Programmierkurs Python

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

Winter 2010/11

Übersicht

- Dateien
  - Ein- und Ausgabe
- Mehr zu Strings
  - Einige wichtige Methoden
  - Formatierte Ausgabe
  - Kodierungen
- Ausnahmen

Dateien

- Bisher haben wir Eingaben stets über die Kommandozeile eingelesen.

```
import sys
eingabe = sys.argv[1]
...
```

- Das ist natürlich nur mit „kleinen“ Eingaben sinnvoll möglich
- Programme können ihre Eingabe auch aus Dateien lesen
  - bzw. ihre Ausgaben in Dateien schreiben.

Dateien

- `f = open(filename, mode)`
  - öffnet die Datei filename zum Lesen (mode 'r') oder Schreiben (mode 'w') und liefert ein „Dateiobjekt“
  - normalerweise werde Dateien im „Textmodus“ geöffnet
- `f.close()`
  - schließt die Datei wieder
- Anmerkungen:
  - Im Textmodus werden bestimmte Zeichen in der Eingabedatei als Zeilenendmarkierungen interpretiert („newline“)
  - Dateien sollten immer geschlossen werden, wenn man nicht mehr auf sie zugreifen möchte (!)

Dateien

- Methoden für die Ein- und Ausgabe
  - `f.write(something)` – schreibt den String `something` in `f`
  - `f.read()` – liest die gesamte Datei
  - `f.readline()` – liest eine einzelne Zeile
  - `f.readlines()` – liest alle Zeilen
- `print(something, file=dest)`
  - schreibt `something` in die Datei `dest`

Ein Beispiel

```
def grep(filename, word):
    '''Enthält die Datei filename das Wort word?'''
    f = open(filename)
    while True:
        line = f.readline()
        # Dateiende erreicht?
        if line == '':
            break
        if word in line:
            return True
    f.close()
    return False
```

Das `with` Statement

- Dateien sollten immer geschlossen werden, sobald man nicht mehr auf sie zugreifen möchte.
- Mit der `with` Anweisung können Dateien automatisch geschlossen werden, sobald der `with`-block verlassen wird.

```
def grep(filename, word):
    with open(filename) as f:
        while True:
            [...]
        return False
```


Das `for` Statement

- `for line in file: block`
  - Dateien sind (wie Listen, Mengen, ...) iterierbar
  - Beachte: line enthält „newline“ am Zeilenende

```
def grep(filename, word):
    with open(filename) as f:
        for line in f:
            if word in line:
                return True
        return False
```

Stdin, Stdout

- Tastatur und Bildschirm sind auch Dateien:
  - sys.stdin („Tastatur“)
  - sys.stdout („Bildschirm“)

```
# grep.py <wort> - sucht <wort> in stdin
import sys
for line in sys.stdin:
    if sys.argv[1] in line:
        sys.stdout.write(line)
```


Puffer

- Dateioperationen werden typischerweise gepuffert
  - Beim Schreiben werden Daten in einem Puffer zwischengespeichert
  - Erst wenn der Puffer voll ist, werden die Daten tatsächlich in die Datei geschrieben
- `f.flush()` erzwingt das Leeren des Puffers

Strings

Strings – einige Methoden

- `s.strip([z]), s.rstrip([z]), s.lstrip([z])`
  - Liefert eine Kopie von s, Leerzeichen [Zeichen aus z] am Rand, bzw. nur am rechten oder nur am linken Rand entfernt
- `s.split([z])`
  - Liefert eine Liste aus Strings, die man erhält, wenn man s an allen Leerzeichen trennt
  - Alternatives Trennzeichen kann mit z spezifiziert werden

Strings – einige Methoden

- `s.isdigit()`
  - sind alle Zeichen in s Ziffern?
- `s.isalpha()`
  - sind alle Zeichen in s Buchstaben?
- `s.isalnum()`
  - sind alle Zeichen in s Ziffern oder Buchstaben oder '_'?
- `s.isspace()`
  - sind alle Zeichen in s Leerzeichen?

