Computergrundlagen
Programmieren lernen — in Python

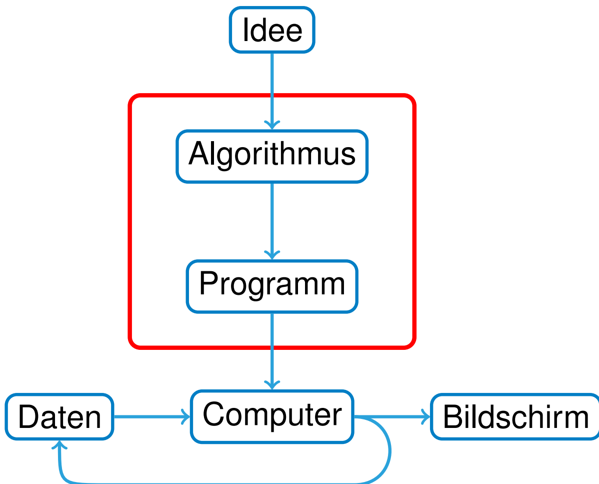
Axel Arnold

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2011/12

Was ist Programmieren?

INSTITUTE FOR COMPUTATIONAL PHYSICS



Algorithmus

Wikipedia:

Ein Algorithmus ist eine aus endlich vielen Schritten bestehende eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen.

Ein Problem besteht dabei meist aus gegebenen Daten und einer Fragestellung. Beispiel:

Gegeben

- Liste aller Teilnehmer der Vorlesung

Fragestellung

- ~~Wer wird die Klausur bestehen?~~ ↯
- Wieviele Studenten haben nur einen Vornamen?
- Wessen Matrikelnummer ist eine Primzahl?

Programm

Ein Programm ist eine Realisation eines Algorithmus in einer bestimmten Programmiersprache.

- Es gibt derzeit mehrere 100 verschiedene Programmiersprachen
- Die meisten sind *Turing-vollständig*, können also alle bekannten Algorithmen umsetzen

Softwareentwicklung und Programmieren

- Entwickeln der Algorithmen
  - Aufteilen in einfachere Probleme
  - Wiederverwendbarkeit
- Umsetzen in einer passenden Programmiersprache
- Das Schwierige sind die Algorithmen!

Von der Idee zum Programm

Schritte bei der Entwicklung eines Programms

1. **Problemanalyse**
   - Was soll das Programm leisten?
     Z.B. eine Nullstelle finden, Molekulardynamik simulieren
   - Was sind Nebenbedingungen?
     Z.B. ist die Funktion reellwertig? Wieviele Atome?
2. **Methodenwahl**
   - Schrittweises Zerlegen in Teilprobleme (Top-Down-Analyse)
     Z.B. Propagation, Kraftberechnung, Ausgabe
   - Wahl von Datentypen und -strukturen
     Z.B. Listen oder Tupel? Wörterbuch?
   - Wahl der Rechenstrukturen (Algorithmen)
     Z.B. Newton-Verfahren, Regula falsi
   - Wahl der Programmiersprache

Von der Idee zum Programm

Schritte bei der Entwicklung eines Programms

3. **Implementation und Dokumentation**
   - Programmieren und *gleichzeitig* dokumentieren
   - Kommentare und externe Dokumentation (z.B. Formeln)
4. **Testen auf Korrektheit**
   - Funktioniert das Programm bei erwünschter Eingabe?
     Z.B. findet es eine bekannte Lösung?
   - Gibt es aussagekräftige Fehler bei falscher Eingabe?
     Z.B. vergessene Parameter, zu große Werte
5. **Testen auf Effizienz**
   - Wie lange braucht das Programm bei beliebigen Eingaben?
   - Wieviel Speicher braucht es?
6. Meist wieder zurück zur Problemanalyse, weil man etwas vergessen hat ...

