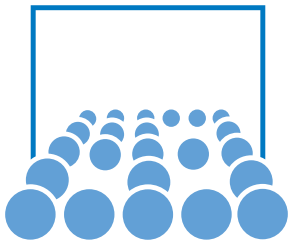
Scientific Computing in Computer Science,
Technische Universität München

Einführung in die wissenschaftliche Programmierung

Dr. Martin Buchholz

Wintersemester 2010/2011



Teil I

Organisatorisches und erste Schritte

Inhalte und Ziele

Einführung in die

- Keine Programmiererfahrung nötig
- Umgang mit PC/Betriebssystem sollte bekannt sein
- Mischung aus einfachen und schwierigen Teilen
- Kratzt an vielen Stellen nur an der Oberfläche

wissenschaftliche

- Mathematisches und physikalisches Grundverständnis
- Beispiele aus dem wissenschaftlichen Bereich
- Hauptziel bleibt Verständnis der Programmierkonzepte

Programmierung

- Hauptsächlich Implementierung
- Aber auch der Entwurf von Programmen
- Übliche Programmierkonstrukte und wissenschaftliche Bibliotheken
- Qualität

Organisatorisches

Raum und Zeit

- Modul IN8008
- Vorlesung Mo 10:15 - 11:45, MI HS1
- Zentralübung Mi 08:30 - 10:00, PH HS1

Informationen

- Webseite: `www5.in.tum.de` → Teaching → Einführung in die wissenschaftliche Programmierung
- FRAGEN!
- Sprechstunde Mo 13:00-14:00 u.n.V (buchholm@in.tum.de)

Nötige Software

- Python 2.x (idle, ipython)
- Bibliotheken: numpy, scipy, matplotlib
- Beliebiger Editor (emacs, vi, nedit, ...)
- Im CIP-Pool ist all das vorhanden

Typen von Programmiersprachen

Deklarative Programmierung

- Funktional (Lisp, Scheme, make, ...)
- Logisch (Prolog)
- Sonstige (Sql, ...)

Imperative Programmierung

- Prozedural (C, Fortran, (Python), Pascal, ...)
- Modular (Bash, ...)
- Objektorientiert (C++, Java, Python, ...)

Skripting vs. Compilieren

Skriptsprachen

- Bash, Python, Perl, postscript, matlab/octave, ...
- Interaktiver Modus
- Teilweise wenig Optimierungen
- Einfach zu verwenden
- Keine Binärdateien (daher für kommerzielle Software meist ungeeignet)

