

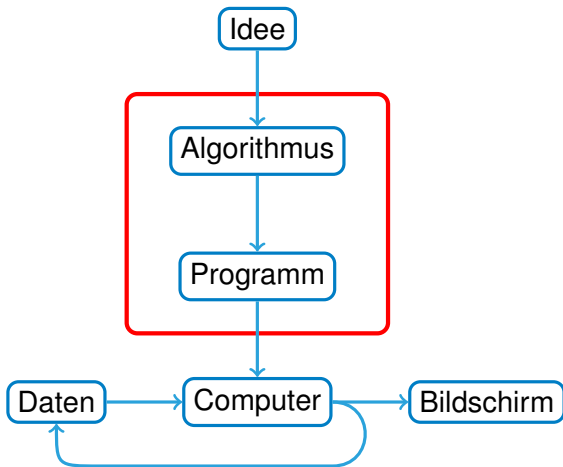
Computegrundlagen Programmieren in Python

F.Uhlig, J.Smiatek, A.Arnold, M.Fyta

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2018/2019

Was ist Programmieren?

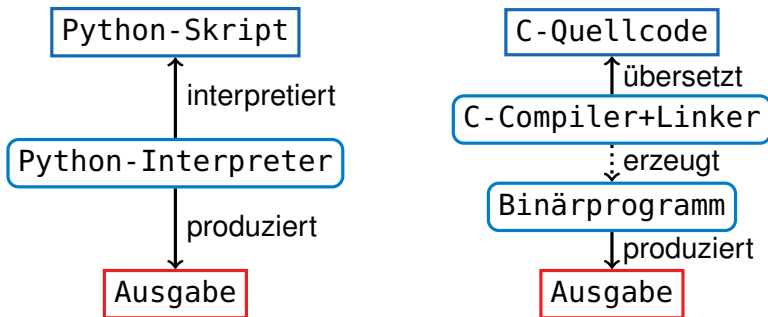


Python



- schnell zu erlernende Programmiersprache
– tut, was man erwartet
- objektorientierte Programmierung ist möglich
- viele Standardfunktionen („all batteries included“)
- breite Auswahl an Bibliotheken
- freie Software mit aktiver Gemeinde
- portabel, gibt es für fast jedes Betriebssystem
- entwickelt von Guido van Rossum, CWI, Amsterdam

Interpretierte- vs. kompilierte Sprachen

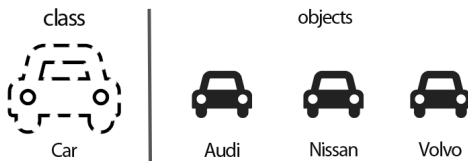


Interpretierte Sprachen

- z.B. Python, Java, C#, Basic, PHP, Shellsprachen
- Das Programm wird vom Interpreter gelesen und ausgeführt
- Es wird kein Maschinencode für den Prozessor übersetzt

Objektorientierte Sprachen

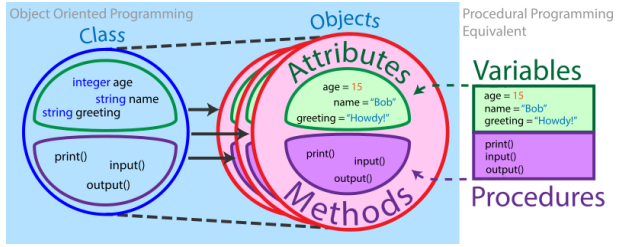
Objekt: Variable, Datenstruktur, etc. (also ein Wert im Speicher der mit einem Kennzeichen identifiziert wird)



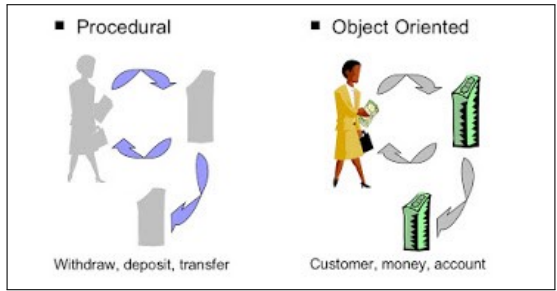
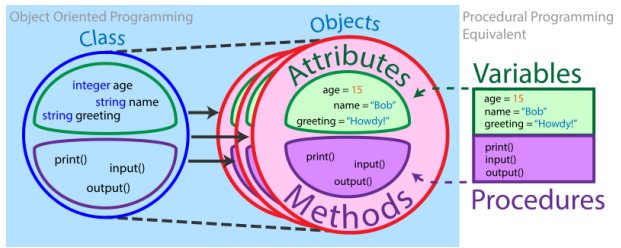
- Objektorientierung (OO) ist ein sprachunabhängiges Programmiermodell
- Ziel ist die bessere Wartbarkeit und Austauschbarkeit von Software durch starke Isolation von Teilproblemen
- Speziell darauf ausgelegt sind z.B. C++, Java, Python,...

Quelle:javatutorial.net

Imperative vs. Objektorientierte Programmierung



Imperative vs. Objektorientierte Programmierung



Informationen zu Python

- aktuelle Versionen 3.7.2
- 2.x ist *noch* weiter verbreitet (z.B. Python 2.7.15rc1 im CIP-Pool)
- diese Vorlesung behandelt daher noch 2.x
- aber längst nicht alles, was Python kann
- Erweiterung über “Python Enhancement Proposals” (PEPs)
<https://www.python.org/dev/peps>

Hilfe zu Python

- offizielle Homepage
<http://www.python.org>
- Einsteigerkurs „A Byte of Python“
<http://swaroopch.com/notes/Python> (englisch)
<http://abop-german.berlios.de> (deutsch)
- mit Programmiererfahrung „Dive into Python“
<http://diveintopython.net>

Wichtige PEPs

PEP 8: Style guide for Python Code

- “[...] code is read much more often than it is written”
- Einheitlichkeit zählt!
 - innerhalb eines Projekts, Moduls, Funktion
 - Einheitlichkeit nie strikt, wenn unsinnig darf gebrochen werden
- leere Zeilen zwischen logisch separierbaren Teilen, für Funktionsdefinitionen
- Zeilenlänge \approx 80 Zeichen
- Blöcke durch Indentierung gegeben (in C: {})
`>>>from __future__ import braces`
- Leerzeichen zur Lesbarkeit verwenden
- <https://www.youtube.com/watch?v=wf-BqAjZb8M>

Wichtige PEPs

PEP 20: The Zen of Python (>>>import this)

- siehe PEP 8 für Schöneheitsempfehlungen
- Vermeidung von Verschachtelungen
- Leerzeichen sind (meist) eine gute Idee
- Fehler niemals ignorieren oder durchgehen lassen
- Raten is schlecht, explizit ist besser als implizit
- immer einen offensichtlich richtigen Weg ein Problem zu lösen
- Problem sollten, wenn möglich immer sofort nach Erkennen behoben werden
- Code simpel halten
 - schwer zu erklärende Implementierung
 - wahrscheinlich eine schlechte Idee
- Namensräume, Namensräume, Namensräume!!!

Namen und Namensräume (in python)

- Namen der Objekte über einer Art Baumstruktur angeordnet und über entsprechenden Pfadnamen eindeutig angesprochen.
- Namensraum (i.e. Klasse): Sammlung von Namen
- Ein Namensraum bildet Namen auf den entsprechenden Objekten
- Unterschiedliche Namensräume können in python existieren, den gleichen Namen haben und komplett isoliert sein.
- In python sind Namensräume über einzelne Module definiert und sind hierarchisch.
- Interpreter erzeugt einen python Namensraum mit allen built-in Namen (weshalb Funktionen wie print() immer verfügbar sind).
- Module erzeugen einen globalen und Funktionen einen lokalen Namensraum.

