

Computergrundlagen Programmieren in Python

Axel Arnold

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2010/11

Python



- schnell zu erlernende Programmiersprache
– tut, was man erwartet
- objektorientierte Programmierung ist möglich
- viele Standardfunktionen („all batteries included“)
- breite Auswahl an Bibliotheken
- freie Software mit aktiver Gemeinde
- portabel, gibt es für fast jedes Betriebssystem
- entwickelt von Guido van Rossum, CWI, Amsterdam

Informationen zu Python

- aktuelle Versionen 3.1 bzw. 2.7
- 2.x ist *noch* weiter verbreitet (z.B. Python 2.6 im CIP-Pool)
- diese Vorlesung behandelt daher noch 2.x
- aber längst nicht alles, was Python kann

Hilfe zu Python

- offizielle Homepage
`http://www.python.org`
- Einsteigerkurs „A Byte of Python“
`http://swaroopch.com/notes/Python` (englisch)
`http://abop-german.berlios.de` (deutsch)
- mit Programmiererfahrung „Dive into Python“
`http://diveintopython.org`

Python starten

Aus der Shell:

```
> python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more...
>>> print "Hello World"
Hello World
>>> help("print")
>>> exit()
```

- >>> markiert Eingaben
- print: Ausgabe auf das Terminal
- help(): interaktive Hilfe, wird mit "q" beendet
- statt exit() reicht auch Control-d
- oder ipython mit Tab-Ergänzung, History usw.

Python-Skripte

als Python-Skript helloworld.py:

```
#!/usr/bin/python
```

```
# unsere erste Python-Anweisung  
print "Hello World"
```

- mit `python helloworld.py` starten
- oder ausführbar machen (`chmod a+x helloworld.py`)
- **Umlaute vermeiden** oder Encoding-Cookie einfügen
- „#!“ funktioniert genauso wie beim Shell-Skript
- Zeilen, die mit „#“ starten, sind Kommentare

**Kommentare sind wichtig,
um ein Programm verständlich machen!**

- und nicht, um es zu verlängern!

Datentypen 1

- ganze Zahlen

```
>>> print 42
```

```
42
```

```
>>> print -12345
```

```
-12345
```

- Fließkommazahlen

```
>>> print 12345.000
```

```
12345.0
```

```
>>> print 6.023e23
```

```
6.023e+23
```

```
>>> print 13.8E-24
```

```
1.38e-23
```

- $1.38e-23$ steht z. B. für 1.38×10^{-23}
- $12345 \neq 12345.0$ (z. B. bei der Ausgabe)

Datentypen 2

- Zeichenketten (Strings)

```
>>> print "Hello World"
Hello World
>>> print 'Hello World'
Hello World
>>> print """Hello
... World"""
Hello
World
```

- zwischen einfachen (') oder doppelten (") Anführungszeichen
- Über mehrere Zeilen mit dreifachen Anführungszeichen
- Leerzeichen sind normale Zeichen!
`"Hello World" ≠ "Hello World"`
- Zeichenketten sind keine Zahlen! `"1" ≠ 1`

Variablen

```
>>> number1 = 1
>>> number2 = number1 + 5
>>> print number1, number2
1 6
>>> number2 = "Keine Zahl"
>>> print number2
Keine Zahl
```

- Variablennamen bestehen aus Buchstaben, Ziffern oder „_“ (Unterstrich)
- am Anfang keine Ziffer
- Groß-/Kleinschreibung ist relevant: Hase \neq hase
- **Richtig:** i, some_value, SomeValue, v123, _hidden, _1
- **Falsch:** 1_value, some□value, some-value

Arithmetische Ausdrücke

+	Addition, bei Strings aneinanderfügen, z.B. $1 + 2 \rightarrow 3$, $"a" + "b" \rightarrow "ab"$
-	Subtraktion, z.B. $1 - 2 \rightarrow -1$
*	Multiplikation, Strings vervielfältigen, z.B. $2 * 3 = 6$, $"ab" * 2 \rightarrow "abab"$
/	Division, bei ganzen Zahlen ganzzahlig, z.B. $3 / 2 \rightarrow 1$, $-3 / 2 \rightarrow -2$, $3.0 / 2 \rightarrow 1.5$
%	Rest bei Division, z.B. $5 \% 2 \rightarrow 1$
**	Exponent, z.B. $3**2 \rightarrow 9$, $.1**3 \rightarrow 0.001$
&, , ^	bitweises und, oder und exklusiv-oder

- mathematische Präzedenz (Exponent vor Punkt vor Strich),
z. B. $2**3 * 3 + 5 \rightarrow 2^3 \cdot 3 + 5 = 29$

Logische Ausdrücke

<code>==, !=</code>	Test auf (Un-)Gleichheit, z.B. $2 == 2 \rightarrow \text{True}$, $1 == 1.0 \rightarrow \text{True}$, $2 != 1 \rightarrow \text{False}$
<code><, >, <=, >=</code>	Vergleich, z.B. $2 > 1 \rightarrow \text{True}$, $1 <= -1 \rightarrow \text{False}$
<code>or, and</code>	Logische Verknüpfungen „oder“ bzw. „und“
<code>not</code>	Logische Verneinung, z.B. $\text{not False} == \text{True}$

- Vergleiche liefern Wahrheitswerte: **True** oder **False**
- Wahrheitstabelle für die logische Verknüpfungen:

<i>a</i>	<i>b</i>	<i>a und b</i>	<i>a oder b</i>
True	True	True	True
False	True	False	True
True	False	False	True
False	False	False	False

