Wissenschaftliches Programmieren in Python

Meik Hellmund

hellmund@math.uni-leipzig.de
www.math.uni-leipzig.de/∼hellmund/python.html

- Mi 7.30 Uhr Felix-Klein-HS
- später weitere Termine im Computerpool der Mathematik, A312

Organisatorisches

Zielgruppen

- Lehramt Sonderpäd/Mittelsch/Gym 8. Sem.
  - Kurs ist Prüfungsvorleistung (Vorauss. für Klausurteilnahme) für Numerikvorlesung Prof. Günther
- Dipl. Math/WiMa 4. Sem.
  - Kurs ist freiwillig zur Vorbereitung auf das Numerische Praktikum (Dipl. Math.) von Prof. Kunkel
  - nicht als Thema in mündl. Prüfungen

Organisatorisches

Erfolgreiche Teilnahme

ist definiert als:

- erfolgreiche Lösung von ≥ 5 Programmieraufgaben
  (s. Webseite zum Kurs)
  davon 2 aus der Gruppe Aufgaben 1–10
  und 3 aus der Gruppe Aufgaben 11–23
- Zweier- und Dreiergruppen zulässig
- nicht im 14-Tage-Rhythmus, bis zur letzten VL-Woche
- Vorstellung im Computerpool bei mir

Organisatorisches

Anmeldungen

- Online-Anmeldung unter
  `http://milab6.mathematik.uni-leipzig.de:8000/a/`
  - Für Computerpool der Mathematik (bitte auch anmelden, wenn Sie schon einen Pool-Account haben) und den Python-Notebook-Webserver
  - Anmeldepasswort: steht an der Tafel
  - verwendet studserv-Account als Namen, hat aber sonst nichts damit zu tun. Mail mit Passwort geht an diese Adresse.

  Bitte bald anmelden!

Computerpool

- im Augusteum 3. Etage, Raum A312

Python-Notebook-Webserver

- `https://misun102.mathematik.uni-leipzig.de:8000/`

Ziel und Inhalt

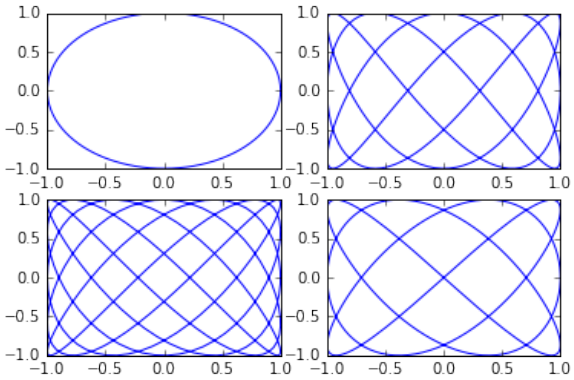
- **Einführung** in Python
  (keine Objektorientierung, Metaklassen, etc...)
- Schwerpunkt auf numerische Anwendungen
- Die Bibliothek **NumPy** für numerisches Python
- Die Bibliothek **MatPlotLib**

