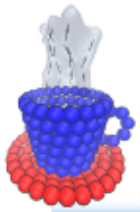


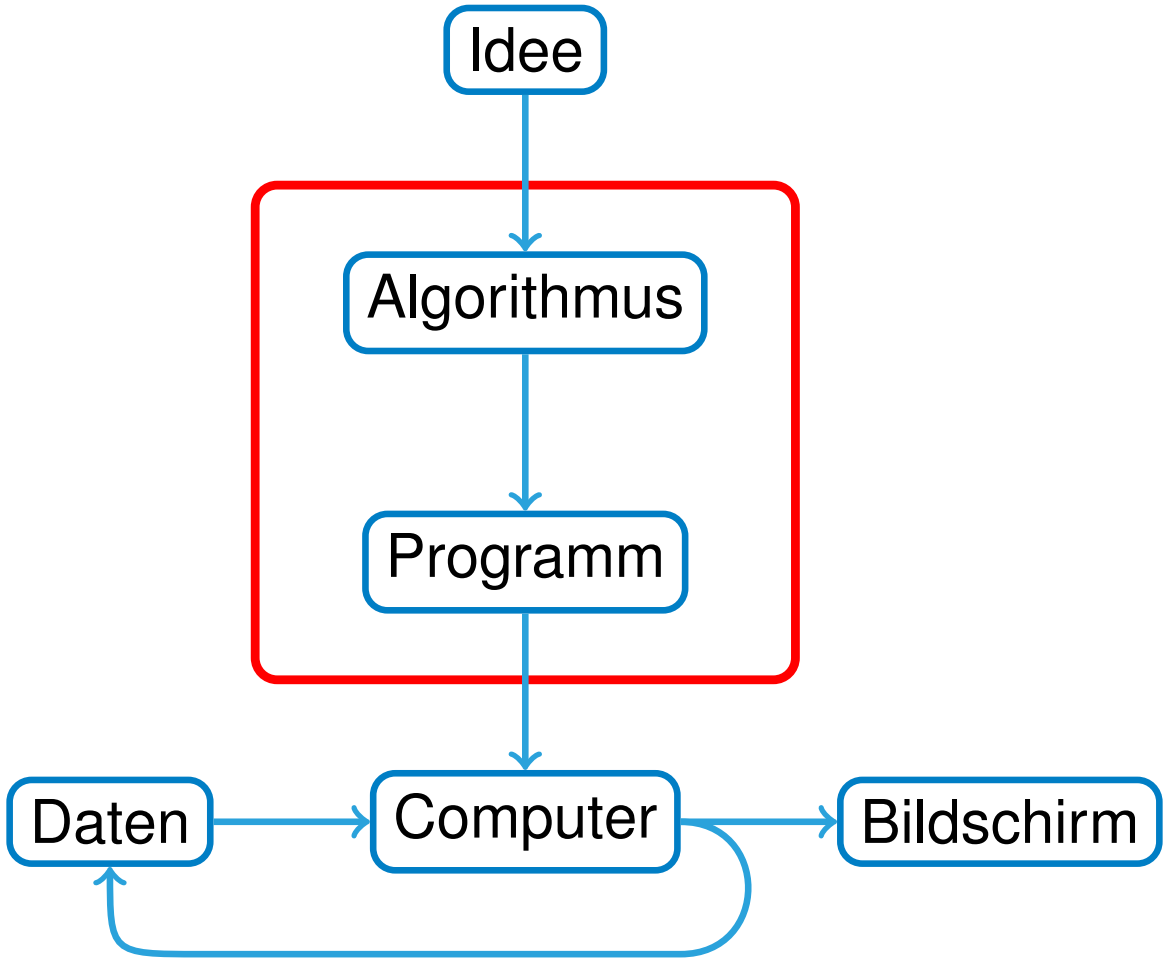
Computergrundlagen
Programmieren lernen — in Python

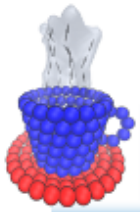
Axel Arnold

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2013/14

Was ist Programmieren?