Alles ist ein Objekt

Python ist eine objektorientierte Programmiersprache

- Objekte sind Pythons Datenabstrahierung
- alle Daten (inklusive Code!) is representiert durch Objekte

Jedes Objekt hat:

- Identität: unveränderlich, denke Adresse im Speicher (**is**, **id**)
- Typ: unveränderlich, bestimmt gültige Operationen für Objekt, mögliche Werte (**type** ← auch ein Objekt)
- Wert: teilweise veränderlich, vom Typ abhängig

Spezielle Methoden:

- *operator overloading, customization*
- definiert das Verhalten im Bezug auf möglichen Sprachoperationen
- Ganzzahldatentyp: 56 spezielle Methoden ??double-check numbers.Integral??

Dynamische Typen

- Objekttyp zur Laufzeit bestimmt (dynamically typed)
 - während Laufzeit möglich anderes Objekt an gleichen Namen zu binden
 - "Duck typing": Wenn es läuft wie eine Ente und quakt wie eine Ente, dann ist es eine Ente. (Objekt-Typ durch Methoden/Attribute und nicht durch seine Klasse beschrieben)
- es wird überprüft, ob eine adequate Methode existiert, die Operator entspricht (double-under (Doppelunterstrich), a.k.a. "dunder", Methoden oder magische Methoden) (sh. Überlastung von Operatoren).

```
>>>jahr=2019  
>>>type(jahr)
```

- Statische Typen: Variable muss vor Start definiert werden

Python starten

Aus der Shell:

```
> python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more...
>>> print "Hello World"
Hello World
>>> help("print")
>>> exit()
```

- >>> markiert Eingaben
- **print**: Ausgabe auf das Terminal
- **help()**: interaktive Hilfe, wird mit "q" beendet
- statt **exit()** reicht auch Control-d
- oder **ipython** mit Tab-Ergänzung, History usw.

Python-Skripte

Als Python-Skript helloworld.py:

```
#!/usr/bin/python
```

```
# unsere erste Python-Anweisung
```

```
print "Hello World"
```

- mit `python helloworld.py` starten
- oder ausführbar machen (`chmod a+x helloworld.py`)
- **Umlaute vermeiden** oder Encoding-Cookie einfügen
- „#!“ funktioniert genauso wie beim Shell-Skript
- Zeilen, die mit „#“ starten, sind Kommentare

**Kommentare sind wichtig,
um ein Programm verständlich machen!**

- und nicht, um es zu verlängern!



Datentypen 1

- ganze Zahlen:

```
>>> print 42
```

```
42
```

```
>>> print -12345
```

```
-12345
```

- Fließkommazahlen:

```
>>> print 12345.000
```

```
12345.0
```

```
>>> print 6.023e23
```

```
6.023e+23
```

```
>>> print 13.8E-24
```

```
1.38e-23
```

- 1.38e-23 steht z. B. für 1.38×10^{-23}
- $12345 \neq 12345.0$ (z. B. bei der Ausgabe)

Datentypen 2

- Zeichenketten (Strings)

```
>>> print "Hello World"
Hello World
>>> print 'Hello World'
Hello World
>>> print """Hello
... World"""
Hello
World
```

- zwischen einfachen (') oder doppelten (") Anführungszeichen
- Über mehrere Zeilen mit dreifachen Anführungszeichen
- Leerzeichen sind normale Zeichen!
"Hello World" \neq "Hello World"
- Zeichenketten sind keine Zahlen! "1" \neq 1

Variablen

```
>>> number1 = 1
>>> number2 = number1 + 5
>>> print number1, number2
1 6
>>> number2 = "Keine Zahl"
>>> print number2
Keine Zahl
```

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

Arithmetische Ausdrücke

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

- mathematische Präzedenz (Exponent vor Punkt vor Strich),
z. B. $2**3 * 3 + 5 \rightarrow 2^3 \cdot 3 + 5 = 29$
- Präzedenz kann durch runde Klammern geändert werden:
 $2**(3 * (3 + 5)) \rightarrow 2^{3 \cdot 8} = 16,777,216$

Logische Ausdrücke

<code>==, !=</code>	Test auf (Un-)Gleichheit, z.B. <code>2 == 2</code> \rightarrow True, <code>1 == 1.0</code> \rightarrow True, <code>2 == 1</code> \rightarrow False
<code><, >, <=, >=</code>	Vergleich, z.B. <code>2 > 1</code> \rightarrow True, <code>1 <= -1</code> \rightarrow False
<code>or, and</code>	Logische Verknüpfungen „oder“ bzw. „und“
<code>not</code>	Logische Verneinung, z.B. <code>not False</code> $==$ True

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

<i>a</i>	<i>b</i>	<i>a</i> und <i>b</i>	<i>a</i> oder <i>b</i>
True	True	True	True
False	True	False	True
True	False	False	True
False	False	False	False

- Präzedenz: Vergleiche vor logischen Verknüpfungen

- Beispiele:

`3 > 2 and 5 < 7` \rightarrow True,
`3 > (2 and 5) < 7` \rightarrow False,
`1 < 1 or 2 >= 3` \rightarrow False

if: bedingte Ausführung

```
>>> a = 1
>>> if a < 5:
...     print "a ist kleiner als 5"
... elif a > 5:
...     print "a ist groesser als 5"
... else:
...     print "a ist 5"
a ist kleiner als 5
>>> if a < 5 and a > 5:
...     print "Das kann nie passieren"
```

- **if-elif-else** führt den **Block** nach der ersten erfüllten Bedingung (logischer Wert True) aus
- Trifft keine Bedingung zu, wird der **else**-Block ausgeführt
- **elif** oder **else** sind optional

Blöcke und Einrückung

```
>>> a=5
>>> if a < 5:
...     # wenn a kleiner als 5 ist
...     b = -1
... else: b = 1
>>> # aber hier geht es immer weiter
... print b
1
```

- Alle *gleich eingerückten* Befehle gehören zum Block
- Nach dem **if**-Befehl geht es auf Einrückungsebene des **if** weiter, egal welcher **if**-Block ausgeführt wurde
- Einzeilige Blöcke können auch direkt hinter den Doppelpunkt
- Einrücken durch Leerzeichen oder Tabulatoren (einfacher)

Blöcke und Einrückung 2

- ein Block kann nicht leer sein, aber der Befehl **pass** tut nichts:

```
if a < 5:  
    pass  
else:  
    print "a ist groesser gleich 5"
```

- **IndentationError** bei ungleichmäßiger Einrückung:

```
>>> print "Hallo"  
Hallo  
>>>     print "Hallo"  
File "<stdin>", line 1  
    print "Hallo"  
    ^
```

IndentationError: unexpected indent

- Falsche Einrückung führt im allgemeinen zu Programmfehlern!

while: Schleifen

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

- Führt den Block solange aus, wie die Bedingung wahr ist
 - kann auch nicht ausgeführt werden:
-

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

for: Sequenz-Schleifen

```
>>> for a in range(1, 3): print a
1
2
>>> b = 0
>>> for a in range(1, 100):
...     b = b + a
>>> print b
4950
>>> print 100 * (100 - 1) / 2
4950
```

