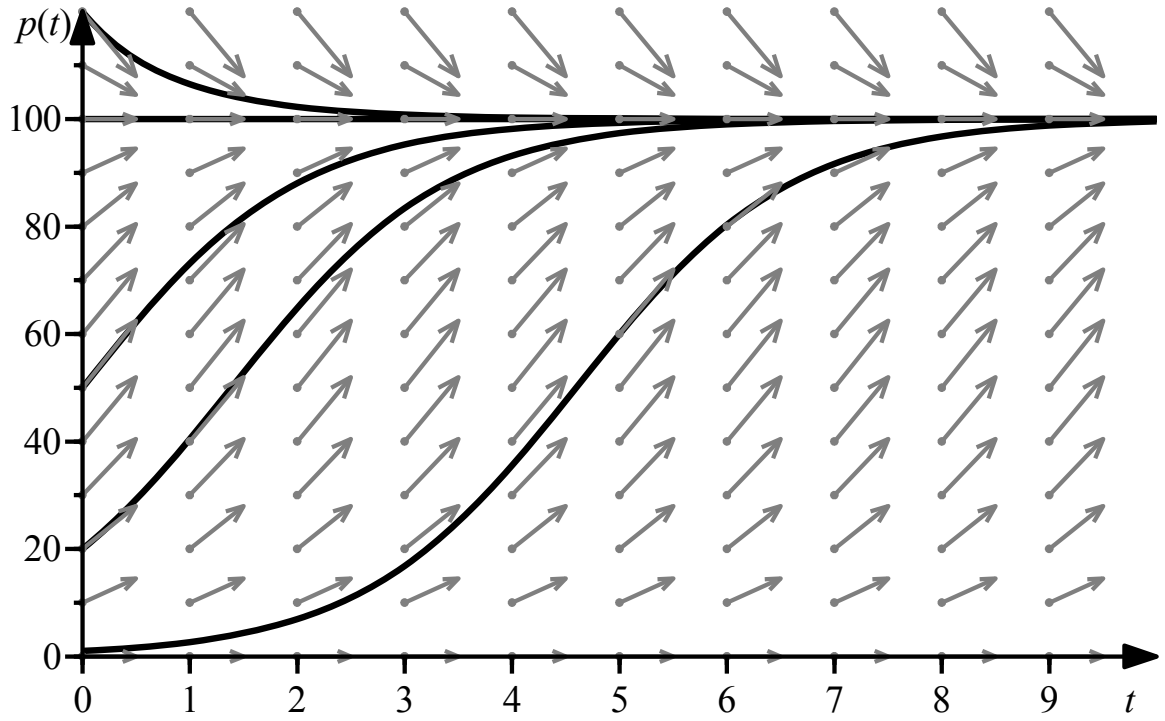
Beispiel: Numerische Lösung von ODEs

- 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)
```

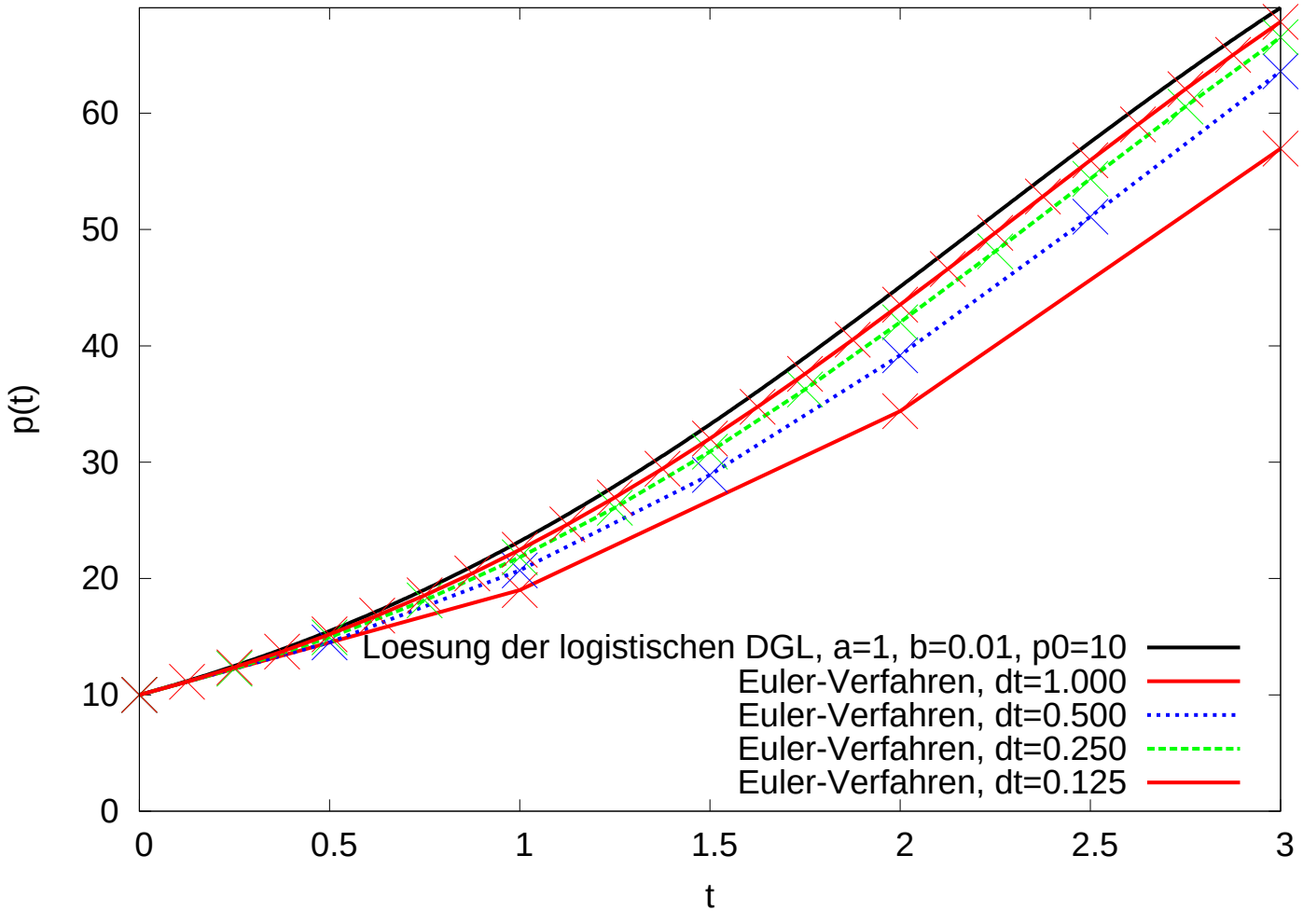
```
   0.0   10.0
   0.5   15.5
   1.0   23.2
   1.5   33.2
   2.0   45.1
   2.5   57.5
   3.0   69.1
   3.5   78.6
   4.0   85.8
   4.5   90.9
   5.0   94.3
```