Algorithmus

Wikipedia:

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

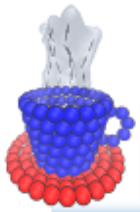
Ein Beispiel-**Problem**:

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

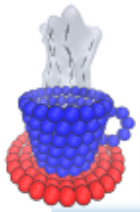
Softwareentwicklung und Programmieren

- Entwickeln der Algorithmen
  - Aufteilen in einfachere Probleme
  - Wiederverwendbarkeit
- Umsetzen in einer passenden Programmiersprache

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



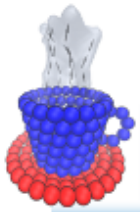
Von der Idee zum Programm

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

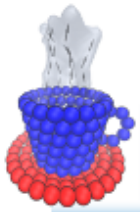


Informationen zu Python

- 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

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 $n! = 1 \cdot 2 \cdots n$ 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

Datentyp: Ganzzahlen

```
>>> 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)

Datentyp: Zeichenketten

```
>>> 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
- Zeichenketten sind keine Zahlen! `"1"` $\neq$ `1`
- `int(string)` konvertiert Zeichenkette in Ganzzahl
- entsprechend `float(string)` für Fließkomma

Sich etwas merken — Variablen

```
>>> 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
```

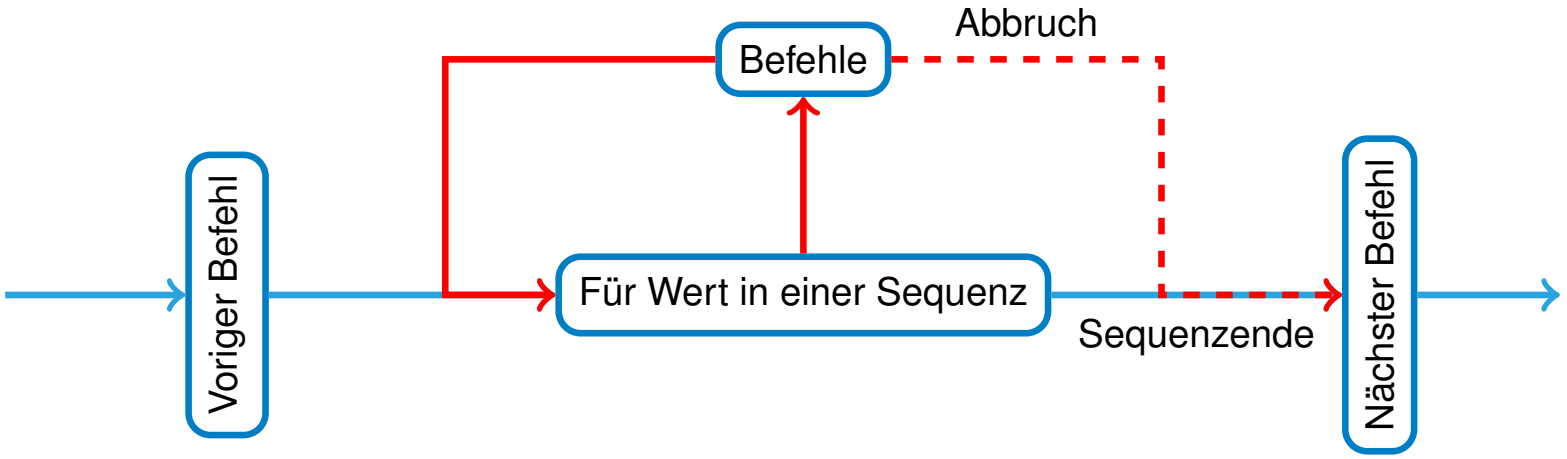
- Variablennamen bestehen aus Buchstaben, Ziffern oder „ _ “ (Unterstrich), am Anfang keine Ziffer
- Groß-/Kleinschreibung ist relevant: Hase $\neq$ hase
- **Richtig:** `i`, `some_value`, `SomeValue`, `v123`, `_hidden`, `_1`
- **Falsch:** `1_value`, `some␣value`, `some-value`

Arithmetische Ausdrücke

+	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



- 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 (Schleifen)