- **for** führt einen Block für jedes Element einer **Sequenz** aus
- Das aktuelle Element steht in **a**
- **range**(k, l) ist eine Liste der Zahlen a mit $k \leq a < l$
- später lernen wir, Listen zu erstellen und verändern

break und continue: Schleifen beenden

```
>>> for a in range(1, 10):
...     if a == 2 or a == 4 or a == 6: continue
...     elif a == 5: break
...     print a
1
3
>>> a = 1
>>> while True:
...     a = a + 1
...     if a > 5: break
>>> print a
6
```

- beide überspringen den Rest des Schleifenkörpers
- **break** bricht die Schleife ganz ab
- **continue** springt zum Anfang

Funktionen

```
>>> def printPi():  
...     print "pi ist ungefaehr 3.14159"  
>>> printPi()  
pi ist ungefaehr 3.14159
```

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

-
- eine Funktion kann beliebig viele Argumente haben
 - Argumente sind Variablen der Funktion
 - Beim Aufruf bekommen die Argumentvariablen Werte in der Reihenfolge der Definition
 - Der Funktionskörper ist wieder ein Block

return: eine Funktion beenden

```
def printMax(a, b):  
    if a > b:  
        print a  
        return  
    print b
```

- **return** beendet die Funktion sofort

Rückgabewert

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

- eine Funktion kann einen Wert zurückliefern
- der Wert wird bei **return** spezifiziert

Lokale Variablen

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

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

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

Vorgabewerte und Argumente benennen

```
>>> def lj(r, epsilon = 1.0, sigma = 1.0):  
...     return 4*epsilon*( (sigma/r)**6 - (sigma/r)**12 )  
>>> print lj(2**(1./6.))  
1.0  
>>> print lj(2**(1./6.), 1, 1)  
1.0
```

- Argumentvariablen können mit Standardwerten vorbelegt werden
 - diese müssen dann beim Aufruf nicht angegeben werden
-

```
>>> print lj(r = 1.0, sigma = 0.5)  
0.0615234375  
>>> print lj(epsilon=1.0, sigma = 1.0, r = 2.0)  
0.0615234375
```

- beim Aufruf können die Argumente auch explizit belegt werden
- dann ist die Reihenfolge egal

Dokumentation von Funktionen

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

```
def min(a, b):  
    ""
```

Gibt das Minimum von a und b zurueck. Funktioniert
ansonsten genau wie die Funktion max.

```
    ""  
    if a < b: return a  
    else:     return b
```

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

Funktionen als Werte

```
def printGrid(f, a, b, step):  
    """  
    Gibt x, f(x) an den Punkten  
    x= a, a + step, a + 2*step, ..., b aus.  
    """  
    x = a  
    while x < b:  
        print x, f(x)  
        x = x + step  
  
def test(x): return x*x  
  
printGrid(test, 0, 1, 0.1)
```

- Funktionen ohne Argumentliste „(..)“ sind normale Werte
- Funktionen können in Variablen gespeichert werden
- ... oder als Argumente an andere Funktionen übergeben werden

Rekursion

```
def fibonacci(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
  
    return fibonacci(n-1) + fibonacci(n-2)
```

- Funktionen können andere Funktionen aufrufen, insbesondere sich selber
- Eine Funktion, die sich selber aufruft, heißt **rekursiv**
- Rekursionen treten in der Mathematik häufig auf
- sind aber oft nicht einfach zu verstehen
- Ob eine Rekursion endet, ist nicht immer offensichtlich
- Jeder rekursive Algorithmus kann auch **iterativ** als verschachtelte Schleifen formuliert werden

Komplexe Datentypen

- Komplexe Datentypen sind zusammengesetzte Datentypen
- Beispiel: Eine Zeichenkette besteht aus beliebig vielen Zeichen
- die wichtigsten komplexen Datentypen in Python:
 - Strings (Zeichenketten)
 - Listen
 - Tupel
 - Dictionaries (Wörterbücher)
- diese können als **Sequenzen** in **for** eingesetzt werden:

```
>>> for x in "bla": print "->", x
-> b
-> l
-> a
```

Listen

```

>>> kaufen = [ "Muesli", "Milch", "Obst" ]
>>> kaufen[1] = "Milch"
>>> print kaufen[-1]
Obst
>>> kaufen.append(42)
>>> del kaufen[-1]
>>> print kaufen
['Muesli', 'Milch', 'Obst']
  
```

- komma-getrennt in eckigen Klammern
- können Daten *verschiedenen* Typs enthalten
- `liste[i]` bezeichnet das *i*-te Listenelement, negative Indizes starten vom Ende
- `liste.append()` fügt ein Element an eine Liste an
- **del** löscht ein Listenelement

Listen 2

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

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

Shallow copies

Shallow copy:

```
>>> bezahlen = kaufen
>>> del kaufen[2:]
>>> print bezahlen
['Muesli', 'Milch']
```

Subliste, deep copy:

```
>>> merken = kaufen[1:]
>>> del kaufen[2:]
>>> print merken
['Milch', 'Obst', 'Oel', 'Mehl']
```

„=" macht in Python flache Kopien komplexer Datentypen!

- Flache Kopien (shallow copies) verweisen auf dieselben Daten
- Änderungen an einer flachen Kopie betreffen auch das Original
- Sublisten sind echte Kopien
- daher ist `l[:]` eine echte Kopie von `l`

Shallow copies 2 – Listenelemente

```
>>> elementliste = []
>>> liste = [ elementliste, elementliste ]
>>> liste[0].append("Hallo")
>>> print liste
[['Hallo'], ['Hallo']]
```

Mit echten Kopien (deep copies)

```
>>> liste = [ elementliste[:], elementliste[:] ]
>>> liste[0].append("Welt")
>>> print liste
[['Hallo', 'Welt'], ['Hallo']]
```

- komplexe Listenelemente sind flache Kopien und können daher mehrmals auf dieselben Daten verweisen
- kann zu unerwarteten Ergebnissen führen

Tupel: unveränderbare Listen

```
>>> kaufen = ("Muesli", "Kaese", "Milch")
>>> print kaufen[1]
Kaese
>>> for f in kaufen[:2]: print f
Muesli
Kaese
>>> kaufen[1] = "Camembert"
TypeError: 'tuple' object does not support item assignment
>>> print k + k
('Muesli', 'Kaese', 'Milch', 'Muesli', 'Kaese', 'Milch')
```

- komma-getrennt in runden Klammern
- können nicht verändert werden
- ansonsten wie Listen einsetzbar
- Strings sind Tupel von Zeichen

Dictionaries

```
>>> de_en = { "Milch": "milk", "Mehl": "flour" }
>>> print de_en
{'Mehl': 'flour', 'Milch': 'milk'}
>>> de_en["Oel"]="oil"
>>> for de in de_en: print de, "=>", de_en[de]
Mehl => flour
Milch => milk
Oel => oil
>>> for de, en in de_en.iteritems(): print de, "=>", en
>>> if "Mehl" in de_en: print "Kann \"Mehl\" uebersetzen"
```

- komma-getrennt in geschweiften Klammern
- speichert Paare von Schlüsseln (Keys) und Werten
- Speicher-Reihenfolge der Werte ist nicht festgelegt
- daher Indizierung über die Keys, nicht Listenindex o.ä.
- mit **in** kann nach Schlüsseln gesucht werden

Formatierte Ausgabe: der %-Operator

```
>>> print "Integer %d %05d" % (42, 42)
Integer 42 00042
>>> print "Fließkomma %e |%+8.4f| %g" % (3.14, 3.14, 3.14)
Fließkomma 3.140000e+00 | +3.1400| 3.14
>>> print "Strings %s %10s" % ("Hallo", "Welt")
Strings Hallo      Welt
```