Strings – einige Methoden

- `s.isupper(), s.islower()`
  - alle Zeichen in s groß? klein?
- `s.upper(), s.lower()`
  - Liefert eine Kopie von s, alle Buchstaben groß bzw. klein
- `s.capitalize()`
  - Liefert eine Kopie von s, erstes Zeichen in Großbuchstaben
- Weitere Methoden:
  - http://docs.python.org/py3k/library/stdtypes.html

Formatierte Ausgabe

- Mit der format-Methode können Strings elegant „formatiert“ werden.

```
>>> '{0}, {1} und {2}'.format('a', 'b', 'c')
'a, b und c'
>>> '{}, {} und {}'.format('a', 'b', 'c')
'a, b und c'
>>> 'int: {0:d}; hex: {0:x}'.format(42)
'int: 42; hex: 2a'
>>> '{0:.2f}'.format(3.1415)
3.14
```

- http://docs.python.org/py3k/library/string.html

Formatierte Ausgabe

- Alternativ kann man den %-Operator verwenden
  - gilt als veraltet (wird nicht mehr unterstützt)

```
>>> '%s %s und %s' % ('a', 'b', 'c')
'a, b und c'
>>> '%.2f' % 3.1415
3.14
```

Unicode-Strings & Byte-Strings

- Python kennt zwei Arten von Strings:
  - Unicode-Strings (`"Hallo"`)
  - Byte-Strings (`b"Hallo"`)
- Bytestring = Folge von Bytes (Zahlen)
  - Einschränkung auf maximal 255 verschiedene Zeichen
  - Beachte: `b"wort"[0]` ⇒ `119`
- Unicode-Strings unterliegen im Unterschied zu Byte-Strings keinerlei Einschränkung

Kodierungen (Encodings)

- Strings sind Folgen von Zeichen
- Computer kennen aber keine Zeichen, nur Zahlen
  - Strings werden intern als Folgen von Zahlen repräsentiert
- Wir brauchen also eine Abbildung, die Zahlen und Zeichen einander zuordnet
- Solche Abbildungen nennt man „Kodierungen“

Kodierungen (Encodings)

- ASCII ist eine einfache (7-Bit) Kodierung, die den Zeichen der englischen Sprache Zahlen zwischen 32 und 127 zuweist (Zahlen ≤ 31 sind sogenannte Steuerzeichen).
  - ASCII kennt keine Umlaute!
  - Byte-String-Literale müssen ASCII-kodiert sein
- Einige Erweiterungen von ASCII
  - ISO-8859-1 („latin1“) – westeuropäische Sprachen
  - ISO-8859-2 („latin2“) – osteuropäische Sprachen
- Die „Latin“ Kodierungen sind 8-Bit Kodierungen, kennen also jeweils 256 verschiedene Zeichen
  - Umlaute können dargestellt werden

Kodierungen (Encodings)

- Die flexibelste Kodierung ist Unicode
  - Unicode deckt (theoretisch) alle Zeichen aller Alphabete ab
  - Aktuell etwa 100.000 verschiedene Zeichen
- Verschiedene „Kodierungsschema“ legen fest, wie die Zeichenfolgen konkret als Zahlenfolge dargestellt werden.
  - UTF-8, UTF-16, UTF-32
- UTF-8 ist kompakteste Darstellung (lateinisches Alphabet)
  - ASCII ist eine Teilmenge von UTF-8

Kodierung im Quellcode

- Wenn der Quelltext nicht in UTF-8 vorliegt, muss die Kodierung für String-Literale explizit festgelegt werden:

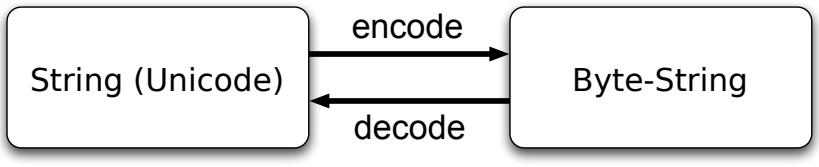
```
# -*- coding: latin1 -*-
print("Hällo, Wörld!")
```

- Ohne explizite Angabe der Kodierung funktioniert obiges Beispiel nicht. Für die gleiche Funktionalität:

```
print("H\xe4llo, W\xf6rld!")
```


Unicode-Strings & Byte-Strings

- Konvertieren:
  - `s.encode([encoding])`
  - `b.decode([encoding])`
- Beispiele:
  - `"Hällo".encode('latin1') ⇒ b'H\xe4llo'`
  - `b'H\xe4llo'.decode('latin1') ⇒ "Hällo"`



Warum zwei Arten von Strings?