- **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

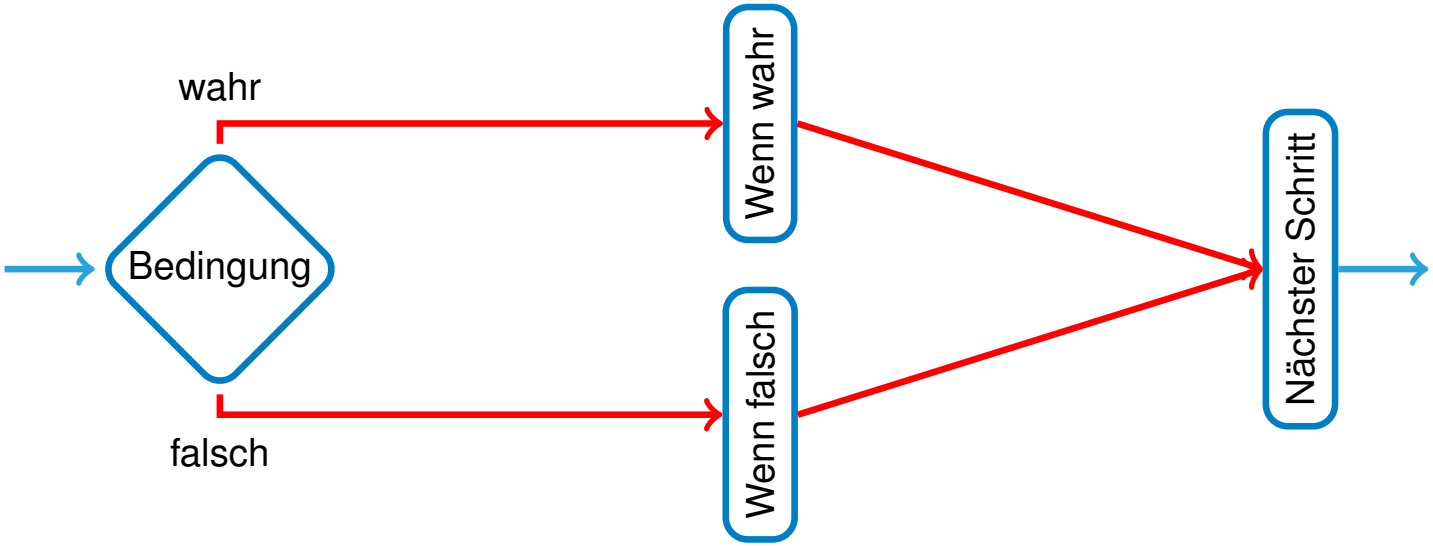
```
# Try to find a pythagorean triple a^2 + b^2 = c^2.
# parameter: rhs number, should be an integer larger than 0
c = 50000

# 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)
            exit()
```

Ausgabe:

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

- Gegebene Daten (c=5) fix ins Programm eingefügt
  $\Longrightarrow$ später lernen wir, Daten einzulesen

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

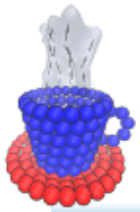
- Präzedenz: logische Verknüpfungen vor Vergleichen

