Einführung in Python (1)

Was ist Python?

- objektorientierte Scriptsprache
- unter freier Lizenz (ohne die Restriktionen der GPL)
- für alle Plattformen erhältlich ⇒ Skripte portabel
- im Unterschied zum älteren Perl ist Python leicht lernbar und fremder Code verständlich

Wo bekommt man Python?

- bei MacOS X und Linux dabei
- für andere Plattformen von http://www.python.org/

Beipiele für Python-Einsatz

- Build-Management Tool *Scons* (www.scons.org)
- Web Application-Server *Zope* (www.zope.org)
- *Gamera* Framework (gamera.sf.net)

Einführung in Python (2)

Programme in *Scriptsprachen* können nicht direkt vom OS ausgeführt werden, sondern brauchen eine Laufzeitumgebung, den *Script-Interpreter*.

- Aufruf des Pythonscripts *script*:
  `python script arg1 arg2 ...`
  arg1, ... sind Kommandozeilenargumente für *script*
- Unter Windows Verzeichnis mit Python-Interpreter an Umgebungsvariable *PATH* anhängen
- Unter MacOS X oder Linux kann *script* direkt ausführbar gemacht werden:
  - als erste Zeile des Scripts einfügen: `#!/usr/bin/python`
  - Script ausführbar machen mit: `chmod +x script`

Einführung in Python (3)

Allgemeine Syntax:

- Python ist casesensitiv
  Kommentare von '#' bis Zeilenende
  Kommando endet am Zeilenende
  - mehrere Kommandos pro Zeile mit ';' möglich
  - Kommando über mehrere Zeilen mit '\' am Zeilenende
- Blockbildung durch Einrücktiefe

```
if x > 2:
    x += 3   # if-Block, da tiefer eingerückt
print x      # if-Block beendet, da weniger eingerückt
```

  - TABs und Spaces dürfen nicht gemischt werden!
  - im Emacs am besten File-Variablen setzen:
    `# -*- mode: python; indent-tabs-mode: nil; py-indent-offset: 4; -*-`
- Umlaute in Scripts
  - bei Verwendung nicht 7-bit Zeichen Encoding in erster/zweiter Zeile angeben mit
    `# coding=latin-1` oder `# coding=utf-8`

Einführung in Python (4)

Datentypen und Variablen

- Variablen haben keinen expliziten Datentyp implizit durch Zuweisung spezifiziert:

```
a = 1        # a wird Integer
b = [2,3,0]  # b wird Liste von Integers
```

- wichtigste eingebaute Datentypen:
  - Numbers (int, float)
    - Achtung: verschiedene Arithmetik: $1 / 2 \neq 1.0 / 2.0$
    - Konvertierung mit Funktionen *int()* und *float()*
  - Strings
  - Lists (analog Arrays in C)
  - Dictionaries (Mapping Types)
    - Sammlung von Key/Value-Paaren
    - entspricht in C++ dem STL-Typ *map*

Einführung in Python (5)

Strings

- angeben mit einfachen (') oder doppelten (") Quotes
  Escape-Sequencen mit Backslash wie in C

```
a = "Rock\'n Roll"
b = "Newline ist \n und Backlash \\"
```

- Zugriff auf einzelne Zeichen wie in Listen

```
firstchar = a[0]
substring = a[2:5]  # Teilstring Position 3 bis 6
```

- Operator == zum Vergleich und + zur Konkatenation

```
if a == "bla":
    a += "fasel"  # anhängen von "fasel" ans Ende von a
```

- Zahlreiche eingebaute Methoden, z.B.

```
t = s.replace("old","new")
t = s.strip()     # entfernt white space vorne und am Ende
occurence = s.find("bla") # liefert -1 wenn nicht gefunden
a = s.split(";")  # zerlegt CSV String in Liste der Felder
```


Einführung in Python (6)

Listen (1)

- Spezialfall des Typs *Sequence.*
  Anlegen mit eckigen Klammern:

```
a = [1,2,6]
b = []      # leere Liste
```

- Elementzugriff über Index-Operator (eckige Klammern)

```
first = a[0]
last = a[-1]
slice = a[2:4]  # Teilliste von Position 3 bis 5
slice = a[2:]   # Teilliste von Position 3 bis Ende
```

