Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

Oktober 2007

Inhalt — Teil 2

Datentypen II

Sets

Set (Menge): ungeordnet, doppelte Elemente werden nur einmal gespeichert

- `s = set([sequence])`
- Teilmenge: `s.issubset(t)`, `s <= t`, echte T.: `s < t`
- Obermenge: `s.issuperset(t)`, `s >= t`, echte O.: `s > t`
- Vereinigung: `s.union(t)`, `s | t`
- Schnittmenge: `s.intersection(t)`, `s & t`
- Differenz: `s.difference(t)`, `s - t`
- Symmetrische Differenz: `s.symmetric_difference(t)`, `s ^ t`
- Kopie: `s.copy()`

Wie für Sequenzen gibt es auch: `x in set`, `len(set)`, `for x in set`, `add`, `remove`

Dictionaries

Dictionary: Zuordnung Schlüssel → Wert

```
>>> a = { "spam": 1, "eggs": 17}
>>> a["eggs"]
17
>>> a["bacon"] = 42
>>> a
{'eggs': 17, 'bacon': 42, 'spam': 1}
```

Über Dictionaries iterieren:

```
for key in a:
    print key, a[key]
```

Operationen auf Dictionaries

- Eintrag löschen: `del`
- alle Einträge löschen: `a.clear()`
- Kopie: `a.copy()`
- Ist Schlüssel enthalten? `a.has_key(k)`, `k in a`
- Liste von (key, value)-Tupeln: `a.items()`
- Liste aller Schlüssel: `a.keys()`
- Liste aller Werte: `a.values()`
- Eintrag holen: `a.get(k[, x])`
- Eintrag löschen und zurückgeben: `a.pop(k[, x])`
- Eintrag löschen und zurückgeben: `a.popitem()`

Objektorientierte Programmierung

Objektorientierte Programmierung

- Bisher: prozedurale Programmierung
  - Daten
  - Funktionen, die auf den Daten operieren
- Alternative: Fasse zusammengehörige Daten und Funktionen zusammen zu eigenem Datentypen
- $\longrightarrow$ Erweiterung von Strukturen/Datenverbünden aus C/Fortran

Einfache Klassen als Structs verwenden

```
class Punkt:
    pass

p = Punkt()
p.x = 2.0
p.y = 3.3
```

- Klasse: Eigener Datentyp (hier: `Punkt`)
- Objekt: Instanz der Klasse (hier: `p`)
- Attribute (hier `x`, `y`) können dynamisch hinzugefügt werden

Klassen

```
class Punkt:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Punkt(2.0, 3.0)
print p.x, p.y
p.x = 2.5
p.z = 42
```

- `__init__`: Wird automatisch nach Erzeugung eines Objekts aufgerufen

Methoden auf Objekten

```
class Punkt:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        n = math.sqrt(self.x**2 + self.y**2)
        return n

p = Punkt(2.0, 3.0)
print p.x, p.y, p.norm()
```

- Methodenaufruf: automatisch das Objekt als erster Parameter
- $\longrightarrow$ wird üblicherweise `self` genannt
- Achtung: Kein Überladen von Methoden möglich!

Objekte in Strings konvertieren

Standard-Rückgabe von `str(...)` für eigene Objekte:

```
>>> p = Punkt(2.0, 3.0)
>>> print p  # --> print str(p)
<__main__.Punkt instance at 0x402d7a8c>
```

Das kann man anpassen:

```
def __str__(self):
    return "(%i, %i)" % (self.x, self.y)
```

```
>>> print p
(2, 3)
```

Objekte in Strings konvertieren

Standard-Rückgabe von `str(...)` für eigene Objekte:

```
>>> p = Punkt(2.0, 3.0)
>>> print p  # --> print str(p)
<__main__.Punkt instance at 0x402d7a8c>
```

Das kann man anpassen:

```
def __str__(self):
    return "(%i, %i)" % (self.x, self.y)
```

```
>>> print p
(2, 3)
```

Objekte vergleichen

Standard: == prüft Objekte eigener Klassen auf Identität.

```
>>> p1 = Punkt(2.0, 3.0)
>>> p2 = Punkt(2.0, 3.0)
>>> p1 == p2
False
```

Das kann man anpassen:

```
def __eq__(self, other):
    return (self.x == other.x) and
           (self.y == other.y)
```

```
>>> p1 == p2
True
>>> p1 is p2 # Identitaet pruefen
False
```