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

Blöcke und Einrückung

```
>>> 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

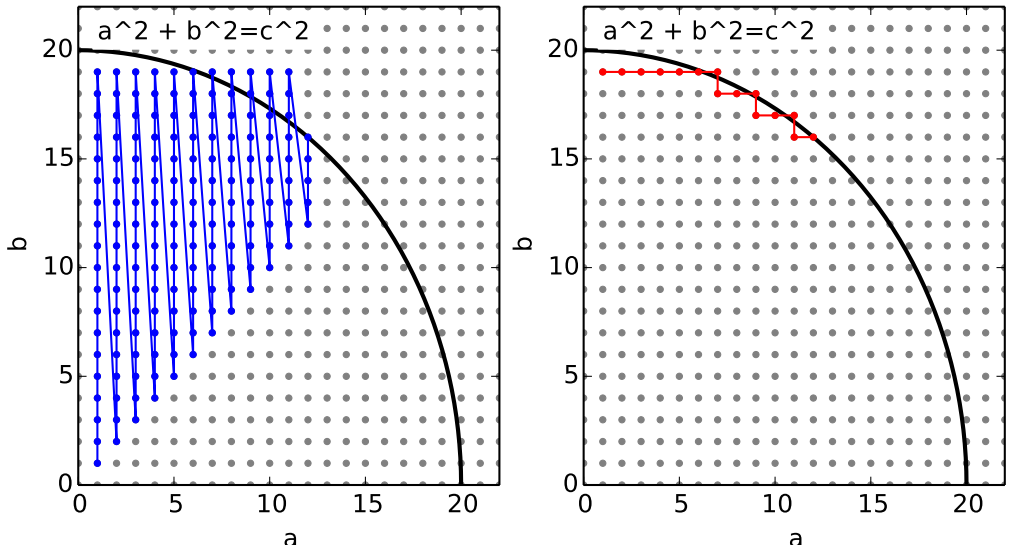
```
>>> print "%d^2 + %d^2 = %05d^2" % (3, 4, 5)
3^2 + 4^2 = 00005^2
>>> print "Strings %s %10s" % ("Hallo", "Welt")
Strings Hallo       Welt
>>> print "Fliesskomma %e |%+8.4f| %g" % (3.14, 3.14, 3.14)
Fliesskomma 3.140000e+00 | +3.1400| 3.14
```

- 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

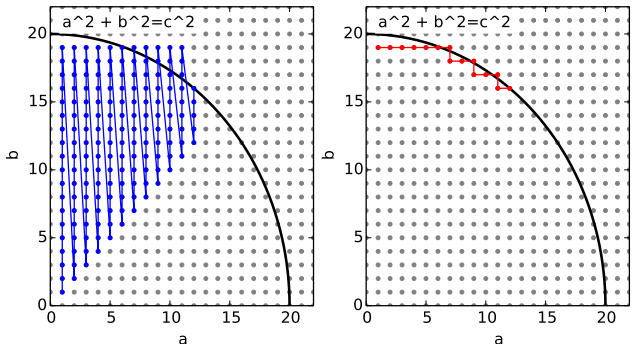
Zahl (alle ohne Lösung)	1236	12343	123456
Zeit	0,2s	18,5s	30m

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



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

Testen auf Effizienz



- **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

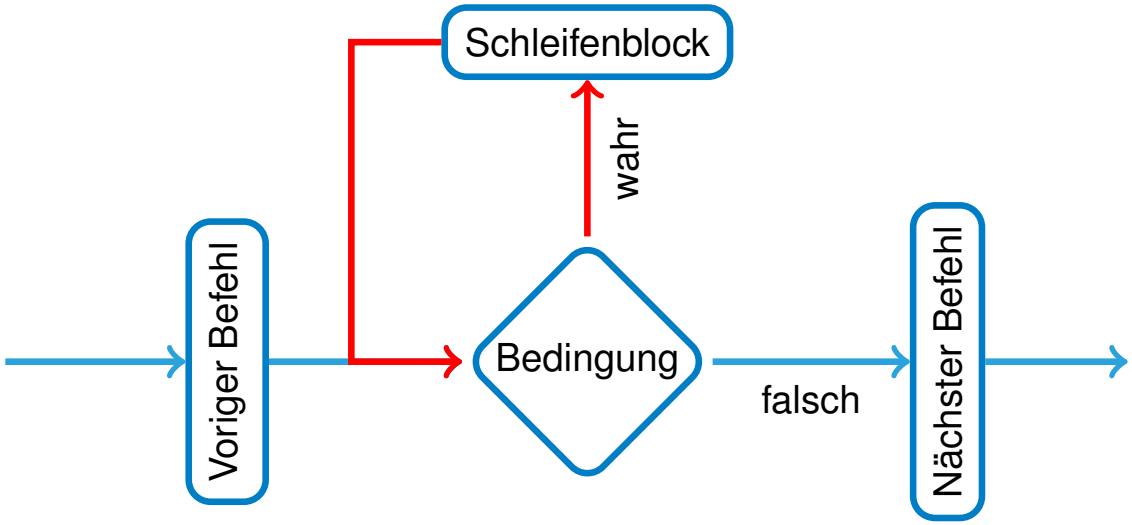
Neue Implementation