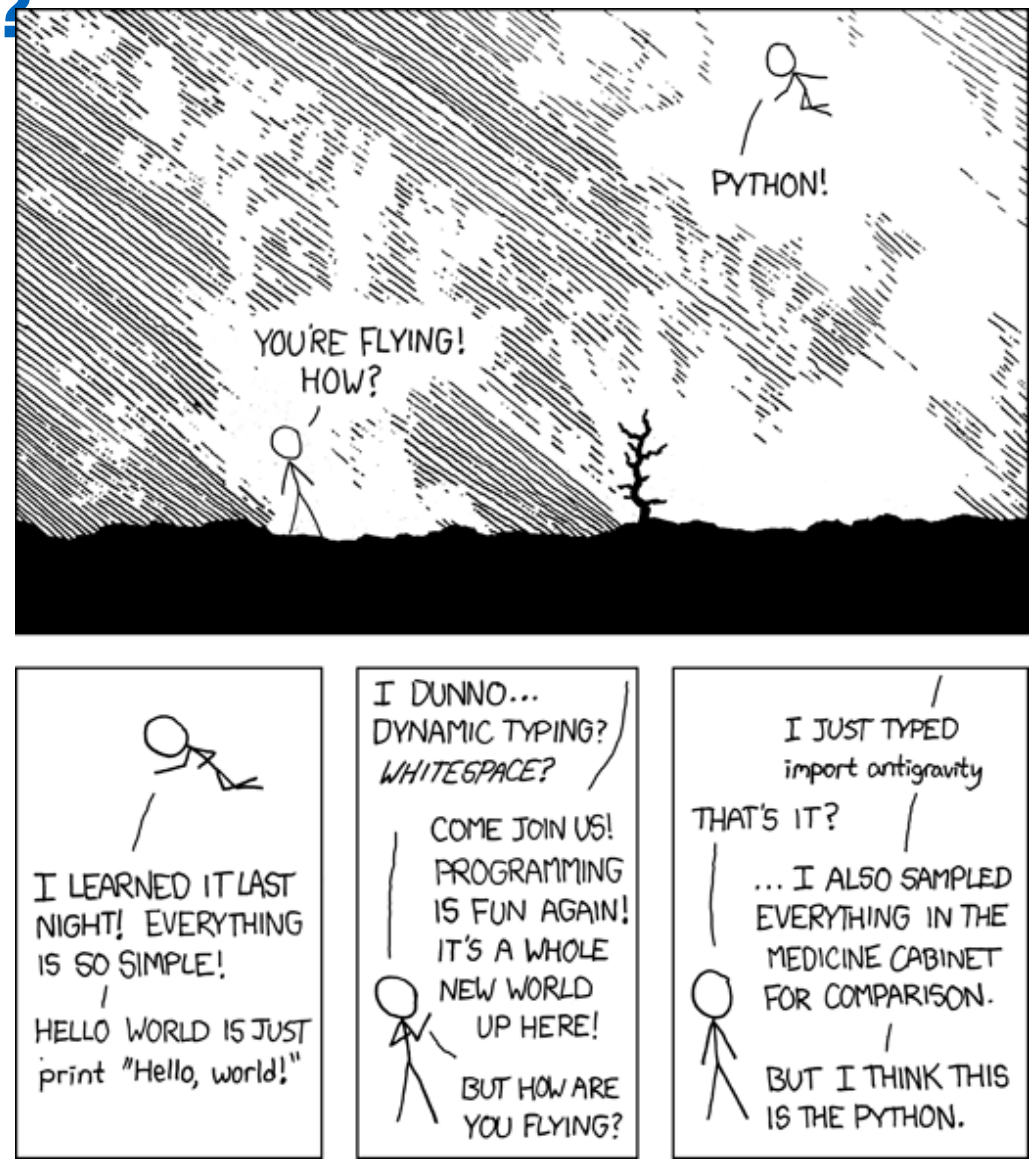
Kompilierte Sprachen

- C, C++, Fortran, ...
- Effizient für sehr rechenaufwändige Aufgaben
- Sourcecode muss in Binärcode übersetzt werden

Sonstige

- Java

Warum Python?



Source: `http://xkcd.com/353/`

Worum geht es in Python?

Python ist ...

- High-level (Wirklich high-level!)
- Komplett objektorientiert
- Interpretiert
- Skalierend
- Erweiterbar
- Portierbar (Windows, Linux, Mac OS X, Amiga, HPC, Cluster, Web Server, Palm/Handhelds, Mobiltelefone, ...)
- Vielseitig
- Einfach zu lernen, zu verstehen, zu verwenden und zu warten
- Kostenlos, open-source

Wenn ich nicht weiter weiß?

Literatur

- RRZN Python-Skript beim LRZ (4,50 Euro)
- David M. Beasley: Python - Essential Reference
- Hans Petter Langtangen: A Primer on Scientific Programming with Python
- Vorlesungsfolien im Web
- http://docs.python.org
- www, google

Interactive Hilfe

- `help()` für interactive Hilfe
- `help('modules)` Liste der verfügbaren Module
- `help(object|'command')` Hilfe zu Objekt oder Befehl

Worum geht es in Python?

Kann verwendet werden für:

- OS Skripting (In vielen Fällen besser als Bash)
- Internet Programmierung
- Wissenschaftliches Rechnen (selbst komplexe Simulationen)
- Parallelisierung
- GUI (TKinter)
- Visualisierung
- Rapid Prototyping
- ... und vieles mehr

Worum geht es in Python?

Ursprünge

- Erfunden 1990 als Lehrsprache
- Hauptziel einfach lesbarer code
- Benannt nach Monty Python

Weitere Features

- Dynamische Typisierung
- Automatische garbage collection
- Verschiedene Programmierparadigmen (Prozedural, OO, funktional, Aspekt-orientiert, ...)

Generelle Programmierregeln

- Kommentare !
- Problem $\Rightarrow$ Algorithmus $\Rightarrow$ Programm
- Modulare Programmierung
- Generisch wo möglich, spezifisch wo nötig

Python ausführen: python...

- Python interpreter starten (python oder ipython)
- Python-prompt: >>> Befehl eintippen und mit <Enter> bestätigen
- ipython-prompt: In [1]:

```
$ python
Python 2.5.2 (r252:60911, Jan 20 2010, 23:16:55)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> help
Type help() for interactive help, or help(object) for
help about object.
>>> help()

Welcome to Python 2.5!  This is the online help
utility. [...]
```

Python ausführen: ipython

Vorteile

- Erweiterte interaktive Shell
- Zusätzliche Shell Syntax
  - “magische” Funktionen, beginnend mit “%”, z.B. `%psearch` um Funktionen im Namensraum zu suchen
  - Zugriff auf Bash-Befehle `%cd`
  - Python-Skripte ausführen mit `run filename.py`
  - Logging, Makros, . . .
  - Pretty printing, umschalten `%Pprint`
- Syntax highlighting
- Tab-Vervollständigung
- History
  - Vorherige Befehlsblöcke wiederholen
- Automatische Einrückung
- Mitgeliefert mit SciPy

Python ausführen: Datei übergeben

- Datei `square_me.py`:

```
print 6**2
```

- Ausführen mit

```
$ python square_me.py
12
```

Python ausführen: ausführbares Skript

- Gleicher Programmcode wie zuvor
- Dem Betriebssystem mitteilen, dass python verwendet werden soll:

```
#!/usr/bin/python
print 6**2
```

- Datei ausführbar machen und ausführen

```
$ chmod u+x square_me.py
$ ./square_me.py
12
```

“Hello World!”

