Vorspann

Simon Clematide
simon.clematide@uzh.ch

Institut für Computerlinguistik
Universität Zürich

Programmiertechniken in die Computerlinguistik I

Übersicht

Objekte
- Typen
- Methoden und Attribute

Zeichen
- Dateikodierung
- Zeichenliterale
- Umkodieren

Regex in Python
- Ersetzen
- Suchen

Lernziele

- Wie versteht man die Python-Hilfe?
- Was sind Objekte? Was sind Typen?
- Was sind Funktionen? Was sind Methoden?
- Welche Rolle spielt die Kodierung des Quellkodes und der zu verarbeitenden Textdateien?
- Wie kann man Zeichenketten notieren (Literale)? Wie kann man Bytefolgen dekodieren? Wie kann man Unicode in UTF-8 enkodieren?
- Anwendung von regulären Ausdrücken in Python zum Suchen und Ersetzen in Zeichenketten mit Nicht-ASCII-Zeichen
- Verstehen der Funktionsweise eines mächtigen regex-basierten Tokenizers

Vorvorspann: Meine Binsenwahrheiten

Warum ist konzeptuelle Klarheit empfehlenswert?

- Solange beim Programmieren alles läuft, wie es soll, ist oberflächliches *How-To*-Wissen genügend.
- Bei Schwierigkeiten musst du verstehen, was du machst!
- Nur so hast du eine Chance, Fehler zu beheben!
- Debugging (Fehler lokalisieren und eliminieren) ist schwieriger als Programme schreiben!

Dos and Don'ts der Datei- und Verzeichnisbenennung

- Benenne Python-Dateien NUR mit Kleinbuchstaben, Unterstrich und Ziffern!
- KEINE Umlaute! KEINE Leerzeichen! KEINE Umlaute!
- KEINE Ordner oder Programme im aktuellen Verzeichnis erstellen, die wie benutzte Python-Module (re.py) heissen!

Python Dokumentation: Erläutertes Beispiel I

Interne Hilfe

```
>>> help(len)
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or collection.
```

Erklärung der Erklärung

- Die Signatur (*signature*) beschreibt die Typen der Funktionsargumente und des resultierenden Funktionswerts
- Bedeutung wird informell erklärt.
- Mehr Info in der Online-Doku `https://docs.python.org/2/library/functions.html?#len`.

Python Dokumentation: Erläutertes Beispiel II

Interne Hilfe ▸1

```
>>> help(re.split)
Help on function split in module re:

split(pattern, string, maxsplit=0, flags=0)
    Split the source string by the occurrences of the pattern,
    returning a list containing the resulting substrings.
```

Erklärung der Erklärung

- *Name*=*Wert* zeigt Standardwert optionaler Argumente.
- Ergebnis- und Argumenttypen werden im Fliesstext informell erklärt.
- Mehr Info in der Online-Doku `http://docs.python.org/2/`.

Ausdrücke evaluieren zu Werten

Taschenrechner können beliebige arithmetische Ausdrücke auswerten

```
sin(0)^2+7*6
```

Python kann beliebige Ausdrücke auswerten

```
>>> len(['a','b','c']) * 14
```

Funktionsaufrufe

Funktionsaufrufe evaluieren zu einem Wert!

Grundlegendes zu Objekten: Typen

Python ist eine objektorientierte Programmiersprache. ▶

Alle Daten (Werte, Datenstrukturen) in Python sind Objekte.

Python ist eine dynamisch getypte Programmiersprache. ▶

Alle Objekte haben einen Typ.

- Warum "dynamisch"? Der Typ einer Variablen wird nicht statisch im Quelltext deklariert, er wird dynamisch zur Laufzeit bestimmt.

Alle Objekte haben einen Typ

Eingebaute Funktion `type()`

Sie bestimmt den Typ von jedem Objekt.

