Python – eine Scriptsprache?

Allgemeines und Historisches

- Entwickelt von Guido van Rossum ca. 1990
- ursprünglich abgeleitet von ABC
- vor Anfang an als einfache zu benutzende Sprache konzipiert
- man sollten schnell zu einem Ergebnis kommen (rapid prototyping)
- Sprachelemente aus verschiedenen Sprachen (z.B. C)
- Bytecode erlaubt Systemunabhängigkeit
- Ansätze für einen Compiler (Psyco)

einfache Datentypen

- int — z.B. 1, 2, 3 oder auch int(3.1415)
  - octal — z.B. 0123 = 83, oct(100) = '0144'
  - hexadezimal — z.B. 0xffff = 65535 , hex(100) = '0x64'
- long — z.B. 1L oder 12345678900L, 10**10 = 10000000000L (64Bit-Systemen ab 10**19)
- float — z.B. f = 1.23 oder pi = 3.1415*4
- complex — z.B. c = 3+2j, abs(c) = 3.605551275463989
- string — z.B. “Klaus” oder 'Mandel', mehrzeilig mit ''' und '''
- unicode — z.B. u'Eißendörf'
- tuple — z.B. t = (1,'Klaus'), t[0]=1, t[1]='Klaus', nicht änderbar

erweiterte Typen

- **Listen:** Felder mit unterschiedlichen Datentypen
  - Index beginnt bei 0, nur numerische Indizes erlaubt
  - Slicing: l = [1,2,3,4] ; l[2:] = [3,4] ; l[-1] = 4; l[:3] = [1,2,3]
  - Funtionen: append, count, extend, index, insert,pop, remove, reverse,sort
- **Dict:** assoziatives Feld mit unterschiedlichen Datentypen für Indizes **und** Werte
  - Funktionen: iterkeys, clear, itervalues, copy, keys, fromkeys, pop, get, popitem, has_key, setdefault, items, update, iteritems, values
  - d = {} ; d['Klaus'] = 1 ; d[2] = 'Mandel'
    print d {2: 'Mandel', 'Klaus': 1}
    print d[2] = 'Mandel'
    print d['Klaus'] = 1

String - Verarbeitung

- einige wichtige Funktionen:
  - center(int) 'Klaus'.center(20) = ' Klaus '
  - capitalize() 'cafe leben'.capitalize() = 'Cafe leben'
  - split(),rsplit '1:2:3:4'.split(':') = ['1','2','3','4']
'Test1uhuTest2'.split('uhu') = ['Test1','Test2']
  - strip, rstrip entfernt whitespaces am Anfang und Ende
  - decode(),encode() 'Eißendorf'.decode('latin-1') = u'Ei\xc3\x9fendorf'
  - find(), index() 'Klaus'.find('z') = -1 ; 'Klaus'.index('z') = ERROR
  - rfind, rindex Suche beginnt von Hinten
  - replace() 'Klaus'.replace('a','b') = 'Klbus'
  - join() 'Klaus'.join(['1','2','3']) = '1Klaus2Klaus3'
  - count() 'Klaaaaus'.count('a') = 4
  - partition 'Klaus'.partition('a') = ('Kl', 'a', 'us')
- spezielles Modul ***string*** vorhanden mit zusätzlichen Funktionen wie : atoi(), atof usw.

Typische Sprachelemente

- Mehrfachzuweisung z.B. a=b=c=1.23
- Funktionen:
  - max, min: min(-1,3,3.1415,5.3,-4) = -4 ;
    max(-1,3,3.1415,5.3,-4) = 5.3
    min([1,2,3,4]) = 1
  - zip : for i,j in zip(liste1,list2):
  - lambda : a=lambda x: x**3 ; a(3) = 27
  - map : map(a, range(10)) = [0, 1, 8, 27, 64, 125,
    216, 343, 512, 729]
  - **in** – Operator : for I in list: ; if I in list:
  - range : range(4) = [0,1,2,3] ; range(1,10,2) = [1, 3, 5, 7, 9]
  - list, tuple : list('Klaus') = ['K', 'l', 'a', 'u', 's']
  - Funktion def func(a=2):
  - beliebige Parameter

```
def func(**kwd):
    print kwd
func(a=1, b=2, c=3)        {'a': 1, 'b': 2, 'c': 3}
```

Objektorientierung

- Klassen werden mit dem Schlüssenwort ***class*** deklariert:
  - einfach Klasse ist z.B.

```
class test:
    I = 0
    j = 0
```

  - Constructor __init__(self)

```
def __init__(self, i, j):
    self.i = I
    self.j = j
```

  - überladbare Operatoren:
    - __add__ Addition
    - __mult__ Multiplikation
    - __repr__ Ausgabe bei print-Anweisung oder str()
    - usw.
  - private Attribute beginnen mit einem bzw. 2 _ z.B. __Wert
  - Mehrfachvererbung möglich
  - Klassenmethoden mit

