Universität Osnabrück
Informatik
Informatik
SS 2000

Alternative Programmiersprachen
Leitung: B. Kühl und E. Ludwig

Einführung in Python

Britta Koch
Am Salzmarkt 3, WG 11
49074 Osnabrück

M.A. Computerlinguistik und Künstliche Intelligenz
CL/KI(6) / Philosophie(6) / Informatik(6)

1 Einführung

1.1 Geschichte von Python

Python wurde 1989 von Guido von Rossum in den Niederlanden erfunden. Es ist nach der Comedytruppe ,,Monty Python" benannt, weshalb in Beispielen nicht die traditionellen Variablennamen ,,foo" und ,,bar" verwendet werden, sondern Wörter aus den Sketchen wie ,,spam" oder ,,eggs". Dieser Tradition möchte ich mich anschließen.

Python ist eine mächtige, objektorientierte Skriptsprache mit Garbage Collector, aber dennoch einfach zu benutzen. Es gibt viele Libraries sowie eine in den Newsgruppen `comp.lang.python.*` und im Internet http://www.python.org aktive Benutzergemeinde.

1.2 Syntax

Die Syntax von Python ist sehr einfach: manche Leute bezeichnen es als Pseudocode, der läuft. Um Blöcke kenntlich zu machen, gibt es keine geschweiften Klammern wie in C oder Java, allein die Einrücktiefe zählt: alles, was auf einer Ebene eingerückt ist, gehört zu einem Block. Nach Vokabeln, die Blöcke einleiten, wie `if, for` etc. steht ein Doppelpunkt. Das Kommentarzeichen ist `#` - alles danach bis zum Ende der Zeile ist ein Kommentar.

Es gibt die folgenden Operatoren: (die Tablle ist nach Vorrang geordnet, d. h. die Operatoren mit niedrigstem Vorrang stehen oben)

Operatoren	Beschreibung
`x or y` `lambda args: expr`	Logisches Oder Anonyme Funktion
`x and y`	Logisches Und
`not x`	Logische Verneinung

`<, <=, >, >=, ==, <>, !=`
`is, is not`
`in, not in`
	Vergleichsoperatoren
	Identitätstests
	Teil einer Sequenz

`x | y`	Bitweises Oder
`x ^ y`	Bitweises exklusives Oder
`x & y`	Bitweises Und
`x << y, x >> y`	Bitshifts
`x + y, x - y`	Addition / Verknüpfung, Substraktion
`x * y, x / y, x % y`	Multiplikation / Wiederholung, Division, Modulo / Formatierung
`-x, +x, ~x`	Unäre Negation, Identität, Bitweises Komplement
`x[i], x[i:j], x.y, x(...)`	Indexierung, Slicing, Qualifizierung, Funktionsaufruf
`(...), [...], {...}, `...`	Tupel, Liste, Dictionary, Stringkonvertierung

Natürlich sollte man, wie in anderen Sprachen auch, lieber Klammern setzen, um die Lesbarkeit zu erhöhen.

An Zahlentypen gibt es `int`, die durch C Integers, und `float`, die durch C Doubles implementiert sind. Deshalb ist auch der Wertebereich, wie in C, nicht genau definiert: man weiß nur die Mindestgröße. Oktale und Hexadezimal-Schreibweise funktionieren wie in C (mit `0` bzw. `0x` am Anfang). Außerdem gibt es extra große Integer, die nur durch den Speicher begrenzt sind, sowie komplexe Zahlen (Schreibweise: `3+4j`). Die Typumwandlung beim Rechnen mit Zahlen von verschiedenen Typen ist auch wie in C: das Ergebenis ist vom „komplexeren" Typ der beiden Operanden.

Wie in Java gibt es keine Zeiger oder Operationen für ihre Manipulation, allerdings kann man bei Vergleichen auf gleichen Inhalt der Speicherzelle bzw. gleiche Speicherzelle testen. Die Vokablen dafür sind allerdings genau umgekehrt wie in Java: dort gibt es `equals()` bzw. `==`, in Python heißen diese Operationen `==` bzw. `is`. Es gibt keinen extra Typ für wahr und falsch: alles, was nicht falsch ist, ist wahr, und falsch sind `0`, der leere String `''`, leere Tupel, Listen und Dictionaries (`(), [], {}`) sowie `None`, was `null` in Java entspricht.

Der Lesbarkeit wegen gibt es weder die Operatoren `+=, -=` etc. noch `++` oder `--`. Allerdings kann man `--a` schon sagen: das ist das Inverse zum Inversen von `a`, also `a`.