Python – lesbarer Code

- In Python

```
>>> print "Hello World!"
Hello World!
```

- In Java

```
public class Hello{
    public static void main(String argv[]){
System.out.println("Hello World!");
    }
}

$ javac Hello.java
$ java Hello
Hello World!
```

Variablen, Zuweisungen und Ausdrücke

```
>>> 3 + 4**2/8 + 1
>>> (6*3.5)*2 - 5
>>> 10e-4
>>> a = 17
>>> a
>>> a - 1.5
>>> a = a - 1
>>> b = "Hello World!"
>>> print b
>>> print a, b, (4 + 5**0.5)
```

- Zuweisung: Variablenname = Ausdruck
- Nicht zu verwechseln mit mathematischem =
- Ausgabe des Ergebnisses nur interaktiv
- `print` gibt eine String-Repräsentation von Objekten aus
- Leerzeichen im Ausdruck egal (Aber nicht VOR dem Ausdruck)

Variablennamen

- a-z, A-Z, 0-9, _
- Erstes Zeichen nicht 0-9
- _ ist erlaubt, hat aber spezielle Bedeutung
- Variablennamen sollten aussagekräftig sein
- Bei mathematischen Ausdrücken entsprechend den mathematischen Variablen
- Ansonsten deskriptiv
- Einige Worte sind reserviert:

```
and, as, assert, break, class, continue, def, del, elif, else,
except, False, finally, for, from, global, if, import, in , is,
lambda, None, nonlocal, not, or, pass, raise, return, True,
try, with, while, yield
```

- Variablen in Python sind dynamisch typisiert

Mathematische Funktionen: Modul math

- Ein Modul ist eine Sammlung von Funktionalitäten
- Einbinden mit `import`
- `math` enthält mathematische Standardfunktionen
- `sin, cos, sqrt, exp, log, pow, ...`
- Zusätzlich Konstanten `pi` und `e`
- Funktioniert nur mit integer und float
- Für komplexe Zahlen: `cmath`

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Teil II

Datentypen

Numerische Typen - ganze Zahlen

- Rechnen wie wir es gewohnt sind (`12*34+567`)
- In Python ist der Zahlenbereich beliebig groß

```
>>> 1234**56
129911902554871451941032084396235137754657820101 27
392384379012704624259433055094648925678485362472 90
201061395156473849109449211865238658490562753590 66
26235291168250476992921 6L
>>> type(1234)
<type 'int'>
>>> type(1234**56)
<type 'long'>
```

- Achtung beim Dividieren: Nachkommastellen werden abgehackt

```
>>> 5/3
1
```

Numerische Typen - float und bool

- `bool` – wahr und falsch

```
print True, False
True == True
True != True
```

- `float` – Fließkommazahl (double precision)
  $f = s \cdot m \cdot 2^e$, 1+52+11 Bit

```
>>> 1.0 - 49.0*(1.0/49.0)
1.1102230246251565e-16
>>> 1 - 49*(1/49)
1
>>> 1 - 49*(1/49.0)
1.1102230246251565e-16
```

- `None` – special type (“undefined”)

Numerische Typen - Komplexe Zahlen

```
>>> 1+4j
(1+4j)
>>> complex(2,3)
(2+3j)
>>> 2*(c**3 - 3j)
(-92+12j)
>>> d.real
-92.0
>>> d.imag
12.0
```

- Imaginärteil hat Zusatz `j`
- Basisoperationen normal anwendbar
- Für komplexere Operationen (z.B. sin) spezielle Bibliotheken
- Alles in Python ist ein Objekt
- Zugriff auf Objektmethoden: `<Objektname>.<Methodenname>`

Numerische Typen - Vergleichsoperatoren

- `a == b` gibt `True` bei Gleichheit (= dient Zuweisung)

```
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 1 == "1"
False
```

- weitere Operatoren: `!=`, `<`, `>`, `<=`, `>=`
- Verknüpfung mit `and` und `or`, Negation mit `not`
- Shifting: `<<`, `>>`
- Bitweise Operatoren: `&`, `|`, `^`, `~`

```
>>> not(1 == 2 or 3 + 1 == 2 << 1)
False
```

Sequenzen - Strings

- Sequenz von Zeichen (character)
- einzelne oder doppelte Anführungszeichen
- dreifache Anführungszeichen für mehrzeiligen String

```
"...␣there␣lived␣a␣hobbit."
"a␣'hobbit'"
'a␣"hobbit"'
'a␣\'hobbit\''
"""In␣a␣hole␣in␣the␣ground
there␣"lived"␣a␣'hobbit'"""
```

- Verknüpfung von Strings

```
s1 = "In␣a␣hole␣in␣the␣ground"
s2 = "there␣lived␣a␣hobbit"
s = s1 + '␣' + s2
print s
```

Sequenzen - Strings (2)

- Replikation

```
>>> "Hi!␣"*3
'Hi!Hi!Hi!'
```

- Indizierung und Abschnitte

```
>>> s = "12345678"
>>> s[3], s[2:4], s[-2]
('4', '34', '7')
>>> s[6:], s[:3]
('78', '123')
>>> s[0:-1:2] # stride 2
'1357'
>>> s[2] = 4  # Fehler: Strings sind unveränderbar
>>> len(s)
8
```

Ausgewählte String-Methoden

```
s = "␣␣Frodo␣and␣␣Sam␣and␣␣␣Bilbo"
s.islower()
s.isupper()
s.startswith("Frodo") # s.startswith("Frodo", 2)
s.endswith("Bilbo")
s = s.strip()
s.upper()
s = s.lower()
s = s.center(len(s)+4)# zentrieren
s.lstrip()
s.rstrip("␣")
s = s.strip()
s.find("sam")
s.rfind("and")
```

