Python-Kurs 2010

Dipl.-Wirt.-Inf. Patrick Holz

RRZK

Universität zu Köln

patrick.holz@uni-koeln.de

Organisatorisches

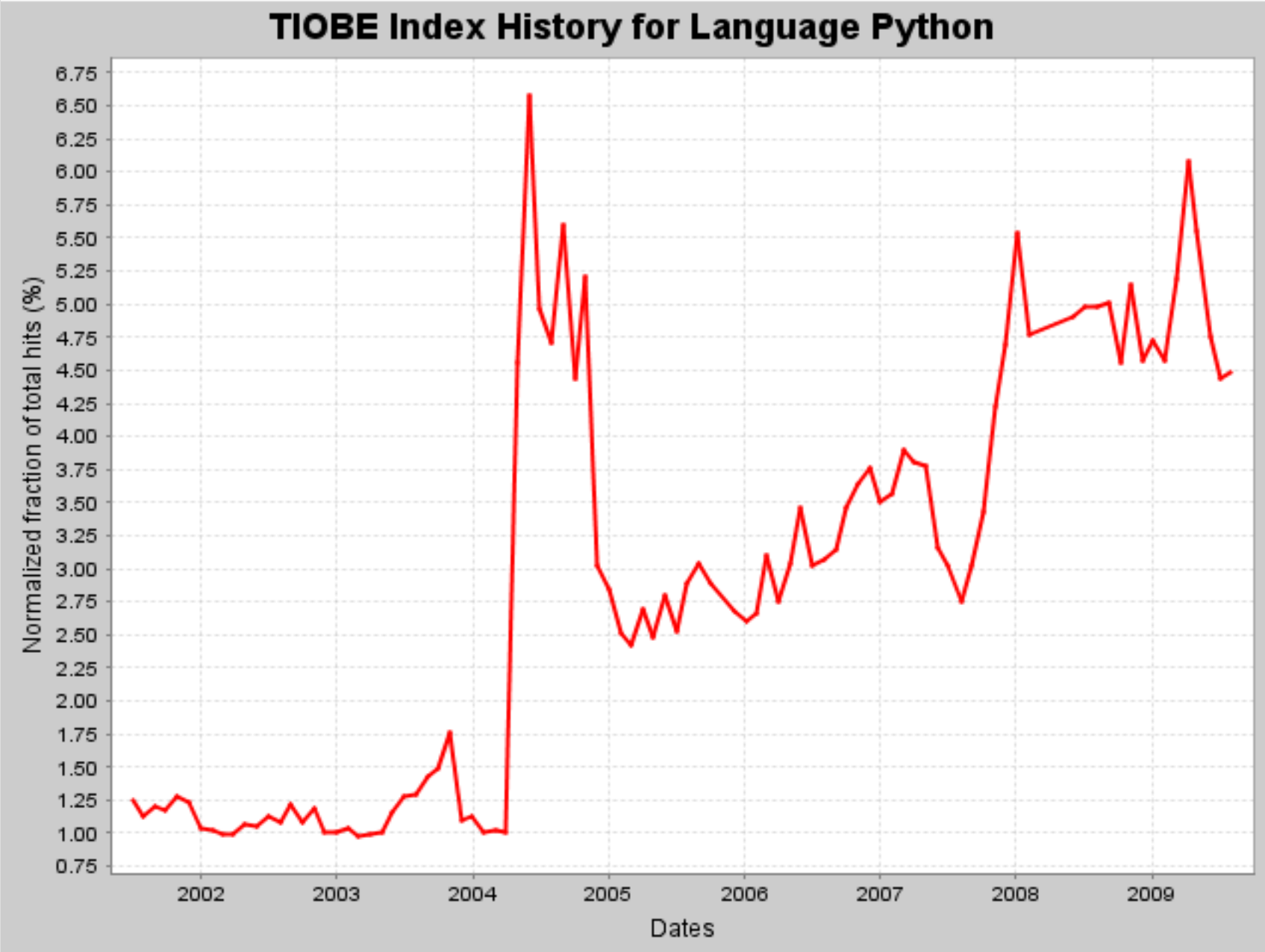
- 4 * 1,5 Stunden – 15. - 18. März 2010, jeweils 15.15 – 16.45 Uhr
- keine Übungen
- Unterlagen:

  http://www.pytheway.de/python.pdf
- Hinweis: Python ist auf den Servern des ZAIK/RRZK installiert (z.Z. in der Version 2.4.3)
- Literatur: RRZN-Handbuch „Python“, erhältlich im RRZK-B