Typen bestimmen

```
>>> type(1)
>>> x = 1+3*42
>>> type(x)
>>> type("ABBA")
>>> type("AB"+'BA')
>>> type("A" == "a")
>>> type(['a','b'])
>>> type(['a','b'][0])
>>> type({})
>>> type(re.search('X','aaa'))
>>> type(None)
>>> type(re.search('a','aaa'))
```

Alle Objekte haben eine kanonische String-Repräsentation

Die eingebaute Funktion `repr()`

Erzeugt eine kanonische Zeichenkette aus jedem Objekt.

```
>>> repr("a")
"'a'"
>>> repr(['a',"b"])
"['a', 'b']"
```

Read-Eval-Print-Loop (REPL)▸2

Im interaktiven Gebrauch wird der zuletzt eingegebene Ausdruck (*expression*) evaluiert und das Resultat als kanonischer String mit `print` ausgegeben.

```
>>> ['a',"b"]
"['a', 'b']"
>>> print repr(['a',"b"])
['a', 'b']
```

Grundlegendes zu Objekten: Methoden

Objekte haben Methoden.

Methoden sind Funktionen, die von einem Objekt aus aufgerufen werden.

Methoden aufrufen

Methoden-Aufrufe (*invocation*) evaluieren zu einem Objekt.
Mit entsprechenden Argumenten!

```
>>> "ABBA".count("B")
2
>>> "ABBA".lower()
'abba'
```

Methoden sind Objekte

Methoden-Namen evaluieren zu einem abstrakten Objekt.

```
>>> "ABBA".count
<built-in method count of
 str object at 0x10f743cc0>
```

Grundlegendes zu Objekten: Attribute

Objekte haben benannte Attribute.

Punktnotation: OBJECT.ATTRIBUT

Attribute sind Objekte.

Methoden = aufrufbare Attribute

Eine Methode ist ein aufrufbares Attribut eines Objekts.

Eine Funktion ist ein aufrufbares Objekt eines Moduls.

Dokumentationsstrings von Funktionen als Attribut `__doc__`

```
>>> len.__doc__
'len(object) -> integer\n\nReturn the number of items of a sequence or colle
>>> help(len)
Help on built-in function len in module __builtin__:
len(...)
    len(object) -> integer
    Return the number of items of a sequence or collection.
```

Zeichenkodierungen und Zeichensätze

Einschränkungen von ASCII

Die weitverbreiteste Zeichenkodierung mit 128 Kodes (7-Bit) unterstützt keine nicht-englischen Buchstaben.

Verschiedene Erweiterungen mit 8-Bit Kodierungen

Zeichenkodierungen mit 256 Kodes (1 Byte) für verschiedene Alphabete oder Betriebssysteme:
ISO-8859-1, ISO-8859-9, Mac-Roman, Windows-1252

Universale Lösung Unicode

Unicode weist jedem Zeichen einen eindeutigen Zahlen-Kode und eine Kategorie zu.
Unicode 6.0 definiert 109'449 graphische Zeichen

Kode-Tabellen von ASCII, ISO-8859-1 und partiell Unicode

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	l	}	~	DEL
8...	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9...	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
A...	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
B...	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C...	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D...	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E...	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F...	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Quelle: Nach http://de.wikipedia.org/wiki/ISO_8859-1

Speicher- und Transportformat UTF-8

Persistente Speicherung und Datenübertragung mit Unicode

UTF (Abk. für *Unicode Transformation Format*) beschreibt Methoden, einen Unicode-Wert auf eine Folge von Bytes abzubilden.

Beispiele für UTF-8-Kodierungen

Zeichen	Unicode	Unicode binär	UTF-8 binär	UTF-8 hexadezimal
Buchstabe *y*	U+0079	00000000 01111001	01111001	0x79
Buchstabe *ä*	U+00E4	00000000 11100100	11000011 10100100	0xC3 0xA4
Eingetragene Marke ®	U+00AE	00000000 10101110	11000010 10101110	0xC2 0xAE
Eurozeichen €	U+20AC	00100000 10101100	11100010 10000010 10101100	0xE2 0x82 0xAC

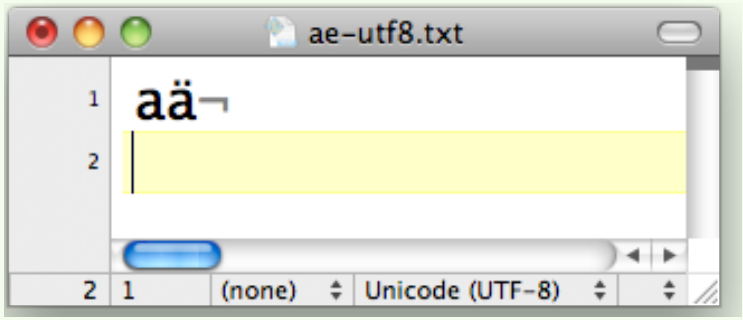
Quelle: `http://de.wikipedia.org/wiki/UTF-8`

