# Python-Crashkurs

Andreas Abel

Lehrstuhl für Theoretische Informatik
Ludwig-Maximilians-Universität München

Wintersemester 2008/09
6.-10. Oktober 2008

# Organisatorisches

- Zeit: Mo-Fr 6-10. Oktober 2008, jeweils 10.15-11.45 und 13.15-14.45
- Raum: Oe 1.05
- Schein durch Bearbeitung einer Programmieraufgabe am Ende des Kurses

# Inhalt

# Ressourcen

- `www.python.org`: Offizielle Webseite.
- Dieser Kurs folgt *Guido van Rossum*'s Tutorial.
- Stabile Version: 2.6 (1. Oktober 2008)
- Nächste Version: 3.0
- Implementiert in C (CPython)

# Python

- Python ist benannt nach *Monty Python's Flying Circus*
- Objektorientierte Sprache für Prototyp-Entwicklung
- Skriptsprache
- Elegant, leicht zu lernen und lesen
- Open source
- Hochportabel
- Erste Version 1990

# Laufzeitverhalten

- Kompiliert nach Byte-Code, der dann interpretiert wird
- Kompilation erfolgt automatisch
- Automatische Speicherverwaltung durch *reference counting*
- Keine unkontrollierten Abstürze *(seg faults)*

# Spracheigenschaften

- Alles ist ein Objekt
- Klassen und Mehrfachvererbung
- Funktionen höherer Ordnung (wie in Scheme)
- Dynamische Typisierung und Polymorphismus
- Ausnahmebehandlung wie in JAVA *(exceptions)*
- Statische Sichtbarkeit und Module
- Operatorüberladung
- Blockstruktur durch Einrücken (wie in Haskell)

# Datentypen

- Zahlen: `int`, `long`, `float`, `complex`
- Zeichenketten (unveränderlich wie JAVA `String`)
- Tupel, Listen, endliche Abbildungen *(dictionaries)*
- Erweiterungsmodule können neue Datentypen definieren
- Beliebige Datenstrukturen durch Klassen

# Warum und wofür Python?

- Kode $2-10\times$ kürzer als C, C++, JAVA
- Kode ist gut lesbar
- *rapid prototyping*
- *web scripting*
- Wissenschaftliche Anwendungen
- XML, Datenbanken, graphische Oberflächen
- *content management* (Zope/Plone)
- GNU Mailman
- JPython

# Python starten

- Interaktive Python-Sitzung: `python`
- Ende mit *eof* (Unix: Ctrl-D, Windows: Ctrl-Z)
- Oder: `import sys; sys.exit()`
- Skript ausführen: `python myfile.py`

  ```
  python ..python-args.. script.py ..script-args..
  ```
- Python-Befehl direkt ausführen

  ```
  python -c "print 5*6*7"
  python -c "import sys; print sys.maxint"
  python -c "import sys; print sys.argv" 1 2 3 4
  ```
- Ausführbares Python-Skript

  ```
  #!/usr/bin/env python
  # -*- coding: iso-8859-15 -*-
  ```

# Ganzzahl-Arithmetik

- `>>>` ist die Python-Eingabeaufforderung.

```
>>> 2+2  # A comment on the same line as code.
4
>>> # A comment; Python asks for a continuation ...
... 2+2
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

# Große Ganzzahlen

- `int` speichert ein vorzeichenbehaftetes Wort (32/64 Bit).

```
>>> import sys; sys.maxint
2147483647
```

- `long` kann beliebig große Ganzzahlen aufnehmen.

```
>>> sys.maxint + 1
2147483648L
>>> 2 ** 100
1267650600228229401496703205376L
```

- Konversion. [“_” ist Platzhalter für letzten Wert.]

```
>>> - 2 ** 31
-2147483648L
>>> int(_)
-2147483648
```

# Zuweisung

- Variablen müssen nicht deklariert werden.

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

- Parallele Zuweisung.

