Informatik I

13. Ein-/Ausgabe: String-Literale, String-Interpolation, Dateien, Dateinamen und Ordner, Persistente Daten, Pipes

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

26. November 2013

Informatik I

26. November 2013 — 13. Ein-/Ausgabe: String-Literale, String-Interpolation, Dateien, Dateinamen und Ordner, Persistente Daten, Pipes

Ein-/Ausgabe

- Wir haben bisher Programm-interne Datenstrukturen kennen gelernt.
- Normalerweise wollen wir aber mit der Umgebung kommunizieren
  - mit dem Benutzer an der Konsole (`input` / `print`)
  - mit dem Benutzer über eine GUI (kommt später)
  - mit Dateien, die auf dem Rechner liegen
  - mit Datenbanken auf dem Rechner
  - mit anderen Programmen
  - mit anderen Rechnern (über das Internet)
  - mit anderen Geräten (normalerweise via Programmen/Treibern)
- Heute wollen wir uns einige der Möglichkeiten anschauen.

13.1 Mehr zu String-Literalen

String-Literale

String-Literale können in Python auf viele verschiedene Weisen angegeben werden:

- `"in doppelten Anführungszeichen"`
- `'in einfachen Anführungszeichen'`
- `"""in drei doppelten Anführungszeichen"""`
- `'''in drei einfachen Anführungszeichen'''`
- Jede dieser Varianten mit vorgestelltem ‚`r`', also z.B. `r"in doppelten Anführungszeichen mit r"`.

Einfach und dreifach begrenzte Strings

- Die normale Variante (mit doppelten Anführungszeichen) verhält sich genau so, wie man es aus anderen Programmiersprachen (C, Java) kennt. Man schreibt also zum Beispiel:
  - Zeilenumbruch als `\n` (Newline)
  - Backslashes als `\\`
  - doppelte Anführungszeichen als `\"`
- Bei Strings mit einfachen Anführungszeichen muss man doppelte Anführungszeichen nicht mit Backslash schützen (dafür aber einfache).
- Bei `"""solchen"""` und `'''solchen'''` Strings kann man beide Sorten Anführungszeichen sorglos verwenden, sofern sie nicht dreifach auftreten und die Strings dürfen über mehrere Zeilen gehen.

Rohe Strings

Der `r`-Präfix kennzeichnet einen *rohen* (raw) String:

- Die Regeln für die *Begrenzung* eines rohen Strings sind genauso wie bei normalen Strings: So sind z.B. `r"di\es\ner hie\"r"` und `r'''Die\\ser\\hi''er'''` zwei rohe Strings.
- In einem rohen String finden aber keinerlei Backslash-Ersetzungen statt:

Python-Interpreter

```
>>> print(r"di\es\ner hie\"r")
di\es\ner hie\"r
>>> print(r'''Die\\ser\\hi''er''')
Die\\ser\\hi''er
```

Rohe Strings sind für Fälle gedacht, in denen man viele (wörtliche) Backslashes benötigt. Wichtige Anwendungen: Windows-Pfadnamen und reguläre Ausdrücke.

13.2 String-Interpolation

String-Interpolation: Beispiele

- *String-Interpolation* ist ein Feature, das mit C's *sprintf* verwandt ist. Beispiel:

Python-Interpreter

```
>>> x, y, z = 7, 6, 7 ** 6
>>> print("%s:  %d ** %d = %d" % ('Rechnung:  ',x, y, z))
Rechnung:  7 ** 6 = 117649
```

- Mittlerweile (Python > 3.0) gibt es eine Alternative: die `format`-Methode von Strings.

Python-Interpreter

```
>>> "{} ** {} = {}".format(2,3,8)
'2 ** 3 = 8'
```

Für Details siehe `http://www.python.org/dev/peps/pep-3101/`

String-Interpolation: Erklärung

- String-Interpolation wird vorgenommen, wenn der %-Operator auf einen String angewandt wird. Interpolierte Strings tauchen vor allem im Zusammenhang mit der `print`-Funktion auf.
- Bei der String-Interpolation werden Lücken in einem String durch variable Inhalte ersetzt. Die Lücken werden mit einem Prozentzeichen eingeleitet; zur genauen Syntax kommen wir noch.
- Bei einem Ausdruck der Form `string % ersetzung` muss `ersetzung` ein Tupel sein, das genau so viele Elemente enthält wie `string` Lücken – oder es muss ein Element für die einzige Lücke sein.
- Soll ein Lückentext ein (wörtliches) Prozentzeichen enthalten, notiert man es als %%.