Objekte vergleichen

Standard: == prüft Objekte eigener Klassen auf Identität.

```
>>> p1 = Punkt(2.0, 3.0)
>>> p2 = Punkt(2.0, 3.0)
>>> p1 == p2
False
```

Das kann man anpassen:

```
def __eq__(self, other):
    return (self.x == other.x) and
           (self.y == other.y)
```

```
>>> p1 == p2
True
>>> p1 is p2 # Identitaet pruefen
False
```

Objekte vergleichen

Weitere Vergleichsoperatoren:

- `<` : `__lt__(self, other)`
- `<=` : `__le__(self, other)`
- `!=` : `__ne__(self, other)`
- `>` : `__gt__(self, other)`
- `>=` : `__ge__(self, other)`

Alternativ: `__cmp__(self, other)`, gibt zurück:

- negativen Integer, wenn `self < other`
- null, wenn `self == other`
- positiven Integer, wenn `self > other`

Datentypen emulieren

Man kann mit Klassen vorhandene Datentypen emulieren:

- Zahlen: `int(myobj)`, `float(myobj)`, Rechenoperationen, ...
- Funktionen: `myobj(...)`
- Sequenzen: `len(myobj)`, `myobj[...]`, `x in myobj`, ...
- Iteratoren: `for i in myobj`

Siehe dazu Dokumentation:
http://docs.python.org/ref/specialnames.html

Klassenvariablen

Haben für alle Objekte einer Klasse stets den gleichen Wert:

```
class Punkt:
    anzahl = 0  #Anzahl aller Punkt-Objekte
    def __init__(self, x, y):
        self.__class__.anzahl += 1
        ...
```

```
>>> p1 = Punkt(2, 3); p2 = Punkt(3, 4)
>>> p1.anzahl
2
>>> p2.anzahl
2
>>> Punkt.anzahl
2
```

Klassenmethoden und statische Methoden

```
class Spam:
    spam = "I don't like spam."

    @classmethod
    def cmethod(cls):
        print cls.spam

    @staticmethod
    def smethod():
        print "Blah blah."
```

```
Spam.cmethod()
Spam.smethod()
s = Spam()
s.cmethod()
s.smethod()
```

Vererbung

Oft hat man verschiedene Klassen, die einander ähneln.
Vererbung erlaubt:

- Hierarchische Klassenstruktur (Ist-ein-Beziehung)
- Wiederverwenden von ähnlichem Code

Beispiel: Verschiedene Telefon-Arten

- Telefon
- Handy (ist ein Telefon mit zusätzlichen Funktionen)
- Fotohandy (ist ein Handy mit zusätzlichen Funktionen)

Vererbung

```
class Telefon:
    def telefonieren(self):
        pass

class Handy(Telefon):
    def sms_schicken(self):
        pass
```

Handy erbt jetzt Methoden und Attribute von Telefon.

```
h = Handy()
h.telefonieren() # Geerbt von Telefon
h.sms_schicken() # Eigene Methode
```

Methoden überschreiben

In der abgeleiteten Klasse können die Methoden der Elternklasse überschrieben werden:

```
class Handy(Telefon):
    def telefonieren(self):
        suche_funkverbindung()
        Telefon.telefonieren(self)
```

Mehrfachvererbung

Klassen können von mehreren Elternklassen erben. Bsp:

- Fotohandy ist ein Telefon
- Fotohandy ist eine Kamera

```
class Fotohandy(Handy, Kamera):
    pass

h = Fotohandy()
h.telefonieren()   # geerbt von Handy
h.fotografieren()  # geerbt von Kamera
```

Attribute werden in folgender Reihenfolge gesucht:
Fotohandy, Handy, Elterklasse von Handy (rekursiv), Kamera, Elternklasse von Kamera (rekursiv).