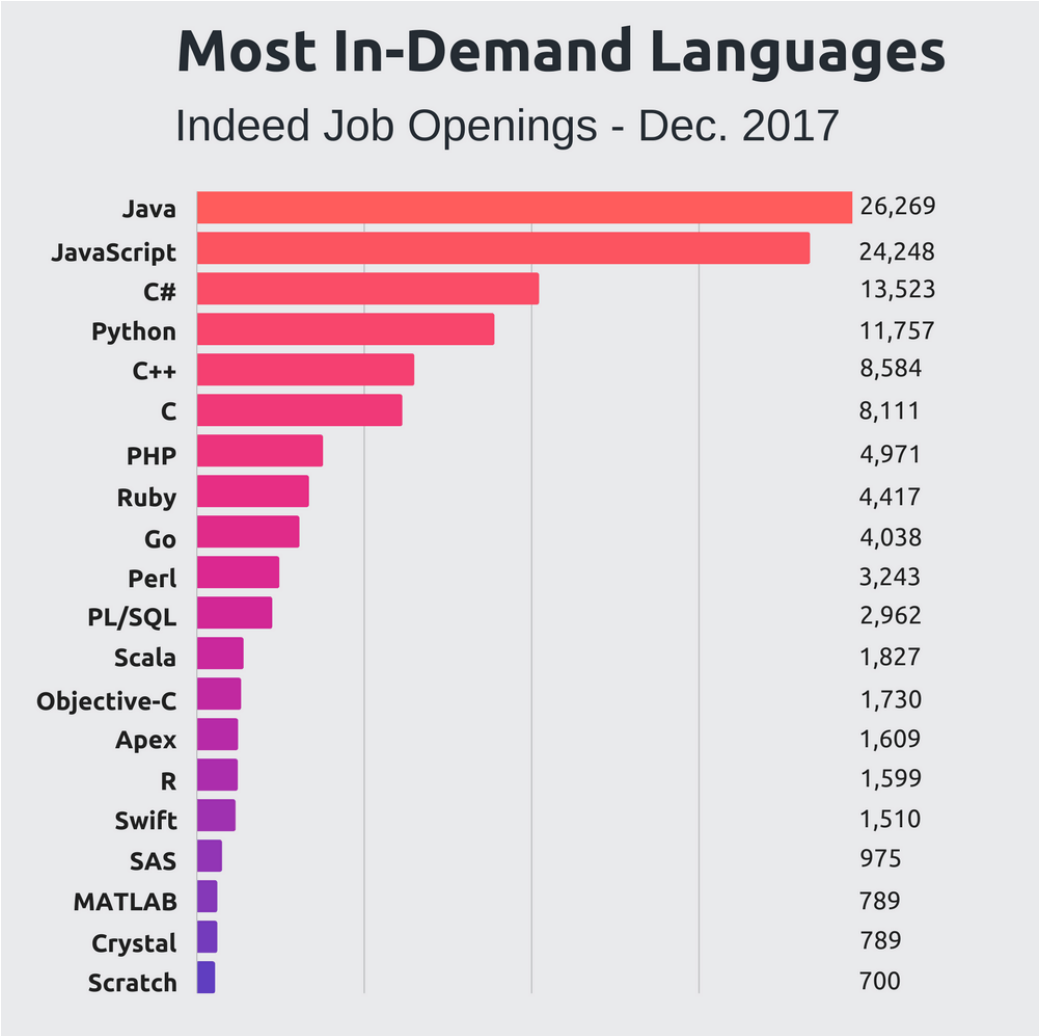
```
import numpy as np
import matplotlib.pyplot as plt

a = [1,3,5,3]
b = [1,5,7,4]
delta = np.pi/2
t = np.linspace(-np.pi, np.pi, 300)

for i in range(0,4):
   x = np.sin(a[i] * t + delta)
   y = np.sin(b[i] * t)
   plt.subplot(2,2,i+1)
   plt.plot(x,y)

plt.show()
```





Warum Python?

- Vielseitige Sprache: prozedural | objektorientiert | funktional
- Free & Open Source
- Implementierungen für alle gängigen Betriebssysteme
- Interpreter, Just-in-time Compiler, Cython, Jython
- Umfangreiche Bibliotheken und Module
- Schlanke Syntax

Schlanke Syntax

Java

```
public class HelloWorldApp {
    public static void main(String[] args) {
         System.out.println("Hello World!");
    }
}
```

Python

```
print("Hello World!")
```

Geschichte

1991 first release, version 0.9
1994 Python 1.0
2000 Python 2.0
2008 Python 3.0

- z. Zt. 2 Dialekte, Python 2 und Python 3
- Aktuelle Arbeit an Python2: *bug fixing* und *backporting* von Python3-Features
- Aktuell: 19. Dez 2017 Python 3.6.4
  16. Sep 2017 Python 2.7.14