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 $\dot{p}(t) = \frac{p(t+\delta t)-p(t)}{\delta t}$
- $\Rightarrow p(t+\delta t) \doteq p(t) + \delta t \cdot f(t, p(t))$
- 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 67

Numerische Lösung: Verfahren von Heun

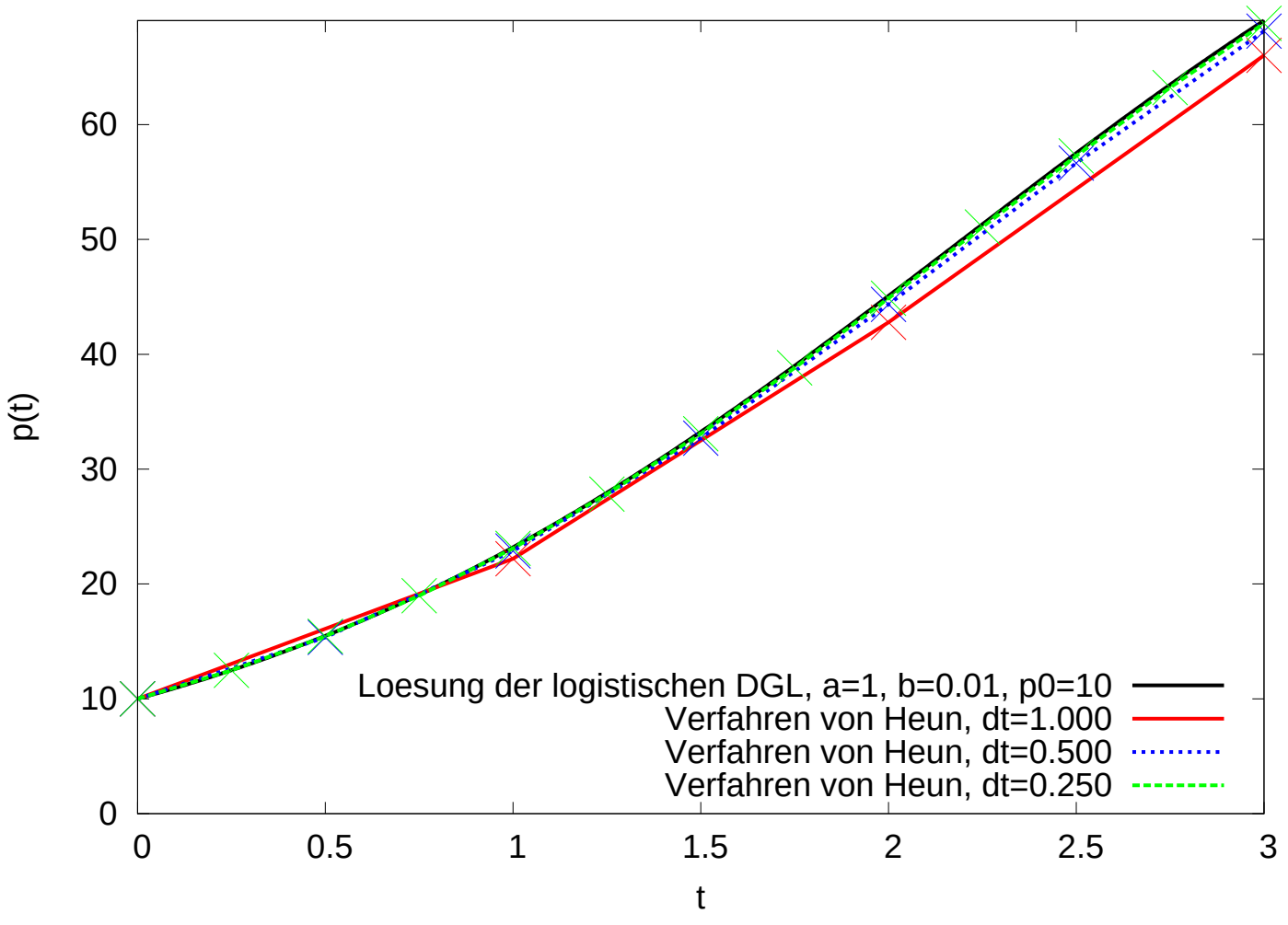
- Konvergenz des Euler-Verfahrens schlecht
- Idee zur Verbesserung: Nicht nur Steigung am aktuellen Punkt
- Mittelwert aus Steigung am aktuellen Punkt und Steigung am nächsten Punkt (bzw. dessen Näherung):

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

- 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

Ausblick

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) # (2,2)
>>> 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
```

