Python 2

Vorlesung “Computerlinguistische Techniken”

Alexander Koller

28. Oktober 2014

Listen

- Sequenz von beliebigen Werten.
- Literale: `[]`, `[1,2,3]`, `["hallo", True, -5.0]`
- Hinten an Liste anhängen: `L.append(27)`
- Länge ist `len(L)`

Slices

- Zugriff auf Teile:

  `L[2]`: drittes Element der Liste (0 ist erstes)

  ```
  L[2:5] = [L[2],...,L[4]]
  L[2:] = L[2:len(L)]
  L[:2] = L[0:2]
  ```

- Man kann Elemente und Slices überschreiben (auch verkürzen und verlängern):

  ```
  L = [1,2,3]   L[1] = 7 => L == [1,7,3]
  L = [1,2,3]   L[0:1] = [5] => L == [5,2,3]
  L = [1,2,3]   L[2:] = [] => L == [1,2]
  ```

Mengen

- Wie mathematische Menge: ungeordnet, jedes Element kommt nur einmal vor

- Erzeugen mit `set()` oder `set([2,3,4])`

- Erweitern mit `s.add("hallo")`

- Mitgliedschaft (effizient) testen mit `2 in s`

Dictionaries

- Dictionary (Wörterbuch) besteht aus einer Menge von Schlüsseln, denen Werte zugewiesen werden.

- Literale: `{“hallo”:3, “welt”:False}`, `{}`

- Hinzufügen mit `dict[“foo”] = 5;`
  Auslesen mit `dict[“foo”];`
  Test, ob Schlüssel existiert, mit `“foo” in dict;`
  Schlüsselmenge mit `dict.keys()`

Anweisungen

- Mit einer Anweisung bringt man Computer dazu, eine bestimmte Aktion auszuführen.
  - Zuweisung ist Anweisung: Wert einer Variable ändern
  - `print(x)`: Zeige Wert des Ausdrucks x an
  - `sys.exit()`: Programm beenden
- Anweisungen sind das zentrale Werkzeug beim imperativen Programmieren.

Kontrollstrukturen

- Ein Programm ist eine Sequenz von Anweisungen.
- In den meisten Programmen sollen bestimmte Anweisungen nur manchmal bzw. mehrmals ausgeführt werden.
- Das erreicht man mit *Kontrollstrukturen*.

If

- Mit der if-Kontrollstruktur wird ein Programmblock nur ausgeführt, wenn ein Ausdruck (die Bedingung) zu `True` evaluiert:

```
if len(L) > 0:
    print L[0]
```

- Wenn die Bedingung zu `False` evaluiert, wird der ganze if-Block übersprungen.

Einrücken

- In Python ist es wichtig, dass Anweisungen innerhalb einer Kontrollstruktur eingerückt werden.

- Beachte die Leerzeichen vor `print L[0]`!

- Wenn Bedingung zu `False` evaluiert, überspringt if genau die eingerückten Zeilen.

If-Else

- Man kann einen else-Block angeben, der ausgeführt wird, wenn die Bedingung zu `False` evaluiert:

```
if len(L) > 0:
    print L[0]
else:
print “Liste ist leer”
```

While

- Manchmal möchte man eine Anweisung mehrfach ausführen, z.B. so lange, bis eine Bedingung `False` wird:

```
while len(L) > 0:
    print L[0]
    L = L[1:]
```

(Dieses Beispiel ist eine schrecklich ineffiziente Art, um eine Liste auszugeben!)

For

- Alternativ kann man auch einen Block von Anweisungen für jedes Element einer Liste oder Menge ausführen:

```
for x in L:
    print x
```

- x kann wie eine Variable verwendet werden.
  Es macht aber normalerweise keinen Sinn, x etwas zuzuweisen.

For mit Zahlen

- Eine häufige Verwendung von for ist es, Anweisungen für alle Zahlen von 0 bis n-1 auszuführen.

- Dazu verwendet man `range(n)`: gibt Liste [0,...,n-1] zurück.

```
sum = 0
for i in range(n):
    sum = sum + i
```

Schleifen

- Anweisungen, die mit den Konstrukten for und while aufgebaut werden, heißen *Schleifen*.

- In imperativen Sprachen kann Rekursion durch Iteration (= Verwendung von Schleifen) ersetzt werden.

- Wahl zwischen Rekursion und Iteration hängt von Lesbarkeit und Effizienz ab.

Funktionen

- Blöcke von Anweisungen, die immer wieder verwendet werden, kann man in *Funktionen* zusammenfassen:

```
def printxpluszwei(zahl):
    output = zahl+2
    print(output)

>>> printxpluszwei(5)
7
```

Funktionen

- Eine Funktion kann 0, 1 oder mehrere Parameter haben:
  - `def keineparameter(): ...`
  - `def einparameter(x): ...`
  - `def zweiparameter(x, y): ...`

- Innerhalb der Funktion kann ein Parameter wie eine Variable verwendet werden.

