufgaben, oft mit Lösungen – probiert euch an ihnen. Noch nützlicher zur Durchdringung des Stoffes ist es natürlich, wenn ihr euch selbst Aufgaben ausdenkt. Zur Ergänzung des Skripts bin ich übrigens an solchen selbst ausgedachten Aufgaben immer interessiert.
3. Es ist sehr einfach, Vor- und Nachbereitung der Vorlesungen sausen zu lassen oder gleich nicht in die Vorlesung zu gehen. Das beste Gegenmittel gegen Prokrastination sind Sozialkontakte – die Zeit, die ihr mit Mitstudis in Cafeterien oder in der Mensa zubringt, ist in

[1] `http://wiki.cl.uni-heidelberg.de/moin/ProgILiteratur`

aller Regel gut investiert. Guckt, dass ihr euch mit ein paar Mitstudis anfreundet, plaudert mit ihnen über der Stoff. Solche „Lerngruppen“ werden euch das Studium und auch das Lernen deutlich versüßen und erleichtern – und ohne werdet ihr oft mit Verzweiflung zu ringen haben. Leider sind viele etwas schüchtern, wenn es daran geht, auf andere Leute zuzugehen. Probiert es trotzdem, nutzt die Tutorien. Lernen geht in Kooperation einfach um Längen besser als in Konkurrenz.

4. Formale Voraussetzung zur Zulassung zur Scheinklausur ist die Lösung von mindestens der Hälfte der Hausaufgaben. Das ist etwas unglücklich, denn eigentlich sind die Hausaufgaben einfach so etwas wie der praktische Teil der Vorlesung, ohne den es völlig unmöglich ist, später auch nur ein paar Zeilen Code hinzukriegen. Mir wäre es lieber, wenn er von Leistungsdruck und Punkteklauberei befreit wäre. Andererseits ist die traurige Erfahrung, dass freiwillige Hausaufgaben ziermlich schnell hinten runterfallen. Bemüht euch trotz der dummen 50%-Regel, die Hausaufgaben als „Inhalt“ und nicht als „Prüfung“ zu sehen.

5. Vorlesungen sind eine der übelsten Formen des früher mal geschmähten „Frontalunterrichts“: Der/die MeisterIn steht vorne und gießt seine/ihre Weisheit über die SchülerInnen. Dies ist keine sehr gute Art der Wissensvermittlung, die Besseren aber funktionieren in der real existierenden Universität meistens noch schlechter. Dennoch: Je größer eure Beteiligung an der Veranstaltung, desto mehr werden alle Beteiligten lernen. Fragt also, macht Anmerkungen, beschwert euch, wenn ihr was nicht versteht. Mir ist klar, dass sich das weit leichter schreibt als es getan ist, aber vielleicht hilft euch die Information, dass ihr, wenn ihr etwas nicht versteht, dieses Schicksal aller Wahrscheinlichkeit nach mit dem größeren Teil des Hörsaals teilt. Ihr habt nichts zu verlieren, wenn ihr (auch dumme, so es so etwas gibt) Fragen stellt oder auch mal eine falsche Anmerkung macht: Ich bin euch in jedem Fall dankbar, und die meisten eurer Mitstudis werden dieses Gefühl teilen. Nach den ersten paar Zwischenreden wirds viel einfacher. Wirklich.

6. Beschwert euch insbesondere, wenn irgendwas nicht so funktioniert wie im Skript versprochen. Ihr steht dabei nie als dumm da, weil sowas in jedem Fall mein Fehler ist, in der einen oder anderen Weise. Ebenfalls dankbar bin ich euch für sonstige Korrekturen oder Verbesserungsvorschläge.

7. Wissenschaft ist ein weites Feld, und für das Programmieren gilt das Gleiche. Selbst wenn ihr alles, was in diesem Skript steht, wisst, werdet ihr zur Lösung konkreter Probleme immer noch mehr wissen müssen. Gewöhnt euch frühzeitig daran, Quellen wie Originaldokumentationen, Artikel, Posts im Usenet (die Newsgroup comp.lang.python ist wirklich lohnend und von netten Leuten frequentiert) zu verwenden. Der größte Teil des Materials ist in Englisch, und dieser Teil wird um so größer, je fortgeschrittener eure Probleme sind. Seht, dass ich damit zurecht kommt und weicht Englisch nicht aus.

8. Vor allem aber: Bei allen Qualen mit Prüfungen, Punkten und Credits – habt Spaß an eurem Studium. Ohne Freude am Fach ist ein Studium eine Qual, mit Freude am Fach (und FreundInnen im Fach) ist es praktisch ein Selbstläufer.

2. Was ist ein Programm?

Programm
Anweisungen
Variablen
Kontrollstrukturen
Sequenz
Selektion
Iteration
Algorithmen
Ausführen
Skripten
Interpreters
shell

Ein *Programm* ist eine Abfolge von *Anweisungen*, die festlegen, wie ein Problem zu lösen ist. Man kann Anweisungen nach ihrer Funktion gruppieren:

- Eingabe (das Programm besetzt *Variablen* mit Daten von der Platte, dem Netz, der Benutzerin)
- Verarbeitung (Rechnen...)
- Ausgabe

Die Abfolge der einzelnen Anweisungen wird durch *Kontrollstrukturen* gesteuert. Üblicherweise unterscheidet man hier drei Typen:

- *Sequenz* (lineare Abfolge von Anweisungen)
- *Selektion* (Anweisungen nur in Abhängigkeit von bestimmten Bedingungen ausführen)
- *Iteration* (Anweisungen mehrfach hintereinander ausführen)

Aus diesen Elementen kann man *Algorithmen* bauen und daraus Programme:

```
name = raw_input("Name? ")
if name:
  for c in name:
      print c
else:
  print "Spielverderber"
```

Ausführen von Programmen (*Skripten*) durch Aufruf des Python-*Interpreters*:
`python erstes.py`

wenn wir das Programm in die erstes.py geschrieben haben und in einer *shell* sind.

Shells

Wie ihr auf eurem System eine Shell bekommt, hängt davon ab, was euer System ist – unter Unix ist es meistens einfach, ein „Terminalfenster“ zu bekommen, unter Windows gibts unter „Zubehör“ oder ähnlich eine „Kommandozeilenaufforderung“ oder ähnlich. Speziell unter Windows ist es gut möglich, dass die Shell den Python-Interpreter nicht findet. Es sollte dann möglich sein, das Programm mit „start erstes.py“ laufen zu lassen, doch ist das nur eine Notlösung – findet heraus, wie ihr den Python-Interpreter in euren Pfad bekommt. Die TutorInnen helfen euch ggf. dabei.

Je nach dem, was für ein Entwicklungssystem benutzt wird, gehen auch andere Methoden (auch wenn ihr in jedem Fall sehen solltet, dass ihr mit der Shell zurechtkommt, schon allein, damit ihr nicht völlig von eurer speziellen Umgebung abhängig seid). In IDEs (Integrated Development Environments) gibt es beispielsweise einen Menüpunkt „Run“ oder ähnlich.

Unter Unix kann man `#!/x/y/z/python` (dabei müsst ihr /x/y/z durch den Pfad zu eurem Python-Interpreter ersetzen, meist ist das /usr/bin – wenn es das nicht ist, solltet ihr selbst wissen, was es ist) oder `#!/usr/bin/env python` (hier sucht das System selbst nach dem Interpreter) in die erste Zeile schreiben, dann mit Kommando `chmod +x erstes.py` das System überreden, das Skript als ausführbare Datei zu akzeptieren und muss nur noch `erstes.py` schreiben, um das Programm zu starten.

Schließlich kann man in grafischen Benutzeroberflächen auch einfach auf das Datei-Icon klicken, hat dann aber gerne das Problem, dass die Ausgabe des Programms verschwindet, bevor sie zu lesen ist. Im allergrößten Notfall könnt ihr ein `raw_input()` an das Ende eures Skripts schreiben, dann wartet das System auf eine Eingabe, bevor es das Fenster schließt.

Interpreter und Code

CPU
Prozessor
Interpreter
Quellcode
interaktiv

Dass ihr ohne weitere Vorkehrungen ein Python-Programm nicht „einfach aufrufen“ könnt, liegt daran, dass die Herzen von Computern (die *CPU*s, Central Processing Units oder einfach *Prozessor*en sehr schlicht gebaut sind. Sie wissen (fast) nichts von Schleifen und Variablen, nichts von Tastaturen und Monitoren – was, genau betrachtet, auch gar nicht so schlecht ist, denn die meisten Computer haben gar keine Tastaturen (etwa die in Autos, Kaffeemaschinen und Routern).

Was so eine CPU wirklich versteht, muss für Menschen immer irgendwie „repräsentiert“ werden. Eine Darstellung, die recht nah an der Maschine ist, könnt ihr auf vielen Unix-Maschinen mit dem Programm xxd bekommen – das Programm zu Anzeigen eines Verzeichnisinhalts sieht damit etwa so aus:

```
> xxd /bin/ls | more
[...]
0002ba0: 0508 a188 a405 0889 1534 a005 08a3 30a0  .........4....0.
0002bb0: 0508 e919 fbff ff8b 1508 a405 08b8 0100  ................
0002bc0: 0000 85d2 7538 31db 83f8 0174 2a83 f802  ....u81....t*...
0002bd0: 7415 891d 98a4 0508 85db 0f84 f0fa ffff  t...............
0002be0: 31c0 e9e2 fcff ffc7 0424 0100 0000 e851  1........$.....Q
0002bf0: edff ff85 c074 dbbb 0100 0000 ebd4 8954  .....t.........T
0002c00: 2404 a1cc a005 08b9 806b 0508 894c 2408  $........k...L$.
[...]
```

(das ist nur ein kleiner Ausschnitt). Es ist klar, dass so (in „Hexcode“) niemand ernsthaft programmieren will. Deshalb haben InformatikerInnen seit langer Zeit versucht, Rechner dazu zu bringen, etwas menschen- oder jedenfalls anwendungsgerechtere Sprachen zu verstehen. Die Ergebnisse dieser Bemühungen sind vielfältig. Die Sprache, mit der wir uns hier beschäftigen, Python eben, ist dabei relativ weit vom Rechner abstrahiert (im nächsten Semester werden wir uns mit C beschäftigen, das näher an der Maschine bleibt – was nicht nur von Nachteil ist).

Weil aber Python eben nicht die Sprache ist, die der Rechner versteht, muss dem Rechner erklärt werden, was wir meinen, wenn wir „print“ schreiben oder „for c in name“. Diese „Erklärung“ bekommt der Rechner in Form eines Programms, das Python in das Kauderwelsch des Prozessors übersetzt, eines *Interpreter*s (was auf Englisch auch nur Übersetzer oder Dolmetscher bedeutet). Also: Das Programm `python`, das wir oben aufgerufen haben, ist nichts als ein Dolmetscher zwischen Python (das wir jedenfalls im Prinzip ganz gut verstehen können) und dem, was der Rechner versteht.

Die Dateien mit dem *Quellcode* werden in der Regel in ganz normalen Editoren – etwa emacs, vi oder auch notepad – geschrieben, besagte IDEs bringen eigene Editoren mit. Word und Co könnten zwar im Prinzip auch verwendet werden, aber einerseits will mensch zum wirklichen Programmieren doch etwas mehr von einem Editor als die üblichen Office-Programme so bieten, und andererseits muss mensch aufpassen, dass diese Programme, die ihre Dokumente üblicherweise in wilden Binärformaten (was meist so aussieht wie das Kauderwelsch oben) speichern, tatsächlich ganz normalen, „nackten“ Text schreiben. Also: tuts nicht.

Details in den Tutorien.

Ergebnis:

```
examples> python erstes.py
Name? Bla
B
l
a
examples> python erstes.py
Name?
Spielverderber
```

Da Programm `python` ein Interpreter ist, kann es Anweisungen auch *interaktiv* ausführen. Ein ganz kleines Programm:

```
examples> python
Python 2.2 (#1, Feb 11 2002, 12:19:15)
[GCC 2.95.3 20010315 (release)] on linux2
Type "help", "copyright"...
>>> print "Hello world"
Hello world
```

Gegensatz: *Compiler*. Während ein Interpreter etwa so funktioniert wie ein Simultandolmetscher und das, was ihr sagt, unmittelbar dem Rechner verständlich macht, funktioniert ein Compiler eher wie ein Übersetzer: Er nimmt eure Eingabe und übersetzt sie in einem Rutsch. Das Ergebnis ist eine Datei, die Kauderwelsch enthält, das der Rechner versteht – etwa wie ein menschlicher Übersetzer, der als Ergebnis ein Buch in der Zielsprache in Händen hält.

Compiler machen die Programmentwicklung mühsamer, führen aber in der Regel zu schneller laufenden Programmen. In Programmieren II werden wir uns damit auseinandersetzen.

(Weiterführend:) Die üblichen Python-Implementationen sind, das nur nebenbei, nicht eigentlich klare Interpreter, sondern *Bytecode-Compiler*, die sozusagen Maschinencode für eine gar nicht real existierende Maschine erzeugen. Diese Maschine wird dann auf dem realen Computer emuliert. Sinn der Sache ist, zwar einige Vorteile des Kompilierens zu behalten, aber die Beschränkungen realer Hardware nicht so deutlich zu spüren.

Problems

(2.1)* Macht euch (ggf. mit Hilfe eurer TutorInnen) mit eurem Python-System vertraut. Es schadet nichts, das zunächst auf den Rechnern bei uns im Pool zu machen, denn dort läuft alles so wie hier beschrieben. Gebt das Beispiel-Programm mit einem Texteditor ein, lasst das Programm laufen, seht nach, was es bei verschiedenen Eingaben so tut. Schreibt etwas wie `"mein String"`, vor das c mit ersten print-Statement und seht, was dann passiert.

(2.2)* Ihr könnt im Python-Interpreter rechnen. Ruft also einfach nur das Python-Programm auf und probiert aus, was der Interpreter sagt, wenn ihr `4+5` oder `2.3/3` tippt. Probiert aus, was die Maschine sagt, wenn ihr demgegenüber `print 2.3/3` sagt. Der Unterschied ist übrigens gerade der zwischen der „Repräsentation“ und der „Ausgabe“ des Ergebnisses – wir werden später darauf zurückkommen.

3. Werte, Typen, Variablen

Ein *Wert* ist etwas, das ein Rechner manipulieren kann, unter anderem:

- ganze Zahlen (*integer*, z.B. 200 oder -30)
- Fließkommazahlen (*float*, z.B. 2.3 oder 6.8e8, was für 6.8×10^8 oder 680 000 000 steht – Vorsicht: Das Dezimalkomma ist in Python US-Amerikanisch, also ein Punkt)
- Zeichenketten (*string*, in Python in einfache oder doppelte Anführungszeichen gesetzt, also etwa `"string"` oder `'50'`)

Werte haben *Typen*, die wesentlich bestimmen, was mit ihnen gemacht werden kann (wie *Operatoren* auf sie wirken):

```
>>> 2+4
6
>>> "2"+'4'
'24'
```

Die *Funktion* `type` sagt einem, welchen Typ ein Wert hat:

```
>>> type(2)
<type 'int'>
>>> type(2+2.4)
<type 'float'>
>>> type("hallo")
<type 'str'>
```

Compiler
Bytecode-Compiler
Wert
integer
float
string
Typen
Operatoren
Funktion

Etwas formaler ist ein Typ ein Paar aus einer Menge von Werten und einer Menge von Vorschriften, was das Ergebnis der Anwendung von Operatoren auf ein, zwei oder mehr Werte des Typs ist. Das lässt sich tatsächlich sehr strikt definieren, aber für uns ist so viel Überbau zunächst nicht nötig.

Variable

Zuweisung

case-sensitive

Eine *Variable* ist ein Name, der auf einen Wert verweist. Im Allgemeinen werden Werte über Variablen manipuliert. Die einfachste Manipulation ist die *Zuweisung*:

```
>>> a = 2
>>> currentWord = "Das"
```

Wo immer ein Wert stehen kann, kann auch eine Variable stehen:

```
>>> print a
2
>>> type(currentWord)
<type 'str'>
```

Dies erklärt zunächst nur, was eine Variable ist, nicht so sehr, wozu sie gut ist. Das Konzept dazu kommt natürlich aus der Mathematik, wo Variablen benutzt werden, um von konkreten Werten zu abstrahieren. So kann man beispielsweise beobachten, dass $4+2\cdot 2\cdot 3+9=(2+3)^2$ ist und $9+2\cdot 3\cdot 4+16=(3+4)^2$, erst die Idee von Variablen erlaubt es aber, die binomische Formel $a^2+2ab+b^2=(a+b)^2$ zu formulieren (und, nebenbei, zu beweisen). Ganz ähnlich erlauben Variablen eine Abstraktion von Programmen, die *ein* Problem lösen (etwa: Berechne das Quadrat von 2), zu Programmen, die *eine Klasse* von Problemen lösen (etwa: Berechne die n-te Potenz einer beliebigen Zahl).

Erfreulicherweise ist das mit den Variablen meistens nach den ersten paar Versuchen kein Problem mehr.

Namen von Variablen

Variablennamen in Python fangen mit einem Buchstaben oder Unterstrich an und gehen mit beliebig vielen Buchstaben, Unterstrichen oder Zahlen weiter. Sie sind *case-sensitive*, Groß- und Kleinschreibung werden also unterschieden.

Konvention: Variablennamen fangen mit Kleinbuchstaben an.

Tatsächlich ist die wahre und gute Methode, sich Namen für Variablen auszudenken, heiß umstritten. Ich empfehle für Namen, die sich aus mehreren Wörtern zusammensetzen einen Stil wie

```
aLongVariableName
```

Grundsätzlich sollten Variablennamen aussagekräftig sein, jedenfalls, wenn die Variable in mehr als zwei Zeilen verwendet wird. Ein Programm ist nicht nur Kommunikation mit dem Rechner (dem Variablennamen egal sind), sondern auch Kommunikation mit ProgrammiererInnen, die den Code später mal lesen wollen (und natürlich den WiHis, die eure Programme korrigieren); Ziel sollte immer sein, ihnen das Verständnis des Programms zu erleichtern.

Problems

(3.1)* Startet in den Interpreter und gebt verschiedene Werte ein, etwa 200, `"string"` oder 40e7. Lasst euch die Typen dieser Werte ausgeben. Das Konzept des Typs versteht ihr vielleicht am Besten, wenn ihr seht, was die Unterschiede zwischen 200, `"200"` und 200.0 sind, sowohl in der Ausgabe (genau beobachten!) als auch in dem, was `type` zurückgibt.

(3.2)* Überlegt euch ein paar Probleme aus dem Alltag, bei denen ihr letztlich mit Variablen umgeht. Überlegt euch Algorithmen, in denen diese Variablen vorkommen. **(L)**

(3.3)* Definiert ein paar Variablen, zur Not die aus dem Beispiel – ihr weist dazu einfach irgendwelchen Namen irgendwelche Werte zu. Seid kreativ bei den Variablennamen, euch kann nichts Schlimmeres passieren als eine Fehlermeldung. Wendet die Funktion `type` auf eure Variablen an.

(3.4)* Die folgenden Anweisungen lösen jeweils einen Syntaxfehler aus. Begründet in jedem Fall, warum.

```
1dozen = 12
```

```
pass = "something"
five$ = "ca4.6Euro"
```

Ausdruck
expression
Operanden
Operatoren
Präzedenztabelle
Anweisung
statement

4. Ausdrücke und Anweisungen

Ein *Ausdruck* (*expression*) ist eine Kombination von *Operanden* (Werten, Variablen) und *Operatoren*. Einige der Operatoren von Python sind:

- `+`, `-`: Addition und Subtraktion
- `*`, `/`: Multiplikation und Division
- `**`, `%`: Exponentiation, Restbildung

Python kann Punkt vor Strich und Klammern:

```
>>> 2*3-4
2
>>> 2*(3-4)
-2
```

Python kennt weit mehr Operatoren, so dass „Punkt vor Strich“ nicht reicht, um zu schließen, welcher Teil eines Ausdrucks zuerst bewertet wird. Der komplette Regelsatz heißt *Präzedenztabelle*. Wir werden später darauf zurückkommen.

Nicht nur Zahlen können Operanden sein:

```
>>> "bla"*3
'blablabla'
>>> "wahl"+"en"
'wahlen'
```

Eine *Anweisung* (*statement*) ist etwas, das den Interpreter veranlasst, etwas zu tun. Wir kennen schon

- Ausgabe: `print`
- Zuweisung: `a = 3`
- Funktionsaufruf: `type(3)`
- Ausdruck: `2+4`

In der Tat ist ein Funktionsaufruf auch ein Ausdruck.

Werden Ausdrücke dem Interpreter interaktiv gegeben, gibt er deren Wert gleich aus.

```
>>> "hallo"
'hallo'
>>> 4+3
7
>>> print 7,"hallo"
7 hallo
```

Die Zuweisung und Ausgabe sind keine Ausdrücke, sie haben demnach auch keinen Wert. Die Werte, die der Interpreter druckt, sind in der Regel in einem Format, das vom Interpreter wieder verstanden würde, mit `print` kann alles mögliche ausgegeben werden.

In Python ist auch „kein Wert“ ein Wert, und zwar der spezielle Wert `None`. Der Python-Interpreter unterdrückt die Ausgabe von `None`. Mehr dazu später.

Der Klarheit halber: Ausdrücke sind Anweisungen, es gibt aber weit mehr Anweisungen als nur Ausdrücke. Dies ist jedenfalls in Python so, es gibt aber Programmiersprachen, in denen alles Ausdrücke sind (und für die deshalb das Konzept einer Anweisung sinnlos ist). Andere Sprachen realisieren bestimmte Konzepte anders, so ist in C die Zuweisung beispielsweise ein Ausdruck. Damit kann man Anweisungen wie `a = (b = 4)+3` schreiben. In Python geht das *nicht*, was eine Designentscheidung von Pythons Autor Guido van Rossum war. Python unterstützt aber

per Sprachdefinition Ausdrücke der Form `a = b = 4`, die den Wert ganz rechts allen Variablen davor zuordnet.

Dies sind jedoch alles Subtilitäten, die vorläufig nicht so wichtig sind.

Das zweite print-Beispiel oben zeigt übrigens, dass hinter einer print-Anweisung mehrere Werte durch Kommata getrennt stehen können. Python gibt sie dann durch Leerzeichen getrennt hintereinander aus. Endet eine print-Anweisung mit einem Komma, wird hinter der Ausgabe kein Zeilenvorschub ausgegeben. Wir werden später sehen, wie man die Ausgaben von Python-Skripten schöner gestalten kann.

Syntax von Anweisungen

Python erwartet in der Regel eine Anweisung pro Zeile, wobei eine Zeile (hoffentlich) so definiert ist, wie sie euer Editor auch sieht (etwas mehr dazu später im Kapitel über Dateien).

Allerdings können Anweisungen auch mal etwas länger werden. Es ist guter Stil, Zeilen nicht länger als etwa 72 Zeichen werden zu lassen, und so würde man dann und wann gerne eine Anweisung auf mehrere Zeilen verteilen. Python bietet dazu zwei Mechanismen an:

Erstens wird eine Anweisung implizit in der nächsten Zeile fortgesetzt, wenn noch eine Klammer offen ist (im interaktiven Interpreter bekommt ihr dann den Prompt „...“). So könnt ihr beispielsweise

```
someVariable = ((otherVariable+sillyExample)**yikes)*wobbadobba
```

als

```
someVariable = (
  (otherVariable+sillyExample)
    **yikes
  )*wobbadobba
```

schreiben – die Einrückung von links, die sonst für Python relevant ist, ist dabei frei wählbar (und sollte natürlich so gewählt werden, dass die Struktur des Ausdrucks klarer wird). Das ganze geht nicht nur für runde Klammern, sondern auch für eckige und geschweifte, deren Funktion wir später kennen lernen werden.

Die andere Möglichkeit, Anweisungen aufzuteilen ist, einen Backslash ans Ende der ersten Teilzeile zu schreiben:

```
print something, other, yetSomethingElse
```

kann so auch als

```
print something, \
  other, yetSomethingElse
```

geschrieben werden. Das sollte man offensichtlich nur tun, wenn sich keine Klammern anbieten.

Problems

(4.1)* Probiert ein paar mathematische Ausdrücke im Interpreter, also Dinge wie

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

Vergleicht das mit dem, was ihr aus der Schulmathematik mitgenommen habt.

(4.2)* Warum gibt

```
someVariable =
  ((otherVariable+sillyExample)
    **yikes)
  *wobbadobba
```

einen Syntaxfehler? Überlegt euch alle Stellen, an denen die Anweisung (ohne Backslash) umgebrochen werden kann und prüft eure Vermutungen mit dem Interpreter.

5. Funktionen I

Funktion

Argumente

Ergebnis

Eine *Funktion* ist etwas, das einen oder mehrere Wert(e) nimmt (die *Argumente*) und einen oder mehrere Wert(e) zurückgibt (das *Ergebnis*). Das funktioniert fast wie in der Mathematik:

$$f\colon x \mapsto x^2 \qquad f(2) = 4 \qquad f(7) = 49.$$

Wir kennen schon `type`. Einige weitere in Python eingebaute Funktionen:

- `raw_input`: Druckt ggf. ihr Argument und gibt eine Eingabe des Benutzers als String zurück
- `int`: Gibt ihr Argument als ganze Zahl zurück
- `float`: Gibt ihr Argument als Fließkommazahl zurück
- `str`: Gibt ihr Argument als String zurück
- `len`: Gibt die „Länge“ ihres Arguments zurück

```
>>> eurInMark = 1.95583
>>> s = raw_input("EUR? ")
EUR? 10
>>> float(s)*eurInMark
19.558299999999999
>>> int(float(s)*eurInMark*100)/100
19
>>> int(float(s)*eurInMark*100)/100.
19.550000000000001
```

Funktionen wie `float` oder `len` kann man sich vorläufig wie Kommunikationsmittel mit Werten vorstellen: `float("19")` fragt den Wert `"19"`, was er als Fließkommazahl sei, und `len("19")` fragt ihn, für wie lang er sich hält.

Es kann passieren, dass ein Wert keine Antwort auf so eine Frage hat. Python reagiert dann mit einem Fehler:

```
>>> int("23.3")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 23.3
>>> float("zwei komma drei")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for float(): zwei komma drei
>>> len(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: len() of unsized object
```

Die Funktion `len` funktioniert also nach dieser Aussage nur auf Werte, die „sized“ sind. Wir werden etliche Werte dieser Sorte kennenlernen; im Augenblick haben nur Strings so eine Größe, nämlich gerade ihre Länge in Zeichen.

Im Übrigen habe ich hier gelogen. Wenigstens in modernen Python-Versionen sind `str`, `float` und `int` keine normalen Funktionen mehr. Das gibt Python auch freimütig zu:

```
>>> type(len)
<type 'builtin_function_or_method'>
>>> type(str)
<type 'type'>
```

Was es mit diesem `type 'type'` auf sich hat, werden wir in einer Weile sehen.

Exkurs: Integer Division und Rundungsfehler

integer division

binär

Rundungsfehler

Am Beispiel oben sieht man ein paar kritische Punkte im Umgang mit Zahlen im Computer.

(1) `1955/100` ist für Python 19. Grund: „/“ ist (bisher) ganzzahlige Division, wenn beide Operanden ganzzahlig sind. Bei der ganzzahligen Division (*integer division*) werden alle Nachkommastellen, die bei einer Division auftauchen, weggeworfen (oder die Reste ignoriert). Demnach ist `1/2==0`.Abhilfe durch Kennzeichnung eines Operanden als `float`:

```
>>> 1/float(2)
0.5
>>> 1/2.
0.5
```

Das verwirrende Verhalten des / gilt als „Warze“ (und Erbe von C) und wird in künftigen Python-Versionen anders sein. In Python 2.2 kann man ein vernünftigeres Verhalten einschalten (die integer division wird dann durch // markiert):

```
>>> from __future__ import division
>>> 1/2
0.5
```

(2) `1955/100.` ist nicht genau 19.55. Grund: Fließkommazahlen werden nur mit endlicher Genauigkeit und *binär* (also mit den Ziffern 0 und 1 statt mit den Ziffern von 0 bis 9) dargestellt. 19.55 bricht dezimal ab, binär aber nicht. Die Folge sind *Rundungsfehler*, mit denen Fließkommazahlen immer gerechnet werden muss.

Folge: Vergleiche von Fließkommazahlen sind unzuverlässig und sollten umgangen oder „fuzzy“ gemacht werden.

Die Hässlichkeit fällt hier auf, weil die Zahl als Ergebnis einer Anweisung dargestellt wurde, so dass Python die volle Genauigkeit (bzw. Ungenauigkeit) darstellt. Eine normale Ausgabe sieht nicht so schlimm aus:

```
>>> print int(float(s)*eurInMark*100)/100.
19.55
```

(was natürlich am Problem nichts ändert).

Problems

(5.1)* Probiert *alle* Beispiele auf dieser Seite aus. Macht euch klar, warum welche Beispiele funktionieren und warum die, die nicht funktionieren, einen Fehler werfen. Wenn ihr dafür sorgen müsstet, dass `int("23.3")` funktionieren sollte, statt einen Fehler zu werfen, was würdet ihr tun?

(5.2)* Führt

```
>>> print type(raw_input())
```

für verschiedene Eingaben aus, etwa 3.3, hallo, //(), 027, 3+3 oder `"ein String"`. Was beobachtet ihr? Warum könnten die Python-MacherInnen das so gemacht haben? **(L)**

(5.3) Vergleicht die Ausgaben von

```
>>> 1/2.,1/4.,1/8.,1/16.,1/32.
```

und

```
>>> 1/5.,1/10.,1/20.,1/25.,1/40.
```

Was fällt auf? Könnt ihr euch (insbesondere im Hinblick darauf, dass Computer intern nur Nullen und Einsen speichern können) einen Reim auf dieses Verhalten machen? **(L)**

6. Strings als Objekte

Objekte

Methoden

Strings sind in Python *Objekte* (Vorsicht! In Python vor 2.0 war das noch etwas anders, das Folgende stimmt für die mittlerweile antike Version 1.5 nicht).

Objekte enthalten neben Daten auch Funktionen, die dann *Methoden* heißen. Methodenaufrufe sehen wie Funktionsaufrufe aus:

```
>>> s = "Hallo"
>>> s.find("ll")
2
```

Zunächst kommt also das Objekt, dessen Methode verwendet werden soll (`s`), dann ein Punkt und dann der Name der Methode; `find` sucht im String nach einem anderen String und gibt dessen Position zurück. Der erste Buchstabe hat die Position 0 (!), ist der Suchstring nicht enthalten, wird -1 zurückgegeben.

```
>>> s.find("Ha")
0
>>> s.find("Ha!")
-1
```

Die -1, die find zurückgibt, wenn es den String nicht gefunden hat, ist etwas wie ein „magischer Wert“ – etwas, das eine besondere und sozusagen verabredete Bedeutung über die eigentliche Interpretation des Wertes hinaus hat. In diesem Fall sagt es einfach: Die -1 ist nicht etwa eine Position, sondern eben etwas wie eine Fehlerkennung.

(Weiterführend:) In diesem Fall ist das nicht sehr schön; es wäre besser gewesen, wenn die Designer hier z.B. None zurück gegeben hätten, denn dann hätte man keine Magie gebraucht. In der Regel hat man in Python aber im Fehlerfall eine ganz andere Reaktion, es wird nämlich eine Exception ausgelöst. Ein Ersatz für find, der das tut, heißt `index`. Allerdings ist nicht ganz klar, dass die Abwesenheit eines Substrings in einem String wirklich ein „Fehler“ ist, und so ist das Verhalten von find schon zu rechtfertigen. Aber das nur nebenbei.

Strings haben viele Methoden (vgl. Python-Referenz[2]), etwa

- `count` (zählt, wie oft ein String im anderen vorkommt)
- `lower`, `upper` (wandelt alle Groß- in Kleinbuchstaben und umgekehrt)
- `replace` (ersetzt Vorkommen eines Substrings durch einen anderen)

```
>>> s = "Labala"
>>> s.replace("a", "ug")
'Lugbuglug'
>>> s.count("la")
1
>>> s.lower()
'labala'
>>> s.lower().count("la")
2
```

Das letzte Beispiel zeigt: Das Ergebnis einer Funktion, die einen String zurückgibt, ist natürlich wieder ein Objekt mit Methoden.

(Weiterführend:) In der Tat sind in modernem Python eigentlich alle Werte in Wirklichkeit Objekte (immerhin nennt sich Python ja auch objektorientiert). Sogar Zahlen haben Methoden, die man allerdings aus syntaktischen Gründen (gegenwärtig) etwas ausbuddeln muss (der Punkt ist in Zahlen als Dezimaltrenner reserviert). So lässt sich etwa der Ausdruck `1+1` etwas verquast als

```
>>> (1).__add__(1)
2
```

[2] `http://docs.cl.uni-heidelberg.de/python/lib/string-methods.html`

schreiben. Warum das wie geht, ist vorerst nicht wichtig, es ist aber nützlich, die Vorstellung zu pflegen, dass die Anwendung von Operatoren und Funktionen auf Werte in Python meistens den Aufruf von Methoden der Operanden nach sich zieht.

In diesem Sinn ist auch die Behauptung von oben, `len` „frage“ sein Argument nach seiner Länge zu verstehen – `len` versucht einfach, eine Methode namens __len__ in seinem Argument aufzurufen. Wenn ein Objekt diese Methode nicht hat, sagt `len` eben, das Objekt habe keine Größe (besser wäre vielleicht: Es kennt sie nicht).

Problems

(6.1)* Macht euch klar, dass Methoden zu ihren Objekten gehören. Es klingt vielleicht ein bisschen blöd, aber es ist keineswegs klar, dass die beiden „replaces“ im folgenden Dialog (probiert ihn!) wirklich verschieden sind:

```
>>> s1 = "foo"
>>> s2 = "bar"
>>> s1.replace("o", "e")
>>> s2.replace("o", "e")
```

(6.2)* Die Methoden von Strings ändern *nie* den String selbst – sie geben neue Objekte zurück. Probiert folgendes und erklärt, was dabei passiert:

```
>>> s = "foo"
>>> s.replace("o", "e")
>>> print s
>>> s = s.replace("o", "e")
>>> print s
```

(6.3) Den Unterschied zwischen Funktionen und Methoden kann man bei Strings besonders gut sehen, weil Python aus den Zeiten, als Strings noch keine Objekte waren, die meisten String-Methoden auch noch als Funktionen bereitstellt. Man muss dem Interpreter allerdings erst sagen, dass man sie verwenden will, und zwar mit der Anweisung `from string import <methodenname>` (wir werden später sehen, was das bedeutet). Probiert also folgendes:

```
>>> from string import replace
>>> "foo".replace("o", "e")
>>> replace("foo", "o", "e")
```

Ihr seht, dass, wie behauptet, ein Methodenaufruf sowas ist wie ein Funktionsaufruf, nur dass das Objekt, auf das die Methode angewandt wird, im ersten Argument der Funktion steht.

Mit anderen Klassen, deren Methoden nicht auch irgendwo als Funktionen stehen, lassen sich dennoch ähnliche Dinge tun. Wir werden das Äquivalent der replace-Funktion aus dem string-Modul später als „unbound method“ kennenlernen.

(6.4)* Lest die Dokumentation zum String-Modul oder zum String-Typ. Ihr werdet vielleicht noch nicht so furchtbar viel verstehen, aber es ist wichtig, dass ihr wisst, wie ihr auf die Dokumentation zugreift. Häufig haben IDEs eingebaute Hilfen, was jedoch immer geht, ist ein Blättern in der Library Reference in HTML[3] und, so euer Python ordentlich installiert ist, mit dem Kommandozeilenprogramm pydoc. Tippt einfach in ein Shellfenster pydoc str oder pydoc string.

[3] `http://docs.cl.uni-heidelberg.de/python/lib/lib.html`

7. Funktionen II

Funktionsdefinition

compound statement

Verbundanweisung

reservierte Wort

Wir können auch eigene Funktionen definieren:

```
>>> def readAndPrintLower():
...     s = raw_input()
...     print s.lower()
...
```

Eine *Funktionsdefinition* wird also mit `def` eingeleitet, dann kommt ein Name (wie Variablenname), dann Klammern und ein Doppelpunkt. Anschließend kommen Anweisungen, alle gleich weit eingerückt (wie weit, ist Geschmackssache. Vorerst empfehle ich, jeweils vier Spaces weit einzurücken.

Grundsätzlich schreibt Python schon in der Sprache relativ viel „code layout“ vor. Trotzdem gibt es Konventionen, wie man Code zu schreiben sollte, um es anderen Menschen leichter zu machen. Autoritativ ist hier PEP 008[4], das ihr jetzt schon mal überfliegen solltet. Die Empfehlungen darin sind nicht entscheidend dafür, dass der Rechner mit eurem Code zurechtkommt, wohl aber dafür, dass routinierte Pythonistas ihn schnell und einfach lesen können. Haltet euch dran, und lest den Text nochmal nach dem Ende dieses Kurses.

Ausnahme: Die Empfehlung, Methodennamen mit underscores zu konstruieren (foo_bar statt fooBar) ist Mist. Ich werde sie in diesem Kurs ignorieren, und mein Tipp ist, dass ihr das ähnlich haltet. .

Die beiden Anweisungen bilden ein *compound statement* (eine *Verbundanweisung*). Sie endet, wenn wieder „ausgerückt“ wird. Compound statements sind sozusagen die Nebensätze von Programmiersprachen: In ihnen wird eine Sequenz von zusammengehörigen Anweisungen zusammengefasst, die dann syntaktisch eine Anweisung bilden. Bei Funktionen ist noch nicht wirklich gut zu sehen, wie das funktioniert und wozu das gut ist, klarer wird das in Verbindung mit Selektion und Iteration.

Solange wir ein compound statement eingeben, gibt der Interpreter statt des normalen Prompts (>>>) drei Punkte aus.

Die Funktion können wir wie eingebaute Funktionen aufrufen:

```
>>> readAndPrintLower()
HaCk3rZ
hack3rz
```

Funktionen haben im Allgemeinen Argumente und geben einen Wert zurück. Das wird so definiert:

```
>>> def incString(s):
...     i = int(s)
...     i = i+1
...     return str(i)
...
>>> incString("5")
'6'
```

Das Argument kommt also in die Klammern, das *reservierte Wort* return steht vor dem, was zurückgegeben wird (und beendet die Funktion).

Reservierte Wörter, von denen es in Python gegen 30 gibt, sind eine Besonderheit. In Python können fast alle Namen mit neuen Bedeutungen belegt werden, insbesondere auch die Namen der eingebauten Funktionen (Falle!):

```
>>> int = 4
>>> int("35")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'int' object is not callable
```

[4] http://www.python.org/peps/pep-0008.html

Bei den reservierten Wörtern geht das nicht, sie sind aus Sicht des Sprachdesigners gleichgestellt mit Operatoren wie + und – oder auch Klammern und Leerzeichen:

stack trace

```
>>> return = 4
  File "<string>", line 1
    return = 4
           ^
SyntaxError: invalid syntax
```

For the record, die reservierten Wörter von Python sind zur Zeit:

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

Wer versucht, reservierte Wörter als Variablennamen zu verwenden, wird, wie oben gezeigt, mit Syntaxfehlern bestraft.

Die Funktion „inkrementiert“ eine Zahl, die in einem String steht. Wir jonglieren hier mit verschiedenen Typen. Addition von Strings und Zahlen geht nicht.

```
>>> "4"+1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Die Funktion wehrt sich, wenn man von ihr etwas will, für das wir sie nicht geschrieben haben:

```
>>> incString("bla")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in incString
ValueError: invalid literal for int(): bla
```

Wir werden bald sehen, was man tun kann, um solche Fehlermeldungen (*stack trace*) zu vermeiden oder Fehlerbedingungen sinnvoll zu behandeln.

Ein weiterer interessanter Aspekt der Funktion ist das `i = i+1`. Mathematisch ist das blanker Unsinn. Für Python bedeutet diese Anweisung aber:

1. Zuweisung! Sehen wir erstmal die rechte Seite an.
2. Die rechte Seite ist ein Ausdruck – das können wir berechnen.
3. Wir nehmen den Wert von `i`, addieren eins drauf und haben jetzt einen Wert
4. So, wir haben einen Wert, und wir waren am Zuweisen – was steht also auf der linken Seite? Oh, `i` zuweisen, lassen wir also jetzt `i` auf den berechneten Wert zeigen. Das, worauf `i` vorher gezeigt hat, interessiert nicht mehr.

Problems

(7.1)* Probiert die Funktion incString aus. Schreibt sie so um, dass sie ihr Argument um 5 erhöht. Was passiert, wenn ihr das int weglasst? Warum?

(7.2)* Schreibt eine Funktion decString, die eine in einen String verpackte Zahl nimmt, eins von der Zahl abzieht und das Ergebnis wieder als String zurückgibt.

(7.3)* Macht euch den Unterschied zwischen reservierten Wörtern und eingebauten Funktionen klar. Probiert folgendes aus und erklärt die Fehlermeldungen, die ihr seht (oder nicht seht):

```
>>> return = 1
>>> int = return
>>> raw_input = 2
>>> b = raw_input("Anzahl: ")
>>> raw_input = int
>>> raw_input("123")
```

"Whereas recognition of the inherent dignity and of. . ."

Referenz

Namespace

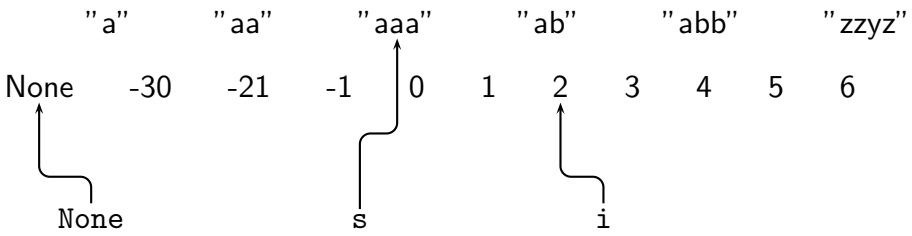


Fig. 1

"Whereas recognition of the inherent dignity and of. . ."

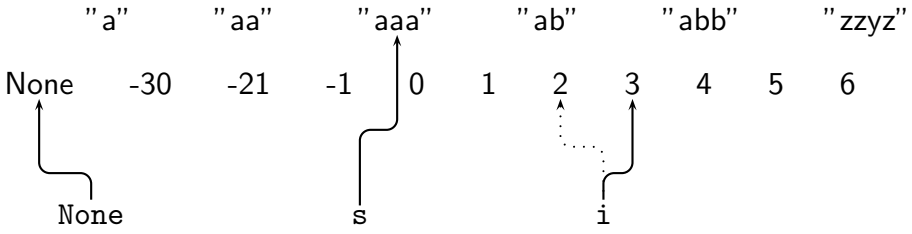


Fig. 2

(7.4)* Macht euch klar, wie i = i+1 funktioniert, etwa, indem ihr beobachtet, was bei einem Dialog passiert, in dem ihr folgendes tippt:

```
>>> i = 2
>>> i+1
>>> print i
>>> i = i+1
>>> print i
```

8. Das Datenmodell von Python

In Python funktionieren Variablen mehr wie in Sprachen der Lisp-Familie, was sich deutlich von dem unterscheidet, was ihr von C, Pascal oder ähnlichem gewöhnt sein mögt.

Die beste Vorstellung ist zunächst, dass in einem großen Raum alle Zahlen, Strings usf. bereits vorhanden sind und eine Zuweisung lediglich die Verbindung zwischen einem Namen und dem Wert herstellt – eine *Referenz*. In diesem Bild haben die Anweisungen

```
>>> s = "aaa"
>>> i = 2
```

folgenden Effekt:

(cf. Fig. 1)

In dieser Grafik stehen oben ein paar der (unendlichen vielen) Werte, die so im „Python-Universum“ umherschwirren – ein paar Strings, darunter natürlich auch die Universelle Erklärung der Menschenrechte, ein paar Zahlen. Unten stehen dann die Namen, die schon definiert wurden – sie sind nicht eigentlich Teil des Python-Universums, sondern bilden einen eigenen Raum, den Namensraum oder *Namespace*.

Da Namen nicht im Python-Universum, sondern quasi im Hyperraum leben, gibt es auch kein Python-Konstrukt, das erlauben würde, Referenzen auf Namen zu bekommen (allerdings kann man Namen ins hoch ins Universum schaffen und geeignete Werte aus dem Universum zu Namen machen – wir werden so etwas später als Introspektion kennenlernen).