1.3 Datentypen und ihre Operationen

Der Vergleich von Datentypen erfolgt bei Zahlen über ihre relative Größe, bei Strings nach ASCII-Ordnung, Bei Listen und Tupeln werden von links nach rechts die einzelnen Komponenten miteinander verglichen. Dictionaries werden als sortierte Key-Value-Paare miteinander verglichen. Anstelle von `a > 0 and a <= 100` kann man auch `0 < a <= 100` schreiben.

Zuweisungen an Variablen gehen in Python auf verschiedene Weise:

```
spam = 'Spam'
spam, eggs = 'yum', 'YUM'
[spam, eggs, ham] = ['yum', 'YUM', 'meat']
spam = ham = 'lunch'
spam, eggs, pie = 'Spam', ['eggs', 4], 3.1415
```

Wenn auf der rechten Seite des Gleichheitszeichens mehrere Werte stehen, werden diese von Python erst in ein Tupel verpackt und dann an die Variablen auf der linken Seite zugewiesen - deshalb nennt man diesen Vorgang auch ,,tuple unpacking". Bei Variablennamen sind Groß- und Kleinschreibung wichtig: `spam` ist etwas anderes als `Spam`. Variablennamen dürfen mit einem Buchstaben oder einem Unterstrich beginnen, danach sind Buchstaben, Zahlen und Unterstriche erlaubt.

Für fast alle Datentypen gibt es einige gemeinsame Operationen, so die Länge, [1] die Position eines Elements, die Aneinanderfügung von 2 gleichen Datentypen, die Wiederholung des Elements, sowie die Mitgliedschaft in Sequenzen. Das Umwandeln in andere (ähnliche [2]) Datentypen funktioniert, indem man den Namen des Datentyps, den man erhalten möchte, auf das Element anwendet.

Mit `print` kann man Elemente ausgeben, und wie in Perl kann man auch die Klammern um die Argumente weglassen. Als Argument bekommt `print` entweder einen String mit `+` zusammengefügt, oder mehrere Argumente durch Kommas getrennt - dann werden sie bei der Ausgabe mit Leerzeichen getrennt.

```
>>> len('spam') #Länge
4
>>> 'spam'[3] #Position
'm'
>>> 'spam' + 'eggs' #Konkatenation
```

[1] Anders als andere Operationen auf Sequenzen wie `sort()`, `reverse()` etc. sind `len()` und `del()` keine Objektmethoden, sondern anscheinend Funktionen, die auf Datentypen angewendet werden.

[2] z.B. Umwandlung von Zahlen in Strings, Strings in Zahlen, Sequenzen(Listen, Tupel, Strings) ineinander

```
'spameggs'
>>> 4 * 'Ni! ' #Wiederholung
'Ni! Ni! Ni! Ni! '
>>> 'a' in 'spam' #Mitgliedschaft
1
>>> 'o' in 'spam' #Mitgliedschaft
0
>>> tuple([1, 2, 3]) #Umwandlungen
(1, 2, 3)
>>> tuple('hallo')
('h', 'a', 'l', 'l', 'o')
>>> list('hallo')
['h', 'a', 'l', 'l', 'o']
>>> str(1)
'1'
>>> int('1')
1
```

1.3.1 Indexing und Slicing

Strings (und andere Sequenzen) gehen von Position 0 bis zur Position Länge des Strings - 1. Um an das Element an der i-ten Stelle zu gelangen, verwendet man die Index-Schreibweise: `a[i]`. Man kann allerdings auch Offsets, negative Längenangaben, als Index angeben: `a[-i]` ist das gleiche wie `a[len(a) - i]`. Mithilfe von „Slicing" (in Scheibchen schneiden) kann man sich auch Abschnitte holen oder belegen. Dafür wird die Doppelpunktnotation verwendet: `a[i:j]` liefert `a` ab Position `i` bis *vor* Position `j`. Schreibt man nichts vor bzw. hinter den Doppelpunkt, geht die Sequenz vom Anfang bzw. bis zum Ende. Auch beim Slicing kann man Offsets benutzen.

```
>>> s = 'lumberjack'
>>> s[5]
'r'
>>> s[-1]
'k'
>>> s[2:4]
'mb'
>>> s[:6]
'lumber'
>>> s[7:]
'ack'
>>> s[6:-1]
'jac'
```