Ausgabe verschiedener Datentypen

print

```
>>> a=3; b=4.5; c='text'
>>> print a, b, c
3 4.5 text
```

Stringinterpolation

- Stringinterpolation gibt hohe Flexibilität
- Platzhalter mit % in einem String
- Nach dem String folgen ein % und ein Tupel mit Werten

```
>>> print "int: %d, float: %f und string: %s" \
          % (4, 3.5, "bla")
int: 4, float: 3.500000 und string: bla
```

Konvertierungszeichen zur Stringinterpolation

- `%d` Dezimal Integer
- `%f` Gleitpunkt (float)
- `%e`,`%E` Gleitpunkt wissenschaftlich (m.ddde+xx)
- `%g`,`%G` kompakteste Gleitpunktdarstellung
- `%s` String
- `%c` einzelnes Zeichen
- `%o` Oktaldarstellung eines Integers
- `%x`, `%X` Hexadezimaldarstellung eines Integers
- `%%` Das Zeichen %

```
>>> a = 7.357
>>> print "%f, %.2f, %8.1f" % (a, a, a)
7.357000, 7.36,      7.4
```

Sequenzen - Tupel

- Sequenzen beliebiger Objekte `(,)`
- Unveränderlich
- Indizierung und Ausschnitte wie bei Strings

```
>>> l = (1, "zwei", (3,4,5))
>>> l[0]
1
>>> l[-1]
(3,4,5)
>>> l[1] = 2   # error
>>> len(l)
3
>>> l + (6,7)
(1, 'zwei', (3, 4, 5), 6, 7)
```

- Minimun und Maximum mit `min(l)` und `max(l)`

Sequenzen - Listen

- Sequenzen beliebiger Objekte `(,)`
- Veränderlich
- Indizierung und Ausschnitte wie bei Strings

```
>>> l = [1, "zwei", (3,4,5)]
>>> l[0]
1
>>> l[1] = 2; l   # works!
[1, 2, (3,4,5)]
>>> l + [6,7]
[1 , 2, (3, 4, 5), 6, 7]
```

- Operations on lists

```
l = [0,1,2,3,4,5,6,7,8,9]
l[2:4] = [10, 11]
l[1:7:2] = [-1, -2, -3]
del l[::2]
```

Sequenzen - Listen (2)

- Listen sind Objekte mit Methoden

```
>>> l = [0,1,2]
>>> l.append("x") # [0, 1, 2, 'x']
>>> l.extend([5,6]) # [0, 1, 2, 'x', 5, 6]
>>> l.insert(2, "x") # [0, 1, 'x', 2, 'x', 5, 6]
>>> l.count("x") # 2
>>> l.sort() # [0, 1, 2, 5, 6, 'x', 'x']
>>> l.reverse() # ['x', 'x', 6, 5, 2, 1, 0]
>>> l.remove("x") # ['x', 6, 5, 2, 1, 0]
>>> l.pop() # ['x', 6, 5, 2, 1]
0
```

- Integerlisten erzeugen: `range` function

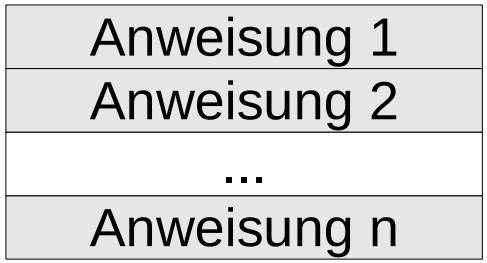
```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(-10, 5, 3)
[-10, -7, -4, -1, 2]
```

Teil III

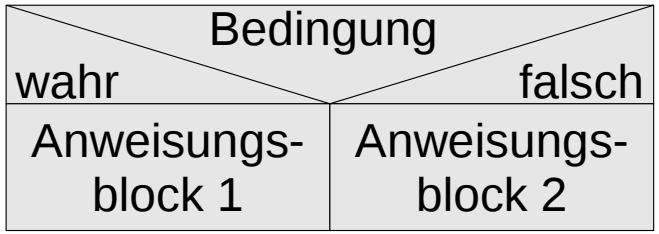
Kontrollstrukturen

Struktogramme

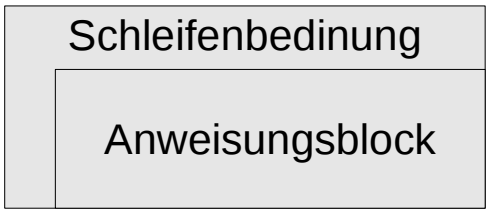
- Graphische Darstellung von Programmen
- Anweisungen