Entstehung von Python

- erste Python-Version von Guido van Rossum 1991 entwickelt
- Name leitet sich von „Monty Python“ ab
- Kombination von C, ABC und Modula-3, ergänzt um Objektorientierung
- wesentliche Ziele: Einfachheit, Übersichtlichkeit, Objektorientierung, RAD (Rapid Application Development)
- heutzutage vor allem eine Alternative zu Perl und TCL (wird also zu den Skript- bzw. Interpretersprachen gezählt)

Position Aug 2009	Position Aug 2008	Delta in Position	Programming Language	Ratings Aug 2009	Delta Aug 2008	Status
1	1	=	Java	19.527%	-2.04%	A
2	2	=	C	17.220%	+1.04%	A
3	4	↑	C++	10.501%	+0.44%	A
4	5	↑	PHP	9.390%	+0.04%	A
5	3	↓↓	(Visual) Basic	8.486%	-2.37%	A
6	6	=	Python	4.489%	-0.49%	A
7	8	↑	C#	4.443%	+0.75%	A
8	7	↓	Perl	4.028%	-0.67%	A
9	10	↑	JavaScript	2.812%	-0.08%	A
10	9	↓	Ruby	2.490%	-0.43%	A
11	11	=	Delphi	2.337%	-0.39%	A
12	13	↑	PL/SQL	0.982%	+0.30%	A
13	14	↑	SAS	0.817%	+0.27%	A
14	27	↑↑↑↑↑↑↑↑↑↑↑↑↑	RPG (OS/400)	0.752%	+0.52%	A
15	26	↑↑↑↑↑↑↑↑↑↑↑	ABAP	0.739%	+0.51%	A
16	16	=	Pascal	0.675%	+0.26%	A-
17	12	↓↓↓↓↓	D	0.662%	-0.69%	B
18	17	↓	Lisp/Scheme	0.630%	+0.25%	B
19	41	↑↑↑↑↑↑↑↑↑↑↑	Objective-C	0.612%	+0.51%	B
20	25	↑↑↑↑↑	MATLAB	0.560%	+0.32%	B


TIOBE Index History for Language Python
Normalized fraction of total hits (%)
6.75
6.50
6.25
6.00
5.75
5.50
5.25
5.00
4.75
4.50
4.25
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
2002
2003
2004
2005
2006
2007
2008
2009
Dates

Warum Python?

- „einfach“, mächtig, modular, plattformunabhängig
- steht unter der GPL und ist kostenlos verfügbar
- „batteries included“ - viele Module bereits enthalten
- objektorientiert (möglich, nicht zwingend)
- „zwingt“ zu ordentlicher Programmierung (PEP-8)
- CGI- und Datenbank-Anbindungen verfügbar
- (mit C/C++-, Perl- und Java-Bibliotheken erweiterbar)

Installation / Aufruf

- bei den meisten Linux-Distributionen integriert
- Quelle für fast alle Systeme (v.a. auch Windows):

  http://www.python.org
- Aufruf auf der Kommandozeile:

  python => Python-Interpreter

  python skriptname.py => Programm „skriptname.py“ starten
- Testen z.B. mit „Taschenrechner“-Funktionen
- Interpreter beenden mit „Strg-D“ (Unix) bzw. „Strg-Z“ (Win)

Editoren / IDEs

- die üblichen Verdächtigen (vi, Emacs, TextPad, jEdit, Geany)
- spezielle Python-IDEs:
  - IDLE
  - Stani's Python Editor (SPE)
  - eric
  - DrPython
  - SciTE
  - EasyEclipse for Python

HelloWorld (es muss ja sein...)

- Shebang-Zeile:

  #!/usr/bin/python (nur sinnvoll unter Unix/Linux)

- Magic Line / Encoding:

  # -*- coding: utf-8 -*-

- Ausgabe einer Zeichenkette (String): *30

  print(„Hello World!“)

- Abfrage einer Eingabe:

  eingabe = raw_input(„Bitte geben Sie etwas ein: “)

  print(eingabe)

