Python für Physiker

Günter Duckeck

20.03. – 24.03.2017, 10:00-12:00, 13:30-16:00 Uhr
CIP-Pool, Schellingstr. 4

Inhalt:

Einführung

Python Grundlagen Variables, Control, Funktionen, Lists, Tuples, Dictionaries, I/O

OO Programmieren – Klassen und Objekte

Python Standardlibs und Python für Wissenschaftler Exceptions, Introspection, Numpy, Matplotlib ...

Folien & Übungen als pdf und im WWW

http://www.etp.physik.uni-muenchen.de/kurs/Computing/python

Schlüsselqualifikation für Bachelor/Master

- Einwöchiger Blockkurs jeweils equivalent zu SQ 1+2, d.h.
  1. Kurs = Python für Physiker = SQ1+2 = 3 ECTS Punkte

- Erfolgreiches Bestehen der Leistungskontrolle (= schriftlicher Kurztest am Ende) ist Kriterium für Punktevergabe
- Zusammen mit den Klausuraufgaben verteilen wir die Scheinformulare, die Sie ausfüllen und mit der Klausur abgeben.
- Nach Korrektur veröffentlichen wir die Liste (Matrikelnummer, bestanden/nicht bestanden) auf dieser Kursseite und leiten die Scheine ans Prüfungsamt weiter. Sie können Ihren Schein dann im Prüfungsamt abholen.

Allgemeines

Python ist moderne Skript–Sprache

- klare Syntax
- einfache Struktur
- Features für objektorientiertes Programmieren
- mächtige Funktionen–Pakete (=Module) mitgeliefert
- Häufig Basis-Sprache für moderne Anwendungen wie Web-Programmierung, Machine-Learning, etc.

$\Rightarrow$ flache Lernkurve

- Einstieg wesentlich schneller als bei Fortran, C/C++, und sogar JAVA
- **Aber:** riesiger Funktionsumfang, Vielzahl von Modulen, Erweiterungen (Databases, XML, Networking, ...), Entwicklungsumgebungen, etc.

Ziel des Kurses Python für Physiker:

- Python Syntax
- Grundlegende Funktionalität: Variablen, Funktionen, I/O, Standard API

und damit

- Fähigkeit zum Erstellen kleiner Programme
- Anregungen zur Verwendung weiterführender Module

Aber Grundkurs:

- Keine umfassende Präsentation des kompletten Python Sprachumfangs
- Keine Schulung objektorientiertes Design/Programmieren
- ...

Ablauf

- Mo, Di, Mi, Do, Fr; jeweils 10:00 – 12:00 und 13:30 – 16:00
- Schwerpunkt praktische Übungen: ca. 1/3 Theorie, ca. 2/3 Übungen am Rechner im CIP.
- 10 Kursblöcke a 2/2.5 h = 18 h
  1. **Python Grundlagen** 3-4 Blöcke
  2. **Klassen und Objekte** ca. 3 Blöcke
  3. **Python Standard-Libs und Anwendungen** ca. 3 Blöcke

Flexible Gestaltung, je nach Kenntnisse und Neigungen der Teilnehmer, insbesondere letzter Teil *Numeric and Scientific Python*.

Literatur und Links

Zu Python gibt es ein großes Angebot an Lehrbüchern, sowie Online Kurse und Tutorials, FAQs und Diskussionsforen im Web. Hier eine kleine Auswahl:

Learning Python Mark Lutz, 5. Auflage, O'Reilly Series, 2013. Gute, ausführliche Einführung in Python, aber über 1600 Seiten!

Python Essential Reference David Beazley, 4. Auflage 2009, übersichtliche, komplette Referenz auf aktuellem Stand der Python Versionen.

A Byte of Python Grundlegende Einführung in Python, Online Buch, wendet sich ausdrücklich an Newcomer im Programmieren.

How to Think Like a Computer Scientist (Python) erhältlich als Buch und online. Gute konzeptionelle Einführung ins Programmieren.

Äquivalente Versionen für JAVA und C++.

A student's guide to python for physical modeling Jesse M. Kinder, Philip Nelson, Princeton University Press, 2015. Aktuelles Buch, nutzt SciPy, Science/Physik Beispiele zu Datenanalyse, Visualisierung und Modellierung.

Das Python-Praxisbuch - Der große Profi-Leitfaden für Programmierer

Farid Hajji, Addison-Wesley, September 2008. Sehr ausführliche Beschreibungen und Beispiele zu fortgeschrittenen Themen in Python.

Online Referenzen

http://www.python.org Offizielle Python Homepage. Unerschöpfliche Quelle für Downloads, Dokumentation, Tutorials, Links.

Python 2.7 Quick Reference Kompakte Online-Referenz zu Python

Python 2.7.6 Documentation Dokumentation zu Python 2.7.6

Python 2.7.6 Library Reference Dokumentation der Python Standard Library

Python Cookbook Umfangreiche Sammlung von Rezepten zur Problemlösung in Python, allerdings eher auf fortgeschrittenem Niveau ...

Hidden Python Features Teils nützliche, teils verstörende Tipps ...

Software Carpentry Handwerkszeug zum Programmieren für Naturwissenschaftler. Nicht nur Python sondern Rundumschlag von Shells, Programmiertechniken, XML, Spreadsheets, Databases, Web-Programming, u.v.m. Python als Glue-language, das die verschiedenen Bereiche verknüpft.

Python for Science Schöne Online Referenz mit vielen Physik–Beispielen

Computational Statistics in Python , Gut gemachter Online Kurs zu Mathematik und Statistik mit Python

SciPy (NumPy, MatPlotLib, ...)

MatPlotLib Tutorial

Physik und Computing

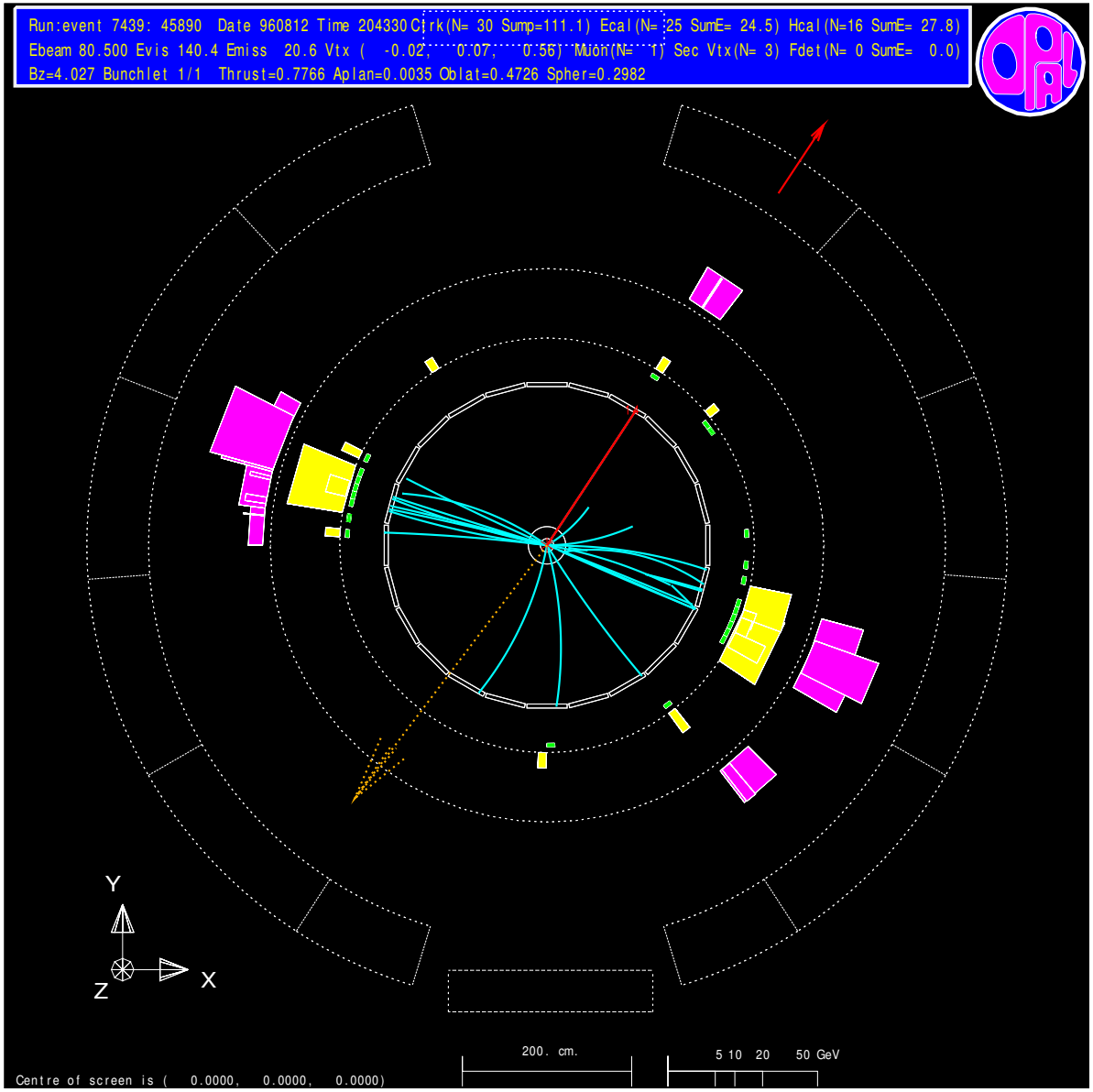
In der Physik sind Computer fast überall von zentraler Bedeutung: Design von Experimenten, Datenauslese, Auswertung und Statistik, Simulation, Theorie, Kommunikation, Recherche, ...

Eine Sonderrolle spielt die Teilchenphysik:

Experimente immer am technologischen Limit bei Datenvolumen und –rate, Prozessorgeschwindigkeit

Teilchenphysik nicht nur Nutzer sondern hat viele Entwicklungen vorangebracht:

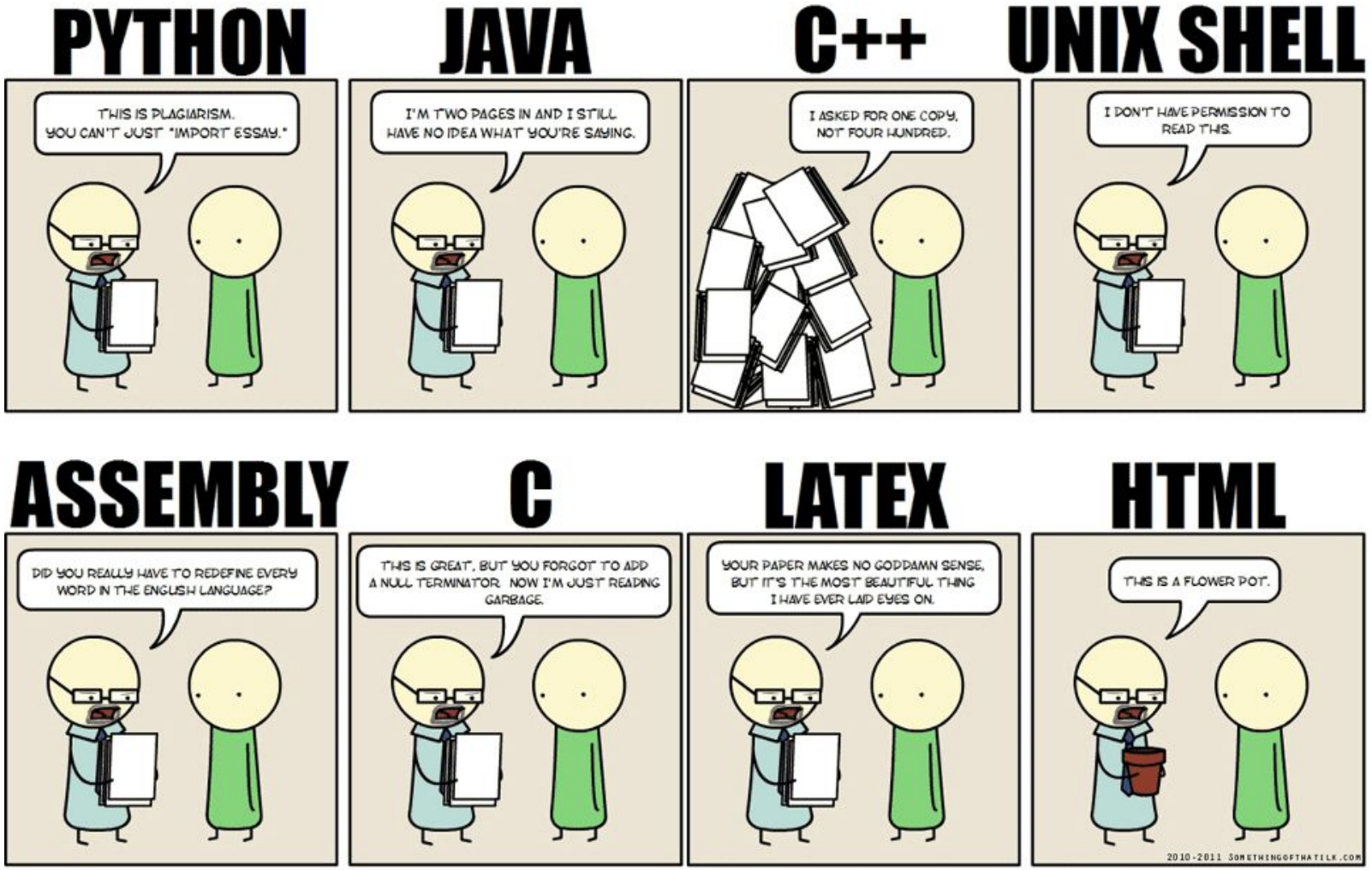
- verteiltes Rechnen
- Realtime computing
- WWW am CERN erfunden
- GRID Projekt


Run:event 7439: 45890 Date 960812 Time 204330 Ctrk(N= 30 Sump=111.1) Ecal(N= 25 SumE= 24.5) Hcal(N=16 SumE= 27.8)
Ebeam 80.500 Evis 140.4 Emiss 20.6 Vtx (-0.02, 0.07, 0.56) Muon(N= 1) Sec Vtx(N= 3) Fdet(N= 0 SumE= 0.0)
Bz=4.027 Bunchlet 1/1 Thrust=0.7766 Aplan=0.0035 Oblat=0.4726 Spher=0.2982
Y
Z
X
200. cm.
5 10 20 50 GeV
Centre of screen is (0.0000, 0.0000, 0.0000)

Computing Kenntnisse für Physiker – eine Wunschliste

- **Alltag:** Textverarbeitung (Office, Latex), Präsentationen (PPT, TeX), WWW Nutzung, E-Mail
  $\Rightarrow$ *Selbststudium*
- **Datenanalyse/Statistik:** Tabellenkalkulation (Excel), SAS, ROOT $\Rightarrow$ *(Selbst/Kurs)*
- **Mathematik/symbolische Algebra:** Maple, Mathematica, ... $\Rightarrow$ *(Selbst/Kurs)*
- **Höhere Programmiersprache:** Fortran, C/C++, Java, ...$\Rightarrow$ *Kurse*
- **Numerik**: Algorithmen (Fortran/C++) $\Rightarrow$ *Vorlesung (Schein)*
- **Advanced concepts:** OO–Programmieren und –Design, GUI, Threads, Container $\Rightarrow$ *Kurse*
- **Hardware Programmierung** $\Rightarrow$ *(Kurs)*
- **High Level IT Anwendungen:** Databases, XML, Skript-Sprachen, Web-Programmierung, Grid, ...
  $\Rightarrow$ *Python Kurs*

Programmiersprachen und Programmieren



Natürliche Sprachen (Deutsch, Französisch, ...) entwickeln sich langsam über viele Jahrhunderte, es gibt keine klaren von vorneherein festgelegten Regeln. Im Gegensatz dazu sind **formale Sprachen** gezielt für bestimmte Aufgaben entworfen worden, z.B. in Musik, Mathematik, Chemie, oder eben die Programmiersprachen um Abläufe in der *Datenverarbeitung* auszudrücken.

Gewisse grundsätzliche Gemeinsamkeiten: *Struktur, Syntax, Begriffe*, aber formale Sprachen viel **präziser, strikter, dichter**. Keine oder kaum Ambiguitäten, wenig Redundanz, genau umrissene Bedeutung, keine Metaphern o.ä.

Konsequenz für Lesen und Verstehen:

- Informationsdichte in Programmtext wesentlich größer
- kein Lesen von Anfang bis Ende, sondern Orientierung an Struktur
- jedes Detail ist wichtig

Was ist Programmieren ?

Ein Programm ist eine Reihen von Anweisungen, die bestimmen wie eine Datenverarbeitung abläuft. Dazu sind nur wenige grundlegende Funktionen nötig, die im Prinzip allen Programmiersprachen gemeinsam sind:

- **Input:** Daten von Tastatur, File, Netzwerk, Sensor, ...
- **Output:** Daten auf Bildschirm, Datei, Drucker, Steuergerät, ...
- **Operation:** mathematischer Ausdruck, Zuweisung, ...
- **Testen und Verzweigen:** Überprüfen von Bedingungen, unterschiedliche Abläufe
- **Schleifen:** Wiederholte Ausführung bestimmter Abschnitte

Python Features

- Python ist objekt-orientierte, plattform-unabhängige Programmiersprache. Entwickelt Ende der 90er Jahre.
- Python ist Interpreter/Skript Sprache, wie Shell-scripts, Perl, Basic, im Gegensatz zu Compiler–sprachen C/C++, Fortran, Cobol, ..., (JAVA irgendwo dazwischen)
- Traditionell werden Interpreter/Skript–Sprachen v.a. für Systemadministration oder Hilfs–macros (z.B. MS Excel/VB) verwendet.
- Python zunehmend als eigenständige Sprache für Vielzahl von Anwendungsbereichen.
- Insbesondere als **glue language** um unterschiedliche Bereiche zu verknüpfen, z.B. *Datennahme via Sensor in C/C++, Zugriff via Web-interfaces, Abspeichern in Datenbank* $\Rightarrow$ Python ideal zur Verknüpfung

Python vs C++

Pros:

- Wohldefinierter, überschaubarer Sprachumfang
- Vielzahl von Hilfspaketen zu I/O, Networking, Graphik, Databases, ... in Standarddistribution integriert.
- Viele Features in Sprache integriert, die alltägliche Programmieraufgaben wesentlich erleichtern.
- Plattform unabhängig
- Flache Lernkurve, hohe Programmiereffizienz

Cons:

- Performance–Nachteile
- Hardwarenahe Programmierung erschwert

Python und C++ nicht wirklich Konkurrenz sondern eher komplementär. In Python viele “einfache” Aufgaben sehr leicht zu lösen. Für zeit-/speicherkritische Probleme C/C++ um Längen besser. *Allerdings: Bei heutiger Computer–performance nur selten der Fall. Dann am besten heterogene Lösung*

1 Python Grundlagen

- Die ersten Schritte – Interaktiv, Skripte ausführen
- Python Operationen
- Grundlegende Datentypen und Definition von Variablen
- Strings, Lists und Tuples
- Control-statements
- Funktionen
- Basic I/O
- Ergänzungen
- Aufgaben

1.1 Die ersten Schritte

Das Standard–Minimal–Programm in Python

```
print "Hello, world"
```

ist absolut minimalistisch ...

- Einziges Statement ist Aufruf von `print` mit String der ausgegeben werden soll

Zum Vergleich, dasselbe in JAVA

```
public class HelloWorld {                                    1
    // A program to display the message                      2
    // "Hello World!" on standard output                     3
                                                             4
    public static void main(String[] args) {                5
    System.out.println("Hello World!");                      6
    }                                                        7
}   // end of class HelloWorld                               8
```

und C++

```
// Hello-world C++ Version                                    1
#include <iostream> // pre-prozessor command                  2
using namespace std; // declare namespace                     3
int main()    // function definition                         4
{                                                             5
  cout << "Hello world" << endl;                             6
  return(0);                                                  7
}                                                             8
```

Grundlegende Syntax ist Python spezifisch:

Ähnlichkeiten am ehesten mit anderen Skript-Sprachen (bash, perl), weniger mit C/C++/JAVA.

- Kommentare beginnen mit #
- Ausdruck wird durch Zeilenende abgeschlossen (**nicht** Semikolon ';')
- **Einrücken** spielt zentrale Rolle: definiert Kontrollstrukturen, Funktionen, Klassen, ...

Python interaktiv

- Start mit `python` im Shell Fenster startet eine interaktive Python Sitzung.
- Beliebige Python Kommandos können eingegeben werden und werden nach return–Eingabe sofort vom Python Interpreter ausgeführt.
- Beenden mit `Ctrl-D` bzw. `Strg-D`
- Sehr nützlich für Tests, Taschenrechner, Fehlersuche, zum Kennenlernen, etc.

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print "Hallo, wie gehts ?"
Hallo, wie gehts ?
>>> 2**32
4294967296
>>> 3**0.5
1.7320508075688772
>>> for i in range(1,11):
...   print i, i**2, i**3
...
...
3 9 27
...
```

Meistens geht's aber darum Programme zu schreiben, die man immer wieder verwenden möchte, dann

Erstellen des Quellprogramms im Editor, z.B.:

```
kate hello.py
```

oder:

```
gedit hello.py
```

```
# hello.py                                                    1
print 'Hallo Welt !'                                          2
```

Python Konvention: Filename des Quellprogramms hat Endung `.py`

Ausführen:

```
python hello.py
```

Der Python–Interpreter liest die Datei und führt die Anweisungen der Reihe nach aus, analog zur interaktiven Eingabe.

Python–Interpreter führt systemabhängige Übersetzung der Anweisungen aus.

1.2 Ein kleines Programm

```
# print Fahrenheit->Celsius Tabelle                                          1
lower = 0                                                                    2
upper = 300                                                                  3
step = 20                                                                    4
fahr = lower                                                                 5
while ( fahr < upper ):                                                      6
    celsius = 5./9 * (fahr - 32) # conversion                                7
#    print 'Fahrenheit %6.1f = Celsius %6.3f' % (fahr , celsius) # formatierte Ausgabe   8
    print "Fahrenheit ", fahr," = Celsius ", celsius                         9
    fahr += step    # increment                                              10
# end of loop                                                                11
```

Hier Vorgriff auf *Variablen, Kontrollstrukturen, arithmetische Ausdrücke, Ausgabe, ...*

- Variablen können in Python jederzeit verwendet werden, die Deklaration erfolgt einfach mittels Zuweisung, z.B. `x = 1`. Es ist keine Typ–Angabe nötig (⇒ später).
- Control-Statement Schleife/Loop:

```
while ( Bedingung ):
    expressions
```

  - Bedingung `fahr < upper` wird getestet
  - wenn erfüllt werden expression ausgeführt
    dann zurück zu (6).
  - Andernfalls zum Ende der Schleife: (11)
- Arithmetischer Ausdruck `(5./9) * (fahr - 32)` wird ausgewertet und dann der Variablen `celsius` zugewiesen.
- Inkrement `fahr += step` : Erhöhe Wert der Variablen `fahr` um `step`.
- Function call für output, Formatierungsangaben für Gleitkommazahlen: `%6.3f` bewirkt Ausgabe von Gleitkommazahl mit 3 Nachkommastellen und 6 Stellen insgesamt (⇒ später).
- In Python wird ein Statement durch das Zeilenende abgeschlossen. Keine Beschränkung der Zeilenlänge !

Definition der Syntax Elemente

- Statement – Ausführbare Anweisung: I.d.R. ist jede Python Zeile eine ausführbare Anweisung, das kann sein:
  - Assignment – Zuweisung: `a = expression` Objekt wird erzeugt und einer Variablen zugewiesen
  - Expression – Anweisung: einfache Konstante, Variable, arithmetische oder logische Operation (`5./9. * ( fahr - 32 )` oder `fahr < upper`),
    Funktionsaufruf: `print 'Fahrenheit ', fahr`
  - Kontroll–struktur: `while, for, if`
- Ausnahmen sind:
  - Funktions– oder Klassendeklaration
  - Kommentare

1.3 Python Operationen

Arithmetische Operationen und Zuweisungen

JAVA/C++	Python	Zweck
$-x$	$-x$	Vorzeichen
$x + y$	$x + y$	Addition
$x - y$	$x - y$	Subtraktion
$x * y$	$x * y$	Multiplikation
x/y	x/y	Division
$x\%y$	$x\%y$	Modulo (Rest nach Div.)
$pow(x, y)$	$x * *y$	Potenzieren
$x = y$	$x = y$	Zuweisung
$x += y$	$x += y$	Zuweisung mit Änderung ($x = x + y$)
$(-=, *=, /=, \%=)$	$(-=, *=, /=, \%=, **=)$	

Kein `x++, ++x, x--, --x` in Python !

Logische Operationen und Vergleiche:

JAVA/C++	Python	Zweck
false / (oder 0)	False	falsch
true / (oder $\neq 0$)	True	wahr
!x	not x	Negation
$x\&\&y$	x and y	AND Verknüpfung
$x\|\|y$	x or y	OR Verknüpfung
$x < y$	$x < y$	kleiner als
$x <= y$	$x <= y$	kleiner als oder gleich
$x > y$	$x > y$	größer als
$x >= y$	$x >= y$	größer als oder gleich
$x == y$	$x == y$	gleich
$x! = y$	$x! = y$	ungleich

Vorrangregeln für Operatoren: `f = a*x**2 + b*x + c` funktioniert wie erwartet nach Mathematik (Punkt vor Strich ...). Weiteres im Prinzip eindeutig festgelegt, dennoch besser mit Klammern unmissverständlich klarmachen: `x = y > z` ? `(x=y) > z` oder `x = (y>z)`

Bit Operationen:

Python, JAVA, C++	Zweck
$\tilde{}x$	Complement
$x\&y$	bitwise AND
$x\|y$	bitwise OR
$x\hat{}y$	bitwise XOR
$x << y$	left shift
$x >> y$	right shift mit sign

```
print(2 | 1)    # = 3
print(2 & 1)    # = 0
print(2 & 3)    # = 2
print(2 <<1)    # = 4
```

Mehr Beispiele später in den Übungen

1.4 Variablen in Python

Wie in anderen Sprachen auch gibt es in Python verschiedene Datentypen für *ganze Zahlen, Fließkommazahlen, Text–Strings, Arrays/Listen, u.a.*.

- In Python werden Variablen **dynamisch** angelegt mittels Zuweisung.
- Die Typ–Zuordnung passiert ebenfalls **dynamisch**, je nachdem welcher Datentyp benötigt wird.
- Im Gegensatz zu JAVA/C++ kann die Zuordung jederzeit geändert werden.