- Der %-Operator ersetzt %-Platzhalter in einem String
- %d: Ganzzahlen (Integers)
- %e, %f, %g: Fließkomma mit oder ohne Exponent oder wenn nötig (Standardformat)
- %s: einen String einfügen
- %x[defgs]: auf x Stellen mit Leerzeichen auffüllen
- %0x[defg]: mit Nullen auffüllen
- %x.y[efg]: x gesamt, y Nachkommastellen

Objekte in Python

```
>>> original = list()
>>> original.append(3)
>>> original.append(2)
```

```
>>> kopie = list(original)
```

```
>>> original.append(1)
>>> original.sort()
```

```
>>> print original, kopie
[1, 2, 3], [3, 2]
```

- In Python können komplexe Datentypen wie Objekte im Sinne der **objekt-orientierten Programmierung** verwendet werden
- Datentypen entsprechen **Klassen** (hier **list**)
- Variablen entsprechen **Objekten** (hier **original** und **kopie**)
- Objekte werden durch durch Aufruf von Klasse() erzeugt
- **Methoden** eines Objekts werden in der Form Objekt.Methode() aufgerufen (hier **list.append()** und **list.sort()**)
- **help(Klasse/Datentyp)** informiert über vorhandene Methoden
- Per **class** kann man selber Klassen erstellen

Stringmethoden

- Zeichenkette in Zeichenkette suchen
"Hallo Welt".**find**("Welt") → 6
"Hallo Welt".**find**("Mond") → -1
- Zeichenkette in Zeichenkette ersetzen
"abcdabcabe".**replace**("abc", "123") → '123d123abe'
- Groß-/Kleinschreibung ändern
"hallo Welt".**capitalize**() → 'Hallo welt'
"Hallo Welt".**upper**() → 'HALLO WELT'
"Hallo Welt".**lower**() → 'hallo welt'
- in eine Liste zerlegen
"1, 2, 3, 4".**split**(",") → ['1', ' 2', ' 3', ' 4']
- zuschneiden
" Hallo ".**strip**() → 'Hallo'
"..Hallo..".**rstrip**(".") → 'Hallo..'

Ein-/Ausgabe: Dateien in Python

```
eingabe = open("ein.txt")
ausgabe = open("aus.txt", "w")
nr = 0
ausgabe.write("Datei %s mit Zeilennummern\n" % eingabe.name)
for zeile in eingabe:
    nr += 1
    ausgabe.write("%d: %s" % (nr, zeile))
ausgabe.close()
```

- Dateien sind mit **open**(datei, mode) erzeugte Objekte
- Mögliche Modi (Wert von mode):

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

- sind Sequenzen von Zeilen (wie eine Liste von Zeilen)
- Nur beim Schließen der Datei werden alle Daten geschrieben

Ein-/Ausgabe: Dateien in Python

```
eingabe = open("ein.txt")
ausgabe = open("aus.txt", "w")
nr = 0
ausgabe.write("Datei %s mit Zeilennummern\n" % eingabe.name)
for zeile in eingabe:
    nr += 1
    ausgabe.write("%d: %s" % (nr, zeile))
ausgabe.close()
```

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

```
l = '3. 4. 1.'
map(float, l.split())
```

Ein-/Ausgabe: Dateien in Python

```
eingabe = open("ein.txt")
ausgabe = open("aus.txt", "w")
nr = 0
ausgabe.write("Datei %s mit Zeilennummern\n" % eingabe.name)
for zeile in eingabe:
    nr += 1
    ausgabe.write("%d: %s" % (nr, zeile))
ausgabe.close()
```

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

Dateien als Sequenzen

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

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

Standarddateien

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

<code>sys.stdin</code>	Eingabe (etwa Tastatur)
<code>sys.stdout</code>	Standard-Ausgabe
<code>sys.stderr</code>	Fehler-Ausgabe

```
import sys
```

```
line = sys.stdin.readline()
```

```
sys.stderr.write("don't know what to do with {}\n".format(line))
```

```
for i in range(10):
```

```
    sys.stdout.write("{},".format(i))
```

```
sys.stdout.write("\n")
```

Ausgabe:

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


Ein-/Ausgabe mittels `raw_input`

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

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

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

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

Umlaute — Encoding-Cookie

```
#!/usr/bin/python
# encoding: utf-8
# Zufällige Konstante  $\alpha$ 
alpha = 0.5
#  $\alpha^2$  ausgeben
print "Mir dünkt, dass  $\alpha^2 = \{:g\}$ ".format(alpha**2)
```

Ausgabe:

Mir dünkt, dass $\alpha^2 = 0.25$

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

Fehlermeldungen: raise

```
def loesungen_der_quad_gln(a, b, c):  
    "loest  $a x^2 + b x + c = 0$ "  
    det = (0.5*b/a)**2 - c  
    if det < 0: raise Exception("Es gibt keine Loesung!")  
    return (-0.5*b/a + det**0.5, -0.5*b/a - det**0.5)  
  
try:  
    loesungen_der_quad_gln(1,0,1)  
except:  
    print "es gibt keine Loesung, versuch was anderes!"
```

- **raise** Exception("Meldung") wirft eine Ausnahme (Exception)
- Funktionen werden nacheinander an der aktuellen Stelle beendet
- mit **try: ... except: ...** lassen sich Fehler abfangen, dann geht es im **except**-Block weiter


 Institute for
 Computational
 Physics

Module

```

>>> import sys
>>> print "program name is {}".format(sys.argv[0])
program name is ""
>>> from random import random
>>> print random()
0.296915031568
  
```

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



Das sys-Modul

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

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

argparse-Modul: Parameter in Python 2.7

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

- Einlesen von Kommandozeilenflags
- add_argument spezifiziert Parameter
 - kurzer + langer Name („-f“, „--file“),
ohne Minus positionsabhängiger Parameter
 - dest: Zielvariable für den vom Benutzer gegebenen Wert
 - **type**: geforderter Datentyp (**type**="int")
- Bei Aufruf python parse.py -f test 1 ist
args.filename = 'test', args.number = 1
- python parse.py -f test a gibt Fehler, da „a“ keine Zahl

math- und random-Modul

```

import math
import random
def boxmueller():
    """
    liefert normalverteilte Zufallszahlen
    nach dem Box-Mueller-Verfahren
    """
    r1, r2 = random.random(), random.random()
    return math.sqrt(-2*math.log(r1))*math.cos(2*math.pi*r2)
  
```

- math stellt viele mathematische Grundfunktionen zur Verfügung, z.B. **floor/ceil**, **exp/log**, **sin/cos**, **pi**
- random erzeugt *pseudozufällige* Zahlen
 - **random**(): gleichverteilt in [0, 1)
 - **randint**(a, b): gleichverteilt ganze Zahlen in [a, b)
 - **gauss**(m, s): normalverteilt mit Mittelwert *m* und Varianz *s*

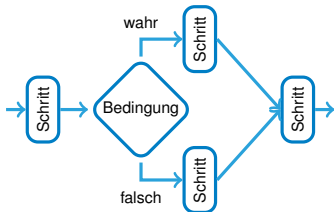
os-Modul: Betriebssystemfunktionen

```
import os
import os.path
# Datei in Verzeichnis "alt" verschieben
dir = os.path.dirname(file)
name = os.path.basename(file)
altdir = os.path.join(dir, "alt")
# Verzeichnis "alt" erstellen, falls es nicht existiert
if not os.path.isdir(altdir): os.mkdir(altdir)
# Verschieben, falls nicht schon existent
newpath = os.path.join(altdir, name)
if not os.path.exists(newpath): os.rename(file, newpath)
```

