Programmierkurs Python II

Michaela Regneri & Stefan Thater
FR 4.7 Allgemeine Linguistik (Computerlinguistik)
Universität des Saarlandes

Sommersemester 2010

Kursübersicht

- Datenstrukturen & Algorithmen
  - Bäume & Graphen
  - Graph-Algorithmen
  - Graph-Ähnlichkeit
- Endliche Automaten & Transduktoren
- Maschinelles Lernen
  - Naive Bayes Classifier
  - Hidden-Markov-Modelle
  - Vektormodelle

Kursübersicht

- Kontextfreie Grammatiken & Parsing
  - Elementare Algorithmen
  - Chart-Parsing
  - Probabilistische kontextfreie Grammatiken
- Vorschläge und Wünsche?

Prüfungsleistungen

- Klausur am Semesterende
  - Zulassung: >50% der Punkte in den Übungsaufgaben
- Programmierprojekt
- Endnote
  - Klausur 50%, Projekt 50%

Programmierprojekt

- Im Programmierprojekt soll eine (etwas) umfangreichere Aufgabenstellung bearbeitet werden.
  - Arbeitsaufwand: etwa 2 Wochen
  - Abgabe: etwa 6 Wochen nach Beginn der Semesterferien
- Themenvorschläge:
  - Implementierung eines einfachen statistischen Parsers
  - Implementierung eines Programms zum Suchen von Bäumen in Baumbanken („tgrep“)
  - Word-Sense Disambiguierung
  - Weitere Vorschläge?

Übersicht (heute)

- Kurze Wiederholung zu Python
- Bäume
  - Definition
  - Implementierung
  - Parsen von Baum-Ausdrücken
  - Suche in Bäumen (Tiefensuche)

Kurze Wiederholung

- Kontrollstrukturen
- Funktionen
- Rekursion
- Klassen
- Iteratoren
- Generatoren
- List-Comprehension

Kurze Wiederholung: Funktionen

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Kurze Wiederholung: Rekursion

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)

def fib(n, a = 0, b = 1):
    if n < 1:
        return a
    else:
        return fib(n - 1, b, a + b)
```


Kurze Wiederholung

```
def wc(filename):
    freq = dict()
    with open(filename) as f:
        for line in f:
            for word in line.split():
                try:
                    freq[word] += 1
                except KeyError:
                    freq[word] = 1
    for (word, frq) in freq.items():
        print('{0:s}\t{1:d}'.format(word, frq))
```

Kurze Wiederholung: Klassen

```
class MyClass(BaseClass):
    def __init__(self, ...):
        <self initialisieren>
    def myMethod(self, ...):
        ...
    @staticmethod
    def myStaticMethod(...): # kein „self“
        ...
    @classmethod
    def myClassMethod(cls, ...): # „cls“ statt „self“
        ...
```


Kurze Wiederholung: Iteratoren

```
class fibit:
    def __init__(self):
        self.a = 0
        self.b = 1
    def __iter__(self):
        return self
    def __next__(self):
        this = self.a
        self.a, self.b = self.b, self.a + self.b
        return this
```

Kurze Wiederholung: Generatoren

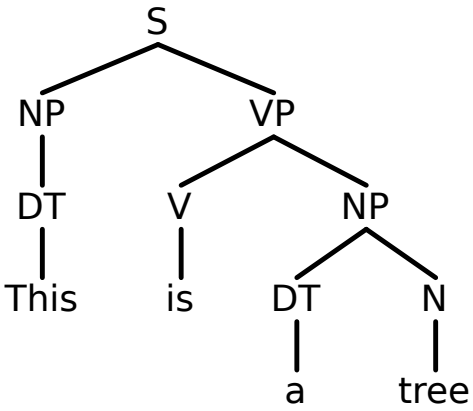
```
def fibit():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```


Kurze Wiederholung: Comprehensions

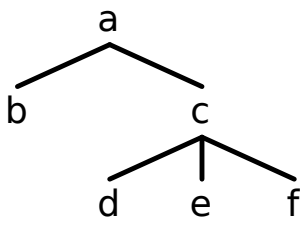
- `lst = [1, 2, 3, 4]`
- `[x * 2 for x in lst]`
  `⇒ [2, 4, 6, 8]`
- `[x for x in lst if x % 2 == 0]`
  `⇒ [2, 4]`
- `(x for x in lst if x % 2 == 0)`
  `⇒ <generator object <genexpr> at ...>`
- `sum(x for x in lst if x % 2 == 0)`
  `⇒ 6`

