

# Programmierkurs Python II

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

Sommersemester 2011



## Prüfungsleistungen

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

# Kursübersicht

- Datenstrukturen & Algorithmen
  - Bäume & Graphen
  - Graph-Algorithmen
- Endliche Automaten & Transduktoren
- Maschinelles Lernen
  - Naive Bayes Classifier
  - Vektormodelle
- Kontextfreie Grammatiken & Parsing
  - Elementare Algorithmen
  - Chart-Parsing
  - Probabilistische kontextfreie Grammatiken

3

# Heute:

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

4

# Kurze Wiederholung

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

# Wiederholung: Funktionen

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

## Wiederholung: Rekursion

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

7

## Wiederholung: Bsp. Wörter zählen

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

8

## 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“  
        ...
```

9

## 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
```

10

## Kurze Wiederholung: Generatoren

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

11

## 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`

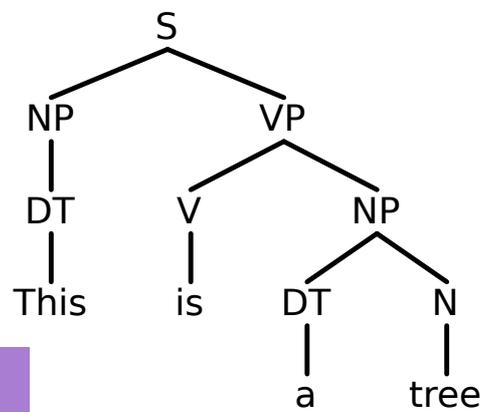
12

# Bäume

Text

13

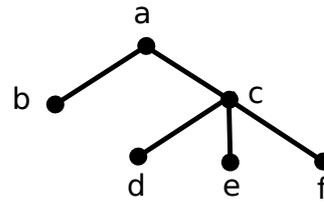
# Bäume



14

# Bäume

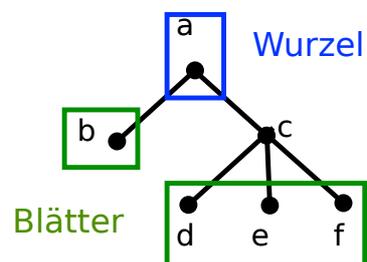
- Bestandteile
  - Menge von **Knoten** (nodes, vertices)
  - Menge von **Kanten** (edges)
- Hier immer gerichtete Bäume:
  - Kanten haben eine *Quelle* (source) und ein *Ziel* (target)
  - Im Bild: Quelle über (nördlich) Ziel
- Jeder Knoten hat höchstens eine eingehende Kante (⇒ keine Zyklen)
- Knoten können etikettiert sein (Etikett = *label*)



15

# Bäume

- Quelle und Ziel nennen wir auch **Mutter** und **Tochter** (auch: *Vater-Sohn*, *Eltern-Kind*, *Vorgänger-Nachfolger*)
- Knoten mit gleichem Vater heißen entsprechend **Geschwister**
- Der einzige Knoten ohne Vorgänger ist die **Wurzel** (root)
- Wenn es mehrere Wurzeln gibt (= die Knoten hängen nicht zusammen) sprechen wir von einem *Wald*
- Knoten ohne Nachfolger heißen **Blätter** (leaves)

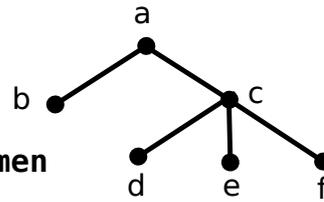


16

# Bäume

- Bäume kann man leicht als rekursive Datenstruktur implementieren:

**Baum = Wurzel-Etikett + Liste von Bäumen**



- Zum Beispiel:  
`('a', [('b', []), ('c', [('d', []), ('e', []), ('f', [])])])`
- Beachte:
  - a, b, ... sind eigentlich nur Etiketten, keine Knoten
  - Ein Knoten repräsentiert Teilbaum von sich selbst
  - Bäume sind hier auch *geordnet*, dh. die Reihenfolge in der Teilbaumliste ist wichtig

17

# Bäume (objektorientiert)

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

18

## Baumausdrücke Parsen (String $\Rightarrow$ Baum)

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

19

## Baumausdrücke Parsen (String $\Rightarrow$ Baum)

```
def parse(strng):  
    tokens = tokenize(strng)  
    return tree(next(tokens), tokens)  
  
def tokenize(strng):  
    return <Iterator über die Tokens in strng>
```

```
(S (NP (DET Der) (N Student)) (VP (V arbeitet)))
```

20

# Baumausdrücke Parsen (String $\Rightarrow$ Baum)

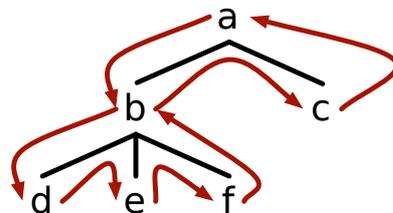
```
def tree(token, tokens):  
    if token == '(':  
        return Tree(next(tokens), list(trees(tokens)))  
    else:  
        return Tree(token, [])  
  
def trees(tokens):  
    while True:  
        token = next(tokens)  
        if token == ')':  
            break  
        yield tree(token,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

```
(S (NP (DET Der) (N Student)) (VP (V arbeitet)))
```

21

## Tiefensuche

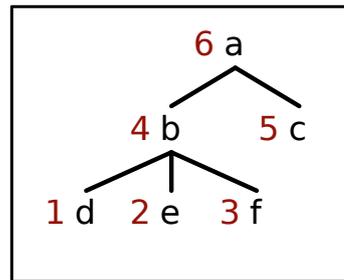
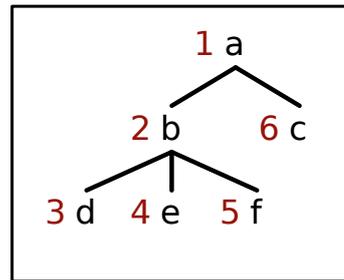
- Mit **Traversierung** bezeichnen wir das Untersuchen der Knoten eines Baumes in einer bestimmten Reihenfolge
- **Tiefensuche** (depth first search)
  - zuerst betrachtet wir die Kinder eines Knotens
  - danach seine Geschwister.



22

## Post-Order vs. Pre-Order

- Es gibt verschiedene Möglichkeiten, die Knoten eines Baumes mit Tiefensuche zu traversieren
- **Pre-Order**: jeder Knoten wird **vor** seinen Kindern betrachtet
- **Post-Order**: jeder Knoten wird **nach** seinen Kindern betrachtet



23

## 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
```

24

## Tiefensuche (Pre-Order, Generator)

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