... und jetzt das Ganze in Python

- schnell zu erlernende, moderne Programmiersprache
  – tut, was man erwartet
- viele Standardfunktionen („all batteries included“)
- Bibliotheken für alle anderen Zwecke
- freie Software mit aktiver Gemeinde
- portabel, gibt es für fast jedes Betriebssystem
- entwickelt von Guido van Rossum, CWI, Amsterdam

Informationen zu Python

- Aktuelle Versionen 3.3.0 bzw. 2.7.3
- 2.x ist *noch* weiter verbreitet (z.B. Python 2.7.3 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`

Python starten

Aus der Shell:

```
> python
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] 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

Institute for Computational Physics

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!

Beispiel: Fakultät

- **Problem**

  Gegeben: Eine ganze Zahl n

  Gesucht: Die Fakultät $c! = 1 \cdot 2 \cdots n$ von n

- **Implementation**

```
# calculate factorial 1*2*...*n
n = 5
factorial = 1
for k in range(1, n+1):
    factorial = factorial * k
print n, "! =", factorial
```

Ausgabe:

```
5 ! = 120
```

- Gegebene Daten (n=5) fix ins Programm eingefügt
  $\Longrightarrow$ später lernen wir, Daten einzulesen
- Was bedeuten die ganzen Befehle?

Datentyp: Ganzzahlen

INSTITUTE FOR COMPUTATIONAL PHYSICS

```
>>> print 42
42
>>> print -12345
-12345
>>> print 20/2
10
>>> print 3/2, -3/2
1 -2
```

- Klassische, mathematische Ganzzahlen
- Division liefert nur ganzzahligen Rest (anders in Python 3!)
- **print** gibt mit Komma auch mehrere Werte aus (nicht nur für Ganzzahlen)

Datentyp: (Fließ-)kommazahlen


```
>>> print 12345.000
12345.0
>>> print 6.023e23, 13.8E-24
6.023e+23 1.38e-23
>>> print 3.0/2
1.5
```

- Reelle Zahlen der Form $6{,}023 \cdot 10^{23}$
- `1.38e-23` steht z. B. für $1{,}38 \times 10^{-23}$
- Achtung: englische Schreibweise, Punkt statt Komma
- Keine Tausenderpunkte (oder -kommas)
- Endliche binäre Genauigkeit von Mantisse und Exponent
- `12345` $\neq$ `12345.0` (z. B. bei der Ausgabe)
- Hier nicht benutzt, da dann die Gleichung immer unendlich viele Lösungen hat

Datentyp: Zeichenketten

INSTITUTE FOR COMPUTATIONAL PHYSICS

```
>>> 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`
- `int(string)` konvertiert Zeichenkette in Ganzzahl
- entsprechend `float(string)` für Fließkomma

Sich etwas merken — Variablen

INSTITUTE FOR COMPUTATIONAL PHYSICS

```
>>> factorial = 2
>>> factor = 3
>>> print factorial, factor
2 3
>>> factorial = factorial * factor
>>> factor = 4
>>> print factorial, factor
6 4
```

- Werte können mit Namen belegt werden **und verändert**
- keine mathematischen Variablen, sondern Speicherplätze
- Daher ist `factorial = factorial * factor` kein Unsinn, sondern multipliziert `factorial` mit `factor`
- Die nachträgliche Änderung von `factor` ändert nicht `factorial`, das Ergebnis der vorherigen Rechnung!

Sich etwas merken — Variablen


```
>>> factorial = 2
>>> factor = 3
>>> print factorial, factor
2 3
>>> factorial = factorial * factor
>>> factor = 4
>>> print factorial, factor
6 4
```