Gründe für Format mit variabler Länge: Kompatibilität mit ASCII, kompakte Repräsentation, Sortierbarkeit, Erkennbarkeit von Zeichenanfängen

Textdatei als Bytefolge

Die Repräsentation der Zeichen mit Kodes > 127 sind unterschiedlich.

Datei in UTF-8-Kodierung

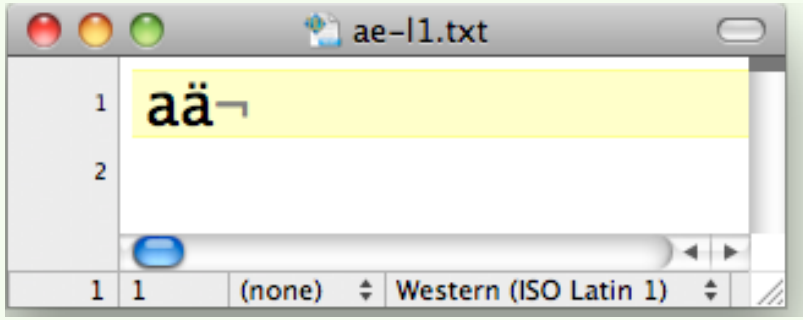


ä = 2 Bytes = C3 A4

```
$ hexdump  ae-utf8.txt
0000000 61 c3 a4 0a
0000004
```

Datei mit 4 Bytes

Datei in Latin-1-Kodierung



ä = 1 Byte = E4

```
$ hexdump  ae-l1.txt
0000000 61 e4 0a
0000003
```

Datei mit 3 Bytes

Zeichen und ihre Zahlenkodes

Zeichenkode berechnen aus Zeichen

```
>>> ord("A")
65
>>> ord('a')
97
>>> ord('\n')
10
>>> ord("\t")
9
>>> ord('\x20') # Hexadezimal
32
>>> ord("\'")
39
```

Zeichen berechnen aus Zeichenkode

```
>>> chr(65)
'A'
>>> chr(97)
'a'
>>> chr(10)
'\n'
>>> chr(9)
'\t'
>>> chr(32)
' '
>>> chr(39)
"'"
```

Datentyp `str`: Folgen von Zeichen als Bytefolgen

Datentyp bestimmen und testen

```
>>> type('ABBA')
<type 'str'>

>>> type("ABBA") == str
True

>>> isinstance('ABBA',str)
True
```

Beispiele immer selber testen!

String-Literale notieren ▸3

```
# Einzeilige (!) Zeichenkette
# mit Escape-Sequenzen
s1 = "a\n\x61"
# Rohe Sequenz r"..."
# ohne Escapes
s2 = r"a\n\x61"

print "Canonical s2: ",repr(s2)
s3 = """a
a"""

print "Canonical s3: ",repr(s3)
a"""

print "Canonical s4: ",repr(s4)
print "Printed s4: ",s4
```

Datentyp unicode: Folgen von Unicodes

Unicode Zeichenkodes

```
>>> ord(u'€')
8364
>>> unichr(8364)
u'\u20ac'
```

Datentyp bestimmen und testen

```
>>> type(u'A')
<type 'unicode'>
>>> type(u'ab') == unicode
True
>>> isinstance('ab',unicode)
False
```

String-Literale notieren ▸4

```
# Einzeilige (!) Zeichenkette
# mit Escape-Sequenzen
s1 = u"\u20aca\n\xe4"

# Rohe Sequenz ur"..."
# \uNNNN werden aufgelöst!
s2 = ur'\u20aca\n\xe4'

# Longstring
s3 = u"""\u20aca

# Roher Longstring
s4 = ur"""€a
\u00e4"""
```

Kodierung der Python-Quellkodes deklarieren

Kodierungskommentar für UTF-8-kodierte Quelltexte

Deklariere Kodierung immer mit Kodierungskommentar, wenn Nicht-ASCII-Zeichen vorkommen!

