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'>
```

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␣das␣identische␣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 (2)

- Wenn zwei Objekte identisch sind (`is` bzw. `id(a) == 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 aus.

Nochmal Typen und Objekte (2)

- Wenn zwei Objekte identisch sind (`is` bzw. `id(a) == 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 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
```

Produktberechnung

Hands-On: Berechnen Sie das Produkt der Zahlen im Tupel (3,7,6,2) mittels einer `for`-Schleife!

Produktberechnung

Hands-On: Berechnen Sie das Produkt der Zahlen im Tupel (3,7,6,2) mittels einer `for`-Schleife!

```
>>> prod=1
>>> for i in (3,7,6,2):
>>>   prod=prod*i
>>> print prod
```

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
```

Kontrollstrukturen - Schleifen mit Bedingungen

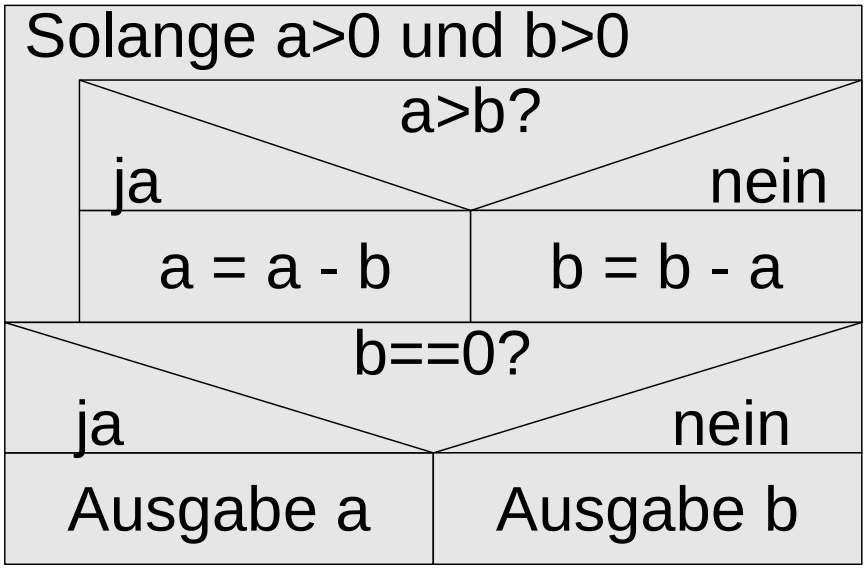
- `continue` startet den nächsten Schleifendurchlauf (`for` und `while`).
- `break` bricht eine Schleife ab (`for` und `while`).

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

- Eine `while` Schleife iteriert, solange ihre Bedingung `True` ist.

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

For vs. While

Jede `for`-Schleife lässt sich als `while`-Schleife formulieren:

```
for i in (2,3,7):
  print i
```

```
i=0
t=(2,3,7)
while (i<len(t)):
  print t[i]
  i=i+1
```

Inneres Produkt

Hands-On: Berechnen Sie das innere Produkt von (2,1,3,2) und (1,4,3,2) mittels einer `for`-Schleife!

Erinnerung:

- Inneres Produkt zweier Vektoren $v, w \in \mathbb{R}^4$: $\sum_{i=1}^{4} v_i \cdot w_i$
- Befehl `range(0,y)`: liefert Liste `[0,...,y-1]`

Inneres Produkt

Hands-On: Berechnen Sie das innere Produkt von (2,1,3,2) und (1,4,3,2) mittels einer `for`-Schleife!

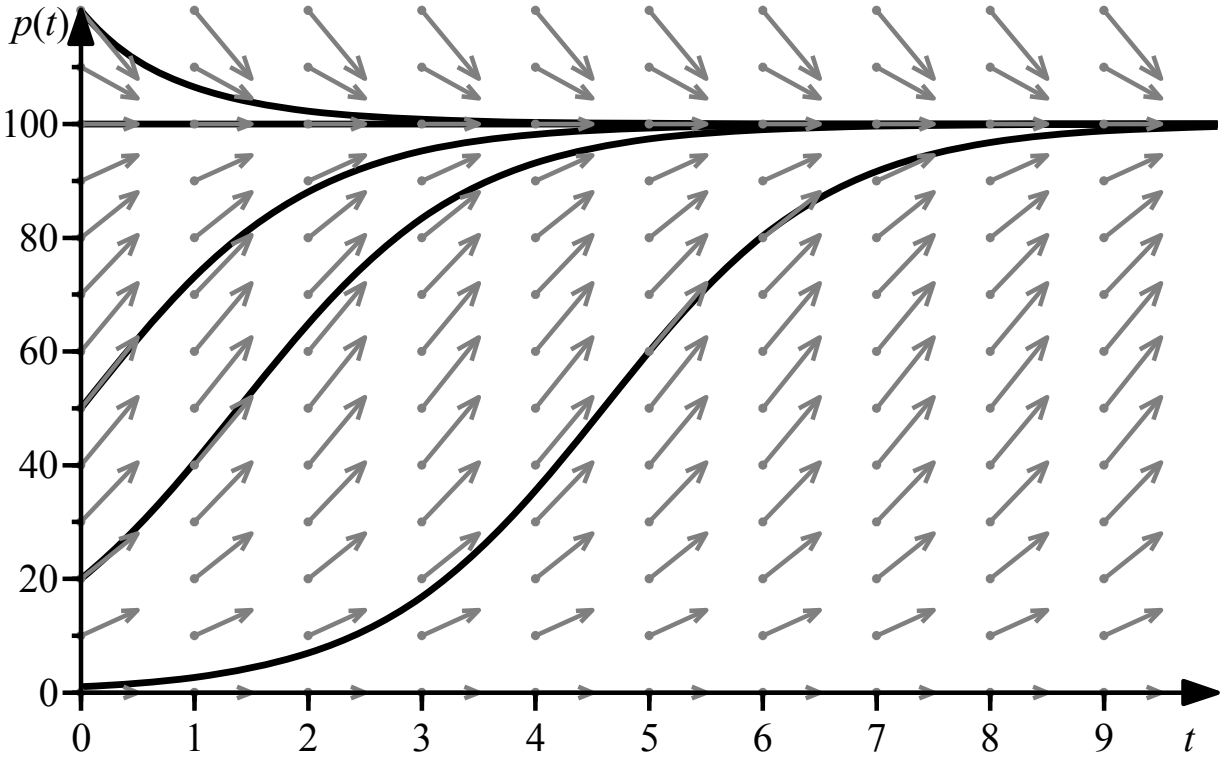
Erinnerung:

- Inneres Produkt zweier Vektoren $v, w \in \mathbb{R}^4$: $\sum_{i=1}^{4} v_i \cdot w_i$
- Befehl `range(0,y)`: liefert Liste `[0,...,y-1]`

```
prod=0
t1 = (2,1,3,2)
t2 = (1,4,3,2)
for i in range(0,4):
  prod = prod +t1[i]*t2[i]
print prod
```

Beispiel: Numerische Lösung von ODEs

- Logistische Differentialgleichung: $\dot{p}(t) = ap(t) - bp(t)^2$ (Modell von Verhulst)
- 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. $a = 1$, $b = 1/100$



Analytische Lösung

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
```