- 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

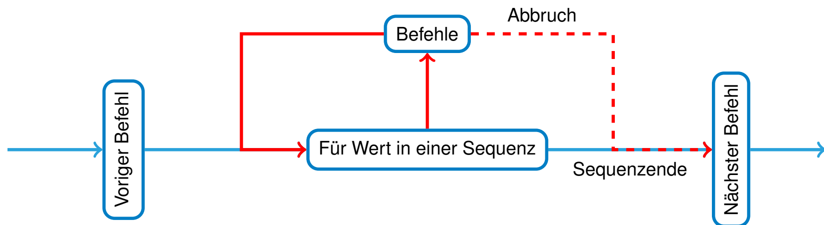
Institute for Computational Physics

+	Addition, bei Strings aneinanderfügen, z.B. 1 + 2 → 3, "a" + "b" → "ab"
-	Subtraktion, z.B. 1 - 2 → -1
*	Multiplikation, Strings vervielfältigen, z.B. 2 * 3 = 6, "ab" * 2 → "abab"
/	Division, bei ganzen Zahlen ganzzahlig, z.B. 3 / 2 → 1, -3 / 2 → -2, 3.0 / 2 → 1.5
%	Rest bei Division, z.B. 5 % 2 → 1
**	Exponent, z.B. 3**2 → 9, 0.1**3 → 0.001

- mathematische Präzedenz (Exponent vor Punkt vor Strich), z. B. `2**3 * 3 + 5` → $2^3 \cdot 3 + 5 = 29$
- Präzedenz kann durch runde Klammern geändert werden: `2**(3 * (3 + 5))` → $2^{3 \cdot 8} = 16.777.216$

for-Schleifen

Institute for Computational Physics



- Wiederholen eines Blocks von Befehlen
- *Schleifenvariable* nimmt dabei verschiedene Werte aus einer *Sequenz* (Liste) an
- Die abzuarbeitende Sequenz bleibt fest
- Kann bei Bedarf abgebrochen werden (Ziel erreicht, Fehler, ...)

Für jeden Studenten in den Computergrundlagen finde einen Übungsplatz

for-Schleifen in Python


```
>>> for v in range(1, 3): print v
1
2
>>> b = 0
>>> for a in range(1, 100):
...     b = b + a
>>> print b
4950
>>> print 100 * (100 - 1) / 2
4950
>>> print range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Das aktuelle Element steht in den Variablen v bzw. a
- `range(k,l)` ist eine Liste der Zahlen a mit $k \leq a < l$
- später lernen wir, Listen zu erstellen und verändern

Beispiel: Pythagoreische Zahlentripel

INSTITUTE FOR COMPUTATIONAL PHYSICS

- **Problem**

  Gegeben: Eine ganze Zahl c

  Gesucht: Zwei Zahlen a, b mit $a^2 + b^2 = c^2$

  1. Verfeinerung: $a = 0$, $b = c$ geht immer $\Rightarrow a,b > 0$
  2. Verfeinerung: Was, wenn es keine Lösung gibt? Fehlermeldung

- **Methodenwahl**

  - Es muss offenbar gelten: $a < c$ und $b < c$
  - O.B.d.A. sei auch $a \leq b$, also $0 < a \leq b < c$
  - Durchprobieren aller Paare a, b mit $0 < a < c$ und $a \leq b < c$:

$c = 5 \implies c^2 = 25$, $a^2 + b^2 =$

	1	2	3	4
1	2	5	10	17
2		8	13	20
3			18	**25**
4				32

Implementation

INSTITUTE FOR COMPUTATIONAL PHYSICS

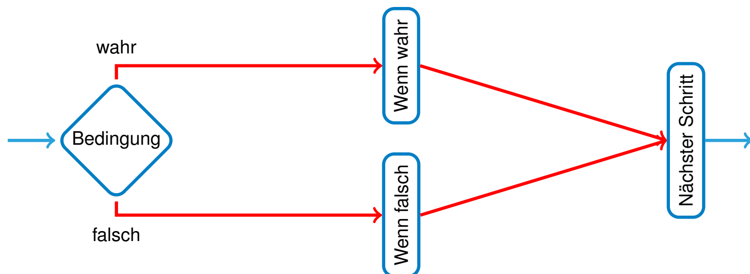
```
# Try to find a pythagorean triple a^2 + b^2 = c^2.
# parameter: rhs number, should be an integer larger than 0
c = 5

# try all possible pairs
for a in range(1,c):
    for b in range(a,c):
        if a**2 + b**2 == c**2:
            print "%d^2 + %d^2 = %d^2" % (a, b, c)
            break
```

Ausgabe:

```
3^2 + 4^2 = 5^2
```

- Gegebene Daten (c=5) fix ins Programm eingefügt
  $\Longrightarrow$ später lernen wir, Daten einzulesen
- Was bedeuten die ganzen Befehle?

Bedingte Ausführung



- Das Programm kann auf Werte von Variablen verschieden reagieren
- Wird als *Verzweigung* bezeichnet
- Auch mehr Äste möglich (z.B. < 0, $= 0$, > 0)

Student hat mehr als 50% Punkte? $\Longrightarrow$ zur Klausur zulassen

if: bedingte Ausführung in Python


```
>>> 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

Logische Ausdrücke

`==, !=`	Test auf (Un-)Gleichheit, z.B. `2 == 2` → `True`, `1 == 1.0` → `True`, `2 == 1` → `False`
`<,>,<=,>=`	Vergleich, z.B. `2 > 1` → `True`, `1 <= -1` → `False`
`or, and`	Logische Verknüpfungen „oder" bzw. „und"
`not`	Logische Verneinung, z.B. **`not`** `False == True`

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

a	*b*	*a* und *b*	*a* oder *b*
True	True	True	True
False	True	False	True
True	False	False	True
False	False	False	False

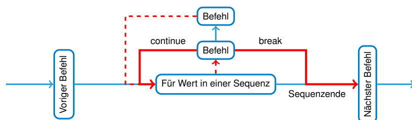
- Präzedenz: logische Verknüpfungen vor Vergleichen

`3 > 2` **`and`** `5 < 7` → `True`, `1 < 1` **`or`** `2 >= 3` → `False`

break und continue: Schleifen beenden


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

- Beide überspringen den Rest des Schleifenkörpers
- **break** bricht die Schleife ganz ab
- **continue** springt zum Anfang
- Im Beispiel beenden wir, sobald das erste Zahlenpaar passt



Blöcke und Einrückung

Institute for Computational Physics

```
>>> b = 0
>>> for a in range(1, 4):
...     b = b + a
...     print b
4
6
9
>>> b = 0
>>> for a in range(1, 3): b = b + a
... print b
9
```

- Alle *gleich eingerückten* Befehle gehören zum Block
- Einzeilige Blöcke können auch direkt hinter den Doppelpunkt
- Einrücken durch Leerzeichen oder Tabulatoren (einfacher)

Blöcke und Einrückung

- 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!

Formatierte Ausgabe: der %-Operator

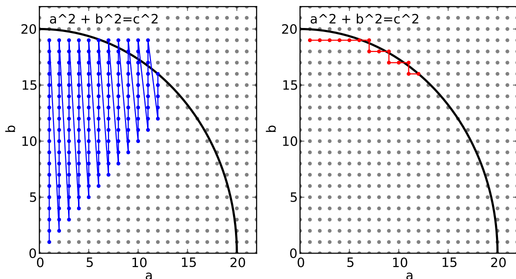

```
>>> print "%d^2 + %d^2 = %05d^2" % (3, 4, 5)
3^2 + 4^2 = 00005^2
>>> print "Strings %s %10s" % ("Hallo", "Welt")
Strings Hallo       Welt
>>> print "Fliesskomma %e |%+8.4f| %g" % (3.14, 3.14, 3.14)
Fliesskomma 3.140000e+00 | +3.1400| 3.14
```

- Im Beispiel bei der Ausgabe benutzt
- Ersetzen von %-Platzhaltern in einem String
- %d: Ganzahlen (Integers)
- %s: einen String einfügen
- %e, %f, %g: verschiedene Fließkommaformate
- %*x*[defgs]: füllt auf *x* Stellen auf
- %*x*.*y*[efg]: zeige *y* Nachkommastellen

Testen auf Effizienz

Zahl (alle ohne Lösung)	1236	12343	123456
Zeit	0,2s	18,5s	30m

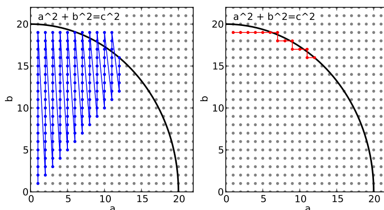
- Das ist sehr langsam! Geht das besser? Ja!



