Programmierkurs Python I

Michaela Regneri

2009-12-17

(Folien basieren auf dem gemeinsamen Kurs mit Stefan Thater)

Heute ...

- Ein buntes Sammelsurium an nützlichem Kleinkram
  - Funktionen & Parameter: Default-Werte & mehr
  - Formatierte Ausgabe
  - Ein paar Module aus der Standardbibliothek
- Hinweise für das Projekt
  - Parsen von „schmutzigem“ XML
  - Nützliche Unix-Werkzeuge
- Beispiellösungen für das letzte Übungsblatt

Funktionen & Parameter

- Bisher haben wir den Fall betrachtet, dass eine feste Anzahl Parameter spezifiziert wird: `def f(a, b): ...`
- Weitere Möglichkeiten:
  - Parameter mit Default-Werten
  - Schlüsselwörter als Parameter
  - Variable Anzahl an Parametern

Schlüsselwörter & Defaults

```
def write(thing, newline = True):
  sys.stdout.write(thing)
  if newline:
    sys.stdout.write('\n')
```

- Aufrufen der Funktion:
  - `write('bla')`
  - `write('bla', False)`
  - `write('bla', newline = False)`
- Funktion kann mehrere Schlüsselword-Parameter haben; sie müssen nach den normalen Argumenten stehen.

Achtung!

- Die Default-Werte werden nur einmal, während der Auswertung der Funktionsdefinition ausgewertet.
- Das kann unerwünschte Nebeneffekte haben:

```
def add1(it, result = []):
  for n in it:
    result.append(n + 1)
  return result
```

```
>>> add1([1,2])
[2, 3]
>>> add1([1,2])
[2, 3, 2, 3]
```


Variable Anzahl an Argumenten

```
def sum(*args):
  result = 0
  for arg in args:
    result += arg
  return result
```

```
>>> sum()
0
>>> sum(1,2,3)
6
```

- Funktionen mit einem *-Parameter können mit beliebig vielen Argumenten aufgerufen werden:
  - Die Argumente werden in ein Tupel verpackt und an den Parameter gebunden.
- Der *-Parameter muss den normalen Parametern folgen.

Variable Anzahl an Argumenten

```
def makedict(**args):
  return args

>>> makedict(foo=1, bar=2)
{foo:1, bar:2}
```

- Das Ganze geht auch mit Schlüsselwort-Argumenten: Schlüsselworte werden in ein Wörterbuch (dict) verpackt und an den **-Parameter gebunden.

Docstrings

```
def max(arg, *args):
    """ max(a, b, c, ...) -> value
        Returns the largest argument """
    for a in args:
        if a > arg:
            arg = a
    return arg
```

- Strings mit drei Anführungszeichen können Zeilenumbrüche enthalten; ansonsten sind sie äquivalent zu normalen Strings.
- `help(max)` und `max.__doc__` liefern den Docstring

Formatierte Ausgabe

- Mit der Methode `str.format` kann man hübsch formatierte Strings erzeugen:
  - print("{0:s}: {1:.2f}".format('foo', 1.0 / 3.0)) ⇒ foo: 0.33
- Aufgerufen wird von einem „Formatierungs-String“
- Der Parameter muss ein Tupel sein
- auf die Parameter wird im String mit geschweiften Klammern zugegriffen ({0} der erste, {1} der zweite...)
- wenn die Parameter in ihrer kanonischen Reihenfolge benutzt werden sollen, kann man die Zahlen weglassen "{0}:{1}:{2}" ≈ "{}:{}:{}"

Formatierte Ausgabe

- Der String kann Formatierungs-Spezifikationen (:x) enthalten; die hilft besonders bei der Formatierung von (Fließkomma-)Zahlen
  - :s – ein String
  - :d – eine Ganzzahl
  - :f – eine Fließkommazahl
  - -:***.x***f - Fließkommazahl auf **x** Nachkomma-Stellen gerundet

Formatierte Ausgabe

- Man kann auch Schlüsselworte verwenden:
  - `'some {bla:s}'.format(bla='blub')⇒ 'some blub'`
- Es gibt auch noch die alten Formatierungs-Strings, die aber demnächst entfallen:
  - "%s: %.2f" % ('foo', 1.0 / 3.0)
- Für Details siehe
  http://docs.python.org/3.1/library/string.html#string-formatting

Das Projekt

- Ziel des Projekts ist es, eine kleine Suchmaschine zu implementieren
  - Es sollen Suchanfragen mit boolschen Operatoren formuliert werden können.
  - Ergebnis: eine Liste von Dokumenten, die die entsprechende Suchanfrage erfüllen.
  - Nur „relevante“ Dokumente
- Die Dokumentsammlung: Das Gigaword-Korpus (NYT).

