Informatik I: Einführung in die Programmierung

Prof. Dr. Bernhard Nebel — Universität Freiburg
Dr. Christian Becker-Asano, Dr. Stefan Wölfl — Institut für Informatik
Wintersemester 2014/2015

Übungsblatt 4

Abgabe: Freitag, 21. November 2014, 18:00 Uhr

Wichtige Hinweise: Zur Bearbeitung der Übungsaufgaben legen Sie bitte ein neues Unterverzeichnis `sheet04` im Wurzelverzeichnis Ihrer Arbeitskopie des SVN-Repositories an. Ihre Lösungen werden dann entsprechend dem ersten Übungsblatt in Dateien in diesem Unterverzeichnis erwartet.

Beachten Sie bitte bei allen Aufgaben die *Hinweise zur Bearbeitung der Übungsaufgaben* unter der folgenden URL:

`http://gki.informatik.uni-freiburg.de/teaching/ws1415/info1/hinweise_uebungen.txt`

Insbesondere müssen alle Python-Dateien und die darin definierten Funktionen entsprechend diesen Hinweisen dokumentiert werden.

Überprüfen Sie, dass Sie alle Lösungen ins Repository hochgeladen haben (z.B. mit dem Befehl `svn status`). Überprüfen Sie auch die Webseite Ihres SVN-Unterverzeichnisses:

`https://daphne.informatik.uni-freiburg.de/svn/infoI1415/$RZLOGIN`

Bewertet wird bei allen Aufgaben die letzte Version, die zur Deadline des Übungsblattes auf dem SVN-Server eingereicht ist.

Aufgabe 4.1 (Josephus-Problem; Datei: `josephus.py`; Punkte: 4+2)

Wir betrachten im Folgenden eine Variante des sogenannten *Josephus-Problem* (über den geschichtlichen Hintergrund des Problems können Sie sich z.B. bei Wikipedia informieren):

> Es werden n Objekte mit dem Nummern $1, \ldots, n$ in dieser Reihenfolge in einem Kreis im Uhrzeigersinn angeordnet. Dann wird beginnend mit der Nummer k, jedes k-te Objekt aus dem Kreis entfernt, wobei der Kreis nach dem Entfernen eines Objektes sofort wieder geschlossen wird. Es wird immer im Uhrzeigersinn gezählt und für $k > n$ wird im Kreis einfach weiter gezählt. Die Reihenfolge, in der die Objekte aus dem Kreis entfernt werden, bezeichnet man als *Josephus-Permutation*. *Problem:* Gegeben natürliche Zahlen n und k, bestimme die Josephus-Permutation.

(a) Definieren Sie in der angegebenen Datei eine Funktion `josephus(n, k)`, die zu den übergebenen Werten `n` und `k` (wie in der Problembeschreibung) die Josephus-Permutation als Liste zurückgibt. Die Elemente der Liste sind also gerade die Objekte in der Reihenfolge, in der sie aus dem Kreis entfernt werden.

(b) Definieren Sie in der Datei eine Testfunktion `test_josephus()`, die die semantische Korrektheit Ihrer Funktion an 4 gut ausgewählen Testfällen aufzeigt (benutzen Sie eine Assertion `assert ...` pro Testfall). Überlegen Sie sich dazu mit Bleistift und Papier vorher, wie in den jeweiligen Testfällen die korrekte Lösung aussehen muss.

Erläuterung zu den Aufgaben 4.2 und 4.3 In den folgenden Aufgaben geht es darum, eine Zeichenfolge (einen String) einzulesen und dabei eine Datenstruktur anzulegen, die es später erlaubt für ein gegebenes Wort zu entscheiden, ob und wie oft dieses Wort in der Zeichenkette vorkommt. Unter einem *Wort* verstehen wir im Folgenden jede endliche Folge von Buchstaben des deutschen Alphabets (also den Zeichen a, ..., z, A, ..., Z, ä, Ä, ö, Ö, ü, Ü, ß) mit Ausnahme der leeren Zeichenfolge. Je zwei Wörter in der Zeichenfolge werden durch eine nicht-leere, endliche Folge von Zeichen, die nicht zu diesen Buchstaben gehören (z.B., Leerzeichen, Satzzeichen, Zeilenumbrüche), getrennt. Somit sind echte Teilfolgen eines Wortes im Folgenden nicht als Wörter zu zählen.

Aufgabe 4.2 (Nächstes Wort; Datei: `words.py`; Punkte: 3+2+1)

Im folgenden Python-Code wird eine Funktion `next_word(s)` definiert, die angewendet auf einen String `s` ein Tupel `(word, rest)` zurückgibt, wobei `word` das erste Wort (im Sinne der Erläuterung) in `s` ist und `rest` die Zeichenfolge ist, die in `s` auf `word` folgt. Falls `s` kein Wort enthält, gibt die Funktion das Tupel `(None, "")` zurück.