```
x = 1      # int                                          1
y = 0.73   # float                                        2
z = 'Hi there' # string                                   3
#                                                         4
a = x + y  # float                                        5
#                                                         6
x = z      # x jetzt string                               7
#                                                         8
a = x + y  # Fehler: string + float geht nicht            9
```

Python ist eine *dynamically-typed language*, die Variablenzuordnung erfolgt zur Laufzeit, aus dem Context.

Gegensatz *statically-typed language* wie C, JAVA: Typzuordnung fix, explizite Typ–Angabe erforderlich.

1.5 Primitive Datentypen

Ganze Zahlen (int)

Ganze Zahlen werden in Python sehr speziell behandelt. Für ”kleine” Zahlen ($\approx \pm 2 \cdot 10^9$) ist es wie in JAVA/C++: 32-bit integer.

Zusätzlich gibt es die *long integers* deren Länge **unbegrenzt** ist *(bis auf System–Grenzen)*. Python erledigt automatisch den Übergang von **32-bit int** zu **long int** falls erforderlich.

```
>>> 25**2
625          # int
>>> 25**6
244140625    # int
>>> 25**7
6103515625L  # long
>>> 25**200
3872591914849318272818030633286351847570219192048790865487762941344416348097685964862682234277014596908057542507554467539370836398992350315522318050653350492002436065270530802738432038373174754090809367646454942400181270162578968846816261130394654088604511343874037265777587890625L  # really long ...
```

Gleitkommazahlen (float)

```
>>> 2.6+7.8
10.4
>>> 2.5**2
6.25
>>> 2.5**(1./10)
1.0959582263852172
```

Entsprechen den *double* in JAVA/C++, d.h. 64 bit werden zur Darstellung verwendet, unterteilt in Mantisse und Exponent:

Zahlen-Bereich: $\approx [\pm 4.9 \cdot 10^{-324}, \pm 1.8 \cdot 10^{+308}]$, ca. 16 Dezimalstellen

Wichtig: Operationen mit Gleitkommazahlen sind in der Regel nicht exakt, da Rundungsfehler auftreten. Selbst einfache Dezimal–Zahlen *(z.B. 0.1, 0.2)* sind nicht exakt als als *double* darstellbar, weil eine unendliche Folge von 2-er Potenzen nötig wäre.

In Python teilweise implizite Typumwandlung (`int` → `float`)
aber Vorsicht, im Zweifelsfall besser explizit (**casts**) `float( .. )`

```
>>> f=2/3
>>> f
0
>>> f=1.*2/3
>>> f
0.6666666666666663
>>> g=float(2)/3
>>> g
0.6666666666666663
```

Mathematische Funktionen, wie `sin, cos, exp, etc.`, sind im Python Modul `math` enthalten und können über `math.function` angesprochen werden, zuvor `import math` ausführen:

```
>>> import math
>>> math.sin(0.5)
0.47942553860420301
>>> math.log10(65)
1.8129133566428555
>>> ...
```

Name	Purpose
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan2(x,y)	Arc tangent of x/y
ceil(x)	Ceiling of x; the largest integer equal to or greater than x
cos(x)	Cosine of x
cosh(x)	Hyperbolic cosine of x
exp(x)	e raised to the power of x
fabs(x)	Absolute value of x
floor(x)	Floor of x; the largest integer equal to or less than x
fmod(x)	x modulo y
frexp(x)	The mantissa and exponent for x.
hypot(x, y)	Euclidean distance, sqrt(x*x + y*y)
ldexp(x, i)	x * (2**i)
log(x)	Natural logarithm of x
log10(x)	Base 10 logarithm of x
modf(x)	Return the fractional and integer parts of x
pow(x, y)	x raised to the power of y
sin(x)	Sine of x
sinh(x)	Hyperbolic sine of x
sqrt(x)	Square root of x
tan(x)	Tangent of x
tanh(x)	Hyperbolic tangent of x

Komplexe Zahlen

In Python sind komplexe Zahlen als Paar von 2 Gleitkommazahlen implementiert.

- **j** kennzeichnet imaginäre Einheit
- komplexe Zahl als **Real + Imagj** definieren, z.B. `3 + 2j`
- Alle gängigen Operationen: `+, -, *, /, **, abs(), ==`
- Zugriff auf Real- bzw Imaginärteil mittels: `var.real, var.imag`

```
>>> z=2+4j
>>> abs(z)
4.472135954999579 6
>>> v=3-2j
>>> z/v
(-0.15384615384615385+1.2307692307692308j)
>>> z**3
(-88-16j)
>>> z**v
(775.7930102695509+262.01856311381101j)
>>> z.imag
4.0
```

```
>>> z.real
2.0
```

Mathematische Funktionen für komplexe Zahlen sind analog zu Gleitkommazahlen definiert, nur muss das Modul `cmath` geladen werden (`import cmath`)

```
>>> import cmath
>>> cmath.exp(1+1j)
(1.4686939399158851+2.2873552871788423j)
>>> cmath.exp(math.pi/2 * 1j)
(6.123031769111886e-17+1j)
>>> ...
```

Liste der reservierten Wörter in Python

Dürfen nicht als Namen für Variablen, Funktionen, Klassen verwendet werden!

```
and del for is raise
assert elif from lambda return
break else global not try
class except if or while
continue exec import pass yield
def finally in print
```

Hinzu kommen noch die Namen gängiger Typen, Klassen und Funktionen wie

```
bool float sin exp ...
```

Wird zwar syntaktisch von Python akzeptiert, führt aber leicht ins Chaos ...

1.6 Strings, Lists und Sequences

Strings – Zeichenketten

In Python spezieller Datentyp *string* zur Darstellung von Zeichenketten und Operationen mit Zeichenketten.

- Definition wahlweise mit single oder double Quotes
- Aneinanderhängen mit `+`
- Sub–Strings mit `[]` Operator
- Viele nützliche Funktionen zum Konvertieren, Zählen, Sub–Strings suchen, ... (siehe Python Strings)

```
a = 'Hello '                  # Single quotes                    1
b = "World "                  # Double quotes                    2
c = "Bob said 'hey there.'" # A mix of both                      3
e = b + a  # concatenate                                         4
print e      # -> World Hello                                    5
f = a*3      # -> 'Hello Hello Hello '                           6
c[5]         # (5+1). Element von c -> 'a'                       7
```

```
c[2:8]        # (3.-9.). Element von c -> 'b said'                          8
c.count('e') # wie oft kommt 'e' vor in c -> 3                             9
c.upper()     # -> "BOB SAID 'HEY THERE.'"                                  10
# string methods can be applied on string constants directly               11
"abracadabra".find("ra") # -> 2 (=Index of substring "ra")                 12
" a b ".strip() # -> "a b" (remove leading and trailing white space)        13
"abracadabra".replace("ra", "lu") # -> "ablucadablu"                       14
#                                                                          15
# string methods don't change original string,                             16
# if needed a new string is created                                        17
e="abracadabra"                                                            18
print e.replace("ra", "lu") # -> "ablucadablu"                             19
print e # -> "abracadabra"                                                 20
```

Man kann mehrere Methodenaufrufe aneinander-reihen (=chaining), oft praktisch, aber leicht unleserlich ...

```
# chaining method calls                                          1
element = "cesium"                                               2
print ':' + element.upper()[4:7].center(10) + ':'                3
#:      UM     :                                                 4
```

List – Arrays in Python

Ein wichtiges Konstrukt in allen Programmiersprachen sind *arrays*, das sind

- Liste von Variablen oder Objekten
- Die einzelnen Elemente können über ihren *Index* angesprochen werden

In Python gibt es dafür die **list**, wie bei *strings* dient sie nicht nur zum Abspeichern der Daten sondern stellt viele nützliche Funktionen bereit.

- Erzeugen mit eckige Klammer und Elemente: `A = [2,5,4,77,43]`
- Python zählt (wie JAVA/C++) von **0 bis Länge–1**
- Länge kann abgefragt werden mit `len(A)`
- Zugriff auf nicht-existierende Elemente gibt Fehler
- Im Gegensatz zu einfachen JAVA/C++ Arrays ist die **Python list** sehr flexibel und bietet viele Hilfsfunktionen:
  - Anhängen, Einfügen, Löschen von Elementen: `A.append(42); A.insert(2,55); del A[3]`
  - Hilfsfunktionen zum Sortieren, Umdrehen, ...
- Kann unterschiedliche Datentypen enthalten, auch andere list

- List bestimmter Länge anlegen und füllen mit z.B.
  `K = [0]*20` (Liste von 20 int alle auf 0 gesetzt)
- List mit aufeinanderfolgenden Zahlen mit z.B.
  `range(10)` (Liste von 10 int mit Werten von 0 bis 9)

```
>>> L = [ 1, 5, 9, 45, 3, 72, 27 ]  # Definition list
>>> len(L) # Laenge
7
>>> L[6] # Element 6+1
27
>>> L[8] # ausserhalb
#Traceback (most recent call last):
#  File "<stdin>", line 1, in ?
#IndexError: list index out of range
>>> L[2] = 11 # aendern von Element 3
>>> L
[1, 5, 11, 45, 3, 72, 27]
>>> L.sort()
>>> L
[1, 3, 5, 9, 27, 45, 72]
```

```
>>> L.append(87) # anhaengen
>>> L
[1, 5, 11, 45, 3, 72, 27, 87]
>>>
>>> L.insert(2,33) # einfuegen
>>> L
[1, 5, 33, 11, 45, 3, 72, 27, 87]
>>>
>>> del L[3:5] # Loeschen Element 4-6
>>> L
[1, 5, 33, 3, 72, 27, 87]
>>>
>>> L.sort() # sortieren
>>> L
[1, 3, 5, 27, 33, 72, 87]
>>>
>>> L.reverse() # Reihenfolge umdrehen
>>> L
[87, 72, 33, 27, 5, 3, 1]
....
```

```
>>> K=[0]*10 # List mit 10 Elementen, alle 0
>>> K
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> M = [1, 2.45, (1+4j), L[1:5]]    # mixed list
>>> M
[1, 2.4500000000000002, (1+4j), [3, 9, 27, 111]]
>>> M[3][2]
27
...
>>> G = range(20)
>>> G
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>>
>>> G = range(1,11) # Zahlen 1-10
>>> G = range(1,11,2) # Zahlen 1,3,5,7,9
```

2D Arrays oder Matrizen

Einfach auf mehr Dimensionen zu erweitern:

- Matrix mit 4×3 `int` Elementen: `M = [[0]*4,[0]*4,[0]*4]`
- oder direkt initialisieren: `M = [[1,2,3,4],[2,3,4,5],[3,4,5,6]]`
- Zugriff via `M[0][0]` (1. Element) bis `M[2][3]` (letztes Element).
- **Vorsicht:** `M = [[0]*4]*3` klappt nicht (produziert 3 Verweise auf diesselbe Liste mit 4 Werten).
  Alternative: **list-comprehension**
  ```
  M = [ [0]*4 for i in range(3)]
  ```
- Analog Erweiterung auf höhere Dimensionen > 2:
  ```
  M = [[[0]*4 for i in range(3)] for j in range(2)]
  ```

Python Sequences

string und **list** gehören beide zur Gruppe der Python Sequences:

- **string** ist eine Art **list**, allerdings sind die Elemente nur *chars*
- **string** ist *immutable*, d.h. einzelne Elemente können nicht überschrieben werden:

```
S = 'Balduin'
S[1] # --> 'a'
S[1]='o' # --> Fehler
```

- Dafür viele extra Funktionen für String (`S.upper(), S.lower(), ...`)
- Variante von **list** ist **tuple**: Anlegen mit runder Klammer () statt [], z.B. `T = (1,4,6,8)`.
- **tuple** sind wie **string** **immutable**, man kann Elemente nicht ändern, löschen, einfügen, anhängen.

Konvertierungen möglich:

```
>>> T = (1,4,6,8) # tuple                                    1
>>> T[1] # -> 4                                              2
>>> T[1] = 7 # error for tuple                               3
#Traceback (most recent call last):                          4
#  File "<stdin>", line 1, in ?                              5
```

```
#TypeError: object doesn't support item assignment
>>> V = list(T) # tuple -> list
>>> V
[1, 4, 6, 8]
>>> V[1] = 7 # ok for list
...
>>> S = 'Balduin'
>>> S[1] = 'o' # Fehler
>>> B = list(S) # Konvertierung in list
['B', 'a', 'l', 'd', 'u', 'i', 'n']
>>> B[1] = 'o' # ok
>>> S = "".join(B) # zurueck nach string
'Bolduin'
```

Slicing – Stückchen rausschneiden

...

```
>>> a=range(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> a[4:7]
[4, 5, 6]
>>> a[-3:]
[7, 8, 9]
>>> a[-3:-1]
[7, 8]
>>> a[-3:2]
[]
#
>>> s = "Donaudampfschiffahrtsgesellschaftkapitaenskajuetenblumenvase"
>>> s[-10:] # extract last 10 chars
'blumenvase'
```

1.7 Control–statements

”Eigentliches Programmieren” braucht Kontrollstrukturen:
Bedingungen testen und ggf. ausführen,
Zähl–Schleifen, Endlosschleifen, etc.

if: Standard-Konstruktion für einfache Verzweigungen

```
if ( expr ):
        statements
```

- True: `statements` werden ausgeführt
- False: `statements` werden übersprungen

if–else: Verzweigung und Alternative

```
if ( expr ):
        statements_1
else:
        statements_2
```

- True: `statements_1` werden ausgeführt
- False: `statements_2` werden ausgeführt

Typ `bool` relativ neu in Python. In älteren Versionen einfach `int` verwendet:

- `0` entspricht `False`
- `!= 0` entspricht `True`

Wichtig: Für `if` bzw. `elif` kann `expr` vom Typ `bool` oder `int` sein.

Häufiger Fehler in C/C++: `if ( a = b ) ...` statt `if ( a == b ) ...`
In Python nicht möglich, Syntax–Fehler !

if–elif–elif–...–else: Verzweigung und mehrere Alternativen

```
if ( expr1 ):
        statements_1
elif ( expr2 ):
        statements_2
elif ( expr3 ):
        statements_3
...
else:
        statements_n
```

```
if ( a < b ): a = b # einfaches if, auch in 1 Zeile
...
```

```
if ( a < b ): # if - else                                   3
    a = b                                                   4
else:                                                       5
    b = a                                                   6
...                                                         7
if ( a < b ): # if - else if - else                         8
    a = b                                                   9
elif ( a > b ):                                             10
    b = a                                                   11
elif ( a == b ):                                            12
    ...                                                     13
else: # wenn alles andre fehlschlaegt ..                    14
    print "What else could I test ?"                        15
```

if Abfrage in Listen

Schönes Feature in Python ist dass auch für Listen einfache if–Abfrage möglich ist:

```
primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]
testnum = 9
if ( testnum in primes ):
    print testnum + " ist prime"
...
```

So nicht möglich in C++, JAVA, ...

Schleifen

while–loops:

```
while ( expr ):
    statements
```

Zeilen `statements` werden ausgeführt solange Ausdruck `expr` wahr ist (`True`)

```
# klassische Zaehlschleife
i=0
while ( i<10):
    print i
    i += 1
#...
# Summiere Quadratzahlen 1**2..10**2
sum=0; i=0
while ( i<10):
    i += 1
    sum += i*i
#...
# Einlesen
n=0
```

```
while ( True ):     # Endlosschleife                    15
    s=f.readline()                                      16
    if ( len(s) == 0 ): break                           17
    n += 1                                              18
#                                                       19
print "Lines read ", n                                  20
```

`while` Schleifen universell verwendbar, insbesondere bei *nicht absehbarer* Abbruchbedingung, Ende als Reaktion auf z.B. bestimmtes Ereignis, End-of-file.

for–loops:
sind Alternative, insbesondere für Zählschleifen oder wenn über Elemente einer Python Sequenz iteriert werden soll.

- Syntax:
  ```
  for var in Sequence:
      statements
  ```
- Für jedes Element einer Sequenz (list, string, ...) wird `statement` ausgeführt. Das jeweilige Element steht in `var`.
- Mit der Funktion `range(n)` kann man leicht Sequenzen erzeugen und damit Abzählschleifen erstellen `for i in range(10):  # Zahlen von 0-9`

```
S = [1,4,5,9,56]
for i in S:
    print i
#
s = "Donaudampfschiffahrtsgesellschaft"
ca = 0
for c in s: # iteriere ueber String
    if (c=='a'):
```

```
    ca += 1    # count 'a'                                                    9
#                                                                            10
print ca, " 'a' in ", s                                                      11
## Note: bad python programming style, better use string function            12
# s.count('a')                                                               13
a=[[0]*3]*4 # 4x3 Matrix                                                     14
for i in range(4): # Verschachtelte Schleifen                                15
    for j in range(3):                                                       16
        a[i][j] = i*j        # Fuellen von Matrix                            17
#                                                                            18
```

break/continue :

Innerhalb des `statements` Blocks von `for` bzw. `while` Schleifen kann man mit

- `break` die Schleife beenden oder mit
- `continue` zur nächsten Iteration springen.

```
while ( True ):
    ...
    if ( ... ): continue # Springt ans Ende der Schleife
    if ( ... ): break;   # Springt aus der Schleife raus
    ...
```

for/while und else

- Interessantes Feature in Python ist die Kombination der Schleifen mit einem anschliessenden **else** Block.
- Dieser wird ausgeführt wenn die Schleife **nicht** mit **break** beendet wurde.
- Typische Anwendung:
  - In der Schleife wird das Eintreten eines bestimmten Ereignisses untersucht, z.B. Suche nach Elementen, Konvergenz, ...
  - Bei Eintritt wird mit **break** abgebrochen und **else** Block ignoriert.
  - Bei Nicht–Eintreten läuft die Schleife bis zum Ende, anschliessend wird **else** Block ausgeführt.

```
#                                                                    1
s = "Donaudampfschiffahrtsgesellschaft"                              2
for c in s: # iteriere ueber String                                  3
    if (c=='x'):                                                     4
        print 'x found in ', s                                       5
        break                                                        6
else:                                                                7
```

```
print 'x not found in ', s                                    8
                                                               9
```

1.8 Funktionen

Die klassische Art (*modulares Programmieren*) ein großes, komplexes Programm überschaubar zu gestalten ist die Aufteilung der Funktionalität in kleinere, eigenständige Untereinheiten für bestimmte Aufgaben, d.h. Einteilung in **Funktionen**.

```
def GravForce( mass1, mass2, distance): # Kopf der Funktion                    1
    GRAV_CONST = 6.673e-11 # Gravitationskonstante m^3 / ( kg s^2 )           2
    f = GRAV_CONST * mass1 * mass2 / distance**2 # Kraft ausrechnen           3
    return(f)                                    # Ergebnis zurueckgeben      4
mSonne = 1.9889e30; mErde = 5.974e24; dErdeSonne = 1.49597e11; rErde = 6.378e6 5
# Aufruf von Funktion GravForce                                               6
gErdeSonne = GravForce( mSonne, mErde, dErdeSonne ) # Kraft Sonne-Erde         7
print gErdeSonne                                                              8
gPerson = GravForce( mErde, 80., rErde )          # Gewichtskraft 80 kg auf Erde 9
print gPerson                                                                 10
```

1-4 Definition der Funktion `GravForce`.

Zeile 1 Deklaration der Funktion `GravForce`:

Angabe von Namen der Funktion sowie Namen der Parameter.

Funktionsaufruf bewirkt :

- Bei Programmlauf wird zu der entsprechenden Funktion gesprungen
- Die *Parameter* Funktion (`mass1, mass2, ...`) erhalten beim Aufruf die übergebenen *aktuellen Werte der Argumente* (`mSonne, mErde, ...`), d.h.

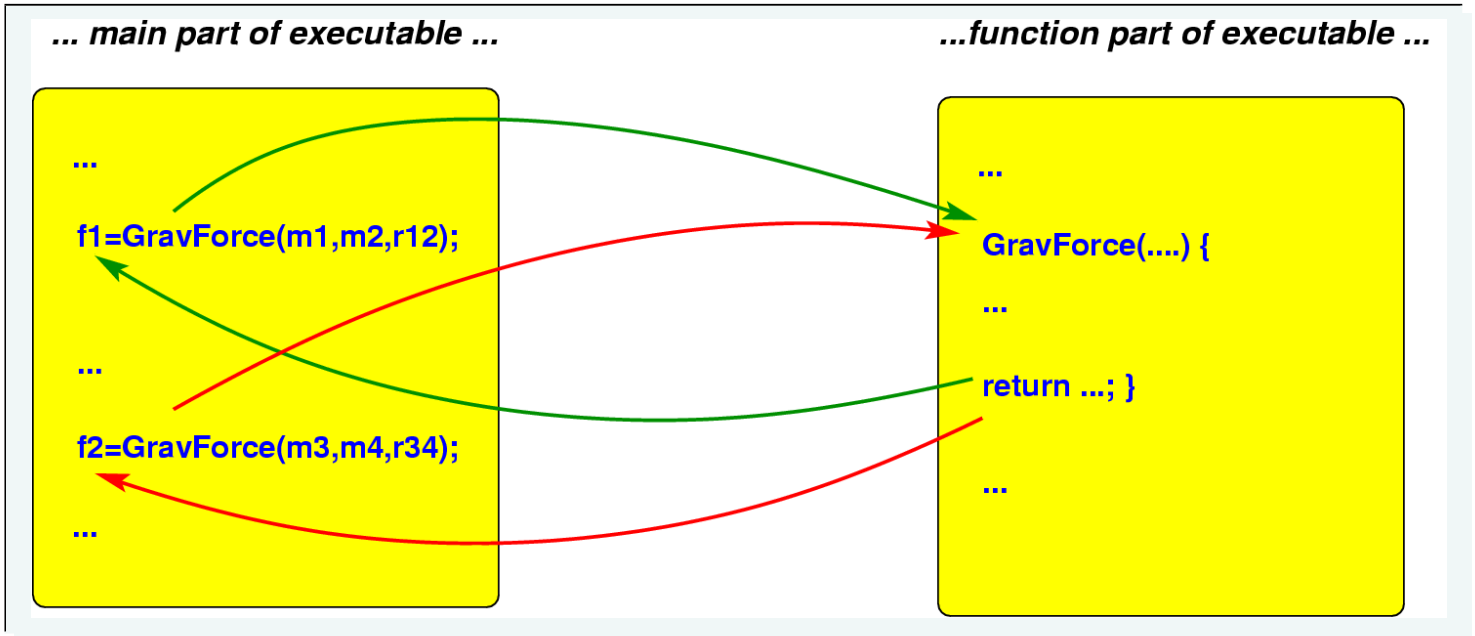
  ```
  mass1 = mSonne; mass2 = mErde; distance = dErdeSonne
  ```

  bzw.

  ```
  mass1 = mErde; mass2 = 80.; distance = rErde
  ```

- `return ...` am Ende der Funktion bewirkt Rückkehr an die rufende Stelle und ggf. Rückgabe des Werts
- Rückgabewert kann Variable in rufendem Programm zugewiesen werden

  ```
  gPerson = 783.
  ```



Programm–Design

- Aus Sicht der Anwendungsprogrammiererin ist die Funktion eine *Black Box*.
- Es interessiert nur, was die Funktion leisten soll und wie die *Schnittstelle* aussieht, d.h. Name der Funktion, Typ und Zahl der Argumente.
- Details der **Implementierung** sollten nicht relevant sein.

Funktionen brauchen nicht immer Argumente ...