```
# 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

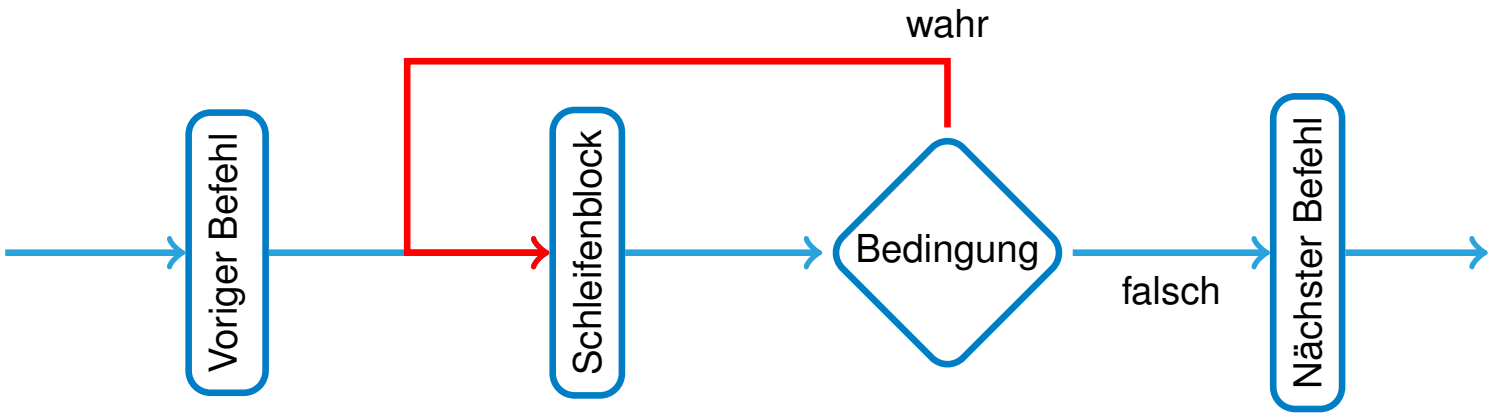
while-Schleifen in Python

```
>>> 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
```

do-while-Schleifen



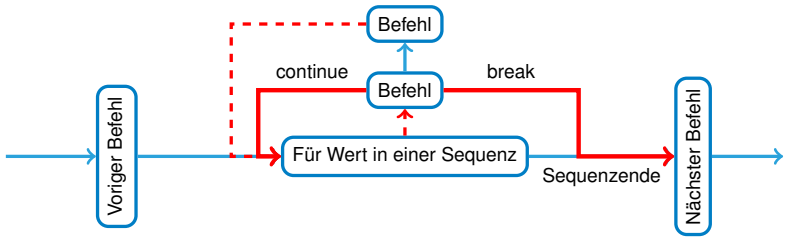
- **`do...while`**-Schleifen führen zunächst den Schleifenblock aus und überprüfen dann die Bedingung
- Nötig, wenn die Bedingung vom Ergebnis des Blocks abhängt
- In Python durch normale **`while`**-Schleife ersetzen:

```
>>> condition = True
>>> while condition:
...     body()
...     condition = check()
```

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
- Aber immer nur die innerste Schleife



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
```

- 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

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 $N-1+N-2+\ldots+1=\mathcal{O}(N^2)$
- Kein zusätzlicher Speicherbedarf

Implementierung

```
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

```
>>> kaufen = [ "Muesli", "Milch", "Obst" ]
>>> kaufen[1] = "Sahne"
>>> print kaufen[-1]
Obst
>>> del kaufen[-1]
>>> print kaufen
['Muesli', 'Sahne']
>>> print "Saft" in kaufen
False
```

- komma-getrennt in eckigen Klammmern
- können Daten *verschiedenen* Typs enthalten
- `liste[i]` bezeichnet das `i`-te Listenelement, negative Indizes starten vom Ende
- **`del`** `elem` löscht das Listenelement `elem`
- `x` **`in`** `liste` überprüft, ob `liste` ein Element mit Wert `x` enthält

Listen in Python

```
>>> kaufen = [ "Muesli", "Milch", "Obst" ]
>>> kaufen.append("Brot")
>>> kaufen.append("Milch")
>>> print kaufen
['Muesli', 'Milch', 'Obst', 'Brot', 'Milch']
>>> kaufen.remove("Milch")
>>> print kaufen
['Muesli', 'Obst', 'Brot', 'Milch']
>>> kaufen.remove("Saft")
ValueError: list.remove(x): x not in list
```

- `liste.append(x)` hängt `x` am Ende der Liste `liste` an
- `liste.remove(x)` entfernt *das erste Element* mit dem Wert `x` aus der Liste
- Fehler, wenn es kein solches gibt (daher mit **`in`** testen)

Listen in Python

```
>>> 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`

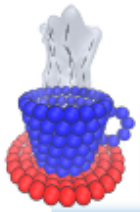
Vorsicht: Flache Kopien!