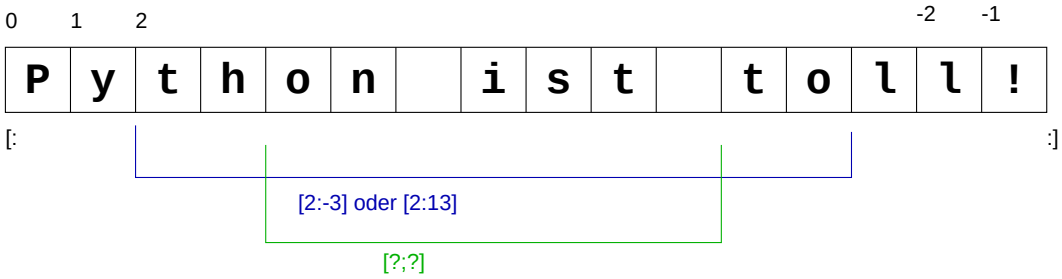

Abbildung 1: Slicing

1.3.2 Unveränderliche Datentypen: Tupel und Strings

Zu den unveränderliche Datentypen gehören Tupel und Strings. Man kann sie als Schlüssel in Dictionaries benutzen, aber nicht in sich verändern, wie sortieren, umdrehen, oder einzelnen Positionen bzw. Slices etwas zuordnen.

Tupel sind unveränderliche Listen. `()` bezeichnet ein leeres Tupel. Die Schreibweise `tupel = (1,)` für ein einstelliges Tupel ist zwar etwas seltsam mit dem Komma am Ende, aber sonst würden die Klammern als Vorrangoperator aufgefaßt. Größere Tupel erhält man, indem man die Elemente einfach in Klammern einschließt. Man kann verschiedene Typen in ein Tupel packen, sogar Tupel. Auf Tupeln funktionieren die schon besprochenen Operationen wie Indexing, Slicing, Wiederholung, etc.

```
>>> t1 = (4, )
>>> t2 = ('a', t1, (3, 'hallo'))
>>> t1
(4,)
>>> t2
('a', (4,), (3, 'hallo'))
>>> t2[1][0]
4
>>> t2[:-1]
('a', (4,))
>>> t2[1:][0]
(4,)
>>> t2[1:][1]
(3, 'hallo')
```

Strings sind der zweite unveränderliche Datentyp. Bis auf die Notation (Strings umschließt man mit `''` oder `""`) verhalten sie sich wie Tupel. Die zwei Arten von Anführungszeichen sind als Bequemlichkeit da, damit man in `"Guido's"` den einfachen Quote `'` nicht mit dem Backslash zu maskieren

braucht. Benutzt man Dreifach-Quotes, können die Strings über mehrere Zeilen hinweg gehen, Newlines werden erhalten.

Man kann in Strings, ähnlich wie in C, mit dem Prozentzeichen eine Formatierung erzielen. Im String benutzt man die aus `printf` bekannten Platzhalter. Vor den Werten, die in einem Tupel stehen müssen, wird das Prozentzeichen dann als Operator benutzt, der anzeigt, daß die Werte folgen. Im Modul `string` gibt es auch noch andere Funktionen zur Stringmodifikation, wie `upper(), find()` etc.

```
>>> long_string = """hallo
... dies ist die zweite Zeile
... 3"""
>>> '%e %f %g' %(1.1, 2.2, 3.3)
'1.100000e+00 2.200000 3.3'
```

1.3.3 Veränderbare Datentypen: Listen und Dictionaries

Listen in Python können von beliebiger Länge sein und, wie Tupel, beliebige Objekte aufnehmen. Man kann auch zyklische Listen erstellen, wobei die Ausgabe diese verkürzt anzeigt. Wie alle veränderlichen Datentypen werden Listen als Referenzen übergeben („call by reference"). `[]` ist die leere Liste, auch sonst werden Listen in eckige Klammern eingeschlossen. Bei Listen gibt es zusätzlich zu den erwähnten Operationen noch `append()` zum Anfügen eines Elements, `sort()` und `reverse()`, die die Liste destruktiv sortieren bzw. umdrehen, sowie `index()`, das einem die Position des übergebenen Elementes sagt, und `del`, dem man die in der Liste zu löschende Position in der Index-Schreibweise übergibt.

```
>>> a = [1, 4, 'spam', 3.14, 'Ni!']
>>> a.sort()
>>> a
[1, 3.14, 4, 'Ni!', 'spam']
>>> a.reverse()
>>> a
['spam', 'Ni!', 4, 3.14, 1]
>>> a[2] = 'eggs'
>>> a
['spam', 'Ni!', 'eggs', 3.14, 1]
>>> a.index(4)
Traceback (innermost last):
File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
>>> a.index('eggs')
2
```

```
>>> del(a[1])
>>> a
['spam', 'eggs', 3.14, 1]
```