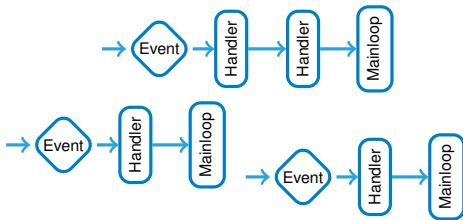
- betriebssystemunabhängige Pfadtools im Untermodul `os.path`:
z.B. `dirname`, `basename`, `join`, `exists`, `isdir`
- `os.system`: Programme wie von der Shell aufrufen
- `os.rename/os.remove`: Dateien umbenennen / löschen
- `os.mkdir/os.rmdir`: erzeugen / entfernen von Verzeichnissen

Graphical User Interfaces (GUI)

Klassisch



Eventgetrieben (GUI)



- Klassisch: Programm bestimmt, was passiert
- Warten auf Benutzereingaben
- GUI: Programm reagiert auf Eingaben (Events)
- Hauptschleife ruft Codestücke (Handler) auf
- Reihenfolge durch Benutzereingaben oder Timer bestimmt
- Programm muss Datenkonsistenz sicherstellen

Das Tkinter-Modul

```
import Tkinter  
# main window and connection to Tk  
root = Tkinter.Tk()  
root.title("test program")  
def quit():  
    print text.get()  
    root.quit()  
# text input  
text = Tkinter.Entry(root) ; text.pack()  
# button ending the program  
end = Tkinter.Button(root, text = "Quit", command = quit)  
end.pack({"side": "bottom"})  
root.mainloop()
```

- bietet Knöpfe, Textfenster, Menüs, einfache Graphik usw.
- mit `Tk.mainloop` geht die Kontrolle an das Tk-Toolkit
- danach eventgesteuertes Programm



Numerik mit Python – numpy

- numpy ist ein Modul für effiziente numerische Rechnungen
- Nicht fester Bestandteil von Python, aber Paket in allen Linux-Distributionen
- Alles nötige unter <http://numpy.scipy.org>
- Bietet unter anderem
 - mathematische Grundoperationen
 - Sortieren, Auswahl von Spalten, Zeilen usw.
 - Eigenwerte, -vektoren, Diagonalisierung
 - diskrete Fouriertransformation
 - statistische Auswertung
 - Zufallsgeneratoren
- Wird bei `ipython --pylab` automatisch unter dem Kürzel `np` geladen

np.ndarray – n -dimensionale Arrays

```
>>> import numpy as np
>>> A = np.identity(2)
>>> print A
[[ 1.  0.]
 [ 0.  1.]]
>>> v = np.zeros(5)
>>> print v
[ 0.  0.  0.  0.  0.]
>>> print type(A), type(v)
<type 'numpy.ndarray'> <type 'numpy.ndarray'>
```

- NumPy basiert auf n -dimensionalem Array `numpy.ndarray`
- Technisch zwischen Array und Tupel
 - Kein `append/remove`
 - Aber elementweiser lesender und schreibender Zugriff
 - Alle Elemente vom selben (einfachen) Datentyp
- Entspricht mathematischen Vektoren, Arrays, Tensoren, ...

Eindimensionale Arrays – Vektoren

```
>>> import numpy as np
>>> print np.array([1.0, 2, 3])
[ 1.,  2.,  3.]
>>> print np.zeros(2)
[ 0.,  0.]
>>> print np.ones(5)
[ 1.,  1.,  1.,  1.,  1.]
>>> print np.arange(2.2, 3, 0.2)
[ 2.2,  2.4,  2.6,  2.8]
```

- `np.array` erzeugt ein `ndarray` aus einer (geschachtelten) Liste
- `np.arange` entspricht **range** für Fließkomma
- `np.zeros/ones` erzeugen 0er/1er-Arrays

Mehrdimensionale Arrays

```
>>> print np.array([[1, 2, 3], [4, 5, 6]])
array([[1, 2, 3],
       [4, 5, 6]])
>>> print np.array([[[1,2,3], [4,5,6]], [[7,8,9], [0,1,2]]])
array([[[1, 2, 3],
       [4, 5, 6]],
       [[7, 8, 9],
       [0, 1, 2]]])
>>> print np.zeros((2, 2))
array([[ 0.,  0.],
       [ 0.,  0.]])
```

- Mehrdimensionale Arrays entsprechen verschachtelten Listen
- Alle Zeilen müssen die gleiche Länge haben
- `np.zeros/ones`: Größe als Tupel von Dimensionen

Array-Informationen

```
>>> v = np.zeros(4)
>>> print v.shape
(4,)
>>> I = np.identity(2)
>>> print I.shape
(2, 2)
>>> print I.dtype
float64
```

- `array.shape` gibt die Größen der Dimensionen als Tupel zurück
- Anzahl der Dimensionen (Vektor, Matrix, ...) ergibt sich aus Länge des Tupels
- `array.dtype` gibt den gemeinsamen Datentyp aller Elemente
- Wichtig beim Debuggen (Warum passen die Matrix und der Vektor nicht zusammen?)

Elementzugriff

```
>>> a = np.array([[1,2,3,4,5,6], [7,8,9,0,1,2]])
>>> print a[0]
[1 2 3 4 5 6]
>>> print a[1]
[7 8 9 0 1 2]
>>> print a[1,2]
9
```

- [] indiziert Zeilen und Elemente usw.
- Anders als bei Pythonlisten können geschachtelte Indizes auftreten (wie bei Matrizen)
- `a[1,2]` entspricht mathematisch $a_{2,3}$ wegen der verschiedenen Zählweisen (ab 0 bzw. ab 1)
- Achtung: in der Mathematik bezeichnet a_1 meist einen *Spaltenvektor*, hier `a[1]` einen *Zeilenvektor*
- Es gilt wie in der Mathematik: **Z**eilen **z**uerst, **S**palten **s**päter!

Subarrays

```
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> print a[1:,1:]
[[5 6]
 [8 9]]
>>> print a[:,2]
[3 6 9]
```

- Wie bei Listen lassen sich auch Bereiche wählen, in allen Dimensionen
- `a[1:, 1:]` beschreibt die 1,1-Untermatrix, also ab der 2. Zeile und Spalte
- `a[:, 2]` beschreibt den 3. Spaltenvektor
- Achtung, dies sind immer *flache* Kopien!

Flache und tiefe Kopien von ndarray

```
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> b = a.copy()           # tiefe Kopie
>>> a[:,0] = np.zeros(3) # flache Kopie, ändert a
>>> print a
[[0 2 3]
 [0 5 6]
 [0 8 9]]
>>> print b
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

- Nützlich, weil häufig Untermatrizen verändert werden
- Anders als bei Python-Arrays sind Unterlisten *keine* Kopien!
- Kopien gibt es nur explizit durch `copy`

Manipulation von ndarrays

```
>>> a = np.array([[1,2], [3,4]])
>>> a = np.concatenate((a, [[5,6]]))
>>> print a
[[1 2]
 [3 4]
 [5 6]]
>>> print a.transpose()
[[1 3 5]
 [2 4 6]]
>>> a = np.array([1 + 2j])
>>> print a.conjugate()
[ 1.-2.j]
```

- `np.concatenate` hängt Matrizen aneinander
- `transpose()`: Transponierte (Spalten und Zeilen vertauschen)
- `conjugate()`: Komplex Konjugierte

np.dot: Matrix-Matrix-Multiplikation