- Demonstration wichtiger Methoden

```
a = [2,1,3]
len(a)       # Länge von a (hier 3)
a.append(7)  #  a == [2,1,3,4]
del a[2]     #  a == [2,1,4]
a.sort()     #  a == [1,2,4]
```


Einführung in Python (7)

Listen (2)

- Python-Besonderheit ist die *List Comprehension*
  zur Erzeugung neuer Listen aus alten:

```
a = [2,1,3]
b = [2*x for x in a]   # b == [4,2,6]
```

  dasselbe auf herkömmlichem Weg über *for*-Schleife:

```
a = [2,1,3]
b = []
for x in a:
    b.append(2*x)
```

- List Comprehension kann optional Bedingung enthalten:

```
a = [2,1,3]
b = [2*x for x in a if x < 3]   # b == [4,2]
```


Einführung in Python (8)

Dictionaries

- Anlegen mit *key: value* Paaren in geschweiften Klammern:

```
a = {"gruen": 2, "rot": 1, "gelb": 2}
b = {}    # leeres Dictionary
```

- Elementzugriff über Key in eckigen Klammern

```
a["gruen"] = 5     # setzt Wert von "gruen"
n1 = a["gruen"]    # liest Wert von "gruen"
# Exception, wenn Key nicht vorhanden, kann vermieden werden
# indem Existenz des Keys vorher abgefragt wird:
if a.has_key("gruen"):
    a["gruen"] += 1
else:
    a["gruen"] = 1
```

- wichtige Methoden:

```
del a["gruen"]           # entfernt Eintrag "gruen"
for k in a.iterkeys():   # Iteration über alle Einträge
                         # Reihenfolge ist dabei zufällig!
```

Einführung in Python (9)

Kontrollfluss (1)

- Bedingte Verzweigung mit *if* und optional *elsif* und *else*

```
if x > y:
    maximum = x
else:
    maximum = y
```

- Kombination von Bedingungen mit den booleschen Operatoren *and, or* und *not*

- unbegrenzte Schleifen mit *while*

```
while d < epsilon:
    d = (f(x+h) - f(x)) / h
    h = 0.5*h
```

- Sprunganweisungen in Schleifen:
  - *break* zum Beenden (innerster) Schleife
  - *continue* für Sprung zum nächsten Schleifenstart

Einführung in Python (10)

Kontrollfluss (2)

- Schleifen fester Länge mit *for*
  geht nur über Objekt mit implementiertem Iterator (z.B. Liste)

```
a = [1,4,5]
for x in a:
    print x
```

- um Zahlenbereich zu durchlaufen, diesen Bereich als Liste mittels *range()* erzeugen

```
range(5)       # erzeugt [0,1,2,3,4]
range(2,5)     # erzeugt [2,3,4]
range(4,10,2)  # erzeugt [4,6,8] (Schrittweite ist 2)
```

- über Index und Wert einer Liste iteriert *enumerate()*

```
a = [2,5,1]
for i,x in enumerate(a):
    print (x == a[i])  # gibt jedes mal True aus
```


Einführung in Python (11)

Input und Output

- *open()* öffnet Datei und gibt File-Objekt zurück

```
f = open("bla")     # öffnet "bla" zum Lesen
f = open("bla","r") # öffnet "bla" zum Lesen (wie oben)
f = open("bla","w") # öffnet "bla" zum Schreiben ("w+" zum Anhängen)
```

- wichtige Methoden für File-Objekte:

```
f.close()           # schließt Datei
s = f.readline()    # liest Zeile (incl. '\n') in String s
a = f.readlines()   # liest alle Zeilen in Array a
f.write(s)          # schreibt String s in Datei
f.flush()           # leert Output-Puffer (sinnvoll bei Logfiles)
```

- über File-Objekte kann iteriert werden (zeilenweises Lesen)

```
f.open("bla")
for line in f:
    line = line.rstrip("\n\r") # entferne Newlines etc. am Ende
    # do some stuff with line
```


Einführung in Python (12)

Vordefinierte Datei-Objekte

- Modul *sys* definiert *stdin, stdout* und *stderr*

```
import sys    # vorher Modul importieren
sys.stderr.write("Dummer Fehler aufgetreten!\n")
```