Guido van Rossum (2006)

Werbepause

```
solve([x^3 + b*x + c == 0],x)
```

$$\left[x = \frac{b\left(-i\sqrt{3}+1\right)}{6\left(-\frac{1}{2}c+\frac{1}{6}\sqrt{\frac{4}{3}b^3+9c^2}\right)^{\frac{1}{3}}} - \frac{1}{2}\left(-\frac{1}{2}c+\frac{1}{6}\sqrt{\frac{4}{3}b^3+9c^2}\right)^{\frac{1}{3}}\left(i\sqrt{3}+1\right), x = \frac{b\left(i\sqrt{3}+1\right)}{6\left(-\frac{1}{2}c+\frac{1}{6}\sqrt{\frac{4}{3}b^3+9c^2}\right)^{\frac{1}{3}}} - \frac{1}{2}\left(-\frac{1}{2}c+\frac{1}{6}\sqrt{\frac{4}{3}b^3+9c^2}\right)^{\frac{1}{3}}(-i\sqrt{3}+\right.$$

```
f = 1/((1+x)*(x-1)^2)
```

```
f.partial_fraction(x)
```

$$\frac{1}{4(x+1)} - \frac{1}{4(x-1)} + \frac{1}{2(x-1)^2}$$

```
diff(sin(x^2), x, 4)
```

$$16x^4\sin\left(x^2\right) - 48x^2\cos\left(x^2\right) - 12\sin\left(x^2\right)$$

```
integral(x^3/sqrt(x^2+1), x)
```

$$\frac{1}{3}\sqrt{x^2+1}x^2 - \frac{2}{3}\sqrt{x^2+1}$$

SageMath auf Python basierendes Computeralgebrasystem
free&open source www.sagemath.org

```
solve([x^3 + b*x + c == 0],x)
```

$$\left[x = \frac{b\left(-i\sqrt{3}+1\right)}{6\left(-\frac{1}{2}c+\frac{1}{6}\sqrt{\frac{4}{3}b^3+9c^2}\right)^{\frac{1}{3}}} - \frac{1}{2}\left(-\frac{1}{2}c+\frac{1}{6}\sqrt{\frac{4}{3}b^3+9c^2}\right)^{\frac{1}{3}}(i\sqrt{3}+1), x = \frac{b\left(i\sqrt{3}+1\right)}{6\left(-\frac{1}{2}c+\frac{1}{6}\sqrt{\frac{4}{3}b^3+9c^2}\right)^{\frac{1}{3}}} - \frac{1}{2}\left(-\frac{1}{2}c+\right.\right.$$

```
f = 1/((1+x)*(x-1)^2)
```

```
f.partial_fraction(x)
```

$$\frac{1}{4(x+1)} - \frac{1}{4(x-1)} + \frac{1}{2(x-1)^2}$$

```
diff(sin(x^2), x, 4)
```

$$16x^4\sin\left(x^2\right) - 48x^2\cos\left(x^2\right) - 12\sin\left(x^2\right)$$

```
integral(x^3/sqrt(x^2+1), x)
```

$$\frac{1}{3}\sqrt{x^2+1}x^2 - \frac{2}{3}\sqrt{x^2+1}$$

Nutzungsmöglichkeiten

1. Computerpool Mathematik (unter Linux)
2. Python-Notebook-Webserver
   `https://misun102.mathematik.uni-leipzig.de:8000/`
3. Wenn Sie möchten:
   - Linux: Installieren Sie mit dem Software-Manager Ihrer Linux-Distribution Python3 sowie die Python3-Pakete numpy, matplotlib, spyder und bei Bedarf scipy, ipython/jupyter
   - Als Alternative können Sie auch Anaconda für Linux installieren.
   - Windows/MacOS: z.B. Python-Distribution „Anaconda“ (enthält alle nötigen Pakete)