String-Interpolation: `str` und `repr` (1)

- Am häufigsten verwendet man Lücken mit der Notation %s.
  Dabei wird das ersetzte Element so formatiert, wie wenn es mit `print` ausgegeben würde.
  - %s ist also nicht — wie in C — auf Strings beschränkt, sondern funktioniert auch für Zahlen, Listen etc.
- Ein weiterer universeller Lückentyp ist %r.
  Hier wird das ersetzte Element so formatiert, wie wenn es als nackter Ausdruck im Interpreter eingegeben würde.

Diese Buchstaben sind in Analogie zu den Funktionen `str` (lesbare Darstellung) und `repr` (eindeutige und von Python evaluierbare Darstellung) gewählt, die ihr Argument in der entsprechenden Weise in einen String umwandeln.

String-Interpolation: `str` und `repr` (2)

Python-Interpreter

```
>>> string = "dead parrot"
>>> string
'dead parrot'
>>> print(string)
dead parrot
>>> str(string)
'dead parrot'
>>> repr(string)
"'dead parrot'"
>>> print("str:  %s  repr:  %r" % (string, string))
str:  dead parrot repr:  'dead parrot'
>>>
>>> a='a=%r;print(a%%a)';print(a%a)
```

Was tut dieses Prográmmchen in der letzten Zeile?

Exkurs: Wo die kleinen Programme herkommen

- Ein Programm, das sich *selbst repliziert*, nennt man Quine (nach dem amerikanischen Philosophen Willard Van Orman Quine). Ist für alle Programmiersprachen mit genügender Ausdrucksfähigkeit möglich!

Python-Interpreter

```
>>> a='a=%r;print(a%%a)';print(a%a)
# ist das Gleiche wie:
>>> print('a=%r;print(a%%a)' % 'a=%r;print(a%%a)')
# D.h. es soll gedruckt werden:
# a=X;print(a%a)
# wobei:
# X == 'a=%r;print(a%%a)'
# D.h. es wird gedruckt
a='a=%r;print(a%%a)';print(a%a)
```


Mindestbreite und Ausrichtung

- Zwischen Lückenzeichen ‚%' und Formatierungscode (z.B. `s` oder `r`) kann man eine *Feldbreite* angeben:

Python-Interpreter

```
>>> text = "spam"
>>> print("|%10s|" % text)
|      spam|
>>> print("|%-10s|" % text)
|spam      |
>>> width = -7
>>> print("|%*s|" % (width, text))
|spam    |
```

- Bei positiven Feldbreiten wird rechtsbündig, bei negativen Feldbreiten linksbündig ausgerichtet.
- Bei der Angabe * wird die Feldbreite dem Ersetzungstupel entnommen.

String-Interpolation: Andere Lückentypen

Weitere Lückentypen sind für spezielle Formatierungen spezieller Datentypen gedacht. Die beiden wichtigsten in Kürze:

- %d funktioniert für `int`s. Formatierung identisch zu %s. Bei vorgestellter '0' wird mit Nullen aufgefüllt.
- %f funktioniert für beliebige (nicht-komplexe) Zahlen. Die Zahl der Nachkommastellen kann mit *.i* oder .* angegeben werden. Es wird mathematisch gerundet:

Python-Interpreter

```
>>> print("|%0*d|" % (7,42))
|0000042|
>>> zahl = 2.153
>>> print("%f %.1f %.2f" % (zahl, zahl, zahl))
2.153000 2.2 2.15
>>> print("|%*.*f|" % (10, 3, 3.3 ** 3.3))
|    51.416|
```


String-Interpolation: Weitere Bemerkungen

- Ist ein Ersetzungstext zu breit für ein Feld, wird er nicht abgeschnitten, sondern die Breitenangabe wird ignoriert.
- Es gibt noch viele weitere Lückentypen:
  - c: Character/Zeichen (aus String oder `int`)
  - i: Integer (wie d)
  - o: Oktaldarstellung
  - x: Hexadezimal
  - X: Hexadezimal (mit Großbuchstaben)
  - e: Exponentenschreibweise
  - E: Exponentenschreibweise (Großbuchstaben)
  - g: e oder f
  - G: E oder f
- Statt '0', kann man auch ein '+', ' ', '-' oder '#' vorangestellt werden.