Variablen

- auf der vorherigen Folie war „eingabe“ eine Variable
- Variablen bilden einen „Platzhalter“ für Werte, die sich während der Programmausführung ändern können
- zulässige Zeichen: Buchstaben, Zahlen, Unterstrich
- dürfen keine Schlüsselnamen sein (z.B. „print“)
- zu beachten: Namensräume und Gültigkeiten (kommt später...)
- Zuweisungen immer mit „=“
- im Gegensatz zu vielen anderen Sprachen ist in Python keine Deklaration von Variablen erforderlich

einfache Datentypen

- Zahlen:
  - Integer (Ganzzahlen von -2^{31} bis $+2^{31}$)
  - Long Integer (Ganzzahlen, Größe durch Rechnerspeicher begrenzt)
  - Float (Fließkommazahlen), z.B. 3.5 (und nicht 3,5!)
- Strings: *30
  - „normale“ Strings (Zeichenketten mit ASCII-Zeichen): „Hallo Welt“
  - Unicode-Strings (Zeichenketten mit Unicode-Zeichen): u“Hallo Welt“
  - Raw-Strings (Zeichenketten ohne Escape-Sequenzen): r“Hallo Welt“

Stringtypen

- ASCII vs. Unicode (sehr vereinfacht): *30
  - ASCII beinhaltet nur 128 Zeichen (keine Umlaute etc.)
  - Unicode enthält (theoretisch) etwa 65000 Zeichen
- Escape-Sequenzen:
  - vorangestellter Backslash bei nicht darstellbaren Zeichen oder Zeichen mit Sonderfunktion
  - Bsp. für ersteres: „\n“ (Return) „\t“ (Tabulator) „\b“ (Backspace)
  - Bsp. für letzteres: „\““ (Anführungszeichen) „\\“ (Backslash)
  - Raw-Strings ignorieren Escape-Sequenzen (größtenteils...)

Operationen einfacher Datentypen

- einfache Rechenoperationen mit Zahlen: *30 + - * / % **
- int(s): wandelt einen String (wenn möglich) in einen Integer um
- analog dazu: long(s) float(s)
- cmp(x,y): vergleicht x mit y und gibt 1 (x>y), 0 (x=y) oder -1 (x<y) aus
- Strings verknüpfen: s1 + s2
- unicode(s): *30 wandelt einen String in einen Unicode-String um
- len(s): gibt die Länge eines Strings zurück
- str(i): wandelt ein Objekt (z.B. eine Zahl) in einen String um

String-Methoden (I)

- Strings sind Objekte (Begriff wird später erläutert) und besitzen daher eingebaute Methoden; Beispiele:
- s.encode wandelt Unicode-Strings (möglichst) in normale Strings um
- s.count(t) gibt die Anzahl der Substrings t in s zurück
- s.find(t) gibt den Index des ersten Auftretens von t in s zurück
- s.isalnum() gibt 1 zurück, wenn s nur alphanumerische Zeichen enthält
- s.isalpha() gibt 1 zurück, wenn s nur Buchstaben enthält
- s.isdigit() gibt 1 zurück, wenn s nur Zahlen enthält
- s.replace(t,u) ersetzt in s alle Vorkommen von t durch u

String-Methoden (II)

- s.strip() entfernt alle Whitespaces (Leerzeichen, Tabulatoren, Zeilenumbrüche, ...) am Anfang und am Ende von s
- s.split(d) gibt eine Liste mit in s durch d getrennten Substrings zurück
- s.lower() ersetzt alle Großbuchstaben in s durch Kleinbuchstaben
- s.upper() ersetzt alle Kleinbuchstaben in s durch Großbuchstaben
- weitere Methoden finden sich in der Python Library Reference:
  http://docs.python.org/lib/string-methods.html
  (allgemein http://docs.python.org/lib/)

Datentypen „None“ und „Boolean“

- None ist ein besonderer Datentyp, sein Wert ist schlicht „nichts“ (dient z.B. zur Prüfung, ob eine Variable einen Wert besitzt
- Boolean ist ein Datentyp, der nur zwei Zustände kennt:
  - „falsch“: entspricht der Zahl 0, einem leeren String, Liste, Tuple oder Dictionary (kommt noch), dem Wert „False“ oder dem Datentyp „None“
  - „wahr“: entspricht allem anderen

Komplexe Datentypen (I)

- Listen sind zusammenhängende Ansammlungen anderer Datentypen:
  - Liste mit Zahlen: zahlenliste = [1,2,3]
  - Liste mit Strings: stringliste = [„Hund“, „Katze“, „Maus“]
  - gemischte Liste: mixliste = [1, „Tiger“, 45, -3, „Huhn“]
  - Liste mit Listen: doppelliste = [[1,2,3],[„a“, „b“, „c“]]
- Tupel sind unveränderliche Listen (d.h., die Werte können nicht mehr nachträglich angepasst werden):
  - Tupel-Beispiel: meintupel = (1,2,3)

Komplexe Datentypen (II)