Private Attribute

- In Python gibt keine privaten Variablen oder Methoden.
- Konvention: Attribute, auf die nicht von außen zugegriffen werden sollte, beginnen mit einem Unterstrich: `_foo`.
- Um Namenskonflikte zu vermeiden: Namen der Form `__foo` werden durch `_klassenname__foo` ersetzt:

```
class Spam:
    __eggs = 3
```

```
>>> dir(Spam)
>>> ['_Spam__eggs', '__doc__', '__module__']
```

Properties

Sollen beim Zugriff auf eine Variable noch Berechnungen oder Überprüfungen durchgeführt werden: Getter und Setter

```
class Spam(object):
    def __init__(self):
        self._value = 0

    def _get_value(self):
        return self._value

    def _set_value(self, value):
        if value <= 0:  self._value = 0
        else:  self._value = value

    value = property(_get_value, _set_value)
```

Properties

Auf Properties wird wie auf gewöhnliche Attribute zugegriffen:

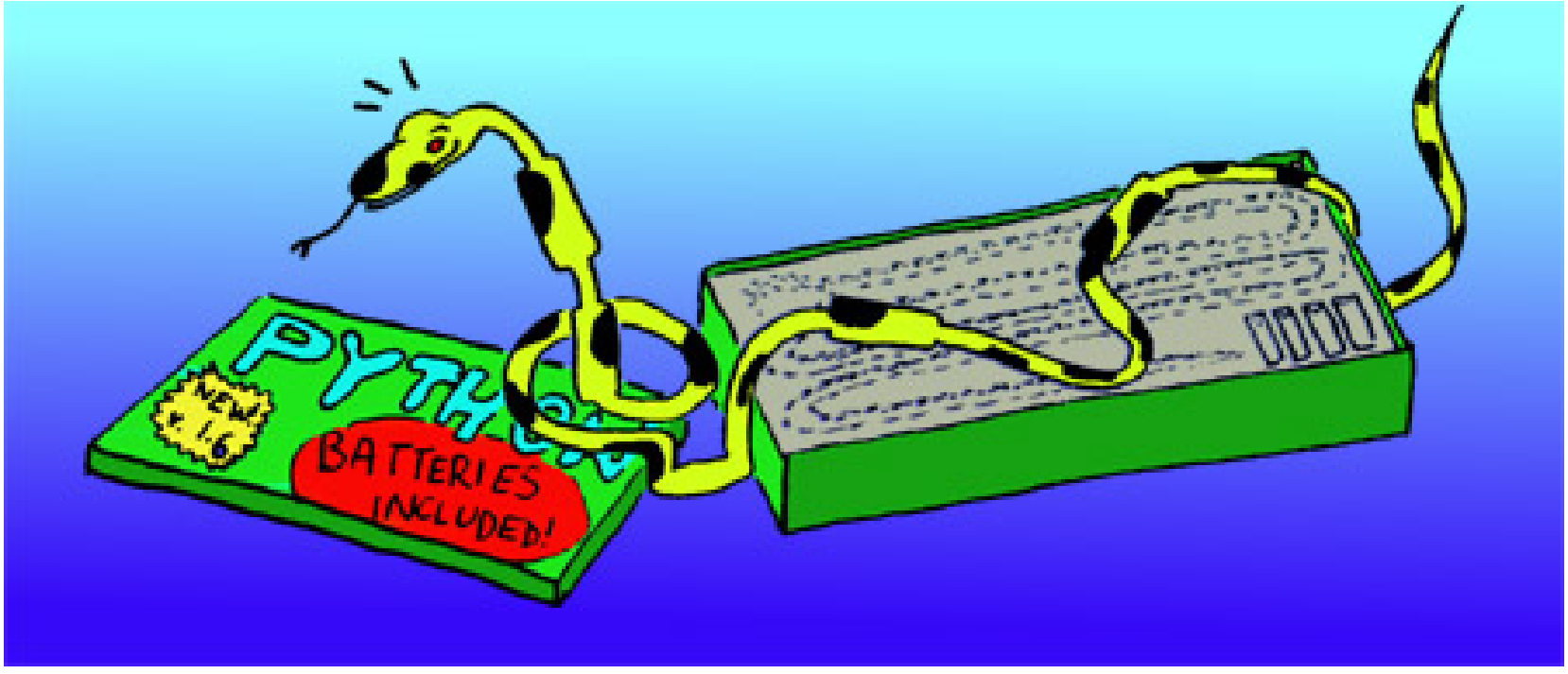
```
>>> s = Spam()
>>> s.value = 6    # set_value(6)
>>> s.value        # get_value()
>>> 6
>>> s.value = -6   # set_value(-6)
>>> s.value        # get_value()
>>> 0
```

- Getter und Setter können nachträglich hinzugefügt werden, ohne die API zu verändern.
- Zugriff auf `_value` immer noch möglich

Pythons Standardbibliothek

Pythons Standardbibliothek

„Batteries included“: umfassende Standardbibliothek für die verschiedensten Aufgaben