Docstrings

- Ein Grund für Funktionen: Funktionalität kapseln
- Jemand, der die Funktion nicht geschrieben hat, soll sie verwenden können. $\Rightarrow$ Dokumentation nötig!
- Ein String (ein- oder mehrzeilig) nach dem Header ist für interaktive Dokumentation vorgesehen.
- Python ignoriert den String (Ausdruck ohne Zuweisung) bei der Funktionsausführung.
- Das Hilfesystem (und der Code-Leser) kann den String verwenden.
- Nicht nur für Funktionen, sondern auch für Klassen, Module, ...

```
>>> def blubb():
>>>     "Unsinnige Funktion, die 'blubb' ausgibt"
>>>     print "blubb"
>>> blubb()
blubb
>>> help(blubb)
```

Call by Reference vs. Call by Value

- Call by Reference: „Adresse“ (id) wird übergeben
- Call by Value: Wert der Variablen wird übergeben
- In Python immer Call by Reference
- Aber ACHTUNG: Zuweisung ändert Referenz innerhalb der Funktion!

```
>>> def add_one(x):
>>>     x = x + 1
>>>
>>> x = 3
>>> add_one(x)
>>> x
3
```

Call by Reference vs. Call by value (2)

- Nach der Zuweisung zeigt die lokale Variable auf ein neues Objekt.
- Mit Methoden kann das bisherige Objekt verändert werden.

```
>>> def append1(l):
>>>   l = l + [4]
>>>
>>> def append2(l):
>>>   l.append(4)
>>>
>>> l = [1,2,3]
>>> append1(l); l
[1,2,3]
>>> append2(l); l
[1,2,3,4]
```

Anonyme Funktionen

- Normale Funktionen müssen separat definiert werden.
- Sie bekommen einen Namen, mit dem auf sie zugegriffen wird.

```
>>> def inc(i):
>>>     return i+1
>>> inc(3)
4
```

- Namenlose/Anonyme Funktionen erhalten keinen Namen.
- Sie können an beliebigen Stellen in den Code eingebaut werden.

```
>>> inc = lambda(i): i+1
>>> inc(3)
4
```

map

- `map` wendet eine Funktion auf alle Elemente einer Liste an
- Die Funktion kann natürlich auch anonym sein.

```
>>> L = range(10)
>>> map(lambda x: x*x+1, L)
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

- In diesem Fall wäre das sogar noch kürzer mit list comprehensions (nächste Folie) gegangen.

map

- `map` wendet eine Funktion auf alle Elemente einer Liste an
- Die Funktion kann natürlich auch anonym sein.

```
>>> L = range(10)
>>> map(lambda x: x*x+1, L)
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

- In diesem Fall wäre das sogar noch kürzer mit list comprehensions (nächste Folie) gegangen.

filter

- `filter` wendet ebenfalls eine Funktion auf alle Elemente einer Liste an.
- Diejenigen Elemente, für die die Funktion `True` zurückgibt, werden zurückgegeben.

```
>>> filter(lambda x: x%2==0, L)
[0, 2, 4, 6, 8]
```

List Comprehension

- Einfache Listen lassen sich z.B. mit `range` erzeugen.
- Für komplexere Listen gibt es list comprehension.

```
>>> b = [i**2 for i in range(0,10) if i%2==0]
>>> b
[0, 4, 16, 36, 64]
```

List Comprehension

- Einfache Listen lassen sich z.B. mit `range` erzeugen.
- Für komplexere Listen gibt es list comprehension.

```
>>> b = [i**2 for i in range(0,10) if i%2==0]
>>> b
[0, 4, 16, 36, 64]
```

- list comprehensions können geschachtelte Schleifen nachbilden:

```
>>> c = [(i,j) for i in range(1,5) if i%2==0
                for j in range(1,5) if j%2!=0]
>>> c
[(2, 1), (2, 3), (4, 1), (4, 3)]
```

List Comprehension (2)

- Generelle Syntax:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN]
```

- Falls keine Bedingung angegeben: automatisch `True`

List Comprehension (2)

- Generelle Syntax:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN]
```

