

# Computergrundlagen Programmieren in Python

**Axel Arnold**

Institut für Computerphysik  
Universität Stuttgart

Wintersemester 2011/12



- 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

- aktuelle Versionen 3.2 bzw. 2.7.2
- 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.net>

## 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.

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!

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

- 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`

```
>>> 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



+	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. $3**2 \rightarrow 9$ , $.1**3 \rightarrow 0.001$

- mathematische Präzedenz (Exponent vor Punkt vor Strich),  
z. B.  $2**3 * 3 + 5 \rightarrow 2^3 \cdot 3 + 5 = 29$
- Präzedenz kann durch runde Klammern geändert werden:  
 $2**(3 * (3 + 5)) \rightarrow 2^{3 \cdot 8} = 16,777,216$

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

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

<i>a</i>	<i>b</i>	<i>a</i> und <i>b</i>	<i>a</i> oder <i>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 and 5 < 7`  $\rightarrow$  True,  
`1 < 1 or 2 >= 3`  $\rightarrow$  False

```
>>> a = 1
>>> if a < 5:
...     print "a ist kleiner als 5"
... elif a > 5:
...     print "a ist groesser 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

```
>>> 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)

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

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

---

- **IndentationError** bei ungleichmäßiger Einrückung:

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

IndentationError: unexpected indent

---

- Falsche Einrückung führt im allgemeinen zu Programmfehlern!

```
>>> 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 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

```
>>> 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



```
>>> 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 der Funktion
- Beim Aufruf bekommen die Argumentvariablen Werte in der Reihenfolge der Definition
- Der Funktionskörper ist wieder ein Block

```
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

```
>>> 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 existieren nur während der Funktionsausführung
- globale Variablen können aber gelesen werden

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

```
>>> 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

```
def max(a, b):  
    "Gibt das Maximum von a und b zurueck."  
    if a > b: return a  
    else:     return b
```

```
def min(a, b):  
    ""
```

Gibt das Minimum von a und b zurueck. Funktioniert  
ansonsten genau wie die Funktion max.

```
    ""  
    if a < b: return a  
    else:     return b
```

- 
- Dokumentation optionale Zeichenkette vor dem Funktionskörper
  - wird bei `help(funktion)` ausgegeben

```
def printGrid(f, a, b, step):  
    """  
    Gibt x, f(x) an den Punkten  
    x= a, a + step, a + 2*step, ..., b aus.  
    """  
    x = a  
    while x < b:  
        print x, f(x)  
        x = x + step  
  
def test(x): return x*x  
  
printGrid(test, 0, 1, 0.1)
```

- Funktionen ohne Argumentliste „(..)“ sind normale Werte
- Funktionen können in Variablen gespeichert werden
- ... oder als Argumente an andere Funktionen übergeben werden

```
def fakultaet(n):  
    # stellt sicher, das die Rekursion irgendwann stoppt  
    if n <= 1:  
        return 1  
    #  $n! = n * (n-1)!$   
    return n * fakultaet(n-1)
```

- Funktionen können andere Funktionen aufrufen, insbesondere sich selber
- Eine Funktion, die sich selber aufruft, heißt **rekursiv**
- Rekursionen treten in der Mathematik häufig auf
- sind aber oft nicht einfach zu verstehen
- Ob eine Rekursion endet, ist nicht immer offensichtlich
- Jeder rekursive Algorithmus kann auch **iterativ** als verschachtelte Schleifen formuliert werden

- Komplexe Datentypen sind zusammengesetzte Datentypen
- Beispiel: Eine Zeichenkette besteht aus beliebig vielen Zeichen
- die wichtigsten komplexen Datentypen in Python:
  - Strings (Zeichenketten)
  - Listen
  - Tupel
  - Dictionaries (Wörterbücher)
- diese können als **Sequenzen** in **for** eingesetzt werden:

---

```
>>> for x in "bla": print "->", x
-> b
-> l
-> a
```

---



```
>>> kaufen = [ "Muesli", "Milch", "Obst" ]
>>> kaufen[1] = "Sahne"
>>> print kaufen[-1]
Obst
>>> kaufen.append(42)
>>> del kaufen[-1]
>>> print kaufen
['Muesli', 'Sahne', 'Obst']
```

- komma-getrennt in eckigen Klammern
- können Daten *verschiedenen* Typs enthalten
- `liste[i]` bezeichnet das *i*-te Listenelement, negative Indizes starten vom Ende
- `liste.append()` fügt ein Element an eine Liste an
- **del** löscht ein Listenelement