Analytische Lösung

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
```

Hands-On: Was passiert, wenn wir in `tend = 5.` den Punkt weglassen?

Numerische Lösung: Explizites Euler-Verfahren

- Verfahren zur numerischen Lösung einer DGL
- DGL: $\dot{p}(t) = f(t, p(t))$
- Ersetze Ableitung durch Vorwärtsdifferenz $\dot{p}(t) \approx \frac{p(t+\delta t)-p(t)}{\delta t}$
- $\Rightarrow p(t + \delta t) = p(t) + \delta t \cdot f(t, p(t))$,
  Logistische DGL: $f(t, p(t)) = ap(t) - bp(t)^2$
- δt positiv und betragsmäßig klein
- Berechnungsvorschrift, um von $p(t)$ einen Schritt δt in die Zukunft zu gehen

$$\boxed{p(t + \delta t) = p(t) + \delta t \cdot (ap(t) - bp(t)^2)}$$

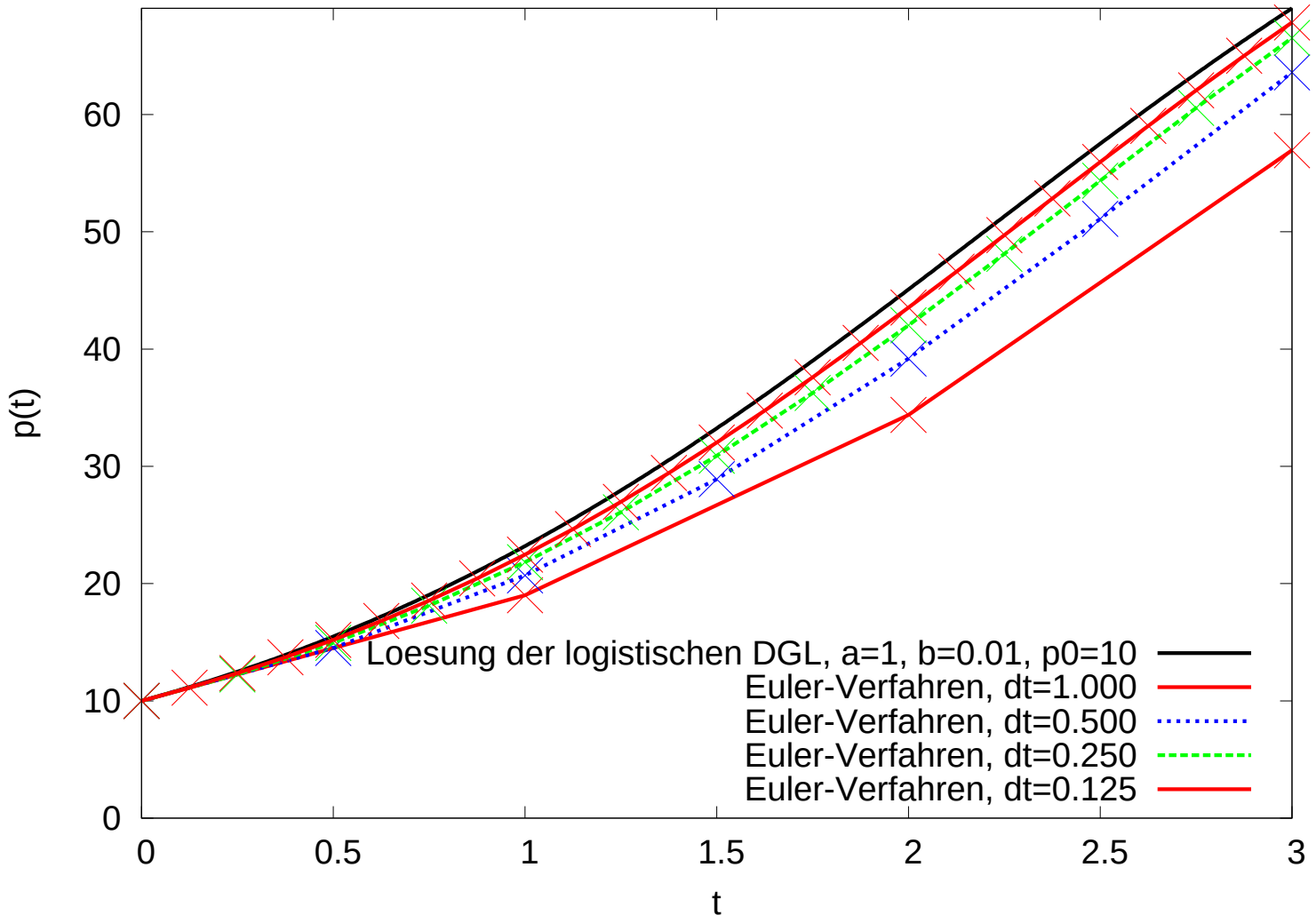
$$p(t + \delta t) = p(t) + \delta t \cdot (ap(t) - bp(t)^2)$$

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
```