- Dictionaries, auch „assoziative Listen“ genannt, bestehen immer aus Schlüssel-Wert-Paaren beliebiger anderer Datentypen:
  - Dictionary-Beispiel: ich = {'vorname':'patrick', 'nachname':'holz'}
- Zugriff auf die Elemente komplexer Datentypen erfolgt per Index oder Slice:

Typ	Index	Slice
Array/Tuple	myarray[1]	myarray[1:3]
Dictionary	ich[„vorname“]	---

- Bemerkungen:
  - Zählung beginnt bei „0“ oder von hinten bei „-1“
  - Slices können auch eine oder keine Grenze haben, z.B. [1:] oder [:]

Listen-Methoden

- a.append(x) Element x zu a hinzufügen (ans Ende)
- a.insert(i,x) Element x zu a hinzufügen (an Stelle i)
- a.index(x) gibt den Index von Element x in a zurück
- a.remove(x) löscht x aus a
- a.pop() gibt das letzte Element von a aus und löscht es
- a.sort() sortiert a lexikographisch
- a.reverse() dreht die Reihenfolge der Elemente in a um

Listen-Funktionen

- max(a) gibt das (lexikographisch) größte Element von a zurück
- min(a) gibt das (lexikographisch) kleinste Element von a zurück
- len(a) gibt die Anzahl der Elemente in a zurück
- del a[x(:y)] löscht in das Element an der Stelle x bzw. die Elemente an den Stellen x bis y
- Sonderfunktion range:
  - range(start,end,step) gibt eine Liste mit Zahlen von start bis end-1 in step Schritten zurück
  - Beispiel: range(3,13,3) ergibt [3,6,9,12]
  - standardmäßig gilt start=0 und step=1: range(4) ergibt [0,1,2,3]

Dictionaries: Methoden und Funktionen

- Methoden: *30
  - d.items() gibt d als Liste mit Schlüssel-Werte-Tuples zurück
  - d.keys() gibt die Schlüssel von d als Liste zurück
  - d.values() gibt die Werte von d als Liste zurück
  - d.has_key(k) wahr, wenn Schlüssel k in d vorkommt, sonst falsch
- Funktionen:
  - len(d) gibt die Anzahl der Paare in d zurück
  - del d[k] löscht das Element mit dem Schlüssel k aus d

Sonstiges zu Datentypen

- Datentyp einer Variablen herausfinden mit type(var)
- Strings, Listen, Tupel und Dictionaries fasst man häufig unter dem Begriff „Sammlungen“ oder „Container“ zusammen
- „in“-Operator dient zur Kontrolle, ob ein Element in einem Container vorkommt

Kontrollstrukturen: Verzweigung (I)

- bei der Verzweigung wird auf eine Bedingung getestet und anschließend entweder ein bestimmter oder kein Anweisungsblock ausgeführt (elif und else sind optional)
- Schema:
  - if *bedingung1*:
    anweisung1
    anweisung2
    elif *bedingung2*:
    anweisung3
    else:
    anweisung4

Kontrollstrukturen: Verzweigung (II)

- Abgrenzung der Blöcke erfolgt durch Einrücken (i.d.R. Tabulatoren)
- Beispiele für Bedingungen:
  - Vergleichsoperatoren: `<` `>` `<=` `>=` `==` `!=`
  - Aussagenlogik:
    - not x wahr, wenn x falsch ist, sonst falsch
    - x and y y, wenn x wahr ist, sonst x
    - x or y y, wenn x falsch ist, sonst x
  - in-Operator (z.B.: if a in b: ...)

Kontrollstrukturen: while-Schleife

- bei der while-Schleife wird ein Anweisungsblock so lange ausgeführt, bis eine Abbruchbedingung erfüllt ist (else ist optional möglich)
- Schema:
  - while *bedingung1*:

    anweisung1

    anweisung2

    else:

    anweisung3
- „break“-Anweisung bricht aus einer Schleife aus und beendet sie
- „continue“-Anweisung beendet den aktuellen Durchlauf der Schleife

Kontrollstrukturen: for-Schleife

- bei der for-Schleife wird ein Anweisungsblock gemäß einer zuvor festgelegten Anzahl an Iterationen ausgeführt
- Schema:
  - for *variable* in *sammlung*:

    anweisung1

    anweisung2
- break, continue und else sind möglich
- häufig wird für die Sammlung die range-Funktion verwendet

Funktionen: Definition

- mehrfach benutzte Befehlsfolgen lagert man i.d.R. in Funktionen aus
- Definition einer Funktion:

```
def funktionsname(argument1,argument2,...):
    „docstring“
    anweisung1
    anweisung2
```