Datei in UTF-8-Kodierung

▸5 ▸6

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

print "Length of 'a':", len('a'), "Canonical:", repr('a')
print "Length of 'ä':", len('ä'), "Canonical:", repr('ä')
print "Length of u'a':", len(u'a'), "Canonical:", repr(u'a')
print "Length of u'ä':", len(u'ä'), "Canonical:", repr(u'ä')
```

Für Latin-1: `# -*- coding: iso-8859-1 -*-`


iso-8859-1 ist in Python 2 Standard.

Enkodieren und Dekodieren von Zeichenketten

Explizites Dekodieren von UTF-8-Repräsentation

```
>>> text = 'B\xc3\xa4h' # UTF-8-Repräsentation von Bäh
>>> unicodetext = text.decode('utf-8')
```

Es entsteht ein Unicode-Objekt!

Explizites Enkodieren von Unicode-Zeichen als UTF-8-Bytefolge

```
>>> unicode_text = u'Bäh'
>>> utf8_text = unicode_text.encode('utf-8')
```

Es entsteht eine Byte-String!

Das Modul codecs

codecs: Kodieren und Dekodieren

Funktionen für Lesen und Schreiben von Unicode-Strings

Einlesen von Latin-1 und schreiben von UTF-8 ▸7

```
import codecs

# Decode from l1 encoded file into unicode strings
f = codecs.open("./ae-l1.txt", "r", "l1")

# Encode unicode strings into UTF-8 encoded file
g = codecs.open("./AE-l1-encoded-as-utf8.txt", "w", "utf-8")
for line in f:
    g.write(line.upper())
```

Beim Einlesen entstehen Zeichenketten vom Typ `unicode`.

Funktion `re.sub()`: Ersetzen mit regulären Ausdrücken

Globales Ersetzen mit Rückreferenz ▸8

```
import re

text = u"Hässliche Köche verdürben das Gebräu"

pattern = ur"([aeioäöü]+)"

# Im Ersetzungstext können gematchte Gruppen eingefügt werden.
# \N (N ist die N-te gruppierende Klammer im Pattern)
replacement = ur"[\1]"

print re.sub(pattern, replacement, text)
```

`replacement` ist eine Zeichenkette, kein regulärer Ausdruck! Falls nichts gematcht wird, bleibt die Zeichenkette unverändert!

Warum *raw strings* für Reguläre Ausdrücke in Python?

`r'REGEX'` oder `ur'REGEX'`

- Generell empfohlen in `http://docs.python.org/2/library/re.html`.
- Für viele Escape-Sequenzen macht es zwar keinen Unterschied, weil Python-Strings und Reguläre Ausdrücke letztlich dieselben Zeichen bedeuten: `\a,\f,\n,\r,\t,\v`
- Andere Escape-Sequenzen existieren nur in der Regulären Notation und Python lässt den Backslash stehen: `\A,\B,\w,\W,\d,\D,\s,\S`
- Aber andere Reguläre Notationen würden beim Einlesen von Nicht-Raw-Strings missinterpretiert: Um einen einzelnen Backslash zu matchen, müssten wir schreiben: `re.match("\\\\", "\\")`
- `\b`: Bell-Zeichen (ASCII-Code 8) im String; aber Grenze zwischen Wortzeichen und Nicht-Wortzeichen in Regex.
  `re.sub("\\bthe\\b","THE", "Other uses of the")`
- Numerische Rückreferenzen `\1`

Gruppierung mit/ohne Rückreferenzen

Runde Klammern ergeben referenzierbare Gruppierungen

```
>>> text = 'Blick-Leser, A-Post-Fans, andere Bindestrich-Komposita'
>>> re.sub(r'(\w+-)+(\w+)', r'\2', text)
```

Nicht alle Gruppen müssen referenzierbar sein! (Effizienzgründe!)

Nichtreferenzierbare Gruppierung: `(?:REGEX)`

```
>>> text = 'Blick-Leser, A-Post-Fans, andere Bindestrich-Komposita'
>>> re.sub(r'(?:\w+-)+(\w+)', r'\1', text)
```

Funktion `re.findall()`: Globale Suche

Alle nicht-überlappenden Matches extrahieren ▸9

```
import re

text = u"Viele Köche verderben den Brei."
pattern = ur"(\w+)"

# Alle Matches finden
m = re.findall(pattern, text)

for g in m:
    print g
```

Pattern und Text müssen immer vom gleichen String-Typ sein!

Regex-Alternative ist nicht kommutativ!

Die Reihenfolge in einer Regex-Alternative ist nicht beliebig!