- Statt alle Paare auszuprobieren, suche nur in der Umgebung des Halbkreises!

Testen auf Effizienz

INSTITUTE FOR COMPUTATIONAL PHYSICS



- **Methodenwahl**, effizienterer Algorithmus:
  - Sei zunächst $a = 1$ und $b = c - 1$
    - Ist $a^2 + b^2 > c^2$, so müssen wir b verringern, und wir wissen, dass es keine Lösung mit $b = c - 1$ gibt
    - Ist $a^2 + b^2 < c^2$, so müssen wir a erhöhen und wir wissen, dass es keine Lösung mit $a = 1$ gibt
  - Mit der selben Argumentation kann man fortfahren
  - Wir haben alle Möglichkeiten getestet, wenn $a > b$
- braucht maximal $|c|/2$ statt $c(c-1)/2$ viele Schritte

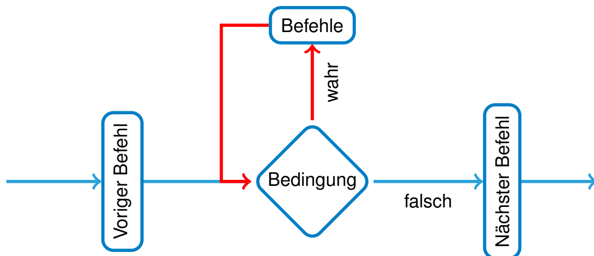
Neue Implementation


```
# parameter: rhs number, should be an integer larger than 0
c = 5
# starting pair
a = 1
b = c - 1
while a <= b:
    if   a**2 + b**2 < c**2: a += 1
    elif a**2 + b**2 > c**2: b -= 1
    else:
        print "%d^2 + %d^2 = %d^2" % (a, b, c)
        break
```

- Effizienz dieser Lösung:

Zahl	12343	123456	1234561	12345676	123456789
Zeit	0.04s	0.08s	0.65s	6.2s	62.4s
Zeit (alt)	0,2s	18,5s	30m	-	-

while-Schleifen



- Wiederholte Ausführung ähnlich wie for-Schleifen
- Keine *Schleifenvariable*, sondern Schleifenbedingung
- Ist die Bedingung immer erfüllt, kommt es zur **Endlosschleife**

Solange $a > 0$, ziehe eins von a ab

Solange noch Zeilen in der Datei sind, lese eine Zeile

while-Schleifen in Python


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

- Führt den Block solange aus, wie die Bedingung wahr ist
- Block wird nicht ausgeführt, wenn Bedingung sofort verletzt ist:

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

Parameter einlesen

Institute for Computational Physics

```
import sys
# get integer c from the command line
try:
    c = int(sys.argv[1])
except:
    sys.stderr.write("usage: %s <c>\n" % sys.argv[0])
    exit(-1)
print c
```

- Bisher ist c fest im Programm $\Longrightarrow$
  wenn wir c ändern wollen, müssen wir das Programm ändern
- Besser von der Kommandozeile lesen!
- So können wir das Programm direkt vom Terminal benutzen
- Wir brauchen keinen Editor, wenn es mal tut

Parameter einlesen


```
import sys
# get integer c from the command line
try:
    c = int(sys.argv[1])
except:
    sys.stderr.write("usage: %s <c>\n" % sys.argv[0])
    exit(-1)
print c
```

- **`import`** `sys` lädt das `sys`-Modul, dazu später mehr
- `sys.argv[i]` gibt dann den i-ten Parameter des Programms
- `sys.argv[0]` ist der Name des Skripts
- `int(string)` konvertiert Zeichenkette in Ganzzahl
- Der **`except`**-Block wird nur ausgeführt, wenn es beim Lesen von c einen Fehler gab

Parameter einlesen: das optparse-Modul

INSTITUTE FOR COMPUTATIONAL PHYSICS

```
from optparse import OptionParser

parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="Ausgabe in FILE", metavar="FILE")

options, args = parser.parse_args()
```