```
i = i+1
```

bewirkt jetzt Folgendes:

"Whereas recognition of the inherent dignity and of. . ."

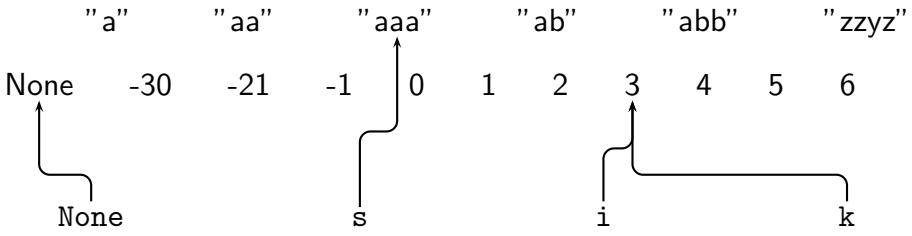


Fig. 3

(cf. Fig. 2)

Gepunktet ist der Zustand vor der Zuweisung eingezeichnet.

Später sehr wichtig wird der Effekt von
`k = i`

– nämlich:

(cf. Fig. 3)

– k teilt den *Wert* von i. Es gibt keine „neue“ Drei. Solange wir Werte nicht verändern können, ist das aber nicht schlimm.

Man Namen auch wieder löschen. Dazu gibt es das reservierte Wort `del`. Del operiert beispielsweise so:

```
>>> i = 2
>>> k = i
>>> k
2
>>> del k
>>> k
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'k' is not defined
```

Beachtet: Wir haben die *Referenz* k gelöscht, aber natürlich nicht den *Wert* 2:

```
>>> i
2
```

Auf der positiven Seite muss man in Python nur selten wirklich mal Namen löschen. del löscht allerdings allgemein Referenzen, und diese Eigenschaft wird uns bei der Manipulation von Sequenzen und Mappings nützlich sein.

(Weiterführend:) In Wirklichkeit sind die Verhältnisse natürlich etwas komplizierter als unser simples Modell mit dem Python-Universum es scheinen lässt. Es ist klar, dass ein Rechner mit endlichem Speicher nicht unendlich viele Werte in seinem Speicher halten kann, ganz abgesehen von der Frage, wie diese eigentlich erzeugt werden. Deshalb stellt Python die meisten Werte quasi on the fly her, wenn sie gebraucht werden und wirft sie wieder weg, wenn sie nicht mehr gebraucht werden (also keine Referenz mehr auf sie verweist). Dabei kann es unter Umständen auch mal zu Überproduktion kommen.

(Weiterführend:) Um sich das anzusehen, könnt ihr die eingebaute Python-Funktion `id` verwenden. Diese gibt eine eindeutige „Kennung“ eines Wertes zurück, mit der sich die einzelnen Objekte des Python-Universums unterscheiden lassen. Unter Python 2.3 passiert dabei etwa so etwas:

```
>>> a = "bla"
>>> b = "bla"
>>> id(a)
1076955040
>>> id(b)
1076955040
```

```
>>> c = "b"+"la"
>>> id(c)
1076954912
>>> a==c
True
>>> a is c
False
>>> a is b
True
```

(Weiterführend:) Hier war Python also schlau genug, bei der Zuweisung von `b` zu merken, dass es `"bla"` schon erzeugt hat und verwendet diesen Wert weiter. Bei der Zuweisung von `c` hat das aber nicht geklappt, im Python-Universum wohnt jetzt zwei Mal der Wert `"bla"` – allerdings so ununterscheidbar wie eineiige Zwillinge, deshalb ist auch `a == c`. Es gibt aber einen Operator, der nachsieht, ob die Werte nicht nur gleich aussehen, sondern auch gleich sind: `is`. Wir werden später noch sehen, wozu das gut sein kann. Bis dahin sind all die Referenzfragen ziemlich akademisch; die Frage der Identität von Strings ist im Rahmen des hier behandelten Materials völlig irrelevant, zumal es sich dabei um ein „Implementierungsdetail“ handelt, der Sprachstandard macht keine Aussagen, ob nach obigem Code `a is c` wahr sei oder nicht. Code, der sich auf ein bestimmtes Verhalten verlässt, wird in späteren Versionen von Python sehr wahrscheinlich nicht mehr funktionieren. Das, was oben beschrieben wurde (das „idealisierte“ Python-Universum) reicht für alle unsere Betrachtungen aus und hat letztlich mehr „Wahrheit“ als die harte Realität in diesen Ausführungen.

Und nochmal sei betont, dass die meisten Sprachen, mit denen ihr in der Schule Kontakt gehabt haben könntet, diese Sachen ganz anders machen.

Die Katze auf der Terrasse

Eine Folge dieser Logik ist, dass Werte in Python nicht wirklich Namen haben – Namen sind an Werte gebunden, nicht umgekehrt. Eine schöne Illustration davon ist folgendes Gleichnis, das auf den effbot (Frederik Lundh) zurückgeht:

> Mit Python-Objekten ist es wie mir der Katze, die du irgendwann schlafend auf deiner Terrasse vorfindest. Ganz wie ein Python-Objekt kann dir die Katze nicht sagen, wie sie heißt – es ist ihr auch ganz egal. Um ihren Namen herauszufinden, wirst du wohl deine Nachbarn fragen müssen, und du solltest nicht überrascht sein, wenn du herausbekommst, dass die Katze viele Namen hat.

Die Nachbarn, die man nach den Namen der Katze fragen kann, sind genau die Namespaces – es ist durchaus möglich, dass ein und dasselbe Objekt in verschiedenen Namespaces verschiedene Namen hat.

Problems

(8.1)* Malt für die folgende Sequenz von Python-Anweisungen passende Ausschnitte des (idealisierten) Python-Universums, in die die dann aktiven Referenzen eingezeichnet sind.

```
>>> a = 3
>>> b = 0
>>> c = a
>>> d = a-c
>>> e = 2.71
>>> f = int(e/e)
```

9. Funktionen III

Faktorisierung

Signatur

Warum Funktionen?

- Zerlegen eines großen Problems in überschaubare Teilprobleme mit klaren *Schnittstellen* (d.h. es ist nur wichtig, was reinkommt und was rauskommt, nicht aber, wie es wirklich geschrieben ist)
- Lokalität (d.h. Veränderungen von Code an einer Stelle werden im Allgemeinen nur dort Konsequenzen haben)
- Wiederbenutzbarkeit (d.h. ein einmal gelöstes Teilproblem muss, wenn es wieder auftritt, nicht noch einmal gelöst werden)

Etwas überspitzt könnte man sagen, die Aufgabe eines/r ProgrammiererIn sei, Probleme in überschaubare Teile zu zerlegen, die dann einzeln implementiert werden können – eben zunächst in Funktionen, aber auch in Klassen und Module, wovon wir später hören werden. In jedem Fall ist diese *Faktorisierung* das, was am (guten) Programmieren eigentlich schwierig ist, und was noch für lange Zeit maschinell nicht zu machen sein wird. Auch die so oft geforderten „Ingenieursmethoden“ bei der Softwareentwicklung können hier allenfalls eine Hilfestellung liefern.

Umgekehrt bleibt die Fähigkeit zu guter Faktorisierung euch erhalten, egal, in welcher Sprache ihr schließlich programmiert, denn auch wenn verschiedene Sprachen verschiedene Mittel bieten, das Grundproblem der gelungenen Problemanalyse teilen sie alle. In diesem Sinn ist das höchste Ziel dieses Kurses, euch ein Stück auf den Weg zu dieser Fertigkeit zu leiten.

Ein kleines Skript

```
def incString(s):
  i = int(s)
  i = i+1
  return str(i)

s = raw_input("Eine Zahl? ")
print incString(s)
print incString(raw_input(
  "Noch eine Zahl? "))
```

Laufen lassen:

```
examples> python simplescript.py
Eine Zahl? 18
19
Noch eine Zahl? -20
-19
```

Signaturen

Die *Signatur* einer Funktion sagt, welche Argumente sie bekommt und was sie zurückgibt, definiert also gerade die erwähnte Schnittstelle zwischen der Funktion und der „Außenwelt“. In Python werden sie konventionell so geschrieben:

```
<Funktionsname>(<Argumente>) -> <Ergebnistyp>
```

Das kann konkret so aussehen:

```
raw_input([prompt]) -> string
int(x[, base]) -> integer
str(object) -> string
len(object) -> integer
```

Für die Methoden von Strings:

```
S.count(sub[, start[, end]]) -> int
```

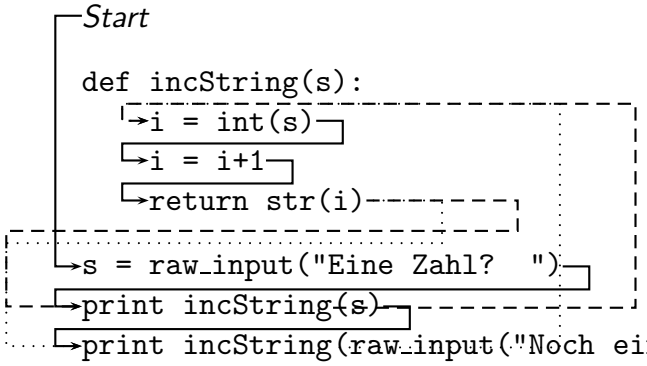


Fig. 4

```
S.lower() -> string
S.replace (old, new[, maxsplit]) -> string
```

Für jedes Argument (von denen es offenbar auch mehr geben kann, sie werden dann durch Kommata getrennt) schreibt man also immer ein Wort, das andeuten soll, was die Bedeutung des Arguments (für die Funktion) ist. In eckigen Klammern sind dabei „optionale“ (d.h., sie können übergeben sein oder auch nicht) Argumente angegeben. Bei unseren Beispielen zu replace hatten wir etwa nur zwei Argumente übergeben. Wir sehen hier, dass wir noch ein drittes hätten übergeben können. Was das dann bedeutet, entnimmt man der Dokumentation, zumal, wenn die Namensgebung so unglücklich ist wie hier: Ist das dritte Argument vorhanden, sollte es eine Zahl sein, die angibt, wie viele Vorkommen von old durch new maximal ersetzt werden sollen. Übergibt man etwa 1, wird nur das erste Vorkommen ersetzt (oder gar nichts, wenn old im String gar nicht vorkommt).

(Weiterführend:) In den Headern von C-Programmen stehen übrigens typischerweise ganz ähnliche Sachen, nämlich die so genannten Prototypen. Da C nicht Pythons dynamische Typisierung hat, muss darin allerdings immer noch der Typ jedes Arguments angegeben werden, und der Typ des Rückkehrwerts steht vor dem Funktionsnamen.

Control Flow

(cf. Fig. 4)

Code in Funktionen wird erst ausgeführt, wenn die Funktion aufgerufen wird. Der *Control Flow* (also die Reihenfolge, in der der Code durchlaufen wird) wird aber ziemlich konfus, wenn viele Funktionen aufgerufen werden. Python kann sich das merken, Menschen nicht. Deshalb:

Über Funktionen nicht im Control Flow nachdenken („goto-Bild“), sondern in Schnittstellen: Was kommt rein, was kommt raus?

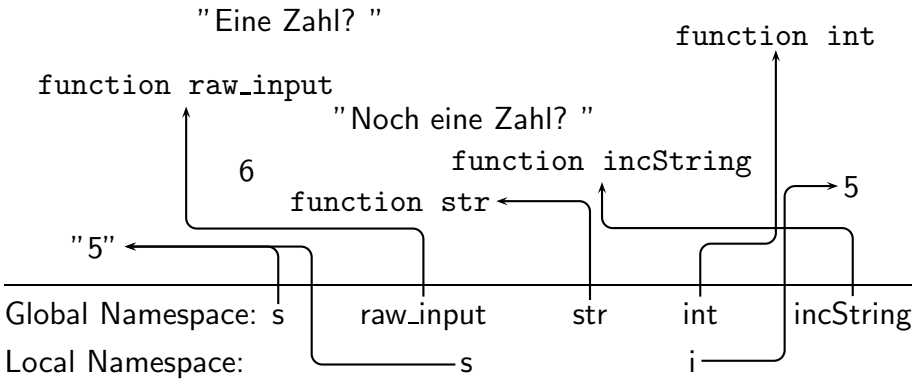
Problems

(9.1)* Wie ist die Signatur der incString-Funktion? **(L)**

(9.2)* Verändert das „Hauptprogramm“ des angegebenen Skripts zu

```
i = "23"
print incString(i)
print i
```

Was beobachtet ihr? **(L)**



lokal

global

Fig. 5

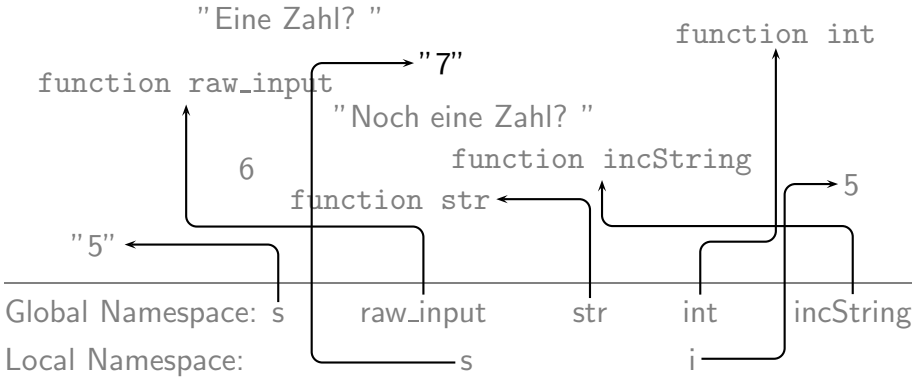


Fig. 6

10. Lokale und Globale Variablen

Damit die Schnittstellen einer Funktion überschaubar bleiben, beeinflusst die Manipulation einer Variable innerhalb einer Funktion den Rest des Programms normalerweise nicht.

In Python ist das dadurch realisiert, dass jede Funktion einen eigenen Namespace hat, der beim Aufruf neu erzeugt wird und verschwindet, wenn sie fertig ist. Dieser Namespace heißt *lokal*.

Der Namespace, in dem man außerhalb einer Funktion ist, heißt *global*er Namespace. Er spielt eine besondere Rolle, weil in ihm nachgesehen wird, wenn die Suche im lokalen Namespace erfolglos war.

Wenn wir 5 eingegeben haben, sieht es in der Funktion incString folgendermaßen aus:

(cf. Fig. 5)

Macht euch klar, was passiert, wenn wir in `incString` die Funktion `int` aufrufen – ein Funktionsaufruf ist ja nichts anderes als die Aufforderung an Python, nach dem an den Namen gebundenen Wert zu suchen und diesem Wert (sinngemäß) zu sagen: Hör zu, hier hast du ein paar Argumente, verarbeite sie doch bitte mal.

Im lokalen Namespace von `incString` stehen zu diesem Zeitpunkt nur `s` und `i`, die Suche nach `int` muss also scheitern. Danach sucht Python im globalen Namespace und wird dort fündig. Wäre schon ein int im lokalen Namespace gewesen, wäre das aufgerufen worden.

Zuweisungen landen immer im lokalen Namespace. Wenn man etwa an den Anfang von `incString` ein `s = "7"` geschrieben hätte, wäre die Situation danach so:

(cf. Fig. 6)

Beachtet: `s` ist im globalen Namespace immer noch an `"5"` gebunden. Sobald die Funktion vorbei ist, erinnert nichts Python daran, dass es mal ein `s` gab, das an `s = "7"` gebunden war.

Es ist in Ordnung, Funktionen und ähnliches aus dem globalen Namespace zu holen.

Es ist gefährlich und böse, Variablen aus dem globalen Namespace zu holen („globale Variablen“). Hauptgrund dafür ist, dass auf diese Weise die Schnittstelle der Funktion nicht mehr nur durch ihre Argumente und ihren Rückgabewert definiert ist, sondern auch noch durch den Wert der verwendeten globalen Variablen. Das entwertet die Lokalität der Funktion erheblich.

Im Prinzip würde das ähnlich natürlich auch für Funktionen gelten – aber die Schnittstelle einer Funktion ändert sich im Allgemeinen über einen Programmlauf hinweg nicht, so dass Funktionen als Konstanten aufgefasst werden können. Konstanten aber gehören nicht zur Schnittstelle einer Funktion. Da sich eine Konstante nämlich per definitionem nicht ändern kann, kann sie auch das Verhalten der Funktion nicht beeinflussen.

Deshalb: Alle Daten, die eine Funktion braucht, müssen aus den Argumenten kommen.

Problems

(10.1)* K. R. Acher hat sein `incString` so geschrieben:

```
def incString(str):
  i = int(str)
  i = i+1
  return str(i)
```

Er ist ein wenig überrascht, als sein Python folgendes tut:

```
>>> incString("7")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 4, in incString
TypeError: 'str' object is not callable
```

Was ist passiert? **(L)**

11. Logik und Selektion

Logische Ausdrücke

Die Operatoren logischer Ausdrücken verknüpfen ihre Operanden so, dass sich entweder wahr oder falsch ergibt; Vergleiche verhalten sich so:

```
>>> s = "bla"          >>> 5 <= 4
>>> "bla" == s         0
1                      >>> s != 5
>>> "aba" > s          1
0
```

Logisch falsch sind in Python die 0 (und der leere String `""` sowie der spezielle Wert `None` und etliche andere „leere“ Datenstrukturen) , alles andere zählt als wahr.

Ab Python 2.3 gibt es extra für Logik einen neuen Typ: `bool`. Das Beispiel oben sieht dann so aus:

```
>>> s = "bla"           >>> 5 <= 4
>>> "bla" == s         False
True                   >>> s != 5
>>> "aba" > s          True
False
```

Die Details[5] sind etwas jenseits unseres gegenwärtigen Horizonts, wie auch die Frage, warum das wohl gemacht wurde. So oder so, abgesehen von einigen interaktiven Antworten ändert sich dadurch am hier dargestellten so gut wie nichts, zumal die neu vordefinierten Namen `True` und `False` für praktische alle Zwecke 1 und 0 sind. Das erlaubt auch Perversionen wie

[5] `http://www.python.org/peps/pep-0285.html`

```
>>> 7+True
8
```

compound statement

– aber sowas will man natürlich wirklich nicht machen.

Weitere Operatoren erlauben die Verknüpfung logischer Werte:

```
>>> 5==s or "bla"==s
1
>>> 5==s and "bla"==s
0
>>> not 5==s
1
```

Selektion

Oft muss ein Programm in Abhängigkeit von Bedingungen verschiedene Dinge tun. In Python gibt es dafür `if`:

```
>>> n = 5
>>> if n>7:
...   print n, "> 7"
... else:
...   print n, "< 8"
...
5 < 8
```

```
>>> s = raw_input("Name? ")
Name? ich
>>> if s=="ich":
...   print "Sehr witzig"
...
Sehr witzig
```

Wieder kommt nach der `if`- oder der `else`-Zeile ein Doppelpunkt, die Zeilen, die in Abhängigkeit von der Bedingung ausgeführt werden sollen sind eingerückt.

Wir haben es hier also wieder – wie schon bei den Funktionskörpern – mit einem *compound statement* zu tun. Die Grundidee ist, dass der Rechner wissen muss, bis wohin eine Funktion oder ein Zweig einer Selektion geht (im Nebensatzbild also: Was alles zum Nebensatz gehört). Offensichtlich ist das wichtig, wenn wir eine Sequenz in einem Selektionszweig haben, wie folgendes Beispiel illustrieren mag:

„Wenn es sonnig ist, hole die Strandmatte und lege dich auf die Neckarwiese“

In der Natursprache wird die Entscheidung, ob die zweite Aufforderung der ersten beigeordnet (in Sequenz mit „holen“) ist oder unabhängig steht (in Sequenz mit „Wenn“), der Semantik oder der Pragmatik überlassen. Computer können das in der Regel nicht, und so brauchen Computersprachen syntaktische Merkmale, um zwischen den beiden Alternativen

Wenn es sonnig ist

    hole die Strandmatte

lege dich auf die Neckarwiese

und

Wenn es sonnig ist

    hole die Strandmatte

    lege dich auf die Neckarwiese

zu entscheiden – und eben das tut die Syntax der compound statements. Dass diese in Python durch Einrückung markiert werden, ist eher ungewöhnlich. Viele Sprachen, vor allem die von C abgeleiteten, verwenden lieber geschweifte Klammern, in Pascal gibt es die Schlüsselwörter begin und end, übliche Unix-Shells machen das eher ad hoc und schließen ein if-statement mit fi, ein case-statement mit esac und haben für Funktionskörper geschweifte Klammern.

Will man, dass Programme für Menschen lesbar sind, wird man aber auch in diesen Sprachen einrücken, weil es für Menschen einfacher ist, einer Einrückung zu folgen als Klammern zu zählen. Guido van Rossum, Pythons Spiritus Rector, dachte sich, dass der Rechner auch nicht (viel) schlechter als Menschen darin ist, Einrückungen zu erkennen und beschloss, dann auf eine

separate Kennzeichnung der Struktur extra für den Rechner gleich zu verzichten. Wie gut diese Idee war, werden wir in Programmieren II sehen, wenn es um das „dangling else problem“ geht.

Vor allem an C angelehnte Sprachen haben ein weiteres Konstrukt zur Selektion, das nicht-bool'sche Entscheidungen (also solche, in denen die Bedingung nicht nur zwei Werte annehmen kann) behandelt. Das wären etwa Dinge wie „Wenn die Taste a gedrückt wurde, öffne die Tür, wenn die Taste b gedrückt wurde, schließe sie, und wenn q gedrückt wurde, verlasse das Programm“ oder so. Solche Konstrukte werden gern mit Schlüsselwörtern wie switch oder case gemacht.

Python hat kein `switch`-Statement, das eine mehrfache Selektion erlaubt („Wenn A, dann x, wenn B, dann y, wenn C, dann z usf.“, wobei A, B, C,... ausschließend sind – andere Sprachen haben dafür eigene Syntax). Ersatz durch `elif`-Ketten:

```
>>> def printRelation(x, y):
...   if x<y:
...     print x, "<", y
...   elif x>y:
...     print x, ">", y
...   else:
...     print x, "==", y
...
```

```
>>> printRelation(4, 3)
4 > 3
>>> printRelation(5, 5)
5 == 5
```

(Weiterführend): Häufig wird man Konstruktionen dieser Art auch durch andere, meist elegantere, Sprachmittel realisieren – so können beispielsweise die uns bald begegnenden Dictionaries auch Funktionen als Werte haben und auf diese Weise bestimmte Anwendungen abdecken. Wer aber an einem switch-ähnlichen Konstrukt für Python hängt, kann sich dieses Rezept[6] ansehen – was dort passiert, könnt ihr allerdings erst später verstehen.

Problems

(11.1)* Überlegt euch, wie ein Satz wie „Wenn die Sonne scheint, hole ich meine Strandmatte, und wenn auf der Neckarwiese Platz ist, lege ich mich dorthin, sonst fahre ich nach Ilvesheim“ interpretiert werden kann und schreibt die zugehörigen „Programme“ nach dem Vorbild im Skript.

(11.2)* Definiert analog zur Funktion printRelation eine, die je nach Vorzeichen einer Zahl Auskunft gibt, ob sie positiv, negativ oder Null ist. Definiert diese Funktion auch mal so, dass sie die Auskunft nicht direkt ausgibt, sondern in einem String an die aufrufende Funktion zurückgibt.

(11.3)* Lasst euch „Wahrheitstafeln“ der logischen Operatoren and, or und not ausgeben, etwa per

```
print 0, 0, 0 and 0
print 1, 0, 1 and 0
```

und so fort. Wie man solch lästigen Kram wie die dauernde Tipperei dem Rechner überlässt, sehen wir auf der nächsten Folie.

[6] `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/410692`

12. Sequenzen und Iteration

Sequenz
Datenstruktur
Index
Liste
mutable
immutable

Eine *Sequenz* ist eine *Datenstruktur*, in der mehrere Dinge stehen, wobei jede einen *Index* hat, der von 0 bis zur (Länge der Sequenz)−1 geht. Strings sind Sequenzen.

Über eine Sequenz kann man *iterieren*, d.h. für jedes Ding in der Sequenz eine Aktion machen:

```
>>> count = 0                    0       M
>>> for ch in "Markus":          1       a
...     print count, "\t", ch     2       r
...     count = count+1           3       k
...                               4       u
                                  5       s
```

Python hat von C geerbt, dass die Indizes mit 0 anfangen. Falle: Markus hat sechs Buchstaben, der letzte Index ist aber 5.

Man kann auch direkt auf Teile einer Sequenz zugreifen:

```
>>> "Markus"[2]     >>> "Markus"[-1]
'r'                 's'
```

Negative Indizes werden von Python als „von hinten gezählt“ interpretiert.

Ein zweiter wichtiger Sequenztyp in Python ist die *Liste*. In Strings stehen nur Zeichen, in Listen können beliebige Daten stehen. Außerdem können in Listen die einzelnen Elemente geändert werden, in Strings nicht. Im Jargon: Listen sind *mutable*, Strings sind *immutable*.

Diese Unterscheidung ist recht fundamental. Dass sie gemacht wird, hat vor allem Gründe in der Implementation – es ist für die AutorInnen des Python-Laufzeitsystems viel einfacher, mit Daten umzugehen, die sich nicht ändern können. Auch aus Sicht eines/r Python-ProgrammiererIn sind unveränderbare Werte unkritischer, wie ihr spätestens sehen werdet, wenn wir das Aliasing diskutieren werden.

Listen werden in eckigen Klammern notiert:

```
>>> eineListe = ["eins", "zwei", 3]
>>> eineListe[1]
'zwei'
>>> eineListe[1] = 2
>>> eineListe[1]
2
```

Listen mit Zahlen von 0 bis zu einer Grenze-1 kann man mit der eingebauten Funktion `range` erzeugen:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]   #
>>> for i in range(10):          ##
...     print "#"*i              ###
...                              ####
                                 #####
                                 ######
                                 #######
                                 ########
                                 #########
```

Mit `for ... in range ...` kann man die Standardanwendungen des `for`-Statements in Sprachen wie C, Pascal oder Basic nachbilden. Meistens kommen dabei aber „falsche Freunde“ raus, Programme, die unpythonesk, unübersichtlich und ungeschickt sind. Wenn ihr meint, sowas zu brauchen, denkt nochmal nach.

Der dritte eingebaute Sequenztyp von Python sind Tupel. Sie verhalten sich wie Listen, indem sie alle möglichen Daten enthalten können (und nicht nur Zeichen), aber wie Strings, indem sie nicht veränderbar sind. Sie werden in runden Klammern notiert:

```
>>> a = (5, "haha")
>>> a[1]
```

```
'haha'
>>> for x in a:
...     print x
...
5
haha
```

Ihre genauere Untersuchung verschieben wir auf später.

Problems

(12.1)* Warum gibt der folgende Code einen IndexError?

```
toy = "Teddy"
length = len(toy)
number = toy[length]
```

(12.2)* Löst die Wahrheitstafelaufgabe von der letzten Folie noch einmal mit Schleifen. **(L)**

(12.3)* Wandelt das Beispielprogramm mit `for ch in "Markus"` so um, dass es eine Funktion wird, die einen String nimmt. Zusätzlich soll jetzt nicht über den String iteriert werden, sondern über die *Indizes* im String. Alle Indizes in einem String s liefert z.B. range(len(s)) (Warum?). Vergleicht die Programme. Könnt ihr euch schon denken, warum ich von Schleifen über range abrate? **(L)**

(12.4)* Warum hat Python keine Probleme mit

```
>>> l = ["eins", 2, 3, "leipzig"]
>>> l[2] = "gobbel"
```

wirft aber bei

```
>>> s = "eins, 2, 3, leipzig"
>>> s[2] = "g"
```

einen Fehler? **(L)**

13. Iteration II

Python kann nicht nur über Sequenzen iterieren, sondern auch nach logischen Bedingungen: `while`. Zu dieser Sorte Schleifen später mehr.

break und continue

Nicht immer ist der normale Kontrollfluss in einer Schleife bequem. Wenn in einer Schleife ein „Notfall“ auftritt, wollen wir sie eventuell vorzeitig beenden: `break`.

```
>>> sum = 0
>>> for i in range(10):
...   sum = sum+i
...   if sum>20:
...     break
...
>>> sum, i
(21, 6)
```

Die Schleife ist also nicht ganz durchgelaufen (sonst wäre `i` gleich dem letzten Element der Sequenz, über die wir iterieren, hier also 9), sie wurde abgebrochen, als `sum` größer als 20 wurde.

Analog kann es vorkommen, dass ein Teil des Schleifenkörpers unter bestimmten Bedingungen nicht ausgeführt, die Iteration aber fortgesetzt werden soll: `continue`.

```
>>> for val in [1, 2, "drei", 4]:
...   if type(val)==type(""):
...     continue
...   print val,
```

```
...
1 2 4
```

Treffen wir bei der Bearbeitung unserer Sequenz also auf einen String, tun wir (hier) nichts und machen einfach beim nächsten Element der Sequenz weiter.

Häufig wird sowas eleganter durch direkte Selektion zu machen sein („Wenn `x` ein String ist, dann mache. . . “), es gibt aber durchaus Fälle, in denen ein `continue` die Intention klarer macht – es sagt so etwas wie „Oh, für diesen Fall bin ich nicht zuständig“ oder „Oh, ausnahmsweise soll dieses Element gar nicht behandelt werden“. Aussagen dieser Art sind bei längeren Selektionsausdrücken nicht mehr unbedingt klar.

Die Sprachelemente `break` und `continue` können ein Programm leicht zu unverständlichen Monstren machen. Es gibt Stilpäpste, die ihren Einsatz komplett ablehnen (strikt notwendig sind sie offenbar nicht – könnt ihr euch Ersatzkonstrukte überlegen?). Meine Meinung ist, dass es genug Beispiele gibt, in denen speziell `break` die Abläufe deutlich klarer macht. Trotzdem solltet ihr euch vor einem Einsatz dieser „Ersatz-gotos“ immer fragen, ob es nicht anders auch geht.

(Weiterführend:) Die Prüfung, ob etwas ein String ist, sollte man übrigens nicht mehr wie oben gezeigt machen, sondern über `isinstance(val, basestring)`. Allerdings ist der Hintergrund dabei etwas komplexer (für später: wir prüfen auf die gemeinsame Basisklasse von Bytestrings und Unicode-Strings), so dass ich es hier beim Rezept bewenden lassen will.

else an Schleifen

Schleifen, die irgendetwas suchen und, wenn sie Erfolg haben, mit `break` oder `return` verlassen werden, können ein sinnvolles „Default-Verhalten“ bekommen, indem man einen `else`-Zweig anfügt. Eine Funktion, die das erste Wort mit „en“ am Ende ausgibt und wenigstens irgendwas sagt, wenn es nichts dergleichen gibt (stellt euch vor, ihr wolltet einen ganz schlechten und dummen Infinitivfinder schreiben):

```
def printFirstInfinitive(wordSeq):
  for word in wordSeq:
    if word.endswith("en"):
      print word
      break
  else:
    print "Keine Infinitive"
```

Das sieht dann so aus:

```
>>> printFirstInfinitive(
  ["Gehen", "laufen", "springen"])
Gehen
>>> printFirstInfinitive(
  ["rechts", "links", "mitte"])
Keine Infinitive
```

Generell sollte man sich bei jeder Schleife dieser Art fragen, was passiert, wenn die Selektionsbedingung nie wahr wird. Wenn es dabei zu einem Fehler kommen kann, ist eine `else`-Klausel vermutlich empfehlenswert, um für irgendeine Sorte von Notfallverhalten zu sorgen. In diesem Beispiel sollte das allerdings eher eine ordentliche Fehlermeldung oder – vermutlich besser – das Auslösen einer Exception sein. Wie sowas geht, werden wir noch sehen.

Problems

(13.1)* Schreibt die Funktion printFirstInfinitive so um, dass sie den eventuell gefundenen Infinitiv nicht druckt, sondern zurückgibt. Wird kein Infinitiv gefunden, soll der spezielle Wert None zurückgegeben werden. **(L)**

14. Strings und Listen

Strings und Listen sind unsere Brot- und Butterdatenstrukturen. Wir wollen sehen, wie sie ineinander verwandelbar sind.

Zunächst gibt es die Funktion `str`, die aus so ziemlich allem einem String macht:

```
>>> str([3, "bla", ["liste"]])
"[3, 'bla', ['liste']]"
```

Das ist aber nicht immer das, was wir wollen. Wenn wir z.B. einen Satz aus mehreren Wörtern in einer Liste vorbereitet haben, wollen wir die einzelnen Wörter mit Leerzeichen verbunden ausgeben. Dazu gibt es die join-Methode von Strings, die als Argument eine andere Sequenz nimmt:

```
>>> l = ["Dies", "ist", "ein", "Satz"]
>>> " ".join(l)
'Dies ist ein Satz'
>>> print "\n".join(["Zeile1", "Zeile2"])
Zeile1
Zeile2
```

Umgekehrt können wir aus einem String eine Liste machen. Die Umkehrung von join ist split; ohne Argument teilt split an irgendwelchen Leerzeichen, mit Argument an Vorkommen des entsprechenden Strings:

```
>>> "Dies ist ein Satz".split()
['Dies', 'ist', 'ein', 'Satz']
>>> "/usr/src/Python-2.3".split("/")
['', 'usr', 'src', 'Python-2.3']
>>> "f1<sep/>f2<sep/>f3".split("<sep/>")
['f1', 'f2', 'f3']
```

Analog zu `str` gibt es eine Funktion `list`, die aus ihrem Argument eine Liste macht (das geht natürlich nur, wenn es eine Sequenz oder etwas ähnliches ist):

```
>>> list("Wort")
['W', 'o', 'r', 't']
```

Dabei werden die einzelnen Elemente der Sequenz die Elemente der Liste. Wenn man wollte, könnte man so Strings elementweise manipulieren:

```
>>> l = list("Wort")
>>> l[0] = "T"
>>> "".join(l)
'Tort'
```

Allerdings gibt es für Manipulationen dieser Art mächtigere und schnellere Werkzeuge in Python.

Problems

(14.1)* Was müsstet ihr tun, wenn ihr die Unix-Pfadtrenner (nämlich die Schrägstriche) etwa im Beispiel `"/usr/src/Python-2.3"` durch DOS-Pfadtrenner (nämlich Backslashes) ersetzen wollt und die replace-Methode für Strings bei eurem Python kaputt gegangen wäre? (Tipp: split und join kombinieren, den Backslash kriegt ihr mit `"\\"`). **(L)**

15. Operationen und Methoden für Sequenzen

Slice

Slices

Ein *Slice* ist ein Ausschnitt aus einer Sequenz:

```
>>> s = "0123455"
>>> s[1:3]
'12'
```

Es werden also die Elemente vom ersten Index einschließlich bis zum zweiten Index *ausschließlich* in eine neue Sequenz des gleichen Typs übernommen.

Es gibt auch offene Slices:

```
>>> l = ["null", 1, [2], "drei"]
>>> l[-2:]
[[2], 'drei']
>>> l[:3]
['null', 1, [2]]
```

Bei ihnen wird alles vom ersten Index bis zum Ende der Sequenz in die neue Sequenz geschrieben, oder vom Anfang bis zum zweiten Index ausschließlich.

Man kann sich fragen, warum die Python-MacherInnen sich entschieden haben, das Element mit dem letzten Index nicht mitzunehmen. Es gibt zwei Möglichkeiten, das zu verstehen. Einerseits kann man nur so `seq[:4]` als „die ersten vier Elemente von seq“ lesen (wäre seq[4] im slice, wären es fünf Elemente – zählt nach!). Andererseits gilt nur mit dieser Definition `seq = seq[:splitPoint]+seq[splitPoint:]`.

Operationen auf Sequenzen

Man kann Elemente auf die Zugehörigkeit zu Sequenzen prüfen, die Länge, das Maximum oder Minimum einer Sequenz bestimmen oder Sequenzen verketten:

```
>>> s = "Ich könnte eine Liste sein"
>>> 'c' in s
1
>>> 'c' not in s
0
>>> s+", oder?"
'Ich k\xf6nnte eine Liste sein, oder?'
>>> len(s)
26
>>> min(s)
' '
>>> max(s)
'\xf6'
```

Warum hier `\xf6` statt eines ö ausgegeben wurde, werden wir später sehen – akzeptiert vorerst, dass Python sich im interaktiven Modus nicht traut, für das System „komische“ Zeichen einfach so auszugeben und es eine Repräsentation wählt, bei der nichts (oder wenig) schief gehen kann, dass aber das `\xf6` aus Python-Sicht eurem ö exakt entspricht.

Methoden von Sequenzen

Vererbung

Auch Sequenzen haben Methoden – natürlich einige weniger als Strings, die sozusagen ein „Spezialfall“ von Sequenzen sind:

```
>>> "aba".index("a")
0
>>> range(10).index(4)
4
>>> (range(10)*7).count(4)
7
```

Hier begegnen wir zum ersten Mal flüchtig dem Konzept der *Vererbung*: Strings können alles, was Sequenzen können (erben also alle Eigenschaften, die Sequenzen haben), können darüber hinaus aber noch erheblich mehr.

Methoden von mutable sequences

Veränderbare Sequenzen haben eine ganze Reihe von Methoden, um sie zu verändern, z.B. Anhängen, Einfügen, Umkehren, Sortieren:

```
>>> l = ["eine", "Liste"]
>>> l.append("lang");print l
['eine', 'Liste', 'lang']
>>> l.insert(1, "veränderte");print l
['eine', 'ver\xe4nderte', 'Liste', 'lang']
>>> l.reverse(); print l
['lang', 'Liste', 'ver\xe4nderte', 'eine']
>>> l.sort(); print l
['Liste', 'eine', 'lang', 'ver\xe4nderte']
```

Diese Operationen geben nichts zurück, wir haben die Liste jeweils explizit gedruckt. Grund dafür ist, dass die Liste „in place“ verändert wird, also keine neue Liste erzeugt wird.

Zu den Semikola: Man kann in Python mehrere Statements in eine Zeile schreiben und durch Semikola trennen, sollte das aber im Regelfall nicht tun.

Wieder „erben“ veränderbare Sequenzen von allgemeinen Sequenzen, erstere können also alles, was letztere können, und noch mehr.

Mehr Methoden von Sequenzen in der Python-Referenz[7].

Problems

(15.1)* Macht euch klar, was bei Strings und Listen jeweils Element der Sequenzen ist. Probiert etwa das folgende und erklärt, was jeweils passiert:

```
>>> "aba".index("a")
>>> ["aba"].index("a")
>>> ["a", "b", "a"].index("a")
>>> ["a", "b", "aba"].count("a")
```

Denkt euch andere Beispiele mit den hier besprochenen Methoden aus.

Eine Ausnahme hier ist der (bool'sche) in-Operator, der (etwa seit Python 2.0) auf Strings anders operiert als auf andere Sequenzen. Vergleicht

```
>>> ["a", "b"] in ["a", "b", "c"]
>>> "ab" in "abc"
```

und erklärt, inwieweit sich in hier anders verhält als die anderen Methoden.

(15.2)* Übersetzt die folgenden Beschreibungen in slices einer Liste l; dabei wollen wir, um zu große Konfusion zu vermeiden, wie Python zählen, d.h. eine Liste besteht aus dem *nullten*, ersten, zweiten, dritten usf. Element.

1. Das vierte und fünfte Element

[7] http://docs.cl.uni-heidelberg.de/python/lib/typesseq.html

2. Das dritte bis einschließlich siebte Element
3. Alle Elemente ab dem dritten
4. Alle Elemente bis einschließlich dem vorletzten
5. Alle Elemente bis auf das nullte
6. Alle Elemente bis auf das vierte
7. Alle Elemente bis auf das dritte, vierte und fünfte
8. Alle Elemente

(L)

(15.3) Im Unterschied zu den Methoden von Strings, die wir kennen, geben die die Methoden, die Listen verändern, nichts zurück. Warum könnten das die Python-Designer wohl so gemacht haben? **(L)**

(15.4) Wie könnte man die Umkehrung eines Strings bekommen, also zu abc die Umkehrung cab usf? Tipp: Für Listen ist die Umkehrung schon in Python eingebaut... **(L)**

template

Formatcode

16. Der Prozentoperator für Strings

Stellt euch vor, ihr müsstet eine Funktion schreiben, die folgendes tut:

```
>>> formatWithDigits(78)
'78 hat 2 Ziffern'
>>> formatWithDigits(4893)
'4893 hat 4 Ziffern'
>>> formatWithDigits(5)
'5 hat 1 Ziffer'
```

Die Funktion soll also einen String zurückgeben (und *nicht* ausgeben – wenn euch der Unterschied noch nicht klar ist, seht euch nochmal die Kapitel über Funktionen an), in dem das Argument in einer Art Schablone gemeinsam mit irgendwelchen Eigenschaften des Arguments steht. Auch wenn dieses Beispiel etwas bemüht wirken mag, tauchen derartige Aufgaben in der Praxis sehr häufig auf.

Wir könnten das Problem so lösen:

```
def formatWithDigits(anInt):
  return anInt+" hat "+len(str(anInt))+" Ziffern"
```

Das ist allerdings langsam und außerdem recht aufwändig. Abgesehen davon haben wir natürlich das Problem einstelliger Zahlen nicht gelöst (was bei negativen Zahlen passieren soll, ist dann noch eine andere Frage).

Eine Lösung mit dem Prozentoperator sieht so aus:

```
def formatWithDigits(anInt):
  return "%d hat %d Ziffern"%(anInt,
    len(str(anInt)))
```

An diesem einfachen Beispiel mag der wahre Nutzen noch nicht ganz erkennbar sein, aber diese Art, Strings zu bauen, ist praktisch immer dem Zusammenkonkatenieren vorzuziehen.

Der Prozentoperator verknüpft einen String, der als Schablone (*template*) dient, mit einem Tupel. In der Schablone sind durch %? Felder markiert, in die der Reihe nach die Werte aus dem Tupel eingetragen werden. Statt des ? steht in Wirklichkeit ein *Formatcode*, der bestimmt, wie der Wert dargestellt wird. Verbreitet sind s (String), d (Integer), f, g und e (Float) sowie % (ein Prozentzeichen).

Zwischen Prozent und Formatcode können noch diverse Optionen kommen, etwa Längen.

```
>>> "|%5d|%05d|%-5d|"%(2,2,2)
'|    2|00002|2    |'
>>> "|%1f|%3.9e|%+1.2g|"%(1.23,1.23,1.23)
'|1.230000|1.230000000e+00|+1.2|'
```

Die Details über all das stehen im Kapitel String Formatting Operations[8] der Python-Referenz – seht euch diese Seite *jetzt* an. Ihr müsst nicht auswendig wissen, wie was genau geht, aber ihr solltet wissen, was es alles gibt.

Was ihr mitnehmen solltet: Normalerweise sagt eine Zahl zwischen % und Formatcode etwas über die Breite des Feldes aus, und ein − vor der Zahl macht etwas links- statt rechtsbündig, was für Strings und Ints ziemlich analog läuft. Für Fließkommazahlen muss es natürlich ein paar mehr Merkmale geben.

Der Längencode darf auch ein Stern sein, dann wird die Länge aus einem weiteren Formatargument gelesen:

```
>>> for l in range(1,6):
...     print "%*d"%(l, l)
...
1
 2
  3
   4
    5
```

Problems

(16.1)* Der Prozentoperator ist euer Freund, er macht das Zusammenbauen von Strings in genau der beabsichtigten Weise zu einer wahren Freude (im Zweifel in Verbindung mit list comprehensions – die uns später begegnen werden – und der join-Methode von Strings). Überlegt euch für die folgenden Fragen jeweils einen Formatstring und probiert aus, ob auch das rauskommt, was rauskommen soll.

1. Ihr habt eine ganze Zahl i und einen String s. Ihr möchtet das im Format s: i haben, wobei s linksbündig in einem 30 Zeichen breiten Feld und i rechtsbündig in einem 5 Zeichen breiten Feld stehen soll.

2. Dito, nur soll s diesmal rechtsbündig sein

3. Dito, nur soll i diesmal mit führenden Nullen ausgegeben werden

4. Dito, nur soll i jetzt eine Fließkommazahl sein und mit zwei Vorkommastellen und drei Nachkommastellen ausgegeben werden (probiert aus, was passiert, wenn die Zahlen zu groß für das Feld werden)

(16.2) Verbessert unsere `formatWithDigits`-Funktion, so dass sie auch einstellige und negative Zahlen korrekt behandelt.

Vielleicht wollt ihr auch mal sehen, wie ihr das Problem mit string.join löst.

[8] `http://docs.cl.uni-heidelberg.de/python/lib/typesseq-strings.html`

17. Encodings

Encoding

ASCII

Computer speichern nur Zahlen, sie wissen zunächst gar nichts von Buchstaben. Um trotzdem Texte verarbeiten zu können, wurden Abbildungen von Zahlen auf Zeichen definiert, so genannte *Encoding*s.

Praktisch alle heutigen Rechner verwenden Abarten von *ASCII*. Das steht für American Standard Code for Information Interchange, was nur insoweit relevant ist, als „American“ drinsteht. Das ist als North American zu verstehen und erklärt, warum die darin erklärten Zeichen nur die sind, die fürs Englische wichtig sind – und warum Umlaute und andere verrückte Zeichen auch heute noch ein Problem für Rechner sind.

ASCII definiert z.B., dass 32 dem Blank entspricht, 33 dem Ausrufezeichen, 65 dem großen A, 97 dem kleinen usf. Insgesamt belegt ASCII die Zahlen zwischen 32 und 127 mit Zeichen. Tatsächlich sind auch die Zahlen zwischen 0 und 31 belegt, und zwar mit so genannten Kontrollzeichen. Darunter befinden sich die Zeilentrenner CR bei 13 und LF bei 10 (CR hat auf Fernschreibern einen „Wagenrücklauf“ gemacht, LF ist in die nächste Zeile gegangen), das oben erwähnte Tab bei 9, der Backspace bei 8 und ein Zeichen namens Escape bei 27. Dass ASCII bei 127 aufhört und nicht erst bei 255, wie es für heutige Rechner günstiger wäre, die in einer Speicherzelle 256 verschiedene Werte speichern können, hat historische Gründe.

Jenseits der ASCII-Zeichen gibt es eine Unzahl weiterer Encodings, die von verschiedenen Systemen benutzt wurden und werden:

- CP1250 (Consumer-Windows)
- iso-8859-1 (viele Daten aus dem Netz)
- CP437, CP850 (westeuropäisches DOS)
- Diverse Darstellungen von ISO10646 bzw. Unicode
- unglaublich viele weitere

Auch Python speichert „normale“ Strings als Sequenz von Zahlen. Die Funktion `ord` gibt euch die Zahl aus, die einem Zeichen entspricht, die Funktion `chr` macht aus einer Zahl ein Zeichen:

```
>>> for c in "(Abakus)": print ord(c),
...
40 65 98 97 107 117 115 41
>>> for i in range(48, 73): print chr(i),
...
0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H
```

Das geht auch für Nicht-ASCII. Auf meiner Maschine:

```
>>> ord("Ü")
220
>>> chr(220)
'\xdc'
```

Die Ausgabe `\xdc` entsteht, weil Python nicht weiß, welches Encoding das Programm verwendet, das die Zeichen wirklich auf den Bildschirm malt. Deshalb gibt es nur eine ASCII-Darstellung des Zeichens aus. Diese Darstellung ist eingestandenermaßen nicht sehr lesbar, und ich werde erst in Programmieren II erklären, was sie soll. Schickt ihr den entsprechenden String durch print, wird übrigens ein Ü ausgegeben – wenn denn das 220 in euerer augenblicklichen Konfiguration tatsächlich dem Ü entspricht.

Darüber entscheidet meist ein so genannter Terminalemulator. Um das noch komplizierter zu machen, ist bei diesen häufig konfigurierbar, welche Encodings sie verwenden (manchmal auch nur implizit, indem man ihnen entsprechend präparierte Fonts unterschiebt). Unter verschiedenen Windows-Versionen verwendet die grafische Oberfläche ein anderes Encoding als die „Eingabeaufforderung“ – kurz: it's a mess. Insbesondere lohnt es sich nur in Ausnahmefällen, Python zu erzählen, welches Encoding euer Terminalemulator verwendet (was durchaus möglich ist, vgl. Wiki).