- Der Docstring enthält eine kurze Beschreibung der Funktion
- Argumente werden der Funktion übergeben (z.B. „gebe_aus('Hallo')“)
- Rückgabe eines Wertes erfolgt mit der „return“-Anweisung

Funktionen: Beispiel

- Beispiel: Funktion zur Berechnung der n-ten Fibonacci-Zahl

```
def fib(n):
    „Returns the n-th Fibonacci number“
    if n < 0:
        return 'unzulaessige Eingabe'
    elif n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

Funktionen: Argumentvarianten

- Argumente können Vorgabewerte haben:

  ```
  def call_URL(protocol='http', adresse)
  ```

- bei einer unbestimmten Anzahl von Argumenten kann man ein Tuple mit variabler Länge als Argumentvariable verwenden:

  ```
  def call_URLs(*adressen)
  ```

- ebenso sind Dictionaries möglich:

  ```
  def call_URLs(**adressen)
  ```

(Aufruf von letzterem: call_URLs(adresse1="http://www.uni-koeln.de",adresse2="...")

Funktionen: Nützliches

- map(*funktion, sammlung):* ist äquivalent zu „for x in *sammlung*: *funktion*(x)“ *30
- pass: ist nützlich, wenn ein Anweisungsblock erwartet wird, aber nichts gemacht werden soll
- filter(*funktion, sammlung*): liefert die Elemente der Sammlung zurück, für die die Funktion „wahr“ ist *30
- exec '*string*': führt den String wie einen Befehl aus (Vorteil: variabler Funktionsaufruf; großer (!) Nachteil: Sicherheitsproblematik)

Module

- Funktionsbibliotheken werden als sog. „Module“ in .py-Dateien ausgelagert (bzw. vorkompiliert in .pyc-Dateien)
- Python sucht Module im aktuellen Verzeichnis und im Pythonpath, den man mit der Umgebungsvariable PYTHONPATH setzen kann:
  - Windows: set PYTHONPATH=%PYTHONPATH%;c:\py_mods
  - Unix (bash): export PYTHONPATH=$PYTHONPATH:/usr/py_mods
- Module werden mit „import“ importiert, z.B. „import sys“
- Zugriff auf Modulfunktionen erfolgt per *modul.funktion*
- Alternative: „from *modul* import *funktion*“ importiert einzelne Funktionen, auf die ohne Modulname zugegriffen werden kann

Namensräume

- Variablen und Funktionen besitzen in Python stets einen von drei Gültigkeitsbereichen, den sogenannten Namensräumen (namespaces):
  - eingebaut: Python-interne Bezeichnungen, gelten immer (können über „dir(__builtins__)“ abgefragt werden)
  - lokal: nur im aktuellen Block (Funktion, Klasse) definiert (locals())
  - modulweit: außerhalb eines Blocks definiert, gilt für die komplette Datei bzw. das komplette Modul (*modul*.__dict__)
- mit der Anweisung „global“ macht man eine Variable in allen drei Namensräumen bekannt (globals()) - sollte vermieden werden

Dateien öffnen und schließen

- Dateien öffnen: *dateivariable* = open('*dateiname*','*modus*')
- Mögliche Modi:
  - r Datei nur zum Lesen öffnen
  - w Datei nur zum Schreiben öffnen (löscht ggf. bestehende Datei)
  - a Datei nur zum Anhängen öffnen
  - r+ Datei zum Lesen und Schreiben öffnen
  - w+ Datei zum Lesen und Schreiben öffnen (löscht ggf. bestehende Datei)
  - a+ Datei zum Lesen und Anhängen öffnen
- Schreibvorgänge abschließen: *dateivariable*.flush()
- Dateien schließen: *dateivariable*.close()

Dateien lesen und schreiben

- Dateien müssen zunächst im korrekten Modus geöffnet werden
- Dateien lesen:
  - *variable* = *dateivariable*.read() liest die ganze Datei aus
  - *variable* = *dateivariable*.read(n) liest n Zeichen der Datei aus
  - *variable* = *dateivariable*.readline() liest eine Zeile der Datei aus
  - *variable* = *dateivariable*.xreadlines() liest alle Zeilen der Datei aus (Liste)
- analog existieren zum Schreiben von Dateien die Methoden „write()“ und „writelines()“
- wichtig: Das Argument der Schreibmethoden muss ein String bzw. eine Liste von Strings sein

Weitere Dateioperationen

- aktuelle Position in der Datei ermitteln mit „*variable*.tell()“
- aktuelle Position in der Datei verändern mit „*variable*.seek(ch,start)“, wobei „ch“ die Anzahl der zu verschiebenden Zeichen und „start“ entweder 0 (Dateianfang), 1 (aktuelle Position) oder 2 (Dateiende) ist
- einfaches Speichern und Auslesen anderer Datentypen mit dem pickle-Modul möglich:
  - einbinden mit „import cPickle“
  - schreiben mit „cPickle.dump(*variable*,*dateivariable*)“
  - lesen mit „*variable* = cPickle.load(*dateivariable*)“

Module: sys

- wichtige Werte von sys:
  - argv Liste mit den Argumenten der Kommandozeile
  - path Pfade, in denen Python nach Modulen sucht
  - stdin Dateivariable für die Standardeingabe, analog stdout und stderr
- sys bietet vor allem die Funktion „exit(*errorcode*)“, mit der Programme beendet werden können; Errorcode 0 bedeutet i.d.R. „kein Fehler“

Module: optparse

- dient der Verarbeitung von Kommandozeilenoptionen
- Beispiel:

```
import optparse
myparser = optparse.OptionParser()
myparser.add_option('-o', '--output',  action = 'store', dest = 'zieldatei', help =
   'Zieldatei fuer die Ausgabe', default = '/tmp/out.txt')
(options, args) = myparser.parse_args()
```

- Alternativen zu „store“: store_true, store_false, help, append, count
- beliebig viele „add_option“-Zeilen sind möglich

Module: os

- wichtige Werte (zur plattformunabhängigen Programmierung):
  - sep Separator für Verzeichnispfade ('/' bei Unix, '\' bei Windows)
  - environ Dictionary mit den Umgebungsvariablen des Systems
- wichtige Funktionen:
  - getcwd gibt das aktuelle Verzeichnis zurück
  - chdir(a) wechselt in das Verzeichnis a
  - mkdir(a) erstellt das Verzeichnis a
  - rmdir(a) löscht das Verzeichnis a
  - remove(d) löscht die Datei d
  - rename(d,e) benennt die Datei d in e um
  - system(x) führt den Befehl x auf der Systemebene aus (VORSICHT!!!)

Module: shutil und time

- wichtige Funktionen von shutil:
  - copy(a,b) kopiert die Datei a nach b (inclusive Dateirechte)
  - copyfile(a,b) kopiert die Datei a nach b (exclusive Dateirechte)
  - copytree(a,b) kopiert ein Verzeichnis mit Inhalt von a nach b
  - move(a,b) verschiebt die Datei (oder das Verzeichnis) von a nach b
- wichtige Funktionen von time:
  - time() gibt die Anzahl der Sekunden seit dem 1.1.1970 zurück
  - sleep(s) verzögert die Ausführung des Programms um s Sekunden
- Anmerkung: Eine schön formatierte Ausgabe von Datum und Uhrzeit erhält man mit „time.asctime(time.localtime(time.time()))“

Module: math und random

- wichtige Werte von math:
  - „pi“ und „e“ entsprechen den gleichnamigen math. Konstanten
- wichtige Funktionen von math:
  - sin(x) / cos(x) / tan(x) Sinus, Kosinus, Tangens von x
  - log(x) / log10(x) natürlicher bzw. Zehner-Logarithmus von x
  - sqrt(x) Quadratwurzel von x
  - ceil(x) / floor(x) Auf- und Abrunden von x
- wichtige Funktionen von random (Achtung: Deterministisch!):
  - seed() startet den Zufallsgenerator neu
  - random() liefert eine zufällige Fließkommazahl zwischen 0 und 1
  - randint(x,y) liefert eine zufällige Integerzahl zwischen x und y

Reguläre Ausdrücke: Allgemeines

- Reguläre Ausdrücke sind ein äußerst mächtiges Werkzeug zum Testen und Bearbeiten von Strings
- Reguläre Ausdrücke waren (sind?) DAS Argument für Perl schlechthin
- in Python ist die Arbeit mit REs (Regular Expressions) etwas komplizierter, dank des re-Moduls aber inzwischen praktikabel
- Arbeit mit regulären Ausdrücken erfolgt (fast) immer in 4 Schritten:
  - Importieren des re-Moduls („import re“)
  - Kompilieren des Suchmusters („muster=re.compile('musterstring')“)
  - Suchen des Musters im Text („treffer=muster.search('text')“)
  - Ausgabe/Verarbeitung der Treffer (z.B. „print(treffer.group())“)

Reguläre Ausdrücke: Optionen

- Alternativen zu „search“-Methode (findet nur die erste Übereinstimmung)
  - „findall“ findet alle Übereinstimmungen
  - „sub(ersatz,text,limit)“ ersetzt maximal limit Vorkommen des Patterns in text durch ersatz; limit ist optional
- Kompilier-Optionen (Syntax „re.compile('musterstring',re.option)“):
  - I ignoriere Groß-/Kleinschreibung
  - M Start- und Endsymbole (s. nä. Folie) beziehen sich auf die einzelnen Zeilen des Textes, nicht auf den gesamten Text

Reguläre Ausdrücke: Konstrukte

- [...] Zeichenklasse
- [^...] Komplementäre Zeichenklasse
- \w [a-zA-Z0-9_] (komplementär: \W)
- \d [0-9] (komplementär: \D)
- \s [\n\r\f\t\v] (komplementär: \S)
- . [^\n] (komplementär: \n)
- ^ Stringanfang
- $ Stringende
- | Alternative

Reguläre Ausdrücke: Wiederholungen

- ? ein- oder keinmal
- + beliebig oft, aber mindestens einmal
- * beliebig oft (auch keinmal)
- {i} genau i-mal
- {i,j} mindestens i-mal, maximal j-mal
- weitere Optionen:
  - (?=xy) gefolgt von „xy“
  - (?!xy) nicht gefolgt von „xy“
  - *? „non-greedy-Matching“ (analog: +?)

Reguläre Ausdrücke: Beispiele

- (abc)
- [abc]
- [1-9]\d*\.\d{2}
- ^A.*e.*
- \w+\.\w+@uni-xy\.de
- D-\d{5}

Fehlerbehandlung (I)

- Fehler können an vielen Stellen auftreten, ohne dass man sie verhindern kann (z.B. falsche Benutzereingaben)
- daher sinnvoll: Möglichkeiten zur Fehlerbehandlung mit try / except:

```
try:
    kritische Anweisungen
except fehler1:
    Anweisungen im Fall einer Ausnahme mit Fehlertyp 1
except fehler2:
    Anweisungen im Fall einer Ausnahme mit Fehlertyp 2
else:
    Anweisungen, wenn kein Fehler auftritt
```

Fehlerbehandlung (II)

- except kann auch ohne konkreten Fehlertyp verwendet werden
- zudem kann man except optional die Meldung des aufgetretenen Fehlers „entlocken“: *30

```
except fehler1, meldung:
    print meldung
```

- es kann auch sinnvoll sein, Fehlerausnahmen selbst zu generieren:

```
try:
    raise fehler
except:
    fehlerbehandlung
```

Arbeitsweise von Skriptsprachen

- Anbindung über das Common Gateway Interface (CGI)
- Skripte werden auf dem Server interpretiert
- Benutzer sieht nur (HTML-)Ergebnis, keinen serverseitigen Skriptcode
- Benutzereingaben werden per „post“ oder „get“ an Skript übergeben und von diesem verarbeitet
- Variante: Skripte als „Cronjob“ laufen und HTML-Dateien generieren lassen (z.B. bei Sitemaps)

CGIs mit Python

- wichtigste Zeile bei der Ausgabe von HTML-Dateien:

  ```
  print "Content-Type: text/html"
  ```

- CGI-Modul importieren mit „import cgi“
- Benutzereingaben einlesen mit „form = cgi.FieldStorage()“
- Zugriff auf einzelne Eingaben per „form.getvalue('name','default')“
- Test, ob Eingabe vorhanden ist, mit „form.has_key('name')“
- Zugriff auf Umgebungsvariablen per „os.environ.get('variable')“

CGI-Umgebungsvariablen

- Standardvariablen, die man in nahezu allen Skriptsprachen
- verwenden kann
- einige Beispiele:
  - QUERY_STRING
  - REQUEST_METHOD
  - REMOTE_ADDR
  - CONTENT_LENGTH
  - HTTP_USER_AGENT
  - DOCUMENT_ROOT

Kleiner Exkurs: Verzeichnisschutz

- der Web-Server Apache sucht i.d.R. nach einer Datei „.htaccess“ im Verzeichnis; hier kann man einen Passwortschutz unterbringen:

```
AuthType Basic
AuthUserFile /pfad/zur/datei/.htpasswd
AuthName beliebiger_name
require valid-user
```

- das Passwort für die Datei .htpasswd kann man mit Python erzeugen:

```
import crypt
crypt.crypt('passwort','XY')
```

E-Mail verschicken

- Python kann aus dem Programm heraus E-Mail versenden
- Voraussetzung: Funktionierender MTA auf dem Server
- Modul smtplib importieren: „import smtplib“
- SMTP-Server festlegen: „smtplib.SMTP('smtp-server')
- Mail versenden: „smtplib.sendmail('*von*','*an*','*nachricht*')“ mit:
  - *von* = String oder Liste/Tuple mit Absender-Adresse(n)
  - *an* = String oder Liste/Tuple mit Empfänger-Adresse(n)
  - *nachricht* = Mailinhalt, erste Zeile ist Betreffzeile „Subject:“, zweite Zeile leer
- Mails mit Anhang (für Fortgeschrittene) mit Modul „email“

Zugriff auf MySQL-Datenbanken

- Modul importieren mit „import MySQLdb“
- Verbindung herstellen: „db=MySQLdb.connect(host, user, passwd, db)“
- Datenbank-Cursor erstellen: „curs=db.cursor()“
- SQL-Befehl ausführen (Bsp.): „curs.execute("SELECT * FROM adressen")“
- Ergebniszeile auslesen: „ergebnis=curs.fetchone()“ - liefert eine Liste!!!
- alternativ: „fetchmany(n)“ oder „fetchall()“
- bei aktiver Transaktionskontrolle: „db.commit()“
- Datenbank schließen: „db.close()“

Ausblick: Python 3000 (I)

- Python 3000 (auch „Python 3K“) ist in vielen Details inkompatibel zu seinen Vorgängern
- Beispiel Fehlerbehandlung:
  - Statt „TypeError, e“ nun „TypeError as e“
  - Statt „raise 'Fehler'“ oder „raise TypeError, 'Fehler'“ nun „raise TypeError('Fehler')“
- Beispiel Division:
  - 1 / 2 Ergebnis nun „0.5“ statt „0“
  - Andere Varianten (1//2 = 0 oder 1/2.0=0.5) bleiben erhalten

Ausblick: Python 3000 (II)

- „raw_input()“ wird unbenannt in „input()“
- Dictionaries: Statt „dic.has_key(k)“ nun „if k in dic“
- Strings: Unicode ist nun Standard, Bytefolgen werden mit vorangestelltem „b“ gekennzeichnet
- Neue Anweisung: „with“
- Neues Modul: „multiprocessing“
- Neue Stringmethode: „format“

Graphische Oberflächen (GUIs)

- In Python stehen diverse externe GUIs zur Auswahl:
  - Tkinter (Tcl/Tk)
  - wxPython (wxWidgets)
  - PyQt (Qt)
  - PyGTK (GTK)
  - curses (ncurses)

Objektorientierte Programmierung

- sofern man in einem Programm viele ähnliche Daten verwalten muss, bietet es sich an, diese als Objekte mit zuvor festgelegten Methoden und Attributen zu definieren
- gleichartige Objekte sind dann Instanzen einer Klasse; Beispiel:
  - Klasse: Kunde
  - Instanzen: Meier, Schmitz, Mueller
  - Methoden: auftrag_hinzufuegen, rechnung_schreiben
  - Attribute: Firma, Strasse, Ort, Kundennummer, Auftraege

OOP in Python

- Klassen werden mit dem Schlüsselwort „class“ deklariert
- Initialisierung erfolgt per Konstruktor „__init__“ (siehe Beispiel)
- Methoden werden innerhalb der Klasse mit „def“ definiert, dabei muss die eigene Instanz als Parameter übergeben werden
- die eigene Instanz wird mit „self“ referenziert
- Zugriff auf Attribute erfolgt per „*instanz.attribut*“
- Zugriff auf Methoden erfolgt per „*instanz.methode(parameter)*“

Klassen: Beispiel

```
class Konto:
    """Simples Bankkonto"""
    def __init__(self,startguthaben=1000):
        """Konstruktor: Erzeugt neues Konto"""
        self.kontostand = startguthaben
    def einzahlung(self, betrag):
        self.kontostand = self.kontostand + betrag
    def auszahlung(self, betrag):
        self.kontostand = self.kontostand - betrag
    def anzeigen(self):
        print self.kontostand
```

Literatur und Links

- „Offizielle“ Dokumentation: http://www.python.org/doc/
- Handbuch des RRZN: „Python – Grundlagen, fortgeschrittene Programmierung und Praxis“, erhältlich im RRZK-B
- Michael Weigend, „Objektorientierte Programmierung mit Python“, mitp-Verlag
- „Dive into Python“, Mark Pilgrim, APress ; Online-Version: http://diveintopython.org/toc/index.html
- „Das Python-Praxisbuch“, Farid Hajji, Addison-Wesley
- „Python – Das umfassende Handbuch“, P. Kaiser / J. Ernesti, Galileo