- Bedingungen

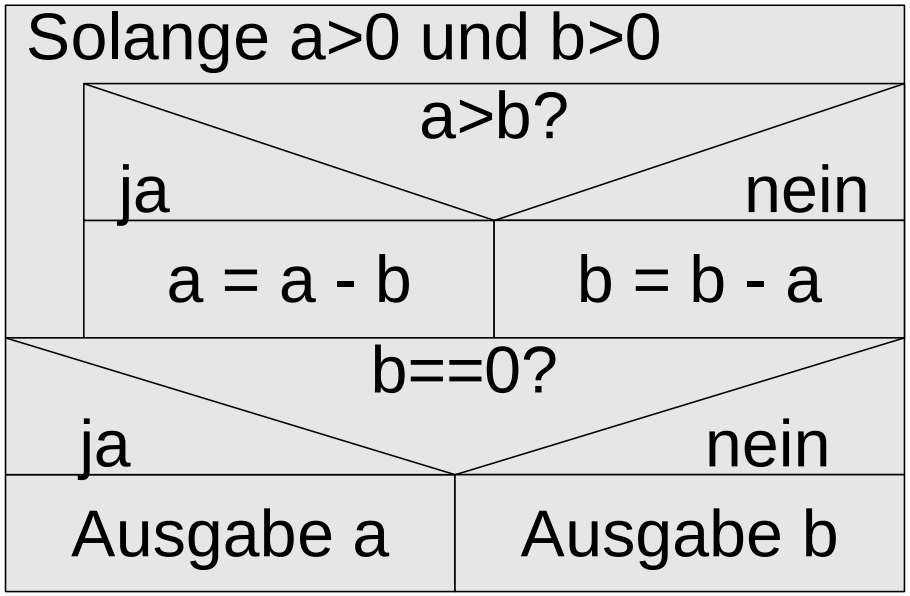


- Schleifen



Struktogramme - Beispiel

- Euklidscher Algorithmus zur Berechnung des ggT:
  Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.



Kontrollstrukturen

Generelle Bemerkungen

- Blöcke (siehe Struktogramme) müssen im Code markiert werden
- In vielen Sprachen üblich: geschweifte Klammern
- Python verwendet stattdessen Einrückungen
- Einrücktiefe entscheidet über Blockzugehörigkeit
- Doppelpunkt leitet Block ein

Fallunterscheidung: if

```
if x < y:
    print "x is smaller"
    print x
elif x == y:
    print "both are equal"
else:
    print "y is smaller"
    print y
```

Nochmal Typen und Objekte

- Jedes Objekt hat Identität (id), Typ (type) und Wert

```
>>> b = 42 # Wert: 42
>>> type(b)
<type 'int'>
>>> id(b)
158788940
>>> type(type(b))
<type 'type'>
```

- Vergleich zweier Objekte

```
if a is b:
  print 'a␣und␣b␣sind␣dasselbe␣Objekt'
if a == b:
  print 'a␣und␣b␣haben␣denselben␣Wert'
if type(a) is type(b):
  print 'a␣und␣b␣haben␣denselben␣Typ'
```

Nochmal Typen und Objekte

- Wenn zwei Objekte identisch sind (`id(a) is id(b)`), haben sie auch den gleichen Wert und Typ
- Gleicher Typ sagt nichts über Identität und Wert aus
- Gleicher Wert sagt nichts über Identität und Objektgleichheit aus

```
>>> b = 42
>>> l = [1,2,3]
>>> type(b) is list
False
>>> type(l) is list
True
>>> filehandle = open("test.txt", "w")
>>> type(filehandle)
<type 'file'>
```

Kontrollstrukturen - Zählschleife

for iteriert über die Elemente einer Sequenz

```
for i in [1,2,3]:
  print "Zahl: ", i
```

```
for i in "Hallo":
  print "Buchstabe: ", i
```

```
for i in (1, 'a', [4,3,2], "Hallo"):
  print "Element: ", i
```

range erzeugt Zähllisten

```
>>> range(3)
[0,1,2]
>> range(10,-5,-2)
[10, 8, 6, 4, 2, 0, -2, -4]
```

Kontrollstrukturen - Schleifen mit Bedingungen

- `continue` startet den nächsten Schleifendurchlauf (`for` und `while`)
- `break` bricht eine Schleife ab (`for` und `while`)

```
a = ("let", "us", "find", "Bilbo", "again")
for word in a:
  if word == "Bilbo":
    print "found␣Bilbo"
    break
print word
```