```
import re
print re.findall(r'a|aa',"Saal")
print re.findall(r'aa|a',"Saal")
```

Gruppierung mit/ohne Rückreferenzen: `re.findall()`

`re.findall()` und gruppierende Klammern

Unterschiedliche Funktionalität, falls gruppierende Klammern im regulären Ausdruck sind oder nicht!

- **Ohne**: Liste der Matches
- **Mit**: Liste von Tupeln, wobei jedes Tupel-Element den gematchten Inhalt der entsprechenden gruppierenden Klammer enthält.

```
>>> re.findall(r'a(h)|a(a)', "kahler Saal")
[('h', ''), ('', 'a')]
```

Was bedeutet das?

```
    kahler Saal
      |      |
    ('h',    '')  # 1. möglicher Match
    ('',    'a')  # 2. möglicher Match
```

Unicode Flag: Was ist ein Wortzeichen?

Unicode-Kategorien aktivieren ▸10

```
import re

text = u"Viele Köche verderben den Brei."
pattern = ur"(\w+)"

# Das Flag (?u) aktiviert Unicode-Kategorien fuer \w und \b
pattern = ur"(?u)(\w+)"

# Resultat ist eine Liste
m = re.findall(pattern, text)

for s in m:
    print s
```

Das Unicode-Flag `(?u)` zählt nicht als Gruppe wie alle `(?...)`.

Lesbare und kommentierte reguläre Ausdrücke

Was matcht dieser Ausdruck?

```
(?:[A-Z]\.)+|\w+(?:-\w+)*|\$?\d+[.\d]*%?|\.\.\.|[.,;?]+|\S+
```

Lesbare und kommentierbare Ausdrücke dank Flag (?x) ▸11

```
import re
text = "That U.S.A. poster-print costs $12.40..."
pattern = r'''(?x)          # set flag (?x) to allow verbose regexps
     (?:[A-Z]\.)+           # abbreviations, e.g. U.S.A.
   | \$?\d+(?:[.,]\d+)*%?   # currency/percentages, $12.40, 82%
   | \w+(?:-\w+)*           # words with optional internal hyphens
   | \.\.\.                 # ellipsis
   | [.,;?]+                # punctuation
   | \S+                    # catch-all for non-layout characters
   '''
m = re.findall(pattern,text)
print m
```

Kontrollfragen

- Was passiert und was entsteht, wenn ein Ausdruck wie `type(2 == 1*1+1)` evaluiert wird? ▶
- Können alle Ausdrücke evaluiert werden?
- Was ist der Unterschied zwischen der Notation von String-Literalen mit `'...'` oder `"..."`?
- Worin unterscheiden sich `r'...'` und `'...'`?
- Worin unterscheiden sich `"""..."""` und `"..."`?
- Wie kann man Reguläre Ausdrücke lesbar schreiben?
- Wie weitet man Reguläre Notationen wie `\w` von reinem ASCII auf alle alphanumerischen Zeichen von Unicode aus?
- Wie kann man in einer Zeichenkette alle mit einem Regulären Ausdruck gematchten Teilzeichenketten ersetzen?

Liste der verlinkten Programme und Ressourcen I

Folie

▸1 Online-Dokumentation: http://docs.python.org/library/re.html?#re.split 6

▸2 http://en.wikipedia.org/wiki/Read-eval-print_loop 10

▸3 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/str_literals.py 18

▸4 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/unicode_literals.py 19

▸5 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/str_representation_utf8.py 20

▸6 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/str_representation_l1.py 20

▸7 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/codecs_open_files.py 22

▸8 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/re_sub.py 23

▸9 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/re_findall_flag_u.py 26

▸10 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/re_findall_flag_u.py 29

▸11 Programm: http://www.cl.uzh.ch/siclemat/lehre/hs15/pcl1/lst/vorspann/re_findall_tokenizer.py 30