Problems

Unicode-String

Bytestring

(17.1) Schreibt ein Programm, das euch eine „ASCII-Tabelle“ ausgibt, das also eine Ausgabe wie

```
32 -> ' '
33 -> '!'
...
```

produziert. Ihr solltet davon absehen, euch die Zeichen unterhalb 32 ausgeben zu lassen. **(L)**

18. Unicode

Programme bekommen Daten in allen möglichen Encodings, z.B. aus dem Web oder in Koropra aus verschiedenen Quellen. Python kennt die üblichen Encodings und kann sie interpretieren, und zwar mit der `decode`-Methode von Strings. Hier habe ich einen String im Encoding CP473:

```
>>> origStr = 'fr\x84\xe1e'
>>> uStr = origStr.decode("cp437")
>>> print uStr
fräße
>>> uStr
u'fr\xe4\xdfe'
```

Was herauskommt, wenn ich `origStr` ausgebe, hängt von eurem System ab – bei mir ist es ziemlicher Quatsch, weswegen ich hier ein `print origStr` gelassen habe. Probiert es bei euch.

Ergebnis der `decode`-Methode ist ein *Unicode-String*. Unsere bisherigen Strings werden im Gegensatz dazu häufig als *Bytestring*s bezeichnet. Dass es ein Unicode-String ist, sieht man an dem „u“ vor dem Anführungszeichen. Toll an Unicode-Strings ist, dass ihre Interpretation über alle Systeme hinweg identisch ist – in einem Unicode-String bedeutet 228 *immer* ein ä, während 951 immer ein griechisches η ist. Insbesondere sind die Codes von Zeichen in Unicode-Strings nicht wie bei Bytestrings auf den Bereich 0 bis 255 beschränkt.

Welche Zahl welchem Zeichen entspricht, könnt ihr den Unicode Code Charts[9] entnehmen. Bevor ihr anfangt, eurem Rechner Tagalog-Zeichen entlocken zu wollen: Nur weil er Unicode *verarbeiten* kann, muss er noch lange nicht alle Unicode-Zeichen auch *anzeigen* können...

Bei der Ein- und Ausgabe müssen Unicode-Strings fast immer kodiert werden. Dazu dient die `encode`-Methode:

```
>>> uStr.encode("iso-8859-1")
'fr\xe4\xdfe'
>>> uStr.encode("utf-8")
'fr\xc3\xa4\xc3\x9fe'
```

UTF-8 ist dabei ein Encoding für Unicode selbst. Dass man hier kodieren muss, hat vor allem historische Gründe – traditionell waren Zeichen im Rechner „klein“, repräsentiert durch ein „Byte“ (eine Gruppe von acht bit) oder manchmal auch durch etwas größere oder kleinere Einheiten, jedenfalls durch *eine* Speicherzelle. Es gibt aber im Augenblick rund 100000 Unicode-Zeichen, und so viele verschiedene Werte lassen sich nicht in einer Speicherzelle repräsentieren.

Nun hätte man sagen können: Was solls? Warum soll man die Computer nicht so machen, dass in ihre Speicherzellen 100000 verschiedene Werte passen. So funktioniert aber die Welt nicht – selbst wenn jemand in der Lage wäre, die alten, byteorientierten Architekturen zu ersetzen, möchte niemand die Terabytes von byteorientierten Daten wegwerfen, schon gar nicht, wenn der Leidensdruck nur außerhalb des „Westens“ erheblich ist. Abgesehen davon ist das das Internet weitgehend byteorientiert. Also: „Grundeinheit“ von Rechnerspeichern, ob Platte oder Hauptspeicher, bleibt das Byte.

Nun kann man einfach ein paar Byte zusammenschnüren und so ausreichend viele Werte darstellen; in der Anfangszeit von Unicode war die Hoffnung, 65536 Zeichen seien genug, und so

[9] `http://www.unicode.org/charts/`

wollte man für die Zeichen einfach zwei Byte pro Zeichen verwenden. Es hat sich aber gezeigt, dass man doch mehr Platz haben will und auch, dass die Verwendung von zwei Bytes pro Zeichen gerade mit bestehenden Anwendungen recht mühsam ist (mal ganz zu schweigen von den Schwierigkeiten, sich zu einigen, ob das „höherwertige“ Byte vorne oder hinten steht).

Codec

Wenn man nun ohnehin schon raffiniertere Möglichkeiten ersinnt, Unicode in Bytestrings darzustellen, kann man ja für „Abwärtskompatibilität“ sorgen – das heißt allgemein, dass ein neues Verfahren mit einem alten Verfahren in irgendeinem Sinn verträglich ist. UTF-8 ist dazu so angelegt, dass alle legalen ASCII-Strings auch legales UTF-8 sind, dass also Probleme erst dann auftauchen, wenn man nationale Sonderzeichen (dazu zählen etwa die deutschen Umlaute) verwendet.

Wenn die alten US-Zeichen nur jeweils ein Byte verwenden, ist klar, dass andere Zeichen mehr, unter Umständen deutlich mehr, als zwei Bytes brauchen. Im Beispiel oben wurden ü und ß zu zwei Zeichen; die (nicht gezeigte) Hiragana Iteration Mark auf Codeposition 12445 wird zur drei Zeichen, während die altgriechische Notation für die Note e auf Codeposition 119296 zu vier Zeichen wird.

Fehlerbehandlung

Bei En- und Dekodieren kann einiges schiefgehen:

```
>>> uStr.encode("ascii")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode
characters in position 2-3: ordinal not in
range(128)
>>> "ärger".decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeDecodeError: 'utf8' codec can't decode
bytes in position 0-2: invalid data
```

Im ersten Fall hat sich der ASCII-Codec (*Codec* ist dabei kurz für Kodierer/Dekodierer, also das Programmstück, das tatsächlich die Übersetzungen besorgt) beschwert, dass er das „ä“ aus dem fräße nicht darstellen kann – in ASCII ist halt kein „ä“ enthalten, ebensowenig wie, meinetwegen, altgriechische Musiknotation im üblichen iso-8859-1.

Der zweite Fall ist subtiler: UTF-8 lässt nur bestimmte Folgen von Bytes zu – so muss z.B. nach irgendeinem Nicht-ASCII-Zeichen (in diesem Fall das „ä“) mindestens ein weiteres kommen, was hier nicht der Fall ist. Der String ist also nicht in UTF-8 kodiert, sondern in etwas anderem.

In der Regel braucht man in solchen Fällen eine Fehleranalyse und -behebung. Wenn man aber in einer Situation ist, in der das nicht machbar oder wünschenswert ist– beispielsweise will man dringend *irgendwas* anzeigen oder auf Fehler in anderen Programmen (z.B. Outlook Express) nicht mit Absturz reagieren –, kann man `encode` zusätzlich „ignore“ oder „replace“ übergeben:

```
>>> uStr.encode("ascii", "ignore")
'fre'
>>> uStr.encode("ascii", "replace")
'fr??e'
```

Es gibt noch ein paar andere Möglichkeiten zur Fehlerbehandlung – wenn ihr sie braucht, findet ihr in der Dokumentation zum Codecs-Modul[10] etwas genauere Informationen.

(Weiterführend:) Dort befindet sich auch eine Liste der standardmäßig unterstützten Codecs, und amusanterweise sind darunter auch auch encoder für zlib (das ist die Kompression, die gzip verwendet) oder base64 (so werden binäre Attachments kodiert). Auf diese Weise kann man sehr einfach packen:

```
>>> s = open("/usr/bin/python").read()
```

[10] `http://docs.cl.uni-heidelberg.de/python/lib/module-codecs.html`

```
>>> len(s)
2179185
>>> len(s.encode("zlib"))
699045
```

Literal

(hier haben wir eine Datei gelesen – was das bedeutet, sehen wir erst später) oder E-Mail-Attachments ein- und auspacken:

```
>>> "Bla".encode("base64")
'Qmxh\n'
```

Problems

(18.1) Besorgt euch den „nackten“ Text eines base64-enkodieren Attachments (dazu könnt ihr z.B. in euren Mailspool gucken und den String mit einem Texteditor herauspröbeln). Schickt ihn durch `decode("base64")`.

19. String-Literale

Ein *Literal* ist die Repräsentation eines Wertes im Programm-Quellcode. So steht 138 in Python beispielsweise für die Zahl 138 – speichert der Rechner diese Zahl, stehen im Speicher nicht die Ziffern 1,3 und 8, sondern eher etwas wie 10001010 – wobei auch das nur eine Repräsentation für ein paar Sammlungen von Elektronen ist. Während nun 138 das Literal für diese Zahl (als abstraktes Konzept) in Python ist, könnten andere Sprachen da andere Vereinbarungen treffen und etwa 8a für unsere 138 verwenden, und auch Python erlaubt, die 138 im Programmtext also 0x8a oder auch `str("138")` darzustellen. Die interne Repräsentation ist jedoch in allen Fällen gleich (also in diesem Fall 10001010).

Strings können in Python-Quellcode auf verschiedene Arten dargestellt werden:

1. in einfachen (') und doppelten (") Anführungszeichen
2. in dreifachen doppelten (""") oder einfachen (''') Anführungszeichen

Dabei gehen die Möglichkeiten bei (1) nur für einzeilige Strings, während (2) Strings zulassen, die über mehrere Zeilen gehen.

Das kann kombiniert werden mit zwei Modifikatoren, die jeweils vor die öffnenden Anführungszeichen geschrieben werden:

- r für raw strings
- u für unicode strings

Dabei werden in raw strings keine Escapes ausgewertet, während Unicode-Literale zu Unicode-Strings führen.

Die raw strings werden vor allem für reguläre Ausdrücke nützlich werden, während ihr Unicode-Literale vor allem braucht, um Unicode-Strings so zu manipulieren, dass sie auch Unicode-Strings bleiben.

Escape-Sequenzen

Wenn man in einem String Zeichen haben will, die z.B. für den Editor oder Python eine spezielle Bedeutung haben (Backspace, Return, Quotes. . .) oder die man über die Tastatur nicht eingeben kann, kann man Escape-Sequenzen verwenden.

Escape-Sequenzen haben weder mit Sequenzen im Python-Sinn noch mit der Escape-Taste viel zu tun (und schon gar nichts mit Flucht, auch wenn manche Leute deutsche Übersetzungen in der Art von „Fluchtsymbol-Darstellung“ machen). In ihnen steht einfach ein Backslash vor einem oder mehreren anderen Zeichen. Python ersetzt dann, wenn es (normalerweise zur Compilezeit) das Stringliteral auswertet, die Zeichen der Escape-Sequenz durch *ein* anderes Zeichen.

Gebräuchliche Escape-Sequenzen:

```
\n   Zeilenvorschub
\r   Wagenrücklauf
\"   Ein doppeltes Anführungszeichen
\b   Backspace
\t   Tab
\\   Backslash
```

Wörter wie Zeilenvorschub oder Wagenrücklauf (line feed bzw. carriage return) kommen offensichtlich aus der Zeit der Fernschreiber. Ihre Wirkung ist aber auch heute noch mehr oder minder so wie damals: Ein Zeilenvorschub lässt den Cursor in die nächste Zeile gehen, ein Wagenrücklauf bewegt ihn zurück an den Anfang der Zeile. Wenn ihr auf einem nicht ganz dämlichen System seid, werdet ihr allerdings feststellen, dass ein einfacher Zeilenvorschub den Cursor auf den *Anfang* der nächsten Zeile setzt. Das liegt an interner Magie des Betriebssystems, das nach der Ausgabe eines Zeilenvorschub einen Wagenrücklauf „reinzaubert“, wenn es das Ziel der Ausgabe für einen Textbildschirm hält. Bei Druckern wirkt diese Magie nicht, und deshalb findet man dort manchmal interessante Treppeneffekte.

Der Backspace lässt den Cursor um einen Schritt zurückgehen. Wenn euer System einen vernünftigen Terminalemulator hat, sollte etwa folgendes passieren:

```
>>> print "Welt\bd\rG"
Geld
```

In Summe:

```
>>> print 'bla', "romp", """Ein
...   ziemlich
... langer String"""
bla romp Ein
  ziemlich
langer String
>>> print "ein\nEscape", r"kein\nEscape"
ein
Escape kein\nEscape
```

Encoding des Quelltexts

Seit Python 2.3 unterstützt Python auch offiziell Quelltexte in verschiedenen Encodings. Ihr könnt auf diese Weise in euren Quelltext Unicode-Stringliterale einbetten, etwa

```
someStr = u"Nüfbö"
```

Wüsste Python nicht um das Encoding des Quelltextes, könnte es die Umlaute nicht in einen Unicode-String übersetzen, denn dort kann nicht einfach irgendeine Zahl stehen, es muss etwas wie „hier steht ein ü“ gespeichert sein. Das ü, das der Editor auf eurem System anzeigt, könnte aber woanders ein ø oder irgendein anderes wildes Zeichen sein, und Python kann nicht wissen, was für einen Editor ihr verwendet habt (bzw. welches Encoding euer Editor) – es sei denn, ihr würdet es ihm sagen.

Deshalb kann man das Encoding deklarieren. Die Art, in der Python das macht, ist nicht unumstritten – man schreibt nämlich

```
# -*- coding: iso-8859-1 -*-
```

in die *erste oder zweite* Zeile des Skripts. Umstritten ist das, weil auf diese Weise ein Kommentar die Semantik des Programms ändert, während eigentlich ein Kommentar doch definiert ist dadurch, dass er an der Semantik nichts ändert. Dass dem hier doch so ist, liegt am Sieg der PragmatikerInnen unter den Menschen, die Python weiterentwickeln, und daran, dass der verbeitetete Editor emacs diese Angabe automatisch versteht.

Kehrseite der an sich guten Einrichtung, das Encoding des Quelltexts definieren zu können ist, dass Python jetzt Warnungen ausgibt, wenn ihr Umlaute in String-Literalen verwendet (und vermutlich irgendwann einen Fehler). Das bedeutet im Effekt, dass eine coding-Angabe jetzt Pflicht ist, sofern euer Skript sich nicht auf reines ASCII beschänkt.

Das iso-8859-1 oben ist natürlich nur ein Beispiel. Ihr kommt nicht drumrum, herauszufinden, welches Encoding euer Editor verwendet. Auf unseren Poolmaschinen stimmt das iso-8859-1, es sei denn, ihr habt etwas anderes bestellt, unter neueren Windows-Versionen, unter BeOS, Mac OS X oder auf geeignet manipulierten Linux-Systemen *kann* (muss aber nicht) es sein, dass ihr schon utf-8 angeben müsst.

Problems

(19.1) Um zu verstehen, wie das mit den Escape-Sequenzen geht, sieht man sich am besten die Zeichen an, die nachher in den Strings stehen, und zwar genauer deren Codes, d.h. die Zahlen, die der Rechner intern zur Repräsentation der Zeichen verwendet. An diese Zahlen kommt ihr über die eingebaute Funktion ord heran. Schreibt also einfach mal verschiedene Strings in ein Programm der folgenden Art und versucht zu verstehen, was passiert. Wie sind die Längen der Strings, die ihr probiert?

```
for ch in "a\tb\r\n\\":
  print  repr(ch), ord(ch)
```

Anmerkung: Die Funktion `repr` gibt die so genannte Repräsentation ihres Arguments zurück. Bei Strings bedeutet das, dass nicht druckbare Zeichen wieder als Escape-Sequenzen geschrieben werden. Probiert aus, was passiert, wenn ihr das repr weglasst.

(19.2) Die split-Methode von Strings trennt ohne weitere Argumente an „irgendwelchen“ Leerzeichen trennen. In der Tat sind für split auch Tabs, Zeilenvorschübe und Wagenrückläufe Leerzeichen (genauer: white space). Seht euch das folgende Beispiel an:

```
>>> s = "Some\tcrazy\nstring\rstuff here"
>>> s
>>> print s
>>> s.split()
>>> s.split(" ")
```

(19.3) Die Funktion `eval` nimmt einen String, in dem ein Python-Ausdruck steht und tut das, was der Interpreter täte, wenn er diesen String sehen würde:

```
>>> eval("5+6")
11
```

Insbesondere wertet er innerhalb des Strings auch String-Literale aus. Damit kann man lustige Spiele treiben. Seht euch die folgenden Ausdrücke an und überlegt euch, was wohl ihr Wert sein wird. Seht nach, ob ihr recht hattet. Es ist natürlich möglich, dass da auch mal SyntaxErrors kommen...

```
eval("'ab'")
eval('\'ab\'')
eval('\'ab\t\'')
eval(r"'ab\n'")
eval("r'ab\n'")
eval(eval("'\"ab\"'"))
```

Schafft ihr es, das letzte Beispiel um noch ein `eval` weiterzuspinnen? **(L)**

20. Dictionaries

Mapping
dictionary
hash
associative array

In Sequenzen werden die Elemente immer durch einen ganzzahligen Index angesprochen. In *Mapping*s (Abbildungen) kann der Index fast beliebig sein. Das Standard-Mapping in Python ist das *dictionary* (in anderen Sprachen heißt sowas auch *hash* oder *associative array*). Dictionaries werden in geschweiften Klammern notiert, auf die Elemente wird wieder mit eckigen Klammern zugegriffen.

```
>>> d = {"eins": '1', "zwei": '2'}
>>> d["plus"] = '+'
>>> d
{'eins': '1', 'plus': '+', 'zwei': '2'}
```

Offenbar sind Dictionaries mutable – schon in diesem Beispiel verändern wir d, wovon ihr euch überzeugen könnt, wenn ihr euch zwischen der ersten und der zweiten Zeile d ausgeben lasst und euch klar macht, dass dem d selbst zwischendurch nicht zugewiesen wird.

Eine Anwendung bietet sich an: Wir wollen den Wert von Sätzen wie „eins plus eins“ ausrechnen.

```
def compute(expr, langDict):
  """computes a pseudo-natural-language expression.

  The expression is passed in the string expr,
  langDict provides a mapping of single words to
  python values and operators.
  """
  pythonExpression = []
  for w in expr.split():
    pythonExpression.append(langDict[w])
  return eval("".join(pythonExpression))
```

Der komische String, der unter dem Funktionskopf steht, heißt Docstring. Wir werden in Kürze sehen, was es damit auf sich hat.

Die hier verwendete eingebaute Funktion `eval` gibt übrigens den Wert eines Python-Ausdrucks zurück:

```
>>> eval("5*4+10")
30
```

Ein Test mit `d` von oben:

```
>>> compute("eins plus zwei", d)
3
```

Es ist bei weitem nicht selbstverständlich, dass eine Sprache eine Funktion wie `eval` hat, die so einfach Ausdrücke aus dieser Sprache bewertet – C beispielsweise hat so ohne weiteres keine Einrichtung dieser Art.

Die Einfachheit, mit der sowas in Python geht, ist ein Nebeneffekt des Umstands, dass Python (mehr oder weniger) interpretiert wird. Auf der anderen Seite sollte man diesen Umstand auch nicht überbewerten – jede Programmiersprache, die mächtig genug ist (und das sind sie alle), erlaubt, dass eine Funktion wie eval in ihr geschrieben wird.

Sehr praktisch ist auch die Kombination von Dictionaries mit dem Prozentoperator für Strings. Wenn nämlich der zweite Operand ein Dictionary ist, muss zwischen Prozent und Formatcode in der Schablone in Klammern ein Schlüssel aus diesem Dictionary stehen. Folgendes sollte illustrieren, was da passiert:

```
>>> "%(prog)s ist %(num)d mal %(prop)s"%{
...     "prop": "toll",
...     "prog": "python",
...     "num": 7,}
'python ist 7 mal toll'
```

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(20.1)* Wenn ihr „Formulare“ ausfüllen müsst, ist der Prozentoperator mit einem Dictionary als zweitem Argument das Mittel der Wahl. Stellt euch vor, ihr bekommt die Ergebnisse einer Mensa-Umfrage in einer Liste von Dictionaries, von denen jedes etwa so aussieht:

```
d = {"kat": "Die Preise", "anteil": 95.7, "urteil": "zu hoch"}
```

Ihr sollt jetzt einen Formatstring angeben, so dass `fmtStr%d` etwas wie „Die Preise der Mensa wurde von 95.7% der Befragten als zu hoch eingeschätzt“ ausgibt. **(L)**

(20.2)* Macht ein Skript aus der Funktion compute (lest den zu berechnenden Ausdruck per `raw_input`) und erweitert das langDict, so dass ein paar mehr Operatoren und Zahlen verstanden werden.

(20.3) Erweitert das Skript aus der letzten Aufgabe um eine weitere Funktion, die etwa für die Zahlen zwischen Null und Zwölf die Ergebnisse als Wort ausgibt (das könnt ihr über eine Liste oder ein Dictionary machen – was sind die Vor- und Nachteile dieser beiden Ansätze?). Was passiert, wenn eine Zahl ausgegeben werden soll, für die ihr kein Wort eingegeben habt? **(L)**

(20.4) Nicht alles kann Schlüssel in einem Dictionary sein. Im Groben taugt nur, was immutable ist. Denkt euch aus, wie ihr euch von dieser Behauptung überzeugen könnt. **(L)**

(20.5) Werte in Dictionaries können tatsächlich alles sein. Was wird etwa folgendes Skript ausgeben?

```
def f1(num):
  print num+1

def f2(num):
  print num -1

{'a':f1, 'b':f2}['a'](1)
```

(L)

21. Methoden von Dictionaries

Häufig nützlich sind Listen der Schlüssel oder Werte eines Dictionaries.

```
>>> d.values()
['1', '+', '2']
>>> d.keys()
['eins', 'plus', 'zwei']
>>> d.items()
[('eins', '1'), ('plus', '+'), ('zwei', '2')]
```

Die von `d.items()` zurückgegebenen Paare sind Tupel.

Die Methode `has_key` prüft, ob ein Schlüssel vorhanden ist:

```
>>> if d.has_key('minus'):
...   m = d['minus']
... else:
...   m = None
>>> print m
None
```

Solche Konstruktionen kommen häufig vor, daher die Abkürzung

```
>>> m = d.get('minus', None)
>>> m = d.setdefault('minus', None)
```

Im zweiten Fall wird `dict['m']` als Seiteneffekt der Wert `None` zugewiesen.

Praktisch ist setdefault besonders, wenn man als Werte irgendwelche veränderbaren Datenstrukturen hat. Will man etwa einfach eine leere Liste anlegen, wenn ein Schlüssel noch nicht vorhanden ist, ansonsten an die bestehende Liste anhängen, ginge das so:

```
d.setdefault(key, []).append(whatever)
```

was kürzer und auch nicht unverständlicher ist als das äquivalente

```
if d.has_key(key):
  d[key].append(whatever)
else:
  d[key] = [whatever]
```

Dictionaries lassen sich kopieren:

```
>>> d2 = d.copy()
>>> d2["drei"] = "3"
>>> d2
{'drei': '3', 'eins': '1', 'plus': '+',
'zwei': '2'}
>>> d
{'eins': '1', 'plus': '+', 'zwei': '2'}
```

Zuweisung ohne Kopieren würde wieder zu zwei Referenzen auf dasselbe Dictionary führen:

```
>>> d1 = d
>>> d1["drei"] = "3"
>>> d
{'drei': '3', 'eins': '1', 'plus': '+',
'zwei': '2'}
```

Manchmal nützlich ist `update`, das die Werte aus einem Dictionary mit denen aus einem anderen überschreibt:

```
>>> d = {"eins": '1', "zwei": '2',
... "plus": '+'}
>>> d.update({"minus": "-", "eins": "e"})
>>> d
{'eins': 'e', 'plus': '+', 'minus': '-',
'zwei': '2'}
```

Problems

(21.1)* Häufig möchte man Verteilungen von Wörtern bestimmen, d.h. Abbildungen von Wörtern auf die Anzahl ihres Vorkommens. Dazu könnte etwa folgender Code dienen, in dem w das zu zählende Wort ist und dist die Verteilung, die natürlich als Dictionary modelliert ist:

```
if dist.has_key(w):
  dist[w] = dist[w]+1
else:
  dist[w] = 1
```

Wozu braucht es die Bedienung auf `has_key` überhaupt? Und wie lässt sich das Ganze kürzer formulieren?

(21.2)* Dictionaries eignen sich gut zur Modellierung von Funktionen, deren Werte nur an endlich vielen (besser wenigen) Stellen bekannt sein müssen. Dies gilt insbesondere dann, wenn die tatsächliche Berechnung der Funktion zeitaufwändig oder unmöglich ist.

Schreibt eine Funktion `tabulate(valList, fct) -> dict`, die eine Liste von Werten und eine Funktion nimmt und ein Dictionary zurückgibt, in dem jeder Wert `w` in `valList` auf `fct(w)` abgebildet wird. Diese Funktion soll etwa folgendes tun:

```
>>> def sqr(x):
...     return x*x
...
>>> tabulate(range(10), sqr)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Beachtet, dass hinter sqr keine Klammern stehen: Gemeint ist die Funktion selbst (das, was der Name `sqr` referenziert) und nicht das Ergebnis eines Aufrufs der Funktion. Probiert auch andere Funktionen aus, die euch so einfallen.

Module

Auf der nächsten Folie werdet ihr sehen, wie ihr all die spannenden Funktionen aus dem Math-Modul[11] für dieses Spiel verwenden könnt. **(L)**

(21.3) Nehmt jetzt an, ihr hättet ein Dictionary, in dem weitere Dictionaries stehen. Ein Anwendungsfall wäre das Zählen von Bigrammen, also aufeinanderfolgenden Wörtern. Der Schlüssel im „Haupt-Dictionary" ist das erste Wort der Bigramme, mit dem man dann ein weiteres Dictionary erhält, in dem das jeweils zweite Wort steht. Dessen Wert könnte dann etwa die Häufigkeit des betreffenden Bigramms sein.

Schreibt eine Funktion `countBigram(w1, w2, dist) -> None`, das `w1` und `w2` in der beschriebenen Weise in `dist` einhängt. **(L)**

(21.4) Erklärt die Ausgabe von

```
>>> l = [{}]*4
>>> l[0]['n'] = 3
>>> l
```

Könnt ihr euch denken, warum

```
>>> l = []
>>> for i in range(4):
...   l.append({})
...
>>> l[0]['n'] = 3
>>> l
```

ganz anders ist? **(L)**

22. Module

Programme greifen auf Sammlungen von Funktionen zurück. Python erlaubt, diese in *Module* zu gruppieren. Python wird mit Modulen für viele Zwecke geliefert, etwa für Stringverarbeitung, grafische Benutzerschnittstelle, Verschlüsselung, Mathematik, Web-Programmierung usf. (vgl. Modul-Index[12]).

Um ein Modul zu verwenden, muss man es importieren:

```
>>> import math
>>> math.sin(0.3)
0.29552020666133955
```

Funktionen aus einem Modul werden also als modulname.funktionsname gerufen.

Eigene Module sind einfach: Man schreibt alles, was ins Modul gehört, in eine Datei mit der Endung py. Unsere compute-Funktion als Modul:

```
examples> cat compute.py
"""
A toy module to demonstrate modules and
dictionaries.  The main function is compute()
that takes a string with the expression and a
language specific dictionary.  Some dictionaries
are provided by this module
"""

germanWords = {"eins": '1', "zwei": '2',
  "plus": '+', "mal": '*'}
englishWords = {"one": '1', "two": '2',
  "plus": '+', "times": '*'}
```

[11] http://docs.cl.uni-heidelberg.de/python/lib/module-math.html

[12] http://docs.cl.uni-heidelberg.de/python/modindex.html

```
def compute(expr, langDict):
  """
  computes a pseudo-natural-language...
```

namespace clash

Ein Vorteil von Modulen ist, dass wir nicht jedes Mal, wenn wir eine Funktion brauchen, ihren Quelltext in das neue Programm kopieren müssen. Außerdem können wir dem Modul gleich noch Daten mitgeben, hier etwa die Wörterbücher für die verschiedenen Sprachen.

Danach:

```
>>> import compute
>>> compute.compute("one plus one",
...   compute.englishWords)
2
```

Man ahnt schon, dass es etliche Parallelen zwischen Modulen und Objekten gibt – in der Tat *sind* Module auch Objekte, aber eben sehr spezielle.

Objekte verfügen wie Funktionen über eigene Namespaces, die in diesem Fall so lange existieren sind wie das Objekt lebt – und diese Namespaces fragt man nach Werten, wenn man `obj.name` sagt.

Innerhalb eines Namespaces müssen natürlich alle Namen verschieden sein, in verschiedenen Namespaces dürfen aber durchaus gleiche Namen vorkommen. So könnte etwa ein Objekt `secureHash` eine Methode `compute` haben, die rein gar nichts mit der im Modul compute zu tun hat. Auch in verschiedenen Modulen dürfen Attribute gleichen Namens vorkommen. Das ist schon deshalb nützlich, weil verschiedene Module üblicherweise von verschiedenen Leuten geschrieben werden und diese natürlich nicht wissen können, was die jeweils anderen so an Namen vergeben.

Die Trennung der Namespaces ist gut und wünschenswert. Deshalb rate ich davon ab, Konstrukte wie `from module import *` zu verwenden, das alle Einträge des Namespace von `module` in den aktuellen Namespace übernimmt – einerseits weiß nachher niemand mehr, welche Namen aus welchen Modulen kommen, andererseits ist es nur eine Frage der Zeit, bis es zu *namespace clash*es kommt, also ein Name den anderen überschreibt. So könnte ein Modul den Namen `open` definieren (das tun nicht wenige), und nachher wundert man sich, warum ein Aufruf von open (das wir später als unseren Schlüssel zu Dateien kennenlernen werden) absolut nicht das tut, was es soll. Noch lustiger wirds, wenn zwei Module jeweils ein eigenes `open` definieren und das Verhalten des Programms plötzlich davon abhängt, in welcher Reihenfolge die Module importiert werden. Kurz: `from module import *` nur interaktiv und in äußersten Notfällen verwenden.

Die Variante `from module import name`, die lediglich `name` in den aktuellen Namespace übernimmt, ist dagegen in Einzelfällen nützlich und immerhin kontrollierbar. Übertreiben sollte man es damit aber auch nicht, zumal sowas subtile Probleme beim Neuladen von Modulen provoziert.

Module in der Programmentwicklung

Die dynamische Natur von Python bringt es mit sich, dass wir Funktion um Funktion entwickeln und in der Regel sofort testen können. Damit das relativ schnell geht, kann man in einem Fenster den Interpreter laufen lassen und in einem anderen den Editor mit dem Quelltext.

Wenn man das so macht, stellt sich allerdings das Problem, wie man dem Interpreter sagt, dass man den Quelltext geändert hat und er diesen neu laden und compilieren soll. Dafür gibt es die eingebaute Funktion `reload`, der man einfach den Namen des neu zu ladenden Moduls übergibt. Wenn ihr nochmal das compute-Skript von oben betrachtet, könntet ihr im Interpreter z.B. folgendes tun:

```
>>> import compute
>>> compute.compute("eins plus eins", compute.germanWords)
2
>>> compute.compute("Eins plus eins", compute.germanWords)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "compute.py", line 18, in compute
    pythonExpression.append(langDict[w])
```

```
KeyError: 'Eins'
```

Ups – vielleicht sollten wir Groß-/Kleinschreibung ignorieren? Wir verlassen den Interpreter nicht, sondern verändern das `expr.split()` aus der `compute`-Funktion in einem anderen Fenster zu `expr.lower().split()`. Zurück im Interpreter kann man es nochmal probieren (in der Regel bekommt man im Interpreter die alten Eingaben mit der Pfeil-hoch-Taste):

```
>>> compute.compute("Eins plus eins", compute.germanWords)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "compute.py", line 18, in compute
    pythonExpression.append(langDict[w])
KeyError: 'Eins'
```

– natürlich hat sich nichts geändert, der Interpreter weiß ja nicht, dass wir etwas am Quelltext geändert haben. Aber

```
>>> reload(compute)
<module 'compute' from 'compute.py'>
>>> compute.compute("Eins plus eins", compute.germanWords)
2
```

From module import name spielt übrigens nicht mit reload – das geht nicht, weil reload nur im Namespace des neugeladenen Moduls hantieren kann, nicht aber in dem aller anderen Module, die vielleicht Referenzen auf Objekte aus dem fraglichen Modul halten, schon, weil bei diesen Objekten gar nicht mehr zwingend klar ist, wem sie denn eigentlich „gehören“. Siehe auch die Aufgabe unten.

Besser als diese Sorte inkrementeller Entwicklung ist es aber, immer gleich Doctests zu den Funktionen zu schreiben. Näheres dazu gleich.

Problems

(22.1) Um einzusehen, was bei `reload` in Kombination mit `from module import` schief geht, probiert Folgendes:

```
>>> import compute
>>> from compute import germanWords
>>> id(germanWords)
>>> id(compute.germanWords)
>>> reload(compute)
>>> id(compute.germanWords)
>>> id(germanWords)
```

Die Funktion `id` ist uns oben schon mal begegnet – sie gibt so etwas wie einen Fingerabdruck zurück und erlaubt es uns, Objekte zu unterscheiden, selbst wenn sie zufällig den gleichen Wert haben. **(L)**

23. Docstrings

docstring

Wir haben sowohl in der compute-Funktion als auch später im Modul compute einen *docstring* verwendet.

```
def compute(expr, langDict):
  """computes a pseudo-natural-language expression.

  The expression is passed in the string expr,
  langDict provides a mapping of single words to
  python values and operators.
  """
```

Ein String, der gleich hinter einem Funktionskopf kommt, wird von Python als Dokumentation der Funktion verstanden und sollte beschreiben, was die Funktion tut, am besten als Verbphrase (Funktionen sollen etwas „tun“).

Das empfohlene Format ist: Drei öffnende Anführungszeichen, ein kurze Beschreibung in einem Satz, eine Leerzeile, weitere Bemerkungen (z.B. Beschreibung der Argumente), abschließend drei Anführungszeichen in einer eigenen Zeile.

Docstrings können zur Laufzeit ausgewertet werden:

```
>>> print compute.__doc__
  computes a pseudo-natural-language
...
>>> print eval.__doc__
eval(source[, globals[, locals]]) -> value
...
```

Es gab und gibt etliche Versuche, in Docstrings etwas strukturiertere Information unterzubringen, aus der z.B. Dokumentationsgeneratoren fertige Programmdokumentation erzeugen können. Das „offizielle“ pydoc übernimmt die Docstrings allerdings ziemlich wörtlich.

In der Python-Distribution enthalten ist das Modul doctest[13], das es erlaubt, in docstrings Beispiele für eine korrekte Operation der Funktion einzubetten und – das ist der Kick – das Python-System nachprüfen zu lassen, ob die Funktion auch wirklich tut, was die Beispiele behaupten.

(Weiterführend:) Inspiriert ist das von einer Software Engineering-Technik namens Extreme Programming[14], die Unit Tests (d.h. einzelne Einheiten des Programms werden getrennt getestet, bis man überzeugt ist, dass sie korrekt arbeiten) und Regression Tests (d.h. man überzeugt sich möglichst oft, dass das Programm nach Änderungen nicht neue Fehler bekommen hat und zumindest das funktioniert, was vorher funktioniert hat) betont. Doctests erlauben die Kombination dieser Techniken mit der dokumentierenden Kraft von Beispielen. Für große und komplizierte Unit Tests gibt es ein eigenes Modul[15].

Idee von doctest ist im Wesentlichen, einen interaktiven Dialog mit Python in den Kommentar zu schreiben, sowohl Ein- als auch Ausgabe. Der Interpreter kann dann später veranlasst werden, nachzuprüfen, ob dieser Dialog so möglich wäre.

Für unser Beispiel könnte so etwas folgendermaßen aussehen:

```
def compute(expr, langDict):
  """computes a pseudo-natural-language expression.

  The expression is passed in the string expr,
  langDict provides a mapping of single words to
  python values and operators.

  Examples:
  >>> langDict = {'eins': '1', 'plus': '+', 'zwei': '2'}
  >>> compute("eins", langDict)
  1
  >>> compute("eins plus zwei plus eins", langDict)
```

[13] http://docs.cl.uni-heidelberg.de/python/lib/module-doctest.html

[14] http://www.extremeprogramming.org/

[15] http://docs.cl.uni-heidelberg.de/python/lib/module-unittest.html

```
  4
  >>> compute("eins plus", langDict)
  Traceback (most recent call last):
  SyntaxError: unexpected EOF while parsing
  """
  pythonExpression = []
  for w in expr.split():
    pythonExpression.append(langDict[w])
  return eval("".join(pythonExpression))
```

– die oben zitierte Dokumentation von doctest gibt ein paar Tips, was „gute“ Tests sind und was nicht. Um Python dazu zu bringen, diese Tests auch durchzuführen (von selbst tut es das nämlich nicht), muss man folgender Schablone folgen:

```
def _test():
  import doctest, compute
  return doctest.testmod(compute)
```

Man muss also sowohl das doctest-Modul als auch „sich selbst“ importieren (was man besser nur in einer Funktion tut, auch wenn Python schlau genug ist, ein globales import des eigenen Moduls ohne Endlosschleife zu verarbeiten). Der eigentliche Test geschieht dann durch Aufruf der Funktion testmod aus dem doctest-Modul mit dem zu testenden Modul als Argument. Die Funktion `_test` kann man dann nach Geschmack z.B. beim Modulimport laufen lassen (was aber normalerweise nicht empfehlenswert ist) oder, wenn das Modul als Programm ausgeführt wird. Letzteres (was ich für alle Module empfehle, die nicht direkt als Programm ausgeführt werden sollen) lässt sich so erreichen:

```
if __name__=="__main__":
  _test()
```

Ändert man dann etwas im Modul, lässt man den Interpreter das Modul einfach ausführen und sieht bei entsprechend geschickt geschriebenen Doctests auch gleich, ob eine Änderung vielleicht ganz woanders etwas kaputt gemacht hat.

Um zu sehen, was passiert, ruinieren wir den Doctest in compute, der einen SyntaxError auslösen soll:

```
  >>> compute("eins plus")
```

(wir haben das zweite Argument gelöscht) und probieren das mal:

```
examples> python compute.py
**********************************************************************
Failure in example: compute("plus eins")
from line #11 of compute.compute
Exception raised:
Traceback (most recent call last):
  File "/usr/local/lib/python2.3/doctest.py", line 442, in
    _run_examples_inner compileflags, 1) in globs
  File "<string>", line 1, in ?
TypeError: compute() takes exactly 2 arguments (1 given)
**********************************************************************
1 items had failures:
   1 of   5 in compute.compute
***Test Failed*** 1 failures.
```

Python hat uns gleich ertappt. Wir bringens in Ordnung und ergänzen in einem Editor wieder das zweite Argument:

```
  >>> compute("eins plus", langDict)
```

Nochmal versuchen:

```
examples> python compute.py
examples>
```

– prima, der Rechner beschwert sich nicht mehr. Wenn man trotzdem sehen will, wie all die Tests erfolgreich laufen, kann man dem Skript die Option -v übergeben:

```
examples> python compute.py -v
Running compute.__doc__
0 of 0 examples failed in compute.__doc__
Running compute._test.__doc__
... viele Zeilen ...
```

```
4 passed and 0 failed.
Test passed.
```

balanciert

Problems

(23.1)* Seht durch, was ihr schon alles an Funktionen geschrieben habt. Schreibt docstrings für sie und nach Möglichkeit auch doctests.

24. Beispiel: Generierung von Sprachen

Bei unserer Erkundung von Python wollen wir uns jetzt eine Weile von einem konkreten Projekt mit einiger Relevanz für die Computerlinguistik leiten lassen, nämlich der Interpretation von Phrasenstrukturgrammatiken.

Sprachen werden gerne durch Grammatiken generiert. Dabei schreiben Ersetzungsregeln vor, was in jedem Schritt wodurch abgeleitet werden kann, ausgehend von einem Startsymbol, das im Allgemeinen S heißt. Beispiel für Regeln:

$$S \to \mathtt{b} \qquad S \to \mathtt{a}S\mathtt{a}.$$

Diese Regelmenge (mit der Menge der Nichtterminalen $\{S\}$ und der Menge der Terminalen $\{\mathtt{a}, \mathtt{b}\}$) erzeugt eine Sprache, in der die gleiche Zahl von a-Symbolen ein b rechts und links umschließen – klassisch könnte das für *balancierte* Klammern stehen (d.h. genauso viele Klammern gehen zu wie vorher aufgegangen sind). Eine mögliche Ableitung wäre

$$S \to \mathtt{a}S\mathtt{a} \to \mathtt{aa}S\mathtt{aa} \to \mathtt{aabaa}.$$

Für unsere Zwecke sollen Nichtterminalsymbole (die also noch ersetzt werden) vorerst durch Großbuchstaben dargestellt werden. Terminalsymbole (die nicht mehr ersetzt werden können und das schließlich erzeugte Wort ausmachen) sind dann alle anderen druckbaren Zeichen.

Weiter wollen wir uns auf kontextfreie Sprachen beschränken, also solche, bei denen auf der linken Seite der Regeln immer nur genau ein Nichtterminalsymbol steht. Wir lassen aber zu, dass auf der rechten Seite gar nichts steht („ϵ-Regeln“).

Wir wollen ein Programm schreiben, das eine von so einer Regelmenge erzeugte Sprache berechnet. Überlegen wir uns zunächst die Datenstrukturen, wie wir brauchen. Wörter sind nach der obigen „Definition“ Sequenzen von Terminalen und Nichtterminalen, also Großbuchstaben und anderen Zeichen. Mithin sind sie durch Strings ausreichend modelliert. Bei der Regelmenge müssen wir uns etwas mehr Gedanken machen. Sie ist letztlich eine Abbildung von Großbuchstaben auf Wörter (also Strings).

Der naive Ansatz zur Modellierung der Regelmenge, ein Dictionary von Strings:

```
rules = {
'S': 'b',
'S': 'aSa',
}
```

(Das Komma hinter der letzten Zeile im Initialisierer von `rules` ist syntaktisch legal und macht spätere Ergänzungen leichter) scheitert:

```
>>> rules
{'S': 'aSa'}
```

Oops – ein Schlüssel kann nur auf einen Wert verweisen. Abhilfe: Werte sind nicht mehr nur Strings, sondern Listen von Strings. Also:

```
rules = {
'S': ['b', 'aSa'],
}
```

Wir haben jetzt als die Datenstrukturen soweit festgelegt. Die Frage ist, was tun wir damit? Wir müssen jetzt also einen Algorithmus ersinnen, der auf den Daten korrekt operiert. Ganz

grob gesprochen müssen wir einfach so lange Nichtterminale ersetzen, bis es entweder keine anwendbaren Regeln mehr gibt oder keine Nichtterminale mehr im betrachteten Wort stehen. Dieses Wort möchten wir dann ausgeben und nach neuen rein terminalen Wörtern suchen.

Zur genaueren Formulierung von Algorithmen, vor allem aber zur Strukturierung der eigenen Gedanken und zur späteren Dokumentation eines Programm wurden diverse Techniken entwickelt. Zumal bei komplizierteren Problemen lohnt es sich, vor der Implementation zunächst mit einer dieser Methoden festzulegen, was ein Programm eigentlich tun soll. In den Urzeiten wurden dazu Flussdiagramme eingesetzt, die ähnlich unserem Bild vom Control Flow aussahen (allerdings mit Schablonen und Kästchen etwas übersichtlicher waren). In solchen Flussdiagrammen lassen sich Kontrollstrukturen wie etwa Schleifen nur sehr krude darstellen, und generell sind die heute eher etwas für WirtschaftswissenschaftlerInnen als für ProgrammiererInnen, weil die Programmiersprachen selbst inzwischen viel problemorientierter sind als die Flussdiagramme.

Als in den siebziger Jahren die strukturierte Programmierung Mainstream wurde, schwörten viele Menschen eine Zeitlang auf so genannte Struktogramme oder Nassi-Shneidermann-Diagramme. Sie sind besser auf moderne Programmiersprachen abgestimmt und können zum Beispiel vernünftige Schleifen darstellen. Letztlich allerdings malt man dabei doch die meiste Zeit, und das Entwurfsniveau ist eher niedriger als etwa beim Entwerfen in Python selbst.

In den letzten 20 Jahren hat sich daher letztlich ein nicht allzu standardisierter Pseudocode als Mittel der Wahl durchgesetzt, um Algorithmen auf einem etwas höheren Niveau als dem der Zielsprache darzustellen. Zum Entwurf eines Programmsystems ist er aber in der Regel nicht ausreichend. Heutige Programme bestehen aus vielen Komponenten, die auf viele Arten wechselwirken – um dies zu modellieren, braucht man andere Werkzeuge; ein Stichwort dabei mag UML sein.

Zurück zu unserem Problem. Ein möglicher Algorithmus zur Durchführung einer Ableitung könnte sein:

Eingabe: Ein Wort mit mindestens einem Nichtterminalsymbol w, eine Regelmenge R

Sei e die leere Liste

Sei n das am weitesten links stehende Nichtterminalsymbol in w

für alle Regeln $n \to x$ in R

> Füge den String, der durch Ersetzen des ersten Vorkommens von n in w durch x entsteht, zu e hinzu

Rückgabe: e, die Liste mit allen Ableitungen, die durch Ersetzen des am weitesten links stehenden Nichtterminals in w entstehen können – insbesondere die leere Liste, wenn keine passenden Regeln vorhanden sind.

Implementation des Ableiters:

```
def deriveLeft(curWord, rules):
  leftmostNonTerm = getLeftmostNonTerm(curWord)
  res = []
  for right in rules.get(leftmostNonTerm, []):
    res.append(curWord.replace(
      leftmostNonTerm, right, 1))
  return res
```

Diese Implementation spiegelt den Algorithmus von oben ziemlich genau wieder. Trotzdem sehen wir uns mal Zeile für Zeile an, was da so passiert:

1. Zunächst sagen wir, dass die Funktion `deriveLeft` heißen soll, ein Wort (`curWord`) und die Regeln nehmen soll. In einem Docstring sollte eigentlich erklärt werden, dass curWord ein String sein soll und rules eine Abbildung von Nichtterminalen auf eine Sequenz von Ersatzstrings.
2. Dann besorgen wir uns das am weitesten links stehende Nichtterminal. Diese Funktionalität hat eine so einfache Schnittstelle, dass wir sie in eine separate Funktion verlagert haben.
3. Jetzt bereiten wir unsere Ergebnisliste vor. Wir haben noch keine Ergebnisse, also ist die Liste leer.

4. Wir iterieren über alle möglichen rechten Seiten. Wenn unser Nichtterminal gar nicht als linke Seite auftritt, iterieren wir über die leere Liste, also gar nicht. Dann geben wir auch nur die leere Liste zurück.
5. Für jede rechte Seite ersetzen wir das erste Vorkommen der linken Seite (fürs „erste“ sorgt das letzte Argument des `replace` – die eins sagt, dass nur ein Zeichen ersetzt werden soll) durch die augenblickliche rechte Seite, die wir in `right` gespeichert haben.
6. Dies ist eine Fortsetzungszeile. Seht euch bei Bedarf nochmal die Anmerkungen über die Syntax von Anweisungen oben an.
7. Wir sind fertig und geben unser Ergebnis zurück.

Die Funktion `getLeftmostNonTerm` muss den ersten Großbuchstaben im String zurückgeben. Eine mögliche Implementation:

```
def getLeftmostNonTerm(word):
  for c in word:
    if c.isupper():
      return c
```

Anmerkungen dazu:

- `isupper` ist eine Methode von Strings und gibt „wahr“ zurück, wenn der betreffende String nur aus Großbuchstaben besteht.
- Wenn in `word` kein Großbuchstabe vorkommt, wird kein `return`-statement ausgeführt. Python-Funktionen ohne return geben den Wert `None` zurück:

```
>>> print getLeftmostNonTerm("allesklein")
None
>>> if getLeftmostNonTerm("allesklein")==None:
...     print "Kein Großbuchstabe"
...
Kein Großbuchstabe
```

Wir speichern `rules` und die beiden Funktionen in gener1.py. Jetzt:

```
>>> from gener1 import *
>>> deriveLeft("S", rules)
['b', 'aSa']
>>> deriveLeft("baSb", rules)
['babb', 'baaSab']
```

Hier dürfen wir `from gener1 import *` mal machen: Im interaktiven Interpreter sparen wir uns damit Tipparbeit (und ich mir Platz auf den Folien). Das ist aber auch so in etwa der einzige legitime Einsatz.

Problems

(24.1)* Tippt unsere Funktionen ab (tut es wenigstens diese eine Mal, auch wenn es erstmal sinnlos erscheinen mag – es wird lehrreich sein, eure Tippfehler zu finden). Probiert deriveLeft mit anderen Grammatiken, etwa

$$S \to \texttt{N} \quad S \to S\,O\,S \quad N \to 0 \quad N \to 1$$
$$N \to 2 \quad O \to + \quad O \to -.$$

Diese Grammatik beschreibt offenbar einfache Ausdrücke. Lasst sie auf Strings wie `"S"` oder `"1+N` laufen, nehmt Ergebnisse von deriveLeft und wendet die Funktion nochmals darauf an.

(24.2) K. R. Acher hat bei seiner Implementation von deriveLeft den Ausdruck `rules.get(leftmostNonTerm, [])` durch `rules[leftmostNonTerm]` ersetzt. Lange Zeit ging alles gut. Dann begann das Programm plötzlich abzustürzen, ohne dass etwas an ihm geändert wurde, nur die Eingabe hatte sich geändert. Was war passiert? **(L)**

25. Rekursion

Wir brauchen jetzt eine Funktion, die `deriveLeft` so aufruft, dass am Schluss alle möglichen Ableitungen erzeugt wurden. Sie soll ein Wort bekommen und für alle Wörter, die sich daraus ableiten lassen, eine neue Ableitung starten.

Überlegen wir uns zunächst wieder Pseudocode dafür:

Funktion generiereSprache

Eingabe: Ein Wort w, eine Regelmenge R

Wenn w nur aus Terminalen besteht

    gib w aus

sonst

    für alle möglichen direkten Linksableitungen w' von w

        rufe generiereSprache mit w' und R auf

Eine direkte Linksableitung ist dabei das, was wir in `deriveLeft` erzeugt haben: Ein Wort, das durch Ersetzung des am weitesten links stehenden Nichtterminals im Quellwort entsteht.

Die Ableitungsfunktion ruft sich hier selbst für jede neu erzeugte Ableitung auf: *Rekursion*. Bei jeder Rekursion ist es wichtig, eine Abbruchbedingung zu haben, damit sich die Funktion nicht endlos selbst aufruft. In unserem Pseudocode ist die Abbruchbedingung, dass das Wort nur aus Terminalsymbolen besteht – es werden dann keine weiteren rekursiven Aufrufe gemacht. Es ist bei jedem rekursiven Algorithmus entscheidend, nachweisen zu können, dass die Abbruchbedingung früher oder später wahr wird, denn endlose Rekursion führt in jedem Fall zu einem zu einem Programmfehler. Häufig ist das kein großes Problem, denn normalerweise besteht ein Rekursionsschritt aus einer Zurückführung einer komplizierteren Aufgabe auf eine einfachere („Lösung für $n+1$ auf Lösung für n“).

In unserem Fall ist das leider nicht so, und in der Tat wird unsere Abbruchbedingung für die viele Sprachen (nämlich die, die beliebig viele Wörter generieren können), nicht wahr. Um dennoch zu einem Abbruch zu kommen, fügen wir in den Code eine weitere Abfrage auf die Länge der Ableitung ein:

```
def isTerminal(word):
  return not getLeftmostNonTerm(word)

def generate(curWord, rules, maxLen):
  if len(curWord)>maxLen:
    return
  if isTerminal(curWord):
    print curWord
    return
  for deriv in deriveLeft(curWord, rules):
    generate(deriv, rules, maxLen)
```

Die Definition von `isTerminal` mag albern erscheinen. Wir werden aber später die Definition von Terminalsymbolen ändern. Wenn so eine Änderung passiert – und das ist in Softwareprojekten die Regel –, müssen wir nicht jeden Aufruf von `isupper` durchsehen, ob er zufällig eigentlich prüft, ob wir ein nur aus Terminalen bestehendes Wort haben, sondern müssen nur die Definition von isTerminal ändern.

Die Funktion `generate` ist eigentlich einfach: Zunächst wird die Abbruchbedingung überprüft, dann wird, wenn wir bereits ein Wort nur aus Terminalen haben, es ausgegeben und die Rekursion ebenfalls abgebrochen. Ansonsten gehen wir einfach die von `deriveLeft` gelieferten Ableitungen durch und leiten weiter ab.

Test:

```
>>> from gener1 import *
>>> generate("S", rules, 6)
b
```

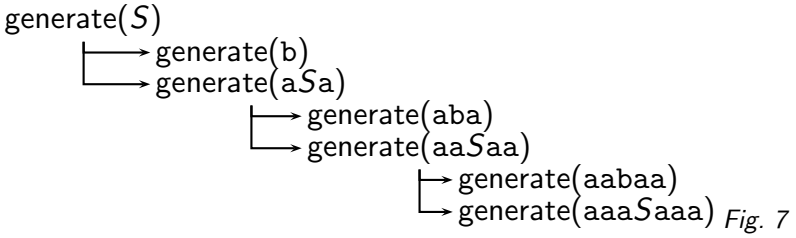


Fig. 7

```
aba
aabaa
```

Rekursion wirkt anfangs manchmal ein wenig magisch. Sehen wir uns an, wie die Rekursion hier abgelaufen ist:

(cf. Fig. 7)

Der Klassiker beim Einführen der Rekursion ist die Fakultät,

$$n! = 1 \cdot 2 \cdot 3 \cdots n.$$

Die rekursive Definition ist:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & \text{sonst} \end{cases}$$

Sie ist unter anderem deshalb ein Klassiker, weil es hier einfacher ist, die Rekursion zu durchschauen. Jeder Funktionsaufruf (mit Ausnahme des letzten) erzeugt nämlich wieder genau einen Funktionsaufruf.

In Python:

```
def fak(n):
  if n==0:
    return 1
  else:
    return n*fak(n-1)
```

Problems

(25.1)* Schreibt eine iterative Fassung der Fakultätsfunktion, d.h. eine, die die Fakultät in einer Schleife durch fortgesetzte Veränderung einer Variable berechnet. Macht euch vorher klar, warum dabei die Konvention, dass `range(n)` die Zahlen von 0 bis $n-1$ zurückgibt, euer Leben auf doppelte Art schwer macht. Hinweis: Man kann `range` auch mit zwei Argumenten aufrufen. Das erste ist dann der Startwert der resultierenden Sequenz, das zweite der Endwert plus eins. **(L)**

(25.2)* Nehmt die rekursive Implementation der Fakultät oben und spickt sie so lange mit `print`-Statements, bis ihr versteht, was vor sich geht.

(25.3) Nehmt die Abfrage auf die Länge der Ableitung aus `generate` heraus und lasst sie mit unserer Standardgrammatik laufen. Was passiert? Ändert sich das, wenn ihr die einfache Ausdrucksgrammatik nehmt? **(L)**

26. Dateien

Datei

file

Was ist, wenn wir die die Grammatik ändern wollen? Wir müssen jedes Mal den Quellcode des Programms ändern. Außerdem ist das Format nicht sehr eingabefreundlich. Wir möchten lieber etwas wie

```
S -> b
S -> aSa
```

in eine *Datei* (ein *file*) schreiben, die das Programm dann liest.

Dateien sind in Python Objekte, die von der eingebauten Funktion `open` erzeugt werden:
`grammarFile = open("grammar")`

In modernen Python-Versionen kann man Dateien auch über die „eingebaute Funktion“ `file` erzeugen. In gewisser Weise ist dies besser, weil es analog zu `int`, `str` und Freunden funktioniert: Es *konstruiert* einen Wert mit dem durch seinen Namen gegebenen Typ aus etwas anderem (im Fall von `file` ist dieses andere praktisch immer ein String mit dem Dateinamen).

Andererseits können alte Python-Versionen nichts damit anfangen (auch wenn man das mit `file = open` beheben könnte), und die meisten Leute verwenden immer noch `open`, so dass man wohl vorerst noch bei `open` bleiben sollte.

Dateien sind eigentlich auch nur eine Datenstruktur – sie unterscheidet sich allerdings von allem, was wir bisher kennen, in einem Punkt: Man kann nicht ohne weiteres auf beliebige Elemente innerhalb der Datei zugreifen. Tatsächlich steht in dem, was wir von `open` zurückbekommen (das `file`-Objekt) nicht der Inhalt der Datei, sondern nur etwas, das einem den Zugriff auf die wirklichen Daten erlaubt, die weiter (etwa) auf der Platte liegen.

Der Grund dafür ist, dass Massenspeicher und ähnliche Geräte, die man mit Dateien abstrahiert, anders gebaut sind als der Hauptspeicher, in dem Listen, Dictionaries usf liegen. Die heute üblichen Festplatten beispielsweise sind als rotierende Scheiben realisert, über denen je ein Schreib-/Lesekopf schwebt. Um ein Byte zu lesen, muss man den Kopf zunächst in die richtige Spur bewegen und dann warten, bis es durch die Rotation vorbeiflitzt. Weil das ziemlich aufwändig ist, ist der Zugriff auf Daten am Massenspeicher tief unten an der Hardware so geregelt, dass man immer nur einen ganzen Haufen Zeichen (ein paar hundert oder tausend davon) auf einmal lesen oder schreiben kann. Besonders gut ist, wenn man die Daten in der Reihenfolge haben will, in der sie auf der Platte stehen. Python und das Betriebssystem verstecken einen Teil dieser Schwierigkeiten, trotzdem muss man mit Dateien anders umgehen als, sagen wir, mit Listen.

Der Fall, dass der Zugriff auf Daten mit relativ großem Aufwand verbunden ist und es noch dazu eine Art natürliche Reihenfolge des Lesens oder Schreibens gibt, ist gar nicht so selten – liest man etwa Daten vom Netz oder schreibt sie in einen Digital-Analog-Wandler, der Töne daraus macht, sehen die Verhältnisse sehr ähnlich aus, und in der Tat werden auch in diesen Fällen die wirklichen Vorgänge hinter Dateien oder Objekten, die ganz ähnlich funktionieren, versteckt.

Ein paar Methoden von Dateien:

- `read([size])` – Liest den ganzen Inhalt der Datei (oder nur `size` Bytes) in einen String, der dann zurückkommt.
- `readline` – Liest die nächste Zeile aus der Datei und gibt sie in einem String zurück. Wenn nichts mehr in der Datei steht, kommt ein leerer String zurück.
- `readlines` – Gibt alle Zeilen der Datei als Liste zurück
- `close` – Schließt die Datei

Damit könnten wir unsere Grammatik so einlesen:

```
def readRules(inFileName):
  rules = {}
  rulesFile = open(inFileName)
  rawRules = rulesFile.readlines()
  rulesFile.close()
  for rawRule in rawRules:
    leftRight = rawRule.split("->")
```

```
    if len(leftRight)==2:
      key = leftRight[0].strip()
      val = leftRight[1].strip()
      rules.setdefault(key, []).append(val)
  return rules
```

Anmerkungen:

- Die Funktion legt als erstes die Datenstruktur an, die sie nachher zurückgibt. In statischeren Sprachen übergibt man häufig eine Datenstruktur, die die Funktion dann füllen soll, in Python mit seiner dynamischen Speicherverwaltung ist das in der Regel schlechter Stil.
- Wenn man eine Datei schließt, werden verschiedene Verwaltungsstrukturen freigegeben. Dateien sollten immer so bald wie möglich geschlossen werden, um nicht zu viele dieser Verwaltungsstrukturen zu belegen.
- (Weiterführend:) In der gegenwärtigen Python-Implementation ist das aber häufig kein großes Problem, weil Python die Datei selbstständig schließt, sobald wir keine Referenz mehr darauf haben. Wenn nun Dateien in Funktionen behandelt werden, wird die Datei automatisch geschlossen, wenn wir die Funktion verlassen, in ihr definierte Variablen also vergessen werden.
- (Weiterführend:) Dieser Rechnung folgend hätten wir hier auch `rawRules = open("grammar").readlines()` schreiben können; die Datei wäre dann schon nach dieser Zeile (genauer: vor der Zuweisung) geschlossen worden, weil nach der Anwendung der `readlines`-Methode keine Referenz mehr auf das Ergebnis von `open` existiert. Leider garantiert die Python-Spezifikation dieses Verhalten nicht, und in der Tat kann es bei der Verwendung des Java-basierten Python Probleme geben. Deshalb ist es (insbesondere für Schreibzugriffe) besser, Dateien explizit zu schließen, auch wenn wir hier sehr häufig das Idiom `open(...).readX()` verwenden werden.
- Die `strip`-Methode, die wir auf das Ergebnis des `split` anwenden, ist nötig, um Leerzeichen und Zeilenvorschübe zu entfernen.
- Die Kondition auf die Länge von `leftRight` dient dazu, leere Zeilen (allgemeiner: Zeilen, die unserer Syntax nicht entsprechen) sicher überlesen zu können. In einem realen Programm würde man sicher eine elaboriertere Behandlung von Formatfehlern in der Eingabe vorsehen wollen.

In Dateien kann man auch schreiben. Man muss sie dann mit `open(<dateiname>, "w")` (für „write“) erzeugen, danach kann man per `write` reinschreiben:

```
>>> f = open("test", "w")
>>> f.write("Hallo Welt\n")
>>> f.close()
>>> ^D
tucana:examples> cat test
Hallo Welt
```

Dateien und Encoding

Aus Dateien bekommt ihr durchweg Bytestrings zurück, für die Interpretation der Bytes seid ihr selbst verantwortlich. Analog könnt ihr in Dateien nur Bytestrings schreiben, der Versuch, Unicode-Strings zu schreiben, scheitert, sobald Nicht-ASCII-Zeichen im String sind.

Das ist natürlich lästig, wenn man Sprache verarbeiten möchte, aber nicht zu vermeiden, weil kein mir bekanntes Betriebssystem zu einer Textdatei das von dem schreibenden Programm verwendete Encoding speichert (so ein Datensatz wäre ein Beispiel für so genannte *Metadaten*, also Daten über Daten – Metadaten, die übliche Betriebssysteme speichern, sind beispielsweise Zugriffszeiten, Eigentümer von Dateien usf.). Verschiedene Kommunikationsprotokolle (z.B. http oder auch das für E-Mail entwickelte MIME) erlauben übrigens, auf Seitenkanälen Information über das Encoding auszutauschen, aber das hilft genau dann, wenn man mit entsprechenden Daten umgeht. Dokumentation dazu findet ihr in den entsprechenden Modulen.

Das bedeutet: Wenn euer Programm Dateien verwendet und ihr Unicode-Strings verwenden wollt, müsst ihr ein Encoding vereinbaren und ensprechend die Strings, die von read() zurückkommen, Dekodieren und zum Schreiben wieder Enkodieren. Im codecs-Modul ist übrigens eine Funktion open enthalten, die File-Objekte zurückgeben, die diese Kodieren und Dekodieren selbst machen. Wir werden im Zusammenhang mit Vererbung sehen, wie man sowas selbst schreiben kann.

Exception

Solange ihr aber mit Bytestrings auskommt (das könnt ihr meist, solange ihr euch auf einem Rechner bewegt und lediglich westeuropäische Sprachen verarbeitet) und euer Programm das gleiche Encoding verwendet wie eure Daten, müsst ihr euch um diese Dinge noch keine Gedanken machen.

Problems

(26.1)* Schreibt eine Funktion `catUpper(fName) -> None`, die den Inhalt der durch fName bezeichneten Datei in Großbuchstaben ausgibt. Wenn ihr keinen anderen Dateinamen zum Probieren wisst, tut es auch der Name des Skripts, in das ihr die Funktion schreibt. Wenn das so ist, solltet ihr euch allerdings schleunigst mit dem Dateisystem eures Betriebssystems auseinandersetzen. **(L)**

(26.2) `readline` und Freunde funktionieren nur für Textdateien, d.h. Dateien, in denen auch wirklich Text steht. Für Binärdateien (also solche, die nicht aus Zeilen mit harmlosen Text bestehen) kommen unter Umständen sehr komische Dinge heraus. Probiert mal, was folgendes tut:

```
>>> f = open("...(Pfad zu Binärdatei)...")
>>> f.readline()
>>> f.readline()
...
```

Unter Unix empfehle ich als Binärdatei etwas wie /bin/cat oder /lib/libc.so, unter Windows ist die Verwendung einer Word-Datei recht instruktiv.

(26.3) Schreibt eine Funktion `filecopy(fName1, fName2) -> None`, die eine Textdatei fName1 nach fName2 kopiert und dabei nicht allzu viel Hauptspeicher braucht. **(L)**

27. Exceptions

Was ist, wenn die Datei nicht existiert?

```
>>> f = open("junk")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory:
'junk'
```

Sowas heißt *Exception* und ist ein allgemeiner Mechanismus zur Fehlerbehandlung. Es gibt verschiedene Sorten von Exceptions:

- SyntaxError – es ist ein syntaktischer Fehler im Quelltext
- IOError – eine Datei existiert nicht, man darf nicht schreiben, die Platte ist voll...
- IndexError – in einer Sequenz gibt es das angeforderte Element nicht
- KeyError – ein Mapping hat den angeforderten Schlüssel nicht
- ValueError – eine Operation kann mit diesem Wert nicht durchgeführt werden (Wurzel aus negativen Zahlen, `int` für einen String wie `"bla"`...)

(und viele mehr).

Exceptions kann man behandeln:

```
>>> import sys
>>> try:
...     f = open("junk")
... except IOError:
...     sys.stderr.write("Hilfe...\n")
```

```
...
Hilfe...
```

Standardfehlerkanal

Standardausgabe

Standardeingabe

Anmerkungen:

- Die Syntax von `try...except` folgt der normalen Syntax von Python: nach `try:` und `except:` wird eingerückt, was hinter `try` kommt, wird ausgeführt, bis ein Fehler auftritt, was hinter `except` kommt, wird nur ausgeführt, wenn im `try`-statement eine Exception der angegebenen Art aufgetreten ist.
- Hinter `except` können ein oder mehrere Exception-Typen stehen (wenn es mehrere sind, in Klammern durch Kommata getrennt schreiben). Wenn keine Typen angegeben ist, reagiert das except auf alle Exceptions (was man allenfalls dann tun sollte, wenn eine Exception dann immer noch eine sichtbare Fehlermeldung erzeugt – ein except:, bei dem anschließend die Exception ignoriert wird, ist die sicherste Art, die Fehlersuche fast unmöglich zu machen).
- `sys` ist ein Modul, in dem allerlei Sachen stehen, die etwas mit dem „System“ zu tun haben, etwa die Standardein- und -ausgabe, Funktionen zum Abbrechen des Programms oder zum Verarbeiten von tracebacks, der Pfad, auf dem Python nach Modulen sucht und vieles mehr.
- `sys.stderr` ist eine in `sys` vordefinierte Datei, der *Standardfehlerkanal*. Fehlermeldungen sollten immer dorthin ausgegeben werden. Vorteil: Sie werden immer noch gesehen, auch wenn die *Standardausgabe* (dorthin gibt `print` aus) umgeleitet ist. In `sys` gibt es noch zwei weitere Dateien: `sys.stdin`, die *Standardeingabe* (von dort liest `raw_input`) und `sys.stdout` (die erwähnte Standardausgabe). Es ist eine von fast allen anderen Systemen übernommene Unix-Tradition, möglichst alles im Rechner als Datei darzustellen, insbesondere also auch „Tastatur“ und „Monitor“ (das sollte man jetzt allerdings nicht so wörtlich nehmen).

Hinter einem `try:` können mehrere `except`-statements für verschiedene Typen von Exceptions kommen, und schließlich noch ein `else`-statement, der ausgeführt wird, wenn keine Exception ausgelöst wurde.

Exceptions kann man auch selbst auslösen:

```
>>> raise IOError("Brainless User")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: Brainless User
```

Hinter `raise` steht also ein (normalerweise) mit einem String instanziertes Exception-Objekt.

Eine gewisse Sonderrolle hat die Exception `KeyboardInterrupt`. Sie wird ausgelöst, wenn die Benutzerin Control-C drückt (allgemeiner die Interrupt-Taste, noch allgemeiner, wenn das Programm das INT-Signal (vgl. man signal) geschickt bekommt). Auch diese Exception kann man fangen.

Schließlich gibt es noch das nicht wirklich mit Exceptions zusammenhängende `try...finally`-Konstrukt, bei dem, ganz egal, was im `try:`-statement passiert, der Code hinter `finally:` auf jeden Fall ausgeführt wird. So etwas ist praktisch, um in kritischen Codepassagen am Ende wieder einen definierten Zustand herzustellen. Wir werden später Beispiele dafür sehen.

Problems

(27.1)* Probiert, Exeptions der erwähnten Arten auszulösen und zu fangen. **(L)**

(27.2)* Wie kann man ein Programm schreiben, dass noch schnell „Röchel“ sagt, wenn es durch ein Control-C abgebrochen wird? Kann man auf diese Weise Control-Cs ganz ignorieren? **(L)**

(27.3) Um zu verstehen, was das mit stderr und stdout soll, schreibt euch ein Programm, das nach einer Weile eine Fehlermeldung ausgibt, einmal auf stderr, einmal auf stdout, also etwa so:

```
import time, sys

time.sleep(3)
sys.stderr.write("Fehler!!!\n")
```

Ruft dieses Programm von der Kommandozeile aus auf und leitet die Ausgabe in eine Datei um (das geht auch unter DOS/Windows):

```
examples> python prog.py > junk
```

Was beobachtet ihr?

28. Tupel

Wir bauen unseren Regelleser mit Exceptions:

```
def readRules(inFileName):
  rules = {}
  for rawRule in open(inFileName).readlines():
    try:
      key, val = [s.strip() for s in
        rawRule.split("->")]
      try:
        rules[key].append(val)
      except KeyError:
        rules[key] = [val]
    except ValueError:
      sys.stderr.write("Syntaxfehler in "+
        rawRule.strip()+"\n")
  return rules
```

Anmerkung: Das try-except-Konstrukt um den Eintrag im Dictionary herum ist hier nur zur Illustration. Wenn man (wie hier) damit zu rechnen hat, dass eine Exception in der Regel ausgelöst wird und nur im Ausnahmefall („as an exception“) nicht ausgelöst wird, ist es besser, den „Ausnahmefall“ manuell zu behandeln (oder eben wie in der Vorfassung von `readRules` Konstrukte zu verwenden, die den Ausnahmefall gleich korrekt behandeln).

Was tut die Zeile mit `key, val = ...`?

Die rechte Seite werden wir uns auf der nächsten Folie ansehen – zunächst kommt da normalerweise eine zweielementige Liste raus:

```
>>> [s.strip() for s in
...         "a -> b".split("->")]
['a', 'b']
```

Tupel

Ein *Tupel* ist eine nichtveränderbare Sequenz; in Python wird sie mit Kommata getrennt und optional in Klammern geschrieben. Tupeln kann man auch zuweisen, wobei automatisch ein- und ausgepackt wird, auch aus anderen Sequenzen:

```
>>> b = (2+3, 4*3)
>>> z1, z2 = b
>>> z1, z2
(5, 12)
>>> z1, z2 = "ab"
>>> z1, z2
('a', 'b')
>>> t = ("a",)
>>> t, ()
(('a',), ())
```

In den lezten beiden Zeilen steht ein einelementiges Tupel und das leere Tupel.

Wenn man versucht, Tupel falscher Größe zu entpacken, kommt ein ValueError – eben der, der oben als Syntaxfehler abgefangen wird:

```
>>> z1, z2 = [1, "zwei", ()]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: unpack list of wrong size
```

Warum Tupel?

Ein Tupel ist also eine inhomogene Sequenz (es können also Werte verschiedener Typen in ihm stehen). Damit unterscheidet es sich nicht wesentlich von Listen. Warum also nicht gleich Listen nehmen?

Der Unterschied ist technisch nur die Unveränderbarkeit. Sie ist für den Interpreter von Vorteil, weil er so nicht damit rechnen muss, dass sich die Daten plötzlich ändern. Wann immer sich die Daten in einer Sequenz nicht ändern werden (z.B. bei den Zuweisungen, aber auch bei der `items`-Methode von Dictionaries), setzt Python Tupel ein (es sei denn, der/die AutorIn des betreffenden Codes hätte Mist gebaut).

(Weiterführend:) Guido van Rossum, immer noch Spiritus Rector Pythons (der Fachbegriff lautet BDFL, Benevolent Dictator For Life), besteht darauf, dass die Vorstellung, Tupel seien einfach unveränderbare Listen, irreführend ist. Seine Einlassung ist, dass Listen für homogene Daten (also Sequenzen gleichen Typs), Tupel dagegen für inhomogene Daten („records“, wenn man so will) verwendet werden sollten. Diese Auskunft (vgl. etwa die FAQ dazu[16]) ist allerdings nicht unumstritten. Meine persönliche Meinung dazu ist, dass die radikale Ablehnung der Verwendung von Tupeln als immutable lists wohl etwas zelotisch ist, dass am aber wohl gut daran tut, die Meinung des BDFL wenigstens im Kopf zu haben.

Durch das automatische Ein- und Auspacken ergeben sich einige nette syntaktische Möglichkeiten. So kann man bequem „mehrere Werte“ aus einer Funktion zurückgeben:

```
>>> def analyze(verb):
...     numerus = {"bist": "sing", "seid": "plur", "sind": "plur"}
...     person = {"bist": 2, "seid": 2, "sind": 3}
...     return numerus[verb], person[verb]
...
>>> num, pers = analyze("bist")
>>> print num
sing
>>> num, pers = analyze("seid")
>>> print pers
2
```

Natürlich würde man eine morphologische Analyse in Wirklichkeit eher nicht über hartkodierte Dictionaries machen, die Schnittstelle der Funktion könnte aber durchaus so aussehen.

Das geht sogar bei Schleifen:

```
>>> def enumerate(seq):
...     res = []
...     for i in range(len(seq)):
...             res.append((i, seq[i]))
...     return res
...
>>> for index, word in enumerate("hallo welt".split()):
...     print "%s ist an %d-ter Stelle"%(word, index+1)
...
hallo ist an 1-ter Stelle
welt ist an 2-ter Stelle
```

Die Funktion `enumerate` mit der von uns verwendeten Schnittstelle ist übrigens in neueren Pythons bereits eingebaut – allerdings in einer etwas effizienteren Implementation, die den Aufbau

[16] `http://www.python.org/doc/faq/general.html#id46`

einer eigenen Liste (die Zeit und Speicher in Anspruch nimmt) vermeidet. Wie sowas geht, werden wir später sehen.

items

Wie oben schon angekündigt, gibt die `items`-Methode von Dictionaries eine Liste von Tupeln zurück. Mensch könnte erwarten, dass man aus so einer Liste von Tupeln dann auch wieder ein Dictionary machen könnte. Das ist tatsächlich so (der Ehrlichkeit halber sollte gesagt werden, dass es beliebige Sequenzen der Länge zwei statt der Tupel auch tun würden):

```
>>> dict([(1, "eins"), (2, "zwei"), (3, "drei")])
{1: 'eins', 2: 'zwei', 3: 'drei'}
```

Hier ist das natürlich nicht so interessant, aber es ist bequem, wenn man z.B. Attribut-Wert-Paare folgendermaßen in einer Datei gespeichert hat:

```
total: 27
adjs: 2
noms: 12
verbs: 8
parts: 5
```

In aller Kürze bekommt man sowas in ein Dictionary durch eine Funktion wie

```
import string

def readDict(fName):
  """reads the contents of fName and interprets each line as
  colon-separated attribute-value pair.

  Returns this mapping as a dictionary.  No error checking, will
  raise all kinds of funky exceptions if you mess anything up.
  """
  def parseLine(ln):
    return tuple([s.strip() for s in  ln.split(":")])
  return dict(
    [parseLine(ln) for ln in
      open(fName).readlines()])
```

Anmerkung: Ja, es ist legal, innerhalb von Funktionen weitere Funktionen zu definieren. Diese „gibt“ es dann auch nur innerhalb der Elterfunktion.

Das automatische Ein- und Auspacken bringt übrigens mit sich, dass die bequemste Art, über Schlüssel und Werte eines Dictionaries zu iterieren, so aussieht:

```
>>> d = {1: "eins", 2: "zwei", 27: "siebenundzwanzig"}
>>> for num, word in d.items():
...     print "%d=%s"%(num, word)
...
1=eins
2=zwei
27=siebenundzwanzig
```

Beachtet die `items`-Methode – wenn eure Dictionaries größer werden, empfiehlt sich hier übrigens die Verwendung der `iteritems`-Methode, die ähnlich zu `items` funktioniert, aber wieder den Aufbau einer temporären Liste vermeidet.

Problems

(28.1) Eine Konsequenz der Veränderbarkeit von Listen ist, dass sie nicht als Schlüssel in Dictionaries taugen (warum?). Probiert, ob Tupel Schlüssel in Dictionaries abgeben. **(L)**

29. List Comprehensions

list comprehension
syntactic sugar
Seiteneffekte

Die *list comprehension* hat Python aus der funktionalen Programmiersprache Haskell. Die Idee ist, eine Liste quasi algorithmisch zu beschreiben, angelehnt an die mathematische Notation

$$\{upper(s) \mid s \in L\} :$$

```
>>> l = ["e", "ba", "guU"]
>>> [s.upper() for s in l]
['E', 'BA', 'GUU']
```

Die Liste wird also innerhalb von eckigen Klammern durch eine Iteration und einen Ausdruck beschrieben. Bei Bedarf kann noch eine Selektion dazu kommen:

```
>>> [str(b) for b in range(10) if b>4]
['5', '6', '7', '8', '9']
```

Man kann auch mehrfach iterieren, es ist aber trotzdem nur eine Bedingung zugelassen:

```
>>> [(a,b) for a in range(10)
   for b in range(10) if a==b]
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5),
(6, 6), (7, 7), (8, 8), (9, 9)]
```

Zunächst sieht das nach *syntactic sugar* aus (also wie Spracheigenschaft, die recht einfach auch mit anderen, sozusagen hässlicheren Sprachelementen nachgebildet werden kann).

Tatsächlich kommen list comprehensions aber aus dem funktionalen Programmieren, einer Technik, die ohne Zuweisungen auskommt – in ihr ist ein Programm einfach eine fortgesetzte Anwendung von Funktionen auf die Ergebnisse anderer Funktionen.

Entscheidend dabei ist der Verzicht auf *Seiteneffekte*, d.h. Änderungen am Zustand des Rechners, die nicht durch die Argumente der Funktion und ihren Rückgabewert beschrieben sind. Pythons list comprehensions sind nach dieser Definition nicht exakt funktional, weil sie die Bindung (mindestens) eines Namens bewirken, nämlich der „Schleifenvariablen“:

```
>>> [s.lower() for s in "tokenize me".split()]
['tokenize', 'me']
>>> s
'me'
```

Vorteil: Rein funktionale Programme lassen sich mit dem λ-Kalkül untersuchen. Für imperative Programme (also Abfolgen von Anweisungen) ist der weit kompliziertere Hoare-Kalkül nötig.

Trotz dieser Anmerkung ist es natürlich nicht schwierig, die List Comprehension durch imperative Methoden nachzubauen. So entspricht etwa

```
l = [ expression for expr in sequence1
        for expr2 in sequence2 ...
        for exprN in sequenceN
        if condition]
```

genau

```
l = []
for expr1 in sequence1:
  for expr2 in sequence2:
  ...
    for exprN in sequenceN:
      if (condition):
        l.append(expression)
```

Problems

(29.1)* Angenommen, ein Algorithmus verlange nach der Menge aller Paare natürlicher Zahlen (die Null soll da nicht dabei sein) kleiner 10, bei denen die erste durch die zweite ohne Rest teilbar ist. Wie würdet ihr das mit list comprehensions formulieren? Gießt das in eine Funktion, die ein Argument nimmt, um von der hartkodierten 10 wegzukommen. **(L)**

(29.2) Formuliert die Funktion zur Wandlung einer Datei mit Attribut-Wert-Paaren in ein Dictionary mit list comprehensions um.

30. Montage des Programms

Wir sammeln die einschlägigen Funktionen in einer Datei gener1.py. Damit das ein komplettes Programm wird, schreiben wir ans Ende der Datei:

```
if __name__=="__main__":
  try:
    rules = readRules(sys.argv[1])
  except IOError:
    sys.stderr.write("Couldn't open %s\n"%
      sys.argv[1])
    sys.exit(1)
  generate("S", rules, int(sys.argv[2]))
```

In `__name__` steht der Name, unter dem das Modul importiert wurde. Wenn das Modul direkt aufgerufen wurde, enthält es den String `__main__`. Die Abfrage sorgt dafür, dass der folgende Code nicht ausgeführt wird, wenn das Modul per `import` ausgeführt wird.

`sys.argv` ist eine Liste der Kommandozeilenparameter.

```
examples> cat printargs.py
import sys
print sys.argv
examples> python printargs.py a 43 par par
['printargs.py', 'a', '43', 'par', 'par']
```

`sys.argv[0]` ist der Name des Skripts.

`sys.exit` beendet das Skript und gibt sein Argument an die Shell zurück. Konventionell geben Programme Null zurück, wenn alles ok ist und eine kleine positive Zahl bei Fehlern.

```
examples> gener1.py num.grammar 3
1
2
11
12                  examples> cat num.grammar
111                 V->+
112                 V->-
121                 Z->1
122                 Z->2
21                  S->N
22                  N->Z
211                 N->ZN
212                 S->VN
221                 S->N.N
222                 S->VN.N
+1
+2
+11
+12
+21     -21
```

```
+22     -22
-1      1.1
-2      1.2
-11     2.1
-12     2.2
```

instanzieren

Instanz

Zustand

Problems

(30.1)* Macht euch klar, wann Python welchen Code in eurem Modul ausführt: Funktionen erst dann, wenn sie aufgerufen werden, Code außerhalb der Funktion beim ersten Importieren des Moduls, und das, was durch unsere Bedingung an `__name__` „geschützt“ ist nur dann, wenn das Modul direkt von der Kommandozeile gestartet wurde. Testet das am Besten durch ein `print __name__` außerhalb jeder Funktion. Wird dieser Code auch bei einem reload ausgeführt? **(L)**

(30.2)* Macht euch klar, wie das mit dem sys.argv funktioniert. Erweitert z.B. das printargs-Programm oben so, dass es seine Argumente sortiert, oder macht etwas wie das Unix-Programm `cat` daraus, indem ihr alle Argumente außer dem Nullten als Dateinamen interpretiert und den Inhalt der Dateien auf die Standardausgabe ausgebt.

(30.3) num.grammar erzeugt schon für kleine maximale Längen Unmengen von Wörtern. Versucht, ob ihr euren Rechner mit der Wahl größerer maximaler Längen in die Knie zwingen könnt.

— Dateien zu diesem Abschnitt in der HTML-Version—

31. Klassen

Die Grammatik haben wir bisher als ein Dictionary dargestellt mit einer separaten Funktion zum Einlesen.

Das ist problematisch:

- Unsere Zugiffsmethode basiert fest auf der Implementation als Dictionary von Listen. Was, wenn wir das mal ändern wollen?
- Wenn wir die Implementation ändern, wird die Funktion zum Einlesen vielleicht vergessen und funktioniert dann nicht mehr.
- Wenn wir mehr Sachen mit der Grammatik machen wollen (etwa Plausibilitätsprüfung, Ausgabe, Speichern...), haben wir immer mehr verstreute Funktionen

Abhilfe: Zusammenfassen von Daten und Methoden zu einer Klasse – ähnlich der bekannten String, Dictionary usf.

Bevor wir uns an die doch recht aufwändige Grammatik-Klasse machen, wollen wir zunächst nochmal überlegen, was eigentlich Klassen sind. Eine Klasse ist sozusagen eine Schablone für Objekte; letztere gehen durch *instanzieren* aus Klassen hervor. Die Klasse sagt, was ein Objekt kann (welche Methoden es hat), und sie bestimmt auch weitgehend, was die Identität des Objekts ausmacht (welche Daten nötig sind, um ihm die Identität zu geben). Die verschiedenen Objekte (man spricht auch gern von *Instanz*en der Klasse) haben dann konkrete Werte für diese Daten, haben aber alle das gleiche (durch die Klasse bestimmte) Verhalten. Diese konkreten Werte bezeichnet man auch gern als *Zustand* des Objekts. Eine schöne Art, diese Sachverhalte auszudrücken ist: „Objekte bestehen aus Zustand und Verhalten“.

Ein ganz einfaches Beispiel ist eine Klasse, die einfach eine Identität bekommt (hier einen Namen). Sie soll zunächst nicht mehr können als piepen und pupen.

```
class Tiny:

  def __init__(self, myName):
    self.myName = myName

  def beep(self):
    print self.myName, "beeps"
```

```
  def poop(self):
    print self.myName, "poops"
```

Konstruktor

Attribute

Instanzvariable

Wenn das in tiny.py gespeichert ist, kann man folgendes tun:

```
>>> from tiny import Tiny
>>> t1, t2 = Tiny("Parrot"), Tiny("Rabbit")
>>> t1.beep()
Parrot beeps
>>> t2.poop()
Rabbit poops
```

Zumindest strengt dieses Spiel intellektuell nicht sonderlich an.

Was passiert hier?

- Die Definition einer Klasse wird mit dem reservierten Wort `class` eingeleitet, danach kommt der Name der neuen Klasse (sollte mit einem Großbuchstaben anfangen), ein Doppelpunkt und wieder ein compound statement (Einrückung!)
- Im compound statement werden dann Funktionen definiert, eben unsere Methoden. Sie alle haben ein festes erstes Argument, konventionell `self`. Python trägt darin beim Aufruf eine Referenz auf die Instanz ein, also auf das konkrete Objekt, auf das die Methode wirken soll. Wird etwa `a.foo()` aufgerufen und ist a eine Instanz einer Klasse `A` mit einer Methode `foo(self)`, so übergibt Python eben a als `self`.
- Der Name der ersten Methode, `__init__`, sieht nicht nur komisch aus, er hat auch eine spezielle Funktion: Er ist ein *Konstruktor*, eine Methode, die beim Erzeugen eines Objekts dieser Klasse (dem Instanzieren) aufgerufen wird. In ihr werden die benötigten Datenstrukturen angelegt und ggf. Arbeit erledigt, die vor der Benutzung des Objekts gemacht werden muss. Die speziellen Python-Namen (wir hatten ja schon `__name__` gesehen) spielen bei Klassen eine sehr große Rolle, weil sie uns erlauben werden, das Verhalten der Objekte sehr weitgehend zu beeinflussen.
- In diesem Fall muss der Konstruktor nicht viel tun – er merkt sich lediglich seinen Namen – mehr Identität hat unser einfaches Spielzeug nicht. Er schreibt dazu einen Namen in den Namespace des *Objekts* self, also sozusagen in sich selbst. Da jede Methode dieses self wieder übergeben bekommt, kann auch in allen Methoden auf diese Daten zugegriffen werden.
- Das funktioniert allgemein so: Jedes Objekt hat einen eigenen Namespace. Namen darin heißen *Attribute* des Objekts. Wenn diese Attribute Variablen sind, heißen sie speziell *Instanzvariable* – das betont nochmal, dass sie zur Instanz und nicht zur Klasse gehören. In diesem Namespace stehen aber auch die Namen der Methoden (die in der Regel aus der Klasse übernommen werden).
- Die Methodenaufrufe sehen jetzt genau aus wie von den „eingebauten“ Objekten gewohnt. Beachtet nochmal, dass ihr das erste Argument der Methoden nicht übergebt, weil es Python für euch ausfüllt.

Problems

(31.1)* Fügt der Klasse `Tiny` noch eine Methode `getName` hinzu, die den Namen der Instanz zurückgibt. Das sollte etwa so funktionieren:

```
>>> from tiny import Tiny
>>> t1, t2 = Tiny("Parrot"), Tiny("Rabbit")
>>> t1.getName()
'Parrot'
```

(L)

(31.2)* Die Plüschtiere, die Tiny modelliert, sollen jetzt nicht nur einen Namen haben, sondern auch noch satt und hungrig sein können. Am Anfang soll die Instanzenvariable hunger (die analog zu myName funktioniert) dabei Null sein, danach können die Methoden `eat` und `wait` den Hunger jeweils um eins reduzieren oder erhöhen. Verändert die beep-Methode, so dass sie, wenn das Plüschtier Hunger hat (also `self.hunger>0` gilt), ihrer Ausgabe ein „Hungry“ voranstellt. **(L)**

(31.3)* Überlegt euch, wie ihr die Funktionalität der Klasse der letzten Aufgabe ohne Klassen hinbekommt, also nur mit Funktionen und Variablen, die zwischen ihnen hin- und hergeschoben werden.

(31.4) Überlegt euch, was alles nötig wäre, um aus der Tiny-Klasse einen Tamagotchi zu bauen. Was davon könnt ihr schon, was nicht?

32. Eine erste Grammatik-Klasse

```
class Grammar:
  def __init__(self, sourceName):
    self._readRules(sourceName)

  def _readRules(self, sourceName):
    self.rules = {}
    for rawRule in open(sourceName).readlines():
      if rawRule.startswith("="):
        self._setStartSymbol(rawRule)
      else:
        self._addRule(Rule(rawRule))

  def _addRule(self, rule):
    self.rules.setdefault(rule.getLeft(),
      []).append(rule)

  def _setStartSymbol(self, rawSym):
    self.startSymbol = rawSym[1]

  def getRulesForNonTerm(self, nonTerm):
    return self.rules.get(nonTerm, [])

  def deriveLeftNonTerm(self, nonTerm, word):
    res = []
    for rule in self.getRulesForNonTerm(nonTerm):
      res.append(rule.applyToLeftmost(word))
    return res

  def getStartSymbol(self):
    return self.startSymbol
```

Anmerkungen:

- Der Konstruktor muss hier die Grammatik aus der angegebenen Datei lesen – wir haben diese Funktionalität aber in eine separate Methode verlegt, um den Konstruktor übersichtlich zu halten (was bei komplexeren Klassen manchmal eine echte Herausforderung ist).
- In `_readRules` wird eine Instanzvariable `rules` (korrekt mit self davor, denn sonst würde es die Variable ja nur in `_readRules` geben) auf das leere Dictionary initialisiert, das dann in `_addRules` gefüllt wird.
- Ein paar Methoden haben Underscores vor dem Namen. Das soll andeuten, dass sie nur klassenintern verwendet und nicht von außen aufgerufen werden sollen. Python erzwingt das aber nicht. Namen mit zwei Underscores am Anfang werden von Python in der Tat „versteckt“. Wer nicht versteht, warum man das vielleicht haben möchte, soll sich vorerst keine Sorgen machen.
- Wir haben die alte `readRules`-Funktion aufgespalten in die Methoden `_readRules` und `_addRules`. Das folgt der Logik, dass eine Funktion *eine* Sache tun soll (und die gut). In

diesem Fall ist plausibel, dass das Parsen einer Regelspezifikation gemäß irgendeiner Vereinbarung unabhängig sein sollte vom Hinzufügen einer bereits maschinengemäß repräsentierten Regel zur Regelbasis der Grammatik. Im Zweifel sollte man Methoden lieber weiter aufspalten – wenn man testen will, ob zwei Aktionen in einer Methode bleiben dürfen, kann man sich überlegen, ob mindestens eine der Aktionen mit einer dritten sinnvoll kombinierbar wäre. Im vorliegenden Fall wäre es beispielsweise denkbar, dass ein zweites Grammatikobjekt einer Grammatik bereits geparste Regeln hinzufügen möchte oder wir Regeln auch in einem anderen Format verstehen wollen.

- In dem Dictionary steht jetzt auch kein nackter String mehr, sondern Objekte einer neuen Klasse, `Rule` (die wir natürlich auch noch schreiben müssen).
- Der alte Dictionary-Zugiff wird ersetzt durch eine Methode `getRulesForNonTerm`, was klarer macht, was der Zugriff eigentlich bedeutet.
- Natürlich sollte auch hinter der Zeile, die die Klassendefinition einleitet, ein Docstring stehen, der grob umreißt, was die Klasse soll und was sie kann.

Inhaltlich haben noch eine Veränderung vorgenommen: In der Grammatikbeschreibung kann man jetzt nach den Regeln der Kunst auch ein Startsymbol angeben, und zwar in einer Zeile, in der nur ein = und danach das Startsymbol steht. Natürlich sollte `_setStartSymbol` etwas besser aufpassen, was es da eigentlich vorgesetzt kriegt und ggf. Fehler auslösen, aber das soll uns jetzt noch egal sein.

Zum Setzen der Instanzvariable `startSymbol` stellen wir eine Methode `getStartSymbol` zur Verfügung. Der Underscore vor `_setStartSymbol` sagt wie oben vereinbart: Klienten sollen diese Methode nicht verwenden (und in der Tat würde sie auch etwas anderes tun als ihr Name die Klienten erwarten lassen würde).

Problems

(32.1) Vergleicht (soweit möglich) die Methoden unserer Grammar-Klasse mit den entsprechenden Funktionen aus gener1.py. Was hat sich geändert, was ist geblieben?

33. Eine Regel-Klasse

Wir definieren noch unsere `Rule`-Klasse:

```
class Rule:
  def __init__(self, rawRule):
    self._parse(rawRule)

  def __str__(self):
    return "%s -> %s"%(self.left, self.right)

  def __repr__(self):
    return "Rule('%s')"%str(self)

  def _parse(self, rawRule):
    try:
      self.left, self.right = [s.strip() for s in
        rawRule.split("->")]
      if not self.left or not self.right:
        raise ValueError
    except ValueError:
      raise ValueError(
        "Syntaxfehler in %s"%rawRule)

  def getLeft(self):
```

```
    return self.left

  def getRight(self):
    return self.right

  def applyToLeftmost(self, word):
    return word.replace(self.left, self.right, 1)
```

Anmerkungen:

- Wir haben zwei weitere spezielle Namen neben init: `__str__` soll einen String zurückgeben, der für den Menschen schön anzusehen ist (wird z.B. bei `print` verwendet oder auch von der `str`-Funktion), `__repr__`(für Repräsentation), soll dagegen tiefere Einblicke in das Objekt liefern, wenn möglich sogar Python-Code, der zum Neu-Erzeugen des Objekts verwendet werden könnte.
- Für die Instanzvariablen, die wir zugänglich machen wollen, haben wir Zugriffs- oder *Akzessorfunktion*en (`getRight` und Freunde). Das ist empfohlene Praxis, auch wenn in Python niemand gehindert wird, trotzdem auch lesend oder schreibend auf Attribute von Objekten zuzugreifen oder gar von außen neue Attribute anzulegen: Python-Objekte haben, im Gegensatz zu Sprachen etwa aus der Familie von Java und C++, keinerlei besondere Kontrolle über ihre Namensräume. Das kann manchmal angenehm sein, führt aber bei Missbrauch meistens zu chaotischem und unwartbarem Code. Daher: Wenn ihr wollt, dass eine Instanzvariable von außen lesbar ist, schreibt eine Methode `get<varname>`, wollt ihr, dass sie schreibbar ist, schreibt eine Methode `set<varname>`. Und schreibt vor, dass von außen *nie* direkter Zugriff auf Instanzvariablen erlaubt ist. Macht es auch bei Objekten anderer Leute nicht, es sei denn, der/die AutorIn der Klasse würde euch explizit dazu auffordern.
- Wieder ist die Klasse so geschrieben, dass der Konstruktor „rohe“ Daten bekommt, sie veredelt und das Produkt dem Objekt zur Verfügung stellt.
- Die Methode `applyToLeftmost` abstrahiert die Anwendung einer Regel – eine der größten Vorteile des objektorientierten Programmierens ist, dass Verhalten explizit gemacht werden kann. Eine Regel kann jetzt einfach „angewendet“ werden, wir sagen also konkret, was die Wirkung einer Aktion in der Anwendung ist. Ganz egal, wie wir eine Regel repräsentieren, in jedem Fall muss sie angewendet werden können.

Noch etwas mehr zur Frage der Akzessorfunktionen: Ein entscheidender Vorteil der Vorschrift, Instanzvariablen nur über Akzessorfunktionen zugänglich zu machen, ist eine Entkopplung zwischen der Implementation einer Klasse (welche Instanzvariablen ich dazu brauche, ist meist eine Implementationsentscheidung) und ihrer Schnittstelle. Diese Entkopplung will man haben, weil mit ihr einzelne Bestandteile eines Programms getrennt implementiert und weiterentwickelt werden können – solange die Schnittstellen sich nicht ändern, sollte sich an der Semantik des Programms nichts ändern.

Hinzu kommt, dass sich während der Weiterentwicklung eines Programms häufig die Notwendigkeit ergibt, beim Setzen einer Instanzvariable zusätzliche Aktionen durchzuführen. Bei unserem `Toy`-Beispiel könnte durch das Setzen einer Instanzvariable `hunger` indirekt der Wert einer anderen Instanzvariable `mood` beeinflusst werden. In einer Akzessorfunktion können wir dafür sorgen, dass der Zustand des Objekts konsistent bleibt, indem wir ggf. `mood` mitaktualisieren. Hätten wir den direkten Schreibzugriff auf `hunger` erlaubt, könnte man von außen inkonsistente Zustände erzeugen.

Auch der Lesezugriff ist nicht unkritisch, vor allem, wenn sich die Implementation ändert. So könnte es beispielsweise sein, dass wir `hunger` irgendwann nicht mehr direkt speichern, sondern aus, beispielsweise, `stomachFill` und `glucoseLevel` berechnen. Eine `getHunger`-Methode kann diese innere Änderung nach außen hin maskieren und dafür sorgen, dass die Klienten ungeändert weiter funktionieren, ein direkter lesender Zugriff auf `hunger` würde dagegen scheitern (alternativ müssten wir eben immer die Instanzvariable `hunger` mitführen, obwohl wir sie eigentlich aus anderen Parametern berechnen können, und auch das wäre nicht schön.

(Weiterführend:) Die mit Python 2.2 eingeführten Deskriptoren erlauben eine gewisse Kontrolle über das, was mit existierenden Einträgen in die Namensräume passieren kann – mit Deskriptoren

und new-style classes könnte man also direkten Zugriff auf Attribute zulassen, ohne sich über die zukünftige Entwicklung Sorgen machen zu müssen. Ich werde in diesem Skript allerdings fast nichts zu diesen Themen sagen, und vorläufig sind sie so esoterisch, dass ihr euch nicht darum kümmern wollt.

Der Code für `Grammar` und `Rule` möge in `grammarsimple.py` stehen. Dann:

```
>>> import grammarsimple
>>> r = grammarsimple.Rule("A->BC")
>>> r
Rule('A -> BC')
>>> print r
A -> BC
>>> r.applyToLeftmost("aaA")
'aaBC'
```

Im gener-Programm muss sich an generate nichts ändern (obwohl es natürlich nett wäre, `rules` jetzt etwa `grammar` zu nennen), `deriveLeft` wird einfacher, weil die Arbeit, die von Grammatik oder Regeln gemacht werden sollte, auch dort gemacht wird:

```
def deriveLeft(curWord, grammar):
  leftmostNonTerm = getLeftmostNonTerm(curWord)
  return grammar.deriveLeftNonTerm(
    leftmostNonTerm, curWord)
```

Das „Hauptprogramm“ könnte dann sein:

```
if __name__=="__main__":
  gram = Grammar("grammar")
  generate(gram.getStartSymbol(), gram, 5)
```

Problems

(33.1)* Sammelt alle nötigen Bausteine für einen kompletten Generierer auf der Basis unserer neuen Klassen aus den hier vorgestellten Bausteinen und den alten Funktionen zusammen (wohl am besten im grammarsimple.py-Modul). Kommentiert die einzelnen Klassen und Funktionen (für Doctests gibts extra Lob). Überzeugt euch, dass das Programm immer noch das tut, was es soll, etwa durch Vergleich der Ausgaben von gener1 und dem neuen Programm (hochtrabend könnte man das einen „Regressionstest“ nennen).

Vergesst beim Testen nicht, dass die Grammatikdateien jetzt um das Startsymbol (also in der Regel eine Zeile „=S“) ergänzt werden müssen, sonst wird euer Programm, wie es jetzt ist, etwas wie „AttributeError: Grammar instance has no attribute startSymbol “ sagen (was zugegebenermaßen für Leute, die die Grammatiken schreiben, keine hilfreiche Fehlermeldung ist – wie ließe sich das am einfachsten verbessern?)

(33.2) Der Umstand, dass Namespaces von Objekten beliebig schreibbar ist, kann manchmal auch praktisch sein. Folgendes könnte z.B. bestimmte Anwendungen von Dictionaries abdecken:

```
>>> class Empty:
...     pass
...
>>> e = Empty()
>>> e.bla, e.rotz = "eins", "zwei"
>>> e.bla
'eins'
```

Denkt euch aus, wie ihr rauskriegt, ob diese Sorte von „emuliertem“ Dictionary oder Dictionaries selbst schneller sind. Überlegt euch, unter welchen Umständen man sowas vielleicht machen wollen könnte. **(L)**

34. Vererbung

Vererbung
Basisklasse
Unterklassen

Bisher haben wir nur Großbuchstaben als Nichtterminale und andere Zeichen als Terminale. Wir wollen aber richtige Sätze haben und `NP` und `VP` als Nichtterminale.

Abhilfe: Wir definieren Klassen, die als Symbole fungieren.

Beachtet, dass das auch gleich bedeutet, dass wir unsere Wörter nicht mehr als Strings modellieren können – Wörter sind Sequenzen von Symbolen, und wenn unsere Symbole keine einfachen Zeichen mehr sind, können die Wörter keine Strings (also Sequenzen einfacher Zeichen) mehr sein. Diese Änderung wird uns also noch viel Arbeit machen.

Terminale und Nichtterminale haben gemeinsame Eigenschaften, unterscheiden sich aber in Details. Dafür gibt es *Vererbung*: Wir definieren eine *Basisklasse* `Symbol` mit den gemeinsamen Eigenschaften und leiten daraus zwei *Unterklassen* ab.

```
class Symbol:
  def __init__(self, rawStr):
    self.content = rawStr

  def getContent(self):
    return self.content

  def __str__(self):
    return self.content

  def __cmp__(self, other):
    try:
      return cmp(self.content, other.content)
    except AttributeError:
      return -1

class Terminal(Symbol):
  def __repr__(self):
    return "Terminal('%s')"%self.content

  def __hash__(self):
    raise TypeError("NonTerminal not hashable")


class NonTerminal(Symbol):
  def __repr__(self):
    return "NonTerminal('%s')"%self.content

  def __hash__(self):
    return hash(self.content)
```

In der Klassendefinition einer Subklasse steht also hinter ihrem Namen der Name der Basisklasse in Klammern. Wenn eine Methode aufgerufen wird, die in der Unterklasse nicht vorhanden ist, wird sie in der Basisklasse gesucht – das geht sogar für `__init__` und die anderen magischen Namen. Ähnliches gilt für andere Attribute.

Wir haben hier zwei weitere magische Namen: `__cmp__`, das zwei Argumente nimmt und bestimmt, wie der Vergleich zwischen zwei Objekten ausgeht, und `__hash__`, das eine Art Kennung des Objekts zurückgibt, die zum Beispiel verwendet wird, wenn ein Objekt als Schlüssel in einem Dictionary verwendet wird.

Wir müssen das hier definieren, weil wir wollen, dass Symbole, die beispielsweise das Wort „Himmel“ enthalten, gleich sind, selbst wenn sie nicht der gleiche Wert sind (also das gleiche Objekt darstellen). Ohne weiteres würden die Objekte nach den Speicheradressen verglichen, an denen sie gespeichert sind, was hier natürlich sinnlos ist. Ein kleines Problem ist bei dem hier

gewählten Ansatz, dass `Terminal("Himmel")==NonTerminal("Himmel")` – aber das stört uns nicht.

Die eingebaute Funktion `cmp`, die in unserer `__cmp__`-Methode verwendet wird, vergleicht übrigens ganz schlicht ihre beiden Argumente und gibt -1 (arg1 $<$ arg2), 0 (arg1 $==$ arg2) oder 1 zurück.

`Terminal` löst eine Exception aus, wenn es nach einem Hash gefragt wird. Wir wollen `Terminals` vorläufig nicht als Schlüssel in einem Dictionary haben und gehen auf Nummer sicher.

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(34.1)* Um der Vererbung etwas näher zu kommen, kann man sich Spielzeug vorstellen. Alle Spielzeuge haben natürlich einen Namen. Definiert also zunächst eine Klasse `Toy`, die mit einem Namen konstruiert wird und eine Methode `getName` hat.

Jetzt soll es Spielzeuge geben, die zusätzlich piepen können. Definiert eine Klasse `BeepingToy`, die von Toy erbt, aber zusätzlich eine Methode `beep` hat, die analog der von `Tiny` funktioniert.

Andere Spielzeuge können grummeln – definiert entsprechend eine Klasse `GrowlingToy`, die von Toy erbt und eine Methode `growl` hat, bei deren Aufruf eine Nachricht ausgegeben wird, das Spielzeug mit entsprechenden Namen würde growlen.

Wir entdecken erst jetzt, dass Kinder ihren Spielzeugen manchmal neue Namen geben. Wie können wir diese Entdeckung verarbeiten? Was müssten wir tun, wenn Spielzeuge auch kaputt gehen können sollen? **(L)**

(34.2)* Klassen können Methoden ihrer Oberklassen überschreiben. In der Tat machen das unsere Symbole schon, denn natürlich hat schon `Symbol` ein `__repr__`, wenn auch nur implizit.

Im Spielzeugbeispiel kann man das etwas expliziter machen: Natürlich mit allen Spielzeugen spielen, aber die konkrete Natur des Spielen hängt vom Spielzeug ab. Definiert also eine Methode `Toy.play`, die vielleicht nur sagt, es sei langweilig (ein generisches Spielzeug ist nicht so spannend...). Definiert dann noch `play`-Methoden der Unterklassen, die etwa einfach `beep` bzw. `growl` aufrufen.

So etwas ist eine einfache Form des so genannten Polymorphismus – das Programm entscheidet sich bei Anwendung ein und derselben Operation (hier ein Methodenaufruf) in Abhängigkeit von den Operatoren (hier die Instanz, für die die Methode aufgerufen wird) für die Ausführung verschiedenen Codes (hier eben die verschiedenen Methoden). Das ist für uns nichts Neues, denn viele Operatoren und Funktionen von Python sind sozusagen „von Natur aus“ polymorph. In Sprachen mit statischer Typprüfung (Java, C++ und Freunde) ist Polymorphismus hingegen viel aufregender. **(L)**

35. Introspektion

Wörter müssen wir jetzt als Folgen von Symbolen darstellen. Ein Wort soll entweder aus einem String (in einem noch zu definierenden Format) oder aus einem anderen Wort (das dann kopiert wird) erzeugbar sein, der Konstruktor muss also beides können.

```
class Word:
  def __init__(self, word):
    self.symList = []
    if isinstance(word, Word):
      self.symList = word.symList[:]
    else:
      self._parseRaw(word)
  ...
```

Der vollständige Quellcode für die Klasse findet sich im Anhang der Seite.

Die eingebaute Funktion `isinstance` prüft, ob ihr erstes Argument (im Regelfall eine Variable) eine Instanz der zweiten (im Regelfall eine Klasse) ist.

Damit:

```
>>> from Word import Word
```

```
>>> w = Word("Do Re Mi")
>>> w2 = Word(w)
>>> import Symbol
>>> w2.append(symbol.NonTerminal("ba"))
>>> w, w2
(Word('Do Re Mi'), Word('Do Re Mi ba'))
```

Introspektion

first class

Das Operieren mit Namen und Typen zur Laufzeit heißt auch *Introspektion*. In Python wirkt das trivial, weil auch Klassen (und nicht nur Objekte) an Funktionen übergeben werden können (Dinge, die an Funktionen übergeben und von Funktionen zurückgegeben werden können, nennt man gerne *first class*).

(Weiterführend:) In kompilierten Sprachen ist Introspektion meistens eher kompliziert, weil bereits der Compiler die Namen verarbeitet und Typen, Klassen und Funktionen in maschinengerechte Form verpackt. Im Kompilat (das ja nur Maschinencode ist) sind im Regelfall diese Informationen nicht mehr (oder nur sehr indirekt) vorhanden und müssen, wenn man Introspektion treiben möchte, sozusagen künstlich wieder eingebaut werden.

Die Funktion `type` kennen wir schon. Für selbst definierte Klassen gibt type immer `instance` zurück.

Für eingebaute Typen sollte man (noch) auf `type` zurückgreifen. Das könnte so aussehen:

```
examples> cat verbosetype.py
class _Empty: pass
_typedict = {type([]): "e Liste",
  type(""): " String", type(1): "e ganze Zahl",
  type(_Empty()): " Objekt"}

def verboseType(ob):
  return "ein%s"%_typedict.get(type(ob),
    " anderes Ding")
```

und könnte so benutzt werden:

```
>>> from verbosetype import verboseType
>>> verboseType(4)
'eine ganze Zahl'
>>> verboseType("bla")
'ein String'
>>> verboseType(verboseType)
'ein anderes Ding'
```

(Weiterführend:) Bis Python 2.1 waren die eingebauten Typen keine richtigen Klassen. Das hat sich in 2.2 geändert, mit dem Preis der zusätzlichen Konfusion, dass es „alte“ und „neue“ Klassen gibt. Die neuen Klassen erben alle von der Klasse `object`. Die eingebauten Funktionen `str` und Freunde sind inzwischen eigentlich Klassen (die Aufrufe sind also Aufrufe von Konstruktoren).

In Python 2.0 etwa ist noch:

```
>>> isinstance("bla", str)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: second argument must be a class
```

Ab Python 2.2 ist:

```
>>> isinstance("bla", str)
1
```

Da Python zwei Sorten von Strings hat (probiert `type("")` und `type(u"")`), tritt beim Testen auf Stringsein übrigens noch eine kleine Komplikation hinzu – man möchte auf die gemeinsame Basisklasse von `str` und `unicode` testen. Diese gemeinsame Basisklasse heißt `basestring`, und das erklärt das oben gegebene Rezept.

Man kann auch von eingebauten Klassen erben. So kann man ein Dictionary, das für unbekannte Schlüssel einfach `None` zurückgibt, so schreiben:

```
class DictWithNone(dict):
  def __init__(self, *args):
    dict.__init__(self, args)

  def __getitem__(self, key):
    if self.has_key(key):
      return dict.__getitem__(self, key)
    return None
```

Kopie

aliasing

(was zugegebenermaßen ein paar Dinge tut, die wir noch nicht können und auch nicht ganz die beste Art ist, sowas zu machen...)

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(35.1)* Überzeugt euch anhand des Toy-Beispiels, dass ininstance schlau genug ist, um mit Vererbung zurechtzukommen:

```
>>> t = BeepingToy("Fred")
>>> isinstance(t, Toy)
```

usf.

(35.2)* Seht nach, was `type` bei euch zurückgibt, wenn ihr ihm `len`, `range` oder eine selbst definierte Funktion übergebt. Was gibt sie bei `str`, `dict`, `list` usf., was bei `open` zurück? **(L)**

(35.3) Schreibt eine Funktion `addTo(number, whatever)->float`, die „irgendwas“ auf eine Zahl addiert. Ist whatever ein integer oder ein float, ist klar, was passieren muss, ist es ein String, würde man wahrscheinlich versuchen, ihn in einen Float zu wandeln. Bei einer Sequenz könnte man z.B. die Länge der Sequenz addieren, bei einer Datei vielleicht die Länge des von read zurückgegebenen Strings. Wie ihr das macht, ist egal, es kommt nur drauf an, dass ihr möglichst viele Typen verarbeiten könnt. **(L)**

36. aliasing

Im Entwurf des Konstruktors für `Word` oben stand

```
self.symList = word.symList[:]
```

Der „Alles-umfassende-Slice“ sorgt dafür, dass wir eine *Kopie* der Liste erzeugen. Bei einfacher Zuweisung hätten wir einen klassischen Fall von *aliasing* – die beiden Word-Instanzen würden sich eine `symList` teilen, Änderungen am einen Wort würden Änderungen am anderen nach sich ziehen.

```
>>> l1 = range(5)
>>> l2 = l1
>>> l3 = l1[:]
>>> l1.append("ha!")
>>> l1,l2,l3
([0, 1, 2, 3, 4, 'ha!'], [0, 1, 2, 3, 4, 'ha!'],
  [0, 1, 2, 3, 4])
```

Über aliasing muss man sich bei veränderbaren Datenstrukturen (Listen, Dictionaries) Gedanken machen, für Strings, Tupel und Zahlen kann aliasing nie ein Problem sein.

Schlimmer:

```
>>> l = [range(2)]*3
>>> l
[[0, 1], [0, 1], [0, 1]]
>>> l[0].append(4)
>>> l
[[0, 1, 4], [0, 1, 4], [0, 1, 4]]
>>> l1 = l[:]
```

shallow

```
>>> l.append("bla")
>>> del l[0][0]
>>> l1, l
([[1, 4], [1, 4], [1, 4]], [[1, 4], [1, 4],
  [1, 4], 'bla'])
```

Anmerkung: Mit `del` kann man generell beliebige Referenzen (und nicht nur, wie oben erwähnt, Namen) löschen. Tatsächlich ist die Verwendung beim Entfernen von Elemente aus Listen oder Dictionaries viel typischer als der Einsatz zur Manipulation von Namensräumen.

Das erzeugen einer Kopie durch Slices ist *shallow*: Es werden keine Kopien der darunterliegenden Datenstruktur gemacht, sondern nur die eigentliche Sequenz kopiert. Das rekursive, „tiefe“ Kopieren ist im allgemeinen ein ziemlich kompliziertes Problem. Python bietet dafür (und für Kopien von Nicht-Sequenzen) ein Modul `copy` an:

```
>>> l = []
>>> for i in range(3): l.append(range(2))
...
>>> l
[[0, 1], [0, 1], [0, 1]]
>>> l[0].append(4)
>>> l
[[0, 1, 4], [0, 1], [0, 1]]
>>> import copy
>>> l2 = copy.deepcopy(l)
>>> l[1].append(4)
>>> l, l2
([[0, 1, 4], [0, 1, 4], [0, 1]], [[0, 1, 4],
  [0, 1], [0, 1]])
```

Moral: Wer mit veränderbaren Objekten umgeht, muss gut verstanden haben, wie das mit den Werten und den Referenzen funktioniert.

Euer Freund beim Untersuchen von aliasing ist die in Aufgaben schon öfter verwendete eingebaute Funktion `id`. Sie gibt eine Zahl aus, die jedes Objekt eindeutig identifiziert (d.h. gibt id verschiedene Zahlen zurück, gehen die Referenzen auf verschiedene Werte, gibt es gleiche Zahlen zurück, gehen die Referenzen auf gleiche Werte). Im Beispiel oben könnt ihr gleich sehen, was schief geht:

```
>>> l = [range(2)]*3
>>> map(id, l)
[1076954988, 1076954988, 1076954988]
```

Außerdem gibt es noch den bool'schen Operator `is`. Er ist vergleichbar mit ==, vergleicht aber, ob die Werte *identisch* und nicht einfach nur gleich sind, ob also ihre ids gleich sind:

```
>>> a = range(2)
>>> b = range(2)
>>> c = a
>>> a is b, a is c
(False, True)
```

Häufig sieht man in Python-Code das Idiom

```
if a is None:
  ...
```

oder ähnliches. Hier geht es weniger darum, die Identität von a zu prüfen (None gibt es in Python nur einmal, und ihr könnt None auch nicht kopieren – probiert es!), als vielmehr um einen geringen Geschwindigkeitsgewinn. Der `is`-Operator ist schneller als == (warum wohl?). In diesem Fall ist der Geschwindkeitsgewinn allerdings recht marginal. Im Fall großer Listen mag es allerdings schon viel ausmachen, nur dass man dann eigentlich meistens an Gleichheit und nicht an Identität interessiert ist. Fazit: `is` ist für Menschen, die den Referenzenkram sicher verstanden haben, für alle anderen ist es erstmal verzichtbar.

Problems

(36.1)* Probiert die Beispiele auf dieser Seite durch, bis ihr verstanden habt, was vorgeht. Benutzt `id`, wo nötig. Malt euch Bilder der Python-Universen (Vorsicht: in manchen Beispielen gibt es Referenzen aus dem Python-Universum ins Python-Universum und nicht nur aus dem Namensraum ins Universum rein).

37. Mehr zur Wort-Klasse

Zurück zur Wortklasse: Wie bei einer Sequenz soll len die Anzahl der Elemente zurückgeben:

```
  def __len__(self):
    return len(self.symList)
```

Wir wollen, dass `Word` weiß, ob es nur aus Terminalen besteht:

```
  def isAllTerminal(self):
    for s in self.symList:
      if isinstance(s, symbol.NonTerminal):
        return 0
    return 1
```

Wir bauchen das am weitesten links stehende Nichtterminalsymbol:

```
  def getLeftmostNonTerm(self):
    for s in self.symList:
      if isinstance(s, symbol.NonTerminal):
        return s
```

Wir wollen ein neues Wort, in dem ein Nichtterminalsymbol durch ein anderes Wort ersetzt ist, erzeugen:

```
  def replaceLeftmost(self, nonTerm, repl):
    applyPos = self.symList.index(nonTerm)
    newWord = Word(word=self)
    newWord.symList[applyPos:applyPos+1
      ] = repl.symList
    return newWord
```

Weist man einem Listenelement einen Wert zu, hat das `Element` den neuen Wert. Wir wollen in `replaceLeftmost` aber ein Element durch mehrere neue Elemente ersetzen:

```
>>> l=range(8)
>>> l[4]=[-4,-5,-6]
>>> l
[0, 1, 2, 3, [-4, -5, -6], 5, 6, 7]
>>> l=range(8)
>>> l[4:5]
[4]
>>> l[4:5]=[-4,-5,-6]
>>> l
[0, 1, 2, 3, -4, -5, -6, 5, 6, 7]
```

Beachte: `l[4]` ist eine Zahl, `l[4:5]` ist eine Liste.

Das vollständige `word`-Modul ist im Anhang der Seite zu finden.

Problems

Reguläre Ausdrücke
regular expressions
Metacharacters
Metazeichen

(37.1)* Geht das Beispiel mit der Zuweisung zu slices und Elementen noch einmal genau durch. Könnt ihr auf dieser Basis eine Funktion `prepend(targetList, srcList) -> None` schreiben, die srcList vor das erste Element von targetList schiebt, also etwa so:

```
>>> l = range(5)
>>> prepend(l, range(2))
>>> l
[0, 1, 0, 1, 2, 3, 4]
```

(L)

38. Reguläre Ausdrücke I

Unsere Grammatiken müssen jetzt auch anders aussehen – wir haben keine automatische Unterscheidung von Terminal- und Nichtterminalsymbolen. Vereinbarung: Terminalsymbole sollen jetzt in doppelten Anführungszeichen stehen, Nichtterminale sind „nackt“:

```
NP -> n
n -> "dog"
```

Spätestens jetzt wird es mit Bordmitteln schwierig. Abhilfe: *Reguläre Ausdrücke* bzw. *regular expressions* (RE).

In Python stehen Routinen für REs im Modul `re`.

Reguläre Ausdrücke sind eine Sprache zur Beschreibung von Zeichenfolgen.

Die einfachsten regulären Ausdrücke sind „normale“ Zeichen, der Punkt (ein beliebiges Zeichen) und Mengen (in eckigen Klammern). Dabei ist ein Zeichen normal, wenn es nicht speziell ist. Spezialzeichen (*Metacharacters*, *Metazeichen*) in Pythons regulären Ausdrücken sind `. ^ $ * + ? [ ] \ | ( )`.

In gewisser Weise lernt ihr mit den regulären Ausdrücken eine Sprache innerhalb der Sprache Python. Zwar ist diese Sprache bei weitem weniger mächtig als Python selbst (was heißen soll, dass man mit dieser Sprache weniger „Probleme lösen“ kann als in Python – klarer wird das in der Vorlesung über formale Grundlagen der Linguistik), aber fürs Zerstückeln von Texten gerade deswegen unwahrscheinlich praktisch.

(Weiterführend:) Wer schon formale Grundlagen gehört hat, wird ahnen, dass reguläre Ausdrücke im Groben reguläre Sprachen beschreiben (tatsächlich sind Pythons reguläre Ausdrücke allerdings erheblich mächtiger), während Python-Programme natürlich im Prinzip alle allgemeinen Regelsprachen beschreiben können (es ist nicht schwierig, eine Turingmaschine in Python zu schreiben).

Ein erstes Beispiel für diese einfachen REs:

```
>>> from re import search as s
>>> s("m", "abc"),s("a", "abc").group(0)
(None, 'a')
>>> s(".", "abc").group(0)
'a'
>>> s("[A-Z]", "abc"),s("[cde]", "abc").group(0)
(None, 'c')
```

Man sucht also mit der Funktion `re.search` im zweiten Argument nach Entsprechungen („matches“) für den regulären Ausdruck im ersten Argument. Die Funktion gibt None oder ein Match-Objekt zurück; im zweiten Fall gibt die group-Methode des Ergebnisses mit dem Argument 0 zurück, auf welchen Substring des zweiten Arguments der reguläre Ausdruck gepasst („gematcht“) hat.

Die Schreibweise `from x import y as z` übrigens setze ich hier zur Platzersparnis ein – ich habe auf dieser Folie nicht genug Platz, um search immer auszuschreiben. Für euch gilt: Verwendet

sowas nicht, bevor ihr etwas mehr Erfahrung habt, wozu sowas gut sein könnte – Einsparung von Tipparbeit ist jedenfalls im Allgemeinen kein guter Grund, Objekte einfach so umzubenennen.

Daraus lassen sich durch | (verodern), * (null oder mehr Vorkommen), + (ein oder mehr Vorkommen), ? (null oder ein Vorkommen) neue Ausdrücke bauen:

```
>>> s("ab|re", "abc").group(0),s("ab|re",
...   "reg").group(0)
('ab', 're')
>>> s("a*b", "aaab").group(0),s("a*bb+", "aaab")
('aaab', None)
```

Mit Klammern lassen sich Gruppen bauen:

```
>>> s("(ab)+", "abba").group(0),s("(ab)+",
...   "abab").group(0)
('ab', 'abab')
```

^ und $ markieren Anfang und Ende:

```
>>> s("^b", "abb"), s("b$", "abb").group(0)
(None, 'b')
```

Unsere Terminalsymbole folgen dem regulären Ausdruck `^"[^"]*"$`. In `[]` werden Mengen von Zeichen angegeben, ^ am Anfang einer Menge hat die spezielle Bedeutung „alle außer die angegebenen Zeichen".

```
>>> (s('^"[^"]*"$', '"Trm"').group(),
  s('"[^"]"', 'NT').group())
("Trm", None)
```

Noch ein bisschen mehr zu Mengen von Zeichen: `[abc]` heißt also eines der Zeichen a, b oder c. Man kann auch „Ranges" bilden: `[A-Z]` sind alle Zeichen zwischen A und Z, `[A-Za-z]` alle „normalen" Klein- oder Großbuchstaben (Umlaute oder das scharfe s sind da nicht dabei, sie wären etwa durch `[A-Za-zÄÜÖüöäß]` erfasst – aber für sowas werden wir später bessere Methoden kennenlernen). Wenn der Bindestrich in der Menge sein soll, kann er an den Anfang oder ans Ende der Menge gestellt werden (sonst würde er einen Range kennzeichnen).

Ein Caret an erster Stelle wählt das Komplement der angegebenen Menge, matcht also alle Zeichen *außer* den angegebenen. `[^0-9]` sind alle Zeichen außer Ziffern, `[^:.;,()-!]` wäre etwas wie „alles außer Satzzeichen" – man sieht hier auch, dass Metazeichen (z.B. der Punkt) in Mengen normalerweise keine spezielle Bedeutung haben. Das Caret an einer anderen als der ersten Position entspricht einem literalen Caret. `[-^]` matcht also entweder einen Bindestrich oder ein Caret.

Will man Metazeichen literal matchen, muss man in der Regel Backslashes vor das Metazeichen oder es einzeln in eine Menge schreiben. `r"\*"` oder `"[*]"` matchen beispielsweise einen Stern, `\(` oder `[(]` eine öffnende Klammer.

Zum Selbststudium geeignet ist Andrew Kuchlings Python RE Howto[17].

Problems

(38.1)* Denkt euch reguläre Ausdrücke aus, die folgende Dinge matchen (in Python-Syntax, für whitespace könnt ihr vorerst Leerzeichen nehmen):

1. Python-Namen
2. Zuweisungen zu Namen, die mit einem Großbuchstaben anfangen
3. Den Kopf einer Klassendefinition (also `class Foo(Bar):`)
4. Den Kopf einer Funktionsdefinition
5. Zuweisungen konstanter Zahlen

Hinweis: Viele Editoren (allen voran vi und emacs) können auch reguläre Ausdrücke, die sich subtil von Pythons REs unterscheiden. Wenn ihr diese Unterschiede kennt, könnt ihr diese Aufgabe auch einfach interaktiv im Editor probieren. **(L)**

[17] `http://www.amk.ca/python/howto/regex/`

(38.2)* Schreibt ein Programm firstmatch.py, das im ersten Kommandozeilenargument einen regulären Ausdruck und im zweiten einen Dateinamen nimmt. Die Ausgabe soll entweder nichts sein, wenn der reguläre Ausdruck in der bezeichneten Datei nicht matcht, oder der erste Match innerhalb der Datei.

match objects

Character Class

Hinweis: Einige der Metazeichen regulärer Ausdrücke werden von den üblichen Shells selbst interpretiert. Ihr solltet die regulären Ausdrücke also zumindest unter den üblichen Unix-Shells immer in einfache Anführungszeichen schreiben (Generalausnahme: Man weiß, was man tut):

```
examples> python firstmatch.py "def[^:]*:" firstmatch.py
def firstMatch(regExp, text):
```

(L)

(38.3) Erweitert das Programm aus der letzten Aufgabe so, dass nach dem regulären Ausdruck beliebig viele Dateinamen kommen können. Verändert es so, dass es alle Matches ausgibt (ein Programm, das das tut, gibt es übrigens auf Unix-Systemen schon – es heißt grep).

39. Reguläre Ausdrücke II

`re.search` gibt *match objects* zurück (ebenso übrigens das verwandte `re.match`, das von vorneherein nur nach Matches am Anfang des Strings sucht, so dass man `re.match(regexp, s)` fast mit `re.search("^"+regexp, s)` identifizieren kann). Diese Objekte haben unter anderem folgende Methoden:

- `start([group])` – Gibt die Position des Anfangs des Matches zurück; `group` ist, wenn nicht angegeben, 0, wodurch der ganze Match angesprochen wird, kann aber auch eine der mit Klammern definierten Gruppen ansprechen.
- `end([group])` – Wie `start`, nur wird die Endposition zurückgegeben
- `group([group1,...])` – gibt den Inhalt der Gruppe(n) zurück, 0 als Argument heißt wieder ganzer Match.

```
>>> mat = re.search(r'(\w+)\s*->\s*"([^"]*)"',
...     ' bla->  "gaga"')
>>> mat.start(1), mat.end(1)
(1, 4)
>>> mat.group(1), mat.group(2)
('bla', 'gaga')
>>> mat.group(0)
'bla->  "gaga"'
>>> mat.group(1,2)
('bla', 'gaga')
```

`\w` und `\s` sind *Character Class*es sie stehen für „Alle Buchstaben“ bzw. „Alles, was ein Leerzeichen sein kann“. Es gibt noch etliche andere character classes dieser Art, vgl. Dokumentation des re-Moduls.

Wir verwenden hier außerdem raw strings – ihr erinnert euch, dass in raw strings keine Escapesequenzen ausgewertet werden, und das ist genau das, was wir hier haben wollen: Die Maschinerie von Regulären Ausdrücken muss das `\w` sehen, und deshalb sollte es besser nicht bereits von Pythons Maschinerie zur Bewertung von String-Literalen gefressen werden. Nebenbei: Für `\w` würde das ohnehin nicht passieren, weil es für Python selbst keine Bedeutung hat, für Sequenzen wie `\1` (die wir bald kennenlernen werden) aber sehr wohl, und drum ist es das beste, sich gleich anzugewöhnen, reguläre Ausdrücke in raw strings zu halten.

lokalisieren

Abwärtskompatibilität

Vorteil der Verwendung dieser Character Classes ist unter anderem, dass das Programm mit relativ wenig Aufwand zu *lokalisieren* ist, d.h. etwa auf die verschiedenen Definitionen von „Buchstabe“ in verschiedenen Sprachen anzupassen. Im Deutschen kämen etwa die Umlaute zu `\w`, im Französischen etliche Zeichen mit allerlei Akzenten usf.

Leider konnten alte Fassungen der Bibliothek, die die regulären Ausdrücke bewertet, nicht mit verschiedenen Encodings und den so genannten Locales (die unter anderem bestimmen, was eigentlich alles Buchstaben sind) umgehen. Als dann die Locale-Unterstützung kam, wollten die Python-MacherInnen das bestehende Verhalten nicht ändern – der Fachbegriff dafür ist *Abwärtskompatibilität*, eine neue Version eines Programms verhält sich in einer bestimmten Beziehung wie die vorherigen. Aus diesem Grund muss man explizit bestellen, dass Dinge wie `\w` wissen soll, was für eine bestimmte Sprache Zeichen sind und was nicht.

Dafür gibt es zwei Möglichkeiten: entweder verwendet ihr das unten diskutierte `re.compile` und gebt als zweites Argument der Funktion `re.L`, oder ihr schreibt ein `(?L)` an den Anfang der RE (was in der Regel einfacher ist) – `"(?L)\w+"` würde also auf alle Wörter (definiert als zusammenhängende Sequenzen von Buchstaben einer Sprache) passen.

Ganz reicht das aber noch nicht, weil ihr dem System ja mitteilen müsst, *welche* Sprache ihr behandeln wollt. Wie das geht, sehen wir im Kapitel zu locales.

sub

Eine weitere ausgesprochen nützliche Funktion, die im re-Modul enthalten ist, ist `sub`. Damit können alle (oder manche) Matches eines regulären Ausdrucks durch etwas anderes ersetzt werden. Eine Funktion, die aus einem Text alle HTML-Tags entfernt:

```
import re

def htmlToTxt(html):
  return re.sub("<[^>]+>", "", html)
```

Sowas könnte etwa so verwendet werden:

```
if __name__=="__main__":
  import sys
  for inName in sys.argv[1:]:
    translated = htmlToTxt(open(inName).read())
    outF = open(inName+".txt", "w")
    outF.write(translated)
    outF.close()
```

Das zweite Argument zu `sub` kann auch eine Funktion sein. Diese Funktion nimmt ein Match-Objekt und sollte einen String zurückgeben. Der Effekt ist, dass ein Match durch den von der Funktion erzeugten String ersetzt wird, dass wir also „berechnete Ersetzungen“ haben.

Ein paar Beispiele dazu:

```
>>> import re
>>> def identity(match):
...     return match.group()
...
>>> def reverse(match):
...     stuff = list(match.group())
...     stuff.reverse()
...     return "".join(stuff)
...
>>> from htmlentitydefs import entitydefs
>>> def entityToLatin1(match):
...     return entitydefs.get(match.group(1), match.group())
...
>>> re.sub("..", identity, "blabla")
```

```
'blabla'
>>> re.sub("..", reverse, "blabla")
'lbbaal'
>>> re.sub("...", reverse, "blabla")
'albalb'
>>> re.sub("....", reverse, "blabla")
'balbla'
>>> re.sub("&([^;]+);", entityToLatin1, "Gr&uuml;&szlig;e")
'Gr\xfc\xdfe'
```

Hier haben wir zunächst drei Funktionen definiert. Die erste, identity, nimmt ein Matchobjekt und gibt den kompletten Match zurück – hier wird also der Match durch den Match ersetzt, was in Summe keine Veränderung gibt.

Amusanter ist schon die zweite Funktion: sie gibt den Match verdreht zurück. Dazu wandelt sie den String zwischenzeitlich in eine Liste, was immer empfohlen ist, wenn am einen String „als Sequenz“, also durch Zugriff auf die einzelnen Elemente, verändern möchte. Die Konsequenz davon ist unten in den Beispielen zu sehen.

Die letzte Funktion tut schon etwas beinahe Nützliches. HTML nämlich ist, auch, wenn der Text selbst allerlei nationale Sonderzeichen enthält, die auf US-Tastaturen nicht draufstehen, ausschließlich mit Zeichen aus ASCII schreibbar. Dazu gibt es die „Entities“, bestehend aus einem Ampersand, ein paar Buchstaben, die beschreiben, welches Zeichen gemeint sei, und dann einem Strichpunkt. So könnt ihr auf HTML-Seiten ä schreiben und bekommt einen a-umlaut, also ein ä, während Ä auf ein Ä und ß („s-z-Ligatur“) auf das ß führt. Nebenbei: Hier haben sich die Namensgeber ordentlich vertan, denn das scharfe s ist historisch mitnichten eine Ligatur aus s und z (auch wenn die Straßenschilder in Berlin das nachträglich so behaupten), sondern eine aus s und s.

Für den heute noch üblichen ISO-Latin1-Zeichensatz enthält das Python-Modul htmlentitydefs ein Dictionary enititydefs, das für viele Zeichenfolgen auf den passenden Ersatztext führt – auml geht auf ä, uuml geht auf ü, mu geht auf μ. Und genau das nutzt entityToLatin1 aus, um HTML-Texte mit Entities so gut wie möglich in Latin-1-Text (der z.B. einfacher mit regulären Ausdrücken zu bearbeiten ist) verwandelt.

40. Reguläre Ausdrücke III

Back references

Mit `\Zahl` können Gruppen in den REs selbst referenziert werden. Dabei entspricht `\1` der ersten, `\2` der zweiten Gruppe usf. `r"(.)\1"` matcht zum Beispiel zwei gleiche Zeichen, die hintereinander stehen, `r"(\w+)\s(\w+)\s\2\s\1"` eine Folge von zwei Wörtern, die gleich dahinter in umgekehrter Reihenfolge kommen.

Besonders praktisch sind die back references in den Ersetzungsstrings von `re.sub`. Beispiel: Datenbanken werden häufig im CSV-Format ausgetauscht, wobei die einzelnen Felder durch Kommata getrennt sind, etwa für Nachname, Name, Telefonnummer:
`Chomsky,Noam,617-555-4543`

Eine kleine Funktion, die das ins Format Name, Telefonnummer, Nachname bringt:

```
def swapFields(ln):
  return re.sub("([^,]*),([^,]*),([^,]*)",
    r"\2,\3,\1", ln)
```

greedy vs. stingy

greedy

stingy

Reguläre Ausdrücke sind in der Regel *greedy*, sie matchen, so viel sie können:

```
>>> re.match("(.*),(.*)",
  "Chomsky,Noam,617-555-4543").groups()
('Chomsky,Noam', '617-555-4543')
```

In unserem Beispiel eben mussten wir deshalb `[^,]` statt dem simplen Punkt schreiben.

Bequemer sind die *stingy* Fassungen von * und +, nämlich *? und +?, die matchen, so wenig sie können:

```
>>> re.match("(.*?),(.*?)",
  "Chomsky,Noam,617-555-4543").groups()
('Chomsky', '')
>>> re.match("(.*?),(.*?),",
  "Chomsky,Noam,617-555-4543").groups()
('Chomsky', 'Noam')
```

Kein Licht ohne Schatten. Erstens sind stingy REs nicht besonders schnell (und können in älteren Python-Implementationen bei sehr langen Gruppen zu Überläufen von Pythons Rekursions-Stacks führen), zweitens kann man auch mit ihnen blöde Fehler machen, wie etwa im oberen Beispiel – da hinter dem zweiten `(.*?)` kein Zeichen mehr kommt, hat Python hier einfach den Leerstring gematcht, nämlich den kleinsten String, der auf den regulären Ausdruck passt.

(Weiterführend:) Wenn ihr tatsächlich CSV-Dateien verarbeiten wollt (und das ist so unwahrscheinlich nicht), solltet ihr das übrigens nicht mit den hier diskutierten rohen Geschichten tun, die etliche Subtilitäten des Formats ignorieren – verwendet stattdessen ab Python 2.3 das csv-Modul[18].

compile

REs, die öfter gebraucht werden, sollte man kompilieren:

```
>>> pat = re.compile(",([^,]*)$")
>>> pat.search("Hello, Dolly!").groups()
(' Dolly!',)
>>> pat.search("Eins, zwei, drei").groups()
(' drei',)
```

Die Kompilation hat im Wesentlichen zwei Vorteile: Erstens laufen die REs dann schneller, zweitens kann man den regulären Audrücken Namen geben, die auf ihre Funktion verweisen (z.B. `nominalPhraseMatcher` o.ä.).

Problems

(40.1)* Probiert back references aus, wenn euch nichts besseres einfällt, an dem CSV-Beispiel. Was passiert, wenn ihr das r vor den Strings mit den back references vergesst? Lasst euch den resultierenden String einmal mit print und dann aus dem Interpreter-Loop (oder mit der eingebauten Funktion repr) ausgeben.

(40.2) Das CSV-Format ist normalerweise noch etwas raffinierter als hier dargestellt; namentlich werden die einzelnen Felder noch in Anführungszeichen gesetzt, etwa so:

```
"Chomsky", "Noam", "617-555-4543"
```

Die Idee ist, dass die Abgrenzung der Felder besser wird. Kommata innerhalb der Anführungszeichen werden natürlich ignoriert, und Whitespace sollte nicht relevant sein. Wie würde ein regulärer Ausdruck aussehen, der damit fertig wird? Noch gemeiner wird es, wenn man auch Anführungszeichen innerhalb der Werte zulassen möchte. Üblicherweise werden diese „escaped“, also mit einem Backslash geschützt: `\"`. Wenn man das hinkriegen möchte, werden die regulären Ausdrücke schon ziemlich furchtbar (Tipp: man muss im Wesentlichen backslashes vor dem schließenden Anführungszeichen verbieten).

(40.3)* Macht euch den Unterschied zwischen greedy und stingy klar, vielleicht zunächst wieder am CSV-Beispiel. Schreibt dann eine Fassung von swapFields, die mit stingy REs arbeitet.

[18] `http://docs.cl.uni-heidelberg.de/python/lib/module-csv.html`

(40.4) Bieten stingy REs einen Ausweg aus dem beim „Parsen“ von Markupsprachen (XML, HTML) häufigen Problem, dass verschachtelte Elemente falsch erkannt werden, dass also die Größe mindestens eines em-Elements in Ausdrücken wie `<em>Dies <em>ist</em> betont</em>` falsch berechnet wird? **(L)**

(40.5)* Probiert den Effekt von compile. Nehmt dazu einen einigermaßen komplexen regulären Ausdruck (der von swapFields sollte es tun) und führt einige hunderttausend Mal eine Ersetzung durch. Stoppt die Zeit mit und ohne compile. Um den Effekt klarer zu machen, könnt ihr statt einer Ersetzung auch einfach nur ein search laufen lassen.

factory function

anonyme Funktionen

41. Factory Functions und Anonyme Funktionen

In `Word` hatten wir die Funktion `newSymbol` verwendet, die einen String nimmt und je nach Inhalt ein `Terminal` oder `NonTerminal` zurückgibt. Mit regulären Ausdrücken kann die Funktion so aussehen:

```
def newSymbol(sym):
  mat = re.match('"([^"]*)"$', sym)
  if mat:
    return Terminal(mat.group(1))
  return NonTerminal(sym)
```

Eine Funktion, die wenig anderes macht, als ein Objekt zu erzeugen, nennt man gern *factory function*.

In Python sind auch Funktionen nur Objekte. Man kann Factory Functions schreiben, die Funktionen zurückgeben. Besonders praktisch dabei sind *anonyme Funktionen*.