Ein Wort zur „Relevanz“

- Bei der Suche nach Dokumenten, die ein bestimmtes Wort enthalten, ist man normalerweise vor allem an „relevanten“ Dokumenten interessiert.
- Einfaches Relevanzmaß: Wort-Frequenz
  - Problem: es wird nicht zwischen Wörtern, die insgesamt häufig vorkommen („ein“, „der“, ...), und Wörtern, die nur in einzelnen Dokumenten häufig vorkommen, unterschieden.
- Besseres Relevanzmaß: tf-idf

tf-idf

- Das tf-idf Maß setzt die Frequenz eines Wortes in einem Dokument mit der Frequenz über alle Dokumente in Beziehung:
  - $tf_{i,j} = n_{i,j} / \sum_k n_{k,j}$
  - $idf_j = \log (D / D_j)$
  - $tfidf_{i,j} = tf_{i,j} * idf_i$

  - $n_{i,j}$ = Anzahl Vorkommen von Wort w_i in Dokument d_j
  - D = Anzahl aller Dokumente
  - D_j = Anzahl der Dokumente, in denen w_j vorkommt
- Eine hoher td-idf Wert ergibt sich für Wörter mit hoher Termfrequenz (in einem Dokument) und niedriger Dokumentfrequenz.

Das Gigaword-Korpus

- Wir betrachten das NYT-Teilkorpus („New York Times“)
- Dokumente sind nach Jahrgang und Monat in einzelnen komprimierten Dateien zusammengefasst.
  - Dateien können mit gzip.open(...) geöffnet werden.
- Das Format ist leider keine wohgeformtes XML, man kann den SAX-Parser nicht direkt verwenden.
  - Wurzelelement fehlt
  - Nicht deklarierte Entity: &

Parsen von Gigaword-Dateien

- Variante 1: man „erfindet“ ein neues Wurzelelement und deklariert die fehlende Entity:

```
p = xml.sax.make_parser()
p.setContentHandler(MyHandler())
p.feed('<wurzel>')
# rest des Dokuments füttern
p.feed('</wurzel>'
p.close()
```

- Variante 2: man verwendet den HTML-Parser
- Natürlich kann man auch die Quelldateien editieren ...

Große Wörterbücher

- Dictionaries sind oft nütztlich, auch zum Wörterzählen
- Wenn der Speicher (RAM) nicht ausreicht, kann man alternativ mit `shelve` Objekten arbeiten
- Ein `shelve` ist ein persistentes Wörterbuch, das auf der Festplatte gespeichert ist.
- Einschränkungen: mögliche Schlüssel sind Plattformabhängig!
  - „Sicher“: Byte-Strings
  - fast sicher: Unicode-Strings

Wörter zählen: `dict`

```
import sys

def main():
    freqs = dict()
    with open(sys.argv[1]) as f:
        for line in f:
            for word in line.split():
                if word in freqs:
                    freqs[word] += 1
                else:
                    freqs[word] = 1
    for word, freq in freqs.items():
        print('{:d}\t{:s}'.format(freq, word))

if __name__ == '__main__':
    main()
```

Wörter zählen: shelve

```
import sys, shelve

from contextlib import closing

def main():
   with closing(shelve.open('wc')) as freqs:
      with open(sys.argv[1]) as f:
         for line in f:
            for word in line.split():
               if word in freqs:
                  freqs[word] += 1
               else:
                  freqs[word] = 1
      # ...

if __name__ == '__main__':
   main()
```


shelve: Objekte als Werte

- shelves funktionieren *fast* wie Dictionaries
- ABER: wenn die Werte Objekte sind, wird eine Änderung nicht automatisch erkannt

```
> s = shelve.open(datei)
> s[”liste“] = []
> s[”liste“].append(3)
> print(s[”liste“])
[]
```

shelve: Objekte als Werte

- 1. Lösung: „vorsichtig“ (sorgfältig) programmieren

```
> s = shelve.open(datei)
> s[”liste“] = []
> l = s[”liste“]
> l.append(3)
> s[”liste“] = l
> print(s[”liste“])
[3]
```


shelve: Objekte als Werte

- 2. Lösung: writeback-Parameter

```
> s = shelve.open(datei, writeback=True)
> s[”liste“] = []
> s[”liste“].append(3)
> print(s[”liste“])
[3]
```

- Das sieht zunächst kürzer / eleganter aus, ABER:
- alle behandelten Einträge bleiben im Speicher!
- der Vorteil von shelve-Objekten ist so fast weg

with expr [as variable]

- Dateien sollten immer geschlossen werden.
- Bequeme Syntax: das with-Statement

```
with open(filename) as f:
  for line in f:
    # ...
```