- Schöner: `optparse` liest Kommandozeilenflags
- `add_option` spezifiziert eine Option mit
  - kurzer und langer Form des Namens („-f“,„-file“)
  - einer Zielvariablen für den vom Benutzer gegebenen Wert
  - einem zugehörigen Hilfetext
- Bei Aufruf `python parse.py -f test a b c` ist:
  - `options` ein Objekt mit `options.filename = "test"`
  - `args = ['a', 'b', 'c']`

Beispiel: Sortieren

Gegeben: Liste $A = a_o, \ldots, a_N$

Gesucht: Liste $A' = a'_o, \ldots, a'_N$ mit denselben Elementen wie A, aber sortiert, also $a'_0 \leq a'_1 \leq \ldots \leq a'_N$

- Datentyp ist egal, so lange $\leq$ definiert ist
- In Python ganz einfach:
  - `A.sort()` $\Longrightarrow$ A wird umsortiert
  - `B = sorted(A)` $\Longrightarrow$ A bleibt gleich, B ist die sortierte Liste

```
>>> A = [2,1,3,5,4]
>>> print sorted(A), A
[1, 2, 3, 4, 5] [2, 1, 3, 5, 4]
>>> A.sort()
>>> print A
[1, 2, 3, 4, 5]
```

- Aber was passiert da nun? Wie sortiert der Computer?

Sortieralgorithmus 1: Bubblesort

Idee

- paarweises Sortieren, größere Werte steigen wie Blasen auf
- ein Durchlauf aller Elemente $\Longrightarrow$ größtes Element ganz oben
- m Durchläufe $\Longrightarrow$ m oberste Elemente einsortiert
- $\Longrightarrow$ nach spätestens N Durchläufen fertig
- fertig, sobald nichts mehr vertauscht wird

Effizienz

- im Schnitt $N/2$ Durchläufe mit $N/2$ Vergleichen
  $\Longrightarrow$ Laufzeit $\mathcal{O}(N^2)$
- Auch Worst Case $N - 1 + N - 2 + ... + 1 = \mathcal{O}(N^2)$
- Kein zusätzlicher Speicherbedarf

Implementierung

Institute for Computational Physics

```
def sort(A):
    "sort list A in place"
    N = len(A)
    for round in range(N):
        changed = False
        for k in range(N - round - 1):
            if A[k] > A[k+1]:
                A[k], A[k + 1] = A[k+1], A[k]
                changed = True
        if not changed: break

A = [1,3,2,5,4]
sort(A)
print A
```

Ausgabe:

```
[1, 2, 3, 4, 5]
```

Listen in Python

Institute for Computational Physics

```
>>> 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 Klammmern
- können Daten *verschiedenen* Typs enthalten
- `liste[i]` bezeichnet das `i`-te Listenelement, negative Indizes starten vom Ende
- append fügt ein Element an eine Liste an
- **`del`** löscht ein Listenelement

Listen in Python

INSTITUTE FOR COMPUTATIONAL PHYSICS

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

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

Vorsicht: Flache Kopien!


```
>>> kaufen = [ "Muesli", "Milch", "Obst" ]
```

Flache Kopie:

```
>>> merken = kaufen
>>> del kaufen[-1]
>>> print merken
['Muesli', 'Sahne']
```

Subliste, echte Kopie:

```
>>> merken = kaufen[:]
>>> del kaufen[-1]
>>> print merken
['Muesli', 'Sahne', 'Obst']
```

„=“ macht in Python flache Kopien von Listen!

- Flache Kopien (shallow copies) *verweisen* auf dieselben Daten
- Änderungen an einer flachen Kopie betreffen auch das Original
- Sublisten sind echte Kopien (deep copies, weil alles kopiert wird)
- daher ist `kaufen[:]` eine echte Kopie von `kaufen`

Vorsicht: Flache Kopien!