```
>>> def makeIncFun(inc):
...     return lambda a, inc=inc:a+inc
...
>>> i1,i2 = makeIncFun(1), makeIncFun(3)
>>> i1(3),i2(3)
(4, 6)
```

Anonyme Funktionen fangen also mit einem `lambda` an, dann kommen die (formalen) Argumente, dann ein Doppelpunkt und schließlich *ein* Ausdruck, der als Ergebnis der Funktion zurückkommt. Das `inc=inc` hat nichts mit lambda zu tun, es ist ein Keyword-Argument, das wir brauchen, um das Argument inc in den Namespace des lambda einzuschleusen. Keyword-Argumente lernen wir bald kennen.

map, filter, reduce

Python kennt einige Funktionen, die Funktionen auf Sequenzen anwenden:

- `map(fun, seq)` – entspricht `[fun(i) for i in seq]`
- `filter(fun, seq)` – enspricht `[i for i in seq if fun(i)]`
- `reduce(fun, seq)`

`reduce` kann nicht ohne weiteres durch eine list comprehension ersetzt werden – ihr Pseudocode ist:

Eingabe: Eine Funktion *fun*, die zwei Argumente nimmt, eine Sequenz *seq*.

l := `list`(*seq*)

Solange *l* länger als eins ist:

l[:2] = *fun*(*l*[0], *l*[1])

Gib l[0] zurück

Hier ist beispielsweise eine Implementation von string.join, die reduce verwendet:

```
def strjoin(sep, seq):
  return reduce(lambda a, b: a+sep+b, seq)
```

Eine berechtigte Frage ist: Wenn map und filter so einfach durch list comprehensions zu ersetzen sind – wer hat sie dann zu eingebauten Funktionen gemacht?

Zur Antwort muss man die berühmten „historischen Gründe“ anführen – map und Freunde kamen relativ früh zu Python, list comprehensions erst viel später, und Guido van Rossum überlegt schon seit einiger Zeit, wie er sie wieder loswird.

Dennoch sollte man sich mit ihnen auseinandersetzen, unter anderem, weil sie im λ-Kalkül einfach auszudrücken sind und deshalb fester Bestandteil funktionaler Programmiertechniken sind.

Darüber hinaus eignen sie sich gut, um mit dem Konzept, dass Funktionen first class sind und man sie an andere Funktionen übergibt, damit diese sie aufrufen, warm zu werden. Häufig nennt man die Funktionen, die da übergeben werden, etwas respektlos callback. Im dritten Jahrtausend sollte niemand mehr von sowas überrascht sein, und tatsächlich verwenden alle modernen Programmiersprachen entsprechende Methoden in aller Breite.

```
>>> filter(lambda a:len(a)>0, ["bla",
... "", "romp", "bar", ""])
['bla', 'romp', 'bar']
>>> l = range(1,4);l
[1, 2, 3]
>>> reduce(lambda a,b: a*b, l)
6
```

Anmerkungen:

- Das Funktionsargument zu `filter` prüft letztlich auf „Nicht-leer“, was für Strings das gleiche wie „wahr“ ist. Deshalb ginge auch einfach `lambda a:a`. Noch schneller geht es mit dem Modul `operator`. In diesem Modul sind die Python-Operatoren als Funktionen, der Operator, den wir hier brauchen, heißt `truth`. Schneller ist das, weil `operator.truth` in C geschrieben ist, während unser `lambda a:a` immer noch in Python ist. Der Aufruf einer Python-Funktion dauert erheblich länger als der einer C-Funktion.
- `range` kann nicht nur ein Argument nehmen, sondern auch zwei oder drei. Mit zwei Argumenten wird der erste als Anfang genommen, mit drei das dritte als Schrittweite. `range(1,10,3)` ist so `[1, 4, 7]`.

Wir können damit unser Grammatik-Lesen etwas robuster machen:

```
  def _readRules(self, sourceName):
    self.rules = {}
    for rawRule in filter(operator.truth,
        map(str.strip,
          open(sourceName).readlines())):
      self._addRule(Rule(rawRule))
```

Problems

(41.1)* Wenn ihr eine Liste negativer Zahlen zwischen $-n+1$ und 0 haben möchtet – wie bekommt ihr sie mit Hilfe von `map`? **(L)**

(41.2)* K.R. Acher möchte gerne eine Liste von (Wert, Schlüssel)-Paaren aus einem Dictionary haben und beschließt, per map die Paare, die er von der items-Methode zurückbekommt, umzudrehen. Er schreibt:

```
>>> d = {'a':1, 'b':2}
>>> def swapPair(pair):
...     return (pair[1], pair[0])
...
>>> map(swapPair(), d.items())
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: swapPair() takes exactly 1 argument (0 given)
```

Was hat Acher falsch gemacht? **(L)**

(41.3)* Macht euch klar, was die Regeln sind, nach denen das dreiargumentige `range` seine Werte erzeugt. Schreibt Pseudocode, der das Verhalten simuliert. Probiert dazu erst ein paar interessante Grenzfälle am Rechner aus und verifiziert dann, dass sich euer Pseudocode genauso verhält. **(L)**

(41.4) Wie sieht das mit den Grenzfällen bei unserer `strjoin`-Funktion aus? Was gibt es für Grenzfälle, und wie schlägt sich die Funktion? **(L)**

(41.5) Implementiert die Fakultätsfunktion mit reduce. **(L)**

(41.6) Schreibt eine Factory Function `makeBegCharChecker(char) -> Function` für Funktionen, die jeweils Wahr zurückgeben, wenn ein übergebener String mit einem der Factory Function übergebenen Zeichen anfängt. Man soll also so etwas machen können:

```
>>> checkForA = makeBegCharChecker("A")
>>> checkFora = makeBegCharChecker("a")
>>> checkForA("Aachen")
True
>>> checkForA("Berlin")
False
>>> checkFora("Aachen")
False
>>> checkFora("aber")
True
```

(L)

42. Rule revisited

```
import operator, string, Symbol, Word

class Rule:
  def __init__(self, rawRule):
    self._parse(rawRule)

  def __str__(self):
    return "%s -> %s"%(self.left, self.right)

  def __repr__(self):
    return "Rule('%s')"%str(self)

  def _parse(self, rawRule):
    try:
      rawLeft, rawRight = map(str.strip,
        rawRule.split("->"))
      if not rawLeft:
        raise ValueError("Empty left side")
      self.left = symbol.NonTerminal(rawLeft)
      self.right = word.Word(rawRight)
    except ValueError, msg:
      raise ValueError(
        "Syntaxfehler in %s: %s"%(rawRule, msg))

  def getLeft(self):
    return self.left

  def getRight(self):
    return self.right

  def applyToLeftmost(self, word):
```

```
    return word.replaceLeftmost(self.left,
      self.right)
```

- Eine Regel besteht jetzt, wie es sich für kontextfreie Grammatiken gehört, aus einem Nichtterminal auf der linken und einem Wort auf der rechten Seite.
- Eigentlich gehören da noch jede Menge Kommentare rein, vor allem Docstrings zu den Klassen und Methoden. Ich darf die hier weglassen, weil die Folie sonst überlaufen würde, ihr dürft das nicht...
- In der except-Klausel in `Rule._parse` steht ein `, msg`. In dem msg steht ein Objekt, das als String die mit der Exception verbundene Meldung enthält. Es ist häufig eine gute Idee, diese Meldungen nicht wegzuwerfen.

43. Grammar revisited

```
class Grammar:
  def __init__(self, sourceName,
      ruleString=None):
    self.ruleDict = {}
    if not sourceName is None:
      ruleString = open(sourceName).read()
    self._parseRules(ruleString)

  def __str__(self):
    return "\n".join(
      map(str,
        reduce(operator.add,
          self.ruleDict.values())))

  def _parseRules(self, ruleString):
    for rawRule in filter(operator.truth,
        map(str.strip,
        ruleString.split("\n"))):
      if rawRule.startswith("="):
        self._setStartSymbol(symbol.NonTerminal(
          rawRule[1:].strip()))
      else:
        self._addRule(Rule(rawRule))

  def _addRule(self, rule):
    self.ruleDict.setdefault(rule.getLeft(),
      []).append(rule)

  def _setStartSymbol(self, startSymbol):
    self.startSymbol = startSymbol

  def getRulesForNonTerm(self, nonTerm):
    return self.ruleDict.get(nonTerm, [])

  def getStartSymbol(self):
    try:
      return self.startSymbol
    except AttributeError:
      return None

  def deriveLeft(self, nonTerm, word,
```

```
    yieldRules=None):
    res = []
    for rule in self.getRulesForNonTerm(
       nonTerm):
      if yieldRules:
        res.append((rule,
          rule.applyToLeftmost(word)))
      else:
        res.append(rule.applyToLeftmost(word))
    return res
```

Das sieht fürchterlicher aus als es ist. Die grobe Struktur ist weiterhin die unserer alten, simplen Grammatikklasse. Wesentliche Neuerungen:

- `_parseRules` räumt die Strings jetzt etwas besser auf und geht sorgfältiger mit dem Startsymbol um.
- Wir haben ein vernünftiges Verhalten, wenn die Grammatik kein Startsymbol gibt – `getStartSymbol` gibt dann nämlich None zurück. Je nach Zweck wäre hier das Auslösen einer Exception auch denkbar gewesen, aber wir haben uns auf den Standpunkt gestellt, dass eine reine Regelmenge ohne Startsymbol durchaus auch sinnvoll sein kann.
- `deriveLeft` hat ein zusätzliches Argument, bei dem auch noch ein `=None` dahintersteht. Was das ist, sehen wir auf der nächsten Folie, warum man `yieldRules` überhaupt haben will, etwas später.

Problems

(43.1)* Geht den Code durch und versucht, zu verstehen, was da jeweils passiert. Im Anhang dieser Seite gibt es das ganze auch kommentiert.

— Dateien zu diesem Abschnitt in der HTML-Version—

44. Default-Argumente

Unser Programm könnte jetzt so aussehen:

```
import sys
import grammar, word, symbol

def deriveLeft(curWord, grammar):
  leftmostNonTerm = curWord.getLeftmostNonTerm()
  return grammar.deriveLeft(leftmostNonTerm,
    curWord)

def generate(curWord, grammar, maxLen,
  seenWords={}):
  if seenWords.has_key(curWord):
    return
  else:
    seenWords[curWord] = 1
  if len(curWord)>maxLen:
    return
  if curWord.isAllTerminal():
    print curWord
    return
  for deriv in deriveLeft(curWord, grammar):
    generate(deriv, grammar, maxLen, seenWords)
```

```
if __name__=="__main__":
  grammar = Grammar.Grammar(sys.argv[1])
  generate(word.Word(str(
    grammar.getStartSymbol())),
    grammar, int(sys.argv[2]))
```

keyword parameter

Aktualparameter

Formalparameter

Anmerkungen:

- Unsere `generate`-Funktion hat sich bemerkenswert wenig verändert – wir haben sie also von vorneherein ganz gut entworfen.
- Die wesentlichste Änderung ist, dass wir jetzt noch ein Dictionary `seenWords` mitführen, in das wir die wir die Wörter eintragen, die wir schon behandelt haben. Damit verhindern wir, dass Wörter mehrfach erzeugt werden und sparen unter Umständen erheblich Rechenzeit ein. Die Verwendung von Dictionaries als Mengen (etwas ist drin oder nicht) ist ziemlich üblich. Es wäre zwar auch denkbar, für so etwas Listen zu verwenden, Dictionaries haben aber die Vorteile, dass (1) von Natur aus jeder Schlüssel nur vorkommt oder nicht vorkommt, während bei Listen ohne weiteres jedes Element beliebig oft vorkommen könne (was bei Mengen unerwünscht ist) und (2) sehr schnell geprüft werden kann, ob ein Schlüssel vorhanden ist, ein Element also in der Menge ist. Unsortierte Listen müssen für so eine Prüfung immer komplett durchsucht werden.
- Ab Python 2.3 gibt es auch eigene Mengentypen, so dass Mengen nicht mehr so emuliert werden müssen. Die Verwendung von Dictionaries wird uns allerdings später ermöglichen, mit einer kleinen Codeänderung auch Ableitungen verfolgen zu können.

Generate verwendet ein Default-Argument, um das `seenWords`-Dictionary beim Aufruf ohne Argumente (also dem nichtrekursiven Aufruf) korrekt zu initialisieren.

Default-Argumente dienen häufig zur Festlegung eines Standardfalls, etwa bei `open`: Normalerweise wird lesend geöffnet, wenn aber ein zweites Argument vorhanden ist, wird er zum Modus. Anderes Beispiel:

```
>>> def inc(var, amount=1):
...     return var+amount
...
>>> inc(3)
4
>>> inc(3,8)
11
```

Damit ist auch das eigenartige `yieldRules`-Argument aus der Grammatik-Klasse klar: Normal (d.h., wenn wir `yieldRules` nicht geben) gibt `grammar.deriveLeft` nur die Ergebnisse der Anwendung von Regeln zurück. Übergibt man aber ein „wahres“ `yieldRules`, kommt eine Liste von Tupeln aus angewandter Regel und resultierendem Wort zurück.

Man kann Default-Argumente auch als *keyword parameter*s verwenden:

```
>>> def incTuple(tup, am1=1, am2=1):
...     return (tup[0]+am1, tup[1]+am2)
...
>>> incTuple((1,1), am2=3)
(2, 4)
```

Die Regeln für die Zuweisung von *Aktualparameter*n (also dem, was im Funktionsaufruf steht) zu *Formalparameter*n (also dem, was im Funktionskopf steht) sind etwa so:

1. Belege zuerst jeden „freien“ Formalparameter mit dem Aktualparameter mit gleichem Index
2. Belege dann jeden Default-Parameter mit dem im Funktionskopf angegebenen Wert
3. Solange noch „freie“ Aktualparameter übrig sind, belege die Default-Parameter nach Reihenfolge, als wären sie von vorneherein frei gewesen
4. Weise die Werte aus den Keyword-Parametern den Formalparametern mit den passenden Namen zu

5. Sollte noch ein „freier“ Aktualparameter nach dem ersten Keyword-Parameter kommen, löse einen Syntaxfehler aus

Die wirklichen Regeln sind übrigens noch etwas komplizierter – im Zweifel gibt die Python Language Reference[19] erschöpfende Auskunft. Aus diesen Regeln folgt, dass auch ohne Default-Parameter Keyword-Parameter verwendet werden können, um etwa ganz deutlich zu machen, welche Parameter welche Werte bekommen. Bei langen Argumentlisten kann sowas helfen, blöde Fehler zu vermeiden.

```
>>> def foo(bar, baz, bong):
...     print bar, baz, bong
...
>>> foo(bong=4, baz="baz", bar=9)
9 baz 4
```

Eine weitere Anwendung für Default-Argumente ist die Vermeidung von globalen Variablen bei Funktionen. In einer Hausaufgabe haben wir die Fibonacci-Zahlen rekursiv berechnet und wollten uns bereits berechnete Werte merken. Damals konnten wir das nur mit einer globalen Variable, was nicht gut ist. Die folgende Fassung vermeidet das:

```
def fib(n, valueCache={}):
  if valueCache.has_key(n):
    return valueCache[n]
  if n==1 or n==0:
    return 1
  else:
    valueCache[n] = fib(n-1)+fib(n-2)
    return valueCache[n]
```

Das funktioniert, weil der Wert des Default-Arguments dann gesetzt wird, wenn die Funktion „compiliert“ wird, also zu dem Zeitpunkt, zu dem der Interpreter die Zeile mit dem `def` verarbeitet. Bei jedem Aufruf von fib mit nur einem Argument hat danach valueCache im das gleiche Dictionary als Wert, eben das, in das wir all die Werte reingeschrieben haben. Dieser Trick wird uns später nochmal im Zusammenhang mit so genannten „Closures“ begegnen.

Noch eine kurze Anmerkung zur Terminologie: Ich versuche, in diesem Skript meistens von Argumenten zu sprechen, wenn ich „Dinge, die an Funktionen übergeben werden“ meine. Das Wort Parameter ist dazu weitgehend synonym (auch wenn manche Autoren da Unterschiede machen). Im Fall von Formal- und Aktualparametern ist allerdings die Verwendung des Wortes Argument sehr unüblich, weshalb ich hier doch Parameter sage. Der Grund dieser „selectional preference“ ist wohl, dass einige einflussreiche Autoren versucht haben, Parameter als sozusagen syntaktische Kategorie zu fassen, während sie das Wort Argument für den konkreten Wert eines Parameters reservieren wollten. Letztlich ist das aber alles egal, die Leute reden in diesem Bereich, wie es ihnen gerade passt.

Problems

(44.1)* Nehmt an, ihr hättet ein großes Programm, das jede Menge Fehlermeldungen produzieren kann. Da nicht von vorneherein klar ist, wie und in welcher Sprache die Fehlermeldungen ausgegeben werden, soll eine eigene Funktion die Ausgabe der Fehler übernehmen. Dabei ist jedem Fehler eine Zahl zugeordnet (die dann wiederum besser in Klartextkonstanten definiert sein sollte: fileNotFoundError = 7 o.dgl.). Schreibt also eine Funktion `reportError(errCode=-1, errTable=englishTable, fatal=0) -> None`, die die Fehlermeldung zum errCode aus der Liste errTable nach stderr ausgibt und das Programm abbricht, wenn fatal wahr ist. Ist errCode==-1, soll das Programm etwas wie „No error specified“ ausgeben. **(L)**

(44.2) Schreibt eine Klasse Set, die ein paar Mengenoperationen bereitstellt: Man soll Elemente hinzufügen und wegnehmen sowie auf Mitgliedschaft testen können, darüber hinaus wäre vielleicht noch die Berechnung noch Schnitt und Vereinigung zweier Mengen nett. Intern sollte die Menge wohl trotzdem als Dictionary dargestellt werden.

[19] `http://docs.cl.uni-heidelberg.de/python/ref/ref.html`

45. Ableitungen

Unser Programm läuft jetzt. Enthalte `small.gr` folgende Grammatik:

```
S -> NP VP        OP -> det no
NP -> det n       vt -> "bites"
det -> "a"        vt -> "eats"
det -> "the"      no -> "man"
VP -> vt OP       n -> "dog"
```

Dann:

```
examples> gener-with-classes.py small.gr 5
a dog bites a man
a dog bites the man
a dog eats a man
a dog eats the man
the dog bites a man
the dog bites the man
the dog eats a man
the dog eats the man
```

Jetzt wollen wir noch die Ableitungen sehen. Dafür bauen wir `generate` so um, dass in seenDict jetzt immer drinsteht, aus welchem Wort das augenblickliche Wort abgeleitet ist. Außerdem drucken wir die Ableitungen nicht mehr, sondern hängen sie in eine (vorerst globale) Liste.

Das sieht dann so aus:

```
def generate(curWord, grammar, maxLen, sourceWord, seenWords={}):
  if seenWords.has_key(curWord):
    return
  else:
    if sourceWord:
      seenWords[curWord] = sourceWord
  if len(curWord)>maxLen:
    return
  if curWord.isAllTerminal():
    allDerivations.append(curWord)
  for deriv in deriveLeft(curWord, grammar):
    generate(deriv, grammar, maxLen, curWord, seenWords)
```

Damit das Sinn hat, ändern wir noch:

```
if __name__=="__main__":
  allDerivations = []
  grammar = Grammar.Grammar(sys.argv[1])
  derivDict = {}
  generate(word.Word(str(
    grammar.getStartSymbol())),
    grammar, int(sys.argv[2]),
    None, derivDict)
  for d in allDerivations:
    printDerivation(d, derivDict)
```

Das seenWords-Dictionary könnte dann für unsere $a^i b a^i$-Grammatik so aussehen:

```
{Word('b'): Word('S'), Word('a a a S a a a'):
Word('a a S a a'), Word('a b a'): Word('a S a'),
Word('a a S a a'): Word('a S a'),
Word('a a b a a'): Word('a a S a a'),
Word('a S a'): Word('S')}
```

Diese Datenstruktur enthält alles, was wir brauchen, um eine komplette Ableitung zu rekonstruieren. Wir müssen nur beim fertigen Wort anfangen, im Dictionary nachsehen, was der Vorgänger war und dann nach dessen Vorgänger sehen. Als Pseudocode:

Eingabe: Ein abgeleitetes Wort *w*, ein Dictionary, das Ableitungen auf ihre Vorgänger abbildet *d*

Setze *ableitung* auf die Liste, in der nur *w* enhalten ist

Solange am Ende von *w* einen Vorgänger hat:

    Setze *w* auf den Vorgänger von *w*

    Hänge *w* an *ableitung* an

Rückgabe: Die Umkehrung von *ableitung*

polling

Endlosschleife

Das lädt zu einer `while`-Schleife ein:

```
def printDerivation(word, derivDict):
  derivList = [word]
  while derivDict.has_key(word):
    derivList.append(derivDict[word])
    word = derivDict[word]
  if len(derivList)>1:
    derivList.reverse()
    print " -> ".join(map(str, derivList))
```

Problems

(45.1)* Die globale Variable `allDerivations` macht natürlich so in etwa alles falsch, was man (stilistisch) falsch machen kann: Sie ist nicht nur global, sie wird auch noch in tief in rekursiven Funktionen verändert.

Welche Alternativen kennt ihr, um damit besser umzugehen?

46. while-Schleifen

`while`-Schleifen sollten zum Einsatz kommen, wenn etwas ausgeführt werden soll, solange eine Bedingung erfüllt ist, man also nicht von vorneherein weiß, wie lange iteriert werden soll.

Ein etwas albernes Beispiel: Nehmen wir an, am Rechner hängt ein Temperaturfühler, den wir mit einer Funktion `getTemperature()` auslesen können (das ist nicht so, es sei denn, ihr schreibt euch sowas selbst). Wenn wir jetzt ein Programm schreiben wollten, das jedes Mal, wenn die Temperatur einen kritischen Wert überschreitet, eine Warnung ausgibt, dann aber leise sein soll, bis die Temperatur wieder normal ist und erst danach wieder auf Überschreitung kontrollieren soll, könnten wir das so machen:

```
critVal = 40.0
while 1:
  while getTemperature()<critVal:
    time.sleep(1)
  warnUser()
  while getTemperature()>critVal:
    time.sleep(1)
```

Die Funktion `sleep` aus dem `time`-Modul sorgt dafür, dass der Rechner für eine Weile (nämlich so viele Sekunden wie im Argument angegeben) nichts tut. Die Technik, in einer Schleife regelmäßig nach etwas zu sehen, heißt übrigens *polling* und sollte in der Regel vermieden werden – über Alternativen werden wir in Programmieren II reden.

Das Konstrukt `while 1:` erzeugt eine *Endlosschleife*. In diesem Fall wollen wir ja wirklich endlos überwachen.

Häufig sind auch Konstrukte wie

```
while 1:
  doSomeStuff()
  if someCondition():
    break
```

```
  doMoreStuff()
```

– sie sind immer dann Mittel der Wahl, wenn das Abbruchkriterium logisch am Besten mitten im Schleifenkörper selbst berechnet werden kann. Man sollte darauf achten, dass nicht zu viele breaks und continues in einem Schleifenkörper stehen (generell sollten Schleifenkörper ohnehin nicht mehr als vielleicht 10 Zeilen haben), weil sonst der control flow sehr undurchsichtig wird. In Maßen verwendet, kann sowas die Lesbarkeit eines Programms aber – entgegen den Aussagen gewisser Stilpäpste – durchaus erhöhen.

Ein weiteres Beispiel zu while ist die Mandelbrotmenge. Die Details sind nicht so wichtig, die Vorschrift ist jedenfalls, die Gleichung $z_{n+1} = z_n^2 + c$ (in Python wäre das also z=z**2+c) so lange zu iterieren, bis eine vorher vereinbarte Anzahl von Iterationen gemacht wurde oder eine vorher vereinbare Zahl von Iterationen gemacht wurde. Die Funktion, die das tut, soll dann zurückgeben, wie viele Iterationen sie gebraucht hat.

Code zur Berechnung könne so aussehen:

```
def iterate(point, maxIter=200):
  numIter = 0
  z = 0j
  while abs(z)<2 and numIter<maxIter:
    z = z*z+point
    numIter += 1
  return numIter
```

(das 0j sagt, dass es sich bei z um eine komplexe Zahl handelt – ich sagte ja, dass die Details hier nicht so wichtig sind).

Eingebunden in ein Programm wie

```
def scalediter(minVal, maxVal, numPoints):
  scaleFact = (maxVal-minVal)/float(numPoints)
  for val in xrange(numPoints+1):
    yield val, val*scaleFact+minVal

class MandelDrawer:
  def __init__(self, displayer):
    self.displayer = displayer

  def drawImg(self, minX=-1.75, minY=-1, maxX=0.5, maxY=1):
    numPixels = self.displayer.getNumPixels()
    for y, ys in scalediter(minY, maxY, numPixels):
      for x, xs in scalediter(minX, maxX, numPixels):
        val = iterate(complex(xs, ys), 512)
        self.displayer.setPoint(x, y,
          "#%02x00%02x"%(val%256, (256-val/2)%256))
      self.displayer.updateDisplay()

if __name__=="__main__":
  _test()
  p = pixelCanvas.PixelCanvas(2, 100)
  m = MandelDrawer(p)
  m.drawImg()
  p.mainloop()
```

kann man damit schon die vor einigen Jahren furchbar populären Apfelmännchen malen, wenn auch eher langsam:

(cf. Fig. 8)

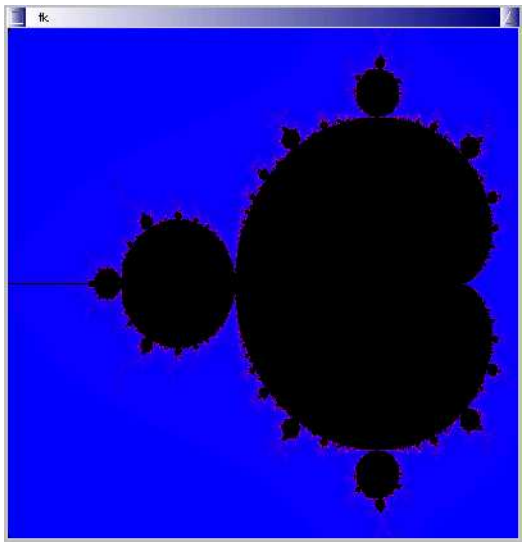


Fig. 8

Dabei gibts das pixelCanvas-Modul im Anhang (was darin geschieht, ist erst nach unserem Ausflug in die GUIs verständlich). In der komischen scaledIter-Funktion haben wir der Bequemlichkeit halber schon auf Generatoren zurückgegriffen, die wir erst auf der Folie „Generatoren und Iteratoren“ diskutieren werden.

Natürlich würde niemand, der/die seine/ihre Sinne noch beieinander hat, diese Sorte von Heavy-Duty-Numerik ernsthaft direkt in Python machen. Wir werden zwar in Programmieren II Techniken kennenlernen, wie wir allzu viel C (oder Fortran oder was immer) in solchen Aufgaben vermeiden können – wer aber jetzt schon schnell schöne Bilder sehen will, sei auf XaoS[20] verwiesen.

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(46.1)* While- und for-Schleifen sind im Wesentlichen äquivalent, d.h. man kann alle for-Schleifen auch durch while-Scheifen ersetzen und umgekehrt (auch wenn letzteres in Python ein wenig Getrickse braucht).

Wie sieht die folgende Schleife in while-Form aus?

```
for i in range(10):
  print i
```

(L)

[20] `http://xaos.theory.org/`

47. Exkurs: Ein einfacher Parser

Parser

Die Infrastruktur, die wir uns gebaut haben, erlaubt auch, relativ einfach einen schlichten Parser zu implementieren.

Ein *Parser* ist dabei ein Programm, das die syntaktischen Beziehungen, die zunächst in der Sequenz von Symbolen versteckt sind, in eine zur Darstellung dieser Beziehungen geeignete Struktur überführt. In dem Satz „Der Mann beißt den roten Hund“ könnte man beispielsweise sagen, dass er aus einer Nominalphrase („Der Mann“) und einer Verbphrase besteht („beißt den roten Hund“). Letztere besteht dann aus einem transitiven Verb („beißt“) und einer weiteren Nominalphrase – und so fort, bis man bei den einzelnen „Wörtern“ (die Anführungszeichen sollen andeuten, dass hier die Alltagsbedeutung von Wort gemeint ist) angelangt ist.

Für die kontextfreien Grammatiken, die wir hier behandelt haben, ist die geeignete Struktur ein Baum. Zu diesen wird später noch mehr zu sagen sein, vorerst reicht es, sich vorzustellen, dass ein Baum die im letzten Absatz dargestellte Verschachtelung, bei der ein paar Konstituenten Kinder *einer* anderen sind, darstellt.

Genau genommen taugt übrigens die letzte Fassung des Generierungsprogramms auch schon als (sehr ineffizienter) Parser – aber dabei gibts dann ein paar Probleme, deren Diskussion hier zu weit führt.

Eine einfacher rekursiver top-down-Parser sieht so aus:

```
class TopDownParser:
  def __init__(self, grammar):
    self.grammar = grammar

  def parse(self, wordToParse, curParse=None):
    if curParse is None:
      curParse = word.Word(str(
        self.grammar.getStartSymbol()))
    if wordToParse==curParse:
      return []
    leftmostNonTerm = curParse.getLeftmostNonTerm()
    if not leftmostNonTerm or len(curParse)>len(wordToParse):
      return
    for rule, deriv in self.grammar.deriveLeft(
        leftmostNonTerm, curParse, yieldRules=1):
      rulesApplied = self.parse(wordToParse, deriv)
      if rulesApplied is not None:
        return [rule]+rulesApplied
```

Im Wesentlichen generieren wir und hören auf, wenn wir das richtige Wort gefunden haben. Wenn das so ist, geben wir die Folge der Regeln, deren (Links-) Anwendung zum gesuchten Wort führt, zurück.

So, wie das geschrieben ist, ist das Verhalten für ambige Grammatiken (also solche, in denen verschiedene Regelfolgen zu einen gegebenen Wort führen können) nicht sinnvoll – der Parser gibt eben nur eine Regelfolge (einen „Parse“) zurück. Außerdem geben wir den Versuch, das Wort zu parsen, auf, wenn `curParse` länger als `wordToParse` wird. Das wiederum funktioniert nur, wenn die Wörter während einer Ableitung nie kürzer werden können. Tatsächlich ist das aber keine wesentliche Beschränkung, da man zeigen kann, dass man kontextfreie Grammatiken (fast) immer so umformen kann, dass die diese Bedingung erfüllen (man spricht da von ϵ-Freiheit).

Die Klasse, die wir hier definiert haben, wirkt ein wenig albern, weil es nur eine wirklich nützliche Methode gibt. Allerdings gibt es mehr als nur einen Parse-Algorithmus, und vielleicht wollen wir ja später mal einen anderen einsetzen. Durch die Verkapselung in eine Klasse ist die Wahrscheinlichkeit größer, dass wir bei einem Wechsel des Algorithmus die Schnittstelle beibehalten können.

Mit einer Grammatik wie

```
S -> NP VP
S -> NP VP PP
VP -> "vi"
```

```
VP -> "vt" NP
VP -> "vt" NP PP
NP -> "n"
NP -> "det" "n"
NP -> "det" "adj" n
PP -> "prep" NP
```

gibt der Parser für das Wort
`word.Word('"det" "n" "vt" "n" "prep" "n"')`

folgenden Parsebaum:

```
+-S
  +-NP
  | +-"det"
  | +-"n"
  +-VP
    +-"vt"
    +-NP
    | +-"n"
    +-PP
      +-"prep"
      +-NP
        +-"n"
```

Die oben angegebene Grammatik könnte schon für eine simple natürliche Sprache gut sein. Dabei sind die Terminale dieser Grammatik so genannte Präterminale, die grammatischen Kategorien entsprechen. Das `n` steht etwa für ein Nomen, das `vi` für ein intransitives Verb. In der Regel werden solche „Tags“ (die aber in der Realität weit komplexer sein können, etwa „Nomen im Nominativ Singular“) von einem dem Parser vorgeschalteten Programm, einem Tagger, vergeben. Der Vorteil dieser Vorgehensweise ist, dass Lexikon und Grammatik getrennt bleiben können.

Im an diese Seite angehängten Code ist eine Funktion enthalten, die aus der Folge von Regeln, die `parse` zurückgibt, einen Baum in geeigneter Darstellung erzeugt, und eine weitere Funktion, die diesen Baum in der gezeigten Weise ausgibt. Wer Lust hat, kann diese Funktionen studieren.

— Dateien zu diesem Abschnitt in der HTML-Version—

48. Exkurs: Ein Parser, der etwas tut

Es ist nett, einen Parsebaum zu haben – häufig will man aber gleich Code ausführen, wenn man eine Regel erkennt. Wir können das mit einer Ableitung unserer Rule-Klasse erreichen:

```
class RuleWithAction(grammar.Rule):
  def _parse(self, rawRule):
    self.action = "$1"
    ruleAction = rawRule.split("---")
    if len(ruleAction)==2:
      self.action = ruleAction[1].strip()
    else:
      if len(ruleAction)!=1:
        raise ValueError("Too many --- in '%s'"%
          rawRule)
    grammar.Rule._parse(self, ruleAction[0])

  def runAction(self, currentNonterms):
    symVals = self._getSymVals(currentNonterms)
    val = eval(
      re.sub(r"\$(\d+)",
        lambda m: symVals[int(m.group(1))-1],
```

```
      self.action)
    )
    currentNonterms.push(self.getLeft(), val)
    return val
```

Im Wesentlichen haben wir die _parse-Methode überschrieben. Sie splittet den String, den sie übergeben bekommt, an einer Folge von drei Minuszeichen. Wenn das Ergebnis die Länge zwei hat, wird der zweite Teil in der Instanzvariablen `action` gespeichert, ansonsten bleibt dort die Vorbelegung `"$1"`. Danach wird einfach die _parse-Methode der Oberklasse aufgerufen. Was bei diesem Aufruf genau vorgeht und warum der so komisch aussieht, werden wir noch sehen.

Der Sinn dieses Vorgehens wird bei einer Inspektion der Methode `runAction` klar – dort wird nämlich `self.action` per eval als Python-Ausdruck ausgewertet. Davor allerdings läuft noch ein regulärer Ausdruck, der alle Strings der Form `$n` (mit einer ganzen Zahl `n`) durch ihre Entsprechung in einer Liste ersetzt. Diese Liste kommt aus einer Methode `_getSymVals`, die zu jeder Konstituente auf der rechten Seite der Regel ihren augenblicklichen Wert (der davon abhängt, welche Regeln vorher liefen) in die Liste schreibt. Haben wir beispielsweise auf der rechten Seite der Regel etwas wie `NP VP "."`, und hätten vorherige Regelauswertungen dazu geführt, dass `NP` auf `"fritz"` und `VP` auf `"isst"` abgeleitet werden, wäre die Liste, die `_getSymVals` zurückgibt, etwas wie `["Fritz", "isst", "."]`.

Leider gibt es dabei ein paar Subtilitäten, die hier zu weit führen und in einen Parsingkurs gehören, weshalb ich `_getSymVals` nicht diskutiere – wenn ihr wollt, könnt ihr euch überlegen, warum es die Hilfsklasse `MultiStack` in `actionrule.py` (vgl. Anhang) braucht, damit `runRuleSeq` in Kooperation mit `_getSymVals` die Werte der Nonterminale in einer Linksableitung korrekt berechnen kann. Gegenstand dieses Programmierkurses ist es jedoch nicht.

Anmerkung: Aus Sicherheitsaspekten ist es häufig keine gute Idee, `eval` auf strings loszulassen, die aus einer Datei oder allgemein von einer/m BenutzerIn kommen. Es kann dabei leicht zu so genannten *privilege escalations* kommen, etwa, wenn ihr euren Parser am Netz laufen lasst und Grammatik aus einem Web-Formular lest. Stellt euch vor, was passiert, wenn der Server das Programm unter euerer User-Id startet und böswillige Menschen ein `os.system("rm -rf ~")` als Aktion eingegeben haben. Das Privileg dazu wolltet ihr ihnen wahrscheinlich nicht einräumen (ganz genau genommen kann hier gar nicht von Eskalation geredet werden, denn wir haben ja gerade beliebige Python-Ausdrücke als Aktionen zugelassen und nur vielleicht nicht überlegt, dass eine solche Aktion eben auch das Löschen des Homeverzeichnesses sein kann).

Dieses Problem taucht immer dann auf, wenn BenutzerInnen von Programmen weniger tun dürfen sollen als das Programm selbst und ist notorisch kompliziert in den Griff zu bekommen. Wir wollen hier davon ausgehen, dass unser Programm nicht in derartigen Situationen läuft.

Diese Regeln müssen jetzt von `Grammar` auch verwendet werden. Hier ist hilfreich, dass Klassen in Python first class sind, d.h., wir können Grammar sagen, welche Regelklasse es verwenden soll. Das geht z.B. so:

```
class Grammar:
  def __init__(self, sourceName, ruleString=None,
     ruleClass=Rule):
    self.ruleClass = ruleClass
    [...]

  def _parseRules(self, ruleString):
    for rawRule in filter(operator.truth,
    [...]
      self._addRule(self.ruleClass(rawRule))
```

Tatsächlich steht das so bereits in dem Code, den ihr von der Vorlesungsseite herunterziehen konntet.

Mit ein paar zusätzlichen Ingredienzien kann man damit beispielsweise Zahlwörter erzeugen:

```
examples> genNumbers.py 2 78 458759456
2 zwei
```

```
78 achtundsiebzig
458759456 vierhundertachtundfünfzigmillionen sie
benhundertneu nundfünfzigtausend vierhundertsech
sundfünfzig
```

Was dazu über unsere Infrastruktur hinaus nötig ist, steht im Anhang dieser Seite („Infrastruktur“ meint hier grammar.py, symbol.py, word.py sowie topDownParser.py).

Die Grammatik, mit der das Beispiel oben gerechnet wurde, findet ihr in genNumbers.py – vielleicht macht es euch ja Spaß, diese Grammatik zu verbessern (Brüche wären nett. . .). Schickt mir gerne Ergebnisse eurer Mühen.

Wenn ihr allerdings ernsthaft Grammatik entwickeln wollt, ist der Top-Down-Parser von der letzten Seite untauglich, weil viel zu langsam – was nicht an Python, sondern am Algorithmus liegt.

Erheblich (bei größeren Zahlen um Größenordnungen) flotter gehts mit dem Modul shiftReduce-Parser, das ihr ebenfalls im Anhang findet. Er wird automatisch verwendet, wenn ihr ihn irgendwo in euren Python-Pfad legt. Dafür sorgt das Konstrukt

```
try:
  from shiftReduceParser import ShiftReduceParser, ShiftReduceGrammar
  ParserClass, GrammarClass = ShiftReduceParser, ShiftReduceGrammar
except ImportError:
  pass
```

(wieder macht uns der Umstand, dass Klassen first class sind, das Leben deutlich einfacher). Es lohnt sich, den Geschwindigkeitsunterschied zwischen den beiden Parsern für etwas größere Zahlen anzusehen – wir werden in Programmieren II noch mehr Beispiele sehen, in denen die Wahl des Algorithmus entscheidend für die Nutzbarkeit eines Programms sein wird.

Dafür ist der Shift-Reduce-Parser in dem Modul relativ verwickelt (er bringt sogar seine eigene Grammatik-Klasse mit, die so tolle und letztlich rekursive Funktionen wie first und follow berechnen kann). Ihr solltet wohl erst dort reingucken, wenn ihr euch mal von der theoretischen Seite her mit tabellengesteuerten Shift-Reduce-Parsern beschäftigt habt. Nur als Warnung: Ich habe gegenüber Hellwigs Rezept einiges geändert, weil unser „Regeln statt Bäume zurückgeben“-Ansatz das nötig gemacht hat. Kommentieren werde ich den Parser erst, wenn mich jemand darum bittet. . .

— Dateien zu diesem Abschnitt in der HTML-Version—

49. Sichtbarkeit und Blöcke

In der letzten Fassung von `generate` haben wir die globale Variable `allDerivations` verwendet. Das ist unschön und sollte natürlich so nicht gemacht werden – globale Variablen hatten wir schon vor geraumer Zeit als gefährlich deklariert. Wir hatten damals aber auch gesagt, dass Zuweisungen immer in den lokalen Namespace gehen. Dass `allDerivations` hier tatsächlich alle Ableitungen speichert, liegt daran, dass wir nicht zuweisen, sondern einen bestehenden Wert (der im globalen Namespace gefunden wird) per `append` ändern.

In Wirklichkeit wäre es an dieser Stelle günstiger, `allDerivations` als weiteres Argument zu übergeben oder (besser) die jeweilig erzeugten Ableitungen zurückzugeben und „auf dem Rückweg“ aus der Rekursion zu sammeln.

An dieser Stelle müssen wir nochmal zur Frage von lokalen und globalen Variablen zurückkehren.

Python hält drei Dictionaries für Namen: Einen für lokale, einen für globale und einen für eingebaute (wie `open`, `map`). Die Zuweisung `a = 3` bedeutet letztlich etwas wie `someDict["a"] = 3`. `someDict` ist das lokale Dictionary, wenn wir in einer Funktion (oder Klassendefinition) sind, das globale sonst.

Wenn wir auf eine Variable `var` zugreifen, passiert in etwa das:

```
try:
```

```
  return localDict["var"]
except KeyError:
  try:
    return globalDict["var"]
  except KeyError:
    return builtinDict["var"]
```

Block

lexikalisches Scoping

Deshalb sieht man globale Variablen, während man bei Zuweisungen ins lokale Verzeichnis schreibt.

Die eingebauten Funktionen `locals()` und `globals()` geben diese Dictionaries zurück. Wer sie verändert, bekommt Ärger.

Globale Variablen sind gefährlich!

In *ganz* seltenen Fällen kann es einmal vorkommen, dass man doch aus einer Funktion heraus das globale Dictionary ändern möchte. Auch in diesen Fällen darf man nicht über `globals()` gehen, sondern kann das Schlüsselwort `global` verwenden:

```
>>> globVar = "ich"
>>> def checkGlob():
...     globVar = "du"
...     print globVar
...
>>> checkGlob()
du
>>> globVar
'ich'
```

```
>>> def checkGlob():
...     global globVar
...     globVar = "du"
...     print globVar
...
>>> checkGlob()
du
>>> globVar
'du'
```

Aber wie gesagt: Ihr könnt ein Leben lang programmieren, ohne je in eine Sitution zu kommen, in der sowas eine gute Lösung wäre. Vergesst es am besten gleich wieder.

Die Regel, welche lokalen Dictionaries für welchen Code zuständig sind, hängt eng mit dem Begriff des Blocks zusammen. Ein *Block* ist ein Stück Code, das als Einheit ausgeführt wird – genauer sind Python Module, Skripte, Funktionskörper und Klassendefinitionen Blöcke, und zusätzlich noch ein paar Exoten wie die Strings, die man an eval oder exec übergibt. Jeder Block hat sein eigenes lokales Dictionary (im Falle von Modulen und Skripten ist das lokale gleich dem globalen Dictionary).

Es ist nützlich, sich klar zu machen, dass Blöcke (in dieser Sprechweise) von compound statements wesentlich verschieden sind, da in der Literatur die beiden Begriffe nicht immer streng getrennt werden. In Sprachen wie C, Java oder Pascal haben compound statements in verschiedenem Maße Eigenschaften von Blöcken – sie können beispielsweise selbst lokale Variablen enthalten, die außerhalb des compound statements nicht sichtbar sind. In Python ist das nicht so; compound statements sind eine rein syntaktische Geschichte, um zusammengehörige Anweisungen zu gruppieren, während Blöcke über die Sichtbarkeit von Variablen entscheiden.

Lexikalisches Scoping

In Wirklichkeit ist es noch komplizierter: Steht ein Block lexikalisch (also sozusagen „im Quelltext“) innerhalb eines anderen, so wird (seit Python 2.3, in 2.2 kann man das Verhalten durch `from __future__ import nested_scopes` bestellen) auch in den umschließenden Namespaces gesucht – das heißt *lexikalisches Scoping*:

```
>>> def makeAppender(aList):
...   return lambda item: aList.append(item)
...
>>> l = []
>>> app = makeAppender(l)
>>> app(1);app("zwei")
>>> l
[1, 'zwei']
```

Um hinter das lexikalische Scoping zu kommen, müsst ihr die Definition des lambda ansehen: In dessen Funktionskörper wird der Name `aList` verwendet, der darin eigentlich gar nicht definiert ist. Er wird zur Laufzeit von `makeAppender` ausgefüllt, und zwar mit dem Wert von `aList` im Namespace von `makeAppender`, ohne dass `aList` jemals im Namespace des lambda (oder eben dem globalen) gewesen wäre.

closure

Auch wenn das jetzt nicht so nützlich aussieht: Funktionen, in die man auf diese Weise Kontext „einfriert“ sind sehr nützlich, wenn man gegenseitige Abhängigkeiten reduzieren und Schnittstellen vereinfachen möchte. Wir haben Ähnliches bisher mit Default-Argumenten gemacht, aber da diese immer noch überschrieben werden können, ist der Umweg über lexikalisches Scoping eleganter.

Im Fachjargon heißen Funktionen mit eingefrorenem Kontext *closure*s.

Problems

(49.1)* Probiert das Beispiel mit der checkGlob-Funktion aus. Wenn ihr nicht ganz sicher seid, was da vorgeht, lasst euch die globals() und locals() zwischen den Zeilen ausgeben und beobachtet, wie sie sich verhalten.

(49.2)* Lasst euch im Interpreter locals() und globals() ausgeben. Dann weist einer Variable einen Wert zu und seht euch nochmal die beiden Dictionaries an. Was beobachtet ihr? Gibts dafür eine Erklärung? Was passiert, wenn ihr das innerhalb einer Funktion macht?

(49.3) Mit der eingebauten Funktion `dir` könnt ihr euch ausgeben lassen, was so alles in einem Namespace drin ist. Euch ist in der Ausgabe von globals() vermutlich der komische Eintrag für `__builtins__` aufgefallen. Benutzt dir, um nachzusehen, was da so alles drinsteht. Und lasst euch nicht dabei erwischen, wie ihr da drin rumpfuscht...

50. Locales

Menschen haben verschiedene Kulturen – während wir den zweiten Juni 1967 als 2.6.1967 schreiben, würden Briten 2/6/1967 und US-AmerikanerInnen 6/2/1967 schreiben. 10,000 würde hier als 10 gelesen, in den USA aber als 10^4. Telefonbücher sind je nach Land anders sortiert, die Währungszeichen stehen mal vor, mal hinter dem Betrag usf.

Rechner sollten wissen, wie solche Dinge behandelt werden. Um eine über Plattformen und Programmiersprachen hinweg einheitliche Methode zu haben, an diese Informationen heranzukommen, wurden vor einigen Jahren die *Locale*s definiert.

Locale

Pythons Schnittstelle zu den Locales ist im Modul locale. Darin ist u.a. enthalten:

- `setlocale`: Setzt das aktuelle Locale
- `strcoll`: Vergleichsfunktion für Strings gemäß dem augenblicklichen Locale
- `localeconv()`: Gibt ein Dictionary mit allerlei Informationen über das Locale zurück.
- `LC_ALL`, `LC_CTYPE`, `LC_COLLATE` und etliche mehr: Symbole für setlocale, erlauben Auswahl der „Aspekte“ eines Locales.

Beispiel:

```
>>> import locale
>>> l = ["Ast", "Äste", "ätzend", "Zeder"]
>>> l.sort(locale.strcoll);l
['Ast', 'Zeder', '\xc4ste', '\xe4tzend']
>>> locale.setlocale(locale.LC_ALL, "de_DE")
'de_DE'
>>> l.sort(locale.strcoll);l
['Ast', '\xc4ste', '\xe4tzend', 'Zeder']
>>> locale.localeconv()
{'mon_decimal_point': ',', 'int_frac_digits': 2,
'thousands_sep': '.', 'n_sign_posn': 1,
'decimal_point': ',', 'n_cs_precedes': 0,
```

```
'negative_sign': '-', 'currency_symbol': 'DM'}
```

(Dictionary ist gekürzt)

Wir haben hier alle (LC_ALL) Kategorien auf das gesetzt, was in de_DE definiert ist. de_DE bedeutet dabei „Deutsch, mit den Konventionen in der BRD“, im Gegensatz etwa zu de_CH, „Deutsch mit den Konventionen in der Schweiz“.

Leider kann man sich nie wirklich darauf verlassen, dass ein gegebenes System ein locale wirklich kann, und es ist ohnehin meist das Beste, das Locale, das der/die BenutzerIn selbst ausgewählt hat, zu verwenden. Das System sorgt dafür, wenn man `setlocale` einen leeren String übergibt. `locale.setlocale(locale.LC_ALL, "")`

Die Auswahl des Locales geht unter Unix typischerweise durch Setzen von Umgebungsvariablen (*Environment Variables*). Dabei handelt es sich um Variablen, die die Shell verwaltet und die Programmen, die von der Shell gestartet werden, weitergegeben werden. Für Locales relevant sind hier LC_ALL oder auch LANG.

Für Bourne-kompatible Shells cann das so aussehen:

```
examples> cat sorttest.py
import locale

locale.setlocale(locale.LC_ALL, "")
l = ["Ast", "Äste", "ätzend", "Zeder"]
l.sort(locale.strcoll)
print " ".join(l)
examples> export LC_ALL=C
examples> chmod +x sorttest.py
examples> sorttest.py
Ast Zeder Äste ätzend
examples> export LC_ALL=de_DE
examples> sorttest.py
Ast Äste ätzend Zeder
```

Das locale mit dem Namen „C“ ist dabei das Locale, in dem ein Programm, das nichts anderes sagt, läuft.

Problems

(50.1)* Macht euch vertraut mit der Art, wie auf eurem System locales gesetzt werden. Probiert ein paar locales durch und beobachtet den Effekt auf locale.localeconv(). Was tut sich bei sorttest.py?

51. Loose Ends

Exceptions selbst definieren

Häufig will man „eigene Fehler“ werfen können, um z.B. zu sagen: „Für dieses Wort existiert keine Ableitung“. Das geht so:

```
class NoDerivation(Exception):
  pass

def getDerivation(word):
  ...
  raise NoDerivation("Sigh")
  ...
```

Man beerbt also die eingebaute Klasse `Exception`.

Übrigens war es früher üblich, die Konstruktion der Exception-Instanz dem raise zu überlassen, was so aussah:

```
raise NoDerivation, "Sigh"
```

Damit konnte man auch allen möglichen anderen Krempel raisen (also Dinge, die nicht von Exception abgeleitet waren). Python unterstützt das noch heute, aber man sollte es nicht mehr tun.

Es ist oft sinnvoll, sich für ein eigenes Modul einen Satz von Exceptions zu definieren, die von einer gemeinsamen Exception erben:

```
class GenerException(Exception):
  pass

class NoDerivation(GenerException):
  pass

class EndlessRecursion(GenerException):
  pass
```

Vorteil dieses Verfahrens ist, dass Programme, die auf das Modul aufbauen, wahlweise alle Exceptions fangen können, die das Modul erzeugt (`except GenerException:` reagiert auf alle von `GenerException` abgeleiteten Exceptions) oder eben nur ganz bestimmte Fehler.

Short Circuit Evaluation

In Python (und in den meisten anderen Sprachen) werden Boole'sche Ausdrücke per *Short Circuit Evaluation* („Kurzschlussauswertung“) berechnet. Das bedeutet, dass mit `and` oder `or` verknüpfte Ausdrücke nur berechnet werden, bis das Ergebnis feststeht. Ein Beispiel:

```
>>> if 1 and sys.stdout.write("Ha!\n"):
...     sys.stdout.write("Hu!\n")
...
Ha!
>>> if 1 or sys.stdout.write("Ha!\n"):
...     sys.stdout.write("Hu!\n")
...
Hu!
```

Im ersten Fall weiß Python nach der Eins noch nicht, ob der Ausdruck wahr ist – der zweite Teil der Konjunktion könnte ja falsch sein, und `1 and 0` ist eben falsch. Demnach wird das `write` ausgewertet, es gibt `None` zurück, die Gesamtbedingung ist falsch, also wird kein „Hu!“ ausgegeben. Hätte es umgekehrt `if 0 and` geheißen, wäre bereits bei der Null klar gewesen, dass der Ausdruck falsch ist („falsch und irgendwas“ ist immer falsch) und das `write` wäre nicht ausgewertet und damit auch nicht ausgeführt worden.

`lambda`
`or`
`and`
`not` *x*
`in` , `not in`
`is` , `is not`
`<`, `<=`, `>`, `>=`, `!=`, `==`
`|`
`^`
`&`
`<<`, `>>`
`+` , `-`
`*` , `/` , `%`
`+`*x* , `-` *x*
`~`*x*
`**`
x.*attribute*
x[*index*]
x[*index*:*index*]
f(*arguments*...)
(*expressions*...)
[*expressions*...]
{*key* : *datum*...}
‘ *expressions*... ‘

Fig. 9

Im zweiten Fall ist schon bei der 1 klar, dass die Bedingung erfüllt ist: „wahr oder irgendwas“ ist immer wahr.

(Weiterführend:) In vielen Programmiersprachen basieren endlos viele Tricks auf diesem Verhalten, vor allem in perl und in der Bourne-Shell. Programme in diesen Sprachen strotzen vor Konstruktionen wie

```
mkdir("stuff") or die("Couldn't do something");
```

(Weiterführend:) Hintergrund ist, dass viele dieser Sprachen kaum über vernünftige Exception-Mechanismen verfügen und Fehlerbedingungen über den Rückgabewert einer Funktion mitgeteilt werden. Wenn nun die Funktion 0 zurückgegeben hat, muss der die zweite Hälfte des Ausdrucks ausgewertet werden, und das ist hier die Funktion `die`, die das Programm mit einer Fehlermeldung abbricht. Hat sie hingegen etwas ungleich Null zurückgegeben, muss der zweite Teil nicht ausgewertet werden, das Programm läuft weiter.

(Weiterführend:) Diese Sorte Krankheit ist in Python in der Regel überflüssig, weil wir Exceptions haben. Auch sonst sollte man wissen, dass es die Short Circuit Evaluation gibt, normalerweise aber nicht zu intensiv darüber nachdenken. Es gibt in Python nicht viele Fälle, in denen man sie aktiv verwenden will (wer unbedingt in funktionalen Programmen Selektion machen möchte, wird das brauchen).

(cf. Fig. 9)

Präzedenz

Präzedenztabelle

Die *Präzedenztabelle* gibt für alle Operatoren die Stärke ihrer Bindung, wobei ganz oben die am schwächsten bindenden Operatoren stehen.

So ist beispielsweise `5*6+b` als
`(5*6)+b` zu lesen, weil `*` in der Tabelle niedriger steht als `+`, also stärker bindet.

Weitere Beispiele:

`4+5 and 4>x[2]` ist
`(4+5) and (4>(x[2]))`

`7*+x.bla**2` ist
`7*(+((x.bla)**2))`

Normalerweise liest Python von links nach rechts.

Eigentlich kann man sich die Tabelle bei Python fast schenken, weil alles so ist „wie man es erwartet“. In anderen Sprachen ist das nicht immer so. In der Regel sollten Klammern gesetzt und nicht die Tabelle befragt werden, wenn irgendeine Unsicherheit besteht. Diese Fälle sind meist so bizarr, dass explizite Klammern ohnehin schlicht hilfreich sind.

Problems

(51.1)* Macht euch klar, wie das, was in der except-Klausel steht mit der Klassenhierarchie z.B. der oben angegebenen Exceptions interagiert. Schreibt also ein Programm, das mal eine Exception der Oberklasse wirft und eine Unterklasse fängt und umgekehrt.

(51.2) Gibt es einen Unterschied zwischen

```
try:
  ...
except Exception:
  ...
```

und

```
try:
  ...
except:
  ...
```

(L)

(51.3)* Was gibt das folgende Programm für verschiedene Kommandozeilenargumente (für welche unterscheidet sich die Ausgabe?) aus – und vor allem: warum?

```
import sys

def printAndReturnInt(s):
  print s
  return int(s)

if (len(sys.argv)>1 and printAndReturnInt(sys.argv[1])
  ) and (len(sys.argv)>2 or printAndReturnInt(0)
  ) and (printAndReturnInt(sys.argv[2])):
  print not int(sys.argv[2]) or 0
```

(L)

(51.4)* Klammert die folgenden Ausdrücke vollständig (d.h. sie sollen nach der Klammerung das gleiche Ergebnis liefern wie nach der Präzendenztabelle ohne Klammern).

```
x+y**2
a.attr+-5
f(a+b) is not a<b
lambda a, b: not a or b.attr[4]
```

(L)

52. Bäume I

Baum
Knoten
Wurzel
Graph
Blätter
Pointer

Ein *Baum* ist eine Datenstruktur, die aus *Knoten* besteht. Dabei hat genau ein Knoten (die *Wurzel*) keinen Vorgänger, alle anderen Knoten haben genau einen Vorgänger (es also keine Verweise von Ast zu Ast oder zurück – eine Datenstruktur, bei der das vorkommen darf, heißt *Graph*) und null oder mehr Nachfolger. Knoten ohne Nachfolger heißen *Blätter*.

Bäume sind Brot- und Butter-Datenstrukturen der Informatik und sind – durchaus nicht nur in ihrer Inkarnation als Ableitungsbäume – auch in der Computerlinguistik ausgesprochen häufig. Der Hauptgrund, warum wir das hier machen, ist jedoch die Vorbereitung auf die Vorlesung Algorithmen und Datenstrukturen. Dort wird, z.B. zum Aufbau von Listen und eben Bäumen, gern mit *Pointer*n gearbeitet, also Verweisen auf Daten.

In vielen Programmiersprachen braucht man für solche Verweise eigene Datentypen, in Python ist jedoch jede „Variable“ lediglich so ein Verweis, eben eine Referenz. Würde man Python-Variablen in C simulieren wollen, wären sie am ehesten `void *`. Meine Hoffnung ist, dass nach diesem Kapitel klar wird, wie Datenstrukturen, wie sie in der Informatik-Vorlesung dargestellt werden, in Python implementiert werden können.

Ein Baum, in dem jeder Knoten maximal zwei Nachfolger hat, heißt *binärer Baum* – für viele Anwendungen sind sie ausreichend, weshalb wir unsere Untersuchungen auf sie beschränken.

Ein Knoten muss wissen, welcher Wert in ihm steht und was seine rechten und linken Kinder sind. Das realisiert folgende Python-Klasse:

```
class BinaryTree:
  def __init__(self, value):
    self.value = value
    self.right = self.left = None

  def getValue(self):
    return self.value

  def setRight(self, node):
    self.right = node

  def setLeft(self, node):
    self.left = node

  def getRight(self):
    return self.right

  def getLeft(self):
    return self.left
```

Dabei ist die Idee, dass `left` und `right` auf `None` stehen, solange der Knoten noch keine Nachfolger hat, und eine andere Klasse oder Funktion nach Bedarf weitere `BinaryTrees` über die `setLeft` und `setRight`-Methoden einhängt.

Wir haben hier sowohl einen Knoten als auch einen Baum modelliert. Das ist möglich, weil ein Baum eine selbstähnliche Datenstruktur ist – an jedem seiner Knoten beginnt eine neuer Baum, bis hinunter zu den Blättern, die letztlich Bäume mit nur einem Wurzelknoten sind.

Es wäre durchaus denkbar (und manchmal ist das auch sinnvoll), den Baum und die Knoten zu trennen, vor allem dann, wenn der ganze Baum andere Eigenschaften hat als ein Teilbaum (bei den Sortierbäumen wird das so sein) – aber wir haben hier die *Designentscheidung* getroffen, dass ein Knoten eben immer auch die Wurzel eines Baumes sein soll. Hauptgrund für die Entscheidung hier ist, dass ein paar Dinge im Code einfacher aussehen, Hauptnachteil ist, dass die Identität von Baum und Knoten etwas kontraintuitiv sein mag.

Zunächst: Arithmetische Ausdrücke in Bäumen: z.B. $(5-3)/(2\times(5+4))$. Der Ausdrucksbaum dazu:

(cf. Fig. 10)

Zur Auswertung kann man den Baum runterlaufen und von unten anfangend den Wert an jedem Knoten mit einem Operator bestimmen:

(cf. Fig. 11)

Traversierung

preorder

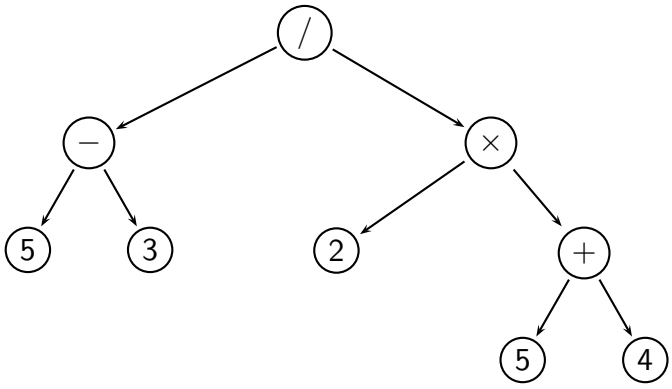


Fig. 10

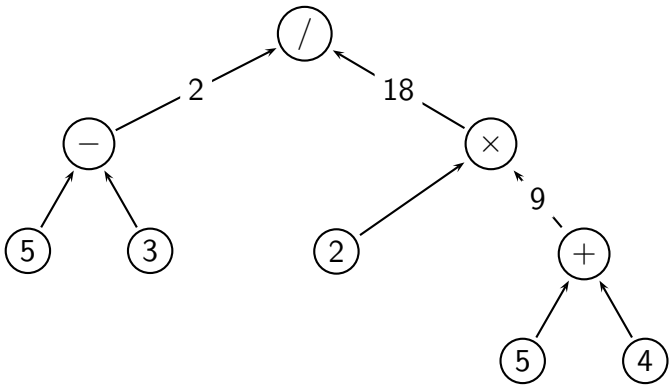


Fig. 11

Problems

(52.1)* Probiert die Klasse BinaryNode aus. Baut „per Hand“ den Ausdrucksbaum. Das könnte so aussehen:

```
bn = BinaryTree
root = bn("/")
l1 = bn("-")
root.setLeft(l1)
...
```

53. Bäume II: Traversierungen

Als erstes wollen wir uns Bäume per `print` ansehen können:

```
def __str__(self):
  nodeStrs = []
  def visitFun(node, depth, nodeStrs=nodeStrs):
    nodeStrs.append("  "*depth+
      repr(node.getValue()))
  self.traversePreorder(visitFun)
  return "\n".join(nodeStrs)
```

Was passiert in `__str__`? Eine *Traversierung* ist ein systematisches Laufen durch den Baum. Zur Repräsentation des Baums als String müssen wir genau das tun, und für jeden Knoten eine Funktion aufrufen, und zwar zur richtigen Zeit^{TM}.

Es gibt drei Möglichkeiten:

- *preorder* – in jedem Teilbaum erst den Wurzel-, dann den linken, dann den rechten Knoten bearbeiten.

- *inorder* – in jedem Teilbaum erst dann den linken, dann den Wurzel-, schließlich den rechten Knoten bearbeiten.
- *postorder* – in jedem Teilbaum erst den linken, dann den rechten Knoten, schließlich den Wurzelknoten bearbeiten.

inorder
postorder
level order
Protokoll

Bestimmte Algorithmen brauchen auch noch die *level order*-Traversierung. Dabei arbeitet man immer Ebene um Ebene (also Knoten gleicher Tiefe) ab. Das passt allerdings nicht ganz in dieses Schema.

Preorder und postorder gehen offenbar auch für nicht-binäre Bäume, inorder hat nur für Binärbäume Sinn.

Welche Sorte von Traversierung gewählt wird, hängt davon ab, was man wie gemacht haben möchte. Im Beispiel möchten wir, dass die Eltern schließlich vor den Kindern stehen.

Eine Methode zur Präorder-Traversierung könnte so aussehen:

```
  def traversePreorder(self, visitFun, depth=0):
    visitFun(self, depth)
    for node in [self.left, self.right]:
      if not node is None:
        node.traversePreorder(visitFun, depth+1)
```

Problems

(53.1)* Implementiert noch die Methoden `traversePostorder` und `traverseInorder`. Probiert, was passiert, wenn ihr `__str__` mit diesen Methoden baut.

54. Iteratoren und Generatoren

Warnung: Die folgenden Geschichten sind relativ neu. In der Regel wollt ihr Python 2.3 haben, um damit zu spielen. Siehe aber unten zu `__future__`, Iteratoren selbst gibts seit Python 2.2.

Die Callbacks in der letzten Folie sind häufig eher umständlich. Wäre es nicht einfacher, wenn man über einen Baum wie über eine Liste iterieren könnte: `for node in tree:`?

Hier hilft das Iterator-Pattern. In Python: Iteratoren implementieren eine Methode `next`, die das „nächste“ Objekt zurückgeben oder eine `StopIteration`-Exception auslösen. Aus technischen Gründen müssen Iteratoren auch noch die Methode `__iter__` unterstützen – diese gibt einfach das Objekt selbst zurück (`return self`). Man spricht dann auch davon, dass ein Objekt das Iterator-*Protokoll* implementiert.

Analog kann ein Objekt auch das Sequenzen-Protokoll und damit mindestens mal `__len__` und `__getitem__` implementieren – Python kann über ein solches Objekt „aus dem Stand“ iterieren, so dass quasi automatisch

```
for item in ob:
  ...bla...
```

in

```
for index in range(len(ob)):
  item = ob[index]
  ...bla...
```

verwandelt wird.

Richtige Iteratoren sind aber meistens schneller und fast immer eleganter.

Ein Beispiel, das eine Iteration über die ersten `seqLen` Werte der Fakultätsfunktion iteriert:

```
class FacIterManual:
  def __init__(self, seqLen):
    self.seqLen = seqLen
    self.curFac = 1
```

```
    self.curInd = 0

  def __iter__(self):
    return self

  def next(self):
    if self.curInd>=self.seqLen:
      raise StopIteration
    self.curFac = max(1, self.curInd*self.curFac)
    self.curInd += 1
    return self.curFac
```

Schreibt man `for v in ob`, versucht Python die `__iter__`-Methode von ob auszuführen und iteriert über das Ergebnis. Die eingebaute Funktion `iter` liefert einen Iterator über ihr Argument. Letzteres klappt natürlich nur, wenn das Argument tatsächlich eine `__iter__`-Methode hat oder das Sequenzenprotokoll implementiert. `iter` ist vor allem dann nützlich, wenn man einen schon bestehenden Iterator recyclen will. Vgl. auch FacIterCheat in iterobs.py (Anhang).

Generatoren

Im letzten Beispiel mussten wir `curFac` und `curInd` als Zustand im Objekt mitschleppen. Bei komplizierteren Problemen ist das lästig.

Abhilfe: Generatoren. Ein *Generator* sieht aus wie eine Funktion, nur, dass statt `return` das Schlüsselwort `yield` steht (es ist erst seit Python 2.3 ein Schlüsselwort, in Python 2.2 muss man es noch mit `from __future__ import generators` „freischalten“, vorher gab es keine Generatoren). Ruft man einen Generator auf, bekommt man einen Iterator als Ergebnis. Lest den letzten Satz nochmal. Ja, ihr solltet ein wenig verwirrt sein. Dieser Iterator sorgt quasi dafür, dass als würde bei jeder Ausführung eines `yield`-Statements die Funktion „eingefroren“ und der Ausdruck hinter `yield` als Ergebnis von `next` verwendet wird. Wenn das nächste Element der Iteration angefragt wird (also `next` aufgerufen wird), wird die Funktion „aufgetaut“ und läuft weiter, als wäre nichts geschehen, bis das nächste `yield`-Statement kommt.

Unser Beispiel mit den Fakultäten mit einem Generator:

```
class FacIterGenerator(FacIterator):
  def __iter__(self):
    curFac = 1
    yield curFac # 0! = 1
    for i in range(1, self.seqLen):
      curFac = curFac*i
      yield curFac
```

Es ist ohne Probleme möglich, Iteratoren aus Methoden zurückzugeben. Beispiel Dictionaries: Es gibt `itervalues`, `iterkeys`, `iteritems`. `iter(dict)` ist `iterkeys`.

Ein weiteres Beispiel für die Nützlichkeit von Generatoren ist `enumerate`. Diese Funktion ist ab Python 2.3 eingebaut und liefert aus einer Sequenz (oder überhaupt etwas Iterablem) einen Iterator über Tupel von (index, element) für alle Elemente der Sequenz. Es ist natürlich nicht schwer, sowas ohne Iteratoren nachzurüsten:

```
def enumerate(seq):
  return [(i, seq[i]) for i in range(len(seq))]
```

– wenn wir das so machen, müssen wir aber eine, potenziell lange, Liste erzeugen, nur, um über sie zu iterieren. Das braucht Zeit und vor allem Speicherplatz.

Eleganter geht das so (gleich mit Code, der dafür sorgt, dass, wenn enumerate schon definiert ist, wir diese Definition – hinter der sich hoffentlich schnellerer Code verbirgt – nicht überschreiben):

```
from __future__ import generators
try:
  enumerate(range(2))
except NameError:
```

```
def enumerate(seq):
  ind = 0
  for el in seq:
    yield ind, el
    ind += 1
```

Leute, die Iteratoren mögen, sollten sich weiter unbedingt das sehr schöne Modul itertools[21] (ab Python 2.3) ansehen.

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(54.1)* Studiert das angehängte Modul iterobs.py und macht euch klar, wo die Vor- und Nachteile der verschiedenen verwendeten Techniken liegen.

(54.2) Erweitert BinaryTree so, dass ihr einfach etwas wie

```
for node in someTree:
  print node.getValue()
```

schreiben könnt und danach die Knoten in preorder ausgegeben werden. Das ist hier etwas tricky, weil ihr dazu rekursive Iteratoren braucht. Ihr solltet `__iter__` als Generator implementieren, der zunächst den Node selbst `yield`et und dann das `yield`et, was die Iteratoren der Kinder zurückgeben. **(L)**

(54.3) Baut BinaryTree noch so um, dass sie Methoden `iterPreorder`, `iterPostorder` und `iterInorder` haben, die jeweils Iteratoren zurückgeben, die in der entsprechenden Traversierung über den Baum gehen.

(54.4) Im os-Modul[22] gibt es seit 2.3 die Funktion walk, die ebenfalls ein Generator ist (d.h. einen Iterator zurückgibt – in diesem Fall über alle Dateien und Verzeichnisse unterhalb eines Verzeichnisses). Schreibt z.B. ein Programm, das die Namen aller Dateien unterhalb des augenblicklichen Verzeichnisses ausgibt, in denen etwas vorkommt, was auf einen bestimmten regulären Ausdruck passt.

Vergleicht das mit der Vorgehensweise, die beim älteren `os.path.walk` aus dem os.path-Modul[23] nötig ist.

(54.5) Für Leute, die sich itertools angesehen haben: Baut einen enumerate-Generator mit itertools. **(L)**

55. Bäume III: Rechnen

Jetzt wollen wir einen Baum aus einem mathematischen Ausdruck bauen. Dazu leiten wir aus unserem Ur-Baum einen Baum ab, der die Ausdrücke gleich aus einer tokenisierten Präfix-Notation herausparst:

```
class ExpressionTree(BinaryTree):
  def __init__(self, tokens):
    try:
      val = float(tokens[0])
    except ValueError:
      val = tokens[0]
    del tokens[0]
    BinaryTree.__init__(self, val)
    if not self.isLeaf():
      self.setLeft(self.__class__(tokens))
      self.setRight(self.__class__(tokens))

   def isLeaf(self):
    """returns true if the node is the leaf of an expression tree.
    """
```

[21] http://docs.cl.uni-heidelberg.de/python/lib/module-itertools.html

[22] http://docs.cl.uni-heidelberg.de/python/lib/module-os-file-dir.html

[23] http://docs.cl.uni-heidelberg.de/python/lib/module-os.path.html

```
    return type(self.value)==type(1.0)
```

Tokenizing

lexikalischer Analyse

Tokens

In der Präfix-Notation kommt der Operator vor den Operanden (die übliche mathematische Notation heißt Infix-Notation). Aus $c \times (a + b)$ wird auf diese Weise `* c + a b`. Der Vorteil ist, dass es sehr leicht ist, aus dieser Notation einen Baum zu bauen, nicht nur, weil man sich nicht um Präzedenz kümmern muss und auch Klammern verzichtbar sind. Sprachen wie Lisp verwenden diese Art von Notation, die übrigens eng verwandt ist mit der Art, wie wir Funktionen aufrufen (`mul(c, add(a, b))`), während Forth oder Postscript eine Postfix-Notation benutzen; da heißt es dann `c a b + *`. Taschenrechner von HP werden so ähnlich bedient, und wer sich einmal dran gewöhnt hat, will andere Taschenrechner gar nicht mehr anfassen.

Der Konstruktor erwartet eine Liste von Tokens. Dabei versteht man unter *Tokenizing* oder auch *lexikalischer Analyse* die Umwandlung eines Strings in einzelne *Tokens*. Ein Token wiederum ist ein „Atom“ der Sprache, die man nachher parsen möchte, also eine unteilbare Einheit, deren innere Struktur irrelevant für den Parser ist. Ein Parser für Ausdrücke interessiert sich nicht dafür, dass 3.1415 vielleicht durch einen String der Länge Sechs repräsentiert wird oder durch 80 Bits. Was ihn interessiert, ist dass hier ein Float-Literal steht.

Um den Baum aus so einer Präfix-Notation herauszuparsen, gehen wir wie folgt vor:

Wenn noch Eingabetoken vorhanden sind

    Erzeuge einen Knoten aus dem nächsten Eingabetoken und lösche es

sonst

    Löse einen Fehler aus

Wenn der Knoten einen Operator enthält:

    Parse einen Baum aus der Eingabe und hänge ihn links ein

    Parse einen Baum aus der Eingabe und hänge ihn rechts ein

Gib den Knoten zurück

Die Entscheidung, ob ein Knoten einen Operator enthält oder nicht, ist einfach die Entscheidung, ob wir es mit einem inneren Knoten oder einem Blatt zu tun haben, denn unsere Grammatik sagt, dass literale Zahlen Blätter sind. Deshalb konditionieren wir den rekursiven Aufruf auf `isLeaf`, das seinerseits einfach nachsieht, ob unser Wert ein Float ist oder nicht.

Um jetzt den Wert des Ausdrucks zu finden, machen wir eine implizite Inorder-Traversierung wie etwa in folgender Methode:

```
  def compute(self):
    if self.isLeaf():
      return float(self.getValue())
    else:
      return eval("%f%s%f"%(
        self.left.compute(),
        self.getValue(),
        self.right.compute()))
```

Probieren wirs mal aus:

```
>>> import tree
>>> et = tree.ExpressionTree(
  "/ 8 + 3 * 4 7".split())
>>> print et
/
        8.0
        +
                3.0
                *
                        4.0
                        7.0
>>> et.compute()
```

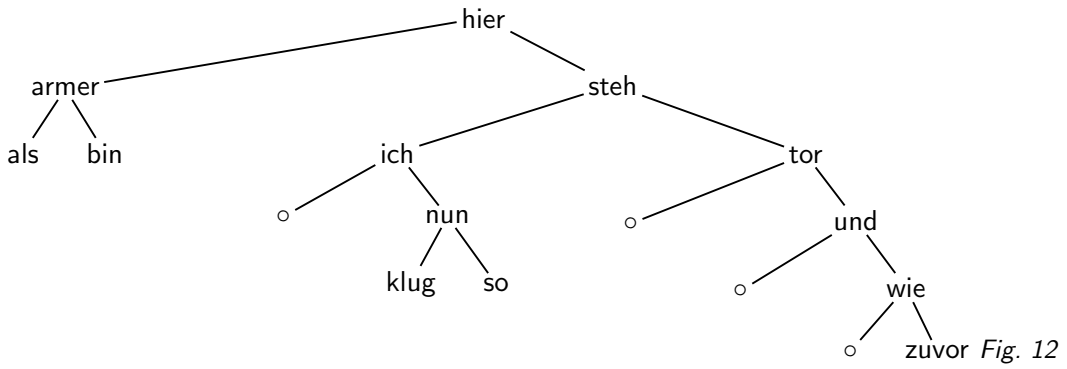


Fig. 12

```
0.25806451612903225
>>> 8./(3+4.*7.)
0.25806451612903225
```

Wir scheinen also richtig zu rechnen.

Problems

(55.1)* Macht euch klar, wie das mit den Präfix- und Postfix-Notationen aussieht. Führt die Ausdrücke `+ 3 / 4 - 4 5` und `/ * + 4 5 - 3 7 + 5 6` in die Infix-Notation um und rechnet sie aus.

(55.2)* Sammelt euch ein Modul zusammen, in der ExpressionTree leben kann und vollzieht das Beispiel nach. Überprüft eure Antworten aus der Übung (1).

(55.3) Kann das Programm ohne Änderung mit weiteren binären Operatoren laufen? Wie sieht es mit unären Operatoren aus? Wie könnten da Funktionsaufrufe reinpassen? Was müsste man ändern, um logische Ausdrücke behandeln zu können?

(55.4) Überlegt euch, wie das Programm für logische Ausdrücke mit short circuit evaluation umgeschrieben werden könnte.

56. Bäume IV: Sortierte Bäume

Bäume können gut zum Speichern und Wiederfinden von Information verwendet werden. Beispiel Wörterbuch: Ist ein Wort im Wörterbuch oder nicht?

Technik: Baue einen Baum, so dass in jedem Knoten ein Wort steht. Unter jedem Knoten befinden sich „kleinere“ Worter links, die anderen rechts.

Nach dem Lesen von „hier steh ich nun ich armer tor und bin so klug als wie zuvor“:

(cf. Fig. 12)

Problem: Sehr lange Äste brauchen lange zum Durchsuchen; Reorganisation des Baums würde zu viel flacherem Baum führen, da mehr Knoten zwei Kinder hätten.

Lösung: Ausgeglichene Bäume. Häufig verwendete Algorithmen: AVL-Bäume (immer optimal ausgeglichen, dafür komplizierte Rotationen), Rot-Schwarz-Bäume (Pfade können schlimmstenfalls um einen Faktor zwei verschieden lang sein, dafür konzeptionell einfacher; Ableitung über quartäre Bäume).

Zu diesen Themen werdet ihr in der Vorlesung Algorithmen und Datenstrukturen mehr hören.

Eine Implementation auf der Basis unserer Tree-Klasse:

```
class SortedTree(BinaryTree):
  """
  def insert(self, item):
    if self.getValue()>item:
      setMeth, getMeth = self.setLeft, self.getLeft
```

```
    elif self.getValue()==item:
      return
    else:
      setMeth, getMeth = self.setRight, self.getRight
    if not getMeth() is None:
      getMeth().insert(item)
    else:
      setMeth(SortedTree(item))
```

Lassen wirs laufen:

```
>>> import tree
>>> print tree.SortedTree("hier steh ich nun ich armer tor und bin"
...   " so klug als wie zuvor".split())
hier
        armer
                als
                bin
        steh
                ich
                        nun
                                klug
                                so
                tor
                        und
                                wie
                                        zuvor
```

Problems

(56.1)* Vollzieht anhand des tree.py-Moduls auf der Webseite alle Beispiele der letzten Seiten nach.

(56.2)* Schreibt eine Methode `has_node` für den SortedTree, die nachsieht, ob ein Wort schon mal gesehen wurde.

(56.3) SortedTree braucht eigentlich nicht mehr viel Arbeit, um als eine Art Ersatz-Dictionary arbeiten zu können, also Schlüssel-Wert-Paare zu speichern und einen (halbwegs) schnellen Zugriff darauf zu ermöglichen.

Weil unser SortedTree etwas zu schlicht ist, um sowas problemlos zu unterstützen, solltet ihr dabei direkt von BinaryTree ableiten. Lästigerweise ist das, was darin value heißt, in jetzt unser Schlüssel (key), und das, was wir jetzt Wert (value) nennen, gar nicht vorhanden.

Wir werden `value` intern nicht umbenennen wollen – definiert also zunächst im Konstruktor (der jetzt die Argumente key und value nehmen soll) einfach ein weiteres Attribut, meinetwegen `nodeValue`.

Danach solltet ihr `insert(self, node) -> None` und `retrieve(self, key) -> val`-Methoden schreiben, die ziemlich analog zu `insert` und `has_node` von `SortedTree` gestrickt sind, nur eben jeweils ein Argument mehr nehmen und ggf. alte `nodeValues` überschreiben. Beim retrieval sollten ein KeyError ausgelöst werden, wenn der angefragte Schlüssel nicht im Baum steht.

Um diese Funktionen klarer schreiben zu können, lohnt es sich wahrscheinlich, die alte `getValue`-Methode zu `getKey` umzutaufen (denkt scharf nach, in welchem Namespace das gut gehen könnte – wir haben eine solche Operation bisher noch nicht gemacht). Dazu sollte dann eine neue `setValue`-Methode kommen, während das neue `getValue` jetzt `nodeValue` zurückgeben sollte.

Als Sahnehäubchen könnt ihr dann noch Methoden `__setitem__(self, key, val) -> None` und `__getitem__(self, key) -> val` definieren. Python wird sie aufrufen, wenn ihr euer Objekt mit eckigen Klammern traktiert, also etwa so:

```
>>> t = tree.KeyValTree()
>>> t["l"] = "b"
>>> t["a"] = 1
>>> t["z"] = ['a', 'b', 'c']
>>> print t
```

```
None
        b
                ['a', 'b', 'c']
                1
>>> t["z"]
['a', 'b', 'c']
>>> t["gibtsnicht"]
Traceback (most recent call last):
<...irgendwas...>
KeyError: 'gibtsnicht'
```

(Zusatzaufgabe: Sorgt dafür, dass die Exception nicht einen kilometerlangen Traceback verursacht) **(L)**

— Dateien zu diesem Abschnitt in der HTML-Version—

57. XML I

XML (Extensible Markup Language) ist eine Familie von formalen Sprachen, die mittlerweile für alle möglichen Anwendungen benutzt wird. Beispiele sind XHTML, RDF oder sogar die neuesten Dateiformate von Microsoft-Programmen.

XML schreibt eine Syntax vor, aber (fast) keine Semantik – jedes Programm, das XML lesen kann, könnte im Prinzip Daten jedes anderen XML-erzeugenden Programmes lesen. Es wird aber in der Regel nichts damit anfangen können.

Die Syntax beschreibt im Prinzip die von kontextfreien Sprachen. Damit ist die Struktur eines XML-Dokumentes letztlich auch immer die eines Baumes.

XML-Dokumente bestehen im (ganz) Groben aus Elementen, die sich wiederum aus einem öffnenden Tag (etwas wie `<el>`), ggf. mit Attributen, eventuell einem Inhalt (andere Elemente oder Daten) und einem schließenden Tag (etwas wie `</el>`) zusammensetzen. Die Wahrheit[24] ist erheblich komplizierter. Ein Beispiel:

```
<sp who="Faust" desc='leise' xml:lang="de">
  <l>Habe nun, ach! Philosophie,</l>
  <l>Juristerei, und Medizin</l>
  <l>und leider auch Theologie</l>
  <l>durchaus studiert mit heißem Bemüh'n.</l>
</sp>
```

Wir haben ein Element `sp`, das offenbar eine Äußerung eines Theatercharakters repräsentieren soll. Im öffnenden Tag sind drei Attribute gegeben, nämlich `who`, das eine Angabe macht, wer spricht, `desc`, das eine Art Regieanweisung gibt, und `xml:lang`, das die Sprache angibt, in der die Äußerung getätigt wird. Attribute sind Schlüssel-Wert-Paare, in denen die Schlüssel aus Buchstaben bestehen (der : spielt eine spezielle Rolle, die etwa der des Punktes in Python entspricht – er trennt Namespaces –, aber das ist eher ein Thema für eine XML-Veranstaltung), die Werte aber Strings sind (sowohl einfache als auch doppelte Anführungszeichen sind erlaubt). Die einzelnen Schlüssel-Wert-Paare sind durch Whitespace getrennt.

Der Inhalt dieses `sp`-Elements besteht aus vier weiteren Elementen, deren jedes wieder Zeichen enthält.

Übrigens regelt XML auch, dass es immer genau ein Wurzelelement gibt, damit wir auch wirklich einen Baum haben.

Nochmal: Welche Elemente mit welchen zulässigen Unterelementen es gibt, wird von XML nicht festgelegt, das hängt von der Anwendung ab. XML legt nur fest, wie Tags aussehen und wie Attribute.

[24] `http://www.w3.org/XML/`

Ein „richtiges“ XML-Dokument enthält in der Regel noch weitere Angaben, etwa zum Encoding und wie ein Programm prüfen kann, ob die einzelnen Elemente auch am richtigen Platz sind. Diese Fragen sind nicht Gegenstand dieser Veranstaltung, bei Bedarf könnt ihr euch bei xmlfiles[25] oder in einem der Bücher in der entsprechenden Sektion unseres Literaturverzeichnisses[26] umsehen. SAX

Um XML-Eingaben zu verarbeiten, könnte man versucht sein, es mit Regulären Ausdrücken zu versuchen.

Das funktioniert aber nicht so einfach, weil XML-Anwendungen nicht regulär sind (sie sind normalerweise kontextfrei). Darüber hinaus hat XML jede Menge Feinheiten, die das Schreiben eines korrekten XML-Parsers schwierig machen.

„When in doubt, steal code“: Es gibt fertige XML-Parser samt Schnittstellen zu Python. Je nach Anwendungsfall kann man eine von zwei Schnittstellen im Basis-Python wählen:

- SAX – eine „ereignisorientierte“ Schnittstelle in xml.sax
- DOM – eine Schnittstelle, die ein XML-Dokument als Baum liefert, in xml.dom oder xml.dom.minidom

Das „eine von zwei“ ist natürlich dreist gelogen. Es gibt einen Haufen Schnittstellen zu XML, unter anderem auch Frederik Lundhs ElementTree, das deutlich pythonesker ist SAX oder DOM, die beide „sprachübergreifend“ definiert sind. SAX und DOM wollen wir uns hier aber gerade wegen ihrer sprachübergreifenden Natur ansehen.

Problems

(57.1) Macht euch kundig, wo überall in der Computerlinguistik XML en vogue ist. Sucht insbesondere nach Informationen über das Resource Desription Format RDF.

58. XML II: SAX

SAX (Simple API for XML) ist eine Schnittstelle zu XML-Parsern, die ähnlich in anderen Sprachen auch vorhanden ist. In ihr schickt der Parser „Ereignisse“ (Öffnender Tag, Text, Schließender Tag) an die Anwendung. Sie bietet sich an, wenn man nur Teile einer XML-Datei verabeiten möchte oder das, was man liest, unmittelbar verarbeiten kann.

Zentrale Funktion ist `xml.sax.parse`, die ein Datei und einen „Handler“ übergeben bekommt. Der Handler ist ein Objekt, das die Ereignisse in Form von Methodenaufrufen geschickt bekommt. Ggf. kann die Funktion auch noch ein Objekt zur Fehlerbehandlung übergeben bekommen. Details findet ihr in der Doku zu xml.sax[27].

Der Handler ist in der Regel eine Unterklasse von xml.sax.handler.ContentHandler. Schlichte ContentHandler ignorieren einfach alle Ereignisse. Um das zu ändern, überschreibt man Methoden, am häufigsten

- `startElement(name, attrs)` wird bei einem öffnenden Tag aufgerufen.
- `endElement(name)` wird bei einem schließenden Tag aufgerufen.
- `characters(content)` wird für Text aufgerufen

Als Beispiel für den Umgang mit XML-Daten mit SAX wollen wir XML „prettyprinten“, also ein XML-Dokument so umformatieren, dass es etwas übersichtlicher wird, Einrückung und alles. Damit die Dinge einfach bleiben, kümmern wir uns erstmal nicht um Attribute und schon gar nicht um die Frage des Whitespace-Handlings in XML[28].

Wir wollen einfach den Inhalt eines Elements gegenüber den umschließenden Tags einrücken.

[25] `http://www.xmlfiles.com`
[26] `http://wiki.cl.uni-heidelberg.de`
[27] `http://docs.cl.uni-heidelberg.de/python/lib/module-xml.sax.html`
[28] `http://www.w3.org/TR/xml11/#sec-white-space`

Stack

Was braucht es dazu? Nun, zunächst müssen wir offenbar in jedem startElement das Startelement soweit eingerückt ausgeben, wie es unserer augenblicklichen Schachtelungstiefe entspricht (diese werden wir uns also merken müssen, womit auch schon der erste Zustand der Klasse klar ist).

Bei jedem endElement müssen wir dann entsprechend „ausrücken“ und den schließenden Tag mit der Einrückung des entsprechenden öffenenden Tags ausgeben.

Tags können wir damit, nicht aber eventuelle Daten. Hier wollen wir einerseits wordwrap machen (damit nicht die Zeilenenden „hinten rausgucken“), andererseits muss gewährleistet sein, dass die Daten nicht an Tags „vorbeitrutschen“. Ersteres impliziert, dass wir nicht einfach jedes Mal, wenn wir Zeichen bekommen, diese blind auf den Schirm ausgeben wollen, zweiteres, dass wir nicht einfach alle Daten zu einem Element sammeln und geschlossen ausgeben können, wenn der schließende Tag kommt des Elements kommt – wir müssen bei jedem Tag sehen, ob Daten zur Ausgabe anstehen.

Die Lösung ist, einfach die über characters einkommenden Daten zu sammeln und bei jedem start- oder endElement-Event eine Funktion aufzurufen, die eventuell angefallene Daten ausgibt.

Abweichend vom Plan oben merken wir uns die Schachtelungstiefe nicht direkt, sondern bauen lieber einen *Stack* (hier als Liste modelliert) von Tags, so dass wir jederzeit sämtliche einschließenden Elemente im Zugriff haben könnten, wenn wir das wollten. Die Länge dieser Liste gibt die Schachtelungstiefe.

Hier brauchen wir das eigentlich nicht, aber bei der Verarbeitung kontextfreier Sprachen (in denen Verschachtelung vorkommen kann) ist der Stack eine ganz typische Datenstruktur, sie wird in SAX-Handlern also häufig vorkommen. Die Idee eines Stacks ist, dass man an einem Ende Daten anhängen kann („Daten auf den Stack legen“, push) und sie von dort auch wieder wegnehmen kann („vom Stack runternehmen“, pop), sonst aber nicht viel damit tun kann. Diese Beschränkung mag zunächst willkürlich erscheinen, sie hilft aber sehr, Dinge zu strukturieren.

Python-Listen haben die beiden Methoden schon; pop heißt pop (tatsächlich ist die Python-Version etwas allgemeiner und erlaubt durch Angabe eines optionalen Elements ein anderes als das letzte Element zu holen und gleichzeitig zu entfernen, aber genau das wollen wir hier nicht), währen push einfach unser gutes altes append ist. Wir modellieren daher den Stack einfach durch eine Liste, bei der wir *vereinbaren*, dass wir nur pop und append verwenden. Durch Verkapseln in einer Klasse könnte man diese Vereinbarung auch, nun, betonen, aber das ist zunächst nicht nötig.

Hier der entscheidende Code (der volle Code ist im Anhang dieser Seite):

```
class XmlPrettyPrinter(ContentHandler):
  def __init__(self, scrwid=79, *args):
    ContentHandler.__init__(self, *args)
    self.scrwid, self.indentSize = scrwid, 2
    self.elementStack, self.charCache = [], []

  def startElement(self, name, attrs):
    self._emitChars()
    self._printIndented("<%s>"%name)
    self.elementStack.append(name)

  def endElement(self, name):
    self._emitChars()
    self.elementStack.pop()
    self._printIndented("</%s>"%name)

  def characters(self, chars):
    self.charCache.append(chars)

  def _emitChars(self):
    contents = "".join(self.charCache).strip()
    if contents:
```

```
        self._printIndented(contents)
        self.charCache = []

parse(sys.argv[1], XmlPrettyPrinter())
```

Das komische *args klären wir auf der nächsten Folie.

Vielleicht ahnt man an dieser Stelle schon den Pferdefuß von SAX: Man muss beim Parsen unter Umständen Unmengen von State, also Informationen zum Kontext eines Elements, mitschleppen. Das wird besonders schlimm, wenn man z.B. Referenzen innerhalb eines Dokuments auflösen muss, das Dokument in nichttrivialer Weise manipulieren will oder auch nur etwas komplexere Datenstrukturen aus XML-Dateien aufbauen will. In diesen Fällen bietet sich eher DOM an.

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(58.1)* Wir haben hier den Stack einfach durch eine Liste simuliert (die pop-Methode der Listen hat uns etwas geholfen). Schöner ist es natürlich, wenn man eine richtige Stack-Klasse hat, die die Methoden `push(el) -> None` („auf den Stack legen“), `pop() -> element` („vom Stack nehmen“) und `peek() -> element` (das oberste (letzte) Element des Stacks „ansehen“, ohne den Stack zu verändern) unterstützen soll. Außerdem soll len(aStack) die augenblickliche Zahl der Elemente im Stack zurückgeben. Schreibt eine Klasse, die das tut. **(L)**

(58.2)* Schreibt einen FilteringXmlPrettyPrinter, der alle Elemente außer denen in einer Menge (bzw. den Schlüsseln eines Dictionaries) ignoriert. Man soll die Klasse also mit einem Dictionary instanzieren, und alle Elemente, deren Schlüssel nicht in dem Dictionary vorkommen, sollen (samt Inhalt!) ignoriert werden. **(L)**

(58.3) Sorgt dafür, dass auch eventuelle Attribute der Elemente ausgegeben werden (d.h. scheibt in startElement etwas rein, das sinnvoll mit dem atts-Argument umgeht). Ihr wollt dafür wahrscheinlich die Dokumentation zu ContentHandlern[29] lesen – andererseits unterstützt das Objekt, das wir über attr bekommen, im Groben alles, was Dictionaries auch können. **(L)**

59. Variable Argumente

Manchmal ist nicht klar, welche Argumente eine Funktion wohl nehmen soll. Das ist inbesondere dann so, wenn wir Methoden von Oberklassen überschreiben – wir möchten vielleicht nur die Argumente auswerten, deren Bedeutung wir kennen, und alle anderen unverändert weitergeben.

In Python gibt es dazu die Schreibweise *args. Sie sagt: Stecke alle positionalen Argmente, die nicht von formalen Parametern abgedeckt sind, in ein Tupel und binde es an args. Das kann z.B. auch verwendet werden, um Funktionen mit beliebig vielen Argumenten zu schreiben. Eine Emulation der C-Bibliotheksfunktion `printf` könnte beispielsweise so aussehen:

```
>>> def printf(format, *args):
...     print format%args,
...
>>> printf("%s: %d\n", "gezählt", 2)
gezählt: 2
>>> printf("Das ist %s vo%s\n", "das Haus",
  "m Nikolaus")
Das ist das Haus vom Nikolaus
```

Analog bekommt ein Formalparameter, der mit zwei Sternchen gekennzeichnet ist, alle nicht schon „verbrauchten“ Keyword-Argumente in einem Dictionary zugewiesen, in dem jedem übergebenen Namen der entsprechende Wert zugeordnet ist.