`http://docs.python.org/3.3/library/stdtypes.html#old-string-formatting`

String-Interpolation: Weitere Beispiele

Python-Interpreter

```
>>> print("|%c:  %+*i|" % (42,7,42))
|*:  +42|
>>> print("|%c:  %+*i|" % (43,7,-42))
|+:  -42|
>>> zahl = 215345.79
>>> print("|% E %+e % 4.1f|" % (zahl, zahl, zahl))
| 2.153458E+05 +2.153458e+05 215345.8|
>>> zahl = 43983
>>> print("%4x %04X %o %+O" % (zahl, zahl, zahl, zahl))
abcf ABCF 125717 +125717
```


13.3 Dateien

Dateien

- Unsere Programme kranken bisher daran, dass sie kaum mit der Außenwelt kommunizieren können. Um das zu ändern, beschäftigen wir uns jetzt mit Dateien.
- Dateien werden in Python mit `open` geöffnet.

Dateien öffnen: Die open-Funktion

- `open(filename, mode, bufsize)`:
  Öffnet die Datei mit dem Namen `filename` und liefert ein entsprechendes `file`-Objekt zurück.
  `mode` und `bufsize` sind optionale Parameter und haben folgende Bedeutung:
  - `mode` bestimmt, ob die Datei gelesen oder geschrieben werden soll (oder beides). Mögliche Werte werden auf der nächsten Folie beschrieben. Lässt man den Parameter weg, wird die Datei zum Lesen geöffnet.
  - `bufsize` gibt an, ob und wie Zugriffe auf diese Datei gepuffert werden sollen.

Modi für open

open unterstützt u. a. folgende Modi:

- Lesen: `"r"` für Textdateien, `"rb"` für Binärdateien.
- Schreiben: `"w"` bzw. `"wb"`.
  Achtung: Existiert die Datei bereits, wird sie überschrieben (gelöscht).
- Lesen und Schreiben: `"r+"` bzw. `"r+b"` (Für uns nicht relevant).
- Anhängen: `"a"` bzw. `"ab"`.
  Schreibt an das Ende einer (bestehenden) Datei.
  Legt eine neue Datei an, falls erforderlich.

Um mit binären Dateien umzugehen, braucht man neue Datentypen `bytearray` (mutable) und `bytes` (immutable): Sequenzen von Zahlen zwischen 0 und 255.

Dateien schließen

- `f.close()`:
  Schließt eine Datei.
  - Geschlossene Dateien können nicht weiter für Lese- oder Schreibzugriffe verwendet werden.
  - Es ist erlaubt, Dateien mehrfach zu schließen.
  - Es ist normalerweise nicht nötig, Dateien zu schließen, weil dies automatisch geschieht, sobald das entsprechende Objekt nicht mehr benötigt wird.
    Allerdings gibt es alternative Implementationen von Python, bei denen dies nicht der Fall ist. Vollkommen portable Programme sollten also `close` verwenden.

Operationen auf Dateien

- Lesen aus der Datei `f`:
  - `f.read(n)`: Lese *n* Zeichen oder alle Zeichen bis zum Ende der Datei, wenn der Parameter nicht angegeben wurde.
  - `f.readline(limit)`: Lese eine Zeile, aber höchstens *limit* Zeichen, wobei das Zeilenendezeichen erhalten bleibt. Letzte Zeile ist leer!
  - `f.readlines(hint)` Liest alle Zeilen in eine Liste, wobei aber nur so viele Zeilen gelesen werden, dass *hint* Zeichen nicht überschritten werden, falls angegeben.
- Schreiben in die Datei `f`:
  - `f.write(string)`: Hängt einen String an die Datei an (oder überschreibt)

Dateien: Iteration

Zum Einlesen von Dateien verwendet man üblicherweise die Iteration (`for line in f`):

- Über Dateien kann ebenso wie über Sequenzen oder Dictionaries iteriert werden.
- Dabei wird in jedem Schleifendurchlauf eine Zeile aus der Datei gelesen und der Schleifenvariable (hier `line`) zugewiesen, inklusive Newline-Zeichen am Ende (auch unter Windows!).