Python ausführen

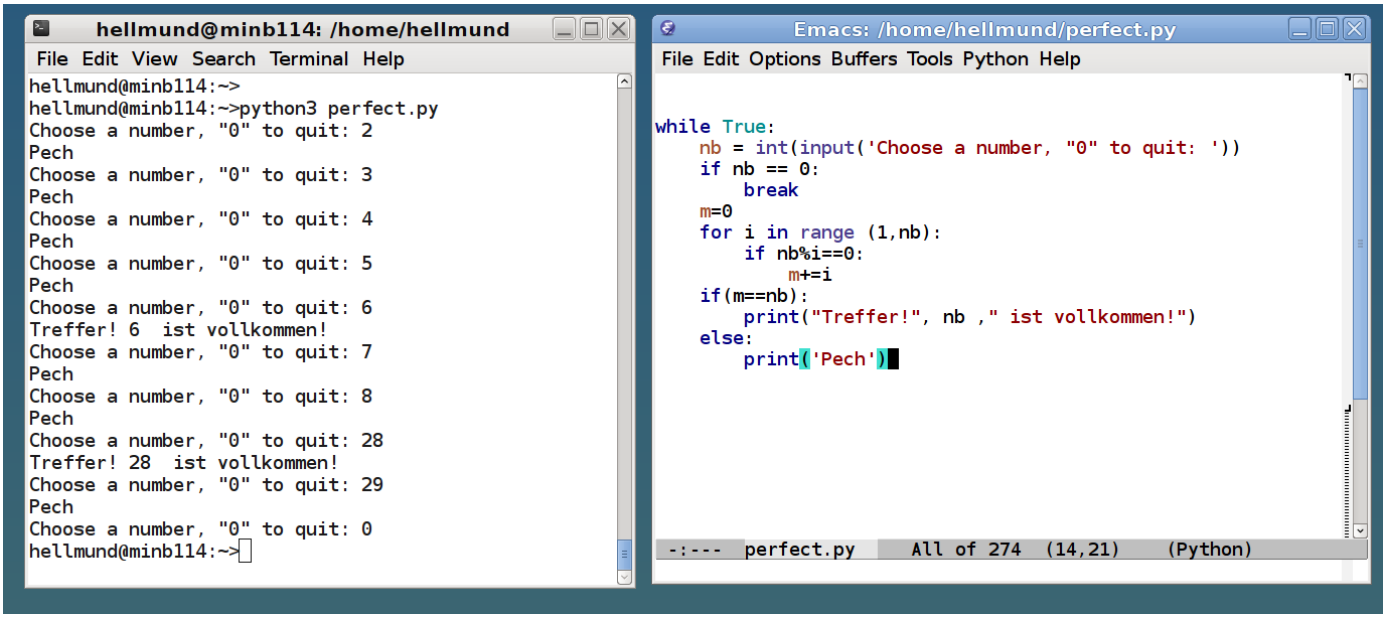
1. Interaktiv: Konsole/Terminal/Eingabeaufforderung/cmd.exe

```
#> python3
Python 3.5.1+ (default, Feb 21 2016, 23:11:32)
[GCC 5.3.1 20160220] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
>>> 2**64-1
18446744073709551615
>>> def sum_powers(n):
...     s=0
...     for i in range(n):
...         s += 2**i
...     return s
...
>>> sum_powers(5)
31
>>> sum_powers(64)
18446744073709551615
>>> sum_powers(64) == 2**64 - 1
True
>>> quit()
```