Mathematik: `math`

- Konstanten: `e`, `pi`
- Auf- und Abrunden: `floor(x)`, `ceil(x)`
- Exponentialfunktion: `exp(x)`
- Logarithmus: `log(x[, base])`, `log10(x)`
- Potenz und Quadratwurzel: `pow(x, y)`, `sqrt(x)`
- Trigonometrische Funktionen: `sin(x)`, `cos(x)`, `tan(x)`
- Kovertierung Winkel $\leftrightarrow$ Radiant: `degrees(x)`, `radians(x)`

```
>>> import math
>>> math.sin(math.pi)
1.2246063538223773e-16
>>> math.cos(math.radians(30))
0.8660254037844387
```

Zufall: `random`

- Zufällige Integers:
  `randint(a, b)`, `randrange([start,] stop[, step])`
- Zufällige Floats (Gleichverteilg.): `random()`, `uniform(a, b)`
- Andere Verteilungen: `expovariate(lambd)`,
  `gammavariate(alpha, beta)`, `gauss(mu, sigma)`, ...
- Zufälliges Element einer Sequenz: `choice(seq)`
- Mehrere eindeutige, zufällige Elemente einer Sequenz:
  `sample(population, k)`
- Sequenz mischen: `shuffle(seq[, random])`

```
>>> s = [1, 2, 3, 4, 5]
>>> random.shuffle(s)
>>> s
[2, 5, 4, 3, 1]
>>> random.choice("Hallo Welt!")
'e'
```

Operationen auf Verzeichnisnamen: os.path

- Pfade: `abspath(path)`, `basename(path)`, `normpath(path)`, `realpath(path)`
- Pfad zusammensetzen: `join(path1[, path2[, ...]])`
- Pfade aufspalten: `split(path)`, `splitext(path)`
- Datei-Informationen: `isfile(path)`, `isdir(path)`, `islink(path)`, `getsize(path)`, ...
- Home-Verzeichnis vervollständigen: `expanduser(path)`
- Umgebungsvariablen vervollständigen: `expandvars(path)`

```
>>> os.path.join("spam", "eggs", "ham.txt")
'spam/eggs/ham.txt'
>>> os.path.splitext("spam/eggs.py")
('spam/eggs', '.py')
>>> os.path.expanduser("~/spam")
'/home/rbreu/spam'
>>> os.path.expandvars("/bla/$TEST")
'/bla/test.py'
```

Dateien und Verzeichnisse: os

- Working directory: `getcwd()`, `chdir(path)`
- Dateirechte ändern: `chmod(path, mode)`
- Benutzer ändern: `chown(path, uid, gid)`
- Verzeichnis erstellen: `mkdir(path[, mode])`, `makedirs(path[, mode])`
- Dateien löschen: `remove(path)`, `removedirs(path)`
- Dateien umbenennen: `rename(src, dst)`, `renames(old, new)`
- Liste von Dateien in Verzeichnis: `listdir(path)`

```
for datei in os.listdir("mydir"):
    os.chmod(os.path.join("mydir", datei),
             stat.S_IRUSR|stat.S_IWUSR)
```

Verzeichnislisting: `glob`

Liste von Dateien in Verzeichnis, mit Unix-artiger Wildcard-Vervollständigung: `glob(path)`

```
>>> glob.glob("python/[a-c]*.py")
['python/confitest.py',
 'python/basics.py',
 'python/curses_test2.py',
 'python/curses_keys.py',
 'python/cmp.py',
 'python/button_test.py',
 'python/argument.py',
 'python/curses_test.py']
```

Dateien und Verzeichnisse: `shutil`

Higher Level-Operationen auf Dateien und Verzeichnissen.

- Datei kopieren: `copyfile(src, dst)`, `copy(src, dst)`
- Rekursiv kopieren; `copytree(src, dst[, symlinks])`
- Rekursiv löschen:
  `rmtree(path[, ignore_errors[, onerror]])`
- Rekursiv verschieben: `move(src, dst)`

```
shutil.copytree("spam/eggs", "../beans",
                symlinks=True)
```

Andere Prozesse starten: subprocess

Einfaches Ausführen eines Programmes:

```
p = subprocess.Popen(["ls", "~"], shell=True)
returncode = p.wait() # Auf Ende warten
```

Zugriff auf die Ausgabe eines Programmes:

```
p = Popen(["ls"], stdout=PIPE, stderr=STDOUT,
          close_fds=True)
p.wait()
output = p.stdout.read()
```