In Python bezeichnet man als Dictionaries das, was man aus Java als Hashtable oder aus awk als assoziatives Array kennt. Die abgelegten Objekte können verändert werden. Ein leerer Dictionary ist `{}`, die Schreibweise für eine bestimmte Zuweisung ist `dict = {'schluessel1': 'wert1', 'schluesseln': 'wertn'}`. Will man auf Werte zugreifen, so macht man dies mit der Index-Notation, gibt aber den Schlüssel als Index an, so z. B. `dict['schluessel1']`. Mit der Index-Schreibweise kann man Werte verändern oder auch neue ablegen. Dictionaries kann man nicht slicen, man kann aber die Funktionen `len()` und `del` auf sie anwenden. Auch die Überprüfung auf Mitgliedschaft mit `in` funktioniert nicht, es gibt aber `has_key()`, das abfragt, ob der Dictionary das Argument als Schlüssel enthält, sowie die Funktionen `keys()` und `values()`, die Sequenzen der Schlüssel bzw. Werte zurückliefern.

```
>>> d = {'spam' : 2, 'eggs' : 3}
>>> d
{'spam': 2, 'eggs': 3}
>>> d['sub'] = {'john': 'cleese', 'rowan': 'atkinson'}
>>> d
{'spam': 2, 'eggs': 3, 'sub': {'john': 'cleese', 'rowan': 'atkinson'}}
>>> d['sub']['john'] = 'smith'
>>> d
{'spam': 2, 'eggs': 3, 'sub': {'john': 'smith', 'rowan': 'atkinson'}}
>>> d.has_key('john')
0
>>> d['sub'].has_key('john')
1
>>> d.keys()
['spam', 'eggs', 'sub']
>>> len(d)
3
```

1.3.4 Noch ein eingebauter Datentyp: Files

Der letzte eingebaute Datentyp unterscheidet sich von den anderen dadurch, daß er keine Sequenz ist. Aber der Dateizugriff ist auch so ganz nützlich. Mit der Funktion `open()`, die Parameter wie in C bekommt, also den Dateinamen und die Zugriffsart ('r' für lesend, 'w' für schreiben, 'a' für anhängend), erhält man ein Filehandle. Dieses kann man jetzt je nach Zugriffsart lesen: mit `read()`, was die ganze Datei in einem String liefert, `readlines()`, was eine Liste mit den Zeilen der Datei liefert, `readline()`, was jeweils die

nächste Zeile liefert, sowie `read(N)`, was die nächsten `N` Bytes zurückgibt. Für schreibende Filehandles gibt es die Funktionen `write()`, die den übergebenen String in die Datei schreibt, sowie `writelines()`, die eine Liste von Strings erwartet und sie alle in die Datei schreibt. Allgemein kann man Filehandles mit `close()` schließen, mit `flush()` die Ausgabe erzwingen und mit `seek()` in der Datei herumwandern.

```
>>> a = ["I'm a lumberjack and I'm okay\n",
...'I drink all night and I work all day\n',
...'(3, 4)\n', '3.1415\n']
>>> output = open('/tmp/spam', 'w')
>>> output.writelines(a)
>>> output.write('hallo\n')
>>> output.close()
>>> input = open('/tmp/spam', 'r')
>>> line = input.readline()
>>> line
"I'm a lumberjack and I'm okay\012"
>>> line = input.readline()
>>> line
'I drink all night and I work all day\012'
>>> five_bytes = input.read(5)
>>> five_bytes
'(3, 4'
>>> input.readlines()
[')\012', '3.1415\012', 'hallo\012']
>>> input.close()
```

2 Kontrollstrukturen

2.1 if, for, while, def

Wie gesagt ist die Syntax in Python sehr einfach. Bei Kontrollstrukturen wie `if, for` und `while` wird die Bedingung nicht in Klammern gesetzt, sondern einfach danach hingeschrieben, und der folgende Block wird nach einem Doppelpunkt eingerückt begonnen. In den Bedingungen sind, anders als in C, keine Zuweisungen möglich, der Lesbarkeit wegen.