- Hier wird nach dem with-block die Datei automatisch geschlossen.
- expr muss zu einem Objekt auswerten, das die beiden Methoden __enter__ und __exit__ implementiert.

with closing(expr) as var

- Leider implementieren nicht alle Objekte die beiden Methoden __enter__ und __exit__
- ... und können nicht (direkt) mit with verwendet werden.

```
import shelve
with shelve.open(filename) as f:
    # ...

=> AttributeError: 'DbfilenameShelf' object has
no attribute '__exit__'
```

contextlib

- Das contextlib-Modul implementiert einige nützliche Funktionen, die die fehlende Funktionalität „nachrüstet“

```
from contextlib import closing
import shelve
with closing(shelve.open(filename)) as f:
    # ...
```

- Randbemerkung: `shelve.open(”blah“)` erzeugt eine Datei namens `blah.db` im aktuellen Verzeichnis

Nützliche Unix-Werkzeuge: ssh

- Mit ssh kann man sich auf einen entfernten Rechner im Netzwerk anmelden:
  - ssh regneri@login.coli.uni-saarland.de
- Man beachte: wenn die ssh-Verbindung getrennt wird, werden (normalerweise) alle laufenden Prozesse getötet.

Nützliche Unix-Werkzeuge: Screen

- Screen erlaubt es, mehrere „virtuelle“ Terminal-Sitzungen in einem Terminal auszuführen.
- Mit „ctrl-a d“ kann man eine Screen-Sitzung „abtrennen“
- Mit „screen -r“ holt man sich eine abgetrennte Screen-Sitzung zurück.
- Extrem nützlich bei zeitaufwändigen Berechnungen: man kann sich abmelden, ohne dass der Prozess getötet wird.

Prettyprinter XML

```
import xml.sax
from xml.sax.handler import ContentHandler
import sys

class PrettyHandler(ContentHandler):

    def __init__(self, fileout):
        self.fileout = fileout
        self.current_cdata = ""
        self.current_prefix = ""
        self.last_tag_opened = "" #fuer leere Tags
        self.buffer =""           # fuer leere Tags
```

Prettyprinter XML

```
def startElement(self,name, attrs):
    # alles, was in buffer gespeichert wird,
    # kann man bei Nichtbeachtung
    # leerer Tags gleich in fileout schreiben.

    self.fileout.write(self.buffer)
    self.buffer = ""
    if self.current_prefix:
        self.buffer += "\n" + self.current_prefix + "<" +name
    else:
        self.buffer += "<" +name
    if attrs:
        for a,v in attrs.items():
            self.buffer += " " + a + '="' + v + '"'
    self.buffer +=  ">"
    self.current_prefix +=  "\t"
    self.last_tag_opened = name
```


Prettyprinter XML

```
def endElement(self, name):
   # verarbeitet die CDATA des aktuellen Texts  (wenn es sie
   # gibt) und entscheidet, ob der Schließende Tag in eine eigene
   # Zeile kommt, oder hinter die CDATA, oder ob ein leerer Tag vorlag
   self.current_cdata = self.current_cdata.strip()
   self.current_prefix = self.current_prefix[1:]
   if self.current_cdata:
      # normale Tags mit CDATA (-> Endtag auf gleiche Zeile wie CDATA)
       self.fileout.write(self.buffer)
       self.buffer = ""
       self.fileout.write(self.current_cdata)
   else:
       if name == self.last_tag_opened:
           # = leere tags
           self.fileout.write(self.buffer[:-1] + "/>")
           self.buffer =""
           self.current_cdata = ""
           return
       else:
           # keine CDATA, aber auch kein leerer
           # Tag (-> Endtag auf eigene Zeile)
           self.fileout.write(self.buffer)
           self.buffer = ""
           self.fileout.write("\n" + self.current_prefix)
   self.fileout.write("</" + name + ">")
   self.current_cdata = ""
   self.lasttag_opened = ""
```

Prettyprinter XML

```
        #....

      def characters(self, characters):
         self.current_cdata = self.current_cdata + characters


with open('text.xml','w') as f:
    parser = xml.sax.make_parser()
    parser.setContentHandler(PrettyHandler(f))
    parser.parse('sentence.xml')
```


Prettyprinter HTML

```
    from html.parser import HTMLParser
import re

class PrettyHTMLParser(HTMLParser):

    def __init__(self, fileout):
        super().__init__()
        self.fileout = fileout
        self.current_cdata = ""   # wie xml-Parser
        self.numbers = []   # für nummerierte Listen
        # für nicht-nummerierte Listen
        self.nextsign = -1
        self.current_prefix = ""
        self.signs = [ "* ","- ", "# "]

        self.num = False          # Typ der aktuellen Liste
        self.charcounter = 0      # für Textumbruch
```