```
time.ctime() # returns current date and time
'Fri Mar 18 14:06:00 2005'
```

Funktionen müssen nicht unbedingt was zurückgeben

```
def greeting(): # Begruessung                                    1
    hour=time.localtime()[3] # extract Stunde                    2
    gruss='Guten Tag'                                            3
    if (4<hour<10):                                              4
        gruss='Guten Morgen'                                     5
    elif   ( 11<hour<13 ):                                       6
        gruss='Mahlzeit'                                         7
    elif   ( 18<hour<21 ):                                       8
        gruss='Guten Abend'                                      9
    elif   (hour>21 or hour < 4):                                10
        gruss='Gute Nacht'                                       11
#                                                                12
    print gruss                                                  13
```

Funktionen können mehrere Werte zurückgeben

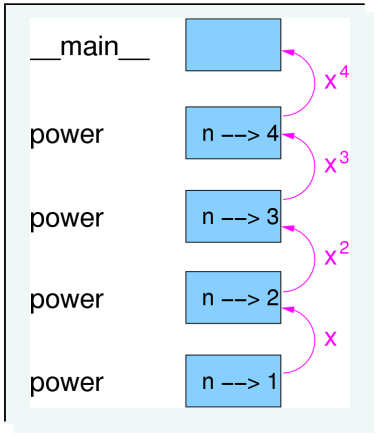
(... nicht möglich in Java, FORTRAN, C/C++)

```
import math
def root( A, B, C):
    """ Returns the two roots of
      the quadratic equation A*x*x + B*x + C = 0."""
    disc = B*B - 4*A*C
    return (   (-B - math.sqrt(disc)) / (2*A),     (-B + math.sqrt(disc)) / (2*A) )
#
x1,x2 = root( 1., 5., -3.)      # return both solutions
```

Rekursive Funktionsaufrufe (Funktion ruft sich selbst) sind möglich:

```
def power( x, n ): # define power function                    1
    if ( n > 1 ) :                                            2
    return( x * power(x, n−1) ) # rekursiver Aufruf           3
    else:                                                     4
        return( x )                                           5
#                                                             6
```

```
power( x, 4 )
```



Default Werte für Parameter

- Es können Default Werte (=Standard–Vorgaben) für die Parameter bei Definition der Funktion angegeben werden. `par=value`
- Bei Aufruf können diese Argumente weggelassen werden, dann werden die default Werte benutzt.

Keyword Argumente

Normalerweise wird beim Aufruf nur die Liste der Argumente übergeben und die Zuordnung der Argumente zu den Funktionsparametern geht nach der Position,
d.h. *1. Par = 1. Arg, 2. Par = 2. Arg, ...*

Alternativ kann man beim Aufruf explizit den Namen **(=keyword)** einzelner Parameter angeben zusammen mit dem Argument: `par-name=arg-wert`

```
def parabel( x, a=0., b=0., c=0. ): # define parabel (poly 2)
    return( x**2 * a + x * b + c )
#
>>> parabel(2, 1., 2., 3.)
11.0
>>> parabel(2) # default Werte fuer a,b,c
0.0
```

```
>>> parabel(2, 1.) # a=1, default Werte fuer b,c
4.0
>>> parabel(2, c=3) # keyword-argument c=3, default Werte fuer a,b
3.0
>>> parabel(2, b=2) # keyword-argument b=2, default Werte fuer a,c
4.0
>>> parabel(2, c=0, b=5)  # keyword-argument c,b, default Wert fuer a
10.0
>>> parabel()
#Traceback (most recent call last):
#  File "<stdin>", line 1, in ?
#TypeError: parabel() takes at least 1 argument (0 given)
```

lambda Funktionen

Anonyme Kurzform für Funktionen. Sie werden nicht mit `def function-name ...` deklariert wie bei den üblichen Funktionen, sondern quasi anonym in einer Zeile:

```
def squareAdd( x, y):
    return( x**2 + y**2)
#
f = lambda x, y : x**2 + y**2    # definition als lambda function in einer Zeile
#
# Aufruf
squareAdd( 2., 3. ) # = 13
#
f( 2., 3.) # aequivalent = 13
```

lambda functions sind vor allem dann nützlich wenn als Argument beim Aufruf einer Funktion eine Funktion benötigt wird. Sehr häufig bei **GUI Funktionen**: Verknüpfung von Ereignis (z.B. Mausklick) und Aktion (*call-back function*). ⇒ Später

Weiteres Beispiel: Numerik – Nullstellen von Funktion

```
def suche( x1, x2, f ) :
    " suche Nullstelle von Funktion f zwischen x1 und x2"
    f1 = f(x1);
    f2 = f(x2);
    i = 0
    while  i < 1000 : #  Abbruch nach 1000 Iterationen
        xn = (x1+x2)/2.     # Neues x in Mitte
        fn = f(xn);         # Funktionswert dazu
        if   abs(fn) < 1e-6 : break # Konvergenz: 1e-6 an Nullstelle, dann Abbruch
        if   fn*f2 > 0.   :  # gleiches Vorzeichen wir f2 ?
            x2,f2 = xn,fn      #  dann ersetze x2,f2
        else:
            x1,f1 = xn,fn      # andernfalls x1,f1

        i = i+1
    else:                      # keine Konvergenz
        print 'Error: Keine Konvergenz fuer Nullstellensuche', fn

    return xn
#
```

```
# Test                                                                    21
def myf(x) :                                                              22
    return( math.exp(-x) -x);                                             23
#                                                                         24
suche( 0., 3., math.cos)                  # Funktion aus math             25
suche( 0., 3., myf)                       # selbst-definierte Funktion    26
suche( 0., 3., lambda x : math.exp(x) -2. ) # Kurzform mit lambda func     27
```

- Algorithmus zur Nullstellensuche ist generisch, d.h. unabhängig von spezifischer Funktion.
- Konkrete Funktion als Argument bei Aufruf `nullstelle.suche( ...)`
- lambda functions erlauben kompakte Verwendung/Schreibweise

1.9 Input/Output Grundlagen

I/O in Python recht einfach, mächtige Funktionen eingebaut.

Standard-Ausgabe mittels `print` schon gezeigt:

```
>>> print 'Pi = ', math.pi
Pi = 3.14159265359
```

Gegenstück ist **Standard-Eingabe** mittels `input()` :

```
>>> h=input()                                    1
2.7                                              2
>>> h                                            3
2.7000000000000002                               4
>>> h=input()                                    5
"baby"                                           6
>>> h                                            7
'baby'                                           8
>>> h=input()                                    9
8+8j                                             10
>>> h                                            11
(8+8j)                                           12
```

```
>>> h=input("Zahl:")
Zahl:100
>>> (a,b,c)=input("Drei Zahlen: ")
Drei Zahlen: 25, 64, 77
#
>>> d=raw_input()
hello there , wie geht es heute?
>>> d
"hello there , wie geht es heute?"
>>> d=raw_input()
123
>>> d # raw_input liefert immer string
'123'
```

- Zeilenweise Eingabe
- automatische Konvertierung in Standard-Datentypen
- Als Argument kann `input()` ein prompt-String übergeben werden (Eingabe-Aufforderung) `input(prompt-string)`

- Gleichzeitig können mehrere Variablen gelesen werden
- Bei komplexem Input besser mittels `raw_input()` in String einlesen und mittels string Funktionen weiter verarbeiten

Ein/Ausgabe mit Dateien

ist sehr komfortabel in Python:

- File Objekt erstellen mit `f=open("filename")` (Lesen) bzw. `f=open("filename","w")` zum Schreiben.
- Lese– bzw Schreib–operationen immer mit Strings
- `f.readlines()` liest alle Zeilen und liefert *list of strings* zurück
- Alternativ iterieren mit `for line in f:` über alle Zeilen
- Ausgabe mit `f.write("Some Txt String")`
- Schliessen nicht vergessen `f.close()`

```
f=open("numbers.dat") # Oeffne file                                    1
zeilen=f.readlines()      # Lese alle Zeilen -> list of strings        2
len(zeilen) # 100                                                      3
zeilen[8] # -> '0.720203\n'                                            4
b = float(zeilen[8]) # Konversion string -> float                      5
b # -> 0.72020300000000004                                             6
f.close # file schliessen                                              7
```

```
#                                                                    8
f=open("numbers.dat") # Oeffne file                                  9
for line in f: # iteriere ueber alle Zeilen                          10
    b = float(line)      # Konversion string -> float              11
f.close # file schliessen                                            12
# ...                                                                13
f=open("newfile.txt","w") # Oeffne File zum Schreiben                14
f.write("Nummer "+str(8.23)+"\n") # Output muss string sein, "\n" fuer Zeilenvorschub  15
```

Konversionen in/von Strings

- Für Standard-Datentypen gibt es Funktion `str()` um `int, float, complex` in String umzuwandeln:
  ```
  str(667); str(3.1415); str(1+6j)
  ```
- Flexible Formatierung ähnlich wie in C mittels Format string:
  ```
  "Fahrenheit %6.1f = Celsius %6.3f" % (fahr , celsius)
  ```
- Konvertierung String nach ... einfach mit:
  ```
  int("667"); float("3.1415"); complex("1+6j")
  ```

Kommandozeile:

Eine weitere Möglichkeit zur Kommunikation mit dem Programm ist die Übergabe von Argumenten beim Start:

```
# ReadArgs.py                                                    1
# Aufruf in shell:                                               2
# python ReadArgs.py 99 3.1415 blabla                            3
import sys                                                       4
n=0                                                              5
for arg in sys.argv: # Argumente als Liste von strings           6
    print "arg %d : %s " % ( n, arg )                            7
    n += 1                                                       8
#                                                                9
# arg 0 : ReadArgs.py                                            10
# arg 1 : 99                                                     11
# arg 2 : 3.1415                                                 12
# arg 3 : blabla                                                 13
```

- Argumente als list von strings verfügbar in `sys.argv`

- 1. Argument (`sys.argv[0]`) ist automatisch immer Name des Programms

I/O über Internet

Sehr einfach lassen sich auch Files direkt über das Internet öffnen und lesen:

- Module laden: `import urllib2`
- File über URL–Adresse öffnen: `f=urllib2.urlopen("URL-name")`
- Und dann wie bei normalem File-IO Zeilen prozessieren, z.B: `f.readlines()`
- ...

```
import urllib2
# f=urllib2.urlopen("http://www-static.etp.physik.uni-muenchen.de/kurs/Computing/python/source/vorwahl.txt")
# goo.gl shortened URL
f=urllib2.urlopen("http://goo.gl/8AfpKr")
# f=open("numbers.dat") # Oeffne file
zeilen=f.readlines()      # Lese alle Zeilen -> list of strings
len(zeilen) # 100
```

1.10 Python Module

Per Default bietet Python eine gewisse Grundfunktionalität. Für weitergehende Anwendungen müssen aber i.d.R. weitere **Python Module** eingebunden werden.

Drei Arten dieser Module kann man unterscheiden:

- Module der **Python Standard–Distribution**, z.B. *math, cmath, sys, random, Tkinter, ...* (siehe Python 2.7.6 Module Index).
- **Externe Module:** Riesige Zahl an weiteren Python Modulen erhältlich. Aber individueller Download und Installation erforderlich. *(⇒ Später)*
- **Eigene Module:** Grundlegendes Design–Prinzip ist Programm in Untereinheiten aufzuteilen, d.h. in verschiedene Python Module.

Verwendung von Modules einfach durch Statement:
`import module-name`

Das funktioniert ohne weiteres

- für die Python Standard Modules
- sowie für selbst–erstellte Modules, die den Filenamen `module-name.py` tragen und sich im momentanen Arbeitsverzeichnis befinden.

(Für andere, externe Module muss i.a. der PYTHONPATH angepasst werden ⇒ später)

Beispiel: Python Module für weitere mathematische Funktionen

```
# mathutil.py: Verschiedene mathematische Hilfsfunktionen
import math
pi2=math.pi*2.
def fak(n):
    "Berechne Fakultaet"
    ...
def gcd( a, b):
    "Berechne groessten gemeinsamen Teiler von a und b"
    ...
def fibonacci(n):
    "Berechne n-te Fibonacci Zahl"
    ...
...
```

Verwendung in anderem Python Skript oder interaktiv:
Zuerst Laden des Moduls und dann nutzen der Variablen/Funktionen via `module-name.function(), module-name.variable`, z.B.:

```
import mathutil # import package                                    1
f20 = mathutil.fak(20) # call function from mathutil               2
umfang = mathutil.pi2 * radius # use variable from mathutil        3
```

1.11 Python Namespaces

Wie in JAVA/C++ gibt es in Python das Konzept der Namespaces *(=Namensräume)*.

- Namespaces helfen Konflikte mit Variablen/Funktions–Namen zu vermeiden.
- Per default definiert jedes Python Module seinen **eigenen Namespace**, bei Verwendung in anderen Modulen wird Namespace (=module-name) explizit mit angegeben, bei Verwendung von Funktionen oder Variablen des entsprechenden Moduls.

```
import math      # import package                                    1
import mathutil  # import package                                    2
umfang = mathutil.pi2 * radius  #                                   3
...                                                                 4
pi2=math.pi**2  # Pi^2, kein Konflikt mit pi2 aus      mathutil     5
...                                                                 6
```

Variante von *import*, so dass Module gleichen namespace verwendet: Statt

```
import sys; print sys.argv # explizite namespace Angabe
```

auch möglich:

```
from sys import argv; print argv # argv jetzt im std namespace
```

oder

```
from sys import *; print argv # alle Variablen/Funktione aus sys jetzt
im std namespace
```

Inbesondere letzteres besser nicht verwenden!

Globale und lokale Variablen:

Variablen, die auf Ebene eines Modules definiert sind, sind **globale** Variablen

- sie sind überall verfügbar, wo das jeweilige Module verfügbar ist
- sie bleiben erhalten während des gesamten weiteren Programmlaufs

Dagegen sind Variablen, die innerhalb einer Funktion definiert sind, per default **lokale** Variablen

- sie sind nur inerhalb der jeweiligen Funktion verfügbar
- sie existieren nur solange die Funktion aktiv ist, d.h. Python die Anweisungen in der Funktion abarbeitet.

Alternativ kann man innerhalb einer Funktion Variablen als **global** deklarieren, dann haben sie globale Gültigkeit und Lebensdauer, sobald die Funktion einmal gerufen wurde.

```
def setx1(a):                                    1
    x=a  # local x                               2
    print 'x in setx1 = ', x                     3
    y1=111                                       4
def setx2(a):                                    5
    global x, y2 # global x, y2                  6
```

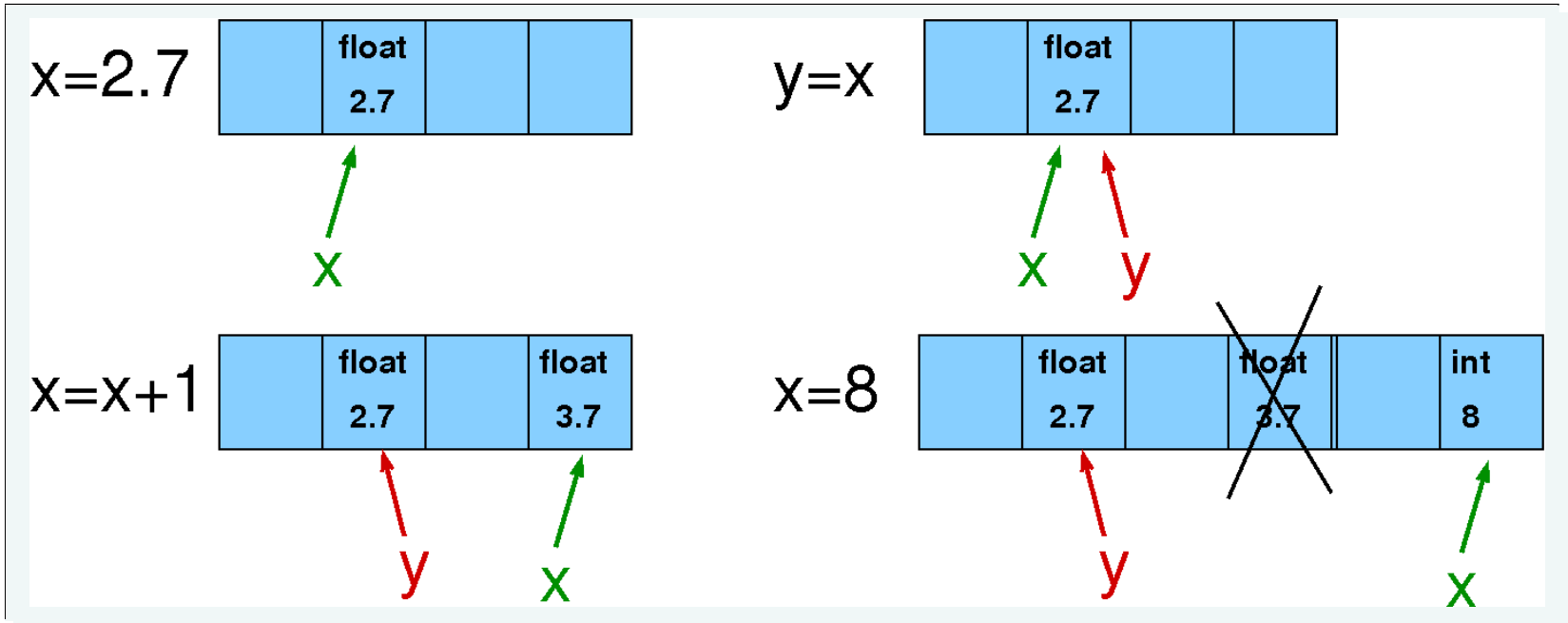
```
    x=a                                                                    7
    print 'x in setx2 = ', x                                               8
    y2 = 222                                                               9
x=5                                                                        10
setx1(11)                                                                  11
print 'x after setx1 = ', x                                                12
print y2 # Error global in y2, but not yet defined since setx2 not called  13
setx2(22)                                                                  14
print 'x after setx2 = ', x                                                15
#                                                                          16
print y1 # error only local in setx1                                       17
print y2 # ok now                                                          18
```

1.12 Variablen vs Objekte in Python

Python Variablen spielen eine ganz andere Rolle als Variablen in C++/JAVA.

- Zentrales Element in Python sind *Python-Objects*. Diese entstehen als Ergebnis einer Operation, Funktionsaufruf, o.a.
- *Python-Objects* haben einen bestimmten Typ *(int, float, string, list, class name, ...)* und belegen den entsprechenden Speicherplatz.
- Zuweisung in Python, z.B. `x = 2.7` bewirkt lediglich, dass das float-Object mit Wert 2.7 und Speicherplatz 64 bit über den Namen `x` angesprochen werden kann.
- Zuweisung `y = x` bewirkt **nicht** Kopie sondern einfach nur weiterer Namen mit dem dasselbe Objekt angesprochen wird.
- "Änderung" der Variablen `x = x + 1` bewirkt nicht Änderung des ursprünglichen Python-Objects sondern Erzeugung eines neuen unabhängigen Objekts mit Wert 2.7 + 1 = 3.7. Mit `x` wird jetzt dieses neue Objekt angesprochen, das ursprüngliche ist nach wie vor vorhanden, mit `y` kann es angesprochen werden.

In C++/JAVA Variable und Objekt dasselbe (zumindest für primitive Datentypen int, float, ...).


x=2.7
float
2.7
x
y=x
float
2.7
x
y
x=x+1
float
2.7
float
3.7
y
x
x=8
float
2.7
float
3.7
int
8
y
x

1.13 Programme debuggen

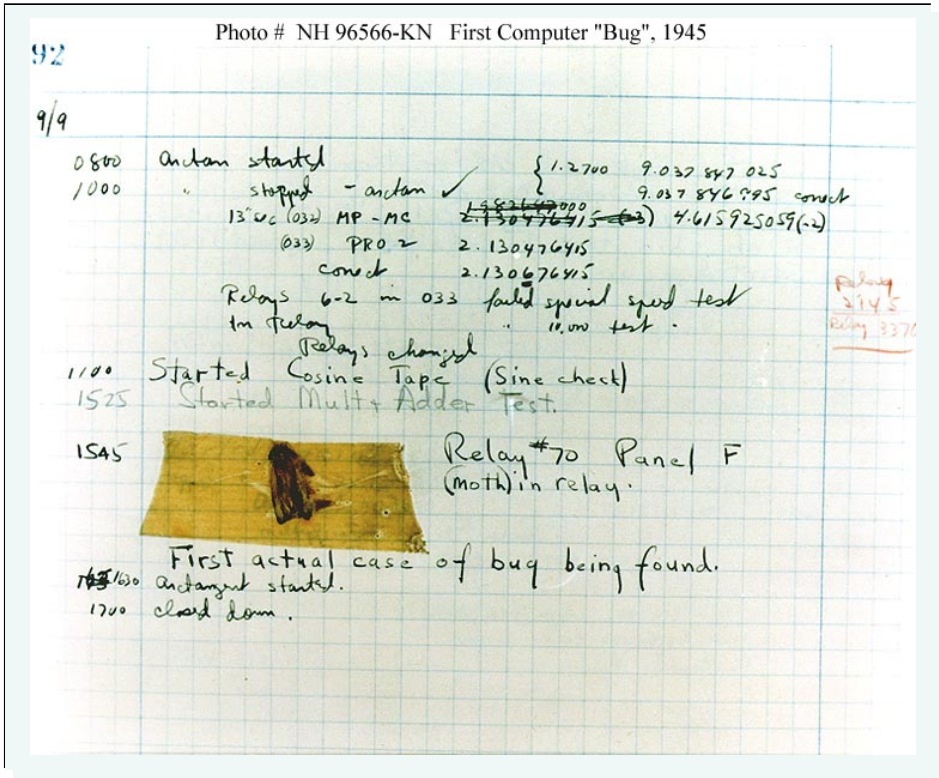
Programmieren ist eine komplexe Angelegenheit. Nur die wenigsten schaffen es mehr als 10 Zeilen fehlerfrei zu schreiben. Erst recht nicht als Anfänger. Häufig meiste Zeit bei Programmentwicklung für Fehlersuche (=debuggen) und Korrektur.

Hilfreich Fehler zu klassifizieren:

- **Syntax–Fehler:** Der Python Interpreter kann ein Programm nur übersetzen wenn die Syntax des Programms korrekt ist, d.h. den Python Regeln entspricht. Beispielsweise nicht kompatible Datentypen in Operationen, falsche key-words, fehlende Klammern, unbekannte Funktionen oder Klasse, und vieles andere. Besonders für Anfänger häufigste Fehlerquelle und ziemlich lästig wegen kryptischer Fehlermeldungen des Interpreters, dennoch einfachste Art von Fehler.
- **Laufzeit–Fehler** treten beim Ausführen des Programms auf (= run-time errors) und führen i.d.R. zum Abbruch des Programms, z.B. Division durch Null, File existiert nicht, Speicherüberlauf, Zugriff auf nicht-existierende Array Elemente, etc. Manchmal offensichtlich, oft aber auch schwer und mühsam zu finden. Python ziemlich sicher und gutmütig im Vergleich mit C, C++, FORTRAN, ... In Python integriertes System zum **exception–handling** (weiteres später).
- **Semantik–Fehler** Programm kompiliert und läuft, tut aber nicht das was es soll. Mal einfach, kann aber auch Menschenleben oder Milliarden kosten (Ariane-5 Absturz, Patriot-Crash, ..., siehe Collection of Software Bugs).

The First ”Computer Bug”

Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945. The operators affixed the moth to the computer log, with the entry: ”First actual case of bug being found”. They put out the word that they had ”debugged” the machine, thus introducing the term ”debugging a computer program”. In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia.



Apple goto-fail bug

Aktuelles (2014) Beispiel für sicherheitsrelevanten Bug in Apple IOS 7 *(C-Programmcode)*.