```
>>> element = []
>>> liste = [ element, element ]      # flache Kopien!
>>> liste[0].append("Hallo")
>>> print liste
[['Hallo'], ['Hallo']]
```

Mit echten Kopien (deep copies)

```
>>> liste = [ element[:], element[:] ] # tiefe Kopien!
>>> liste[0].append("Welt")
>>> print liste
[['Hallo', 'Welt'], ['Hallo']]
```

- Auch Listen in Listen sind flache Kopien
  und können daher mehrmals auf dieselben Daten verweisen
- kann zu unerwarteten Ergebnissen führen

Tupel: unveränderbare Listen

Institute for Computational Physics

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

- komma-getrennt in runden Klammern
- Solange eindeutig, können die Klammern weggelassen werden
- können nicht verändert werden
- ansonsten wie Listen einsetzbar
- Zeichenketten sind Tupel von Zeichen

Tupel: unveränderbare Listen

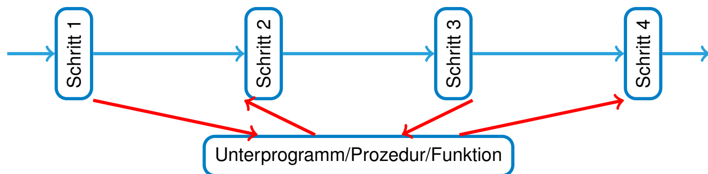

```
>>> options, args = parser.parse_args()
>>> A=1
>>> B=2
>>> A, B = B, A
>>> print A, B
2 1
```

So hingegen nicht:

```
>>> A = B
>>> B = A
>>> print A, B
1 1
```

- Listen und Tupel können links vom Gleichheitszeichen stehen
- Elemente werden der Reihe nach zugeordnet
- `A,B = B,A` tauscht also die Werte zweier Variablen aus (Tupelzuweisung!)

Funktionen



- Funktionen (Unterprogramme, Prozeduren) unterbrechen die aktuelle Befehlskette und fahren an anderer Stelle fort
- Kehren an ihrem Ende wieder zur ursprünglichen Kette zurück
- Funktionen können selber wieder Funktionen aufrufen
- Vermeiden Code-Duplikation
  - kürzerer Code
  - besser wartbar, Fehler müssen nur einmal verbessert werden

Funktionen in Python


```
>>> 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 verhalten sich wie Variablen
- Beim Aufruf bekommen die Argumentvariablen Werte in der Aufrufreihenfolge
- Der Funktionskörper ist wieder ein Block

Lokale Variablen


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

- neue Variablen innerhalb einer Funktion sind *lokal*
- existieren nur während der Funktionsausführung
- globale Variablen können nur gelesen werden

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

Vorgabewerte und Argumente benennen

Institute for Computational Physics

```
>>> 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

return: eine Funktion beenden

Institute for Computational Physics

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

- **`return`** beendet die Funktion sofort (vgl. **`break`**)
- eine Funktion kann einen Wert zurückliefern
- der Wert wird bei **`return`** spezifiziert

Dokumentation von Funktionen


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

def min(a, b):
    """
Gibt das Minimum von a und b aus. Funktioniert
ansonsten genau wie die Funktion max.
    """
    if a < b: print a
    else:     print b
```

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

Sortieralgorithmus 2: Quicksort

Idee

- Teile und Herrsche (Divide & Conquer): Aufteilen in zwei kleinere Unterprobleme
- Liste ist fertig sortiert, falls $N \leq 1$
- wähle *Pivot*- (Angel-) element p
- erzeuge Listen K und G der Elemente kleiner/größer als p
- sortiere die beiden Listen K und G
- Ergebnis ist die Liste $K \oplus \{p\} \oplus G$

Effizienz

- im Schnitt $\log_2 N$-mal aufteilen, dabei N Elemente einordnen $\Longrightarrow$ Laufzeit $\mathcal{O}(N \log N)$
- Aber Worst Case $N - 1 + N - 2 + \ldots + 1 = \mathcal{O}(N^2)$

Implementation


```
def sort(A):
    if len(A) <= 1: return A
    pivot = A[0]
    smaller = [a for a in A[1:] if a < pivot]
    larger  = [a for a in A[1:] if a >= pivot]
    print "s=",smaller, "p=",pivot, "l=",larger
    return sort(smaller) + [pivot] + sort(larger)

A = [1,3,2,5,4]
print sort(A)
```