- eingebaute Funktion *print* gibt nach stdout aus. Besonderheiten:
  - *print* fügt am Ende Zeilenumbruch ein
  - nicht-Strings werden automatisch formatiert
  - Funktionsaufruf ohne Klammern möglich, z.B. `print "Nr:", n`

Formatierte Ausgabe

- Formatierungen mit '%' wie bei C-Funktion printf() möglich, aber viel allgemeiner: % ist ein Operator auf Strings!

```
s = ("Nr: %05d" % n) # gibt formatierten String an s zurück
```

- bei mehreren Formatvariablen Argumente als Tupel (Klammern)

```
print "Nr: %05d  Wert: %5.2f" % (n,f)
```

Einführung in Python (13)

Kommandozeilenargumente

- stehen in Liste *argv* des Moduls *sys* (wobei sys.argv[0] Scriptname)
- Beispiel für Parsen der Kommandozeile:

```
import sys
i = 1
while i < len(sys.argv):
    if sys.argv[i] == "-o":
        i += 1
        opt_outfile = sys.argv[i]
    elif sys.argv[i] == "-n":
        opt_n = True
    elif sys.argv[i][0] == "-":
        sys.stderr.write("Falsche Option: %s\n" % sys.argv[i])
        sys.exit(1)
    else   # Option ohne '-' sei Inputdatei
        opt_infile = sys.argv[i]
```


Einführung in Python (14)

Module

- Bibliotheken werden mit *import* geladen. Varianten:
  - *import sys* lädt alle Funktionen aus sys in Namespace "sys"
  - ⇒ Zugriff über Namespace-Qualifier "sys" nötig, z.B. *sys.argv*
  - *from sys import exit* lädt nur die Funktion exit aus sys in aktuellen Namespace
  - *from sys import ** lädt alle Funktionen aus sys in aktuellen Namespace
- Mehrere Module gleichzeitig laden durch Kommas getrennt

```
import sys, os, re
```

- Suchpfad für Module steht in *sys.path*
- wichtige Module (siehe Python-Doku für mehr Info):
  - *os* und *os.path*: Systemnahe Funktionen wie Dateieigenschaften, anonyme Pipes etc.
  - *re*: Suchen und Ersetzen regulärer Ausdrücke (vgl. THI)
  - *math*: mathematische Funktionen (z.B. cos) und Konstanten (z.B. pi)
  - *time*: Funkiton time.time() gibt Zeitstempel zurück (für Laufzeitmessungen)

Einführung in Python (15)

Benutzerdefinierte Funktionen

- werden definiert mit *def*

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

- auch mehr als ein Returnwert möglich

```
def max_und_min(a):
    return (max(a), min(a))
(a1,a2) = max_und_min([9,2,5,8])   # Zuweisung Ergebnis an a1 und a2
```

- Aufruf auch über "Keyword-Arguments" möglich
  ⇒ Reihenfolge wird egal beim Aufruf

```
def potenz(basis,exponent):
    return basis**exponent
x = potenz(exponent=2, basis=3)
```

- Default-Werte möglich wie in C++:

```
def potenz(basis,exponent=2):
    return basis**exponent
x = potenz(4)  # berechnet 4**2
```


Einführung in Python (16)

Benutzerdefinierte Klassen

- werden definiert mit *class*. Beispiel:

```
class Linie:
    def __init__(self,p1,p2):  # Konstruktor
        self.point1 = p1
        self.point2 = p2
    def angle(self):           # Member-Funktion
        dy = float(p2.y - p1.y)
        dx = float(p2.x - p1.x)
        return math.atan(dy/dx) * 180 / math.pi
```

- Beispiel für Benutzung:

```
linie = Linie(Point(3,4), Point(10,20))
print "Winkel: %f" % linie.angle()
```

- Erläuterungen:
  - erstes Argument in Memberdefinition ist Klasseninstanz (entspricht "this" in C++)
  - kann beliebig genannt werden; typische Kovention ist "self"
  - es gibt keine privaten Methoden, sondern nur die Konvention
  - private Methoden mit zwei Unterstrichen beginnen zu lassen