- Wert des Parameters: durch Aufruf bestimmt.

Funktionen

- Funktionen können mit einer `return`-Anweisung Werte zurückgeben. Dieser Wert wird dann der Wert des Funktions-Ausdrucks.

```
def plus(x,y):
    return x + y
```

```
>>> print plus(3,4)
7
```

Rekursion

- Auch in Python dürfen Funktionen rekursiv sein, d.h. sich selbst aufrufen.

```
def ggT(klein, gross):
    if klein == 0:
        return gross
    else:
        rest = gross % klein
        return ggT(rest, klein)
```

- Aber Rekursion viel seltener als in Prolog: häufig stattdessen Schleifen.

Imperatives Programmieren

- Grundlegende Struktureinheit ist die Anweisung.
  - ggf. komplex (Kontrollstrukturen)
  - häufig verwendete Anweisungen in Funktionen
  - Anweisungen operieren auf Datenstrukturen
- Wird für große Programme mit vielen Datenstrukturen leicht unübersichtlich; Bugs können weit entfernt Effekte haben.

Objektorientiertes Programmieren

- Grundlegende Struktureinheit ist die *Klasse*.
  - fasst Daten (in *Feldern*, ≈ Variablen) und *Methoden* (≈ Funktionen), die mit diesen Daten arbeiten, zusammen

- Alle Daten und Anweisungen, die logisch zusammengehören, stehen im Programm beieinander.

- Klasse verspricht bestimmtes Verhalten nach außen; Implementierungsdetails von außen nicht zu sehen.

Klassen und Objekte

- Klassen sind Datentypen, definieren also eine Menge von Werten und die Operationen, die man mit ihnen ausführen kann.

- Objekte (oder Instanzen) sind die einzelnen Werte dieses Datentyps.

Eingebaute Datentypen

- Eingebaute Datentypen sind auch Klassen:

```
>>> type(“hallo welt”)
<type ‘str’>
>>> type (27)
<type ‘int’>
```

- Man kann auf ihren Werten die Methoden der Klasse ausführen:

```
>>> “   viele spaces drumrum    “.strip()
‘viele spaces drumrum’
```

Methoden

- Methoden sind Funktionen, die in einer Klasse definiert werden.

- Wenn o ein Objekt der Klasse K ist und K die Methode m definiert, kann man Methode mit o.m() auf o ausführen.

- Methoden dürfen Parameter haben:

```
>>> ":".join(["hallo", "welt"])
'hallo:welt'
```

Objekte erzeugen

- Man kann Klassen instanziieren und so neue Objekte der Klasse erzeugen.

- Jedes Objekt hat sein eigenes Exemplar der Felder; deshalb können verschiedene Objekte verschiedene Werte haben.

- Z.B. Klasse set:

```
>>> x = set()
>>> y = set([1,2,3])
>>> x.issubset(y)
True
```

Eigene Klassen schreiben

- Sie können eigene Klassen mit dem Schlüsselwort `class` definieren:

```
class MeineKlasse1:
    def printplus(self, x):
        print x + 2
```

```
>>> a = MeineKlasse1()
>>> a.printplus(5)
7
```

self

- In den Methoden einer Klasse (und nur dort) kann man auf das aktuelle Objekt (und seine Felder und Methoden) mit `self` zugreifen.
  - Felder: `self.feldname`
  - Methoden: `self.methode(arg1, ..., argn)`
- In Python hat jede normale Methode einer Klasse `self` als ersten Parameter.

self: Ein Beispiel

```
class MeineKlasse2:
    def set_addthis(self, addthis):
        self.addthis = addthis

    def printplus(self, x):
        output = x + self.addthis
        print output

>>> a = MeineKlasse2()
>>> a.set_addthis(3)
>>> a.addthis
3
>>> a.printplus(5)
8
```

Objekte initialisieren

- Die spezielle Methode `__init__` wird jedesmal aufgerufen, wenn ein neues Objekt erzeugt wird.

```
class MeineKlasse3:
    def __init__(self, addthis):
        self.addthis = addthis

    def printplus(self, x):
        print x + self.addthis

>>> x = MeineKlasse3(4)
>>> x.printplus(7)
11
```

OOP: Zusammenfassung

- Klassen, Objekte, Methoden, Felder

- Datentypen = Klassen, und wir können neue Klassen definieren.

- Hier: in erster Linie existierende Klassen verwenden, aber manchmal ist es nützlich, eigene Klassen zu bauen.

- Es gibt noch viel zu OOP zu sagen, siehe Literatur.

Arbeit mit Dateien

- Die Klasse `file` erlaubt es uns, Dateien zu lesen und zu schreiben.

- Datei lesen am einfachsten so:

```
with open("dateiname") as f:
    print f.read()
```

- Man kann aus einer Datei nur einmal von vorne bis hinten lesen; danach muss man sie schließen und nochmal neu öffnen.