```
// ...                                                                    1
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;                               2
hashOut.length = SSL_SHA1_DIGEST_LEN;                                     3
if ((err = SSLFreeBuffer(&hashCtx)) != 0)                                 4
    goto fail;                                                            5
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)                       6
    goto fail;                                                            7
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)             8
    goto fail;                                                            9
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)             10
    goto fail;                                                            11
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)             12
    goto fail;                                                            13
    goto fail;                                                            14
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)                   15
    goto fail;                                                            16
err = sslRawVerify(...);                                                  17
//...                                                                     18
```

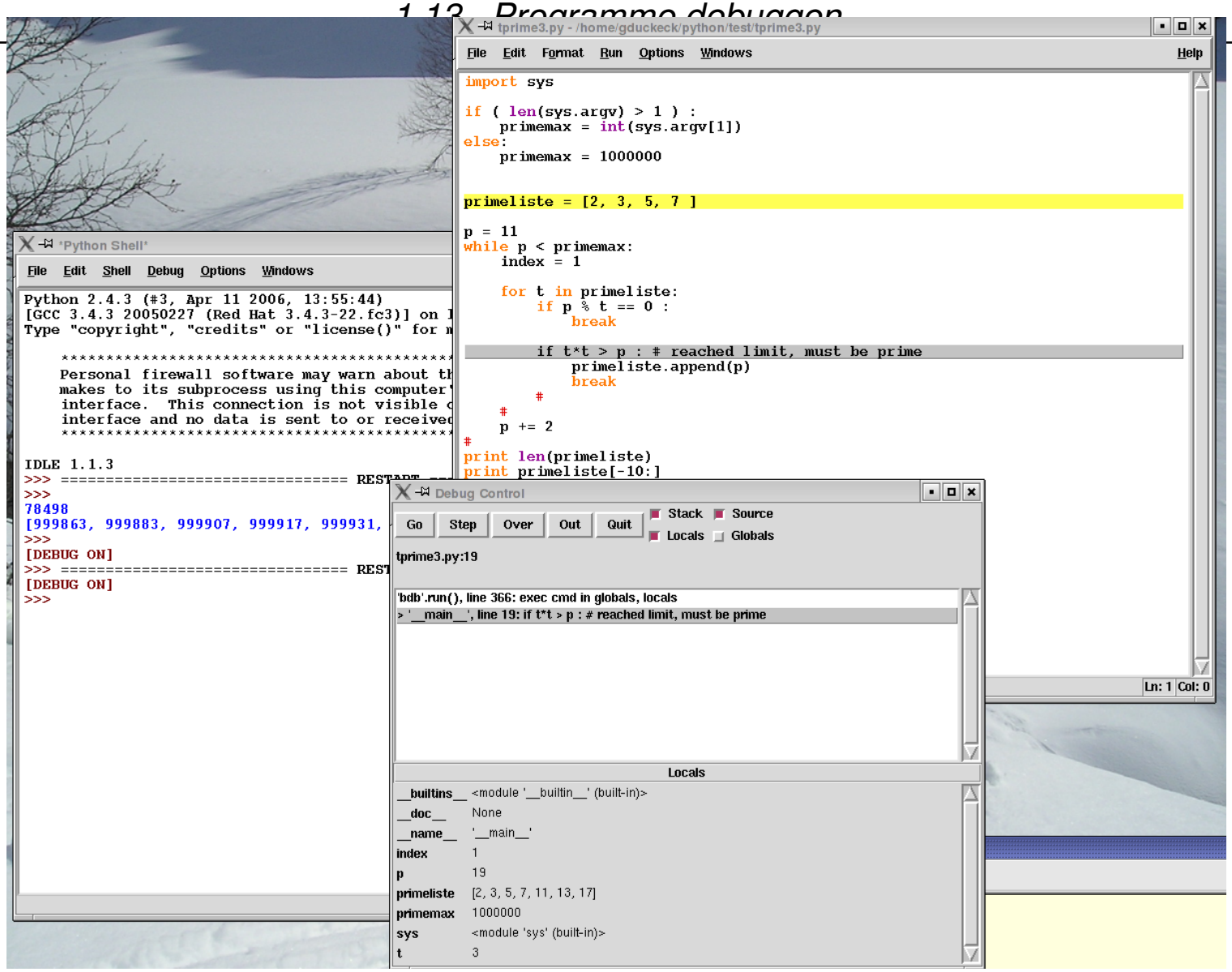
Entscheidender Check (`sslRawVerify()`) wird u.U. nicht ausgeführt, ermöglicht Angriffe auf verschlüsselte (TLS/SSL) Verbindungen.

Debugger

Meiste Zeit bei Programmentwicklung i.d.R. für (Laufzeit-/Semantik-) Fehlersuche.
In Python oft interaktives Ausprobieren möglich zur schnellen Fehlersuche, ansonsten ist Standardverfahren `print` statements an den kritischen Stellen.
$\Rightarrow$ umständlich, zeitraubend

Wesentlich eleganter mit richtigem *debugger*

- Programm läuft unter Kontrolle des Debuggers
- Zeile für Zeile dem Source-code nach
- oder *breakpoints* an den kritischen Stellen setzen und direkt dahin laufen.
- Inhalt von Variablen kann angezeigt werden. Bewirkt dass zusätzliche Info in den kompilierten code geschrieben wird.
- Klassischer Kommandozeilen-Debugger:
  Starten mit Module pdb: `python -m pdb tprime.py`
- Oder Integrierter Debugger mit GUI, z.B. in Python Entwicklungsumgebung **idle** : `idle tprime.py`


tprime3.py - /home/gduckeck/python/test/tprime3.py
File Edit Format Run Options Windows Help
import sys
if (len(sys.argv) > 1) :
primemax = int(sys.argv[1])
else:
primemax = 1000000
primeliste = [2, 3, 5, 7]
p = 11
while p < primemax:
index = 1
for t in primeliste:
if p % t == 0 :
break
if t*t > p : # reached limit, must be prime
primeliste.append(p)
break
#
#
p += 2
#
print len(primeliste)
print primeliste[-10:]
Ln: 1 Col: 0
Python Shell
File Edit Shell Debug Options Windows
Python 2.4.3 (#3, Apr 11 2006, 13:55:44)
[GCC 3.4.3 20050227 (Red Hat 3.4.3-22.fc3)] on
Type "copyright", "credits" or "license()" for
Personal firewall software may warn about th
makes to its subprocess using this computer
interface. This connection is not visible
interface and no data is sent to or received
IDLE 1.1.3
>>> ========== RESTART
>>>
78498
[999863, 999883, 999907, 999917, 999931,
>>>
[DEBUG ON]
>>> ========== REST
[DEBUG ON]
>>>
Debug Control
Go Step Over Out Quit
Stack Source
Locals Globals
tprime3.py:19
'bdb'.run(), line 366: exec cmd in globals, locals
> '__main__', line 19: if t*t > p : # reached limit, must be prime
Locals
__builtins__ <module '__builtin__' (built-in)>
__doc__ None
__name__ '__main__'
index 1
p 19
primeliste [2, 3, 5, 7, 11, 13, 17]
primemax 1000000
sys <module 'sys' (built-in)>
t 3

1.14 Python Versionen

Zwei unterschiedliche Python Versionen *(eigentlich Branches oder Zweige)* zur Zeit gebräuchlich:

- **Python2**, immer noch default (`python` in Kommandozeile), **Basis für diesen Kurs**, aktuelle Version im CIP: **2.7.6**
- **Python3**, (`python3` in Kommandozeile) immer noch default, aktuelle Version im CIP: **3.4.3**.

Im Prinzip Python3 als Nachfolger gedacht, aber zur Zeit werden beide Zweige verwendet und weiterentwickelt.

Keine grossen Unterschiede in Syntax und Verwendung, aber etliche, nicklige Details, die die Portierung existierender Python Module erschweren.
Mehr Info z.B. in http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

1.15 Python Grundlagen Zusammenfassung

Soweit Schnelldurchgang

- Syntax
- Basic Datentypen
- Operationen, Kontrollstrukturen

Zwangsläufig unvollständig und oberflächlich

Aber keine Panik, auch wenn vieles unklar ist

- Grundlegende Sprachelemente tauchen immer wieder auf
- Man braucht Zeit zum Verdauen und eigene Erfahrungen zu sammeln

$\Rightarrow$ fragen, üben, fragen, üben, fragen, üben, ...

1.16 Aufgaben

Es sind mit Absicht viele, es reicht wenn Sie etwa die Hälfte (gründlich) bearbeiten. Der Rest ist für zu Hause, als Anregung gedacht.

Grün markierte sind das Minimalprogramm, die sollten auf jeden Fall bearbeitet werden. **Rot** gekennzeichnete sind anspruchsvoller und v.a. für diejenigen gedacht, die schon Programmierkenntnisse mitbringen.

1. **Warmlaufen**
   - Benutzen Sie Python interaktiv, geben Sie Zahlen und Strings aus, verwenden Sie Python als Taschenrechner.
   - Erstellen Sie das ”Hello world” Programm im Editor, dann mit dem Python Interpreter ausführen.
   - Dasselbe mit der *Fahrenheit/Celsius* Konversion
     modifizieren Sie es: Schrittweite, Bereich, umdrehen in *Celsius⇒Fahrenheit*.

2. **Quadrat– und Kubik–Zahlen**
   (a) Erstellen Sie ein Programm, das die Quadrat– und Kubik–Zahlen von 1 bis 150 ausgibt und am Ende die Summe der Quadrat– bzw. Kubik–Zahlen

   (b) Demonstrieren Sie folgende mathematische Identität für $n = 1..200$:

$$\sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2$$

3. **Integer vs Strings**

   Führen Sie die folgenden Python Anweisungen aus und erklären Sie die Ausgabe bzw. die Unterschiede:

```
a = 1
print a
print a+a
b = '1'
print b
print b+b
```

4. **Fließkommazahlen**

   (a) Lassen Sie das Programm `TestFloat.py` laufen.
   Warum "können Rechner nicht rechnen" ?

   (b) Genauigkeit von `float` Operationen: Reduzieren Sie schrittweise

```
eps = 1.
```

```
while (...):
    eps /= 2.
```

addieren Sie's zu

```
    onePlusEps = 1.0 + eps
```

solange bis

```
    if ( onePlusEps == 1.0 ) ...
```

5. **Fibonacci–Zahlen**

   Fibonacci–Zahlen spielen eine wichtige Rolle in der Zahlentheorie und haben viele interessante Eigenschaften. Sie sind definiert als:

$$F_n = F_{n-1} + F_{n-2};\ F_0 = 0,\ F_1 = 1$$

   (a) Erstellen Sie eine Liste der Fibonacci–Zahlen.

   (b) Demonstrieren Sie dass gilt:

$$F_{n+1} \cdot F_{n-1} - F_n^2 = (-1)^n$$

6. **Bit Operationen**

   Programmieren Sie die binäre Ausgabe von Integerzahlen mit Hilfe von Bit Operationen.

   `z.B.: 5 -> 101`

   Hinweis: Bit `15` in `n` abfragen mit z.B.: `if ( ( ( 1 <<15) & n ) != 0 )`

7. **Prim–Zahlen**

   (a) Erstellen Sie ein Programm, das testet ob eine gegebene Zahl eine Primzahl ist.

   (b) Erweitern Sie das Programm, so dass es abzählt wieviele Primzahlen es gibt, die kleiner als eine vorgegebene Zahl sind, z.B. $1\,000\,000$.

   Hinweis: Es geht nicht darum einen schnellen Algorithmus zu finden, machen Sie's so simpel wie möglich.

   (c) Sieb des Erasthones ist ein klassisches Verfahren zur Bestimmung aller Primzahlen zwischen 2 und n. Der Algorithmus geht folgendermassen:

   - Erstelle Liste aller Zahlen von 2 bis n
   - Nimm schrittweise jede Zahl i dieser Liste, falls sie nicht gestrichen ist.
   - Streiche alle Vielfachen von i aus der Liste

   Am Ende bleiben genau die Primzahlen übrig in der Liste. *(Tipp: Test auf Vielfaches mit modulo Operator:* `m%i == 0` *)*

8. **Lineare Algebra**

(a) Vektoroperationen: Legen Sie 2 Arrays mit den Elementen `[ 0.3, 1.8, -2.2 ]` bzw `[ -2.5, 3.8, 0.4]` an und berechnen Sie das Skalarprodukt und das Vektorprodukt.

(b) Matrixmultiplikation: Schreiben Sie ein Programm zur Multiplikation dieser beiden Matrizen:

```
C = [ [ 0.61, 0.24, 1.16 ], [ 0.14, -0.82, 0.92 ], [ -1.25, 0.96, -0.23 ] ]
D = [ [ 0.40, -0.68, -0.68 ], [ 0.65, -0.75, 0.23 ], [ 0.52, 0.51, 0.31 ] ]
```

9. **Freier Fall**

   Die Bewegungsgleichunen für den freien Fall sind (Erdbeschleunigung $g = 9.81 m/s^2$):

$$y = h_0 - \frac{1}{2} g t^2 , \quad v = v_0 - gt$$

   Berechnen Sie y und v für den freien Fall eines Balles mit $h_0 = 10\, m$ und $v_0 = 0$ für 30 Zeitpunkte von $t = 0..1.5\, s$.

10. **Programmlogik bei Schleifen**

    Folgendes Beispiel ist ein funktionierendes Python Programm (übernommen aus Learning Python). In einer Liste soll nach einem bestimmten Wert gesucht werden. Allerdings ist der Stil eher C++/JAVA.

```
L = [1, 2, 4, 8, 16, 32, 64]
```

1

```
X = 5                                                    2
found = i = 0                                            3
while not found and i < len(L):                          4
    if 2 ** X == L[i]:                                   5
        found = 1                                        6
    else:                                                7
        i = i+1                                          8
if found:                                                9
    print 'at index ', i                                10
else:                                                    11
        print X, ' not found'                            12
```

In Python lässt es sich eleganter lösen, implementieren Sie verschiedene Varianten:

(a) `found` und `if` am Ende sind überflüssig wenn man stattdessen eine `while: ... else:` Kombination macht.

(b) `for` Loop statt `while`, ohne *indexing* logic, stattdessen mit `L.index(X)` Position abfragen.

(c) `in` Operator alleine: `if X in L: ...`

11. **Funktion für Fakultät**

Schreiben Sie eine Funktion `fak( n)`, zur Berechnung der Fakultät $n!$. Bis zu welchem n lässt

sich $n!$ für *float* Typen berechnen ?

12. **Rekursive Funktion**
    Ein Paradebeispiel für Rekursion ist Euklid's Algorithmus zur Bestimmung des *Größten Gemeinsamen Teilers* zweier Zahlen $GGT(a, b)$.

$$GGT(a,b) = \begin{cases} GGT(b, a \bmod b) & \text{wenn } a \text{ nicht durch } b \text{ teilbar ist} \\ b & \text{sonst} \end{cases}$$

    Erstellen Sie eine solche Funktion in Python.

13. **Mathematische Funktionen**
    In dem Python Module *math* (siehe Python math oder `help(math)` im Python Interpreter) sind die Standard mathematischen Funktionen dokumentiert. Zur Benutzung in ihrem Programm zunächst das Modul laden:
    `import math` und dann den Module-Namen *math.* voranstellen, also z.B.

```
mypi = 4.  * math.atan(1.)
```

    Machen Sie sich mit den Funktionen vertraut, z.B.

    - Was ist das Ergebnis von `math.sqrt(144), math.sqrt(-1.), math.log(10), math.atan(1.)  - math.pI/4., ...`
    - Zum Rechnen mit komplexen Zahlen gibt es das entsprechende Python Module *cmath*. Testen Sie diese Funktionen.

14. **Funktionen als Argumente und lambda functions**

    (a) Implementieren Sie das vorgestellte Beispiel mit der Nullstellen-Suche, benutzen Sie andere Funktionen,

    $$e^x - x^{10}$$

    , ...

    (b) Machen Sie analog eine Funktion zur Integration mit Trapez– oder Simpson–Regel und frei wählbarer Zahl von Stützstellen.

15. **Einlesen und Arrays**

    In der Datei `numbers.dat` finden Sie eine Liste mit 100 Fließkommazahlen. Lesen Sie diese ein und

    (a) finden den kleinsten und größten Wert.

    (b) erlauben Sie die Übergabe einer Zahl per Argument und finden Sie die Zahl aus `numbers.dat`, die am nächsten liegt.

    (c) speichern Sie alle Zahlen in eine Liste von float Zahlen. Sortieren Sie die Liste in absteigender Reihenfolge und geben Sie sie aus.

16. **Files kopieren**

    Schreiben Sie ein Programm zum kopieren von Files, z.B.

    ```
    python FileCopy.py file1.txt file2.txt
    ```

soll file1.txt nach file2.txt kopieren.

17. **Zeilen, Wörter, Zeichen zählen**
    Schreiben Sie ein Programm, das die Zahl der Zeilen, Wörter und Zeichen bestimmt für ein File das auf der Kommandozeile angegeben wird, z.B.
    `python count.py /etc/passwd`

18. **String Funktionen**
    Für Python Strings sind eine Reihe nützlicher Hilfsfunktionen definiert, siehe https://docs.python.org/release/2.7.6/library/stdtypes.html#string-methods.

    In `kant.txt` finden Sie eine elektronische Fassung von Immanuel Kant's *"Kritik der reinen Vernunft"*.

    (a) Wieviele Zeilen enthält der Text ?

    (b) Finden Sie die String Funktion die Groß-Buchstaben in Klein-Buchstaben umwandelt und transformieren sie damit den ganzen Text.

    (c) Wie oft kommt das Wort *Vernunft* in dem Text vor ?

19. **Strings und Sequenzen**
    Studieren Sie folgende sechs Code-Segmente, d.h. überlegen Sie zunächst welches Ergebnis Sie jeweils erwarten, probieren es dann aus und versuchen es zu verstehen falls die Erwartung nicht eingetroffen ist.

```
#
"aaaaa".count("aaa")
#
#
x = ['a', 'b', 'c', 'd']
x[0:2] = []
#
x = ['a', 'b', 'c', 'd']
x[0:2] = ['q']
#
x = ['a', 'b', 'c', 'd']
x[0:2] = 'q'
#
x = ['a', 'b', 'c', 'd']
x[0:2] = 99
#
x = ['a', 'b', 'c', 'd']
x[0:2] = [99]
    #
```

2 Klassen und Objekte

- Warum objektorientiertes Programmieren ?
- Ein einfaches Beispiel – 3D Vektor
- Klassen und Objekte, Methoden und Variablen
- Vererbung
- Abstrakte Methoden und Klassen
- Interfaces
- Aufgaben

2.1 Warum objektorientiertes Programmieren ?

Intrinsische Datentypen (`int, float, string, ...`) sind unzureichend für die allermeisten praktischen Probleme.

Offensichtlich im Bereich Verwaltung, Wirtschaft, Handel, ...

- Studenten–Daten an der LMU
- Fahrscheine buchen bei DB
- Ebay Auktionen, ...

Praktisch immer komplexe *Datenmodelle*, d.h. zusammengesetzt aus Strings, int und float Zahlen, Querverweise auf weitere Infos, usw.

Aber auch im naturwissenschaftlich/technischen Umfeld – mit überwiegend numerischen Informationen – sind Messdaten keine isolierten Zahlenkolonnen sondern i.d.R. komplex strukturiert und ergeben nur im Kontext mit dem Experiment *(Aufbau, Parameter, Außenbedingungen)* einen Sinn.

- Spuren im Detektor sind 3er oder 4er Vektoren, bestehen aus Hits, gehören zu einem Sub–Detektor, ...
- Fluoreszenz–Spektrum: Wertepaare *(Frequenz, Intensität)* plus Zusatzinfo zu Probe, Apparatur, Kalibration, äussere Parameter, ...
- ...

Im Prinzip mit **Python list** möglich Problem zu lösen, da beliebige andere Elemente in einer Python list abgelegt werden können, beliebige Datenstrukturen sind möglich. Aber OO programmieren geht weiter:

Verknüpfung von Daten und Methoden

Historische Entwicklung:

Unstructered Programming: Ein Hauptprogamm + Daten

Procedural Programming: Ein Hauptprogamm + globale Daten + kleinere Funktionen (prozedures)

Modular Programming: Ein Hauptprogamm + globale Daten + Module mit lokalen Daten und Unterprozeduren

Object-oriented: Daten und Prozeduren integriert in Klassen, d.h. *direkte Kopplung von Daten und Prozeduren*. Keine globalen Daten, kein eigentliches Hauptprogramm, Objekte kommunizieren direkt

Im Prinzip natürlicher & intuitiver Ansatz:
⇒ Alltag **Datum** & Methode verknüpft

Auto	..., Fahren, Schalten, Blinken, ...
Wiesn	..., Maß, Schunkeln, Brechen, ...
Fußball	..., Blutgrätsche, Schwalbe, ...

2.2 Von primitiven Datentypen zu komplexen Datenstrukturen

Standard Datentypen alleine ungeschickt bzw. unzureichend für praktische Anwendungen, z.B. Studentendaten: *Name, Studiengang, Alter, Semester, Matrikel-Nummer, Noten, ...* Kombination aus `string, int, float` Daten.

In Python mittels einfachster Form von Klasse möglich solche Datenstrukturen selbst zu definieren:

```
class Stud: # dumb class                                    1
    pass                                                    2
sta = Stud() # Erzeugung Variable vom typ Stud              3
sta.name = "Albert Unirock"                                 4
sta.fach = "Physik"                                         5
sta.alter = 25                                              6
...                                                         7
stb = Stud() # Erzeugung Variable vom typ Stud              8
stb.name = "Berta Bohne"                                    9
stb.fach = "Informatik"                                     10
stb.alter = 19                                              11
...                                                         12
```

- Leere Klasse **Stud**
- Erzeugen eines **Objektes** einer Klasse und Zuweisung an entsprechende Variable:
  `sa = Stud()`
- Beispielklasse hier zunächst völlig leer und ohne weitere Funktionalität, kann aber als Hülle/Container für beliebige Variablen dienen
- Erzeugen/Zugriff auf Variablen des Objektes mittels **objectname.variable**: `sa.alter = 21;`

Weiteres Beispiel: Klasse für einfachen Dreier-Vektor

```
class Dumb3Vec: # weitere leere Klasse, nur Huelle fuer 3-er Vektoren
    pass # empty statement, hier noetig
# Anwendung
v = Dumb3Vec() # Erzeugung Variable vom typ Dumb3Vec
w = Dumb3Vec() # Erzeugung Variable vom typ Dumb3Vec
v.x = 1.0
v.y = 0.5
v.z = -0.8
w.x = 1.5
w.y = -0.5
w.z = -2.8
...
// Laenge ausrechnen
len = math.sqrt(v.x*v.x + v.y*v.y + v.z*v.z )
u = Dumb3Vec()
# v und w addieren
u.x = v.x + w.x
u.y = v.y + w.y
```

```
u.z = v.z + w.z                                                    19
...                                                                20
```

Anstatt gängige Operationen jedesmal wieder neu zu programmieren wäre es geschickt, wenn dies gleich die Klasse übernehmen könnte, d.h. Klassen nicht nur zum Definieren beliebiger Datenstrukturen sondern auch gleich Operationen bzw. Methoden mit diesen Daten

---

```
import math                                                         1
class Smart3Vec: # Klasse fuer 3-er Vektoren mit Methoden           2
    x = 0  # default values                                         3
    y = 0                                                           4
    z = 0                                                           5
    def Length( self ):                                             6
        return(math.sqrt(self.x*self.x + self.y*self.y + self.z*self.z ))  7
    def Add( self,    a ):                                          8
        t = Smart3Vec()                                             9
        t.x = self.x + a.x                                          10
        t.y = self.y + a.y                                          11
```

```
            t.z = self.z + a.z                                    12
        return(t)                                                 13
    # Anwendung                                                   14
v = Smart3Vec()  # Erzeugung Variable vom typ Smart3Vec           15
w = Smart3Vec()  # Erzeugung Variable vom typ Smart3Vec           16
v.x = 1.0                                                         17
v.y = 0.5                                                         18
v.z = -0.8                                                        19
w.x = 1.5                                                         20
w.y = -0.5                                                        21
w.z = -2.8                                                        22
# v kann seine Laenge selbst ausrechnen ...                       23
print v.Length()                                                  24
# ... und weiss auch wie es einen anderen Vektor addiert          25
u = v.Add(w)                                                      26
print u.Length()                                                  27
```

⇒ **Grundkonzept für Objektorientiertes Programmieren**

2.3 Eine richtige Klasse für 3D Vektor

```
#                                                                    1
import math                                                          2
#                                                                    3
class ThreeVec(object):                                              4
    "Class for 3 Vector and operations"                              5
    def __init__( self, x=0., y=0., z=0.):                           6
        self.x = x                                                   7
        self.y = y                                                   8
        self.z = z                                                   9
    def Add( self, tv ): # add two 3 vecs                            10
        newvec = ThreeVec( self.x+tv.x, self.y+tv.y, self.z+tv.z)    11
        return newvec                                                12
    def Length( self ): #                                            13
        return math.sqrt( self.x**2 + self.y**2 + self.z**2 )        14
    def Scale( self, a ): # Skalierung mit float                     15
        pass                                                         16
#         ...                                                        17
    def Angle( self, a ): # Winkel zwischen 2 Threevectors           18
```

```
        pass                                                                    19
#       ...                                                                     20
    def ScalProd( self, tv ): # Skalarprodukt von 2 Threevectors                21
        pass                                                                    22
#       ...                                                                     23
    def CrossProd( self, tv ): # Kreuzprodukt von 2 Threevectors                24
        pass                                                                    25
#       ...                                                                     26
#                                                                               27
#                                                                               28
# use ThreeVec                                                                  29
#                                                                               30
tv1 = ThreeVec( 1., 2., 0.)      # create ThreeVec object ( calls ThreeVec.__init__ ) 31
tv2 = ThreeVec( 1., -1., 1.)     # create ThreeVec object                       32
tv3 = ThreeVec( 3, -2, 4)        # create ThreeVec object                       33
tv4 = tv3.Add( tv1 )             # Add two ThreeVecs                            34
print tv1.Length()               #                                              35
print tv3.Length()                                                              36
print tv4.Length()                                                              37
```

- Eine Klasse definiert einen **neuen Typ**
- Typ enthält massgeschneiderte Daten und v.a. Methoden
- Die **Klasse** ist zunächst eine abstrakte Definition eines Datentyps mit zugehörigen Methoden
- Ein **Objekt** wird daraus wenn eine *Instanz* der Klasse erzeugt wird:
  ```
  v = ThreeVec( 1., 0.5, 2.)
  ```
- Bei Erzeugung wird implizit spezielle Methode gerufen mit Namen `__init__(...)`, diese dient zur Initialisierung des Objekts.
  Nicht zwingend nötig, aber i.d.R. praktisch zur Initialisierung. Entspricht in etwa der *Konstruktor–Methode* in C++/JAVA.
- Anschliessend können Methoden für die erzeugten Objekte gerufen werden.
  ```
  tv4 = tv3.Add( tv1 )
  tv1.Length()
  ```

2.4 Daten und Methoden in Klassen

Variablen und **Funktionen**, die **innerhalb** einer Klasse definiert sind, sind die sogenannten **member-variables** bzw. **member-functions**.

Bei der Verwendung, d.h. **ausserhalb der Klasse** werden sie mittels
objectname.variable oder objectname.function() angesprochen, z.B.