Für Wenn-Dann-Abfragen braucht man in Python die Vokabeln `if, elif` und `else`. Die Syntax ist wie folgt:

```
if <Bedingung>:
  #tu was...
elif <andere Bedingung>:
```

```
    #tu was anderes
  else:
    #ansonsten mach was
```

Da Python interpretiert ist, gibt es keinen Ausdruck für `switch-case`. Aber stattdessen kann man manchmal ein Dictionary einsetzen, wo man Funktionen oder Lambda-Ausdrücke unter Zahlen oder Strings ablegt und mit `apply` anwendet. Beispiel:

```
a = 3
b = 5
c = 7
if a < b < c:
    print a, b, c, "geordnet!"

if (a * b) < c:
    print "Produkt von a und b > c!"
elif (a * b) == c:
    print "c = a * b!"
else:
    print "c < a * b!"


weekdays = {'Monday' : 'Schoene Woche!',
            'Tuesday' : 'Viel Spass bei Robotik!',
            'Wednesday' : 'Lern mal neue Sprachen!',
            'Thursday' : 'Schon einen Roboter gebaut?',
            'Friday' : 'Schoenes Wochenende!'}

day = 'Tuesday'
print weekdays[day]
```

Ausgabe:

```
3 5 7 geordnet!
c < a * b!
Viel Spass bei Robotik!
```

Bei Schleifen gibt es die aus anderen Sprachen bekannten Ausdrücke `break` und `continue`, um aus einer Schleife auszusteigen bzw. mit dem

nächsten Wert weiterzumachen. Auch die leere Anweisung `pass`, die dem Semikolon in C entspricht, kann man in diesem Zusammenhang erwähnen[3]. Nach allen Schleifen kann man mit `else` einen Block beginnen, der am Ende der Schleife ausgeführt wird, falls diese nicht mit `break` verlassen wurde.

Die Syntax für `while`-Schleifen ist folgendermaßen:

```
while <Bedingung>:
  #Schleife
  #break oder continue?
else:
  #falls kein break, mach noch was
```

Beispiel:

```
file = open('while.py')
while 1:
    line = file.readline()
    if not line:
        break
    print line[:-1]

print '-----------------------------'

y = 101
x = y / 2
while x > 1:
    if y % x == 0:
        print y, 'has factor', x
        break
    x = x - 1
else:
    print y, 'is prime'
```

Ausgabe:

```
file = open('while.py')
while 1:
  line = file.readline()
  if not line:
    break
  print line[:-1]
```

[3]Sei es nur, um eine Endlosschleife zu schreiben: `while 1: pass`

```
print '------------------------------'

y = 101
x = y / 2
while x > 1:
  if y % x == 0:
    print y, 'has factor', x
    break
  x = x - 1
else:
  print y, 'is prime'
------------------------------
101 is prime
```

`for`-Schleifen iterieren über einer Sequenz, also z. B. einem Tupel, einer Liste oder einem String. Wenn man über einer Liste von Zahlen iterieren möchte, ist die Funktion `range()` interessant, die bei einem Parameter eine Liste von Null bis zum Parameter, bei 2 Parametern vom ersten bis zum zweiten zurückgibt. Übergibt man 3 Parameter, wird vom ersten zum zweiten Parameter gezählt, das Inkrement wird durch den dritten Parameter bestimmt. `range(10)` ist also das gleiche wie `range(0, 10)` oder `range(0, 10, 1)`. `xrange()` funktioniert genauso, liefert aber ein Tupel zurück.

`for`-Schleifen haben folgende Syntax:

```
for <variable> in <sequenz>:
  #mach was mit <variable> oder auch ohne
else:
  #falls nix break
```

Beispiel:

```
for i in range(10):
    print i, '...'
print 'and liftoff!'

print '2 loops:'
items = ['spam', 3.14, 911, (123, 456), 'Ni!']
test = [1.44, 'Ni!', (123, 456)]
for key in test:
    for item in items:
```

```
        if item == key:
            print key, 'was found'
            break
    else:
        print key, 'not found'

print '1 loop'
for key in test:
    if key in items:
        print key, 'was found'
    else:
        print key, 'not found'
```

Ausgabe:

```
0 ...
1 ...
2 ...
3 ...
4 ...
5 ...
6 ...
7 ...
8 ...
9 ...
and liftoff!
2 loops:
1.44 not found
Ni! was found
(123, 456) was found
1 loop
1.44 not found
Ni! was found
(123, 456) was found
```

Funktionen definiert man mit dem Schlüsselwort `def`. Wenn man einen Wert zurückgeben möchte, tut man das mit `return`, man muß aber nicht - dann ist der Rückgabewert `None`. Man kann auch mehrere in ein Tupel verpackte Werte zurückgeben - dann wird, wenn man die Rückgabewerte an Variablen zuweist, Tuple unpacking betrieben.