```
>>> kaufen = kaufen + [ "0e1", "Mehl" ]
>>> print kaufen
['Muesli', 'Sahne', 'Obst', '0e1', 'Mehl']
>>> for l in kaufen[1:3]:
...     print l
Sahne
Obst
>>> print len(kaufen[:4])
3
```

- „+“ fügt zwei Listen aneinander
- `[i:j]` bezeichnet die Subliste vom `i`-ten bis zum `j-1`-ten Element
- Leere Sublisten-Grenzen entsprechen Anfang bzw. Ende, also stets `liste == liste[:] == liste[0:]`
- `for`-Schleife über alle Elemente
- `len()` berechnet die Listenlänge

Shallow copy:

```
>>> bezahlen = kaufen
>>> del kaufen[2:]
>>> print bezahlen
['Muesli', 'Sahne']
```

Subliste, deep copy:

```
>>> merken = kaufen[1:]
>>> del kaufen[2:]
>>> print merken
['Sahne', 'Obst', 'Oel', 'Mehl']
```

„=" macht in Python flache Kopien komplexer Datentypen!

- Flache Kopien (shallow copies) verweisen auf dieselben Daten
- Änderungen an einer flachen Kopie betreffen auch das Original
- Sublisten sind echte Kopien
- daher ist `l[:]` eine echte Kopie von `l`

```
>>> elementliste=[]
>>> liste = [ elementliste, elementliste ]
>>> liste[0].append("Hallo")
>>> print liste
[['Hallo'], ['Hallo']]
```

### Mit echten Kopien (deep copies)

```
>>> liste = [ elementliste[:], elementliste[:] ]
>>> liste[0].append("Welt")
>>> print liste
[['Hallo', 'Welt'], ['Hallo']]
```

- komplexe Listenelemente sind flache Kopien und können daher mehrmals auf dieselben Daten verweisen
- kann zu unerwarteten Ergebnissen führen

```
>>> kaufen = ("Muesli", "Kaese", "Milch")
>>> print kaufen[1]
Kaese
>>> for f in kaufen[:2]: print f
Muesli
Kaese
>>> kaufen[1] = "Camembert"
TypeError: 'tuple' object does not support item assignment
>>> print k + k
('Muesli', 'Kaese', 'Milch', 'Muesli', 'Kaese', 'Milch')
```

- komma-getrennt in runden Klammern
- können nicht verändert werden
- ansonsten wie Listen einsetzbar
- Strings sind Tupel von Zeichen

```
>>> de_en = { "Milch": "milk", "Mehl": "flour" }
>>> print de_en
{'Mehl': 'flour', 'Milch': 'milk'}
>>> de_en["Oel"]="oil"
>>> for de in de_en: print de, "=>", de_en[de]
Mehl => flour
Milch => milk
Oel => oil
>>> for de, en in de_en.iteritems(): print de, "=>", en
>>> if "Mehl" in de_en: print "Die Uebersetzung von \"Mehl\""
```

- komma-getrennt in geschweiften Klammern
- speichert Paare von Schlüsseln (Keys) und Werten
- Speicher-Reihenfolge der Werte ist nicht festgelegt
- daher Indizierung über die Keys, nicht Listenindex o.ä.
- mit **in** kann nach Schlüsseln gesucht werden

```
>>> print "Integer %d %05d" % (42, 42)
```

```
Integer 42 00042
```

```
>>> print "Fliesskomma %e |%+8.4f| %g" % (3.14, 3.14, 3.14)
```

```
Fliesskomma 3.140000e+00 | +3.1400| 3.14
```

```
>>> print "Strings %s %10s" % ("Hallo", "Welt")
```

```
Strings Hallo      Welt
```

- Der %-Operator ersetzt %-Platzhalter in einem String
- %d: Ganzzahlen (Integers)
- %e, %f, %g: Fliesskomma mit oder ohne Exponent oder wenn nötig (Standardformat)
- %s: einen String einfügen
- %x[defgs]: auf x Stellen mit Leerzeichen auffüllen
- %0x[defg]: mit Nullen auffüllen
- %x.y[efg]: x gesamt, y Nachkommastellen

```
>>> original = list()
>>> original.append(3)
>>> original.append(2)

>>> kopie = list(original)
```

```
>>> original.append(1)
>>> original.sort()