```
u = ThreeVec(1.,2.,-1.)
v = ThreeVec(0.,2.,1.)
u.x = 7
w = u.Add(v)
```

Bei der Definition der Member-functions selbst, d.h. **innerhalb der Klasse**, werden die member-variables bzw. andere member-functions über **self** angesprochen.

- **self** ist Schlüsselwort und bezieht sich auf das jeweilige Objekt für das die Methode gerufen wird.

- **self** muss immer als erster Parameter der member-functions angegeben werden, aber **nicht** als Argument beim Aufruf, das macht Python implizit. Aufruf:

```
v = ThreeVec(0.,2.,1.)
v.Length()
```

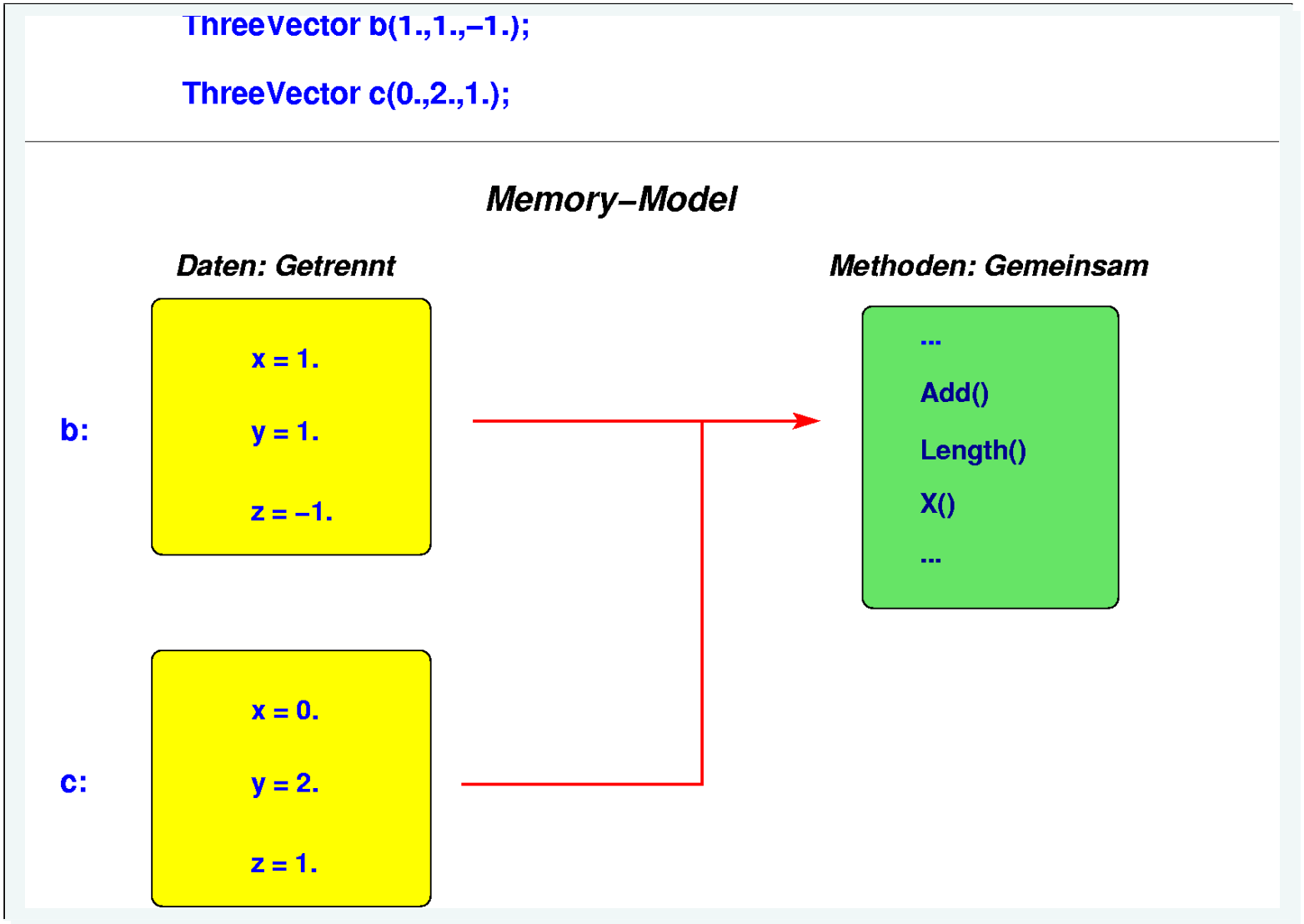
- Innerhalb der member-function bezieht sich **self** jetzt auf den **ThreeVec v**, d.h. `v.x == self.x, v.y == self.y, ...`

```
class ThreeVec(object):                                          1
    ...                                                          2
    def Length( self ):                                          3
        return math.sqrt( self.x**2 + self.y**2 + self.z**2 )    4
```

- **self** spielt ähnliche Rolle wie **this** in C++/JAVA

Besondere Bedeutung hat **__init__ (...)** Methode

- Dient zum Anlegen des Objekts, i.d.R. sollten alle verwendeten Member-Variablen hier erzeugt werden. (Aber kein Zwang in Python, Definition/Zugriff kann von überall erfolgen, kein *Verstecken* der Daten möglich wie in C++/JAVA mit *private*.)
- Wird implizit gerufen bei Erzeugen eines Objektes der Klasse, d.h.
  `v = ThreeVec( 1., 0.5, 2.)` entspricht in etwa
  `v = ThreeVec(); v.__init__( 1., 0.5, 2.)`


ThreeVector b(1.,1.,–1.);
ThreeVector c(0.,2.,1.);
Memory–Model
Daten: Getrennt
Methoden: Gemeinsam
b:
x = 1.
y = 1.
z = –1.
c:
x = 0.
y = 2.
z = 1.
...
Add()
Length()
X()
...

Vergleich prozedural vs objektorientiert

Mit den **ThreeVecs** soweit jetzt diskutiert kann man sehr einfach und effizient Code für Dreier-Vektor Manipulationen schreiben, z.B. Vektoren addieren, mit Skalar multiplizieren, ...

```
u = ThreeVec(1.,1.,-1.)
v = ThreeVec(2.,0.,3.)
w = u.Add( v.Scale(3.)  )
```

Wie würde man es ohne Klassen/Objekte lösen ?

Variante 1: Direkt kodieren:

```
a = [ 1, 1, -1 ];
b = [ 2, 0, 3 ];
c = [0]*3
for i in range(3):
    c[i] = a[i] + 3. * b[i]
#
```

Variante 2: Entsprechende globale Python Funktionen definieren/verwenden:

```
def ScaleVector( vin, scale ):
    "Multiply vector elemtns by scalar"
    vout = [0]*len(vin)
    for i in range( len(vin) ) :
        vout[i] = scale*vin[i]
    return vout
#
def AddVector( v1, double v2 ):
    "Add two vectors"
    vout = [0]*len(v1)
    for i in range( len(v1) ) :
        vout[i] = v1[i] + v2[i]
    return vout
#
a = [ 1, 1, -1 ];
b = [ 2, 0, 3 ];
c = ScaleVector( b, 3.)
c = AddVector( a, c )
```

Geht natürlich auch, aber unhandlich, kryptisch, fehleranfällig, ...

2.5 Dynamische Speicherverwaltung

In modernen Sprachen, wie Python und C++/JAVA, kann ein Programm jederzeit eine im Prinzip beliebig grosse Menge Speicher für Objekte oder Arrays anfordern, mittels z.B.

```
a = [some-object]*length
```

Eine besonders schönes Feature von Python (und JAVA) ist, daß das schon alles ist; das aufräumen passiert automatisch.

In C/C++ dagegen muss sich die Programmiererin selbst darum kümmern den allozierten Speicher wieder zu räumen, eine sehr heikle und fehleranfällige Aufgabe.

```
for ( i = 0; i < 1000000; i++ ) { // C++ killer, fine in JAVA
    double [] space = new double[1000000];
    ...
    // delete [] space;     // C++ memory leak
}
...
// oder
int [] room = new int[100];
int [] room2 = room;
```

```
    ...                                                                    10
    delete [] room;    // C++ room2 still defined, dangling ref           11
}                                                                          12
```

Python merkt sich alle Referenzen auf die angelegten Objekte bzw. den allozierten Speicherbereich. Sobald keine mehr vorhanden sind wird der Bereich zum Abschuss für den *garbage-collector* freigegeben. Dieser wird bei Bedarf automatisch gerufen. In der Regel kein Eingriff des Programmierers nötig.

2.6 Scope und Lifetime

'Klassen'–Variablen, d.h. Variablen, die mit einer Klasse definiert sind

- werden initialisiert wenn das Objekt angelegt wird
- existieren solange das Objekt existiert
- sind überall verfügbar, wo das Objekt verfügbar ist.

Lokale Variable, d.h. Variablen die innerhalb einer Funktion oder Methode definiert sind

- **Scope** (Gültigkeitsbereich) auf die Funktion beschränkt, unbekannt ausserhalb
- **Lebensdauer** ditto, während Programm in Funktion

Python–Objekte existieren solange irgendwo eine Referenzvariable dafür existiert.

2.7 Operator overloading

In Python können Standardoperationen, wie +, *, etc für beliebige Klassen definiert werden. Dazu muss nur eine Methode mit Namen **__add__(...)** in der Klasse definiert sein:

```
class ThreeVec(object):                                                    1
    ...                                                                    2
    def Add( self, tv ): # Add function                                    3
        newvec = ThreeVec( self.x+tv.x, self.y+tv.y, self.z+tv.z)          4
        return newvec                                                      5
    def __add__( self, tv ): # dasselbe aber fuer + Operator               6
        "method for overloading the + operator"                            7
        newvec = ThreeVec( self.x+tv.x, self.y+tv.y, self.z+tv.z)          8
        return newvec                                                      9
```

und schon kann man statt

```
u = ThreeVec(1.,1.,-1.)
v = ThreeVec(2.,0.,3.)
w = u.Add( v )
```

einfach schreiben

```
z = u + v # equivalent, calls u.__add__(v)
```

Operator overloading prinzipiell möglich für alle Python–Operationen

```
* :  __mul__ , - :  __sub__ , / :  __div__ ,
< :  __lt__ , > :  __gt__ , == :  __eq__ , .....
```

aber nicht immer sinnvoll:

```
z = u * v
```

Multiplikation von Dreier-Vektoren ist Skalar– oder Vektorprodukt ??

Besser nicht implementieren in diesem Fall und stattdessen aussagekräftige Funktionsnamen verwenden, `u.SkalProd(v); u.CrossProd(v)`

2.8 Klassen erweitern – Vererbung

In der Physik sind häufig Vierer–Vektoren *(=Lorentz–Vektor)* gefragt, d.h. Dreier–Vektoren erweitert um Zeit bzw. Energie–Komponente. Mögliche LorentzVektor Klasse:

```
class LorentzVec(object):                                                    1
    "Class for 4 Vector and operations"                                      2
    def __init__( self, t=0., x=0., y=0., z=0.):                            3
        self.x = x                                                           4
        self.y = y                                                           5
        self.z = z                                                           6
        self.t = t                                                           7
    def Add( self, lv ):                                                     8
        newvec = LorentzVec( self.t+lv.t, self.x+lv.x, self.y+lv.y, self.z+lv.z)  9
        return newvec                                                        10
    def Angle( self, lv ):                                                   11
        ...                                                                  12
    def Mass( self ):                                                        13
        ...                                                                  14
```

Viele Gemeinsamkeiten mit *ThreeVec* und ein paar Erweiterungen

- 3 alte, 1 neues Datenelement
- einige Methoden identisch *Angle(), Theta(), ...*
- einige komplett neu (Masse zweier 4-Vectors)
- einige müssen neu implementiert werden (Add, ...)

Design Prinzip Code-Reuse statt cut&paste !

⇒ LorentzVector enthält ThreeVector: **"has-a"** relationship *(Aggregation)*.

```
class LorentzVec(object):                                              1
    "Class for 4 Vector and operations"                                2
    def __init__( self, t=0., x=0., y=0., z=0.):                       3
        self.v3 = ThreeVector( x, y, z )      # contains ThreeVector   4
        self.t = t                                                     5
    def Add( self, tv ):                                               6
        w = self.v3 + tv.v3 # add ThreeVectors first                   7
        newvec = LorentzVec( self.t+tv.t,      w.x, w.y, w.z)          8
        return newvec                                                  9
    def Angle( self, tv ):                                             10
```

```
        return( self.v3.Angle( tv.v3) # use ThreeVector Method                    11
    def Mass( self ):                                                             12
        ...                                                                       13
```

Im Prinzip ok, allerdings viele stumpfsinnige *Mapping–Funktionen* von DreierVektor auf ViererVektor nötig.

3. Möglichkeit: **Vererbung**

ViererVektor **ist** *DreierVektor* mit ein paar Ergänzungen ....

```
class LorentzVec(ThreeVec): # inherits from ThreeVec                    1
    "Class for 4 Vector and operations"                                 2
    def __init__( self, t=0., x=0., y=0., z=0.):                       3
        ThreeVec.__init__( self, x, y, z )      # initialize  ThreeVec  4
        self.t = t                                                      5
    def Add( self, tv ):                                                6
        w = ThreeVec.Add( self, tv ) # call ThreeVec method first       7
        newvec = LorentzVec( self.t+tv.t, w.x, w.y, w.z)                8
        return newvec                                                   9
#   def Angle( self, tv ): no need to define here, available from ThreeVec 10
    def Mass( self ):                                                   11
        ...                                                             12
```

Jetzt übernimmt bzw. **erbt** *LorentzVector* alle Funktionen von *ThreeVec*, z.B. Winkelberechnung funktioniert sofort:

```
c = LorentzVec(1.000001,1.,0.,0.)
```

```
d = LorentzVec(2.,1.,1.,0.)
c.Angle(d)
```

- Für die *abgeleitete* Klasse (LorentzVec) sind Methoden und Variablen, die in der *Basisklasse* (ThreeVec) definiert sind, direkt verfügbar, z.B. `ThreeVec.Angle(..)`. Sie müssen nicht nochmals extra angegeben werden.
- Die *abgeleitete* Klasse **kann** Methoden, die in der *Basisklasse* schon definiert sind überschreiben, d.h. neu implementieren, z.B. `LorentzVec.Add`. Bei Verwendung wird dann automatisch die für LorentzVec definierte Klasse genommen.
- Memberfunktionen der abgeleiteten Klasse können explizit auf Funktionen der Basisklasse zugreifen: `Classname.method( self, ...)` , z.B. in `LorentzVec.Add` kann `ThreeVec.Add( self, ...)` gerufen werden.
- Abgeleitete Klasse kann beliebige weitere Methoden und Variablen einführen.

Vererbung (**"is-a"** relationship) bedeutet alle Eigenschaften und Funktionen einer Grund-Klasse (base class) in eine weitere Klasse (derived class) zu übernehmen. Daraus ergibt sich eine enorme Erweiterung der Funktionalität. Ausgehend von vorhandenen, simplen Grundklassen können relativ leicht neue Klassen abgeleitet werden, ohne jedesmal diesselben grundlegenden Funktionen neu zu implementieren.

Allerdings: Vererbung nicht übertreiben, nicht immer sinnvoll, im Zweifelsfall Aggregation verwenden.

Rechteck und Quadrat

Zunächst naheliegend *Quadrat* von *Rechteck* abzuleiten (=vererben), viele Funktionen und Eigenschaften gemeinsam. Aber grundsätzliche Probleme bei Verhalten, z.B.:

```
class Rechteck:                                          1
    def __init__( self, w, l ):                         2
        self.w = w                                       3
        self.l = l                                       4
  // ...                                                 5
  def setLength( self, len):                             6
      ...                                                7
  def setWidth( self, wid):                              8
      ...                                                9
};                                                       10
```

Was soll ein *Quadrat* mit diesen Funktionen machen ?

`setLength(..)` bei *Quadrat* ändert immer auch `width` und vice-versa. **Verhalten** nicht kompatibel mit *Rechteck* .

Wichtig für OO Programmieren ist **konsistentes Verhalten** von Klassen, dazu wird Vererbung eingesetzt. Es geht weniger um *bequemes* Übernehmen von Funktionalität.

Mathematisch ist Quadrat Unterklasse von Rechteck, aber nicht im Sinne von **OOP** !

2.9 Abstrakte Klassen und Polymorphismus

In C++/JAVA wichtige Konzepte für Objektorientiertes Programmieren:

- **Abstrakte Klassen** sind Basisklassen, die Methoden vorgeben, jedoch ohne sie zu implementieren. Damit wird erzwungen, dass alle Klassen die sich von so einer Klasse ableiten diese Methoden eigenständig implementieren müssen. In C++/JAVA gibt es dafür ein explizites Schlüsselwort **abstract**. In Python so nicht vorhanden.
- **Polymorphismus** hängt eng mit abstrakten Klassen zusammen. In C++/JAVA bedeutet es, dass Objekt–Methoden einer abgeleiteten Klasse über Referenzen auf die Basisklasse angesprochen werden können. Dabei genügt es wenn diese Methoden als abstrakte Methoden in der Basisklasse deklariert sind.

In Python ist Polymorphismus selbstverständliche Beigabe, aufgrund des **dynamic-typing** gibt es keine starren Datentypen, erst bei konkreter Verwendung eines Objektes zur Laufzeit wird nach der aufgerufenen Methode gesucht. Deshalb ist das Fehlen von expliziten abstrakten Klassen in Python weniger wichtig. Falls dennoch benötigt kann die Funktionalität auch mit einfachen Programmiertricks emuliert werden.

2.10 Ergänzung – Standard Methoden für Klassen

Für Klassen, die zur *Aufnahme von Daten* dienen, empfiehlt es sich Methoden zu implementieren, die Vergleiche und Ausgabe unterstützen.

Das sind:

- **__str__(self)** : Soll das Objekt als string darstellen. Wenn Methode vorhanden, wird sie implizit bei *print* verwendet, d.h.
  ```
  u = ThreeVec (1.,2.,0.)
  print u
  ```
- **__cmp__(self, other)** : Methode zum Vergleichen, soll `-1, 0, 1` zurückgeben wenn `self < other, self == other, self > other` ist. Methode wird z.B. von **list.sort()** gerufen, wenn also Objekte in einer Liste sortiert werden.

2.11 Aufgaben

1. Vektor-Klasse

   Entwerfen Sie ausgehend von dem Beispiel die `ThreeVec` Klasse.

   Führen Sie zusätzliche *sinnvolle* Methoden ein, z.B. Länge eines Vektors, Winkel zwischen zwei Vektoren, Skalarprodukt, Vektorprodukt, Ausgabe, Vergleich.

   Anschliessend sollte man diese Vektoren so benutzen können:

```
from ThreeVec import ThreeVec                                    1
u = ThreeVec( 1., 0.5, 2.)                                       2
v = ThreeVec( 1., 2.5, -1.)                                      3
w = u.Add( v)                                                    4
t = u.Subtract( v)                                               5
t = t.Scale( 2.)                                                 6
print "Vector w = ", w.x, w.y, w.z                               7
print "Vector t = ", t.x, t.y, t.z                               8
r = u.CrossProduct(v)                                            9
x = w.ScalarProduct(v)                                           10
print   w.Angle(u)                                               11
```

```
#                                                                    12
```

Und in späterer Variante mit *Operator Overloading* und `__str__()` Methode:

```
from ThreeVec import ThreeVec                                        1
u = ThreeVec( 1., 0.5, 2.)                                           2
v = ThreeVec( 1., 2.5, -1.)                                          3
w = u + v                                                            4
t = u - v                                                            5
t = t * 2.                                                           6
print "Vector w = ", w                                               7
print "Vector t = ", t                                               8
r = u.CrossProduct(v)                                                9
x = w.ScalarProduct(v)                                               10
angle = w.Angle(u)                                                   11
#                                                                    12
```

(Ausgearbeitetes Beispiel: `py`, `html`)

2. Statistik

   Die Klasse `StatCalc.py`. implementiert einige grundlegende Statistikfunktionen, wie Mittelwert, Standardabweichung, ...

   (a) Legen Sie ein Objekt an `stat = StatCalc()` und füllen in einer Schleife Zufallszahlen `stat.enter(random-number)` und bestimmen Mittelwert und Standardabweichung.

   (Zufallszahlen in Python über `import random`*. Funktion* `random.gauss(0.,1.)` *liefert normal-verteilte Zufallszahlen,* `random.random()` *gleich-verteilte zwischen 0 und 1).*

   (b) Erweitern Sie `StatCalc.py` um Methoden zur Ausgabe von Minimum und Maximum.

   (c) In `semester.dat` finden Sie die Semesterzahl bis zum Physikdiplom für zufällig ausgewählte Studenten. Die ersten 100 Einträge sind von Studenten der LMU, die restlichen 100 von Studenten der TUM. Lesen Sie die Daten ein und füllen LMU bzw TUM Zahlen jeweils in ein `StatCalc` Objekt. Sind die Mittelwerte im Rahmen der Schwankungen konsistent ?

   (d) Überlegen Sie wie man das Problem aus (c) (mehrere Statistiken parallel führen) in einer prozeduralen Sprache (Fortran, C) angehen könnte, d.h. ohne Klassen und Objekte, nur mit Arrays und Funktionen.

3. Vererbung

   Leiten Sie die Lorentz-Vektor Klasse (Vierer-Vektor $= \ (E, p_x, p_y, p_z) = (E, \vec{p})$) von der Dreier-Vektor Klasse ab.

Lorentz-Vektoren sollten als zusätzliche Methoden die Masse berechnen können:

$$M = \sqrt{E^2 - \vec{p}^2}$$

sowie die invariante Masse zweier Lorentzvektoren:

$$M_{inv} = \sqrt{(E_1 + E_2)^2 - (\vec{p_1} + \vec{p_2})^2}$$

Benutzung:

```
a = LorentzVector(45.0002,0.,0.,45.0)
b = LorentzVector(45.0002,31.8198,0.,31.8198)
print "Angle = " , a.Angle(b)
print "Mass a = " , a.Mass()
print "Mass b = " , b.Mass()
print "Mass a+b = " , b.InvMass(a)
...
```

(Ausgearbeitetes Beispiel: `py`, `html`)

4. Standard–Methoden

   Implementieren Sie die Standard Methoden für die ThreeVec–Klasse, d.h. **__str__(self)** und **__cmp__(self, other)**. Testen SIe es indem Sie eine größere Anzahl von ThreeVectors erzeugen, diese in eine **list** packen und anschliessend sortieren und ausgeben.