```
>>> 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

```
>>> 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 + ("Kaese", "Milch")
('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!)

Beispiel: Sieb des Eratosthenes (Listen)

- **Problem**
  Gegeben: Eine ganze Zahl N
  Gesucht: Alle Primzahlen kleiner als N
- **Methode**: Sieb des Eratosthenes
  - Betrachte Liste aller Zahlen zwischen 2 und N
  - Streiche nacheinander alle echten Vielfachen von (Prim-)zahlen
  - Was übrig bleibt, sind Primzahlen $\Longrightarrow$ Sieb

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49

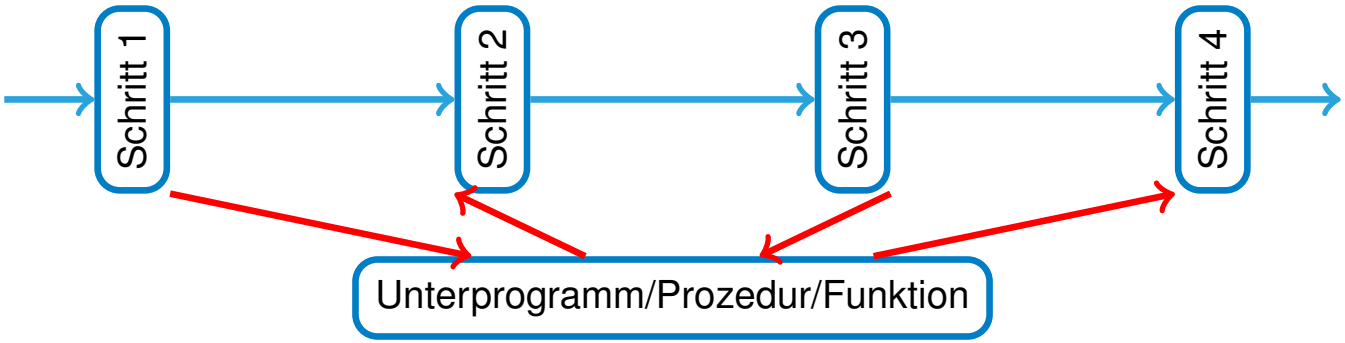
Implementation

```
import sys
# last prime to print out
N = int(sys.argv[1])
primes = range(2,N)
for number in range(2,N/2):
    # not a prime, multiples already deleted
    if not number in primes: continue
    # remove all multiples
    multiple = 2*number
    while multiple < N:
        if multiple in primes: primes.remove(multiple)
        multiple += number
print primes
```

Ausgabe:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

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

```
>>> 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

```
>>> 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

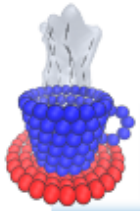
- im Schnitt $\log_2 N$-mal aufteilen, dabei N Elemente einordnen
  $\Longrightarrow$ Laufzeit $\mathcal{O}(N \log N)$
- Aber Worst Case $N - 1 + N - 2 + \ldots + 1 = \mathcal{O}(N^2)$

Implementation

```
def sort(A):
    print "sorting", 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 "smaller=",smaller,"pivot=",pivot,"larger=",larger
    return sort(smaller) + [pivot] + sort(larger)
A = [3,1,2,5,4,2,3,4,2]
print sort(A)
```

Ausgabe:

```
sorting [3, 1, 2, 5, 4, 2, 3, 4, 2]
smaller= [1, 2, 2, 2] pivot= 3 larger= [5, 4, 3, 4]
sorting [1, 2, 2, 2]
...
```



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)
n = 5
print n, "! =", factorial(n)
```

- 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

```
>>> 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 (dicts)