Pipes zwischen Prozessen (`ls -l | grep txt`)

```
p1 = Popen(["ls", "-l"], stdout=PIPE)
p2 = Popen(["grep", "txt"], stdin=p1.stdout)
```

Threads: threading

Programmteile gleichzeitig ablaufen lassen mit Thread-Objekten:

```
class Counter(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.counter = 0

    def run(self):  # Hauptteil
        while self.counter < 10:
            self.counter += 1
            print self.counter

counter = Counter()
counter.start() # Thread starten
# hier etwas gleichzeitig tun...
counter.join() # Warte auf Ende des Threads
```

Threads: `threading`

- Problem, wenn zwei Threads gleichzeitig auf das gleiche Objekt schreibend zugreifen wollen!
- $\longrightarrow$ Verhindern, dass Programmteile gleichzeitig ausgeführt werden mit Lock-Objekten
- Locks haben genau zwei Zustände: `locked` und `unlocked`

```
lock = threading.Lock()

lock.aquire() # Warte bis Lock frei ist
              # und locke es dann
#... wichtiger Code
lock.release() # Lock freigeben fuer andere
```

Threads: `threading`

- Kommunikation zwischen Threads: Z.B. mittels Event-Objekten
- Events haben zwei Zustände: gesetzt und nicht gesetzt
- ähnlich Locks, aber ohne gegenseitigen Ausschluss

Bsp: Event, um Threads mitzuteilen, dass sie sich beenden sollen.
Methoden auf Event-Objekten:

- Status des Events abfragen: `isSet()`
- Setzen des Events: `set()`
- Zurücksetzten des Events: `clear()`
- Warten, dass Event gesetzt wird: `wait([timeout])`

Zugriff auf Kommandozeilenparameter: optparse

- Einfach: Liste mit Parametern → `sys.argv`
- Komfortabler für mehrere Optionen: `OptionParser`

```
parser = optparse.OptionParser()
parser.add_option("-f", "--file",
                  dest="filename",
                  default="out.txt",
                  help="Output file")
parser.add_option("-q", "--quiet",
                  action="store_false",
                  dest="verbose",
                  default=True,
                  help="don't print info")

(options, args) = parser.parse_args()
print options.filename, options.verbose
print args
```

Zugriff auf Kommandozeilenparameter: optparse

So wird ein optparse-Programm verwendet:

```
$ ./test.py -h
usage: test.py [options]

options:
  -h, --help            show this help message and exit
  -f FILENAME, --file=FILENAME
                        Output file
  -q, --quiet           don't print status messages
```

```
$ ./test.py -f aa bb cc
aa True
['bb', 'cc']
```

Konfigurationsdateien: `ConfigParser`

Einfaches Format zum Speichern von Konfigurationen u.Ä.:
Windows INI-Format

```
[font]
font = Times New Roman
#this is a comment
size = 16

[colors]
font = black
pointer = %(font)s
background = white
```

Konfigurationsdateien: ConfigParser

Config-Datei lesen:

```
parser = ConfigParser.SafeConfigParser()
parser.readfp(open("config.ini", "r"))
print parser.get("colors", "font")
```

Weitere Parser-Methoden:

- Liste aller Sections: `sections()`
- Liste aller Optionen: `options(section)`
- Liste aller Optionen und Werte: `items(section)`
- Werte lesen: `get(sect, opt)`, `getint(sect, opt)`, `getfloat(sect, opt)`, `getboolean(sect, opt)`

Konfigurationsdateien: ConfigParser

Config-Datei schreiben:

```
parser = ConfigParser.SafeConfigParser()
parser.add_section("colors")
parser.set("colors", "font", "black")
parser.write(open("config.ini", "w"))
```

Weitere Parser-Methoden:

- Section hinzufügen: `add_section(section)`
- Section löschen: `remove_section(section)`
- Option hinzufügen: `set(section, option, value)`
- Option entfernen: `remove_option(section, option)`

CSV-Dateien: csv

CSV: Comma-seperated values

- Tabellendaten im ASCII-Format
- Spalten durch ein festgelegtes Zeichen (meist Komma) getrennt

```
reader = csv.reader(open("test.csv", "rb"))
for row in reader:
    for item in row:
        print item
```

```
writer = csv.writer(open(outfile, "wb"))
writer.writerow([1, 2, 3, 4])
```

CSV-Dateien: csv

Mit verschiedenen Formaten (Dialekten) umgehen:

```
reader(csvfile, dialect='excel') # Default
writer(csvfile, dialect='excel_tab')
```

Einzelne Formatparameter angeben:

```
reader(csvfile, delimiter=";")
```

Weitere Formatparameter: `lineterminator`, `quotechar`, `skipinitialspace`, ...