Dateien: Beispiel zur Iteration

grep_spam.py

```
def grep_spam(filename):
  for line in open(filename):
    if "spam" in line:
      print(line)

grep_spam("spam_sketch.txt")
```


Dateien: Anmerkung zur Iteration

An dieser Stelle lohnt es sich anzumerken, dass viele Funktionen, die wir im Zusammenhang mit Sequenzen besprochen haben, mit *beliebigen* Objekte funktionieren, über die man iterieren kann, also beispielsweise auch mit Dictionaries und Dateien.

- Beispielsweise kann man mit `list(f)` eine Liste mit allen Zeilen einer Datei erzeugen oder mit `max(f)` die lexikographisch größte Zeile bestimmen.
- Es gibt allerdings auch Ausnahmen: `len(f)` funktioniert beispielsweise nicht. Im Zweifelsfall hilft Ausprobieren oder die Dokumentation.

Dateien: Ausgabe

Auch Ausgaben werden selten mit `write` direkt ausgeführt.
Stattdessen verwendet man oft eine erweiterte Form der `print`-Funktion:

- In der Form
  `print(ausdruck1, ausdruck2, ..., file=f)`
  kann `print` benutzt werden, um in eine Datei `f` statt in die Standardausgabe zu schreiben.
- Die Form
  `print(file=f)`
  schreibt eine Leerzeile (genauer: ein Zeilenende) in die Datei `f`.

Nebenbemerkung: Wo wir schon mal bei `print` sind

- Tatsächlich funktioniert `print(..., file=f)` für beliebige Objekte `f`, die über eine `write`-Methode verfügen. Wird kein `f` angegeben, so wird in die Standardausgabe geschrieben.

Ein weiteres Feature von `print` blieb bisher unerwähnt und komplettiert die Beschreibung dieser Funktion:

- Gibt man der `print`-Funktion das Argument `end=" "`, etwa wie in `print("spam", "egg", end=" ")`, dann wird kein Zeilenende erzeugt.
- Stattdessen wird die Ausgabe von nachfolgenden Ausgaben durch ein Leerzeichen getrennt.

Dateien: Beispiel zur Ausgabe

grep_and_save_spam.py

```
def grep_and_save_spam(in_filename, out_filename):
    outfile = open(out_filename, "w")
    for line in open(in_filename):
        if "spam" in line:
            print(line, file=outfile, end='')

grep_and_save_spam("spam_sketch.txt", "spam.txt")
```


13.4 Dateinamen und Ordner

Dateinamen und Ordner

- Dateien (*Files*) sind auf einem Rechner in Ordnern (*Folder* oder *Directories*) zusammengefasst, wobei Ordner auch selbst Bestandteil eines Ordners sein können.
- Um eine bestimmte Datei anzusprechen, kann man einen absoluten Pfadausdruck angeben, eine Kette von Ordnernamen, beginnend beim Wurzelordner gefolgt vom Dateinamen, getrennt durch das Zeichen „/“ (unter Windows „\“: `/Users/nebel/Documents/test.txt`
- Ein Programm befindet sich immer in einem aktuellen Ordner (*current working directory*). Man kann auch relativ dazu eine Datei mit einem relativen Pfadausdruck ansprechen (kein „/“ am Anfang): `../Documents/test.txt`.
- Dabei steht „..“ dafür, eine Ordnerebene hoch zu gehen; „.“ ist der aktuelle Ordner.

Der aktuelle Ordner

- Initial ist der *aktuelle Odner* der, in dem das Skript gestartet wurde. IDLE hat immer einen fixen Ordner.
- Das Modul os enthält Funktionen, um den aktuellen Ordner festzustellen, zu ändern und den absoluten Pfadnamen zu bestimmen.

Python-Interpreter

```
>>> import os
>>> print(os.getcwd()) # gibt aktuellen Ordner
/Users/nebel/Documents
>>> # bestimmt absoluten Pfad
>>> print(os.path.abspath('../memo.txt'))
/Users/nebel/memo.txt
>>> os.chdir('../tmp') # ändert aktuellen Ordner
>>> print(os.getcwd())
/Users/nebel/tmp
```

Windows-Pfadnamen

- Unter Windows werden die Pfadnamensbestandteile nicht durch „/“ durch sondern durch „\“ getrennt.
- Bei der Angabe von Pfadnamen kann man aber problemlos „/“ verwenden.