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

ein eigene Klasse

```
# A simple vector class


def vec(*v):
    return Vec(*v)



class Vec:

    def __init__(self, *v):
        self.v = list(v)

    def fromlist(self, v):
        if not isinstance(v, list):
            raise TypeError
        self.v = v[:]
        return self

    def __repr__(self):
        return 'vec(' + repr(self.v)[1:-1] + ')'

    def __len__(self):
        return len(self.v)

    def __getitem__(self, i):
        return self.v[i]

    def __add__(self, other):
        # Element-wise addition
        v = map(lambda x, y: x+y, self, other)
        return Vec().fromlist(v)

    def __sub__(self, other):
        # Element-wise subtraction
        v = map(lambda x, y: x-y, self, other)
        return Vec().fromlist(v)

    def __mul__(self, scalar):
        # Multiply by scalar
        v = map(lambda x: x*scalar, self.v)
        return Vec().fromlist(v)



def test():
    a = vec(1, 2, 3)
    b = vec(3, 2, 1)
    print a
    print b
    print a+b
    print a-b
    print a*3.0

test()
```

Ergebnis:

```
vec(1, 2, 3)
vec(3, 2, 1)
vec(4, 4, 4)
vec(-2, 0, 2)
vec(3.0, 6.0, 9.0)
```

Modul – Konzept

- Module sind in sich abgeschlossenen Klassen oder Funktions-sammlungen
  - Einbinden mit: import module [as module2]
    oder from module import *|function|class
  - import-Aufrufe können überall stehen
  - bedingter Import ist möglich
  - zur Kontrolle des Modul-Imports läßt sich die Funktion **__import__** überladen
  - Module werden im Standardinstallationspfad von Python gesucht ; änderbar durch die $PYTHONPATH oder sys.path
  - interne Attribut **__name__** ermöglicht Unterscheidung, ob ein Modul wirklich als Modul geladen oder als Programm gestartet wird
    - mit: if __name__ == "__main__":
      main()
  - je nach Art der Einbindung haben Module einen eigenen Namensraum
    z.B. modulname.funktionsname()
    oder werden in den aktuellen Namensraum integriert

Ausnahme - Konzept

- Fehler führen normalerweise zum Abbruch mit Ausgabe einer Fehlermeldung, die auf den Fehlerort hinweist:

```
>>> c=c+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

- Fehlerbehandlung mit try: ... except [ERROR]: ...
  (oder try: ... finally: ...)
  - Beispiel:

```
try:
    f = open('test.dat', 'r')
except IOError:
    print 'Datei ist nicht vorhanden'
```

  - mit finally lassen sich Programmteile definieren, die **auf jeden Fall** ausgeführt werden
- Exceptions lassen sich auch im Programm auslösen mit raise z.B. raise RuntimeError, 'startparser: still busy'
- diese Möglichkeit wird in größeren Module genutzt, um im aufrufenden Programm auf Fehler reagieren zu können

Dokumentation von Python

- Kommentare lassen sich auf verschiedene Weise schreiben
  - alles nach eine #-Zeichen wird ignoriert, auch am Ende einer Pythonzeile
  - Beschreibungen von Funktionen oder Klassen lassen sich direkt nach der Deklaration einfügen:
    z.B. def test():
    “Eine Test-Funktion”
    return 'Test'

    oder def test2(a):
    “””Eine weitere Test-Funktion:
    Parameter:
    a - erster Wert
    “””
  - mit dem Python-Tool ***pydoc*** lassen sich dann Dokumentationen erzeugen:
    z.B. pydoc test_module.py
    ergibt:

```
Help on module testhelp:

NAME
    testhelp

FILE
    /home/vt4/ma/Documents/Vortrag_Python/testhelp.py

FUNCTIONS
    test2(a)
        Eine weitere Test-Funktion:
        Parameter:
                a       -       erster Wert
```

Python-Unittest -- doctest

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For
       example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
    Traceback (most recent call last):
       ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact
       integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
       ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
       ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result

def _test():
    import doctest
    doctest.testmod()

if __name__ == "__main__":
    _test()
```

Was ich nicht gesagt habe

- reguläre Ausdrücke mit Modul ***re***
- XML-Verarbeitung
- Pickle, serialisieren von Objekten ***cPickle***
- Threads ***thread, threading***
- Netzwerk ***sockets***
- HTML-Parser ***htmllib***
- Email-Package ***email, poplib, imaplib***
- XMLRPC
- RemoteObjects mit ***Pyro***
- Datenbank, Standard ist ***sqlite***
- GUI – Programmierung : Tkinter, QT3 und QT4, KDE, gnome,WXwindows
- Kryptographie: ***sha, hashlib, ssl***
- Kompression ***zlib, bz2***