```
>>> width, height = height, width + height
```

- Abkürzende Schreibweise für Mehrfachzuweisung.

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> z
0
```

# Fließkommazahlen

- Arithmetische Operationen sind überladen.
- Ganzzahlen werden bei Bedarf konvertiert.

```
>>> 3 * 3.75 / .5
22.5
>>> 7. / 2
3.5
>>> float(7) / 2
3.5
```

- Exponentschreibweise: `1e0 1.0e+1 1e-1 .1e-2`
- Typisch sind 53 Bit Genauigkeit (wie `double` in C).

```
>>> 1e-323
9.8813129168249309e-324
>>> 1e-324
0.0
```

# Weitere arithmetische Operationen

- Restberechnung:

```
>>> 4 % 3
1
>>> -4 % 3
2
>>> 4 % -3
-2
>>> -4 % -3
-1
>>> 3.9 % 1.3
1.2999999999999998
```

- Abrundende Division:

```
>>> 7.0 // 4.4
1.0
```

# Komplexe Zahlen

- Imaginäre Zahlen tragen das Suffix `j`.

```
>>> 1j * complex(0,1)
(-1+0j)
>>> complex(-1,0) ** 0.5
(6.123031769111886e-17+1j)
```

- Real- und Imaginärteil:

```
>>> a=1.5+0.5j
>>> a.real + a.imag
2.0
```

- Absolutwert funktioniert auch für `complex`.

```
>>> abs(3 + 4j)
5.0
```

# Bit-Operationen

- Links- (<<) und Rechtsschiebung (>>)

```
>>> 1 << 16
65536
```

- Bitweises Und (&), Oder (|), Xor (^) und Nicht (~).

```
>>> 1000 & 0377
232
>>> 0x7531 | 0x8ace
65535
>>> ~0
-1
>>> 0123 ^ 0123
0
```

# Zeichenketten

- Typ `str`.
- Einfache oder doppelte Anführungszeichen.

```
Eingabe                          ergibt
-------                          ---------
'Python tutorial'                'Python tutorial'
'doesn\'t'                       "doesn't"
"doesn't"                        "doesn't"
'"Yes," he said.'                '"Yes," he said.'
"\"Yes,\" he said."              '"Yes," he said.'
'"Isn\'t," she said.'            '"Isn\'t," she said.'
```

# Escape-Sequenzen

| | |
|---|---|
| `\\` | backslash |
| `\'` | single quote |
| `\"` | double quote |
| `\t` | tab |
| `\n` | newline |
| `\r` | carriage return |
| `\b` | backspace |

# Mehrzeilige Stringkonstanten

- Der Befehl

```
print "This is a rather long string containing\n\
several lines of text as you would do in C.\n\
    Whitespace at the beginning of the line is\
 significant."
```

- ergibt

```
This is a rather long string containing
several lines of text as you would do in C.
    Whitespace at the beginning of the line is sign
```

# Dreifach-Anführungszeichen

- Mehrzeilige Zeichenketten mit Zeilenumbrüchen:

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

- ergibt

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

# Roh-Zeichenketten

- Ein vorgeschobenes `r` erhält die Escape-Sequenzen.

```
>>> print "Hello! \n\"How are you?\""
Hello!
"How are you?"
>>> print r"Hello! \n\"How are you?\""
Hello! \n\"How are you?\"
```

- Roh-Zeichenketten haben auch Typ `str`.

```
>>> type ("\n")
<type 'str'>
>>> type (r"\n")
<type 'str'>
```

# Unicode

- Unicode-Zeichenketten (eigener Typ) beginnen mit `u`.

```
>>> print u"a\u0020b"
a b
>>> u"ö"
u'\xf6'
>>> type (_)
<type 'unicode'>
```

- Normale Zeichenketten werden bei Bedarf nach Unicode konvertiert.

```
>>> "this " + u"\u00f6" + " umlaut"
u'this \xf6 umlaut'
>>> print _
this ö umlaut
```

# Zeichenkettenoperationen

```
"hello"+"world"     "helloworld"        # concat.
"hello"*3           "hellohellohello"   # repetition
"hello"[0]          "h"                 # indexing
"hello"[-1]         "o"                 # (from end)
"hello"[1:4]        "ell"               # slicing
len("hello")        5                   # size
"hello" < "jello"   True                # comparison
"e" in "hello"      True                # search
```