```
>>> a = np.array([[1,2],[3,4]])
>>> i = np.identity(2)           # Einheitsmatrix
>>> print a*i                   # punktweises Produkt
[[1 0]
 [0 4]]
>>> print np.dot(a,i)          # echtes Matrixprodukt
[[1 2]
 [3 4]]
>>> print np.dot(a[0], a[1])   # Skalarprodukt der Zeilen
11
```

- Arrays werden normalerweise *punktweise* multipliziert
- `np.dot` entspricht
 - bei zwei eindimensionalen Arrays dem Vektor-Skalarprodukt
 - bei zwei zweidimensionalen Arrays der Matrix-Multiplikation
 - bei ein- und zweidim. Arrays der Vektor-Matrix-Multiplikation

Lineare Algebra

```
>>> a = np.array([[1,0],[0,1]])
>>> print a.min(), print a.max()
0 1
>>> print np.linalg.det(a)
1
>>> print np.linalg.eig(a)
(array([ 1.,  1.]), array([[ 1.,  0.],
                          [ 0.,  1.]])
```

- **min, max**: Minimum und Maximum aller Elemente
- **cross**: Vektorkreuzprodukt
- **linalg.det, .trace**: Determinante und Spur
- **linalg.norm**: (2-)Norm
- **linalg.eig**: Eigenwerte und -vektoren
- **linalg.inv**: Matrixinverse
- **linalg.solve(A, b)**: Lösen von $Ax = b$

Beispiel: Rücksubstitution

Problem

Gegeben: Rechte obere Dreiecksmatrix A , Vektor b

Gesucht: Lösung des linearen Gleichungssystems $Ax = b$

$$\begin{array}{rcccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ & & & & \ddots & & \vdots & & \vdots \\ & & & & & & a_{nn}x_n & = & b_n \end{array}$$

Zum Beispiel aus dem Gaußschen Eliminationsverfahren
(dazu mehr in „Physik auf dem Computer“)

Methode: Rücksubstitution

- Letzte Gleichung: $a_{nn}x_n = b_n \implies x_n = b_n/a_{nn}$
- Vorletzte Gleichung:

$$x_{n-1} = (b_{n-1} - a_{n-1,n}x_n)/a_{n-1,n-1}$$

- Und so weiter...

Implementation

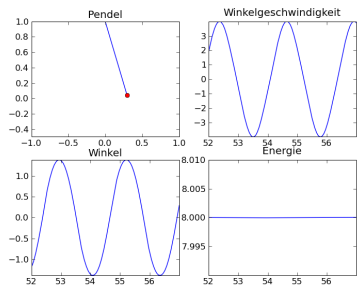
```
import numpy as np
def backsubstitute(A, b):
    rows = b.shape[0]           # length of the problem
    x = np.zeros(rows)         # solution, same size as b
    for i in range(1, rows + 1): # loop rows reversely
        row = rows - i
        x[row] = b[row] - np.dot(A[row,row+1:], x[row+1:])
        x[row] /= A[row,row]
    return x
A = np.array([[1, 2, 3], [0, 4, 5], [0, 0, 6]])
b = np.array([1, 2, 3])
print backsubstitute(A, b), np.linalg.solve(A, b)
```

Ausgabe

```
[-0.25 -0.125  0.5  ] [-0.25 -0.125  0.5  ]
```

Analyse und Visualisierung

Zeit	Winkel	Geschw.	Energie
55.0	1.1605	2.0509	8.000015
55.2	1.3839	0.1625	8.000017
55.4	1.2245	-1.7434	8.000016
55.6	0.7040	-3.3668	8.000008
55.8	-0.0556	-3.9962	8.000000
56.0	-0.7951	-3.1810	8.000009
56.2	-1.2694	-1.4849	8.000016
56.4	-1.3756	0.43024	8.000017
56.6	-1.1001	2.29749	8.000014
56.8	-0.4860	3.70518	8.000004



- Zahlen anschauen ist langweilig!
- Graphen sind besser geeignet
- Statistik hilft, Ergebnisse einzuschätzen (Fehlerbalken)
- Histogramme, Durchschnitt, Varianz

Durchschnitt und Varianz

```
>>> samples=100000
>>> z = np.random.normal(0, 2, samples)

>>> print np.mean(z)
-0.00123299611634
>>> print np.var(z)
4.03344753342
```

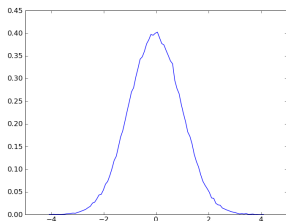
- Arithmetischer **Durchschnitt**

$$\langle z \rangle = \sum_{i=1}^{\text{len}(z)} z_i / \text{len}(z)$$

- **Varianz**

$$\sigma(z) = \langle (z - \langle z \rangle)^2 \rangle$$

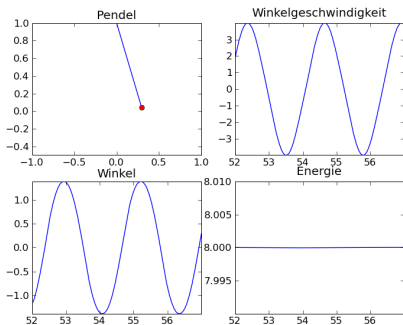
Histogramme



```
>>> zz = np.random.normal(0,1,100000)
>>> werte, raender = np.histogram(zz, bins=100, normed=True)
```

- Histogramme geben die Häufigkeit von Werten an
- In `bins` vielen gleich breiten Intervallen
- `werte` sind die Häufigkeiten, `raender` die Grenzen der Intervalle (ein Wert mehr als in `werte`)

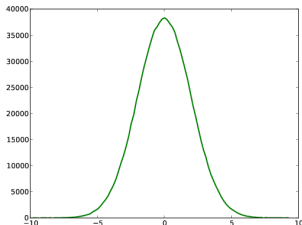
Malen nach Zahlen – matplotlib



- Ein Modul zum Erstellen von Graphen
- 2D oder 3D, mehrere Graphen in einem
- Speichern als Bitmap
- Kann auch animierte Kurven darstellen

2D-Plots

```
import matplotlib
import matplotlib.pyplot as pyplot
...
x = np.arange(0, 2*np.pi, 0.01)
y = np.sin(x)
pyplot.plot(x, y, "g", linewidth=2)
pyplot.text(1, -0.5, "sin(2*pi*x)")
pyplot.show()
```



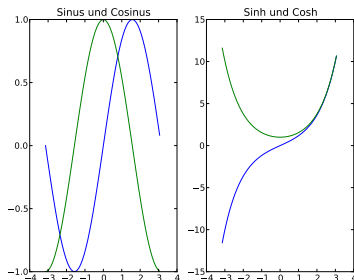
- `pyplot.plot` erzeugt einen 2D-Graphen
- `pyplot.text` schreibt beliebigen Text in den Graphen
- `pyplot.show()` zeigt den Graphen an
- Parametrische Plots mit Punkten ($x[t]$, $y[t]$)
- für Funktionen Punkte ($x[t]$, $y(x[t])$) mit x Bereich
- Farbe und Form über String und Parameter – ausprobieren

Mehrfache Graphen

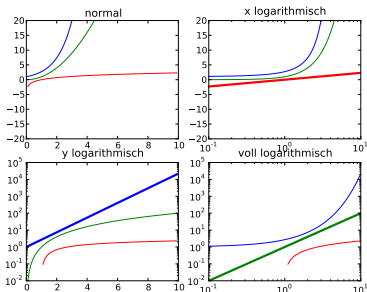
```
bild = pyplot.figure()
```