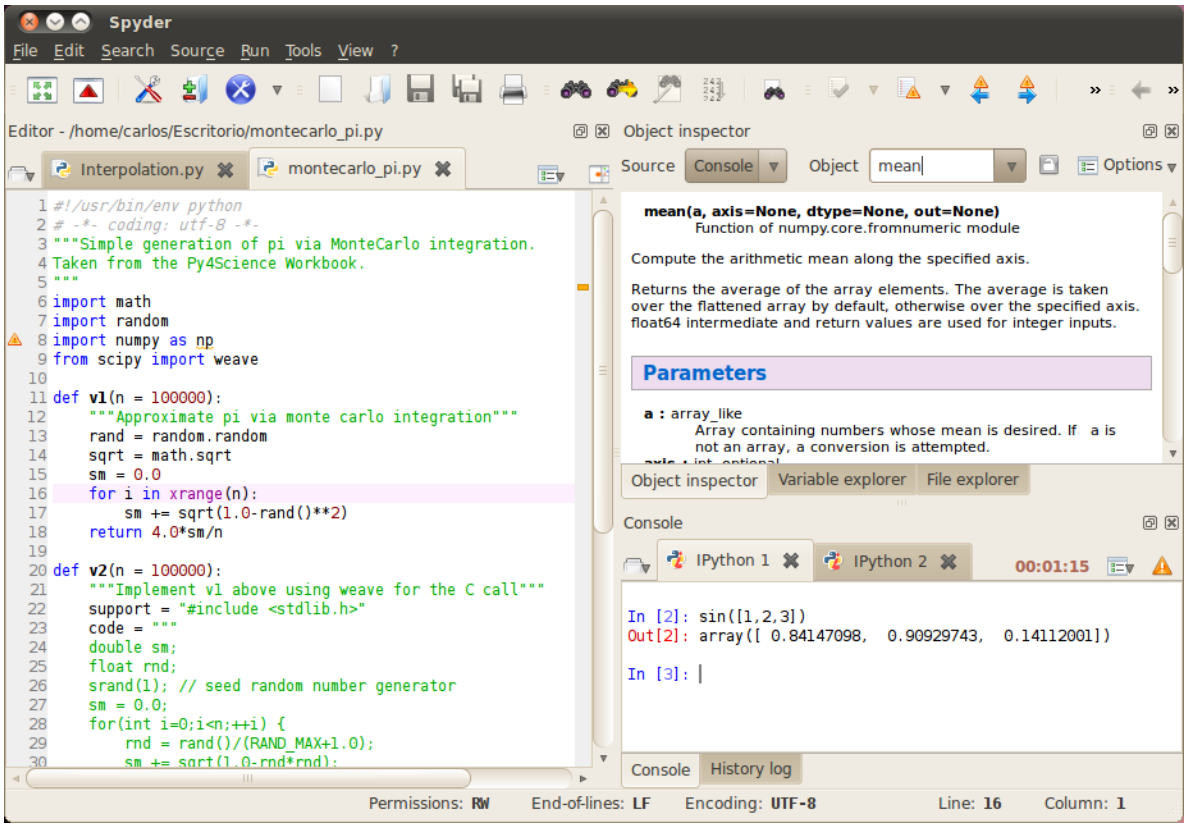

Python ausführen

2. Konsole + Texteditor



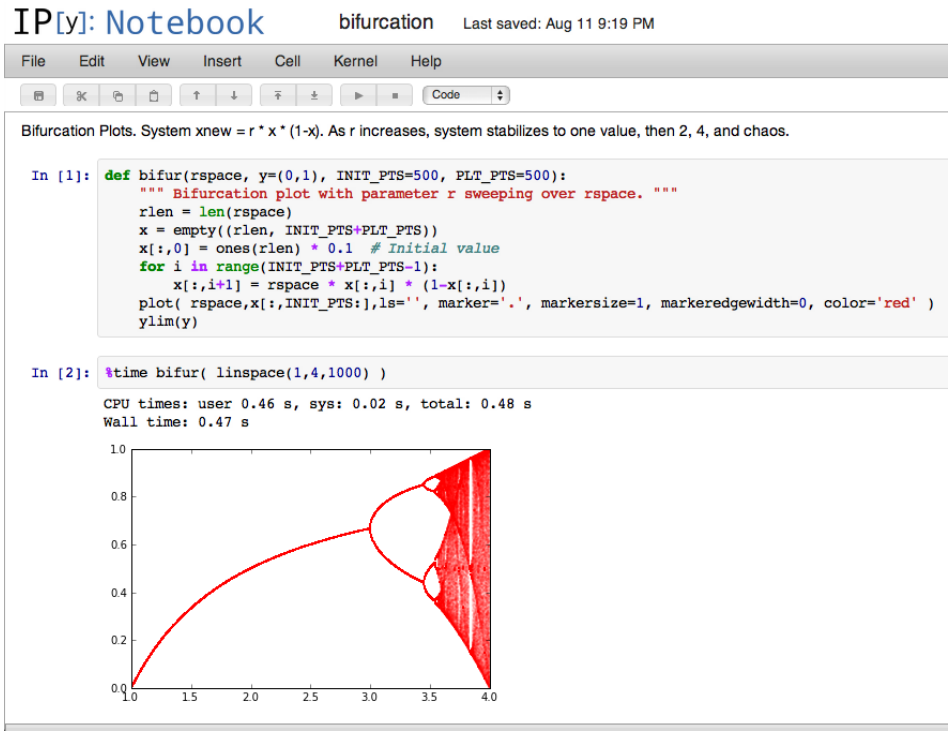
Python ausführen

3. IDE (Integrated development environment), z.B. Spyder:



Python ausführen

4. Jupyter/IPython Webinterface – Python Notebooks



[Quelle: http://www.collindelker.com/wp/2012/08/fractal-art/]

Python Notebook Webinterface “Jupyter”

- Einmalig auf `http://milab6.mathematik.uni-leipzig.de:8000/a/` anmelden, Passwort wird an `studserv`-Adresse zugeschickt
- Auf `https://misun102.mathematik.uni-leipzig.de:8000/` einloggen
- Verzeichnis Ihrer Dateien erscheint:
  Python Notebooks haben die Endung `.ipynb`
- Notebook laden oder mit `New -> Python3` neues Notebook aufmachen oder mit `Upload` hochladen
- Am Ende bitte immer rechts oben `Logout`!

Python Notebook Webinterface “Jupyter”

- Notebooks bestehen aus `cells`, diese können Python-Anweisungen oder anderen Text enthalten.
- Nützlich: `File-> Save&Checkpoint`, `File-> Rename`, `Help-> Keyboard Shortcuts`
- Ausfühen von Python-Cells:
  - `Strg-Enter`: run cell
  - `Shift-Enter`: run cell, move cursor to next cell
  - `Alt-Enter`: run cell, insert new cell below

Jupyter magic commands

Wir brauchen nur:

```
%matplotlib notebook
```

damit Matplotlib-Plots im Notebook als Grafiken erscheinen.

```python
def teiler_test(i):
    """ berechnet Anzahl der Teiler und Summe der Teiler von i """
    n = 0;  s = 0
    for k in range(i):
        if i%(k+1) == 0:
            n += 1
            s += k+1
    return (n, s)

# main program starts here
while True:
    str = input("Bitte natürliche Zahl eingeben, '0' für Abbruch:")
    i = int(str)
    if i==0:
        break
    numTeiler, sumTeiler = teiler_test(i)
    if numTeiler == 2:
        print("Oha! Eine Primzahl!")
    elif sumTeiler == 2*i:
        print("Glückwunsch! Eine vollkommene Zahl!")
    else:
        print("Leider keine vollkommene Zahl. Sie hat übrigens ",
              numTeiler, " verschiedene Teiler.")
```


Grundlagen der Syntax

Namen von Variablen, Funktionen, Klassen usw.