Python-Interpreter

```
>>> import os
>>> os.getcwd() # gibt aktuellen Ordner
c:\\Python33
>>> os.chdir('Tools/Scripts')
>>> os.getcwd()
c:\\Python33\\Tools\\Scripts
```


Tests

- Es gibt einige os.path-Methoden, mit denen man wichtige Dinge abtesten kann:
  - `os.path.exists(path)` testet, ob unter dem Pfad beschrieben durch `path` eine Datei oder ein Ordner existiert.
  - `os.path.isdir(path)` testet, ob es ein Ordner ist.
  - `os.path.isfile(path)` testet, ob es eine Datei ist.

Python-Interpreter

```
>>> import os
>>> os.path.exists('parrot.txt')
False
>>> f = open('parrot.txt', 'w'); f.write('Dead!\n')
>>> f.close(); os.path.exists('parrot.txt')
True
>>> os.path.isdir('parrot.txt')
False
```


Ordnerliste

- Wir können uns den Inhalt eines Ordners mit `os.listdir(path='.')` anschauen.

Python-Interpreter

```
>>> import os
>>> os.getcwd()
'/Users/nebel/Documents'
>>> os.listdir()
[ '.DS_Store', 'desktop.ini', 'pdfs', 'Processing',
'RECYCLER', 'tex', 'Thumbs.db' ]
>>> os.path.isdir('desktop.ini')
False
>>> os.path.isdir('pdfs')
True
```


Rekursive Dateiliste

- Mit `os.path.join(dir, name)` kann man Pfadbestandteile intelligent zusammen setzen.

`walk_dir.py`

```
def walk(dir):
    for name in os.listdir(dir):
        p = os.path.join(dir, name)
        if os.path.isfile(p):
            print(p)
        else:
            walk(p)
```

13.5 Persistente Daten / Datenbanken

Persistente Daten: Shelves

- Oft sollen Informationen über das Programmende hinaus gerettet werden, z.B. Einstellungen für das Programm.

→ persistente Daten

- Es gibt ein einfaches Modul `shelve`, das die gleiche Basisfuktionalität wie ein Dictionary bietet.
- Die Funktion `shelve.open(filename, flag='c', writeback=False)` öffent solch ein shelf, `flag=`
  - c: Lesen & Schreiben, Kreieren wenn nicht vorhanden
  - w: Lesen & Schreiben
  - r: Lesen
  - n: Neues, leeres Shelf
- `writeback` gibt an, ob jeder zugegriffene Wert zurückgeschrieben werden soll (wenn `True`) oder nur bei Zuweisungen an einen neuen Schlüssel.

Shelve: Beispiel

Python-Interpreter

```
>>> import shelve
>>> sh = shelve.open('addresses.db', 'c')
>>> sh['Cleese'] =['London']
>>> sh['Idle'] = ['Los Angeles']
>>> sh.close()
>>> sh = shelve.open('addresses.db', 'w')
>>> list(sh.items())
[('Idle', ['Los Angeles']), ('Cleese', ['London'])]
>>> sh['Cleese'].append('Berlin')
>>> sh['Cleese']
['London'] # [ da writeback=False]
>>> sh['Cleese'] += ['Berlin']
>>> sh['Cleese']
['London, Berlin']
```


13.6 Pipes

Einfache Kommunikation mit externen Programmen: Pipes

- Um Programme, die in einer *Shell* gestartet werden können, aufzurufen und um ihre Ausgaben zu lesen, kann man Pipes einsetzen (bei Unix-Shell-Kommandos „|“)
- Starte Programm und kommuniziere über die Pipe mit der Standardausgabe.

Python-Interpreter

```
>>> p = os.popen('date')
>>> print(p.read())
Mon Nov 25 21:45:44 CET 2013

>>> print(p.close())
None
```

- Es gibt im Modul `subprocess` die Funktion `subprocess.popen()`, die mehr Kontrolle über den Aufruf gibt.