Die Syntax für Funktionen:

```
def <funktionsname>(<parameter>):
  #tu was
  return <was>
```

In Funktionen lokale Variablen sind nach außen nicht sichtbar, außer man sagt dies explizit mit dem Schlüsselwort `global`. Wie aus anderen Sprachen bekannt, verdecken lokale Variablen in Blöcken Variablen außerhalb. Wenn aber der Interpreter eine Variable nicht lokal im aktuellen Block oder auf globaler Ebene findet, schaut er bei builtin-Funktionen nach.

Parameter können bei der Funktionsdefinition auch auf Default-Werte gesetzt werden - wenn dann weniger Parameter angegeben werden, sind die anderen mit diesen Defaults belegt. Außerdem kann man beim Aufruf einer Funktion, wenn man die Reihenfolge der Parameter nicht weiß, aber ihre Namen, diese explizit mit angeben - dann ist die Reihenfolge egal, solange die Schlüsselwort-Argumente am Schluß stehen. Wenn man an übergebene Parameter innerhalb der Funktion zuweist, ist das nach außen egal (call by value), aber wenn man veränderliche Datentypen innerhalb einer Funktion verändert, wirkt sich das auf sie auch außerhalb der Funktion aus.

Beispiel:

```
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res

def return_tuple(arg1, arg2, arg3):
    return (arg1, arg2, arg3)

def times(x, y=1):
    return x * y

print 'intersections:'
print intersect('spam', 'scam')
print intersect([1, 2, 3, 4,], (4, 2, 5))
print 'tuple stuff:'
return_tuple(3, 17, 'a')
spam = return_tuple(1, 2, 3)
print spam
eggs, ham, spam = return_tuple(1, 2, 3)
print eggs, spam, ham
print 'keywords:'
```

```
print times(4, 'Ni!')
print times('Ni!')
print times(x=4, y='Ni!')
print times(y=4, x='Ni!')
print times(4, y='Ni!')
```

Ausgabe:

```
intersections:
['s', 'a', 'm']
[2, 4]
tuple stuff:
(1, 2, 3)
1 3 2
keywords:
Ni!Ni!Ni!Ni!
Ni!
Ni!Ni!Ni!Ni!
Ni!Ni!Ni!Ni!
Ni!Ni!Ni!Ni!
```

2.2 Exceptions

Mit Exceptions hat Python ein Fehlerkonzept, wie man es aus Java kennt. Das Werfen bzw. Auffangen von fehlerspezifischen Exceptions verkürzt den Code enorm. Die Vokabeln hierfür sind `try` und `except` und `else` bzw. `finally` um Blöcke, die Exceptions werfen könnten, zum Auffangen von Exceptions bzw. zur Ausführung egal was passiert (anders als in Java), sowie `raise` zum Verursachen von Exceptions. Dabei schließen sich `except` und `finally` aus. Mit `except` kann man Variablen, Klassen von Exceptions oder in einem Tupel zusammengefaßte Exceptions auffangen. Nach der Exception kann man auch eine Variable angeben, um mehr Details zu erfahren (diese gibt man nach der Exception bei `raise` an) - per Default ist dies `None`.

Die Syntax von Exceptions sieht folgendermaßen aus:

```
try:
  #mach was
except <name>:
  #fang auf
```

```
except (<name1>, <name2>):
  #mehrere auf einmal abarbeiten
except <name>, <data>:
  #extra infos
except:
  #default Fehler
else:
  #nix exception

try:
  #mach was
finally:
  #mach dies, egal was passiert ist
```

Beispiel:

```
def excep1(a=[]):
  try:
    for i in range(3):
      5 / a[i]
  except (ZeroDivisionError, IndexError), data:
    print data
  else:
    print 'Liste ok!'

def excep2(error='error'):
  try:
    raise error, 'oh-oh!'
  finally:
    print 'vorbei!'

excep1([1, 2, 0])
excep1()
excep1([3, 4, 6, 'a'])
mess = 'Fehler'
try:
  excep2(mess)
except mess, data:
  print mess, data
```

Ausgabe:

```
integer division or modulo
list index out of range
Liste ok!
vorbei!
Fehler oh-oh!
```

2.3 Lambda-Expressions

Mit `lambda` kann man Funktionen generieren, aber weil es ein Ausdruck ist, kann es auch in einer Liste etc. stehen. Allerdings kann man nur eine Zeile ohne Kontrollstrukturen in einen Lambda-Ausdruck stecken, so daß die Komplexität begrenzt ist. Da man auch kein `return` benutzen kann, sondern nur den Ausdruck aufzuschreiben braucht, der zurückgegeben werden soll, kann man Lambda-Expressions eher mit Makros als mit Funktionen vergleichen.