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



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

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 59

Numerische Lösung: Verfahren von Heun

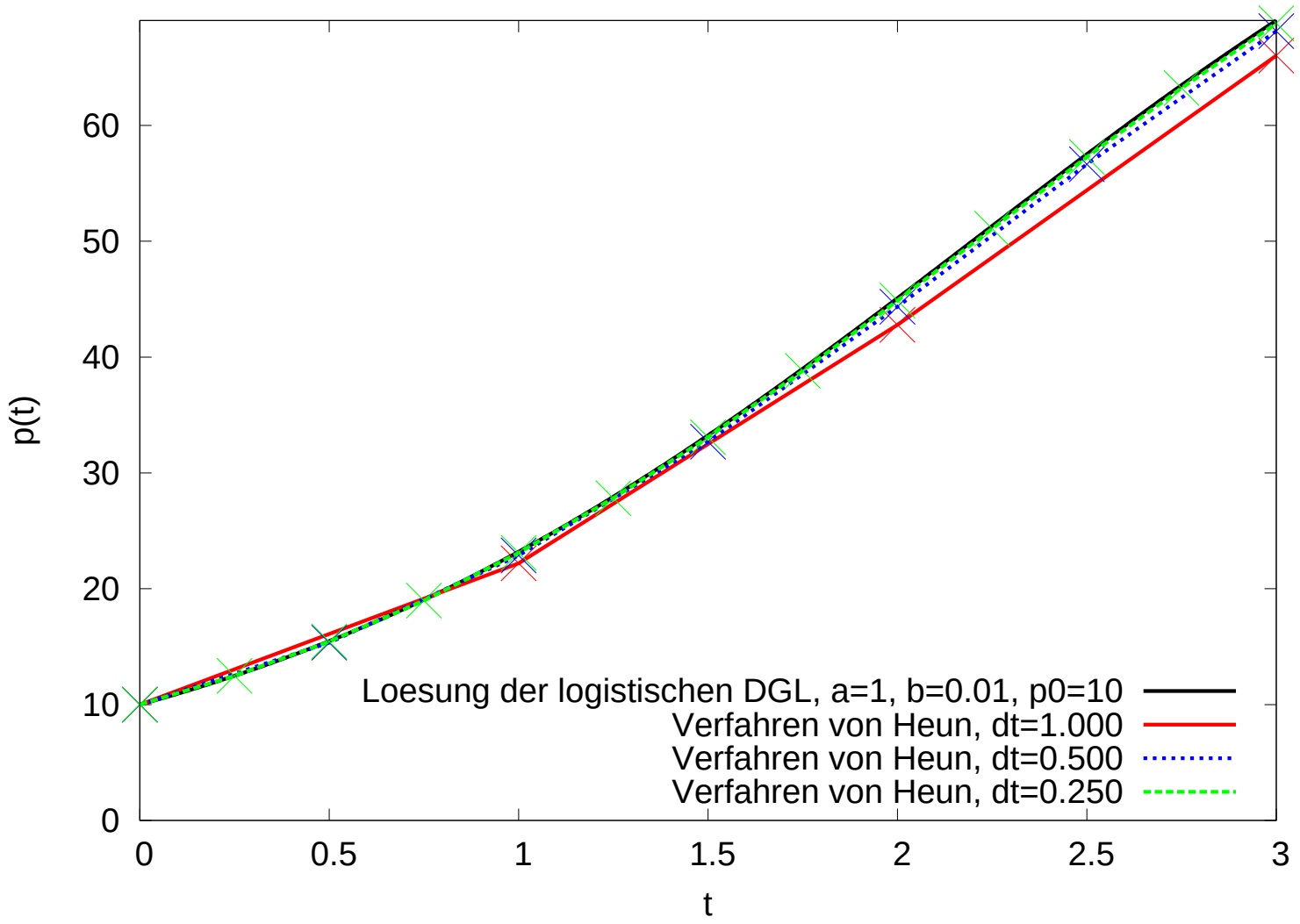
- Konvergenz des Euler-Verfahrens schlecht
- Idee zur Verbesserung: Nicht nur Steigung am aktuellen Punkt
- Mittelwert aus Steigung am aktuellen Punkt und Steigung am nächsten Punkt (bzw. dessen Näherung):

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

- 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 können wir schon?

Ausblick

Was können wir schon?

- Codeblöcke separieren
- Bedingungen
- Schleifen

Ausblick

Was können wir schon?

- Codeblöcke separieren
- Bedingungen
- Schleifen

Was fehlt?

Ausblick

Was können wir schon?

- Codeblöcke separieren
- Bedingungen
- Schleifen

Was fehlt?

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

Ausblick

Was können wir schon?

- Codeblöcke separieren
- Bedingungen
- Schleifen

Was fehlt?

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

Lösung?

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

Teil IV

Funktionen und Module

Wozu Funktionen?

Wo sie schon verwendet wurden:

- Mathematische Funktionen: `sqrt(s)`, `sin(s)`, `exp(x)`, `...`
- Methoden von Sequenzen: `s.islower()`, `l.append(x)`, `...`
- Erzeugung von Listen: `range(x)`
- Typ und Identität: `type(x)`, `id(x)`
- Alle bisherigen haben keinen oder einen Übergabewert und einen Rückgabewert

Wozu Funktionen?

Wo sie schon verwendet wurden:

- Mathematische Funktionen: `sqrt(s)`, `sin(s)`, `exp(x)`, ...
- Methoden von Sequenzen: `s.islower()`, `l.append(x)`, ...
- Erzeugung von Listen: `range(x)`
- Typ und Identität: `type(x)`, `id(x)`
- Alle bisherigen haben keinen oder einen Übergabewert und einen Rückgabewert

Vorteile

- Zerlegung eines großen Problems in viele kleine
- Zusammenfassung von Anweisungsblöcken
- Klare Abgrenzung von Funktionalitäten
- Vermeidung von Code-Duplikation
- Höhere Code-Lesbarkeit (Wenn man statt `sin(x)` jedesmal den kompletten Code ...)
- ...