- Eine `while` Schleife iteriert solange eine Bedingung `True` ist

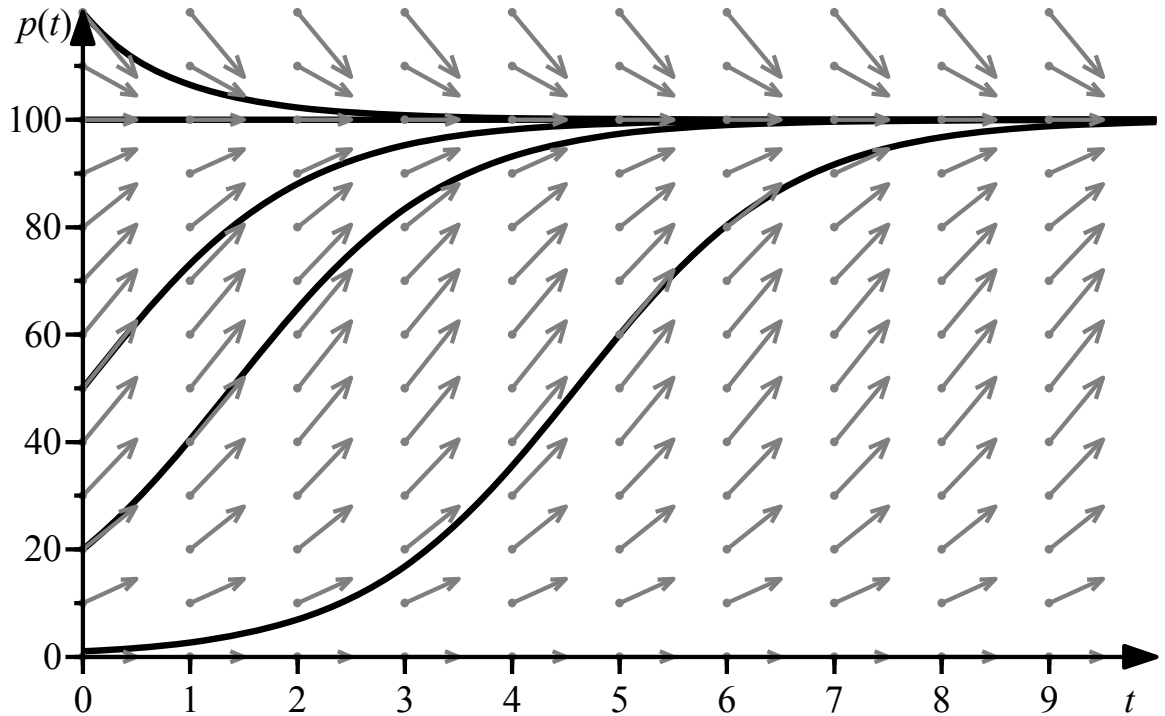
```
a = 2048; exp = 0
while a > 1:
  a /= 2          # a = a / 2
  exp+=1          # exp = exp + 1
print a, "=", 2, "^", exp
```

Euklidscher Algorithmus mit Python

```
while a>0 and b>0:
  if a>b:
    a = a-b
  else:
    b = b-a
if (b==0):
  print a
else:
  print b
```

Beispiel: Numerische Lösung von ODEs

- Logistische Differentialgleichung: $\dot{p}(t) = ap(t) - bp(t)^2$
- Anfangswert $p(0) = p_0$
- Lösung $p(t) = \frac{a \cdot p_0}{b \cdot p_0 + (a - b \cdot p_0) \cdot e^{-at}}$
- Z.B. $a = 1$, $b = 1/100$



Mittels einer Schleife können wir uns eine Wertetabelle drucken lassen:

```
from math import exp
a = 1.      # Parameter der DGL
b = 0.01
p0 = 10.    # Anfangswert
tend = 5.  # Intervall [0, tend]
N = 10      # Teilintervalle
for i in range(0, N+1):
    t = (tend*i)/N
    p = a*p0 \
      /(b*p0 + (a-b*p0)*exp(-a*t))
    print '%6.1f %6.1f' % (t, p)
```

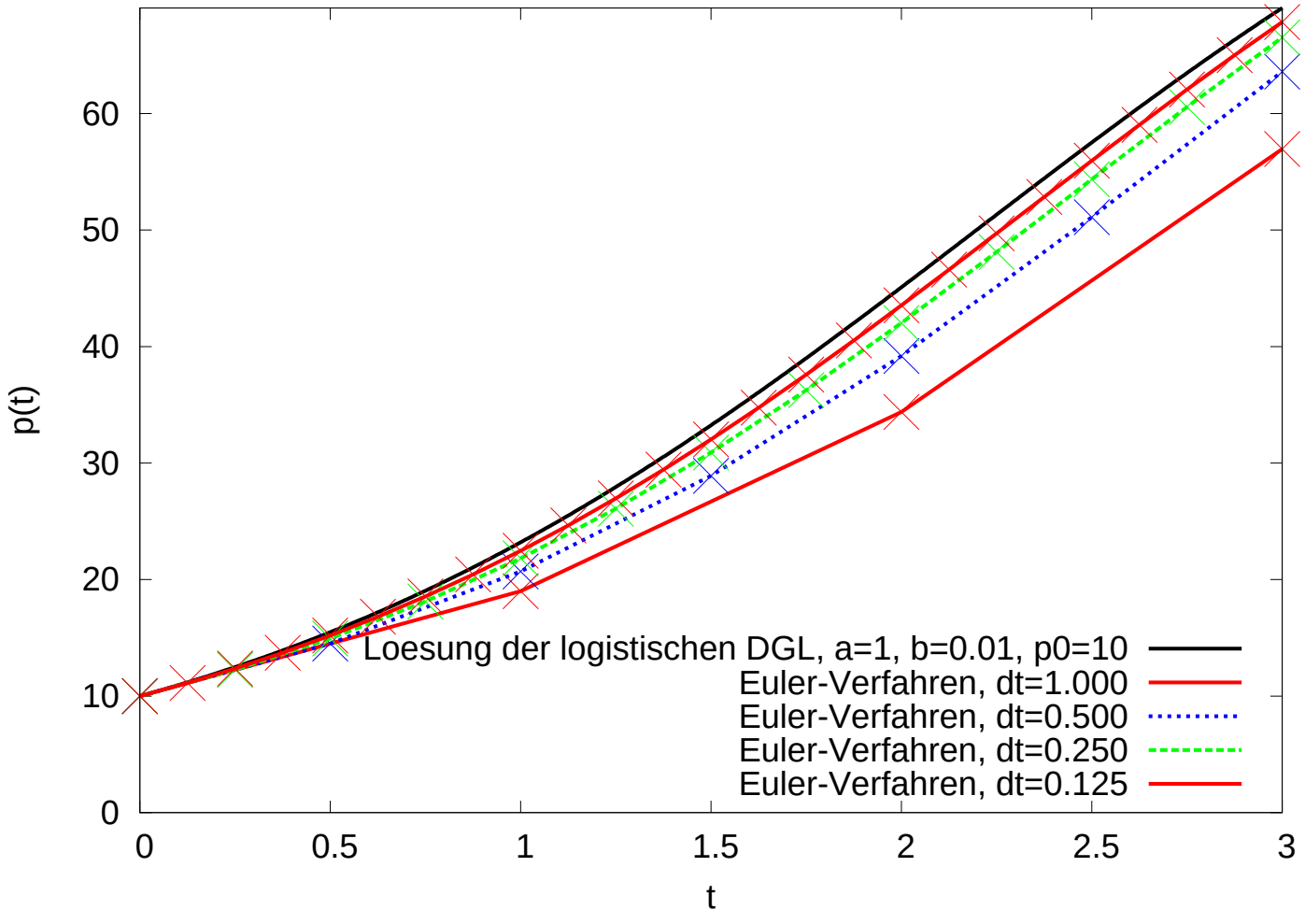
```
   0.0   10.0
   0.5   15.5
   1.0   23.2
   1.5   33.2
   2.0   45.1
   2.5   57.5
   3.0   69.1
   3.5   78.6
   4.0   85.8
   4.5   90.9
   5.0   94.3
```