- Präzedenz: logische Verknüpfungen vor Vergleichen
- Beispiele: $3 > 2 \text{ and } 5 < 7 \rightarrow \text{True}$, $1 < 1 \text{ or } 2 >= 3 \rightarrow \text{False}$

if: bedingte Ausführung

```

>>> a = 1
>>> if a < 5:
...     print "a ist kleiner als 5"
... elif a > 5:
...     print "a ist größer als 5"
... else:
...     print "a ist 5"
a ist kleiner als 5
>>> if a < 5 and a > 5:
...     print "Das kann nie passieren"
  
```

- if-elif-else führt den *Block* nach der ersten erfüllten Bedingung (logischer Wert True) aus
- Trifft keine Bedingung zu, wird der else-Block ausgeführt
- elif oder else sind optional

Blöcke und Einrückung

```
>>> a=5
>>> if a < 5:
...     # wenn a kleiner als 5 ist
...     b = -1
... else: b = 1
>>> # aber hier geht es immer weiter
... print b
1
```

- Alle *gleich eingerückten* Befehle gehören zum Block
- Nach dem `if`-Befehl geht es auf Einrückungsebene des `if` weiter, egal welcher `if`-Block ausgeführt wurde
- Einzeilige Blöcke können auch direkt hinter den Doppelpunkt
- Einrücken durch Leerzeichen oder Tabulatoren (einfacher)

Blöcke und Einrückung 2

- ein Block kann nicht leer sein, aber der Befehl `pass` tut nichts:

```
if a < 5:
    pass
else:
    print "a ist groesser gleich 5"
```

- bei nicht gleichmäßiger Einrückung gibt es einen `IndentationError`

```
>>> print "Hallo"
Hallo
```

```
>>>     print "Hallo"
```

```
File "<stdin>", line 1
    print "Hallo"
    ^
```

IndentationError: unexpected indent

- Falsche Einrückung führt im allgemeinen zu Programmfehlern!

while: Schleifen

```
>>> a = 1
>>> while a < 5:
...     a = a + 1
>>> print a
5
```

- Führt den Block solange aus, wie die Bedingung wahr ist
- kann auch nicht ausgeführt werden:

```
>>> a = 6
>>> while a < 5:
...     a = a + 1
...     print "erhoehe a um eins"
>>> print a
6
```

for: Sequenz-Schleifen

```

>>> for a in range(1, 3): print a
1
2
>>> b = 0
>>> for a in range(1, 100):
...     b = b + a
>>> print b
4950
>>> print 100 * (100 - 1) / 2
4950
  
```

- `for` führt einen Block für jedes Element einer *Sequenz* aus
- Das aktuelle Element steht in `a`
- `range(k, l)` ist eine Liste der Zahlen a mit $k \leq a < l$
- später lernen wir, Listen zu erstellen und verändern

break und continue: Schleifen beenden

```

>>> for a in range(1, 10):
...     if a == 2 or a == 4 or a == 6: continue
...     elif a == 5: break
...     print a
1
3
>>> a = 1
>>> while True:
...     a = a + 1
...     if a > 5: break
>>> print a
6
  
```

- beide überspringen den Rest des Schleifenkörpers
- break bricht die Schleife ganz ab
- continue springt zum Anfang

Funktionen

```

>>> def printPi():
...     print "pi ist ungefaehr 3.14159"
>>> printPi()
pi ist ungefaehr 3.14159

>>> def printMax(a, b):
...     if a > b: print a
...     else:     print b
>>> printMax(3, 2)
3
  
```

- eine Funktion kann beliebig viele Argumente haben
- Argumente sind Variablen
- Beim Aufruf bekommen die Argumentvariablen Werte in der Reihenfolge der Definition
- Der Funktionskörper ist wieder ein Block

return: eine Funktion beenden

```
def printMax(a, b):
    if a > b:
        print a
    return
print b
```

- return beendet die Funktion sofort

Rückgabewert

```
>>> def max(a, b):
...     if a > b: return a
...     else:     return b
>>> print max(3, 2)
3
```

- eine Funktion kann einen Wert zurückliefern
- der Wert wird bei return spezifiziert

Lokale Variablen

```
>>> def max(a, b):  
...     if a > b: maxVal=a  
...     else:     maxVal=b  
...     return maxVal  
>>> print max(3, 2)  
3  
>>> print maxVal  
NameError: name 'maxVal' is not defined
```

- Variablen innerhalb einer Funktion sind *lokal*
- lokale Variablen verschwinden nach der Funktionsausführung
- globale Variablen können aber gelesen werden

```
>>> faktor=2  
>>> def strecken(a): return faktor*a  
>>> print strecken(1.5)  
3.0
```

Vorgabewerte und Argumente benennen

```
>>> def lj(r, epsilon = 1.0, sigma = 1.0):
...     return 4*epsilon*((sigma/r)**6 - (sigma/r)**12 )
>>> print lj(2**(1./6.))
1.0
>>> print lj(2**(1./6.), 1, 1)
1.0
```

- Argumentvariablen können mit Standardwerten vorbelegt werden
- diese müssen dann beim Aufruf nicht angegeben werden

```
>>> print lj(r = 1.0, sigma = 0.5)
0.0615234375
>>> print lj(epsilon=1.0, sigma = 1.0, r = 2.0)
0.0615234375
```

- beim Aufruf können die Argumente auch explizit belegt werden
- dann ist die Reihenfolge egal