```
# 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)

```
>>> 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

- 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
while True:
    line = input.readline()
    if not line: break
    linenr += 1
    output.write("%d: %s\n" % (linenr, line))
output.close()
```

- Dateien sind mit `open(datei, mode)` erzeugte Objekte
- 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

Ein-/Ausgabe: Dateien in Python

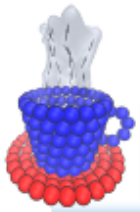
```
input  = open("in.txt")
output = open("out.txt", "w")
linenr = 0
while True:
    line = input.readline()
    if not line: break
    linenr += 1
    output.write("%d: %s\n" % (linenr, line))
output.close()
```

- `datei.read()`: Lesen der *gesamten* Datei als Zeichenke
- `datei.readline()`: Lesen einer Zeile als Zeichenkette
- Je nach Bedarf mittels `split`, `int` oder `float` verarbeiten

Ein-/Ausgabe: Dateien in Python

```
input  = open("in.txt")
output = open("out.txt", "w")
linenr = 0
while True:
    line = input.readline()
    if not line: break
    linenr += 1
    output.write("%d: %s\n" % (linenr, line))
output.close()
```

- `datei.write(data)`: *Zeichenkette* data zur Datei hinzufügen
- Anders als **`print`** *kein* automatisches Zeilenende
- Bei Bedarf Zeilenumbruch mit „\n“
- Daten, die keine Zeichenketten sind, mittels %-Operator oder `str` umwandeln



Dateien als Sequenzen

```
input  = open("in.txt")
output = open("out.txt", "w")
linenr = 0
for line in input:
    linenr += 1
    output.write(str(linenr) + ": " + line + "\n")
output.close()
```

- Alternative Implementierung zum vorigen Beispiel
- Dateien verhalten sich in **`for`** wie Listen von Zeilen
- Einfache zeilenweise Verarbeitung
- Aber kein Elementzugriff usw.!
- `write`: alternative, umständlichere Ausgabe mittels `str`-Umwandlung

Standarddateien

- wie in der bash gibt es auch Dateien für Standard-Eingabe, -Ausgabe und Fehler-Ausgabe
- Die Dateivariablen sind

`sys.stdin`	Eingabe (etwa Tastatur)
`sys.stdout`	Standard-Ausgabe
`sys.stderr`	Fehler-Ausgabe

```
import sys
line = sys.stdin.readline()
sys.stderr.write("don't know what to do with %s\n" % line)
for i in range(10):
    sys.stdout.write("%d, " % i)
sys.stdout.write("\n)
```

Ausgabe:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Ein-/Ausgabe mittels `raw_input`

```
>>> passwd = raw_input("enter password to continue: ")
enter a password to continue: secret
>>> control = input("please repeat the password: ")
please repeat the password: passwd
>>> if passwd == control: print "both are the same!"
both are the same!
```

- Tastatureingaben können einfach über `raw_input` in eine Zeichenkette gelesen werden
- `input` wertet diese hingegen als Python-Ausdruck aus
- Dies ist eine potentielle Fehlerquelle:

```
>>> passwd = input("enter a password to continue: ")
enter a password to continue: secret
NameError: name 'secret' is not defined
```

- Eingaben über die Kommandozeile sind meist praktischer — oder wäre Dir ein mv lieber, dass nach den Dateien fragt?

Umlaute — Encoding-Cookie

```
#! /usr/bin/python
# encoding: utf-8
# Zufällige Konstante α
alpha = 0.5
# α² ausgeben
print "Mir dünkt, dass α² = %g" % (alpha**2)
```

Ausgabe:

```
Mir dünkt, dass α² = 0.25
```

- Umlaute funktionieren bei Angabe der Codierung
- Muss in den ersten beiden Zeilen stehen
- Variablennamen trotzdem in ASCII!