Was passiert, wenn wir in `tend = 5.` den Punkt weglassen?

Explizites Euler-Verfahren

- Verfahren zur numerischen Lösung einer DGL
- DGL: $\dot{p}(t) = f(t, p(t))$
- Ersetze Ableitung durch Vorwärtsdifferenz $\dot{p}(t) = \frac{p(t+\delta t)-p(t)}{\delta t}$
- $\Rightarrow p(t+\delta t) \doteq p(t) + \delta t \cdot f(t, p(t))$
- δt positiv und betragsmäßig klein
- Berechnungsvorschrift, um von $p(t)$ einen Schritt δt in die Zukunft zu gehen

Das Euler-Verfahren für die logistische Differentialgleichung, vier verschiedene Zeitschrittweiten δt:



Das Programm für das Euler-Verfahren sieht ganz ähnlich aus wie das für die Wertetabelle von vorhin:

```
a = 1.        # Parameter der DGL
b = 0.01
p0 = 10.      # Anfangswert
tend = 3.     # Interv. [0, tend]
N = 6         # Teilintervalle
dt = tend/N # Zeitschrittweite
p = p0
t = 0.
for i in range(0, N):
    t = t + dt
    p = p + dt*(a*p - b*p**2)
    print '%6.1f %6.1f' % (t, p)
```

```
   0.5   14.5
   1.0   20.7
   1.5   28.9
   2.0   39.2
   2.5   51.1
   3.0   63.6
```

Zusammenhang zwischen Zeitschrittweite und Genauigkeit

- Programm umbauen, um in Abhängigkeit von der Arbeit (Teilintervalle N) die Genauigkeit (Fehler) anzugeben
- Vergleich nur am Ende des Intervalls
- `from math import exp` zur Berechnung der exakten Lösung
- `pend = p0*a/(p0*b + (a-b*p0)*exp(-a*tend))`
- `print '%6d␣%7.3f␣%7.3f'% (N, p, pend-p)`
- Für verschiedene $N = 6 \cdot 2^j, j = 0...9$:

```
for j in range(0,10):
  N = 6*2**j # Anzahl Teilintervalle
```

Erweitertes Programm

```
>from math import exp
a = 1.        # Parameter der DGL
b = 0.01
p0 = 10.      # Anfangswert
tend = 3.     # Interv. [0, tend]
>pend = p0*a/(p0*b + (a-b*p0)*exp(-a*tend))
>for j in range(0,10):
  >N = 6*2**j # Anzahl Teilintervalle
  dt = tend/N # Zeitschrittweite
  p = p0
  t = 0.
  for i in range(0, N):
      t = t + dt
      p = p + dt*(a*p - b*p**2)
  >print '%6d␣%7.3f␣%7.3f' % (N, p, pend-p)
```

Ergebnisse Euler

N	Näherung	Fehler
6	63.590	5.467
12	66.529	2.528
24	67.847	1.209
48	68.466	0.591
96	68.765	0.292
192	68.912	0.145
384	68.984	0.072
768	69.021	0.036
1536	69.039	0.018
3072	69.048	0.009