Syntax von Funktionen

- Funktionen werden definiert mit dem Schlüsselwort `def`
- Nach Funktionsname und Parameterliste kommt ein Doppelpunkt.
- Der darauf folgende Block (Einrückung!) wird bei Aufruf der Funktion ausgeführt.
- Ausführung bis zum Ende des Blocks oder bis zu einen `return`
- Rückgabewert kann beliebiger Typ sein (Zahl, Liste, String, ...)
- Ohne `return` oder ohne Wert wird `None` zurückgegeben

```
>>> def hi():
>>>     print "Hello World!"
>>>
>>> hi()
Hello World
>>> a = hi()
Hello World
>>> type(a)
<type 'NoneType'>
```

Funktionen sind Objekte

```
>>> def factorial(n):
>>>     fac = 1
>>>     for i in range(2, n+1):
>>>         fac = fac*i
>>>     return fac
>>>
>>> factorial(3)
6
>>> f = factorial
>>> f(4)
24
>>> type(f)
<type 'function'>
```

Funktionsparameter

- Parameter stehen in runden Klammern hinter dem Funktionsnamen.
- Formalparameter: Bei der Definition der Funktion
- Aktualparameter: Folge von Ausdrücken beim Aufruf der Funktion
- Beliebig viele Parameter möglich, normalerweise gleich viele Formal- und Aktualparameter

```
>>> def potenziere(basis, exponent):
>>>     return basis**exponent
>>>
>>> potenziere(2,10)
1024
```

Standardwerte für Parameter

- Formalparameter können mit Standardwerten belegt werden.
- Zuerst alle Formalparameter ohne Standardwerte, dann die mit

```
def potenziere(basis=2, exponent):   # Falsch
def potenziere(basis, exponent=10):
def potenziere(basis=2, exponent=10):
```

```
>>> potenziere(2,10) # (2,10)
>>> potenziere(2) # (2,10)
>>> potenziere(exponent=7) # (2,7)
```

- Für diese Funktion sind Standardwerte also relativ nutzlos.
- Bei vielen Parametern oder bestimmten Situationen kann es aber hilfreich sein.

Rückgabewerte

- Bei mehreren Rückgabewerten wird ein Tupel zurückgegeben.

```
>>> def return_two():
>>>     return "x", "y"
>>>
>>> h = return_two();   # h = ('x', 'y')
>>> (a,b) = return_two() # a = 'x', b = 'y'
```

Rückgabewerte

- Bei mehreren Rückgabewerten wird ein Tupel zurückgegeben.

```
>>> def return_two():
>>>     return "x", "y"
>>>
>>> h = return_two();   # h = ('x', 'y')
>>> (a,b) = return_two() # a = 'x', b = 'y'
```

- Eine Liste oder explizites Tupel ist ein einzelner Wert.

```
>>> def primfaktoren(x):
>>>   l=[]; n=2
>>>   while n <= x:
>>>     while x%n==0:
>>>       x /= n
>>>       l.append(n)
>>>     n += 1
>>>   return l
```

Gültigkeitsbereich

Funktionen

- Erinnerung: Python ist eine interpretierte Sprache!
- Funktionen müssen definiert sein, bevor sie aufgerufen werden können.

Gültigkeitsbereich

Funktionen

- Erinnerung: Python ist eine interpretierte Sprache!
- Funktionen müssen definiert sein, bevor sie aufgerufen werden können.

Variablen

- Unterscheidung zwischen lokalen und globalen Variablen
- Eine Variable, die in einer Funktion definiert (an einen Wert gebunden) wird, ist lokal und insbesondere außerhalb nicht sichtbar.
- Grundregel: GLOBALE VARIABLEN VERMEIDEN!!!
- Globale Variablen können gelesen werden.
- Um globale Variablen in einer Funktion zu schreiben, wird das Schüsselwort `global` verwendet.

Beispiele

```
>>> def zaehle():
>>>     global anzahl
>>>     anzahl += 1
>>>
>>> anzahl = 0
>>> zaehle(); zaehle(); zaehle()
>>> anzahl
3
```

Beispiele

```
>>> def zaehle():
>>>     global anzahl
>>>     anzahl += 1
>>>
>>> anzahl = 0
>>> zaehle(); zaehle(); zaehle()
>>> anzahl
3
```

```
>>> def no_effect():
>>>     xyz = 5**2
>>>
>>> no_effect()
>>> xyz
NameError: name 'xyz' is not defined
```