# Listen

- Listen sind veränderbare Arrays.

```
a = [99, "bottles of beer", ["on", "the", "wall"]]
```

- Zeichenkettenoperationen funktionieren auch auf Listen.

```
a+b, a*3, a[0], a[-1], a[1:], len(a)
```

- Einträge und Abschnitte können neu belegt werden.

```
a[0] = 98
a[1:2] = ["bottles", "of", "beer"]
              # -> [98, "bottles", "of", "beer",
                             ["on", "the", "wall"]]
del a[-1] # -> [98, "bottles", "of", "beer"]
```

# Weitere Listenoperationen

```
>>> a = range(5)                # [0,1,2,3,4]
>>> a.append(5)                 # [0,1,2,3,4,5]
>>> a.pop()                     # [0,1,2,3,4]
5
>>> a.insert(0, 42)             # [42,0,1,2,3,4]
>>> a.pop(0)                    # [0,1,2,3,4]
42
>>> a.reverse()                 # [4,3,2,1,0]
>>> a.sort()                    # [0,1,2,3,4]
```

# While

- Drucke die Fibonacci-Zahlen bis 100

```
>>> a, b = 0, 1
>>> while b <= 100:
...         print b
...         a, b = b, a+b
...
```

- Vergleichsoperatoren: `==` `<` `>` `<=` `>=` `!=`
- Einrückung beginnt Block
- Ausrückung beendet Block
- Zweizeiler:

```
>>> a, b = 0, 1
>>> while b <= 100: print b; a,b = b, a+b
...
```

# If

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
     x = 0
     print 'Negative changed to zero'
elif x == 0:
     print 'Zero'
elif x == 1:
     print 'Single'
else:
     print 'More'
```

- `elif` statt `else if`, um weitere Einrückung zu vermeiden.

# For

- `for` iteriert über eine Sequenz (e.g. Liste, Zeichenkette)

```
a = ['cat', 'window', 'defenestrate']
for x in a:
    print x, len(x)
```

- Die iterierte Sequenz darf im Rumpf der Schleife nicht modifiziert werden. Man kann jedoch eine Kopie erstellen, z.B. mit der Abschnitts-Notation.

```
for x in a[:]:
    if len(x) > 6: a.insert(0,x)
print a
```

- Ergibt

```
['defenestrate', 'cat', 'window', 'defenestrate'].
```

# Bereichsfunktion

- Iteration über eine Zahlenfolge leicht mittels `range()`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

- Iteration über die Indizes einer Sequenz:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print i, a[i]
```

# For-/While-Schleifen: `break`, `continue`, `else`

- `break` (wie in C), beendet die umschliessende Schleife vorzeitig
- `continue` (wie in C), beginnt sofort mit der nächsten Iteration.
- Der `else`-Teil einer Schleife wird nur dann ausgeführt, wenn die Schleife nicht durch `break` abgebrochen wurde.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print n, 'equals', x, '*', n/x
            break
    else: # loop completed, no factor
        print n, 'is a prime number'
```

# Die leere Anweisung

- Die Anweisung `pass` tut nichts.

```
while True:
    pass # Busy-wait for keyboard interrupt
```

- Wird verwendet, wenn eine Anweisung syntaktisch notwendig ist, aber nichts getan werden soll.

# Prozeduren

- Prozeduren werden mit dem Schlüsselwort `def` deklariert.

```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
```

- Variablen `n`, `a`, `b` sind lokal.
- Der Rückgabewert ist `None`.

```
print fib(10)
```

# Die Prozedur als Objekt

- Prozeduren sind selber Werte.

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

# Call-by-value

- Zuweisungen an Funktionsparameter sind lokal.

```
def bla(l):
    l = []

l = ['not', 'empty']
bla(l)
print l
```

- `l` ist eine Referenz auf ein Objekt.
- Das referenzierte Objekt kann jedoch geändert werden

```
def exclamate(l):
    l.append('!')

exclamate(l)
print l
```

# Globale Variablen

- Zugriff auf globale Variablen muss explizit deklariert werden.