Ausgabe:

```
s= [] p= 1 l= [3, 2, 5, 4]
s= [2] p= 3 l= [5, 4]
s= [4] p= 5 l= []
[1, 2, 3, 4, 5]
```

Rekursion


```
def factorial(n):
    "calculate the faculty of n, n >= 0."
    # termination for n=0,1
    if n <= 1: return 1
    # n! = n * (n-1)!
    return n * factorial(n-1)
```

- Eine Funktion, die sich selber aufruft, heißt **rekursiv**
- Rekursionen treten in der Mathematik häufig auf
- Viele Algorithmen lassen sich so elegant formulieren
- Die Verarbeitung ist dann aber meist nicht einfach zu verstehen
- Schon ob eine Rekursion endet, ist nicht immer offensichtlich
- Für die Fakultät ist die Schleife vom Anfang der Vorlesung effizienter

Listen aus Listen erzeugen

Institute for Computational Physics

```
>>> print [a**2 for a in [0,1,2,3,4]]
[0, 1, 4, 9, 16]
>>> print sum([a**2 for a in range(5)])
30
>>> print [a for a in range(10) if a % 2 == 1]
[1, 3, 5, 7, 9]
>>> print [(a,b) for a in range(3) for b in range(2)]
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

- Listen können in neue Listen abgebildet werden
- Syntax: `[ausdruck for variable in liste if bedingung]`
  - `ausdruck`: beliebige Formel, die meist von `variable` abhängt
  - `variable,liste`: wie in einer for-Schleife
  - `bedingung`: welche Werte für `variable` zulässig sind
- mehrere **`for`**s können aufeinander folgen (rechteckiges Schema)

Beispiel: Wörter zählen


```
# count words in "gpl.txt"
count = {}
for line in open("gpl.txt"):
    # split into words at blanks
    text = line.split()
    for word in text:
        # normalize word
        word = word.strip(".,:;()\"").lower()
        # account: if already known, increase count
        if word in count: count[word] += 1
        # other create counter
        else:             count[word] = 1

# sort according to count and print 5 most used words
c_sorted = sorted(count, key=lambda word: count[word])
for word in reversed(c_sorted[-5:]):
    print "%s: %5d" % (word, count[word])
```

Wörterbücher (dicts)

```
>>> de_en = { "Milch": "milk", "Mehl": "flour" }
>>> de_en["Eier"]="eggs"
>>> print de_en["Milch"]
milk
>>> if "Mehl" in de_en: print "I can translate \"Mehl\""
I can translate "Mehl"
```

- Komma-getrennte Paare von Schlüsseln (Keys) und Werten in geschweiften Klammern
- Die Werte sind zu den Schlüsseln **assoziiert**
- Vergleiche Wörterbuch: Deutsch → Englisch
- Mit **in** kann nach Schlüsseln gesucht werden
- Gut für unstrukturierte Daten

Wörterbücher (dicts)

Institute for Computational Physics

```
>>> for de in de_en: print de, "=>", de_en[de]
Mehl => flour
Eier => eggs
Milch => milk
>>> de_en["Mehl"] = "wheat flour"
>>> for de, en in de_en.iteritems(): print de, "=>", en
Mehl => wheat flour
Eier => eggs
Milch => milk
```

- Werte sind änderbar (siehe auch Zählprogramm)
- Indizierung über die Keys, nicht Listenindex o.ä.
- **`for`** iteriert auch über die Schlüssel
- Oder mit **`iteritems`** über Schlüssel-Wert-Tupel

Stringmethoden

Institute for Computational Physics

- 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..".**lstrip**(".") → 'Hallo..'

Ein-/Ausgabe: Dateien in Python


```
input  = open("in.txt")
output = open("out.txt", "w")
linenr = 0
for line in input:
    linenr += 1
    output.write("%d: %s" % (linenr, line))
output.close()
```

- Dateien sind mit `open(datei, mode)` erzeugte Objekte
- sind Sequenzen von Zeilen (wie eine Liste von Zeilen)
- Nur beim Schließen (`close`) werden alle Daten geschrieben
- Mögliche Modi (Wert von `mode`):

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