```
from ThreeVec import ThreeVec
import random # random numbers
tvc = [] # empty list
for i in range(100):
    # create random ThreeVec
    u = ThreeVec( random.gauss(0,1), random.gauss(0,1), random.gauss(0,1))
    tvc.append(u) # store in list
tvc.sort() # sort ThreeVec-list
for tv in tvc:
        print tv.Length(), tv # output length and Threevec (via __str__())
```

5. **Vererbung und Polymorphismus 1:**

   Übernommen von http://www.lrz-muenchen.de/ẽbner/C++/Uebung/aufgaben_004.html

   Das source file `Particles.py` (src, html) enthält ein Grundgerüst zu einer allgemeinen Basisklasse *Particle* und davon abgeleiteter Klassen.

   (a) Vervollständigen Sie die deklarierten `__init__` Funktionen jeder Klasse.

   (b) Erstellen Sie eine Member-Funktion

   `move ( dx, dy, dz)` ,

   die ein Particle um die angegebenen Werte in den drei Koordinaten verschiebt (also x+dx, y+dy, z+dz).

   Hinweis: Überlegen Sie sich, wo diese Funktion überall deklariert/definiert werden muss, um möglichst wenig Arbeit zu haben. Zum Test löschen Sie die Kommentarzeichen in Schleife 1 in main().

   (Ausgearbeitetes Beispiel: `py`)

6. **Mondlandung**

   Ein Spieleklassiker ist die Mondlandung:

   - Raumfähre wird von Mond angezogen
   - Mit Gegen-Schub kann man Fall kontrollieren, allerdings sind Treibstoffvorräte begrenzt
   - Ziel ist möglichst weiche Landung auf Mond.

   Erstellen Sie eine Python Klasse für Mondfähre, welche Datenelemente, welche Methoden werden benötigt?

   Lösungsbeispiel nach Buch *Coding for Fun mit C++ bzw Python* `mondlandung.py`.

   Modifizieren Sie das Programm, z.B. zufällige Starthöhe, zufällige Variation des Schubfaktors, etc.

3 Python Standard–Bibliotheken

- Exception handling
- Mehr zu Python Container
- Dokumentation und Introspection

3.1 Exceptions

Wahrscheinlich hat jeder schon mal *exceptions* gesehen:

```
import sys                                  1
print "Hallo ", sys.argv[1]                 2
```

Wenn man das ohne Argument startet

```
python Hello2.py
```

bekommt man

```
Traceback (most recent call last):
File "Hello2.py", line 2, in ?
print "Hallo ", sys.argv[1]
IndexError:  list index out of range
```

weil man versucht auf `argv[1]` zuzugreifen, das aber nicht existiert, wenn man keine weiteren Argumente beim Aufruf übergibt. ist.

Zwei Arten das zu vermeiden:

Die *klassische* mittels **control-statements**:

```
import sys                                                    1
if len(sys.argv) > 1 :                                        2
    print "Hallo ", sys.argv[1]                               3
else:                                                         4
    print "Hallo ", "wer auch immer da hockt"                 5
#                                                             6
                                                              7
```

Dadurch wird verhindert dass Exception auftritt.

Und die **moderne** via **try–except**

```
import sys                                                    1
try:                                                          2
    print "Hallo ", sys.argv[1]                               3
except:                                                       4
    print  "wer auch immer da hockt"                          5
    # Programm bricht nicht ab !                              6
print "\n Und weiter geht's ..."                              7
#                                                             8
                                                              9
```

Das fängt die Exception auf innerhalb des `except:` Blocks.

Wozu das Ganze ?

Programme sollen **robust** sein, d.h. sie sollen falsche Eingaben, extreme Datenwerte oder generell unerwartete Situationen vernünftig abfangen und nicht bei jeder Kleinigkeit das ganze Programm abbrechen.

Die klassische Methode mit `if` Abfragen hat eine Reihe von Nachteilen

- u.U. viele Abfragen nötig $\Rightarrow$ unleserliche Programme
- schwer alle Möglichkeiten abzudecken
- check kann nicht erzwungen werden, z.B. in C
  `int fd = open("file.dat",0)` ist ein return Wert (`fd = -1`) Zeichen für Probleme beim Öffnen des files, aber das **muss** nicht behandelt, sondern kann einfach ignoriert werden.

Exceptions bieten elegante Lösung, insbesondere können Sie nicht einfach ignoriert werden.

`if` Abfragen vs Exceptions ist klassisch–philosophisches IT–Problem, siehe z.B. **LBYL** vs **EAFP**.

Exceptions sind in Python als Klassen definiert und hierarchisch aufgebaut. Hier die wichtigsten, ausgehend von der Basisklasse **Exception**:

```
Exception
 ..
 +-- StandardError
 |    +-- KeyboardInterrupt
 |    +-- ImportError
 |    +-- EnvironmentError
 |    |    +-- IOError
 |    +-- EOFError
 |    +-- RuntimeError
 |    +-- AttributeError
 |    +-- TypeError
 |    +-- AssertionError
 |    +-- LookupError
 |    |    +-- IndexError
 |    |    +-- KeyError
 |    +-- ArithmeticError
 |    |    +-- OverflowError
 |    |    +-- ZeroDivisionError
```

```
|     |     +-- FloatingPointError
|     +-- ValueError
|     |     +-- UnicodeError
|     |           +-- UnicodeEncodeError
|     |           +-- UnicodeDecodeError
|     |           +-- UnicodeTranslateError
|     +-- ReferenceError
|     +-- SystemError
|     +-- MemoryError
```

Verschiedene Optionen mit Exceptions umzugehen

Einfach ignorieren, dann Abbruch des Programms wenn Exception auftritt.

Oder man fängt sie mit **try–except** ab:

Generisch jede Exception

```
#M=[[0.]*2]*1
try:
    determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0]
except Exception, x:
    print  "Troubles: ", x.__class__.__name__ , ':', x
#
```

Oder spezifisch die einzelnen Möglichkeiten mit mehreren `except` statements

```
M=[[0.]*2]*1
try:
    determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0]
except IndexError, x:
    print "M is the wrong size to have a determinant.", x
except NameError, x:
    print "Programming error! M doesn't exist: ", x
#
```

Ablauf:

- Keine Exception, alles ok, dann wird nur `try:` Block durchlaufen
- Exception tritt auf:
  - `except Exception-(sub)class` für diese Exception vorhanden, dann wird dieser `except` Block ausgeführt.
    Das übergebene *Exception Objekt* enthält weitere Informationen.
    Anschliessend läuft das Programm weiter.
  - kein entsprechendes `except` vorhanden:
    $\Rightarrow$ Abbruch des Programms

Selber Exceptions auslösen

Man kann selbst Exceptions auszulösen, mit

`raise SomeExceptionClass`

```
import math
def root( A, B, C):
    """ Returns the two roots of
      the quadratic equation A*x*x + B*x + C = 0.
      (Throws an exception if A == 0 or B*B-4*A*C < 0.)"""
    if (A == 0):
        raise ValueError("A can't be zero.")
    else :
        disc = B*B - 4*A*C
        if (disc < 0):
            raise ArithmeticError("Discriminant < zero.")
        else:
            return (   (-B - math.sqrt(disc)) / (2*A),     (-B + math.sqrt(disc)) / (2*A) )
#
print root( 1., 5., -3.)      # ok
root( 0., 2., 1.)       # causes ValueError exception
```

```
root( 3., 2., 15.)      # causes ArithmeticError exception                    17
```

Insgesamt ermöglichen Exceptions eine Funktionalität, die allein mit
`return errnum` und `if` Abfragen nicht zu erreichen wäre:

- **Entwickler** einer Klasse/Methode kann entscheiden was ein fataler Fehler ist und eine entsprechende Exception auslösen (oder sogar selbst eigene Exceptions definieren und verwenden $\Rightarrow$ Literatur)
- Es ist dem **Anwender** der Klasse/Methode überlassen, es mit **try–except** abzufangen oder einen Programmabruch in Kauf zu nehmen.

3.2 Python–Dictionaries

Neben **list** und **tuple** gibt es einen weiteren wichtigen Container, das **dict**. *(Heisst in C++/JAVA map, in Perl associative array)*

Bei *list, tuple* sind die einzlnen Elemente in einer festen Reihenfolge geordnet, der Zugriff läuft meist über eine Index-Nummer, d.h. die numerische Position ist mit dem Objekt assoziert.

Bei **dict** dagegen werden Paare von Werten gespeichert, **key** und **value**.

Zugriff auf die Elemente erfolgt über den **key**, ähnlich wie mit dem numerischen Index bei *array, list, tuple*, deshalb auch die Bezeichnung assoziativer Array.

Beispiel Wörterbuch:

```
#                                                    1
engdeut={} # create empty dict                       2
engdeut["hello"]    = "Hallo"                        3
engdeut["world"]    = "Welt"                         4
engdeut["computer"] = "Rechner"                      5
engdeut["physics"] = "Physik"                        6
engdeut["physicist"] = "Physiker"                    7
engdeut["physician"] = "Arzt"                        8
```

Verwendung:

- Initialisierung: Leeres **dict** anlegen
  `D = {}`
- Elemente einfügen mit
  `D[key] = value`
  key und **value** können im Prinzip beliebige Datentypen sein, also sowohl die Python–Standardtypen als auch eigene Klassen. Für **key** gibt es eine Einschränkung: die Klasse muss eine sogennante *hash* Methode bereitstellen (implizit alle Python Standard–Datentypen und Sequences).
- Lese–Zugriff analog: Direkt mit Angabe eines *keys* als *array-index*, z.B. `engdeut["physics"]`.
  Oder sequentiell durchlaufen mit Iteratoren:

```
#                                                                    1
..                                                                   2
print len(engdeut) # Laenge = 6                                      3
#                                                                    4
print engdeut.keys() # list der keys ['physician', 'physicist', ...] 5
#                                                                    6
for key in engdeut.keys(): # iterate over keys                       7
    print engdeut[key]                                               8
#                                                                    9
for (e,d) in engdeut.items(): # iterate over key/value pairs         10
    print e, d                                                       11
    #                                                                12
```

Ein **dict** ist eine Art **Liste von Paaren**. Diese Liste ist sortiert nach dem *key*. Deshalb:

- Jeder *key* kann nur einmal vorkommen. Bei erneuter Zuweisung wird existierender Wert überschrieben:
  ```
  engdeut["computer"] = "Gombuder" # fraenkische Version
  ```
- Zugriff relativ effizient über sog. *hash* Algorithmen, dennoch langsamer und umständlicher als normale tuple/list über numerischen Index.

Test ob dict Element existiert

Lese–Zugriff auf dict Element (`dwort = engdeut["music"]`) geht nur, wenn Element schon angelegt ist, andernfalls wird Exception ausgelöst.

Zwei Möglichkeiten damit umzugehen:

- Explizit testen mit **has_key()** dict Methode, z.B.

```
engdeut.has_key("music") # False
engdeut.has_key("physics") # True
```

- oder einfach mal probieren mit **try–except**:

```
try:
    dwort = engdeut["music"]
except KeyError:
    fcolr = "No translation for ", "music"
```

3.3 Dokumentation und Introspection

In Python ausgefeilte Mechanismen zur Dokumentation von Funktionen und Klassen sowie zur Abfrage von Informationen.

- **Docstring**: Es ist Python Konvention in die erste Zeile nach Definition einer Klasse oder Funktion in einem Textstring eine Kurzbeschreibung der Klasse/Funktion zu machen. Diese Beschreibung ist mehr als ein Kommentar, sie kann zusammen mit der Klasse/Funktion zur Laufzeit abgefragt werden.

```
class ThreeVec:                                                    1
    "Class for 3 Vector and operations"                            2
    def Add( self, tv ): # add two 3 vecs                          3
        "Add two ThreeVecs"                                        4
        newvec = ThreeVec( self.x+tv.x, self.y+tv.y, self.z+tv.z)  5
            return newvec                                          6
```

- Entsprechende Info zur Laufzeit abfragbar:
  `print ThreeVec.__doc__` oder
  `print ThreeVec.Add.__doc__`

- oder sehr komfortabel interaktiv mit `help(ThreeVec)`

```
Help on class ThreeVec in module ThreeVec:

class ThreeVec
 |  Class for 3 Vector and operations
 |
 |  Methods defined here:
 |
 |  Add(self, tv)
 |
 |  Length(self)
 |
 |  SkalProd(self, tv)
 |
 |  __add__(self, tv)
 |      method for overloading the + operator
 |
 |  __cmp__(self, tv)
 |
 |  __init__(self, x=0.0, y=0.0, z=0.0)
```

```
 |
 |  __str__(self)
...
```

- oder Basis–Info zu Methoden: `dir(ThreeVec)`

```
['Add', 'Length', 'SkalProd', '__add__', '__cmp__', '__doc__',
'__init__', '__module__', '__str__']
```

- genauso für Objekte, Info zu Methoden und Variablen:
  `v = ThreeVec(4.,5.,7.); dir(v)`

```
['Add', 'Length', 'SkalProd', '__add__', '__cmp__', '__doc__',
'__init__', '__module__', '__str__', 'x', 'y', 'z']
```

- und schliesslich noch Abfrage von Typ über `type(Object)` oder direkt `Object.__class__.__name__`

```
>>> a=5.2
>>> b=[1,6,"abc"]
>>> c="Hello"
>>> type(a)
<type 'float'>
>>> type(b)
```

```
<type 'list'>
>>> type(c)
<type 'str'>
>>> type(v)
<type 'instance'>
>>> c.__class__.__name__
'str'
>>> b.__class__.__name__
'list'
>>> v.__class__.__name__
'ThreeVec'
```

3.4 Umlaute

Umlaute sowohl im Quelltext als auch in der Benutzereingabe und Ausgabe müssen in Python speziell behandelt werden. Untenstehendes Beispiel basierend auf den folgenden Stackoverflow Webseite demonstiert kurz, wie man Umlaute behandeln kann (umlaut.py):

```
1  #!/usr/bin/env python
2  # -*- coding: iso-8859-1 -*-
3
4  import sys
5  # Encoding der Standardausgabe herausfinden
6  stdout_encoding = sys.stdout.encoding or sys.getfilesystemencoding()
7
8  temp = "Mnchen"
9  utemp = u"Mnchen"
10
11 print temp
12 print utemp
13
14 print temp.decode("iso-8859-1").encode(stdout_encoding)
15
16 print utemp.encode(stdout_encoding)
```

3.5 Aufgaben

1. **Quadratische Gleichung**
   Erstellen Sie ein Programm zur Lösung der quadratischen Gleichung

   $$A\,x^2 + B\,x + C = 0$$

   basierend auf obigem Beispiel `root`

   - Die Koeffizienten `A,B,C` von standard-input lesen
   - **try–except** im rufenden Programm um auf die Exceptions *sinnvoll* zu reagieren.

2. **Idiotensichere Fakultät**
   Modifizieren Sie ihre Funktion zur Berechnung der Fakultät für *float Zahlen* so, dass kein Überlauf mehr auftreten kann, d.h. lösen Sie eine Exception aus
   `raise ArithmeticError("Number too large")`
   wenn der Wert zu gross wird.

   Hinweis: Maximaler Float Wert auf Linux ist 1.7976931348623157E308. Jetzt muss man nur noch überlegen wie man's programmiert **ohne** *dass zunächst die Grenze überschritten wird.*

3. **Vorwahl–Dict**

   In der Datei `vorwahl.txt` stehen alle Vorwahlen und zugehörige Orte in Deutschland. Lesen Sie diese ein, speichern Sie's in einer **dict** und machen damit ein kleines Programm, das zu einer gegebenen Vorwahl den Ort ausgibt, und umgekehrt.

   Ausgearbeitetes Beispiel: `py`

4. **Genom Projekt**

   Eine DNA Sequenz kann als Array von *N Char* Werten dargestellt werden (N sehr gross). Das Problem ist, wiederkehrende Strukturen zu finden, d.h. Patterns der Länge M, wobei M fix und klein ist. In der Datei genom.txt finden Sie einen Abschnitt einer solchen DNA Sequenz. Überlegen Sie Algorithmen um signifikant häufige Patterns für vorgegebene Länge M zu finden.

   Ausgearbeitetes Beispiel: `py`

5. **Poker simulieren**

   Mit *python-lists* und *random.shuffle(list-name)* kann man leicht Spiele simulieren, und damit die Wahrscheinlichkeit für bestimmte Kombinationen abschätzen (ohne sich in den Feinheiten der Kombinatorik zu verirren). Simulieren Sie z.B. das Pokerspiel, was ist die Wahrscheinlichkeit ein Full-House auf die Hand zu bekommen ?

   Lösungsbeispiel: source

4 Weitergehende und häufig verwendete Python Features

Python bietet etliche sehr nützliche weitergehende Features, die über das Standard-Repertoire gängiger Programmiersprachen hinausgehen und die in Python häufig verwendet werden. Wir behandeln hier kurz *Tools für Listen und Dicts, Generators, flexible Funktionsaufrufe, reguläre Ausdrücke*

4.1 Tools für Listen und Dicts

Aus einer Liste möchte man oft Elemente auswählen, die ein bestimmtes Kriterium erfüllen. Oder es soll eine Liste in eine andere Liste transformiert werden. Man könnte auch eine Kombination von Filtern und Transformieren anwenden. Hierzu stehen einerseits die Funktionen `filter` und `map` zu Verfügung. Andererseits kann man sog. list comprehensions verwenden.

- Als Beispiel soll eine Liste `[ 1, 2, 3, 4, 5]` dienen.
- Jedes Listenelement soll mit 10 multipliziert werden.
- Nur gerade Elemente sollen ausgewählt werden
- Nur gerade Elemente sollen ausgewählt und mit 10 multipliziert werden.

Zunächst `filter` und `map`:

```
>>> liste1 = [ 1, 2, 3, 4, 5 ]
```

```
>>> map(lambda x: x*10, liste1)
[10, 20, 30, 40, 50]
>>> filter(lambda x: x % 2 == 0, liste1)
[2, 4]
>>> map(lambda x: x*10, filter(lambda x: x % 2 == 0, liste1))
[20, 40]
```

Aber man kann auch sog. **list comprehensions** verwenden:

```
>>> [element*10 for element in liste1]
[10, 20, 30, 40, 50]
>>> [element for element in liste1 if element % 2 == 0]
[2, 4]
>>> [element*10 for element in liste1 if element % 2 == 0]
[20, 40]
```

List comprehensions haben folgenden allgemeine Form:

```
[ expr(element) for element in iterable if pred(element) ]
```

mit

- `expr(element)` ein beliebiger Ausdruck abhängig von `element`,
- `iterable` eine beliebige Sequenz und
- `pred(element)` eine Funktion, die `True` oder `False` liefert und von `element` abhängig ist.

Man kann auch mehrere Listen kombinieren:

```
[(x,y) for x in range(5) for y in range(5) ]
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (
```

und jeweils auch noch *if* Bedingung einbauen:

```
[(x,y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

damit erhält man eine Kombination aller geraden Zahlen von 0 bis 4 und aller ungeraden Zahlen von 0 bis 4.

Es entspricht einer doppelten *for* Schleife:

```
result = []
for x in range(5):
   if x % 2 == 0:
      for y in range(5):
         if y % 2 == 1:
            result.append((x,y))
```

Mit expliziten for Schleifen übersichtlicher und leichter verständlich, aber deutlich aufwendiger beim Schreiben und wesentlich langsamer bei der Ausführung.

Analog zu **list comprehensions** gibt es die **dict comprehensions**

```
sqdict = { i : i**2 for i in range(10) }
print ( sqdict )
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Listen zusammenführen mit zip

```
a=[ 1,2,3]
b=['a','b','c']
zip(a,b)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Ergibt kombinierte Liste von *tuples*

Kann leicht erweitert werden um 2 Listen in ein dict zu kombinieren:

```
d = { x[1] : x[0] for x in zip(a,b) }
print(d)
{'a': 1, 'c': 3, 'b': 2}

# Oder direkter ...
d = dict(zip(b,a))
```

defaultdict

Im *collections* module gibt es nützliche Hilfs-Klassen zur Arbeit mit Listen und Dicts:

```
import sys
import urllib2
# open Kant's Text
# f=urllib2.urlopen("http://www-static.etp.physik.uni-muenchen.de/kurs/Computing/python/source/kant.txt")
f=urllib2.urlopen("https://goo.gl/rGqW4k")
# split into words
words=[]
for line in f: # iteriere ueber alle Zeilen
    words += line.split() # packe Words in list
# or more direct w/ double list-comprehension:
# words=[ word for line in f for word in line.split() ]
#
# count words v1
word_counts = {}
for word in words:
    if word in word_counts:
        word_counts[word] += 1
```

```
    else:                                                                    18
        word_counts[word] = 1                                                19
# Umstaendlich ...                                                           20
#                                                                            21
# count words v2                                                             22
word_counts = {}                                                             23
for word in words:                                                           24
    try:                                                                     25
        word_counts[word] += 1                                               26
    except:                                                                  27
        word_counts[word] = 1                                                28
# Auch umstaendlich ...                                                      29
#                                                                            30
# count words v3                                                             31
from collections import defaultdict                                          32
word_counts = defaultdict(int)                                               33
for word in words:                                                           34
    word_counts[word] += 1                                                   35
# defaultdict(int) initialisiert Eintraege beim Ansprechen automatisch auf int() = 0  36
#                                                                            37
```

```
# oder noch einfacher ...
from collections import Counter
word_counts=Counter(words)
#
# Counter liefert eine Art dict zurueck mit das als Wert die Haeufigkeit enthaelt:
word_counts["Vernunft"]
# und weitere Methoden ...
word_counts.most_common(10) # die 10 haeufigsten ...
```

4.2 Iterables und Generatoren (yield)

Listen, Dicts, etc, sind sogenannte *Iterables*, d.h. alles über was man in einer `for ... in ...` Schleife drüberlaufen kann, also z.B.:

```
1 mylist = [x*x for x in range(3)]
2 for i in mylist:
3     print(i)
4
5 mylist
6 [0, 1, 4]
```

Bei Listen werden alle Elemente der Liste erzeugt und im Speicher abgelegt, das kann ggf. sehr viel sein.

Als Alternative gibt es *Generators*:

```
1 mygenerator = (x*x for x in range(3))
2 for i in mygenerator:
3     print(i)
4
5 mygenerator
6 <generator object <genexpr> at 0x7f63a2c05320>
```

Verwendung hier fast identisch, nur Erzeugung mit runden Klammern statt eckigen. Und es wird

Generator-Objekt angelegt, das man benutzen kann um **einmal** die Werte **nacheinander** abzurufen, es wird dabei jeweils nur das aktuelle Element angelegt, und am Ende ist das Generator Objekt fertig, d.h. man kann mit `for i in mygenerator:` nicht nochmal durchlaufen.

Man kann diese Funktionalität auch mittels sogenanter *Generator-Funktionen* und **yield** erreichen:

```
1  # a generator that yields items instead of returning a list
2  def firstnsq(n):
3      num = 0
4      while num < n:
5          yield num*num
6          num += 1
7
8  mygen = firstnsq(3)
9  for i in mygen:
10     print(i)
```

Das Key-word **yield** entspricht etwa dem **return** bei normalen Funktionen, nur der Ablauf ist anders:

- Aufruf `mygen = firstnsq(3)` führt nicht den Generator-Code aus sondern erzeugt nur Generator-Objekt
- Beim 1. Aufruf des Objekts in `for` Schleife werden Anweisungen wie in Funktion ausgeführt bis `yield` kommt, dann bricht Generator ab und liefert Wert zurück
- Bei weiteren Aufrufen wird die Schleife im Generator bis zum nächsten `yield` fortgeführt.

- Falls kein `yield` mehr kommt ist der Generator zu Ende (=fertig).

Statt mit `for ... in ...` Schleife kann man auch mit `next(...)` die einzelnen Werte abrufen:

```
1 mygen = firstnsq(3)
2 mygen
3 <generator object firstnsq at 0x7f63a2c05140>
4 next(mygen)
5 0
6 next(mygen)
7 0
8 ...
```

Mehr dazu in dieser Erklärung.

4.3 Funktionsaufrufe

Eine Funktion in allgemeiner Form sieht folgendermassen aus:

```
def f(param1, param2='dummy', *liste1, **dict1):
    """Eine Funktion die ihre Argumente ausgibt"""
    print "param1: ", param1
    print "param2: ", param2
    print "liste1: ", liste1
    print "dict1: ", dict1
    try:
        nachname1 = dict1.get('nachname')
        print nachname1
    except:
        pass

    return [param1, param2]

f('hello', 'world', 'mehr', 'Argumente', nachname = 'max', vorname='mueller')
```

Die Ausgabe diese Beispielprogramms sieht dann so aus:

```
param1:  hello
param2:  world
liste1:  ('mehr', 'Argumente')
dict1:  {'nachname': 'max', 'vorname': 'mueller'}
max
```

Eine Funktion hat die Parameter:

- `param1` ohne ‚default'-Wert,
- `param2='dummy'` mit einem ‚default'-Wert,
- `*liste1`, die eine unspezifizierte Anzahl von Parametern in einer Liste speichert
- `*dict1`, die eine unspezifizierte Anzahl von Parmetern mit key,value in einem dictionary abspeichert. Die einzelen Werte können mit z.B. `get` abgefragt werden.

4.4 Reguläre Ausdrücke

In Strings kann man meist einfache Teil-Strings suchen und evt. ersetzen. Hierzu kann man die einfachen String-Methoden `index`, `rindex`, `find`, `rfind`, `replace` und deb Operator `in` verwenden.

Mit Hilfe von regulären Ausdrücken kann man in Strings nach komplizierten Mustern suchen und Teile des Strings ersetzen. Eine ausführliche Beschreibung mit Beispielen zu regulären Ausdrücken gibt es unter: Regular Expressions

Das Python Modul **re** stellt zahlreiche Funktionen zur Verwendung von regulären Ausdrücken zur Verfügung.

`re.search` im Vergleich zu `in`:

```
>>> import re
>>> input = 'Franz jagt im komplett verwahrlosten Taxi quer durch Bayern'
>>> re.search(r'Taxi', input)
<_sre.SRE_Match object at 0x7f9d50536e68>
>>> 'Taxi' in input
True
>>> re.search(r'Bus', input)
>>> 'Bus' in input
```

```
False
```

Ein String, der mit `r` eingeleitet wird, heisst ‚roher' String. In diesem müssen keine backslashes entwertet werden, d.h. man gibt in Folgendem Beispiel entweder `r'\bTaxi\b'` oder `'\\bTaxi\\b'` an.

Falls nach einzelnen Wörtern gesucht werden soll, zeigt `re.search` mit dem Extra Parameter `\b` seinen Vorteil:

```
>>> input1 = 'Franz jagt im komplett verwahrlosten Taxi quer durch Bayern'
>>> input2 = 'Der Taxibus ist zu spaet'
>>> 'Taxi' in input1, 'Taxi' in input2
(True, True)
>>> re.search(r'\bTaxi\b', input1), re.search(r'\bTaxi\b', input2)
(<_sre.SRE_Match object at 0x7f9d50536ed0>, None)
```

Zum Ersetzen benutzt man `re.sub`:

```
>>> input1 = 'Franz jagt im komplett verwahrlosten Taxi quer durch Bayern'
>>> output=re.sub(r'Taxi','Bus',input)
>>> output
'Franz jagt im komplett verwahrlosten Bus quer durch Bayern'
```

Mit dem `match` Objekt kann man auf Teile des String zurückgreifen, die zu regulären Ausdruücken passen. Mit

```
r'(\b\w+\b)\s+\1'
```

lässt sich nach einem doppelt vorkommendem Wort suchen:

```
>>> input = 'Franz jagt im komplett verwahrlosten Taxi quer quer durch Bayern'
>>> mo=re.search(r'(\b\w+\b)\s+\1',input)
>>> mo
<_sre.SRE_Match object at 0x7f9d50555300>
>>> mo.group(0)
'quer quer'
>>> mo.group(1)
'quer'
>>> mo.start()
42
>>> mo.span()
(42, 51)
>>> input[42: 51]
'quer quer'
```

`re.search` liefert nur das **erste** Vorkommen eines Such-Musters. Alle Vorkommen erhält man mit

`re.findall` oder `re.finditer`.