>>> print original, kopie
[1, 2, 3], [3, 2]
```

- In Python können komplexe Datentypen wie Objekte im Sinne der **objekt-orientierten Programmierung** verwendet werden
- Datentypen entsprechen **Klassen** (hier `list`)
- Variablen entsprechen **Objekten** (hier `original` und `kopie`)
- Objekte werden durch Aufruf von `Klasse()` erzeugt
- **Methoden** eines Objekts werden in der Form `Objekt.Methode()` aufgerufen (hier `list.append()` und `list.sort()`)
- `help(Klasse/Datentyp)` informiert über vorhandene Methoden
- Per `class` kann man selber Klassen erstellen



- Zeichenkette in Zeichenkette suchen  
`"Hallo Welt".find("Welt")` → 6  
`"Hallo Welt".find("Mond")` → -1
- Zeichenkette in Zeichenkette ersetzen  
`"abcdabcabe".replace("abc", "123")` → '123d123abe'
- Groß-/Kleinschreibung ändern  
`"hallo".capitalize()` → 'Hallo'  
`"Hallo Welt".upper()` → 'HALLO WELT'  
`"Hallo Welt".lower()` → 'hallo welt'
- in eine Liste zerlegen  
`"1, 2, 3, 4".split(",")` → ['1', ' 2', ' 3', ' 4']
- zuschneiden  
`" Hallo ".strip()` → 'Hallo'  
`"..Hallo..".rstrip(".")` → 'Hallo..'

```
eingabe = open("ein.txt")
ausgabe = open("aus.txt", "w")
nr = 0
ausgabe.write("Datei %s mit Zeilennummern\n" % eingabe.name)
for zeile in eingabe:
    nr += 1
    ausgabe.write("%d: %s" % (nr, zeile))
ausgabe.close()
```

- Dateien sind mit `open(datei, mode)` erzeugte Objekte
- Mögliche Modi (Wert von `mode`):

r oder leer	lesen
w	schreiben, Datei zuvor leeren
a	schreiben, an existierende Datei anhängen

- sind Sequenzen von Zeilen (wie eine Liste von Zeilen)
- Nur beim Schließen der Datei werden alle Daten geschrieben

```
def loesungen_der_quad_gln(a, b, c):  
    "loest  $a x^2 + b x + c = 0$ "  
    det = (0.5*b/a)**2 - c  
    if det < 0: raise Exception("Es gibt keine Loesung!")  
    return (-0.5*b/a + det**0.5, -0.5*b/a - det**0.5)
```

```
try:  
    loesungen_der_quad_gln(1,0,1)  
except:  
    print "es gibt keine Loesung, versuch was anderes!"
```

- **raise** Exception("Meldung") wirft eine Ausnahme (Exception)
- Funktionen werden nacheinander an der aktuellen Stelle beendet
- mit **try: ... except: ...** lassen sich Fehler abfangen, dann geht es im **except**-Block weiter

```
>>> import random
```

```
>>> print random.random(), random.randint(0, 100)  
0.576899996137, 42
```

```
>>> from random import randint
```

```
>>> print randint(0, 100)  
14
```

- enthalten nützliche Funktionen, Klassen, usw.
- sind nach Funktionalität zusammengefasst
- werden per **import** zugänglich gemacht
- Hilfe: `help(modul)`, alle Funktionen: `dir(modul)`
- einzelne Bestandteile kann man mit **from ... import** importieren
  - bei Namensgleichheit kann es zu Kollisionen kommen!

```
import math
import random
def boxmuller():
    """
    liefert normalverteilte Zufallszahlen
    nach dem Box-Mueller-Verfahren
    """
    r1, r2 = random.random(), random.random()
    return math.sqrt(-2*math.log(r1))*math.cos(2*math.pi*r2)
```

- math stellt viele mathematische Grundfunktionen zur Verfügung, z.B. **floor/ceil**, **exp/log**, **sin/cos**, **pi**
- random erzeugt *pseudozufällige* Zahlen
  - **random**(): gleichverteilt in  $[0, 1)$
  - **randint**(a, b): gleichverteilt ganze Zahlen in  $[a, b)$
  - **gauss**(m, s): normalverteilt mit Mittelwert  $m$  und Varianz  $s$

stellt Informationen über Python und das laufende Programm selber zur Verfügung.

- `sys.argv`: Kommandozeilenparameter, `sys.argv[0]` ist der Programmname
- `sys.path`: Liste der Verzeichnisse, in denen Python Module sucht
- `sys.exit("Fehlermeldung")`: bricht das Programm ab
- `sys.stdin`,  
`sys.stdout`,  
`sys.stderr`: Dateiobjekte für Standard-Ein-/Ausgabe

---

```
zeile = sys.stdin.readline()
```

```
sys.stdout.write("gerade eingegeben: %s" % zeile)
```

```
sys.stderr.write("Meldung auf der Fehlerausgabe")
```

---