Objekte serialisieren: `pickle`

Beliebige Objekte in Dateien speichern:

```
obj = {"hallo": "welt", "spam":1}
pickle.dump(obj, open("bla.txt", "wb"))
# ...
obj = pickle.load(open("bla.txt", "rb"))
```

Objekt in String unwandeln (z.B. zum Verschicken über Streams):

```
s = pickle.dumps(obj)
# ...
obj = pickle.loads(s)
```

Persistente Dictionaries: `shelve`

Ein Shelve benutzt man wie ein Dictionary, es speichert seinen Inhalt in eine Datei.

```
d = shelve.open("bla")
d["spam"] = "eggs"
d["bla"] = 1
del d["foo"]
d.close()
```

Tar-Archive: `tarfile`

Ein tgz entpacken:

```
tar = tarfile.open("spam.tgz")
tar.extractall()
tar.close()
```

Ein tgz erstellen:

```
tar = tarfile.open("spam.tgz", "w:gz")
tar.add("/home/rbreu/test")
tar.close()
```

Log-Ausgaben: `logging`

Flexible Ausgabe von Informationen, kann schnell angepasst werden.

```
import logging
logging.debug("Very special information.")
logging.info("I am doing this and that.")
logging.warning("You should know this.")
```

```
WARNING:root:You should know this.
```

- Messages bekommen einen Rang (Dringlichkeit): `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`
- Default: Nur Messages mit Rang `WARNING` oder höher werden ausgegeben

Log-Ausgaben: `logging`

Beispiel: Ausgabe in Datei, benutzerdefiniertes Format, größeres Log-Level:

```
logging.basicConfig(level=logging.DEBUG,
  format="%(asctime)s %(levelname)-8s %(message)s",
  datefmt="%Y-%m-%d %H:%M:%S",
  filename='/tmp/spam.log', filemode='w')
```

```
$ cat /tmp/spam.log
2007-05-07 16:25:14 DEBUG    Very special information.
2007-05-07 16:25:14 INFO     I am doing this and that.
2007-05-07 16:25:14 WARNING  You should know this.
```

Es können auch verschiedene Loginstanzen gleichzeitig benutzt werden, siehe Python-Dokumentation.

Reguläre Ausdrücke: `re`

Einfaches Suchen nach Mustern:

```
>>> re.findall(r"\[.*?\]", "a[bc]g[hal]def")
['[bc]', '[hal]']
```

Ersetzen von Mustern:

```
>>> re.sub(r"\[.*?\]", "!", "a[bc]g[hal]def")
'a!g!def'
```

Wird ein Regex-Muster mehrfach verwendet, sollte es aus Geschwindigkeitsgründen compiliert werden:

```
>>> pattern = re.compile(r"\[.*?\]")
>>> pattern.findall("a[bc]g[hal]def")
['[bc]', '[hal]']
```

Reguläre Ausdrücke: `re`

Umgang mit Gruppen:

```
>>> re.findall("(\[.*?\])|(<.*?>)",
                "[hi]s<b>sdd<hal>")
[('[hi]', ''), ('', '<b>'), ('', '<hal>')]
```

Flags, die das Verhalten des Matching beeinflussen:

```
>>> re.findall("^a", "abc\nAbc", re.I|re.M)
>>> ['a', 'A']
```

- `re.I`: Groß-/Kleinschreibung ingnorieren
- `re.M`: `^` matcht am Anfang jeder Zeile (nicht nur am Anfang des Strings)
- `re.S`: `.` matcht auch Zeilenumbruch

Sockets: socket

Client-Socket erstellen und mit Server verbinden:

```
sock = socket.socket(socket.AF_INET,
                     socket.SOCK_STREAM)
sock.connect(("whois.denic.de", 43))
```

Mit dem Server kommunizieren:

```
sock.send("fz-juelich.de" + "\n")
print sock.recv(4096) # Antwort lesen
sock.close()
```

Sockets: socket

Server-Socket erstellen:

```
server_socket = socket.socket(socket.AF_INET)
server_socket.bind(("localhost", 6666))
```