Ein schnelleren Zugriff auf Suchergebnisse vorallem bei grösseren Strings oder dem zeilenweisen Lesen/Suchen durch eine Datei erhält man mit `re.compile`. Der Such-Begriff wird einmal ‚kompiliert' und kann anschliessend wiederverwendet werden.:

```
>>> input3 = 'Franz jagt im komplett verwahrlosten Taxi quer durch Bayern'
>>> input4 = 'Franz jagt im komplett verwahrlosten Taxi quer quer durch Bayern'
>>> regdoub = re.compile(r'(\b\w+\b)\s+\1')
>>> regdoub
<_sre.SRE_Pattern object at 0xb75c71a0>
>>> regdoub.search(input3)
>>> regdoub.search(input4)
<_sre.SRE_Match object at 0xb75999a0>
>>> regdoub.sub(r'\1',input3)
'Franz jagt im komplett verwahrlosten Taxi quer durch Bayern'
>>> regdoub.sub(r'\1',input4)
'Franz jagt im komplett verwahrlosten Taxi quer durch Bayern'
```

Mit dem Python Modul `fnmatch` kann mit der Unix Dateiname-Suche Konvention in `strings` gesucht werden. Hierbei werden die von der bash-Kommandozeile bekannten Regeln verwendet:

```
*  entspricht allem
```

```
?  entpricht einem Buchstaben
[seq]  entspricht einem Buchstaben in seq
[!seq]  entspricht einem Buchstaben nicht in seq
```

Folgendes Beispiel zeigt alle Dateinamen im aktuellen Verzeichnis mit der Endung `.txt`:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print file
```

4.5 Aufgaben

- **Zufallszahlen erzeugen**
  Mit Hilfe von list comprehensions erstellen Sie eine Liste von Zufallszahlen, z.B. einen Würfel-Wurf oder zwei-Würfel-Würfe gleichzeitig. Verwenden Sie hierzu `random.randrange(1,7)`.
  Lösung: wuerfel.py

- **Funktionsparameter**
  Schreiben Sie eine Funktion, die nur ein Dictionary als Argument-Liste benutzt und werten Sie die Parameter in der Funktion aus.

- **Suchen in Strings**
  Schreiben Sie ein Programm, das einen String und einen Substring als Eingabe nimmt. Der Substring soll im String gesucht werden und die Position und evt. mehrmaliges Vorkommen geprüft werden.
  Lösung: stringsearch.py

- **Wörter zählen**
  Das Programmbeispiel zum Zählen von Wörtern in `kant.txt` ist etwas schlampig gemacht, weil Satzzeichen nicht korrekt behandelt werden (z.B. `Vernunft` und `Vernunft,` werden getrennt gezählt). Wie lässt sich das beheben?
  Lösung: wordcount.py

- **Strings kodieren**

  Das Programm text_encode.py zeigt ein kurzes Beispiel zur Verschlüsselung von Text-Strings nach dem sog. Caesar–Algorithmus.

  (a) Versuchen Sie die einzelnen Programmschritte nachzuvollziehen

  (b) Ändern Sie den Algorithmus, so dass statt fester Verschiebung ein zufälliges Mapping der Characters gemacht wird (Funktion `random.shuffle(list)` bringt Elemente einer Liste in zufällige Reihenfolge)

5 Numpy,Scipy und Matplotlib

Es gibt zahlreiche wissenschaftliche Programme, Pakete und Biliotheken, die in verschiedenen Sprachen geschrieben worden sind: Mathematica, Maple, Matlab, Root, Numerical Recipes, etc. Für große wissenschaftliche Anwendungen sind oft Ausführgeschwindigkeit wichtig. Es existieren zahlreiche externe Biliotheken auf die mit einer Python API zugegriffen werden kann. Im Folgenden werden folgende Pakete besprochen:

- numpy
- scipy
- matplotlib

Zur Demonstration was damit gemacht werden hier ein Beispiel zur Lösung eines klassischen Problems aus der Physik: Zwei Massen sind über Federn gekoppelt. Die Bewegungsgleichungen dieses Systems sollen gelöst und der Zeitverlauf der Feder-Auslenkungen graphisch dargestellt werden.

Die Lösung unter der Benutzung von `scipy` und `matplotlib` ist in diesem Wiki demonstriert.
(Source code: two_springs.py, two_springs_solver.py, two_springs_plot_new.py)

5.1 Jupyter notebook

Eine tolle Bedienungsoberfläche für Python bietet das **Jupyter Notebook**, damit kann man interaktive Python Umgebung über Browser Fenster starten, die viele nützliche Features für interaktives Arbeiten bietet: history, tab completion, system interface, u.v.m.

Eine besonders schönes Feature sind die eingebetteten Grafiken von Matplotlib.

- Starten mit

```
module load jupyter
jupyter notebook --browser=firefox
```

- Beenden mit `Strg-C` im Shell Fenster bzw. `Logout` im Browser.

Weitere Infos z.B. https://jupyter.readthedocs.io/en/latest/index.html

IP[y]: Notebook tmatpl Last Checkpoint: Oct 01 09:40 Autosave Failed!

File Edit View Insert Cell Kernel Help

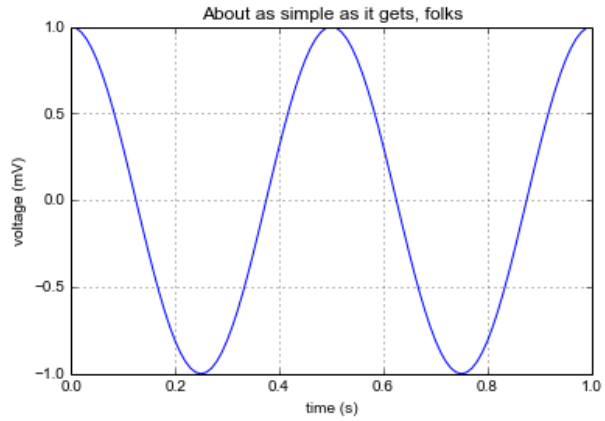
Code Cell Toolbar: None

In [1]:
```
5+5
```

Out[1]: 10

In [2]:
```
import numpy as np
import pylab as pl

t = np.arange(0.0, 1.0+0.01, 0.01)
s = np.cos(2*2*np.pi*t)
pl.plot(t, s)
pl.xlabel('time (s)')
pl.ylabel('voltage (mV)')
pl.title('About as simple as it gets, folks')
pl.grid(True)
pl.savefig('simple_plot')
```



5.2 NumPy

In der Praxis oft sehr rechenintensive numerische Operationen nötig:

- große Gleichungssysteme
- Matrizen
- Numerische Integration
- Fourier-Transformation
- Statistische Auswertungen und Berechnungen

Oft gibt es sehr ausführliche numerische Bibliotheken geschrieben in Fortran oder C/C++ (z.B. Numerical Recipes). `numpy` bietet Schnittstellen zu BLAS (Basic Linear Algebra Library) und ATLAS (Automatically Tuned Linear Algebra Software).

`numpy` bietet einen sehr effizienten Array-Datentyp `ndarray` mit Matrix-Aussehen und Funktionen aus der BLAS und ATLAS Bibliothek. Eine homogene Sammlung von `float`- und `int`-Werten wird hinter den Kulissen mit C- oder Fortran verwaltet und bietet zusätzlich die Möglichkeit *Parallelisierung oder Vektorisierung* zu nutzen anstatt einzelner Operationen in Schleife.

Folgendes Programm benötigt 15-25s

```
a = range(10000000)
b = range(10000000)
c = []
for i in range(len(a)):
  c.append(a[i] + b[i])
```

Mit `numpy` ca. 0.5s

```
import numpy as np
a = np.arange(10000000)
b = np.arange(10000000)
c = a + b
```

Module von `numpy`

Interessant: `core, fft, linalg, lib`

```
['__config__', '_import_tools', 'add_newdocs', 'char', 'core',
 'ctypeslib', 'emath', 'fft', 'lib', 'linalg', 'ma', 'math',
 'random', 'rec', 'testing', 'version']
```

Funktionen in `numpy.linalg` :

```
['cholesky', 'cond', 'det', 'eig', 'eigh', 'eigvals', 'eigvalsh',
 'inv', 'lstsq', 'matrix_power', 'norm', 'pinv', 'qr', 'solve',
 'svd', 'tensorinv', 'tensorsolve']
```

Hilfetext zu `help(numpy.linalg.qr)`

```
Help on function qr in module numpy.linalg.linalg:

qr(a, mode='full')
    Compute QR decomposition of a matrix.

    Calculate the decomposition :math:`A = Q R` where Q is orthonormal
    and R upper triangular.
......
```

Der Datentyp `numpy.ndarray`, den man mit `numpy.array` erhält, ist ein homogener Datentyp, also eine Liste von `int` oder `float` mit Struktur. `ufunc`s (Universalfunktionen) operieren auf jedes Element und sind durch Fortran oder C-Implementierung sehr schnell.

```
1
2  In [23]: import  numpy as np
3  In [24]: print(typeDict)
4  {0: <type 'numpy.bool_'>,
5  ....
6   'u1': <type 'numpy.uint8'>,
7   'u2': <type 'numpy.uint16'>,
8   'u4': <type 'numpy.uint32'>,
9   'u8': <type 'numpy.uint64'>,
10  'ubyte': <type 'numpy.uint8'>,
11  'uint': <type 'numpy.uint32'>,
12  'uint0': <type 'numpy.uint32'>,
13 ....
14 In [26]: a=np.array([[1,2,3,4],[4,5,6,7],[9,10,11,12]])
15
16 In [27]: a
17 Out[27]:
18 array([[ 1,  2,  3,  4],
19        [ 4,  5,  6,  7],
20        [ 9, 10, 11, 12]])
21
22 In [28]: print a
```

```
23 ----------> print(a)
24 [[ 1  2  3  4]
25  [ 4  5  6  7]
26  [ 9 10 11 12]]
27 In [29]: type(a)
28 Out[29]: <type 'np.ndarray'>
29 In [32]: [m for m in dir(a) if not m.startswith('__')]
30 In [35]: a.shape
31 Out[35]: (3, 4)
32 In [36]: a.reshape(4,3)
33 Out[36]:
34 array([[ 1,  2,  3],
35        [ 4,  4,  5],
36        [ 6,  7,  9],
37        [10, 11, 12]])
38 In [37]: a.reshape(2,6)
39 Out[37]:
40 array([[ 1,  2,  3,  4,  4,  5],
41        [ 6,  7,  9, 10, 11, 12]])
42 In [39]: a[2,3]
43 Out[39]: 12
44 In [40]: a.T
45 Out[40]:
46 array([[ 1,  4,  9],
47        [ 2,  5, 10],
48        [ 3,  6, 11],
```

```
        [ 4,  7, 12]])
In [41]: -1*a
Out[41]:
array([[ -1,  -2,  -3,  -4],
        [ -4,  -5,  -6,  -7],
        [ -9, -10, -11, -12]])
In [42]: a+a
Out[42]:
array([[ 2,  4,  6,  8],
        [ 8, 10, 12, 14],
        [18, 20, 22, 24]])
In [43]: a-a
Out[43]:
array([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
In [44]: a*a
Out[44]:
array([[  1,   4,   9,  16],
        [ 16,  25,  36,  49],
        [ 81, 100, 121, 144]])

In [45]: np.dot(a,a)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

```

```
/tmp/examples/scipy/<ipython console> in <module>()

ValueError: objects are not aligned

In [47]: np.dot(a,a.reshape(4,3))
Out[47]:
array([[ 67,  75,  88],
       [130, 147, 175],
       [235, 267, 320]])
In [48]: np.sqrt(a)
Out[48]:
array([[ 1.        ,  1.41421356,  1.73205081,  2.        ],
       [ 2.        ,  2.23606798,  2.44948974,  2.64575131],
       [ 3.        ,  3.16227766,  3.31662479,  3.46410162]])
In [50]: type(np.sin)
Out[50]: <type 'numpy.ufunc'>
```

Arrays indizieren

```
In [1]: import numpy as np

In [2]: a=np.array([[1,2,3,4],[4,5,6,7],[9,10,11,12]])
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 9, 10, 11, 12]])

In [4]: a[1,2] # Element 2. Zeile, 3. Spalte
Out[4]: 6

In [5]: a[1] # 2. Zeile
Out[5]: array([4, 5, 6, 7])

In [6]: a[:2] # 1. und 2. Zeile
Out[6]:
array([[1, 2, 3, 4],
       [4, 5, 6, 7]])

In [7]: a[:,2] # 3. Spalte
Out[7]: array([ 3,  6, 11])

In [10]: a[:,-1] # Letzte Spalte
Out[10]: array([ 4,  7, 12])

In [12]: a[:2,1:3] # ...
```

```
Out[12]:
array([[2, 3],
       [5, 6]])
```

Werte zählen:

```
In [51]: a=np.array(range(100))
In [25]: a
Out[25]:
array([ 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, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])

In [26]: np.where(a>50)
Out[26]:
(array([51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]),)

In [27]: len(np.where(a>50))
Out[27]: 1
```

```
In [28]: len(np.where(a>50)[0])
Out[28]: 49


In [16]: a>80
Out[16]:
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,  True], dtyp

In [17]: a[a>80]
Out[17]:
```

```
array([81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
       98, 99])
```

Daten können gespeichert werden:

```
import numpy as np                                                    1
a=np.array(np.arange(1,10).reshape(3,3))                              2
print a                                                               3
#                                                                     4
#[[1, 2, 3],                                                          5
# [4, 5, 6],                                                         6
# [7, 8, 9]])                                                        7
a.tofile('a.data')                                                    8
b=np.fromfile('a.data', dtype=int)                                    9
print b                                                               10
#[1, 2, 3, 4, 5, 6, 7, 8, 9])                                         11
```

Statistische Berechnungen

Numpy liefert grosse Zahl nützlicher Funktionen mit, siehe Numpy_Example_List

Mittelwert, Standardabweichung, ...

```
import numpy as np
T=np.array([1.3,4.5,2.8,3.9])
print T.mean(), T.std(), T.var()
```

Korrelationen

Beispiel mit T (Temperatur), P (Druck) und ρ (Dichte):

```
import numpy as np
T=np.array([1.3,4.5,2.8,3.9])
P=np.array([2.7,8.7,4.7,8.2])
print np.corrcoef([T,P])
#[[ 1.          0.98062258]
# [ 0.98062258  1.        ]]
rho=np.array([8.5,5.2,6.9,6.5])
data=np.column_stack([T,P,rho])
print data
# [[ 1.3,  2.7,  8.5],
#  [ 4.5,  8.7,  5.2],
#  [ 2.8,  4.7,  6.9],
#  [ 3.9,  8.2,  6.5]])
print np.corrcoef([T,P,rho])
#[[ 1.          0.98062258 -0.97090288]
# [ 0.98062258  1.         -0.91538464]
# [-0.97090288 -0.91538464  1.        ]]
```

Diagonalwerte einer Matrize:

```
In [27]: a=np.arange(12).reshape(3,4)
In [28]: a
Out[28]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
In [29]: a.diagonal()
Out[29]: array([ 0,  5, 10])
In [30]: a.diagonal(offset=1)
Out[30]: array([ 1,  6, 11])
In [31]: a.diagonal(offset=-1)
Out[31]: array([4, 9])
In [32]: np.diagonal(a)
Out[32]: array([ 0,  5, 10])
```

Standardmatrizen:

```
In [33]: np.ones(3)
Out[33]: array([ 1.,  1.,  1.])
In [34]: np.zeros(3)
Out[34]: array([ 0.,  0.,  0.])
In [35]: np.eye(3)
Out[35]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Gleichungssysteme lösen:

```
In [36]: from numpy.linalg import inv
In [37]: a=np.array([[3,1,5],[1,0,8],[2,1,4]])
In [38]: a
Out[38]:
array([[3, 1, 5],
       [1, 0, 8],
       [2, 1, 4]])
In [39]: inva=inv(a)
In [40]: inva
Out[40]:
array([[ 1.14285714, -0.14285714, -1.14285714],
       [-1.71428571, -0.28571429,  2.71428571],
       [-0.14285714,  0.14285714,  0.14285714]])

In [41]: np.dot(a,inva)
Out[41]:
array([[  1.00000000e+00,   5.55111512e-17,   1.38777878e-16],
       [  0.00000000e+00,   1.00000000e+00,   0.00000000e+00],
```

```
       [ -1.11022302e-16,   0.00000000e+00,   1.00000000e+00]])
In [42]: from numpy.linalg import solve
In [43]: a
Out[43]:
array([[3, 1, 5],
       [1, 0, 8],
       [2, 1, 4]])
In [45]: b=np.array([6,7,8])
In [46]: x=solve(a,b)
In [47]: x
Out[47]: array([-3.28571429,  9.42857143,  1.28571429])
In [48]: np.dot(a,x)
Out[48]: array([ 6.,  7.,  8.])
```

Nullstellen eines Polynom mit `roots` bestimmen:

```
In [48]: import numpy as np
In [49]: a=np.array([1,-6,-13,42])
In [50]: np.roots(a)
Out[50]: array([ 7., -3.,  2.]
```

Lineare Algebra

Einzelwertzerlegung: $A = U\sigma V$

```
In [67]: import numpy as np
In [68]: A=np.array([[1.,3.,5.],[2.,4.,6.]])
In [69]: U,sigma,V=np.linalg.svd(A)
In [70]: print U
-------> print(U)
[[-0.61962948 -0.78489445]
 [-0.78489445  0.61962948]]
In [71]: print sigma
-------> print(sigma)
[ 9.52551809  0.51430058]
In [72]: print V
-------> print(V)
[[-0.2298477  -0.52474482 -0.81964194]
 [ 0.88346102  0.24078249 -0.40189603]
 [ 0.40824829 -0.81649658  0.40824829]]

In [74]: Sigma = np.zeros_like(A)
```

```
In [75]: n=min(A.shape)
In [76]: sigma[:n,:n]=np.diag(Sigma)
In [77]: sigma
Out[77]:
array([[ 9.52551809,  0.        ,  0.        ],
       [ 0.        ,  0.51430058,  0.        ]])
In [78]: print np.dot(U,np.dot(Sigma,V))
-------> print(np.dot(U,np.dot(Sigma,V)))
[[ 1.  3.  5.]
 [ 2.  4.  6.]]
```

5.3 Matplotlib (pylab)

Mit `matplotlib` können professionelle 2D-Graphen aus z.B. `scipy` und `numpy` Daten erstellt werden.

Schöne Beispiele gibt es unter: http://scipy-cookbook.readthedocs.org/items/idx_matplotlib_simple_plotting.html

`matplotlib` hat folgende Eigenschaften:

- Interaktive und programmatische Benutzung
- Speichern der erzeugten Graphen in PS, EPS, SVG, PNG, ...
- interaktiver Viewer

Beim Arbeiten mit *Jupyter notebooks* hat man die Wahl wo die Grafiken/Plots angezeigt werden sollen:

- Direkt im Notebook: default bzw. Direktive `% matplotlib inline`
- Separates Grafikfenster: Direktive `% matplotlib qt`, damit Zoom Funktion, etc.

Einlesen und Darstellen von Daten

Folgendes Beispiel demonstriert, wie man einfache Daten aus der Datei `numbers.dat` einliest und in einem Diagramm darstellt:

```
#!/usr/bin/env python
import pylab as pl
if __name__ == '__main__':
    # read in data to list
    data = pl.loadtxt('numbers.dat')
    # create x data
    x = range(0,len(data))
    # create empty figure
    pl.figure(1,figsize=(6,4))
    pl.xlabel('x')
    pl.ylabel('numbers')
    # plot data with data points
    pl.plot(x,data,'.')
    pl.title('Content of numbers.dat')
    # save figure to PNG file
    pl.savefig('numbers.png',dpi=72)
```

```
# show canvas                                                    17
pl.show()                                                        18
```

Eine einfache Funktion darstellen:

```
#!/usr/bin/env python
"""
Example: simple line plot.
Show how to make and save a simple line plot with labels, title and grid
"""
import numpy as np
import pylab as pl
#%matplotlib inline
#%matplotlib qt # separate window
t = np.arange(0.0, 1.0+0.01, 0.01)
s = np.cos(2*2*np.pi*t)
pl.plot(t, s)
pl.xlabel('time (s)')
pl.ylabel('voltage (mV)')
pl.title('About as simple as it gets, folks')
pl.grid(True)
pl.savefig('simple_plot')
pl.show()
```

Ein Histogram mit Fit-Funktion:

```
#!/usr/bin/env python
import numpy as np
import pylab as pl
mu, sigma = 100, 15
x = mu + sigma*np.random.randn(10000)
# the histogram of the data
n, bins, patches = pl.hist(x, 50, normed=1, facecolor='green', alpha=0.75)
# add a 'best fit' line
y = pl.normpdf( bins, mu, sigma)
l = pl.plot(bins, y, 'r--', linewidth=1)
pl.xlabel('Smarts')
pl.ylabel('Probability')
pl.title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
pl.axis([40, 160, 0, 0.03])
pl.grid(True)
pl.show()
```

Viele weitere Beispiele zu Matplotlib in Tutorial:

http://www.labri.fr/perso/nrougier/teaching/matplotlib

5.4 SciPy

`scipy` ist eine Erweiterung von `numpy` mit vielen Bibilotheken für numerische Berechnungen, wie z.B. Optimierung, Numerische Integration, Statistik usw.

Zahlreiche Beispiele gibt es z.B. in **http://scipy-cookbook.readthedocs.io/index.html**

Wir diskutieren einige Beispiele zu Differentialgleichung, Regression und Fitten sowie Fourier-Transformation.

Differentialgleichung lösen

Viele Systeme in Physik und Science generell werden durch Differentialgleichungen (DGL) beschrieben. Analytische Lösungen gibt es nur für wenige Spezial–Fälle, meist sind numerische Verfahren nötig.

SciPy stellt dafür (u.a.) die Funktion `odeint` zur Verfügung:

```
from scipy.integrate import odeint
```

`odeint` kann zunächst nur DGL 1. Ordnung lösen, klassisches Physik-Beispiel sind die Bewegungsgleichungen, das sind DGL 2. Ordnung:

$$\frac{d^2y}{dt^2} = F(y,t)$$

Das lässt sich mit Standard-Trick umgehen, man kann DGL 2. Ordnung immer auf System gekoppelter DGL 1. Ordnung umschreiben mit:

$$y_1 = y, \quad y_2 = \frac{dy_1}{dt}$$

kommat man zu zwei DGL 1. Ordnung:

$$\frac{dy_1}{dt} = y_2, \quad \frac{dy_2}{dt} = F(y_1,t)$$

`odeint` wird so gerufen:

```
y = odeint( F, y0, t)
```

mit

F – Eine Python Funktion `F(yi,ti)`, die input-array `yi` und Skalar `ti` erhält und array mit den 1. Ableitungen zurückgibt.

y0 – 1-d array mit Startwerten

t – Array mit t-Werten für die `y` berechnet werden soll

y – Ergebnis Array mit Werten `y(t)`

Konkretes Beispiel für freien Fall:

```
# example for solving simple differential equation for free falling object
# using odeint
#
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
def F(y, t):
        """
```

```
        Return derivatives for 2nd order ODE y'' = g.                    9
        """                                                              10
        g = 9.81                                                         11
        dy = [0, 0]                          # preallocate list to store derivatives   12
        dy[0] = y[1]                  # first derivative of y(t)         13
        dy[1] = -g                    # second derivative of y(t)        14
        return dy                                                        15
# array of time values to study                                          16
t_min = 0; t_max = 2; dt = 0.02                                          17
t = np.arange(t_min, t_max+dt, dt)                                       18
# initial conditions                                                     19
y0 = (10.0, 0.0)                                                         20
# get series of points                                                   21
# y[:,0] corresponds to position (=height)                               22
# y[:,1] corresponds to velocity                                         23
y = odeint(F, y0, t)                                                     24
# display result                                                         25
plt.figure()          # Height: Create figure; then, add plots.          26
plt.plot(t, y[:, 0], linewidth=2)                                        27
skip = 5                                                                 28
```

```
t_test = t[::skip]          # compare at a subset of points                    29
plt.plot(t_test, y0[0]-0.5*9.81*t_test**2, 'bo') # exact solution for y0 = (1,0) 30
plt.figure()          # Velocity Create figure; then, add plots.               31
plt.plot(t, y[:, 1], linewidth=2)                                              32
plt.plot(t_test, y0[1]-9.81*t_test, 'bo') # exact solution for y0 = (1,0)      33
plt.show()                                                                     34
```

Lineare Regression:

```
#!/usr/bin/env python
import scipy as sp
import pylab as pl
#Linear regression example
# This is a very simple example of using two scipy tools
# for linear regression, polyfit and stats.linregress
#Sample data creation
#number of points
n=50
t=sp.linspace(-5,5,n)
#parameters
a=0.8; b=-4
x=sp.polyval([a,b],t)
#add some noise
xn=x+sp.randn(n)
#Linear regressison -polyfit - polyfit can be used other orders polys
(ar,br)=sp.polyfit(t,xn,1)
xr=sp.polyval([ar,br],t)
```

```
#compute the mean square error
err=sp.sqrt(sum((xr−xn)**2)/n)
print('Linear regression using polyfit')
print('parameters: a=%.2f b=%.2f \nregression: a=%.2f b=%.2f, ms error= %.3f' % (a,b,ar,br,err))
#matplotlib ploting
pl.title('Linear Regression Example')
pl.plot(t,x,'g.--')
pl.plot(t,xn,'k.')
pl.plot(t,xr,'r.-')
pl.legend(['original','plus noise', 'regression'])
pl.show()
#Linear regression using stats.linregress
(a_s,b_s,r,tt,stderr)=sp.stats.linregress(t,xn)
print('Linear regression using sp.stats.linregress')
print('parameters: a=%.2f b=%.2f \nregression: a=%.2f b=%.2f, std error= %.3f' % (a,b,a_s,b_s,stderr))
```

Least Square Fit:

Anpassen von beliebigen (nicht-linearen) Funktionen an gewichtete Punkte (= Mess-Punkte mit Unsicherheit).