Programmierpraxis: Funktionen statt Copy&Paste

- Fehler in kopiertem Code sind schwer *an allen Stellen* zu beheben
- Prozedurnamen beschreiben, was passiert

Daher besser so:

```
def countlines(filename):
  c = 0
  for l in open(filename):
    c += 1
  return c
print countlines("in1.txt")
print countlines("in2.txt")
```

Und nicht so:

```
# was tut das bloss?
c = 0
for l in open("in1.txt"):
  c += 1
print c
c = 0
for l in open("in2.txt"):
  c += 1
print c
```

Programmierpraxis: Variablen statt Konstanten

- Willkürliche Konstanten erschweren Austauschbarkeit und Wiederbenutzung
- Variablen mit *sprechenden* Namen sind einfacher zu verstehen
- Bei Funktionen: zu Parametern machen

Daher besser so:

```
start = 0.01
tolerance = 0.0001
min = minimum(pot, start,
              tolerance)
```

Und nicht so:

```
# welche Genauigkeit hat das?
min = minimum(pot, 0.01,
              0.0001)
```

Und auch nicht so:

```
a = 0.01
b = 0.0001
x = minimum(pot, a, b)
```

Module

```
>>> import sys
>>> print "program name is \"%s\"" % sys.argv[0]
program name is ""
>>> from random import random
>>> print random()
0.296915031568
```

- Bis jetzt haben wir einen Teil der Basisfunktionalität von Python gesehen.
- Weitere Funktionen sind in **Module** ausgelagert
- Manche sind nicht Teil von Python und müssen erst nachinstalliert werden
- Die Benutzung eines installierten Moduls muss per **`import`** angekündigt werden („Modul laden“)
- Hilfe: `help(modul)`, alle Funktionen: `dir(modul)`

Das sys-Modul

- Schon vorher für Eingaben benutzt
- Stellt Informationen über Python und das laufende Programm selber zur Verfügung
- `sys.argv`: Kommandozeilenparameter, `sys.argv[0]` ist der Programmname
- `sys.stdin`, `sys.stdout`, `sys.stderr`: Standard-Ein-/Ausgabedateien

```
import sys
sys.stdout.write("running %s\n" % sys.argv[0])
line = sys.stdin.readline()
sys.stderr.write("some error message\n")
```

optparse-Modul: Parameter einlesen

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

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

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

argparse-Modul: Parameter in Python 2.7

```
from argparse import ArgumentParser
parser = ArgumentParser()
parser.add_argument("-f", "--file", dest="filename",
                    help="write to FILE", metavar="FILE")
parser.add_argument("positional", help="fixed argument")
parser.add_argument("number", type=int, help="integer arg")
args = parser.parse_args()
```

- Ab Python 2.7: verbessertes Modul argparse
- `add_argument` unterstützt auch positionale Argumente, die ohne Optionsmarker auf der Kommandozeile stehen
- Fehler bei falschem Typ oder fehlenden Parametern
- Bei Aufruf `python parse.py -f test a 1` ist:
  - `args.filename = 'test'`
  - `args.positional = 'a'`
  - `args.number = 1`

math- und random-Modul

```
import math
import random
def boxmuller():
    """
calculate Gaussian random numbers using the
Box-Muller transform
    """
    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 *pseudo*zufällige 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*

os-Modul: Betriebssystemfunktionen

```
import os
import os.path
dir = os.path.dirname(file)
name = os.path.basename(file)
altdir = os.path.join(dir, "alt")
if not os.path.isdir(altdir):
    os.mkdir(altdir)
newpath = os.path.join(altdir, name)
if not os.path.exists(newpath):
    os.rename(file, newpath)
```

- betriebssystemunabhängige Pfadtools im Untermodul `os.path`: z.B. `dirname`, `basename`, `join`, `exists`, `isdir`
- `os.system`: Programme wie von der Shell aufrufen
- `os.rename`/`os.remove`: Dateien umbenennen / löschen
- `os.mkdir`/`os.rmdir`: erzeugen / entfernen von Verzeichnissen

GUI: das Tkinter-Modul

```
import Tkinter

# main window and connection to Tk
root = Tkinter.Tk()
root.title("test program")
# a simple button, ending the program
end = Tkinter.Button(root, text = "Quit",
                     command = root.quit)
end.pack({"side": "bottom"})

root.mainloop()
```

- bietet Knöpfe, Textfenster, Menüs, einfache Graphik usw.
- mit `Tk.mainloop` geht die Kontrolle an das Tk-Toolkit
- danach eventgesteuertes Programm (Eingaben, Timer)
- lehnt sich eng an Tcl/Tk an (`http://www.tcl.tk`)