Auf Client-Verbindungen warten und sie akzeptieren:

```
server_socket.listen(1)
(sock, address) = server_socket.accept()
```

Mit dem Client kommunizieren:

```
sock.send("Willkommen!\n")
# ...
```

Zusammenfassung und Ausblick

Zusammenfassung

Wir haben kennengelernt:

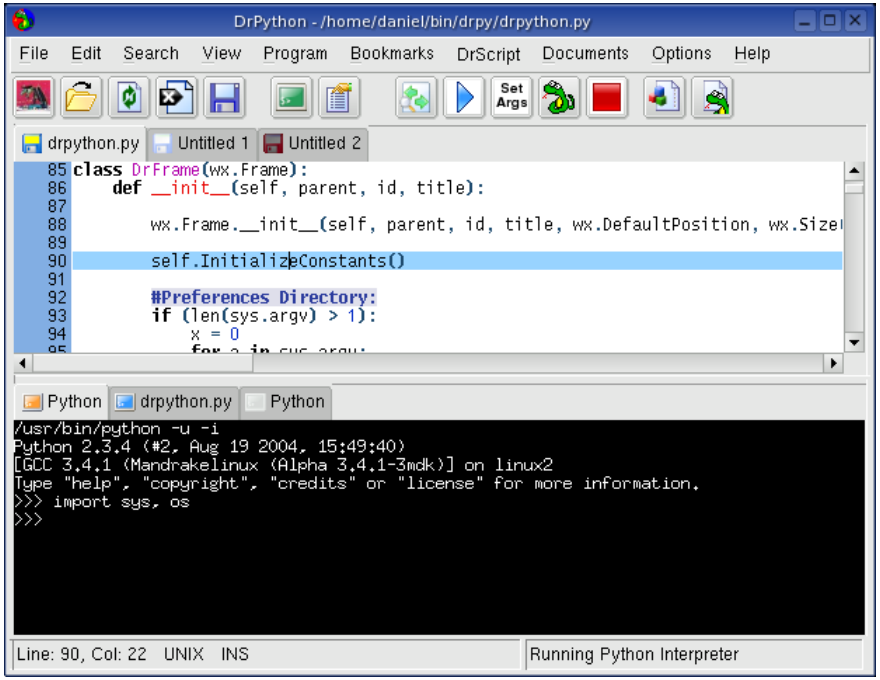
- verschiedene Datentypen (tw. „High Level“)
- die wichtigsten Statements
- Funktionsdeklaration und -Benutzung
- Module und Pakete
- Fehler und Ausnahmen, Behandlung selbiger
- objektorientierte Programmierung
- einige häufig verwendete Standardmodule

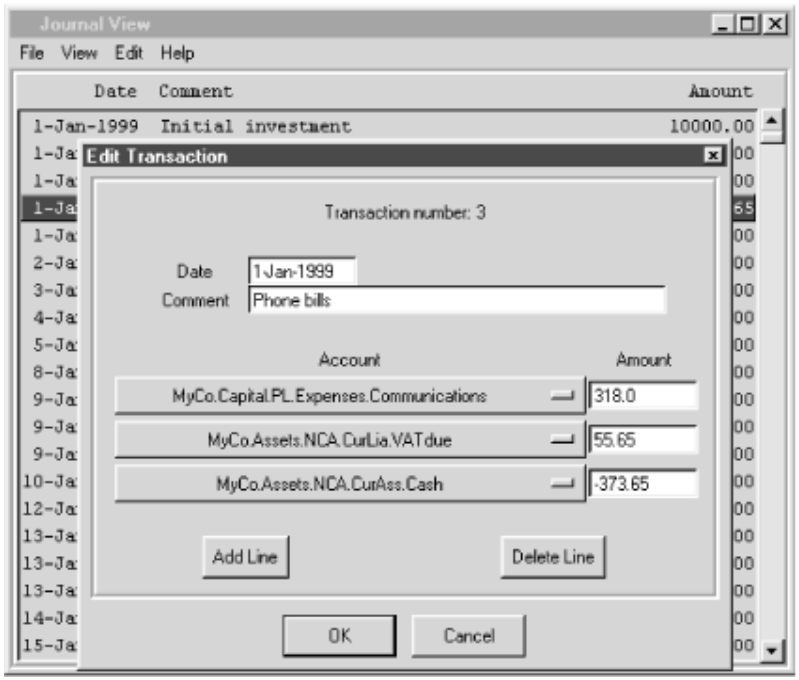
Offene Punkte

Nicht behandelte, tw. fortgeschrittene Themen:

- funktionale Techniken mit Listen: List Comprehensions, `filter()`, `map()`, `reduce()`
- Lambda-Funktionen, Funktionen mit variablen Parametern
- Iteratoren, Generatoren
- Closures, Dekoratoren (Funktionswrapper)
- Ausnutzung von Pythons Dynamik:
  `getattr`, `setattr`, Metaklassen, ...
- Weitere Standardmodule: Mail, HTML, XML, Zeit&Datum, Profiling, Debugging, Unittesting, ...
- Third Party-Module:
  GUI, Grafik, Webprogrammierung, Datenbanken, ...

Grafische Benutzeroberflächen





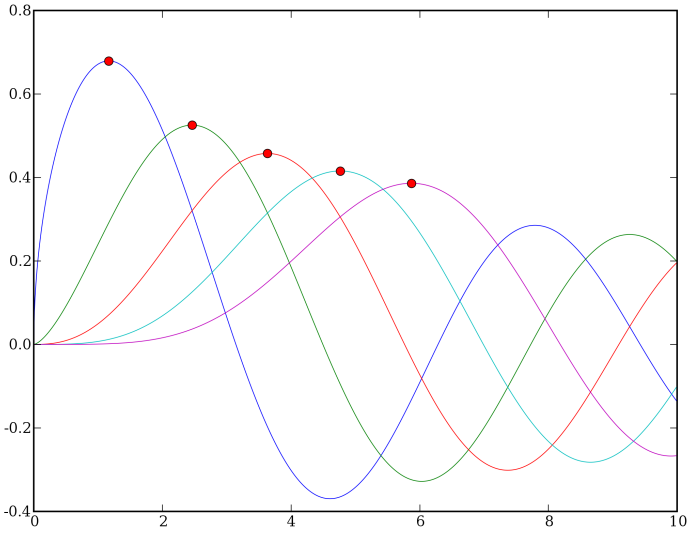
- Tk (aus Standardbibliothek)
- wxWidgets (GUI-Toolkit je nach Betriebssystem)
- GTK
- QT

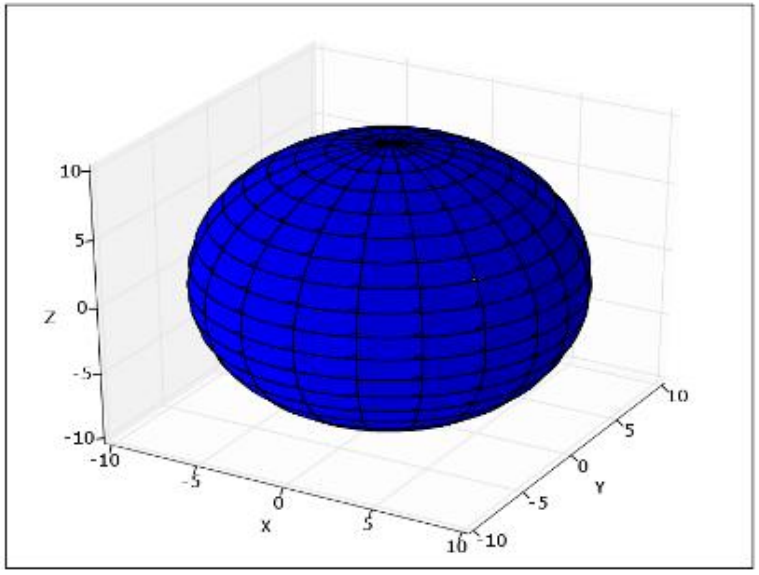
Web-Programmierung

- CGI-Scripte: Modul `cgi` aus Standardbibliothek
- Webframeworks: Django, TurboGears, Pylons, ...
- Templatesysteme: Cheetah, Genshi, Jinja, ...
- Content Management Systeme (CMS): Zope, Plone, Skeletonz, ...
- Wikis: MoinMoin, ...

NumPy + SciPy + Matplotlib = Pylab

Ein Ersatz für MatLab: Matritzenrechnung, numerische Funktionen, Plotten, ...






```
A = matrix([[1, 2], [2, 1]]); b = array([1, -1])
matshow(A)
(eigvals, eigvecs) = eig(A)
x = linalg.solve(A, b)
```

Viel Spaß mit