```
>>> def makedict(**kwargs):
...     return kwargs
```

[29] http://docs.cl.uni-heidelberg.de/python/lib/content-handler-objects.html

```
...
>>> makedict(schluss=1, muss=2, rhein=3, fein=4)
{'muss': 2, 'fein': 4, 'schluss': 1, 'rhein': 3}
```
– was natürlich so nur geht, wenn wir Dictionaries mit als Python-Namen zugelassenen Strings als Schlüsseln bauen wollen.

ungebunden
unbound
DOM

Im Allgemeinen kann man für den Aufruf von Methoden der Oberklasse, deren Argumentlisten wir Umständen nicht kennen (wollen), folgendes Pattern verwenden:

```
class Daughter(Mother):
  def __init__(self, myarg, *args, **kwargs):
    Mother.__init__(self, *args, **kwargs)
    self.myarg = myarg
```

Da Klassen auch Objekte sind, liefert Mother.__init__ einfach eine Referenz auf diese Methode. Da sie aber zur Klasse (und nicht zu einer Instanz) gehört, ist sie *ungebunden* (*unbound*). Deshalb müssen wir die konkrete Instanz (`self`) selbst übergeben.

60. XML III: DOM

Das *DOM* (Document Object Model) ist eine alternative Schnittstelle zu XML-Daten. In ihr baut eine Bibliothek einen Baum aus dem Dokument (warum geht das?). Diesen Baum kann man dann manipulieren.

Vorteil des DOM ist, wie gesagt, dass zur Auswertung von Daten, die in einem DOM-Baum gehalten werden, meist weit weniger State durch das Programm geschleppt werden muss als bei SAX. Hauptnachteil ist, dass DOM (fast) immer das ganze Dokument in den Speicher ziehen muss (siehe aber auch xml.dom.pulldom[30]).

Leider ist das scheinbar so einfache XML doch so kompliziert, dass auch DOM ein recht komplizierter Standard geworden ist. Zum Glück kommt man für die meisten Anwendungen mit einer kleinen Untermenge aus. Eine weitere gute Nachricht ist, dass DOM in moderne Web-Browser eingebaut ist und Kenntnisse von DOM helfen, ganz tolle, bunte und dynamische Webseiten zu bauen. Das soll man zwar eigentlich aus vielen Gründen nicht tun, aber Spaß macht es doch...

Als Anwendung für DOM wollen wir uns eine hypothetische lexikalische Datenbank ansehen, in der Lemmata, Lesungen und Lexikalische Relationen vermerkt werden. Dabei besteht ein Lemma aus mehreren Lesungen, zwischen denen Relationen wie Hypernymie, Antonymie usf bestehen können. So eine Datei könnte dann etwa folgendermaßen aussehen:

```
<lexdata>
<lemma name="Wissenschaft" id="l1">
<lesung id="l1_1"><def>Was Schlaues</def>
  <lexrel type="anto" idref="l2_1"/>
  <lexrel type="anto" idref="l2_2"/>
</lesung>
<lesung id="l1_2"><def>Was weniger Schlaues</def>
  <lexrel type="hyper" idref="l2_1"/>
  <lexrel type="hyper" idref="l2_2"/>
</lesung>
</lemma>
<lemma name="Wissenschaftspolitik" id="l2">
<lesung id="l2_1">
  <def>Was extrem Dummes</def>
  <lexrel type="anto" idref="l1_1"/>
  <lexrel type="hypo" idref="l1_2"/>
</lesung>
<lesung id="l2_2">
```

[30] http://docs.cl.uni-heidelberg.de/python/lib/module-xml.dom.pulldom.html

```
  <def>Wichtige Leute machen totalen Quatsch</def>
  <lexrel type="anto" idref="l1_1"/>
  <lexrel type="hypo" idref="l1_2"/>
</lesung>
</lemma>
</lexdata>
```

Wir haben dabei von einer weiteren Standardeigenschaft von XML Gebrauch gemacht: id/idref. Man kann Elementen ein id-Attribut geben, das von bestimmten XML-Technologien speziell ausgewertet wird. Im Effekt gibt man dem Element einen Schlüssel, über den wir nachher schnell und einfach zugreifen können – insbesondere können wir damit quer über verschiedene Äste des Baumes referenzieren. ids spielen z.B. in XSLT eine Rolle, DOM hat ab Level 2 (den minidom nicht implementiert, aber siehe unten) eine Methode `getElementById`, die zu einer Id das passende Element liefert. Nebenbei bemerkt ist es natürlich ein Fehler, wenn zwei verschiedene Elemente die gleiche id tragen.

Wie Profis Daten dieser Art verwalten, könnt ihr übrigens beispielsweise auf der Wordnet-Seite[31] nachlesen.

Was wir brauchen, steht in xml.dom.minidom[32]. Die `parse`-Funktion aus diesem Modul nimmt einen Dateinamen (oder eine Datei) und gibt (im Wesentlichen) ein `xml.dom.Document`-Objekt zurück. Dieses modelliert einen Baum, dessen Wurzelobjekt im `documentElement`-Attribut steht.

Die Knoten des Baumes stehen in `Node`-Objekten; sie haben eine Unzahl von Attributen (auf die man hier getrost lesend zugreifen darf). Elemente stehen in `Element`-Objekten, die von `Node` abgeleitet sind. Sie unterstützen z.B. die `getAttribute`-Methode.

Genaueres in der Dokumentation zu xml.dom[33]. Dort gibt es auch Links auf die DOM-Spezifikationen des W3C.

Ein paar Funktionen, die die Suche nach Lesungen erlauben, die in einem bestimmten semantischen Verhältnis `targetLexRel` zu den Lesungen eines Lemmas stehen:

```
def findLemma(rootNode, lemma):
  for node in nodeIter(rootNode):
    if (node.nodeName=="lemma" and
      node.getAttribute("name")==lemma):
      return node

def getRelLexForLesung(lesungNode, targetLexRel,
  lexDb):
  result = []
  for lexrel in lesungNode.getElementsByTagName(
    "lexrel"):
    if lexrel.getAttribute("type")==targetLexRel:
      result.append(lesungAsStr(
        lexDb.getElementById(lexrel.getAttribute(
        "idref"))))
  return result

def findRelLemma(lemma, targetLexRel, lexDb):
  relLes = []
  lemmaNode = findLemma(lexDb.documentElement,
    lemma)
  if lemmaNode:
    for lesung in lemmaNode.getElementsByTagName(
      "lesung"):
```

[31] http://www.cogsci.princeton.edu/~wn/

[32] http://docs.cl.uni-heidelberg.de/python/lib/module-xml.dom.minidom.html

[33] http://docs.cl.uni-heidelberg.de/python/lib/module-xml.dom.html

```
      lesRelLes = getRelLexForLesung(lesung,
        targetLexRel, lexDb)
      if lesRelLes:
        relLes.extend([(lesungAsStr(lesung), a)
          for a in lesRelLes])
  return relLes
```

(die kompletten Quellen sind im Anhang dieser Seite).

In `findLemma` traversieren wir den Baum, bis wir ein `lemma`-Element gefunden haben, dessen `name`-Attribut gerade das gesuchte Lemma ist. Wir gehen dabei vom `rootNode` aus, also dem `documentElement` des DOM-Objekts, das wir von `parse` zurückbekommen. Die hier verwendete Funktion `nodeIter` ist dabei ein Generator, den man folgendermaßen definieren kann:

```
def nodeIter(aNode):
  """returns an iterator over all nodes below aNode
  """
  yield aNode
  for node in aNode.childNodes:
    for child in nodeIter(node):
      yield child
```

– DOM-Objekte haben sowas nicht als Methode, weil diese Sorte Generator nicht in allen Sprachen verfügbar ist und DOM den Anspruch der Sprachunabhängigkeit hat.

Die Funktion `getRelLexForLesung` soll eine Liste von String-Repräsentationen für Lesungen zurückgeben, die in der Relation (also etwa „anto“, „syn“, „hypo“ usf.) `targetLexRel` zur Lesung `lesungNode` stehen. Es iteriert dazu über eine Liste aller Nodes (das sind dann automatisch Elemente) mit Tagnamen `lexrel` (die entsprechende Methode ist Teil von DOM), prüft für jeden, ob das typ-Attribut gleich der gesuchten Relation ist und hängt, wenn das so ist, eine String-Repräsentation des „Ziels“ der entsprechenden Relation.

Wir brauchen zum Finden des Ziels die oben erwähnte `getElementById`-Methode. Da das hier verwendete minidom-Modul diese (noch) nicht bereitstellt, müssen wir sie quasi per Hand nachrüsten. Der Umstand, dass man in Python-Elementen im Nachhinein wild rumrühren kann, hilft uns hier – man sollte Entsprechendes aber wirklich nur dann tun, wenn alles andere vergeblich war:

```
def _makeIdCache(domOb):
  """adds a getElementById-Method to the DOM object domOb.  This is
  a bad hack and will be obsoleted when minidom becomes DOM level 2
  compliant.
  """
  idDict = {}
  for node in nodeIter(domOb.documentElement):
    try:
      idDict[node.getAttribute("id")] = node
    except (AttributeError, KeyError):
      pass
  domOb._mdsIdDict = idDict
  domOb.getElementById = new.instancemethod(
    lambda self, id: self._mdsIdDict.get(id, None), domOb)
```

Interessant ist hier vielleicht, dass wir (1) für das Dictionary, das von ids auf die zugehörigen Elemente zeigt, einen Namen verwendet haben, der es unwahrscheinlich macht, dass wir ein Attribut, das die Document-Klasse selbst definiert, überschreiben (`_mdsIdDict`) und (2) dem Objekt eine neue Methode unterschieben. Das ist etwas aufwändig, weil wir die Methode an das Objekt „binden“ müssen. Wir verwenden dafür die `instancemethod`-Funktion aus dem `new`-Modul. Die Details würden den Rahmen dieses Skripts sprengen – nehmt es als Hinweis, dass Python immer noch viele spannende Ecken hat, von deren Existenz ihr bisher noch nichts gehört habt...

Fig. 13

Toolkit

Widget Set

Die `findRelLemma`-Funktion schließlich kombiniert die anderen Funktionen schließlich so, dass sie schließlich sämtliche Ziele der gesuchten Relationen in einer Liste zurückgeben kann.

— Dateien zu diesem Abschnitt in der HTML-Version—

61. GUIs

GUI steht für Graphical User Interface – gemeint sind WIMP (Windows, Icons, Menus, Pointer). GUIs werden in der Regel unter Rückgriff auf Bibliotheken geschrieben (*Toolkit*, *Widget Set*), die schon wissen, was ein Knopf und was ein Menü ist. Das immer noch verbreitetste Toolkit für Python ist Tk.

Tk wurde ursprünglich als Ergänzung zur Skriptsprache tcl entwickelt und ist – wenigstens im Vergleich zu anderen Toolkits – relativ einfach zu programmieren.

Python unterstützt eine Vielzahl weiterer Toolkits, angefangen von Gtk (dem unter Linux wohl verbreitetsten Toolkit) über Qt (dem Toolkit hinter KDE) bis zu Exoten wie Fltk – wer Tk verstanden hat, kann sich in die anderen Toolkits meist recht schnell einarbeiten.

Alles, was zu Tk gehört, befindet sich im Modul Tkinter. Ein Programm, das auf Tkinter aufsetzt, definiert meist eine Klasse, die von Tkinter.Tk oder Tkinter.Toplevel erbt:

```
import Tkinter

class Beeper(Tkinter.Tk):
  def __init__(self):
    Tkinter.Tk.__init__(self)
    Tkinter.Button(self, text="Beep",
      command=self.beep).pack()
    Tkinter.Button(self, text="Quit",
      command=self.quit).pack()

  def beep(self):
    print "Beep"

if __name__=="__main__":
  b = Beeper()
  b.mainloop()
```

(cf. Fig. 13)

Fig. 14

Widget

Callback

62. Widgets

Ein *Widget* ist ein (meist sichtbares) GUI-Element. In Python gehört zu jedem Widget eine Klasse. Im Beispiel verwenden wir das Widget `Button`, einen Knopf mit Label (das `text`-Argument) und einer Aktion, die ausgeführt wird, wenn der Knopf gedrückt wird (das `command`-Argument).

Eine Funktion, die aufgerufen wird, wenn sich an einem Widget etwas tut (Knopf gedrückt, Scrollbar gezogen, Knopf gedrückt usf.), heißt *Callback* – mit dem `command`-Argument setzt man also einen Callback

Tk hat etliche vordefinierte Widgets, unter anderem

- `Toplevel`: Ein eigenes Fenster in der GUI
- `Label`: Ein Stück fester Text
- `Entry`: Eine Eingabezeile
- `Text`: Ein einfacher Texteditor
- `Checkbutton`: Ein Feld zum „anchecken“
- `PhotoImage`: Ein Bild
- `Scrollbar`: Eine Scrollbar eben
- `Listbox`: Eine Liste mit Auswahlmöglichkeit
- `Menu`: Pulldown- oder Popup-Menüs
- `Menubar`: Eine Menüzeile

Daraus lassen sich z.B. durch Ableitung eigene Widgets erzeugen.

Widgets werden entweder bei der Instanzierung durch Keyword-Parameter oder nachher durch die allen Widgets gemeinsame Methode `config` konfiguriert (z.B. Text auf einem Button ändern).

```
b = Button(root, text="Press", command=bla)
b.config(text="Here")
```

Jedes Widget außer Toplevel und Tk hat ein Elterwidget, das als erstes Argument übergeben wird. In diesem Elterwidget wird das Widget in der Regel dargestellt.

(cf. Fig. 14)

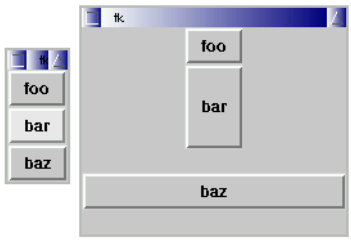


Fig. 15

63. Geometry Management

Wenn wir ein Widget instanzieren, erscheint es noch nicht auf dem Schirm (Ausnahmen: Tk, Toplevel) – es weiß ja nicht, wo.

Position und Größe von Widgets wird von einem *Geometry Manager* gesteuert. Im Tk gibt es davon drei:

- grid: Positionierung in Zeilen und Spalten
- pack: „Aneinanderpacken“ in horizontaler oder vertikaler Richtung
- place: Positionierung an feste Koordinaten

In Tkinter sind die Geometriemanager als *Mixins* realisiert, als Klassen, die in die eigentlichen Widgets eingebettet sind („reingemischt“). Daher erscheinen die Methoden zum Geometriemanagement als Methoden der Widgets. In Wirklichkeit *delegieren* die Widgets die entsprechenden Aufrufe an den Geometriemangager, rufen also einfach direkt dessen Methoden auf.

In anderen Toolkits tauchen die Geometriemanager expliziter auf und sind häufig eigene Klassen, denen man auf die eine oder andere Art sagt, dass sie sich um ein Widget kümmern sollen.

Um ein Widget einem Geometriemanager zu unterstellen, ruft man eine der Methoden `grid`, `pack` oder `place` auf. Innerhalb eines *Container*s (das ist ein Widget, das andere Widgets enthält; bisher kennen wir `Tk`, häufiger wird man aber `Frame` verwenden) sollte immer nur ein Geometriemanager verwendet werden.

```
import Tkinter
root = Tkinter.Tk()
b1, b2, b3 = [Tkinter.Button(root, text=label)
  for label in ["foo", "bar", "baz"]]
b1.pack()
b2.pack(expand=1, fill=Tkinter.Y)
b3.pack(expand=1, fill=Tkinter.X)
root.mainloop()
```

(cf. Fig. 15)

Aufrufe an `pack` nehmen verschiedene Optionen. Die wichtigsten sind

- `side` – entweder `Tkinter.TOP` (default, die Widgets werden übereinandergepackt) oder `Tkinter.LEFT` (die Wigets werden nebeneinandergepackt)
- `expand` – wenn das Elterwidget mehr Platz bekommt, bekommt auch das Widget mehr Platz (1) oder nicht (0).
- `fill` – wenn das Widget mehr Platz bekommt, wächst es nicht mit (`Tkinter.NONE`), horizonal (`Tkinter.X`), vertikal (`Tkinter.Y`) oder in beide Richtungen (`Tkinter.BOTH`)

Auch bei den anderen Geometriemanagern gibt es Optionen, nur sehen sie eben anders aus. Wer etwas mit Tk machen will, sollte sich auf jeden Fall neben dem Pack-Manager auch noch den Grid-Manager ansehen.

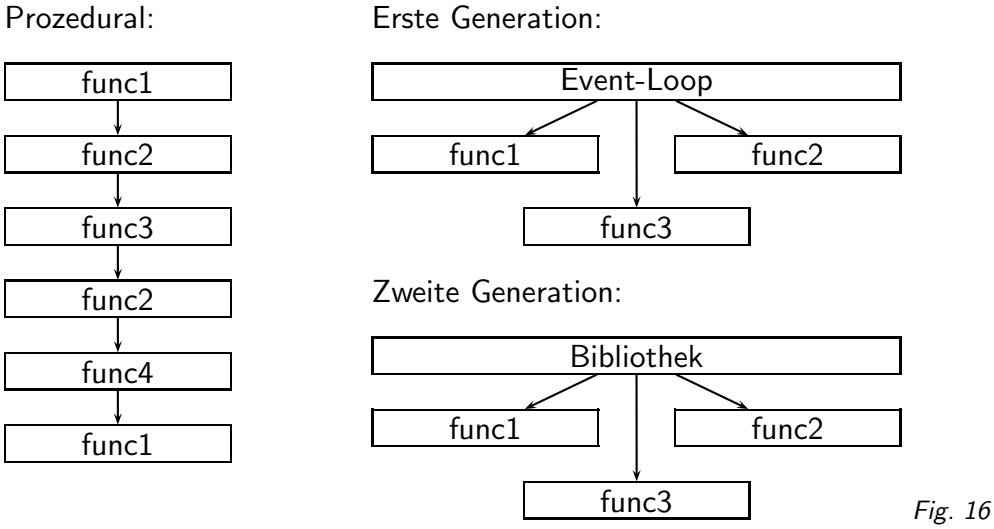


Fig. 16

64. Eventorientiertes Programmieren

Ein normales Programm ruft Funktionen in vorgegebener Reihenfolge auf. Für GUI-Programme ist das nicht gut – BenutzerInnen können zu jeder Zeit irgendwelche Menüeinträge anklicken oder Knöpfe drücken oder den Fenster-zu-Knopf drücken.

Stattdessen: Eventorientierte Programmierung.

(cf. Fig. 16)

In GUIs der ersten Generation lief im Programm ein „Event Loop“, der von einer Systemfunktion gesagt bekommen hat, ob es irgendwo einen Mausklick gab oder einen Tastendruck oder ob etwas neu gemalt werden musste.

GUIs der zweiten Generation haben den Event Loop schon eingebaut – man sagt ihnen nur noch, was passieren soll, wenn ein ein Menüeintrag gewählt ist, irgendwas neu gemalt werden soll oder sich das Programm beenden soll.

Tk ist ein Toolkit der zweiten Generation.

Struktur eines Tkinter-Programms

1. Ein Widget der (Ober-) Klasse Tkinter.Tk instanzieren, das Hauptfenster
2. Widgets des Programms erzeugen und dem Geometriemanager des Hauptfensters oder eines seiner Kinder unterstellen und/oder
3. Weitere Widgets der (Ober-) Klasse Tkinter.Toplevel erzeugen (weitere Fenster der Applikation)
4. Die `mainloop`-Methode des Hauptfensters aufrufen

Die `mainloop`-Methode sorgt dafür, dass die Callbacks aufgerufen werden und kehrt erst zurück, wenn die `quit`-Methode des Widgets aufgerufen wird.

65. Events

Event

Der Callback, den Buttons im command-Parameter nehmen, ist nur ein Spezialfall. Generell können Widgets fast beliebige *Event*s verarbeiten. Mögliche Event-Typen sind u.a.:

- '<Button-1>' – der erste Mausknopf wurde gedrückt.
- '<Double-Button-2>' – der zweite Mausknopf wurde doppelgeklickt
- '<Enter>' – der Mauszeiger ist in das Feld des Widgets geführt worden
- '<Key>' – eine Taste wurde gedrückt
- '<Configure>' – das Widget wurde umkonfiguriert

Die Strings oben können als erstes Argument im Aufruf der bind-Methode von Widgets dienen. Das zweite Argument ist der Callback, der wiederum als Argument ein Event-Objekt bekommt, in dem unter anderem der Event-Typ, das Widget, auf dem das Event ausgelöst wurde, die Mausposition und ggf. die gedrückte Taste steht.

Ein etwas albernes Beispiel für Eventverarbeitung: Ein Widget, in dem ein Quit-Knopf umherhüpft, sobald der Mauszeiger es erreicht und der durch Eingabe von „komm“ unter den Mauszeiger gezwungen werden kann. Es ist nicht so, dass bei GUI-Programmen Leerzeilen unüblich wären, der Platz auf der Folie hat einfach nicht gereicht.

```
import Tkinter, random

class ButtonJump(Tkinter.Frame):
  def __init__(self, master, **kwargs):
    apply(Tkinter.Frame.__init__, (self, master),
      kwargs)
    self.bt = Tkinter.Button(self, text="Quit",
      command=self.quit)
    self.bt.bind('<Enter>', self.doJump)
    self.keyStrokes = []
    self.bind('<Key>', self.doKey)
    self.doJump(); self.focus()
  def doKey(self, event):
    self.keyStrokes.append(event.char)
    if len(self.keyStrokes)>4:
      del self.keyStrokes[0]
    if self.keyStrokes==['k','o','m','m']:
      self.bt.bind('<Enter>', lambda ev, s=self:
       s.bt.bind('<Enter>', s.doJump))
      self.bt.place_forget()
      self.bt.place(x=event.x, y=event.y,
        anchor=Tkinter.CENTER)
  def doJump(self, event=None):
    self.bt.place_forget()
    self.bt.place(relx=random.random(), rely=
    random.random(), anchor=Tkinter.CENTER)

root = Tkinter.Tk()
ButtonJump(root, width=300, height=300).pack(
  expand=1, fill=Tkinter.BOTH)
root.mainloop()
```

66. Ein fast nützliches Programm I

Wir wollen einen kleinen Editor schreiben, der verschiedene Encodings kann und erlaubt, experimentell zu bestimmen, in was für einem Encoding eine Datei geschrieben ist. Dazu brauchen wir zunächst einen `Text` mit Scrollbalken. Kombinationen von Widgets packt man meist in einen `Frame`. Also:

```
class ScrollableText(Tkinter.Frame):
  def __init__(self, master, *args, **kwargs):
    Tkinter.Frame.__init__(self, master,
      *args, **kwargs)
    self.encoding = "iso-8859-1"
    self.textField = Tkinter.Text(self, width=60,
      height=20, wrap=Tkinter.NONE)
    self.scrollVert = Tkinter.Scrollbar(self,
      command=self.textField.yview)
    self.scrollHorz = Tkinter.Scrollbar(self,
      command=self.textField.xview,
      orient=Tkinter.HORIZONTAL)
    self.textField.config(xscrollcommand
      =self.scrollHorz.set,
      yscrollcommand=self.scrollVert.set)

    self.textField.grid(row=0, col=0, sticky
      =Tkinter.N+Tkinter.S+Tkinter.W+Tkinter.E)
    self.scrollVert.grid(row=0, col=1,
      sticky=Tkinter.N+Tkinter.S)
    self.scrollHorz.grid(row=1, col=0,
      sticky=Tkinter.W+Tkinter.E)
    self.columnconfigure(0, weight=1)
    self.rowconfigure(0, weight=1)
    self._doBindings()
```

Anmerkungen:

- Wir verwenden zum Aufruf des Konstruktors der Elterklasse das oben vorgestellte Pattern.
- `Tkinter.Text` hat haufenweise Optionen, wir stellen hier die Anfangsgröße und das Verhalten bei zu langen Zeilen ein. Da wir auch horizontal Scrollen können, soll gar nicht umgebrochen werden.
- Die Scrollbars haben einen Callback `command`. Wenn an ihnen rumgeschoben wird, rufen sie diesen Callback auf, um die Änderungen dem von ihnen gesteuerten Widget mitzuteilen. In unserem Fall hat `Text` schon Methoden, die genau so gemacht sind, wie Scrollbar das braucht, nämlich `yview` und `xview`.
- Umgekehrt muss auch der `Text` die Möglichkeit haben, den Scrollbars zu sagen, wenn sich was am dargestellten Ausschnitt ändert, etwa, weil mit der Tastatur gescrollt wurde oder weil der Text sich geändert hat. Dafür dienen die `.scrollcommand`-Methoden, die wir im Nachhinein durch `config` setzen.
- Wir verwenden hier den grid-Geometriemanager. Dessen `sticky`-Option dient etwa dem gleichen Zweck wie die `fill`-Option des pack-Managers, nur dass hier angegeben werden kann, an welchen Grenzen einer Zelle das Widget „kleben“ soll, und zwar nach Himmelsrichtung. Ein Widget, das mit `sticky=Tkinter.N` gegridet wurde, wird immer mit der Oberkante seiner Zelle abschließen.
- Ein Äquivalent der expand-Option des pack-Managers hat grid nicht – das ginge auch nicht, weil ja alle Zellen einer Spalte bzw. Zeile in gleicher Weise wachsen müssen. Daher kann das Wachstum auch nur zeilen- oder spaltenweise festgelegt werden. Genau das tun row- bzw. columnconfigure. Hier legen wir einfach fest, dass das ganze Wachstum des Containers auf Spalte und Zeile 0 gehen soll, eben dort, wo das `Text` ist.

- Wir wollen das Verhalten des Standard-`Text`-Widgets noch ändern. Dazu werden wir Bindings verwenden, wollen den Code dazu aber aus dem Konstruktor draußen haben und lagern ihn in die Funktion `_doBindings` aus.
- Außerdem soll unser Text-Widget gegenüber dem Text-Widget aus Tkinter auch um Encodings wissen – letzteres nimmt an, dass es (im Groben) Unicode-Strings bekommt. Dazu machen wir uns ein Attribut `encoding`, in dem wir das augenblicklich verwendete Encoding speichern.

Wir delegieren das Holen und Setzen der Texte in unserem Widget an `Text` und kümmern uns ums Encoding:

```
  def getText(self):
    return self.textField.get(1.0,
      Tkinter.END).encode(self.encoding)

  def setText(self, tx):
    utx = tx.decode(self.encoding)
    self.textField.delete(1.0, Tkinter.END)
    self.textField.insert(1.0, utx)

  def setEncoding(self, encoding):
    tx = self.getText()
    oldEnc = self.encoding
    try:
      self.encoding = encoding
      self.setText(tx)
    except UnicodeDecodeError:
      self.encoding = oldEnc
      raise
```

Anmerkungen:

- Hier verwenden wir unser Encoding-Attribut, um zwischen dem vom Text-Widget verwendeten Unicode und dem von der einbettenden Anwendung verwendeten Encoding (das wir auf den in Westeuropa sicheren Fallback iso-8859-1 gesetzt haben) zu übersetzen. Das ist nicht ganz ungefährlich, weil nicht alles in allem kodiert werden kann. Die encode- und decode-Methoden können Exceptions werfen. Diese geben wir hier einfach an die einbettende Anwendung weiter, die das irgendwie behandeln sollte (wir tun das im Beispielprogramm nicht). Dadurch, dass wir zunächst dekodieren und dann erst den alten Text löschen, vermeiden wir, dass, wenn das Dekodieren nicht möglich sein sollte, gar kein Text mehr im Widget steht.
- In setEncoding müssen wir Dekodierungsfehler aber selbst behandeln. Wenn nämlich nicht dekodiert werden kann, steht der Text immer noch im alten Encoding im Widget. Deshalb müssen wir das alte Encoding speichern und es restaurieren, wenn der Wechsel nicht geklappt haben sollte. Die Exception müssen wir aber trotzdem an die einbettende Anwendung weitergeben – in unserem Beispiel müsste dann die Anzeige des Encodings auf den alten Wert gesetzt werden. Ich haben das nicht gemacht. Probiert es selbst (ihr braucht dafür wahrscheinlich eine Methode `getEncoding` von `ScrollableText`; nützlich dabei ist, dass ihr `useEncoding` setzen könnt und die Radiobuttons automatisch den neuen Zustand reflektieren, siehe unten).
- Methoden wie `get`, `insert` oder `delete` des `Text`-Widgets von Tkinter können auch nur Teile des Textes bearbeiten. Deshalb nehmen sie Argumene wie 1.0 („Erste Zeile, Nulltes Zeichen“, also Anfang des Textes) oder `Tkinter.END`, das sich, egal wie viel Text da ist, immer auf das Ende des Textes bezieht.

Die `_doBindings`-Methode soll hier – nur als Beispiel – das Scrollen mit dem Mausrädchen unter X (wo die Bewegung des Rädchens in Mausklicks mit den imaginären Maustasten 4 und 5 übersetzt wird) aufsetzen:

```
  def _doBindings(self):
```

```
    self.textField.bind("<Button-4>", lambda ev,
      self=self: self.textField.yview(
      Tkinter.SCROLL, -1, Tkinter.UNITS))
    self.textField.bind("<Button-5>", lambda ev,
      self=self: self.textField.yview(
      Tkinter.SCROLL, 1, Tkinter.UNITS))
```

Anmerkung: Es ist nicht immer ganz einfach, zu sehen, an welche Widgets Bindings kommen sollen, und in der Tat sind die Regeln, wer alles Events zum Prozessieren vorgelegt bekommt, nicht einfach. Für Maus-Events ist das in aller Regel das Widget, das gerade „direkt“ unter dem Mauszeiger liegt, nicht aber eventuelle Container. Für Tastaturevents hat wenigstens X einen *Focus*, eben das Widget, das diese Events bekommt. Wie das funktioniert, will ich hier nicht erklären, die Frage selbst ist jedoch unter Umständen sehr relevant. In der Tkinter-Doku[34] erfährt man einiges dazu u.a. im Kapitel über Dialog Windows.

67. Ein fast nützliches Programm II

Wir wollen jetzt das Hauptfenster mit einer Menüzeile, einer Instanz von ScrollableText und einer Leiste mit Radiobuttons erzeugen:

```
class Editor(Tkinter.Tk):
  def __init__(self, initFile=None, *args,
    **kwargs):
    apply(Tkinter.Tk.__init__, (self,)+args,
      kwargs)
    self._generateMenu()
    self.text = ScrollableText(self)
    self.text.pack(expand=1, fill=Tkinter.BOTH)
    self._generateEncodingFrame().pack(expand=1,
      fill=Tkinter.X)
    if initFile:
      self.openFile(initFile)
```

Anmerkungen:

- Wir erben hier von `Tkinter.Tk`. Das hat den Vorteil, dass wir Tk nicht anderweitig initialisieren müssen, aber den Nachteil, dass wir nicht mehrere Fenster gleichzeitig aufmachen können. Würden wir von `Tkinter.Toplevel` erben, hätten wir dieses Problem nicht (aber dafür andere, die nicht hierher gehören).
- Toplevel-Windows (also auch `Tk`) können eine Menüzeile haben. Wir werden sie in der `_generateMenu`-Methode erzeugen.
- Den `ScrollableText` packen wir so, dass er das Fenster so gut wie möglich ausfüllt.
- Wir werden noch einen zusätzlichen Frame erzeugen; da die Buttons darin jedenfalls vertikal nicht sinnvoll wachsen können, sorgen wir beim Packen dafür, dass sie es auch nicht tun.
- Schließlich erlauben wir noch, dass bereits beim Konstruieren unseres Anwendungsfensters eine Datei geöffnet werden kann.

Die Radiobuttons erzeugen wir so:

```
  def _generateEncodingFrame(self):
    encodingFrame = Tkinter.Frame(self)
    self.useEncoding = Tkinter.StringVar()
    encodings = ["iso-8859-1", "iso-8859-2",
      "iso-8859-3", "CP437", "UTF-8"]
    for enc in encodings:
      Tkinter.Radiobutton(encodingFrame,
```

[34] `http://docs.cl.uni-heidelberg.de/python/tkinter/index.htm`

```
        text=enc, variable=self.useEncoding,
        value=enc, command=self._changeEncoding
      ).pack(side=Tkinter.LEFT)
    self.useEncoding.set(encodings[0])
    self._changeEncoding()
    return encodingFrame
```

Anmerkungen:

- Es wäre denkbar gewesen, hier eine neue Klasse zu definieren. In der Tat wäre das sogar recht schlau, die Klasse könnte dann Methoden wie `getEncoding` oder `setEncoding` haben und also das Verhalten „Wähle eine von mehreren Optionen aus“ sinnvoll verkapseln. Hier erzeugen wir stattdessen recht ad hoc einen `Frame`.
- Radiobuttons sind GUI-Elemente, von denen immer nur einer zu einer Zeit „aktiv“ (also ausgewählt) sein kann. Da wir immer nur ein Encoding zu einer Zeit anzeigenk können, ist dies eine ganz brauchbare Metapher.
- Radiobuttons in Tkinter sind immer an einer tcl-Variable gebunden, die die augenblickliche Auswahl reflektiert und die (cool Feature!) Änderungen an sich auch an die Radiobuttons weitergibt. Diese tcl-Variablen bekommt man durch Aufruf von `Tkinter.StringVar`. Wir werden diese Variable später brauchen und speichern daher eine Referenz auf sie in einer Instanzvariablen.
- Wenn man die Radiobuttons konstruiert, muss man (a) das Elterwidget, (b) das Label, das hinter dem Radiobutton steht, (c) die tcl-Variable, die von den Buttons gesteuert wird und (d) den Wert, den die Variable annimmt, wenn der Radiobutton ausgewählt wird angeben.
- Ein Satz von Radiobuttons mit einer gemeinsamen Variable heißt Gruppe, und es kann immer nur ein Radiobutton aus einer Gruppe selektiert sein – daher auch der Name: Bei alten Radios gibt es Knöpfe, die genauso funktionieren: Wird einer gedrückt, springen alle anderen raus.
- Eine tcl-Variable ist ganz anders als eine Python-Variable – ihr wird nicht zugewiesen (in Python: Verbindung eines Namens mit einem Wert), ihre Werte werden mit einer Methode `set` gesetzt (ein halbwegs brauchbares Analogon dafür in Python ist das Setzen eines Listenelements: `a[2] = 3` – `a` bleibt dabei in gewisser Weise unverändert, nur der an `a` gebundene Wert ändert sich).

Wir setzen auch für die Radiobuttons einen Callback, und zwar einen für alle. Wir müssen in diesem Callback eigentlich nur dem Text-Widget Bescheid sagen, dass es sein Encoding ändern muss, etwa so:

```
  def _changeEncoding(self):
    self.text.setEncoding(self.useEncoding.get())
```

Unser Menü machen wir so:

```
  def _generateMenu(self):
    menubar = Tkinter.Menu(self)

    filemenu = Tkinter.Menu(menubar)
    filemenu.add_command(label="Open...",
      command=self.openFile)
    filemenu.add_command(label="Save As...",
      command=self.saveFile)
    filemenu.add_separator()
    filemenu.add_command(label="Quit",
      command=self.quit)
    menubar.add_cascade(label="File",
      menu=filemenu)

    specialmenu = Tkinter.Menu(menubar)
    specialmenu.add_command(label="Permute",
```

```
      command=self.permute)
    menubar.add_cascade(label="Special",
      menu=specialmenu)

    self.config(menu=menubar)
```

Anmerkungen:

- `Menu` ist zunächst ein ganz normales Widget – wir könnten, so wir wollten, das auch Packen. Wir haben aber hier andere Pläne.
- `menubar` wird später die Menüzeile werden, in die wir hier einfach zwei Untermenüs einhängen, die wiederum `Menu`-Widgets sind.
- Um Menüpunkte in `Menus` unterzubringen, kann die `add_command`-Methode verwendet werden, die ein Label und einen Callback setzt. Die Methode `add_separator` fügt einen Trennstrich ein, während die Methode `add_cascade` ein Untermenü einhängt. Letzteres wird hier benutzt, um die Pull-down-Menüs in die Menüzeile einzuhängen, es ist aber auch erlaubt, auf diese Weise Untermenüs in die Pull-Downs einzufügen. In neueren Style Guides wird aber von solchen Konstrukten in der Regel abgeraten.
- Zum Konstruieren der Menüs braucht man also schon irgendwelche Callbacks. Ich empfehle, die Funktionen, die das Programm später haben soll, bereits bei der Definition des Menüs mitzudefinieren und in ihnen Exceptions auszulösen, die sagen, die entsprechend Funktion sei noch nicht implementiert. Auf diese Weise kann man das Programm allmählich mit Funktionalität ausstatten, hat aber trotzdem schon von Anfang an etwas, das funktioniert (sowas hieß eine Weile lang „rapid prototyping“).
- Schließlich haben Toplevel-Windows eine Option `menu`. Darüber kann man ihnen (im Gegensatz zu allen anderen Widgets) eine Menüleiste geben, was in der letzten Zeile der Funktion geschieht.
- Zur Benennung der Menüpunkte nur so viel, dass gängige Styleguides vorschreiben, dass ein Eintrag, dessen Aufruf auf einen Dialog führt, mit drei Punkten enden soll, solche, die unmittelbar aktiv werden, ohne drei Punkte.

68. Ein fast nützliches Programm III

Wir müssen jetzt noch `Editor`-Methoden schreiben, die die wirkliche Arbeit tun:

```
  def openFile(self, fname=None):
    if not fname:
      fname = tkFileDialog.askopenfilename(
        parent=self, title="Open File...")
    if not fname: return
    try:
      tx = open(fname).read()
    except IOError:
      tkMessageBox.showerror(title="Error",
        message= "Couldn't open %s"%fname,
        parent=self)
    else:
      self.text.setText(tx)

  def saveFile(self):
    fname = tkFileDialog.asksaveasfilename(
      parent=self, title="Save As...")
    if not fname: return
    try:
      f = open(fname, "w")
```

```
      f.write(self.text.getText())
      f.close()
    except IOError:
      tkMessageBox.showerror(title="Error",
        message="Write on %s failed"%fname,
        parent=self)
```

Anmerkungen:

- Tkinter kommt mit einigen nützlichen Modulen, die Dialoge vordefinieren, die man häufig braucht. Einer davon ist die Dateiauswahlbox, die im Modul tkFileDialog liegt. Eine einfache Möglichkeit, mit der Dateiauswahlbox zu kommunizieren, sind die Funktionen `askopenfilename`, die den Namen einer lesbaren Datei zurückgibt, die die Benutzerin ausgewählt hat, und `asksaveasfilename`, die etwa noch überprüft, ob der gewählte Name noch „frei“ ist (es also noch keine Datei gleichen Namens gibt) und beim Benutzer rückfragt, wenn das nicht so ist. Beide Funktionen lassen sich über unzählige Optionen steuern und geben ggf. den gewünschten Dateinamen zurück. Das `parent=self` sollte dafür sorgen, dass die Dateiauswahlbox über dem Anwendungsfenster erscheint. Ob das wirklich so ist, hängt vom verwendeten Window Manager ab.
- Ein weiteres nützliches Modul ist tkMessageBox. Wir verwenden daraus hier `showerror`, um eine Fehlermeldung anzuzeigen.

Schließlich definieren wir uns noch eine Spaßmethode zum Verwürfeln von Texten:

```
  def permute(self):
    import random
    tx = self.text.getText()
    parts = tx.split(" ")
    random.shuffle(parts)
    self.text.setText(" ".join(parts))
```

und können dann noch etwas Code hinzufügen, um das ganze zu einem Programm zu machen:

```
  import sys
  if len(sys.argv)>1:
    e = Editor(sys.argv[1])
  else:
    e = Editor()
  e.mainloop()
```

Auch wenn das Programm in der Tat schon ziemlich viel kann, bliebe natürlich noch etliches an der Benutzbarkeit zu feilen. So könnten wir uns z.B. den Namen der Datei, die wir gerade editieren, merken und einen Save-Menüpunkt einführen. Wenn ein Zeichen geschrieben werden soll, das im gewählten Encoding nicht darstellbar ist, wirft das Programm im Augenblick noch eine Exception. Es wäre viel besser, wenn die abgefangen würde und zu einer ordentlichen Fehlermeldung auf der GUI führen würde.

Es wäre auch gut, das Menü tastaturbedienbar zu machen („Accelerator Keys“). Gute GUI-Programme merken sich auch z.B. ihre Geometrie oder erlauben, ein bevorzugtes Encoding voreinzustellen. Und wenn man schon Konfigurationsdateien hat, warum dann nicht auch die unterstützten Encodings wählbar machen? Schließlich gäbs noch Kleinigkeiten, etwa einen vernünftigen Namen in der Titelzeile.

Wer also in den Ferien noch nichts vorhat...

(cf. Fig. 17)

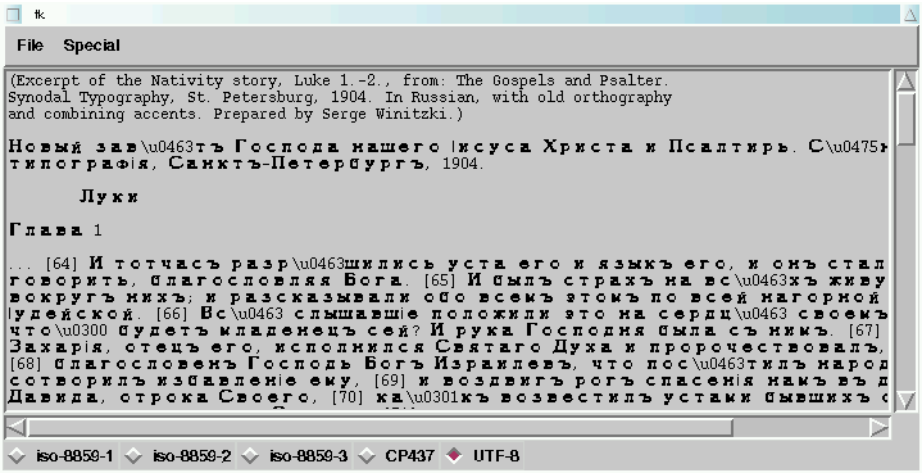

tk
File Special
(Excerpt of the Nativity story, Luke 1.-2., from: The Gospels and Psalter.
Synodal Typography, St. Petersburg, 1904. In Russian, with old orthography
and combining accents. Prepared by Serge Winitzki.)
Новый зав\u0463тъ Господа нашего Іисуса Христа и Псалтирь. С\u0475н
типографія, Санктъ-Петербургъ, 1904.
Луки
Глава 1
... [64] И тотчасъ разр\u0463шились уста его и языкъ его, и онъ стал
говорить, благословляя Бога. [65] И былъ страхъ на вс\u0463хъ живу
вокругъ нихъ; и разсказывали обо всемъ этомъ по всей нагорной
Іудейской. [66] Вс\u0463 слышавшіе положили это на сердц\u0463 своемъ
что\u0300 будетъ младенецъ сей? И рука Господня была съ нимъ. [67]
Захарія, отецъ его, исполнился Святаго Духа и пророчествовалъ,
[68] Благословенъ Господь Богъ Израилевъ, что пос\u0463тилъ народ
сотворилъ избавленіе ему, [69] и воздвигъ рогъ спасенія намъ въ д
Давида, отрока Своего, [70] ка\u0301къ возвестилъ устами бывшихъ с
iso-8859-1 iso-8859-2 iso-8859-3 CP437 UTF-8

Fig. 17