```
def clear_l():
    global l
    l = []

l = ['not', 'empty']
clear_l()
print l
```

- ...druckt die leere Liste.

# Ergebnisrückgabe

- Mit `return` kann die Prozedur vorzeitig beendet werden.
- Mit `return ...value...` wird ein Ergebnis zurückgegeben.

```
def fib2(n):
    """Return the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)    # see below
        a, b = b, a+b
    return result

f100 = fib2(100)     # call it
f100                 # write the result
```

# Standardwerte für Funktionsparameter

- Bei der Funktionsdefinition können Defaults für Parameter angegeben werden.

```
def ask(prompt, retries=4, complaint='Yes/no?'):
  while True:
    ok = raw_input(prompt)
    if ok in ('y', 'ye', 'yes'): return True
    if ok in ('n', 'no'): return False
    retries -= 1
    if retries < 0: raise IOError, 'refused'
    print complaint
```

- Beim Aufruf kann man Argumente mit Defaults weglassen.

```
ask ("Continue (y/n)?", 3, "Yes or no, please!")
ask ("Continue (y/n)?", 3)
ask ("Continue (y/n)?")
```

# Standardwerte für Funktionsparameter

- Fehlerhaft:

```
ask ("Continue (y/n)?", "Yes or no, please!")
ask ()
```

- Benannte Argumente (*keyword arg*) helfen hier weiter:

```
ask ("Continue (y/n)?", complaint="Yes or no?")
ask (prompt="Continue (y/n)?")
```

- Fehlerhaft:

```
ask (prompt="Continue (y/n)?", 5)
ask ("Yes/no?", prompt="Continue (y/n)?")
```

# Auswertung der Standardwerte

- Standardwerte werden nur einmal, bei Funktionsdefinition, ausgewertet.

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

- Welche Zahl wird gedruckt?

# Auswertung der Standardwerte

- Vorsicht mit veränderbaren Objekten!

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
```

- ... druckt `[1]` und `[1, 2]`. Stattdessen:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

# Argumentlisten

- Mit $\star$ vor dem Parameter kann eine flexible Anzahl von Argumenten aufgenommen werden.

```
def fprintf(file, format, *args):
    file.write(format % args)
```

- Eine Liste kann mit $\star$ als einzelne Argumente übergeben werden.

```
>>> args = [3, 6]
>>> range(*args)
[3, 4, 5]
```

# Dok-Strings

- Die erste Anweisung einer Funktion kann ein String sein.

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass
```

- Erste Zeile: Zusammenfassung (Beginn: Großbuchst., Ende: Punkt).
- Dann weitere erläuternde Absätze.
- Wird mit `.__doc__` oder `help` abgefragt.

```
my_function.__doc__    # return doc string
help(my_function)      # print  doc string
```

# Anonyme Funktionen

- Eine Funktion kann als Ausdruck angegeben werden.

```
>>> lambda x, y: x
<function <lambda> at 0xb77900d4>
```

- Hier eine Fabrik für Erhöher:

```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```

- Funktionen werden durch Adresse verglichen:

```
>>> (lambda x: x) == (lambda x: x)
False
```

# Übungsaufgabe: Sudoko

- Beim Sudoku muss man jedes Feld eines 9x9-Quadrats mit einer der Ziffern 1-9 füllen.
- Das 9x9-Quadrat ist in 9 3x3-Unterquadrate aufgeteilt.
- In jeder Zeile, jeder Spalte und jedem Unterquadrat muss jede Ziffer genau einmal vorkommen.
- Aufgaben:
  1. Überlegen Sie sich eine Repräsentation eines teilweise ausgefüllten Sudokorätsels.
  2. Schreiben Sie eine Funktion, die für ein Feld berechnet, welche Ziffern dort noch Platz hätten.
  3. Schreiben Sie eine Funktion, die prüft, ob ein Rätsel (noch) lösbar ist.