```
graph_1 = bild.add_subplot(121, title="Sinus und Cosinus")  
graph_1.plot(x, np.sin(x))  
graph_1.plot(x, np.cos(x))  
graph_2 = bild.add_subplot(122, title="Sinh und Cosh")  
graph_2.plot(x, np.sinh(x), x, np.cosh(x))
```

- Mehrere Kurven in einem Graphen:
`plot(x_1,y_1 [,"stil"], x_2,y_2 ,...)!`
- Oder mehrere `plot`-Befehle
- Mehrere Graphen in einem Bild mit Hilfe von `add_subplot`



Logarithmische Skalen



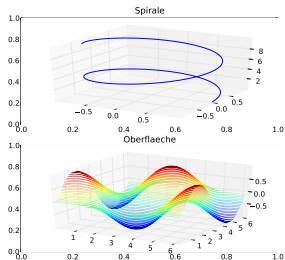
$$y = \exp(x)$$
$$y = x^2$$
$$y = \log(x)$$

- `set_xscale("log")` bzw. `set_yscale("log")`
- y logarithmisch: $y = \exp(x)$ wird zur Geraden $y' = \log(y) = x$
- x logarithmisch: $y = \log(x) = x'$ ist eine Gerade
- x + y logarithmisch: Potenzgesetze $y = x^n$ werden zu Geraden, da $y' = \log(x^n) = n \log(x) = nx'$

3D-Plots

```
import matplotlib
import matplotlib.pyplot as pyplot
import mpl_toolkits.mplot3d as p3d
...
bild = pyplot.figure()
z = np.arange(0, 10, 0.1)
x, y = np.cos(z), np.sin(z)
graph = p3d.Axes3D(bild)
graph.plot(x, y, z)
```

- plot: wie 2D, nur mit 3 Koordinaten x, y, z
- plot_wireframe: Gitteroberflächen
- contourf3D: farbige Höhenkodierung
- Achtung! 3D ist neu und das Interface ändert sich noch



Interaktive Visualisierung

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as pyplot
...
abb = pyplot.figure()
plot = abb.add_subplot(111)
kurve, = plot.plot([],[])
def weiter():
    abb.canvas.manager.window.after(1000, weiter)
    kurve.set_data(x, np.sin(x))
    abb.canvas.draw()
...
abb.canvas.manager.window.after(100, weiter)
pyplot.show()
```

- Update und Timing durch GUI (hier TkInter)
- set_data um die Daten zu verändern

Methodik des Programmierens

Schritte bei der Entwicklung eines Programms

- **Problemanalyse**

- Was soll das Programm leisten?
Z.B. eine Nullstelle finden, Molekulardynamik simulieren
- Was sind Nebenbedingungen?
Z.B. ist die Funktion reellwertig? Wieviele Atome?

- **Methodenwahl**

- Schrittweises Zerlegen in Teilprobleme (Top-Down-Analyse)
Z.B. Propagation, Kraftberechnung, Ausgabe
- Wahl von Datentypen und -strukturen
Z.B. Listen oder Tupel? Wörterbuch?
- Wahl der Rechenstrukturen (Algorithmen)
Z.B. Newton-Verfahren, Regula falsi,...

Methodik des Programmierens

Schritte bei der Entwicklung eines Programms

- **Implementierung und Dokumentation**
 - Programmieren und *gleichzeitig* dokumentieren
 - Kommentare und externe Dokumentation (z.B. Formeln)
- **Testen auf Korrektheit**
 - Funktioniert das Programm?
Z.B. findet es eine bekannte Lösung?
 - Terminiert das Programm?
D.h. hält es immer an?
- **Testen auf Effizienz**
 - Wie lange braucht das Programm bei beliebigen Eingaben?
 - Wieviel Speicher braucht es?

Methodik des Programmierens

- Teillösungen sollen wiederverwendbar sein
– möglichst allgemein formulieren
- Meistens wiederholt sich der Prozess:
 - Bei der Methodenwahl stellen sich weitere Einschränkungen als nötig heraus
Z.B. Funktion darf keine Singularitäten aufweisen
 - Bei der Implementierung zeigt sich, dass die gewählte Methode nicht umzusetzen ist
Z.B. weil implizite Gleichungen gelöst werden müssten
 - Beim Testen zeigt sich, dass die Methode ungeeignet oder nicht schnell genug ist
Z.B. zu langsam, numerisch instabil
- Mit wachsender Projektgröße wird Problemanalyse wichtiger
- *Software Engineering* bei umfangreichen Projekten und -teams

Testen auf Korrektheit und Effizienz

Korrektheit

- Extremwerte überprüfen (leere Eingabe, 0)
- *Generische* Fälle prüfen, d.h. alle Vorzeichen, bei reellen Zahlen nicht nur ganzzahlige Eingaben
- Alle Fehlermeldungen sollten getestet, also ausgelöst werden!
⇒ mehr Tests für unzulässige Eingaben als für korrekte

Effizienz

- Wie viele elementare Schritte (Schleifendurchläufe) sind nötig?
- Möglichst langsam wachsende Anzahl elementarer Schritte
- Ebenso möglichst langsam wachsender Speicherbedarf
- Sonst können nur sehr kleine Aufgaben behandelt werden
- Beispiel: bei $N = 10^6$ ist selbst $0,1 \mu s \times N^2 = 1$ Tag

Beispiel: Pythagoreische Zahlentripel

- **Problemanalyse**

Gegeben: eine ganze Zahl c

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

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

- **Methodenwahl**

- Es muss gelten: $0 < a < c$ und $0 < b < c$
- Wir können also alle Paare a, b mit $0 < a < c$ und $0 < b < c$ durchprobieren – verschachtelte Schleifen
- Unterteilung in Teilprobleme nicht sinnvoll

- **Effizienz?**

Rechenzeit wächst wie $|c|^2$ – das ist langsam!

Beispiel: Pythagoreische Zahlentripel

- Implementierung

```
def zahlentripel(c):
```

```
    """
```

Liefert ein Ganzzahlpaar (a, b), dass $a^2 + b^2 = c^2$ erfüllt, oder None, wenn keine Lösung existiert.

```
    """
```

```
    # Durchprobieren aller Paare
```

```
    for a in range(1,c):
```

```
        for b in range(1,c):
```

```
            if a**2 + b**2 == c**2: return (a, b)
```

```
    return None
```

- Test der Effizienz

Zahl (alle ohne Lösung)	1236	12343	123456
Zeit	0.5s	41s	1:20h

- Das ist tatsächlich sehr langsam! Geht es besser?

Beispiel: Pythagoreische Zahlentripel

- **Methodenwahl** zur Effizienzverbesserung
O.B.d.A. $a \leq b$
 - Sei zunächst $a = 1$ und $b = c - 1$
 - Ist $a^2 + b^2 > c^2$, so müssen wir b verringern, und wir wissen, dass es keine Lösung mit $b = c - 1$ gibt
 - Ist $a^2 + b^2 < c^2$, so müssen wir a erhöhen und wir wissen, dass es keine Lösung mit $a = 1$ gibt
 - Ist nun $a^2 + b^2 > c^2$, so müssen wir wieder b verringern, egal ob wir zuvor a erhöht oder b verringert haben
 - Wir haben alle Möglichkeiten getestet, wenn $a > b$
- **Effizienz?**
Wir verringern oder erhöhen a bzw. b in jedem Schritt. Daher sind es nur maximal $|c|/2$ viele Schritte.