- Halbierung von δt (Verdoppelung von N) halbiert den Fehler
- Vgl. Folie 51

Verfahren von Heun

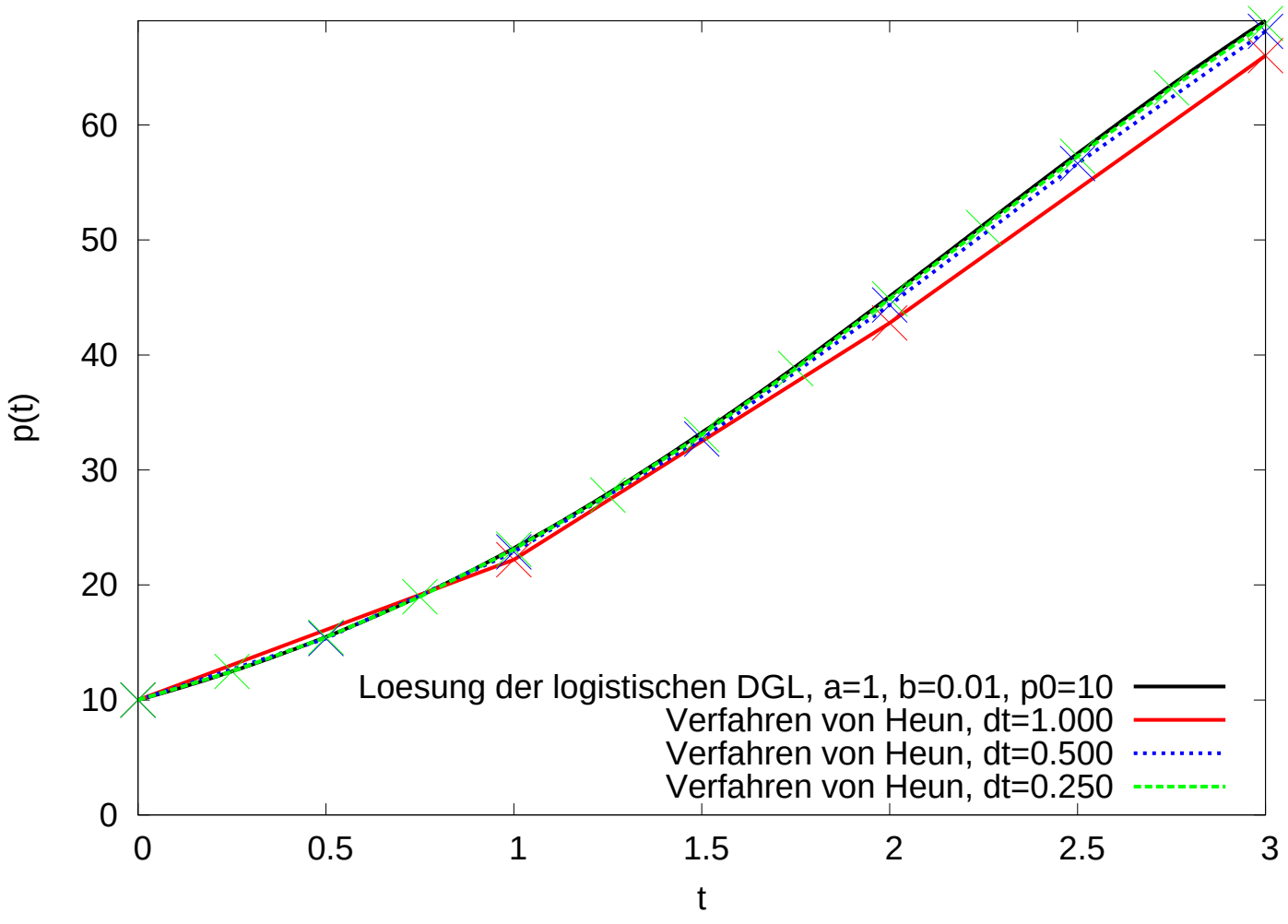
- Konvergenz des Euler-Verfahrens schlecht
- Idee zur Verbesserung: Nicht nur Steigung am aktuellen Punkt
- Mittelwert aus Steigung am aktuellen Punkt und Steigung am nächsten Punkt (bzw. dessen Näherung):

$$p(t+\delta t) \doteq p(t) + \delta t \frac{f(t,p(t)) + f(t+\delta t, p(t) + \delta t \cdot f(t,p(t)))}{2}$$

- bisherige Programmzeile: `p = p + dt*(a*p - b*p**2)`
- wird ersetzt durch;

```
f1 = a*p - b*p**2
ptmp = p + dt*f1
f2 = a*ptmp - b*ptmp**2
p = p + dt*(f1 + f2)/2
```

Das Heun-Verfahren für die logistische Differentialgleichung, drei verschiedene Zeitschrittweiten δt:



Ergebnisse Heun

N	Näherung	Fehler
6	68.140	0.916
12	68.807	0.250
24	68.992	0.065
48	69.040	0.017
96	69.053	0.004
192	69.056	0.001

- Halbierung von δt (Verdoppelung von N) viertelt den Fehler
- Euler und Heun sind Einschrittverfahren
- Neben dem Fehler sind weitere Eigenschaften wichtig (z.B. Energieerhaltung)
- Komplexere Einschrittverfahren, z.B. Runge-Kutta
- Darüber hinaus Mehrschrittverfahren

Ausblick

Was fehlt?

- Kommentare
- An einigen Stellen wurde Code dupliziert
- $\Rightarrow$ schlechte Wartbarkeit
- Irgendwann wird der Code lang und unübersichtlich

Lösung?

- Codstücke zusammenfassen und auslagern in Funktionen
- Übergabe von Parametern an die Funktion
- Auswertung der Rückgabewerte der Funktion