Die Definitionen enthalten einen syntaktischen, einen semantischen Fehler und einen Fehler, der bei vielen (auch kurzen) Eingaben zur Laufzeit auftritt.[1] Fehler sind also in genau 3 Zeilen zu finden.

(a) Finden und korrigieren Sie die Fehler! Kopieren Sie dazu den Code in die angegebene Datei, kommentieren Sie die fehlerhaften Zeilen aus (ergänzen Sie jeweils auch einen Kommentar, der den Fehler benennt und erläutert, warum er auftritt) und fügen Sie eine Korrektur der jeweils fehlerhaften Zeile nach dem Kommentarblock ein.

(b) Ergänzen Sie anschließend Testfälle, die im nicht-korrigierten Code zu semantischen oder Laufzeitfehlern führen können, als Doc-Tests in den Doc-Strings der jeweils korrigierten Funktion (pro Fehler zwei Tests).

(c) Dokumentieren und formatieren Sie den Code entsprechend den bekannten Formatierungshinweisen (in allen zukünftigen Aufgaben wird bei jeder Abgabe die Beachtung dieser Hinweise als selbstverständlich erwartet).

[1] Bei Eingaben mit sehr langen Wörtern gibt es einen weiteren Laufzeitfehler, der hier aber nicht behandelt werden soll.

```
LETTERS = "abcdefghijklmnopqrstuvwxyz" + \
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ" + \
          "ÄÖÜäöüß"

def _next_word_helper(s):
    """Helper function for next_word."""
    if not s:
        return None, s
    if s[0] not in LETTERS
        return None, s
    word = s[0]
    word_rest, s_rest = _next_word_helper(s[1:])
    if word_rest:
        word = word_rest
    return word, s_rest

def next_word(s):
    """Return the first word of an input string s and the rest of it."""
    # eat leading punctuation marks, white spaces, etc.
    while s[0] not in LETTERS:
        s = s[1:]
    return _next_word_helper(s)
```

Aufgabe 4.3 (Wort-Baum; Datei: `words.py`; Punkte: 3+3)
Mit Hilfe der korrigierten Funktion `next_word(s)` aus Aufgabe 4.2 soll nun ein Suchbaum der in einer gegebenen Zeichenfolge `s` vorkommenden Wörter wie folgt erzeugt werden: Jeder Knoten des Suchbaumes wird durch ein Tupel

```
(word, ltree, rtree, n)
```

repräsentiert. Dabei ist `word` ein Wort, `ltree` der linke Teilbaum, `rtree` der rechte Teilbaum und `n` die Anzahl der Vorkommnisse von `word` in `s`. Als Ordnungsrelation verwenden wir Python's lexikographische Ordnung von Strings. Das heißt, ein Wort `w` wird im linken Teilbaum eines Knotens `(word, ltree, rtree, n)` eingefügt bzw. gesucht, falls der Vergleich `w < word` den Wert `True` zurückgibt, etc. In Blattknoten sind `ltree` und `rtree` jeweils der Wert `None`.

(a) Definieren Sie eine Funktion `word_tree(s)`, die bei Eingabe eines Strings `s` diesen Suchbaum erzeugt und zurückgibt. Natürlich kann Ihre Funktion eine selbst-definierte Hilfsfunktion verwenden.

(b) Definieren Sie eine Funktion `freq_word_tree(tree, word)`, die für einen solchen Suchbaum `tree` und ein Wort `word`, die in `tree` hinterlegte Anzahl der Wortvorkommnisse von `word` zurückgibt. Falls das Wort in dem Baum nicht vorkommt, soll die Funktion den Wert `0` zurückgeben.

Schreiben Sie für die Funktionen in (a) und (b) auch geeignete Testfunktionen (also: `test_word_tree()` und `test_freq_word_tree()`), die die Funktionen an jeweils vier geeigneten und selbst gewählten Beispielen als *Assertions* testen.

Benutzen Sie in (b) zum Testen auch längere Texte (mehr als 1000 Wörter). In der auf der Webseite der Vorlesung verfügbaren Datei `words_data.py` finden Sie kleine Beispieltexte, die Sie für Ihre Tests verwenden können. Fügen Sie diese Datei bitte zu Ihrem SVN-Unterverzeichnis hinzu und ergänzen Sie diese um weitere Texte Ihrer Wahl. Eine Quelle ist z.B. `http://www.gutenberg.org/`.
In der Datei `wordtree.py` können Sie die Datendatei importieren, z.B. mit:

```
from words_data import loremipsum
```

Aufgabe 4.4 (Erfahrungen; Datei: `erfahrungen.txt`; Punkte: 2)

Legen Sie im Unterverzeichnis `sheet04` eine Textdatei `erfahrungen.txt` an. Notieren Sie in dieser Datei kurz Ihre Erfahrungen beim Bearbeiten der Übungsaufgaben (Probleme, benötigter Zeitaufwand nach Teilaufgabe, Bezug zur Vorlesung, Interessantes, etc.).