Programmierkurs Python I – WS 09/10

Übung 6

---

1 Bigramme Zählen (4 + 1 Punkte)

Schreibe ein Programm, das Bigramme aus einer Datei extrahiert. Bigramme sind Paare von benachbarten Wörtern. Der String *the problem with the task* enhält die Bigramme *the problem*, *problem with*, *with the*, usw. Das Programm soll zählen, wie häufig welche Bigramme vorkommen. Satzzeichen etc. sollen ignoriert werden. Die Extraktion der Bigramme soll mit einem regulären Ausdruck erfolgen (`re.findall`), und kann Zeilenweise erfolgen.

1. Schreibe eine Klasse `Bigrams`, mit (mindestens) folgenden Methoden:

   - `readFile(self,filename,enc)` soll die übergebene (Text-)Datei einlesen, dabei das Encoding `enc` annehmen und die Bigramme verarbeiten.
   - `__repr__(self)` soll eine String-Repräsentation für die Klasse sein, die Bigramm und die Anzahl seines Vorkommens pro Zeile angibt (z.B. `the dog 15`).
   - `writeToFile(self,filename)` soll die String-Ausgabe aus der `repr`-Methode in eine Datei ausgeben, deren Name als Parameter übergeben wird.

2. *Bonusaufgabe:* Erweitere das Programm so, dass es neben reinen Textdateien auch HTML-Dateien verarbeiten kann. Hierbei sollen alle HTML-Tags (z.B. `<a href="...">`) ignoriert werden, also alles was sich zwischen spitzen Klammern befindet . Eine Testdatei findest du auf der Homepage (`taz.htm`)

2 Web-Crawler (4 + 1 Punkte)

Ein Webcrawler ist ein Programm, das Links aus Webseiten herausfischt, die verlinkten Webseiten wiederum einliest und nach Links durchsucht, und somit theoretisch durch das ganze Internet „kriechen“ kann. Schreibe ein Programm mit der Basis-Funktionalität eines Webcrawlers:

1. Schreibe eine Funktion `extract(url)`, die aus HTML-Text, der von einer URL (`url`-Parameter) gelesen wird, Links extrahiert. Die Methode soll alle gefundenen URLs zurück geben. Benutze hierfür einen regulären Ausdruck. Nimm an, dass alle mit dem HTML-Tag `<a href="URL">Linkbeschreibung</a>` markierten Strings Links sind. Interessant ist hier der URL-Teil. (Unter Umständen sind außer `href` noch andere Attribute im öffnenden `<a>`-tag, wie `<a name="blabla" href="...">` - ignoriere diese Attribute.)

2. Schreibe eine Klasse `Crawler`, die in der `__init__`-Methode eine Start-URL als Parameter nimmt. Die Klasse soll eine Methode `printURLs` haben, die alle bisher gefundenen URLs auf dem Bildschirm ausgibt.

3. Implementiere in `Crawler` eine Methode `crawl(self, depth)`. Diese Methode soll die Funktion `extract` aus 1. benutzen. Zunächst wird der Crawler alle URLs, die er auf der Start-Seite findet, speichern. Mit den gesammelten URLs wird die gleiche Prozedur durchlaufen (Links auf den Seiten sammeln). Jede URL soll im ganzen Programmdurchlauf nur ein einziges Mal besucht werden.

   Der Crawler soll nicht unendlich laufen, dafür gibt es den Parameter `depth`. Der Parameter sagt, wieviele Komplette Durchläufe der Crawler macht. Ein Durchlauf bedeutet, dass alle URLs, die auf den aktuell gespeicherten (und noch nicht betrachteten) Links zu finden sind, besucht werden und aus ihnen wiederum neue URLs extrahiert werden. Für Test-Zwecke empfehlen wir Euch, `depth` kleiner 4 zu wählen.

   Bedenke dass es auf manchen Homepages sog. „tote“ Links gibt, also Links, die ungültig sind. In solchen Ffällen wird `urllib` eine Exception werfen, die sollte den Programmfluss nicht stören.

Zum Testen kannst du z.B. bei `http://www.taz.de/` starten.

Bonusaufgabe: Implementiere den korrekten Umgang mit relativen Links - das sind alle, die kein vollständiges URL-Präfix haben. Die Links funktionieren wie Datei-Pfade, nur mit Angaben relativ zur aktuellen URL: `<a href="../index.html">` aufgerufen von `http://www.test.de/test.html` z.B. leitet zu `http://www.test.de/index.html`. Eine Liste solcher Schemen für relative URLs findet sich hier: `http://www.webreference.com/html/tutorial2/3.html`

3 Einfache Web-Linguistik (Bonus, 4 Punkte)

In der Vorlesung haben wir gezeigt, wir man mit dem Modul `urllib.request` Suchmaschinenergebnisse auswerten kann. Erweitere die Funktionalität der gezeigten Technik um zwei Dinge:

- Schreibe eine Methode, die für eine String-Anfrage die Anzahl der Treffer als Zahl (int) zurück gibt. Bedenke, dass Du Leerzeichen etc. entsprechend ersetzen musst. (→ mit dem Web-Interface der Suchmaschine ausprobieren.)
- Schreibe eine Methode, die erlaubt, nach Wildcards (*) zu suchen. So etwas wird in manchen linguistischen Ansätzen für flache Semantik benutzt, z.B. könnte man im Web suchen nach *„ zuerst * und dann bezahlen“* um herauszufinden, was man typischerweise zeitlich vor *bezahlen* macht. (Anderes Beispiel: *„ * und andere Tiere“*, um verschiedene Substantive zu finden, die alle zur Klasse „Tier“ gehören.) Deine Methode soll einen Such-String mit Wildcards als Parameter nehmen, daraus eine Anfrage machen und alle möglichen „Füller“ für die Wildcards zurückgeben. (Für die Anfrage *„ * und andere Tiere“* sollte z.B,. eine Liste zurückkommen mit *Katzen, Vögel, Schlangen* etc. Bedenke, dass Du für diese Art von Suche Anführungszeichen brauchst. Bei mehreren Wildcards soll eine Liste von Tupeln zurückgegeben werden, die für jedes Suchergebnis alle Wildcard-Entsprechungen enthalten.

4 Erweiterte Textsuche (Bonus, 4 Punkte)

Erweitere den Webcrawler aus Aufgabe 2 so, dass er zusätzlich Suchstrings aus den Texten herausfiltert. Die Strings sollen einfache Reguläre Ausdrücke beinhalten dürfen, die als *Wildcard* dienen (so wie „*“ in der Websuche, nur genauer spezifiziert.) - ohne Capturing Groups in der Anfrage, aber mit Buchstaben-Klassen (`[a-Z]` etc.). Der Crawler soll sich beim crawlen die Matches für den Such-Ausdruck aus den Homepage-Texten merken. Beispiel: Der Crawler bekommt als Suchtext *so \w+ wie ein Hund*. Als Ergebnis soll er alle möglichen Worte liefern, die anstelle des *\w+* stehen können (lt. Web-Suche z.B. *groß*, *anhänglich*, ...). Es sollen alle Matches der regulären Ausdrücke gefunden werden: *so \w+nd wie ein Hund* könnte als Ergebnis z.B. *elend* oder *hechelnd* liefern.

Abgabe bis Donnerstag, 10.12.2009, 14:00 Uhr per Mail an
`regneri@coli.uni-sb.de`
`lcarolyn@coli.uni-sb.de`