Beispiel: Pythagoreische Zahlentripel

- **Implementierung** der effizienten Vorgehensweise

```
def zahlentripel(c):  
    "loest  $a^2 + b^2 = c^2$  oder liefert None zurueck"  
    # Einschachteln der moeglichen Loesung  
    a = 1  
    b = c - 1  
    while a <= b:  
        if a**2 + b**2 < c**2: a += 1  
        elif a**2 + b**2 > c**2: b -= 1  
        else: return (a, b)  
    return None
```

- Demonstration der Effizienz

Zahl	12343	123456	1234561	12345676	123456789
Zeit	0.01s	0.08s	0.63s	6.1s	62s

Bubble-Sort

- **Problemstellung**

Sortieren einer Liste, so dass $a_0 \leq a_1 \leq \dots \leq a_N$

- **Methodenwahl**

Vergleiche jedes Element mit jedem und bringe das größere durch Tauschen nach oben

- **Implementierung**

```
def bubblesort(liste):
```

```
    "sortiert liste in aufsteigender Reihenfolge"
```

```
    N = len(liste)
```

```
    for runde in range(N):
```

```
        for k in range(N - 1):
```

```
            if liste[k] > liste[k+1]:
```

```
                liste[k], liste[k + 1] = liste[k+1], liste[k]
```

- **Effizienz** – wieder nur N^2 !

Motivation

Wozu dienen *reguläre Ausdrücke*?

- Finden von *Mustern* in Text
- Parsen von Informationen (Logfiles)
- Komplexeres Suchen und Ersetzen (z.B. im Texteditor)

Beispiel

```
''This License'' refers to version 3 ...
```



```
<em>This License</em> refers to version 3 ...
```

**Reguläre Ausdrücke sind komplexere
Musterbeschreibungen**

Was sind reguläre Ausdrücke?

- Ein *regulärer Ausdruck* beschreibt ein *Muster*
- Formale Definition
 - Die leere Menge ε ist ein regulärer Ausdruck.
 - Jedes Zeichen ist ein regulärer Ausdruck.
 $\implies a$ passt nur auf „a“.
 - Wenn x und y reguläre Ausdrücke sind, dann ist auch
 - (xy) ein regulärer Ausdruck (*Verkettung*).
 $\implies (ab)$ passt nur auf „ab“.
 - $(x|y)$ ein regulärer Ausdruck (*Alternative*).
 $\implies (ab|c)$ passt auf „ab“ oder „c“.
 - x^* ein regulärer Ausdruck („*Kleenesche Hülle*“)
 $\implies (ab)^*$ passt auf „“ oder „ab“ oder „abab“ oder ...
- Klammern können weggelassen werden, wenn eindeutig.
Präzedenz: * vor Verkettung vor |.

Reguläre Ausdrücke in Python

- `re.match(pattern, string)`
Testet, ob der reguläre Ausdruck *pattern* am Anfang der Zeichenkette *string* passt
- `re.search(pattern, string)`
Sucht, ob *pattern* irgendwo in *string* passt
- Achtung: nur der *Anfang* des Strings muss passen!
- Reguläre Ausdrücke in Python sind Zeichenketten mit Präfix „r“
- Metazeichen werden durch \ zu regulären Zeichen:
\
() [] { } * + . ^ \$

```
>>> import re
>>> if re.match(r'ab|c', 'abirrelevant'): print 'Passt!'
Passt!
>>> if re.search(r'\(1*2\)', '7+(1*2)'): print 'Passt!'
Passt!
```

Erweiterte reguläre Ausdrücke

- `'.'` passt auf jedes Zeichen
`'H.se'` passt auf `'Hase'` oder `'Hose'`
- `'a+'` \equiv `'aa*'` (mindestens einmal)
`'Hallo+'` passt auf `'Hallo'` oder `'Hallooo'`, aber nicht auf `'Hall'`
- `'a?'` \equiv `'(a|)'` (ein- oder keinmal)
`'b?engel'` passt auf `'bengel'` oder `'engel'`
- `'a{2,3}':` zwei- oder dreimal
`'Hallo{2,3}':` passt auf `'Halloo'` oder `'Hallooo'`
- `'^'` / `'$'`: Anfang / Ende der Zeichenfolge

Zeichenklassen

- `'[abc]'` \equiv `'a|b|c'`
• `'[HM] aus'` passt auf `'Haus'` oder `'Maus'`
- `'[^abc]'`: alle Zeichen *außer* a, b oder c
- `'[a-z]'`: alle Zeichen zwischen a und z
- Spezielle Zeichenklassen:
 - `'\w'`: Alphanumerisches Zeichen („word-character“)
 - `'\W'`: Alle Zeichen außer alphanumerischen Zeichen
 - `'\s' / '\S'`: (alles außer) Leerzeichen („space“)

```
>>> floatpattern = r'[+-]?[0-9]*\.[0-9]*(e[+-]?[0-9]+)?'
>>> if re.match(floatpattern, '-1.3e-17'): print 'Float'
Float
>>> if re.match(floatpattern, '17'): print 'Float'
>>> if re.match(floatpattern, '.3'): print 'Float'
Float
```

Suchen und Ersetzen

- `re.sub(pattern, repl, string)`
Ersetzt alle Auftreten von *pattern* in *string* durch *repl*
- Klammern („(“ und „)“) in regulären Ausdruck erzeugen *Gruppen*
- Gruppe 0 ist der Treffer des gesamten Musters
- Andere Gruppennummern nach Reihenfolge im Ausdruck
- *repl* kann Referenzen auf Gruppen des reguläre Ausdrucks enthalten (`\1`, `\2`, ...)

```
>>> import re
>>> s="This License" refers to version 3 ...'
>>> print re.sub(r'("(*)"', r'<em>\1</em>', s)
<em>This License</em> refers to version 3 ...
```

Rückgabewerte

- `re.match` und `re.search` geben *Match-Objekte* zurück
- Diese ermöglichen, mehr über den Treffer herauszufinden
 - `groups()` ergibt die Treffer der einzelnen Gruppen
 - `group(id)` ergibt den Treffer von Gruppe *id*
 - `span(id)` ergibt Tupel mit Anfangs- und Endposition
 - `start(id)` und `end(id)`

```
>>> m = re.search(r'(ab|bc)(c|d)+', 'xxxxabcxxxx')
>>> print m.groups()
('ab', 'c')
>>> print m.group(0), m.group(1), m.group(2)
abc ab c
>>> print m.span(0), m.span(1)
(4, 7) (4, 6)
>>> print m.start(0), m.end(0)
4 7
```

„Greediness“

```
>>> import re
>>> s="Freedom" and "software" ...
>>> print re.sub(r'\"(.*)\"', r'<em>\1</em>', s)
<em>Freedom</em> and "software</em> ...
```

- Kleenesche Hüllen matchen immer so viel wie möglich („greedy“)
 - Zusätzliches ? hinter dem Muster matcht stattdessen so wenig wie möglich
-

```
>>> import re
>>> s="Freedom" and "software" ...
>>> print re.sub(r'\"(.*)\"', r'<em>\1</em>', s)
<em>Freedom</em> and <em>software</em> ...
```



Reguläre Ausdrücke in anderen Programmen

- Alle Programmiersprachen besitzen Implementierungen
- Emacs
 - Edit \rightarrow Replace \rightarrow Replace Regexp
 - Tastenkombination M-C-%
 - Eingabe von regulären Ausdrücken
 - *fast* so wie in Python
 - $(ab) \Rightarrow \backslash(ab\backslash)$
 - $\backslash d \Rightarrow [0-9]$
- egrep ist grep mit regulären Ausdrücken