Prettyprinter HTML

```
def handle_starttag(self, tag, attrs):
      if tag == "ol":
         self.num = True
         self.numbers.append(1)
         self.current_prefix +=  "\t"
      elif tag == "ul":
         self.num = False
         self.nextsign += 1
         if self.nextsign >= len(self.signs):
            self.nextsign = 0
         self.current_prefix +=  "\t"
      elif tag =="li":
         self.charcounter = len(self.current_prefix)
         if self.num:
          self.fileout.write("\n" + self.current_prefix  + str(self.numbers[-1]) + " ")
          self.numbers[-1]+= 1
         else:
          self.fileout.write("\n" + self.current_prefix  + self.signs[self.nextsign])
      elif tag == "p":
            self.fileout.write("\n" + self.current_prefix)
            self.charcounter = len(self.current_prefix)
```


Prettyprinter HTML

```
self.numbers = []     aus __init__
self.nextsign = -1
self.current_prefix = ""
self.signs = [ "* ","- ", "# "]
```

```
    def handle_endtag(self, tag):
    if tag == "ol" or tag == "ul":
        self.current_prefix = self.current_prefix[1:]
        self.fileout.write("\n")
    if tag=="ol":
        del self.numbers[-1]
    elif tag=="ul":
        self.nextsign -= 1
        if self.nextsign < 0:
            self.nextsign = len(self.signs) -1
    elif tag=="p":
        self.fileout.write("\n" + self.current_prefix)
        self.charcounter = len(self.current_prefix)
```

Prettyprinter HTML

```
self.numbers = []    aus __init__
self.nextsign = -1
self.current_prefix = ""
self.signs = [ "* ","- ", "# "]
```

```
def handle_data(self, data):
        for char in data.strip():
            self.charcounter += 1
            if self.charcounter >= 78 and re.match(r"\s", char):
                self.fileout.write("\n" + self.current_prefix)
                self.charcounter = len(self.current_prefix)
            else:
                self.fileout.write(char)
                if(char == "\n"):
                    self.fileout.write(self.current_prefix)
                    self.charcounter = len(self.current_prefix)


def handle_startendtag(self, tag, attrs):
    if tag=="br":
        self.fileout.write("\n" + self.current_prefix)
        self.charcounter = len(self.current_prefix)
```


XHTML-Konverter

- Hier nur die Basisfunktionalität: bestimmte Tags werden automatisch geschlossen, damit das resultierende xhtml-Dokument wohlgeklammert ist.
- Idee: Der Parser (das Handler-Objekt) verwaltet einen Stack, auf dem alle offenen Tags gespeichert werden.
- Wenn das aktuelle Tag besucht wird, werden alle passenden Tags auf dem Stack geschlossen.

XHTML-Konverter

```
# ein <p> schließt ein vorhergehendes <p>,
# ein <li> schließt vorhergehende <p> und <li>, usw.
_closes = { 'p' : ['p'], 'li' : ['p', 'li'], ... }

def closes(tag, other):
   try:
      return other in _closes[tag]
   except KeyError:
      return False
```


XHTML-Konverter

```
class MyParser(HTMLParser):
   def __init__(self):
      super().__init__()
      self._stack = []

   def push(self, tag):
      self._stack.append(tag)
   def pop(self):
      return self._stack.pop()
   def peek(self):
      return self._stack[-1] if self._stack != [] else None
   # ...
```

XHTML-Konverter

```
class MyParser(HTMLParser):
   # ...
   def handle_starttag(self, tag, attrs):
      while closes(tag, self.peek()):
         self.write('</{}>'.format(self.peek()))
         self.pop()
      self.push(tag)
      self.write('<{}{}>'.format(tag,makeAttributeString(attrs)))
   # ...
```


XHTML-Konverter

```
class MyParser(HTMLParser):
   # ...
   def handle_endtag(self, tag):
      while closes(tag, self.peek()):
         self.write('</{}>'.format(self.peek()))
         self.pop()
      self.pop()
      self.write('</{}>\n'.format(tag))
   def handle_data(self, data):
      self.write(data)
   def handle_entityref(self, ent):
      self.write('&{};'.format(ent))
   # ...
```

XHTML-Konverter

```
class MyParser(HTMLParser):
   # ...
   def write(self, thing):
      sys.stdout.write(thing)


def makeAttributeString(attrs):
    ret = ” ”
    for key, val in attrs:
       ret += '{}="{}"'.format(key,val)
    return ret
```


XHTML-Konverter

```
def main():
   with closing(MyParser()) as parser:
      for filename in sys.argv[1:]:
         with closing(open(filename)) as f:
            for line in f: parser.feed(line)
```