Bäume

Bäume

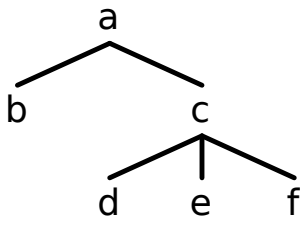


Bäume



- Bestandteile
  - Menge von Knoten (nodes, vertices)
  - Menge von Kanten (edges)
- Hier immer gerichtete Bäume
  - Kanten haben eine Quelle (source) und ein Ziel (target)
- Mit Ausnahme der Wurzel hat jeder Knoten genau eine eingehende Kante (⇒ keine Zyklen).
  - Die Wurzel hat keine eingehende Kante
- Knoten können etikettiert sein

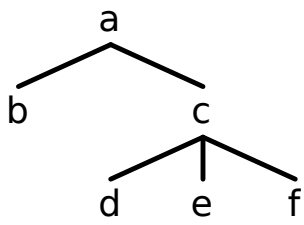
Bäume



- Quelle und Ziel nennen wir auch Mutter und Tochter
  - oder auch Vater und Sohn
- Knoten mit gleicher Mutter heißen entsprechend Geschwister
- Der einzige Knoten ohne Vorgänger ist die Wurzel (root)
- Knoten ohne Nachfolger heißen Blätter (leaves)

Bäume

- Bäume kann man leicht als rekursive Datenstruktur implementieren:
  - Baum = Etikett + Liste von Bäumen



- Zum Beispiel:
  - `('a',[('b',[]),('c',[('d',[]),('e',[]),('f',[])])])`
- Beachte:
  - Wir betrachten hier `a`, `b`, … als Etiketten, nicht als Knoten.
  - Wir setzen Knoten und den Teilbaum unterhalb des Knotens konzeptuell gleich.

Bäume (objektorientiert)

```
class Tree(object):
    def __init__(self, label, children):
        self.label = label
        self.children = children
    ...
```

Baumausdrücke Parsen (String ⇒ Baum)

- Eingabe: Zeichenkette, die einen Baum beschreibt
- Format:
  - Baum ::= Etikett | (Etikett Baum ... Baum)
  - Etikett ::= beliebige Zeichenkette ohne (,), Leerzeichen
- Beipiel:
  - (S (NP (DET Der) (N Student)) (VP (V arbeitet)))
- Ausgabe: Baum als rekursive Datenstruktur

Baumausdrücke Parsen (String ⇒ Baum)

```
def parse(strng):
    tokens = tokenize(strng)
    return tree(next(tokens), tokens)

def tokenize(strng):
    return <Iterator über die Tokens in strng>
```

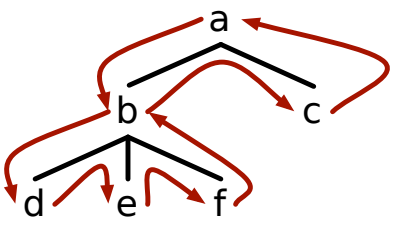
Baumausdrücke Parsen (String ⇒ Baum)

```
def tree(token, tokens):
    if token == '(':
        return Tree(next(tokens), list(trees(tokens)))
    else:
        return Tree(token, [])

def trees(tokens):
    while True:
        token = next(tokens)
        if token == ')':
            break
        yield tree(token, tokens)
```

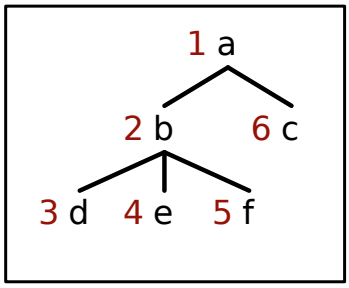

Tiefensuche

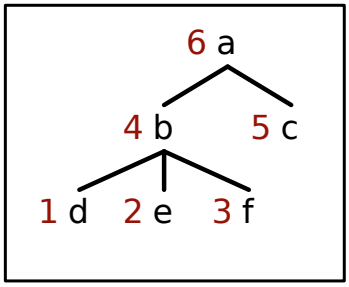
- Mit Traversierung bezeichnen wir das Untersuchen der Knoten eines Baumes in einer bestimmten Reihenfolge
- Tiefensuche (depth first search)
  - zuerst betrachtet wir die Kinder eines Knotens
  - danach seine Geschwister.



Post-Order vs. Pre-Order

- Es gibt verschiedene Möglichkeiten, die Knoten eines Baumes mit Tiefensuche zu traversieren
- Pre-Order:
  - zuerst betrachten wir den Knoten
  - dann die Kinder
- Post-Order:
  - zuerst betrachten wir die Kinder
  - dann den Knoten





Tiefensuche (Pre-Order, Iterator)

```
class TreeIterator:
    def __init__(self, tree):
        self.agenda = [tree]
    def __iter__(self):
        return self
    def __next__(self):
        if self.agenda == []:
            raise StopIteration
        current_tree = self.agenda.pop()
        for child in reversed(current_tree.children):
            self.agenda.append(child)
        return current_tree
```

Tiefensuche (Pre-Order, Generator)

```
def TreeIterator(tree):
    yield tree
    for child in tree.children:
        for desc in TreeIterator(child):
            yield desc
```