Die Syntax von `lambda`:

```
lambda <args..., auch mit defaults>: <Ausdruck>
```

Lambda-Ausdrücke werden oft in Verbindung mit `apply` und `map` benutzt, die eingebaute Funktionen sind. `apply` bekommt 2 Argumente: ein Funktionsobjekt (auch einen Lambda-Ausdruck) sowie ein Argumenttupel. Es ruft die angegebene Funktion mit dem Tupel als Parametern auf - der Vorteil hierbei ist, daß man sowohl Funktion als auch Argumente aus Variablen lesen kann. Wenn man allerdings Keywords nutzen möchte oder zuviele Argumente angibt, klappt dies nicht.

`map` bekommt auch als erstes Argument ein Funktionsobjekt, der Rest sind Sequenzen. Dann wendet es die Funktion nacheinander auf die Elemente der Sequenzen an. Übergibt man mehr als eine Sequenz, wird nacheinander je ein Element pro Sequenz zusammen an die Funktion übergeben. `map` liefert die Ergebnisse als Liste zurück. Es gibt noch weitere Funktionen, die Funktionen nach bestimmten Gesichtspunkten auf Listen anwenden, wie z. B. `reduce` oder `filter`. In diesem Zusammenhang kann man auch noch `eval` erwähnen, das einen übergebenen Python-Ausdruck auswertet.

Beispiel:

```
>>> times = lambda x, y: x * y
>>> times
<function <lambda> at 807f228>
>>> times(4, 2)
8
>>> times(4, 'Ni!')
```

```
'Ni!Ni!Ni!Ni!'
>>> apply(times, (times(4, 2), 'Ni!'))
'Ni!Ni!Ni!Ni!Ni!Ni!Ni!Ni!'
>>> map(times, ['a', 4, 3.14], [2, 'Ni!', 17])
['aa', 'Ni!Ni!Ni!Ni!', 53.38]
```

2.4 Objektorientierung

Die Objektorientierung in Python ist ganz einfach.

Syntax:

```
class <name>(<Superklassen>):
  def __init__(self, <args>):
    self.<slot> = <wert>
    #Konstruktor
  def <funktion>(self, <args>):
    #andere Funktionen
```

Objektvariablen werden wie gezeigt mit `self.<varname> = <wert>` zugewiesen, danach kann man sie mit `self.<varname>` ansprechen. Instanzen erzeugt man sich durch `<instanz> = <Klasse>(<Konstruktor-Argumente>)`, auf deren Slots kann man dann mit `<instanz>.<varname>` zugreifen. Objektmethoden ruft man mit `<instanz>.<methode>(<args>)` auf. Intern wird das als `<methode>(<instanz>, <args>)` gemappt, weshalb auch die Definition als ersten Parameter `self` hat, das der Verweis auf das aktuelle Objekt ist. Man muß diesen Parameter nicht `self` nennen, dies ist aber so üblich. Schlüsselwörter wie `private` oder `protected` gibt es nicht - will man Zugriff auf eine Methode bzw. einen Slot vermeiden, setzt man Unterstriche davor und dahinter.

In Python gibt es auch Mehrfachvererbung - die Reihenfolge der Vererbung ist hier top-down von links nach rechts. Will man die Methoden einer Superklasse aufrufen, benutzt man die Syntax `<Superklasse>.<methode>(self, <params)`. Das Schlüsselwort `super` existiert in Python auch nicht - mit dieser Methode weiß man aber genau, welche Superklasse denn nun die angesprochen wird.

Außer besonderen Methoden wie `__init__()`, den Konstruktor, gibt es noch andere: `__del__()`, den Destruktor, und `__repr__()`, das die Repräsentation als String (bei Verwendung von ‘‘) zurückgibt. Man kann z. B. mit `__add__()` oder `__mul()__` die Additions- bzw. Multiplikationsoperatoren überladen oder auch Index- bzw. Sliceoperationen.

Anders als in Java kann man Methoden nicht überladen, z. B. mit verschiedenen Typen - zum einen verhindert das die schwache Typisierung der

Skriptsprache, zum anderen muß alles in der Symboltabelle einzigartig sein[4].