1. Beispiel: Breit–Wigner Kurve an gemessene Wirkungsquerschnitte

```
#/usr/bin/env python
import numpy as np
import pylab as pl
from scipy.optimize import curve_fit
def BreitWig( x, m, g, spk ):
    " Breit-Wigner function"
    mw2 = m * m
    gw2 = g * g
    eb2 = x * x
    return( spk * gw2*mw2 / ( ( eb2 - mw2 )**2 + mw2 * gw2 ))
# hadronuc cross-section
xv = np.array([ 88.396, 89.376, 90.234, 91.238, 92.068, 93.080, 93.912 ])
yv = np.array([  6.943, 13.183, 25.724, 40.724, 27.031, 12.273, 6.980 ])
ey = np.array([  0.087, 0.119, 0.178, 0.087, 0.159, 0.095, 0.064 ])
pinit = np.array([ 90., 2., 20.])
out,cov=curve_fit(BreitWig, xv, yv, pinit, ey)
print out
print cov
for i in range(3):
    print "%d  %7.4f +- %7.4f " % ( i, out[i], np.sqrt(cov[i][i]))
```

```
l1='data'                                                                    21
pl.errorbar(xv, yv, yerr=ey, fmt='ko',label=l1)                              22
l2='fit'                                                                     23
bins = np.linspace( 88., 94.5, 500)                                          24
pl.plot(bins,BreitWig(bins,out[0],out[1],out[2]),'r-',lw=2,label=l2)         25
pl.legend()                                                                  26
#pl.yscale('log')                                                            27
pl.show()                                                                    28
```

2. Beispiel: Exponential-Funktion an Histogramm

```
#/usr/bin/env python
import numpy as np
import pylab as pl
from scipy.optimize import curve_fit
# Generate data
#from numpy import random , histogram, arange, sqrt, exp, nonzero
n = 1000; isi = np.random.exponential(0.1 , size=n)
db = 0.01; bins = np.arange(0 ,1.0, db)
h = np.histogram(isi, bins)[0]
eh = np.sqrt(h) # stat error
#eh = np.maximum(1.,np.sqrt(h)) # stat error
#
# Function to be fit
# x - independent variable
# p0, p1 - parameters
fitfunc = lambda x, p0, p1 : p1 * np.exp (- x /p0 )
# Initial values for fit parameters
pinit = np.array([ 0.8, 2. ])
# Hist count less than 4 has poor estimate of the weight
# don't use in the fitting process
idx = np.nonzero(h>4)
out,cov=curve_fit(fitfunc,bins[idx]+0.01/2, h[idx], pinit, eh[idx])
l1='data'
#pl.errorbar(bins[idx],h[idx],yerr=eh[idx],fmt='ko',label=l1)
pl.errorbar(bins[:-1],h,yerr=eh,fmt='ko',label=l1)
```

```
l2='fit'                                                                    26
pl.plot(bins,fitfunc(bins,out[0],out[1]),'r-',lw=2,label=l2)                27
pl.legend()                                                                 28
pl.yscale('log')                                                            29
pl.show()                                                                   30
```

Fouriertransformierte eines Kastenpotentials

```
#!/usr/bin/env python
import pylab as pl
import scipy as sp
import numpy as np
# -100 bis 100 in Schirtten von 0.01 -> 20000 Elemente
x = np.arange(-100,100,0.01)
# np array gefuellt mit 0, Laenge wie x
y = np.zeros_like(x)
# y = 1 wenn -0.5<x < 0.5 -> box of height 1
y [ (x > -0.5) &(x <0.5) ] = 1.0
# figure loeschen
pl.clf()
#
# 2 subplots vertical, plot oben
pl.subplot(2, 1, 1)
pl.plot(x,y)
pl.axis([-1,1,-0.5,2])
z=np.fft.fft(y)
```

```
f=np.fft.fftfreq(len(y),d=0.01)
# 2 subplots vertical, plot unten
pl.subplot(2, 1, 2)
pl.plot(f,np.abs(z))
pl.axis([-5,5,0,100])
pl.show()
```

5.5 Aufgaben

1. **Freier Fall**
   Nochmal Aufgabe 9, Kap 1, aber verwenden Sie `numpy` Arrays und Funktionen/Operationen um die Koordinaten (y,v) für 30 Zeitpunkte von $t = 0..1.5\,s$ zu bestimmen, mit $h_0 = 10\,m$ und $v_0 = 0$.
   Und anschließend die umgekehrte Rechnung, geben Sie 20 y-Werte $y = 10..0\,m$ vor und bestimmen Sie dafür t und v.

2. **Datenanalyse**
   In der Datei `rohr1.dat` finden Sie eine Liste von Messungen der Drahtposition in verschiedenen Atlas Muon-Kammer Rohren. Lesen Sie die Zahlen ein:
   ```
   data = numpy.loadtxt('rohr1.dat')
   ```
   (a) Bestimmen Sie Mittelwert und Standardabweichung (Hinweis: numpy-Funktinen)
   (b) Tragen Sie die Werte in Histogramm ein und Plotten es.
   (c) Lesen Sie analog `(x,y)` Koordinaten der Datei `rohr2.dat`:
   ```
   x,y = numpy.loadtxt('rohr2.dat',unpack=True)
   ```
   Bestimmen Sie für `x` und `y` Mittelwert und Standardabweichung sowie die Korrelation.
   Erstellen Sie `(x,y)-Plot`.
   Lösung: read1.py read2.py

3. **Berechnung von PI**

Erzeugen Sie mit dem Zufallszahlen Modul `numpy.random.random` 1 Millionen Zufalls-Punkte mit x,y-Koordinaten zwischen 0 und 1. Wie groß ist der Anteil der Punkte innerhalb eines Radius von 1 vom Ursprung (0,0) ? Wie groß ist der Fehler zum zu erwartenden Ergebnis ?
Lösung: drop.py pidemo.py

4. **Matplotlib**
   Berechnen Sie die Fouriertransformierte einer Gauss- und einer Delta-Funktions-Verteilung und zeichnen Sie die entsprechenden Verteilungen.
   Lösung: fft_aufg.py

5. **Differentialgleichung lösen**

(a) Erweitern Sie das Beispiel für den freien Fall und führen zusätzlichen Term für Luftreibung ein gemäß:

$$F_R = \frac{1}{2} c_W \rho_{Luft} A v^2$$

Das führt zu einer nicht-linearen DGL, die nur numerisch lösbar ist.

$$y'' = -g + \frac{1}{2} c_W \rho_{Luft} A / m_{fb} \times y'^2$$

Nehmen Sie Werte für Fussball:

```
# Luftwiderstandskraft Fussball
rhol = 1.184 # Luftdichte (kg/m^3)
rfb  =  0.11 # Fussballradius (m)
cwfb =  0.45 # Kugel CW-Wert
mfb  =  0.43 # Masse Fussball (kg)
g = 9.81 # Erdbeschleunigung
v = y[1] # Geschwindigkeit
fR = 0.5 * cwfb * rhol * math.pi * rfb**2 * v**2
```

Lösung: odefreefall2.py

(b) Nächster Schritt wäre die Erweiterung auf ”Schiefen Wurf mit Luftreibung”, also die kombinierte Bewegung in horizontaler und vertikaler Richtung.
Konkrete Frage: Wie weit fliegt Fussball der unter 45 Grad mit 140 km/h geschossen wird?
(Die Aerodynamik des Fussballs ist komplex, siehe schöne Diskussion in http://www.weltderphysik.de/gebiet/leben/fussball/coca-cola-formel/) Lösung: odefreefall3.py

6. **Collatz Vermutung**
   ... ein Klassiker der Zahlentheorie ...

   Konstruiere Zahlenfolge:

   - Beginne mit irgendeiner natürlichen Zahl `n > 0`
   - Ist `n` gerade, so nimm als Nächstes `n/2`
   - Ist `n` ungerade, so nimm als Nächstes `3n + 1`
   - Wiederhole die Vorgehensweise mit der erhaltenen Zahl.

   Die Collatz-Vermutung lautet: *Jede so konstruierte Zahlenfolge mündet in den Zyklus 4, 2, 1, egal, mit welcher natürlichen Zahl* `n >0` *man beginnt.*

   Plotten Sie Zahl der Iterationen bis zur `1` gegen die Start-Zahl.

   Lösung: Collatz.py

6 Kurzeinführung in Linux

6.1 Links zu Linux Tutorials

Einige Tutorials zu Linux gibt es z.B. unter:

- SelfLinux
- UNIX Tutorial for Beginners

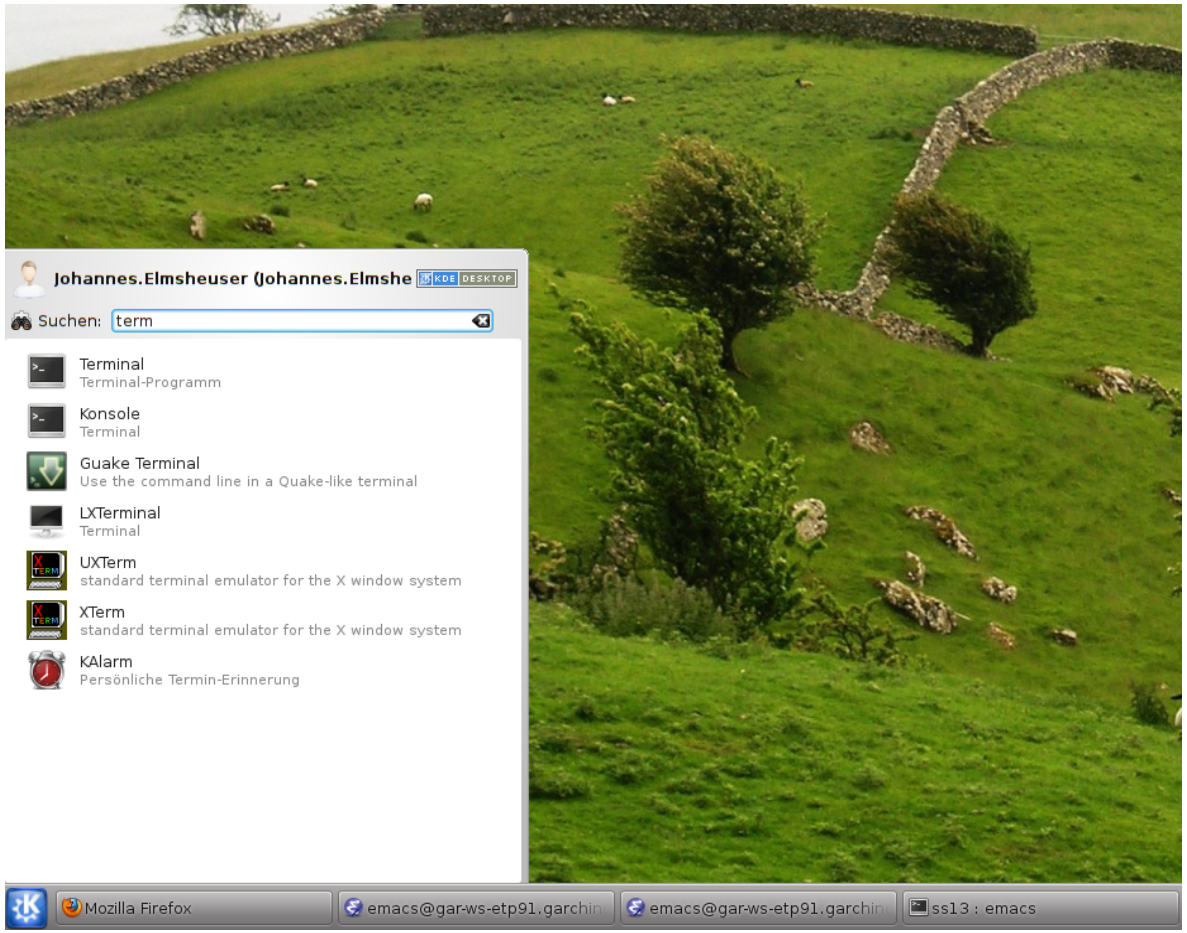
6.2 Die Linux X-Benutzeroberfläche

Die beliebtesten Benutzeroberflächen bzw. Fenstermanager auf Linuxsystemen sind: KDE bzw. GNOME. Diese können auf dem login-screen unter ”Menü” $\rightarrow$ ”Session type” zwischen verschiedenen Fenstermanager aussuchen. Wählen Sie entweder `"KDE"` bzw. `"GNOME Classic"`.

6.3 Terminalfenster starten

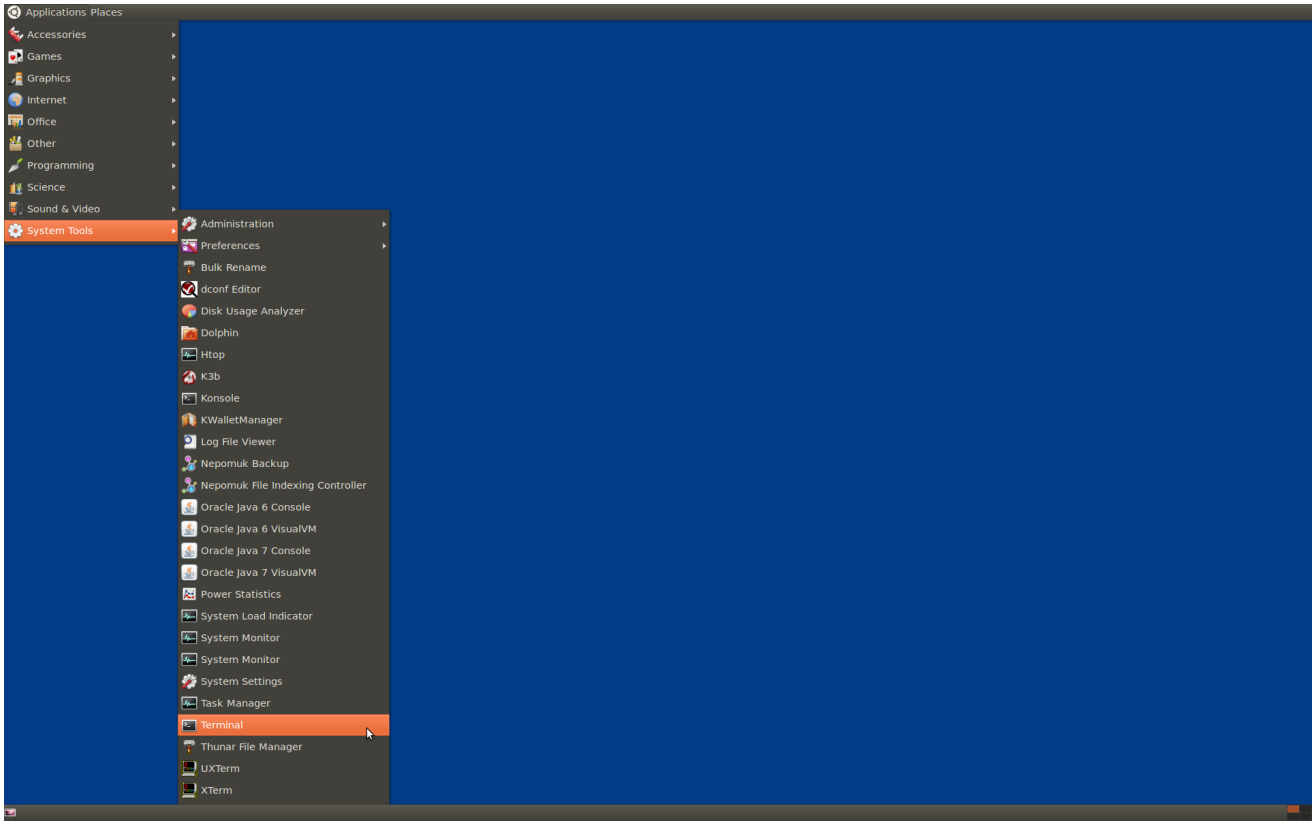
Nachdem Sie eingeloggt sind, starten Sie ein Terminalfenster, mit dem Sie verschiedene Befehle auf der Kommandzeile eingeben können:

- Unter KDE klicken Sie in der rechten unteren Bildschirmecke auf den blauen ”K”-Knopf und tippen in das danach erscheinende Suchfeld des Startmenüs: ”term”. Klicken Sie anschliessend auf den ”Konsole” Menüeintrag und ein Terminalprogramm wird gestartet.

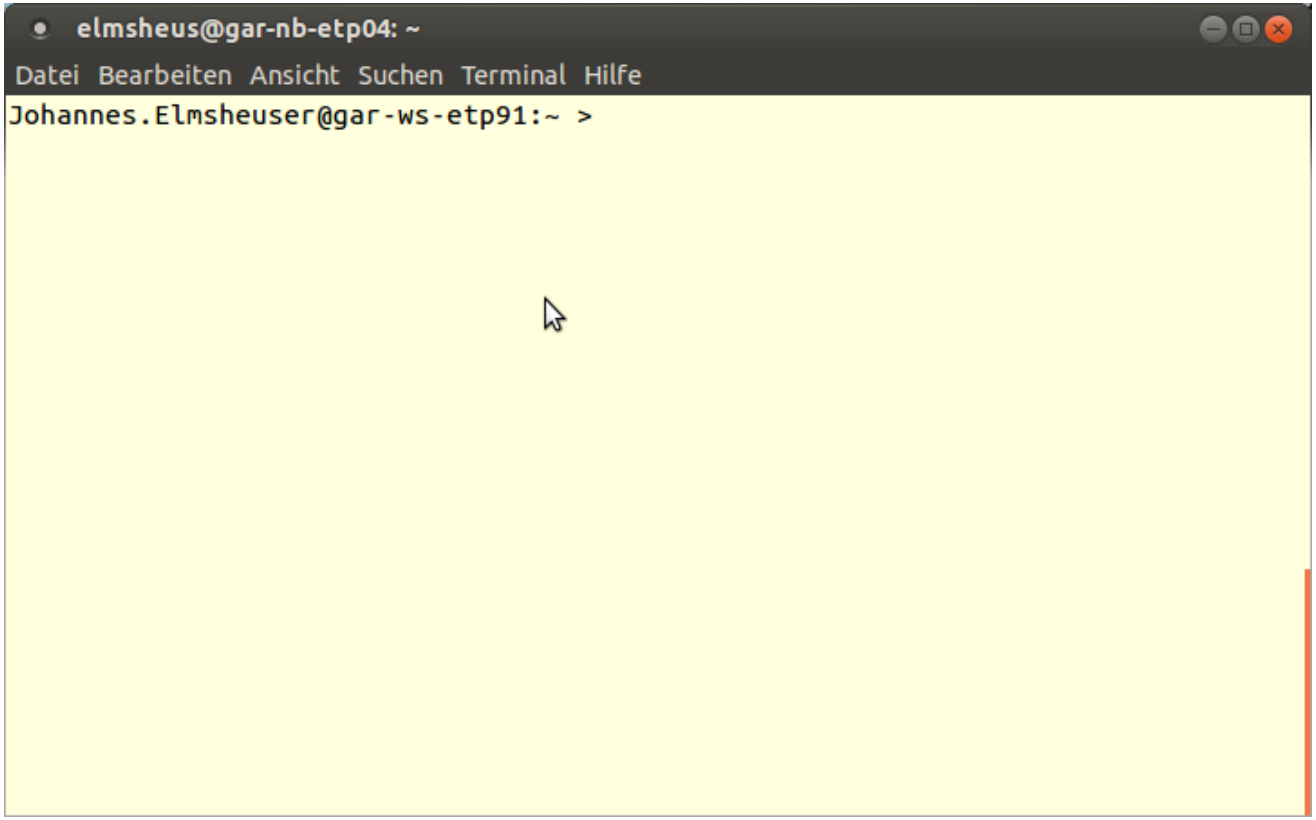


- Unter GNOME classic klicken Sie in der oberen rechten Bildschirmecke auf ”Applications” und

anschliessend im ”System Tools”-Menü auf den Eintrag ”Terminal”:



- Es öffnet sich ein Terminalfenster, in dem Sie Befehle auf der sog. bash Kommandzeile eingeben und ausführen können:



6.4 Befehle im Terminalfenster

Auf der bash Kommandozeile können Befehle eingegeben werden, um Programme zu starten oder z.B. mit dem Dateisystem zu interagieren.

Einfache Befehle:

- Das aktuelle Arbeitsverzeichnis anzeigen:
  pwd
- Den Inhalt des aktuellen Verzeichnis anzeigen:
  ls
- Den Inhalt des aktuellen Verzeichnis als Liste anzeigen:
  ls -l
- Den Inhalt des aktuellen Verzeichnis als Liste mit versteckten Dateien anzeigen:
  ls -al
- Den Inhalt des aktuellen Verzeichnis als Liste sortiert nach Änderungdatum anzeigen:
  ls -rtl
- In das Homeverzeichnis wechseln:
  cd

- Ein neues Verzeichnis anlegen:
  mkdir mycode
- In das neue Verzeichnis wechseln:
  cd mycode

Weitere Befehle:

- Eine leere Datei anlegen:
  touch test.txt
- Eine Datei löschen:
  rm test.txt
- Eine Datei aus dem WWW herunterladen:
  wget http://www.etp.physik.uni-muenchen.de/kurs/Computing/ckurs/source/numbers.dat
- Den Inhalt einer Datei vollständig anzeigen:
  cat numbers.dat
- Den Inhalt einer Datei interaktiv anzeigen (Verlassen mit ”q”, Scrollen mit Pfeiltasten):
  less numbers.dat
- Die Anzahl der Zeilen einer Datei anzeigen:
  wc -l numbers.dat

- Ein leeres Verzeichnis löschen:
  rmdir mytestdir
- Das aktuelle Verzeichnis kann mit "." angesprochen werden:
  ls .
- In das übergeordnete Verzeichnis kann mit ".." angesprochen werden:
  ls ..
- In das übergeordnete Verzeichnis wechseln:
  cd ..
- Eine Datei von einem Verzeichnis in das aktuelle Verzeichnis kopieren:
  cp /path/to/somefile .
- Eine Datei "somefile" vom Verzeichnis "/path/from" in das Verzeichnis "/path/to" kopieren:
  cp /path/from/somefile /path/to/

Programme starten:

- Ein Programm starten Sie einfach durch Eingabe des Befehls auf der Kommandozeile. Dadurch wird die Kommandozeile für weitere Eingaben blockiert. Starten Sie deshalb saemtliche interaktiven Programme wie Editoren etc. immer mit einem zusätzlichem `&` am Ende der Befehlszeile, um die Kommandozeile wieder für neue Befehle freizugeben. Starten Sie den KDE Editor z.B. mit:
  kate &

Befehlseingabe:

- Auf vorher eingegebene Befehle kann mit der Pfeil-nach-oben bzw. Pfeil-nach-unten Taste zugegriffen werden.
- Kommandozeilenvervollständigung: Lange Programmnamen können mit Hilfe der Tabulatortaste vervollständigt werden, d.h. Sie müssen nicht immer lange Programmnamen oder Dateinamen eintippen, sondern brauchen nur die Anfangsbuchstaben eintippen und nach Drücken der Tabulatortasten kann die Befehlszeile vervollständigt werden.

Eingabe-/Ausgabeumleitung:

- Die Ausgabe eines Programms oder eines beliebigen Befehls kann vom Bildschirm des Terminalfensters in eine Datei mit $>$ umgeleitet werden:
  ls -rtl $>$ out.txt
- Die Eingabe in ein Programm kann anstatt von der Tastatur von einer Datei mit $<$ umgeleitet werden:
  cat $<$ numbers.dat

Editoren:

- KDE Editor:
  kate

- GNOME Editor:
  gedit
- Fortgeschrittene Editoren:
  emacs oder **vi**

Entwicklungsumgebungen und Debugger:

- Java, C++, Python Entwicklungsumgebung:
  eclipse
- C++ Entwicklungsumgebung:
  kdevelop
- Qt und C++ Entwicklungsumgebung:
  qtcreator
- Graphischer Debugger:
  ddd

GNU C++ Compiler:

- Ein C++ Programm kompilieren und linken in einem Schritt:
  g++ -o mytest mytest.cpp
- Ein C++ Programm kompilieren:

g++ -c mytest.cpp

- Ein zusammengesetztes C++ Programm kompilieren und linken:
  g++ -o TLVector MyLVector.cpp My3Vector.cpp

Verzeichnisse archivieren:

- Das aktuelle Verzeichnis in eine Datei archivieren und packen:
  tar cvzf myfile.tar.gz .
- Ein Archivdatei in aktuelle Verzeichnis entpacken:
  tar xvzf myfile.tar.gz