- Falls keine Bedingung angegeben: automatisch `True`
- Syntax entspricht:

```
s = []
for item1 in iterable1:
  if condition1:
    for item2 in iterable2:
      if condition2:
        ....
        for itemN in iterableN:
          if conditionN: s.append(expression)
```

Beispiel: Numerische Integration (Quadratur)

Berechnung des Integrals

$$F_1(f, a, b) := \int_a^b f(x)\, dx$$

Beispiel: Numerische Integration (Quadratur)

Berechnung des Integrals

$$F_1(f, a, b) := \int_a^b f(x)\, dx$$

Verschiedene numerische Verfahren; hier jetzt:

Trapezregel

- Bestimmung des Funktionswertes an beiden Rändern
- Multiplikation des Mittelwerts mit der Intervallbreite
- $F_1 \approx T := (b - a) \cdot \frac{f(a)+f(b)}{2}$
- Polynom 1. Ordnung (Fläche entspricht Trapez)

Beispiel: Numerische Integration (Quadratur)

Berechnung des Integrals

$$F_1(f, a, b) := \int_a^b f(x)\, dx$$

Verschiedene numerische Verfahren; hier jetzt:

Trapezregel

- Bestimmung des Funktionswertes an beiden Rändern
- Multiplikation des Mittelwerts mit der Intervallbreite
- $F_1 \approx T := (b - a) \cdot \frac{f(a)+f(b)}{2}$
- Polynom 1. Ordnung (Fläche entspricht Trapez)

Quadraturfehler

$$|T - F_1| \leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot (b - a)^3$$

Summenregeln

- Bei großen Intervallen wird auch der Fehler sehr groß.
- $\Rightarrow$ Unterteilung in n kleine Intervalle
- Jedes Teilintervall hat Länge $h = (b - a)/n$

Summenregeln

- Bei großen Intervallen wird auch der Fehler sehr groß.
- $\Rightarrow$ Unterteilung in n kleine Intervalle
- Jedes Teilintervall hat Länge $h = (b - a)/n$

Trapezsumme

$$TS := h \cdot \left[\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(a + ih) + \frac{f(b)}{6} \right]$$

Implementierung der Trapezsumme

```
def trapezsumme(f, a, b, n):
  h = (b-a)/n
  sum = (f(a) + f(b))/2
  for i in range(1,n):
    sum = sum + f(a + h*i)
  return sum*h
```

Implementierung der Trapezsumme

```
def trapezsumme(f, a, b, n):
  h = (b-a)/n
  sum = (f(a) + f(b))/2
  for i in range(1,n):
    sum = sum + f(a + h*i)
  return sum*h
```

Funktion zum Test

- Funktion, die analytisch leicht zu integrieren ist
- $\Rightarrow$ Fehler lässt sich leicht berechnen

```
import math
def f(x):
  return math.sin(math.pi*x)
def f_int(x):
  return -math.cos(math.pi*x)/math.pi
```

Berechnung mit unterschiedlicher Genauigkeit: Fehler empirisch

- $n = 2^1 ... 2^6$

```
a_bsp = 0.
b_bsp = 1.
exakt = f_int(b_bsp) - f_int(a_bsp)
for i in range(1,7):
    n = 2**i
    t = trapezsumme(f, a_bsp, b_bsp, n)
    print '%4d␣%-12.6g␣%-12.6g' % (n, t, exakt-t)
```

n	Näherung	Fehler
2	0.5	0.13662
4	0.603553	0.0330664
8	0.628417	0.00820234
16	0.634573	0.00204662
32	0.636108	0.000511409
64	0.636492	0.000127837

Fehler der Trapezsumme: Fehler analytisch

- In jedem Teilintervall ist der Fehler

$$\leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot h^3$$

- Bei $n = (b-a)/h$ Intervallen ist damit der Gesamtfehler

$$|TS - F_1| \leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot (b-a) \cdot h^2$$

- Verdopplung des Rechenaufwands (Halbierung von h) reduziert den maximalen Fehler um den Faktor 4