Beispiel:

```
class Basket:

    def __init__(self,contents=None):
        self.contents = contents or []

    def add(self,element):
        self.contents.append(element)

    def __str__(self):
        result = ""
        for element in self.contents:
            result = result + " " + ‘element‘
        return "Contains:"+result

class SpamBasket(Basket):
    def __init__(self, contents):
        Basket.__init__(self, contents)
        self.add(’spam’)

    def add(self, element):
        Basket.add(self, element)
        Basket.add(self, ’spam’)


b = Basket([’apple’,’orange’])
b.add("lemon")
print b
sb = SpamBasket(b.contents)
sb.add(’eggs’)
print sb
```

Ausgabe:

```
        Contains: ’apple’ ’orange’ ’lemon’
        Contains: ’apple’ ’orange’ ’lemon’ ’spam’ ’spam’ ’eggs’ ’spam’
```

[4]so überschreibt eine Variable a auch die Funktion a und umgekehrt

3 Module und mehr

In Modulen kann man bestimmte Funktionalitäten verkapseln, die man öfter braucht. Die Syntax dazu lautet `import <modul>` oder `from <modul> import <funktionen>`, wobei bei der `from`-Schreibweise ein `*` alle Funktionen importiert. Module sind abgeschlossene Namespaces. Sie können auch in C oder C++ geschrieben sein.

Interessante Module sind `Tkinter`, mit dem man GUIs mit TK-Widgets erstellen kann, `re`, das reguläre Ausdrücke nach Perl-Syntax unterstützt, `sys`, das u. a. die wichtige Funktion `exit` bereitstellt oder `os`, in dem betriebssystemabhängige Funktionen wie `fork()` oder `ls()` implementiert sind. Für Persistenz von Objekten gibt es `pickle` oder `shelf`, `whrandom` stellt Zufallsfunktionen bereit und mit `thread` kann man Threads als Prozesse erzeugen. Im Module Index http://www.python.org/doc/current/modindex.html sind viele andere mit Links auf die Dokumentation verzeichnet.

4 Schluß

4.1 JPython

JPython ist eine Implementation von Python in 100% Pure Java. Damit kann man auf Java-Klassen und Packages zugreifen wie auf Python-Module und den Code auch in Java-Code oder Bytecode übersetzen. Einige Python-Module wie `pickle, re` oder `thread` sind auch schon nach Java portiert. Die Probleme bei der Portierung könnten auch CPython einige Anstöße bezüglich Geschwindigkeit etc. geben. Mehr Infos und Downloadmöglichkeiten findet man bei http://www.jpython.org.

Beispielcode:

```
from java.awt import Frame, Button, FlowLayout
from java.lang import System

class ButtonDemo(Frame):
    def __init__(self):
      Frame.__init__(self)
      f = FlowLayout()
      self.setLayout(f)
      self.b1 = Button('Disable middle button', actionPerformed=self.disable)
      self.b2 = Button('Middle button', actionPerformed=self.exit)
      self.b3 = Button('Enable middle button', enabled=0, actionPerformed=self.enabl

      self.add(self.b1)
      self.add(self.b2)
```

```
      self.add(self.b3)
      self.pack()

    def enable(self, event):
      self.b1.enabled = self.b2.enabled = 1
      self.b3.enabled = 0
    def disable(self, event):
      self.b1.enabled = self.b2.enabled = 0
      self.b3.enabled = 1
    def exit(self, event):
      System.exit(0)

bd = ButtonDemo()
bd.setVisible(1)
```

4.2 zope

Zope ist die „Killerapplikation" für Python - ein freies Web-Portal und Application-Server in Python implementiert. Um es normal zu benutzen, braucht man kein Python zu können, aber für Erweiterungen und Sonderwünsche ist dies sicherlich von Vorteil. Die Objektorientierung erlaubt die Trennung von Daten, Logik und Präsentation. Zope hat auch einen eigenen Webserver und eine Datenbank, läuft aber auch auf anderen Webservern. Die URL für weitere Infos ist http://www.zope.org (mit Zope geschrieben!).

4.3 Wofür ist Python eigentlich gut?

Python ist durch seine Objektorientierung mächtiger als Tcl, insgesamt einfacher und lesbarer als Perl, und durch die schwache Typisierung schneller zu schreiben als Java. Es ist eine ideale Lernsprache, auch für Objektorientierung. Man kann es als „Kleber" oder Skriptsprache in C-, C++- oder Java-Anwendungen benutzen. Auch empfiehlt es sich, vor allem im Zusammenhang mit GUIs, als Rapid Prototyping-Sprache - zunächst implementiert man eine grobe Applikation in Python, später kann man geschwindigkeitsrelevante Teile in C oder C++ machen. Allgemein ist Python eine bessere Wahl als Perl, wenn mehrere Leute Code pflegen sollen - aber wenn man viel Textauswertung macht, sollte man doch vielleicht einen Perl-Guru anstellen.