Lesen aus Dateien

- Einige wichtige Methoden:
  - f.read(): gibt den kompletten Inhalt der Datei in einem einzigen String zurück
  - f.readlines(): gibt Liste von Strings zurück, einer für jede Textzeile der Datei
  - for line in f: iteriert über die einzelnen Zeilen einer Datei.

Schreiben in Dateien

- Zum Schreiben Datei mit “w” öffnen:

```
with open(“dateiname”, “w”) as f:
    f.write(“erste zeile\n”)
    f.write(“zweite zeile\n”)
```

  - write schreibt genau das in f, was man ihm sagt, ggf. alles in die gleiche Zeile. Zeilenumbruch mit dem Zeichen \n.

Die Kommandozeile

- Wenn man Python mit `python programm.py` von Kommandozeile aus aufruft, kann man dem Programm Argumente mitgeben.

- Also z.B. Aufruf:

```
koller$ python add.py 5 6
11
```

Zugriff auf Kommandozeile

- Der Programmname und die Argumente stehen in der Liste `sys.argv`:

```
import sys
print sys.argv

koller$ python test.py 1 2 3 4
["test.py", "1", "2", "3", "4"]
```

- Auf Einträge in der Liste kann man ganz normal zugreifen: `sys.argv[1]` usw.

Formatierte Ausgabe

- Python erlaubt es, Werte ordentlich in einen String einzutragen.

```
>>> “arg1={0}, arg2={1}”.format(27, “foo”)
‘arg1=27, arg2=foo’
```

- Zwischen { } steht die Nummer des Wertes. Gleiche Nummer darf mehrmals vorkommen (“gib dem {0}, was des {0}s ist”)

Fortgeschrittenes format()

- Man kann die Breite des Feldes angeben und darin ausrichten mit `{0:<10}`, `{0:>10}`, `{0:^10}`

- In den Klammern können Ausdrücke stehen:
  `“x={0[0]}, y={0[1]}”.format([5,3])`

- Oder Schlüssel von Dictionaries:
  ```
  dict = {“breite”:52, “laenge”:12}
  “b={breite}, l={laenge}”.format(**dict)
  ```

Module

- Wir haben jetzt schon öfter das `import`-Schlüsselwort gesehen
  - `import sys`
  - `from operator import itemgetter`

- Dieses Schlüsselwort macht Namen aus anderen *Modulen* sichtbar.

- Ganz wichtig zur Strukturieren großer Programme und Verwendung von Bibliotheken wie NLTK.

Module

- Ein Modul ist ein Stück Python-Code, der in einer einzigen Datei steht.

- Modulname `abcd` enthält allen Code aus der Datei `abcd.py`.
  - nach `abcd.py` wird z.B. im aktuellen Verzeichnis gesucht
  - oder im Verzeichnis der Python-Standardbibliothek
  - wenn Python nur `abcd.pyc` (= kompiliertes `abcd.py`) findet, ist es auch zufrieden

Laden eines Moduls

- Die Anweisung `import abcd` lädt das Modul `abcd`.

- Beim ersten Laden des Moduls werden alle Anweisungen darin ausgeführt, incl. Deklaration von Funktionen und Klassen.

- Wenn eine Klasse im Modul `K` heißt, dann heißt sie dort, wo importiert wird, `abcd.K`. Ebenso für Funktionen.

from ... import

- Mit `from abcd import` kann man auf die Namen aus dem Modul `abcd` direkt zugreifen.

- "`from abcd import K`": Klasse K ist direkt verfügbar (und nicht nur als abcd.K).

- "`from abcd import *`": alle Namen aus abcd sind direkt verfügbar.

Python-Standardbibliothek

- Python wird mit einer großen Menge Klassen in verschiedenen Modulen (der Standardbibliothek) ausgeliefert.

- Gewöhnen Sie sich an, Problemlösungen in der Standardbibliothek nachzuschlagen.

- http://docs.python.org/library/
  (achten Sie auf korrekte Python-Version)

NLTK

- Neben der Standard-Bibliothek gibt es viele andere Bibliotheken für Python, die man auf seinem Computer installieren kann.

- NLTK (Natural Language Toolkit, Bird/Klein/Loper): Eine spezielle Bibliothek für Computerlinguistik.

- http://www.nltk.org/

NLTK

- Wir werden NLTK im Rest der Vorlesung verwenden.
  - Klassen wie `nltk.grammar.ContextFreeGrammar`
  - einfacher Zugriff auf Korpora: `nltk.corpus.brown.words()`

- Sie sollten NLTK herunterladen und bei sich installieren: http://nltk.org/install.html

- und sich mal die Doku anschauen: http://nltk.org/api/nltk.html

Zusammenfassung

- Objektorientierte Programmierung
- Python und seine Umwelt:
  Dateien, formatierte Strings, Kommandozeile
- Module und Bibliotheken
  - Standardbibliothek
  - NLTK