- Normalerweise arbeiten wir mit (Unicode) Strings
- Einige Funktionen liefern aber Byte-Strings

```
>>> from urllib.request import urlopen
>>> f = urlopen('http://www.python.org/')
>>> f.read(40)
b'<!DOCTYPE html PUBLIC "-//W3C//DTD [...]'
```

- Byte-Strings sind robuster - man muss die Kodierung nicht unbedingt kennen, um mit Dateien arbeiten zu können
  - Die Kodierung brauchen wir erst, wenn wir Strings interpretieren wollen

„Interpretieren“ von Strings

- Manche String-Methoden „interpretieren“ die einzelnen Zeichen (`s.upper()`, `s.isalpha()`, …)
- Einige Methoden sind sensibel gegenüber Kodierungen und funktionieren nicht mit Umlauten
  - `"Hällo".isalpha()` ⇒ `True`
  - `"Hällo".encode("utf-8").isalpha()` ⇒ `False`

Kodierungen von Dateien festlegen

- `f = open(filename, encoding='latin1')`
  - Die Datei muss in der entsprechenden Kodierung vorliegen
  - `f.read()` liefert Strings
- `f = open(filename, 'br')`
  - Die Datei kann in einer beliebigen Kodierung vorliegen
  - `f.read()` liefert Byte-Strings

Ausnahmen

Ausnahmen (Exceptions)

- Ausnahmen sind Fehler, die während des Programmablaufs auftreten
  - `liste = [1,2,3]`
  - `liste[4]` ⇒ `IndexError: list index out of range`
- Wenn eine Ausnahme „geworfen“ wird, bricht Python normalerweise mit der Abarbeitung des Programms ab …
  - … es sei denn, die Ausnahme wird explizit „gefangen“
- Ausnahmen können auch dazu verwendet werden, um „nicht-lokale Sprünge“ zu implementieren.

Ausnahmen: Motivation

- Man kann Fehler in Funktionen bzw. Methoden durch definierte Rückgabewerte anzeigen (z.B. None).
- Manchmal ist es aber furchtbar umständlich, auf mögliche Fehlerwerte zu testen …
- und führt wiederum zu fehleranfälligen Programmen

Ausnahmen fangen: `try` ... `except`

- $\texttt{block}_1$ wird ausgeführt
- Wenn dabei eine Ausnahme auftritt:
  - abbruch der Ausführung von $\texttt{block}_1$
  - stattdessen wird $\texttt{block}_2$ ausgeführt
- Optionales `else`:
  - $\texttt{block}_3$ wird ausgeführt, wenn in $\texttt{block}_1$ keine Ausnahme aufgetreten ist
- Danach läuft das Programm normal weiter

```
try:
   block1
except:
   block2
[else:
   block3]
```


Ausnahmen fangen: `try` ... `except`

- Typischerweise möchte man nur bestimmte Ausnahmen behandeln
- Dazu schreibt man den (Klassen-) Namen der Ausnahme in die except Anweisung
  - except IndexError: [...]
- Wenn man auf unterschiedliche Ausnahmen unterschiedlich reagieren will, können mehrere except-Anweisungen angegeben werden
- else kommt immer nach dem letzten except-Block

```
try:
   block1
except Exn1:
   block2
[except Exn2:
   block3]
[...]
[else:
   blockn]
```

Beispiel

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'hat', len(f.readlines()), 'Zeilen')
        f.close()
```


Noch ein Beispiel

```
# wordcount.py - Zählt Wörter
import sys
wc = dict()
with open(sys.argv[1]) as f:
   for line in f:
      for token in line.split():
         try:
            wc[token] += 1
         except KeyError:
            wc[token] = 1
for (token, count) in wc.items():
   print(token + ' ' + str(count))
```

KeyError wird geworfen, wenn man mit einem nicht vorhandenen Schlüssel auf ein dict zugreift:

```
>>> d = dict()
>>> d[1] = 2
>>> d[1]
1
>>> d[2]
KeyError: 2
```

`finally`

- `finally` garantiert, dass der nachfolgende Code auf jeden Fall ausgeführt wird
- Wenn eine Ausnahme gefangen wird
  - erst wird $\texttt{block}_2$ ausgeführt
  - dann wird $\texttt{block}_3$ ausgeführt
- Wenn eine unbehandelte Ausnahme auftritt,
  - erst wird $\texttt{block}_3$ ausgeführt
  - dann die Ausnahme nochmals geworfen

```
try:
   block1
except Exn:
   block2
finally:
   block3
```


Das with Statement (revisited)

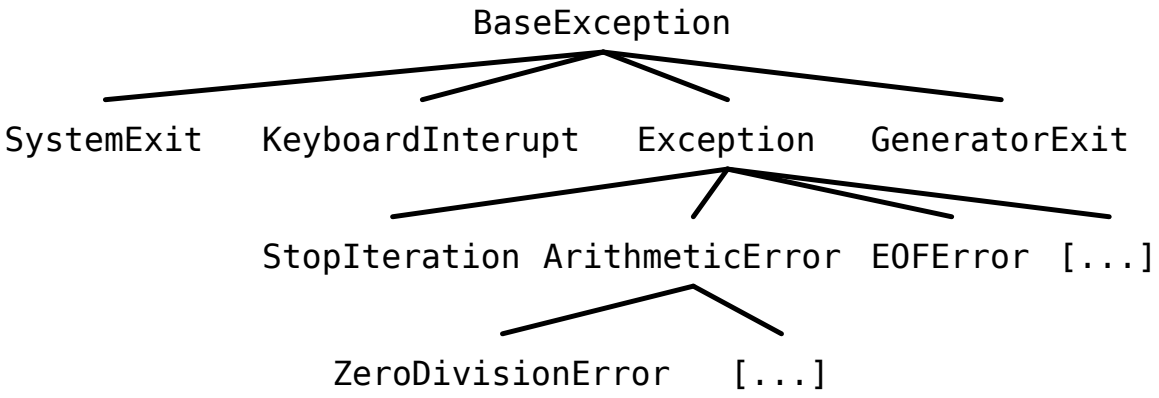
- `with open(filename) as f: block`
  - die Datei `filename` wird geöffnet
  - das entsprechende Dateiobjekt wird `f` zugewiesen
  - `block` wird ausgeführt
  - die Datei wird automatisch geschlossen, wenn `block` verlassen wird (normal oder durch Werfen einer Ausnahme)

Ausnahmen als Klassen

- Ausnahmen sind (Instanzen von) Klassen
- Alle in Python eingebauten Ausnahmen sind von der Klasse `Exception` abgeleitet
  - `except Exception` fängt alle Ausnahmen
  - (ist also äquivalent zu `except` ohne Argument)
- Variablen kann man die konkrete Instanz einer Ausnahme zuweisen:

```
try:
   block1
except Exception as e:
   print(e)
```


Klassenhierarchie



Ausnahmen werfen

- Ausnahmen wirft man mit `raise ⟨Ausnahme⟩`
- `⟨Ausnahme⟩` ist eine Instanz einer von `Exception` abgeleiteten Klasse
- Wenn die `__init__` Methode der Klasse keine Argumente hat, kann man einfach den Klassennamen hinschreiben:
  - `raise Exception` statt `raise Exception()`

Eigene Ausnahmen definieren

- Man kann sich selbst Ausnahmen definieren
- Ausnahmen sollten von `Exception` erben
  - Sie müssen aber von `BaseException` erben
- Die Methode __str__ definiert die Fehlermeldung

```
class MyError(Exception):
   def __init__(self, msg):
      self.msg
   def __str__(self):
      return self.msg
...
try:
   ...
   raise MyError("Ooops")
   ...
except MyError as e:
   print(e)
```

Zusammenfassung

- Dateien
  - Ein- und Ausgabe
- Strings
  - Unicode vs. Byte-Strings
  - Kodierungen
- Nächste Woche
  - reguläre Ausdrücke
  - Ein- und Ausgabe über das Netzwerk