- beliebig lang, aus Buchstaben, Ziffern und Unterstrich _
- erstes Zeichen muß Buchstabe sein
- Groß- und Kleinbuchstaben werden unterschieden
  Nmax und NMAX sind verschiedene Variablen
- Zeichensatz: Unicode in UTF-8 Kodierung
- 33 reservierte *keywords*:

```
False    None     True     and      as       assert   break
class    continue def      del      elif     else     except
finally  for      from     global   if       import   in
is       lambda   nonlocal not      or       pass     raise
return   try      while    with     yield
```

- zulässig: `i, x, π, x23, DieUnbekannteZahl, neuer_Wert, Zähler, אבג, delta, δ`
  unzulässig: `3achsen, A#b, $this_is_not_Perl, del`

Python ist zeilenorientiert

- Im Normalfall enthält jede Zeile genau eine Anweisung.
- Wenn eine Zeile mit einem *backslash* endet, wird die Anweisung fortgesetzt:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:
        date_valid = True
```

- Bei einer offenen Klammer `(), [], {}` kann der *backslash* entfallen:

```
Monatsnamen = ['Januar', 'Februar', 'März', 'April', 'Mai',
               'Juni', 'Juli', 'August', 'September',
               'November', 'Dezember']
```

- Mehrere Anweisungen pro Zeile können durch ein Semikolon getrennt werden:

```
x=33; x += 22; print(x)
```


Blöcke werden durch Einrückung beschrieben

- Rücke ein, um einen Block zu beginnen.
- Rücke die nächste Zeile wieder raus
  bis zum vorherigen *indentation level*, um den Block zu beenden.
- Größe der Einrückung egal. Empfohlen: 4 Leerzeichen.
- Don't mix tabs and spaces.
- Anweisungen, die einen Block einleiten, enden in der Regel mit einem Doppelpunkt.

```
def f(x, z):
    n=17
    while z>0:
        if x==0:
            return n
        else:
            z-=1
            n=n+z
        z=z-x
    return 2*n

m=f(3,4)
```

Blockstruktur

```
def f(x,z):
    n=17
    while z>0:
        if x==0:
            return n
        else:
            z-=1
            n=n+z
        z=z-x
    return 2*n
```

```
m=f(3,4)
```


Grundlagen der Syntax

Kommentare

- Kommentare erstrecken sich von einem # bis zum Zeilenende
- Wie man es nicht unbedingt machen sollte:

```
#*****************************************************
# Die folgende Funktion berechnet das Quadrat einer Zahl
#*****************************************************
def square(x):
    x2 = x * x      # hier wird das Quadrat berechnet!!!!
    return x2
```

Nutzung von Modulen

Module stellen Funktionen, Klassen, Objekte, Konstanten,... zur Verfügung.
Sie müssen vor ihrer Verwendung `import`iert werden.
Verschiedene Formen:

```
import math
x = math.sin(3/20 * math.pi)
y = math.sqrt(7)
```

```
import math as mt
x = mt.sin(3/20 * mt.pi)
y = mt.sqrt(7)
```

```
from math import sin, sqrt as qwurz, pi
x = sin(3/20 * pi)
y = qwurz(7)
```


Python-Module

Python standard library

- über 100 Module, die bei jeder Python-Installation dabei sind
- `math, cmath, decimal, fractions, re, textwrap, datetime, itertools, os, threading, email, html, xml, urllib, ...`

PyPI - The Python Package Index

- `pypi.python.org`
- über 50 000 Pakete, leicht nachinstallierbar
- z.B. `matplotlib, numpy, scipy`

- Andere und selbstgeschriebene Module

Datentypen

- Python ist **stark typisiert:** Alle Objekte haben einen Typ.
  Ein String (Zeichenkette) `str` ist etwas anderes als eine ganze Zahl `int` ist etwas anderes als eine Fließkommazahl `float` etc...
- **Aber:** Variablen sind nur typlose Referenzen auf Objekte.
  Sie werden nicht deklariert und können verschiedene Typen referenzieren.
- Python hat ein **dynamisches Typsystem**. Der Typ eines Objekts wird erst geprüft, wenn das Objekt verwendet wird.

```
x = 3           #  x existiert jetzt und ist ein Integer
x = "bla"       #  jetzt ist x ein String
x = x + "blub" #  ist für den Typ von x, also  "str",
                #  eine Addition definiert? Ja!
print(x)
'blablub'
```


Datentypen

Numerische Typen: `int, float, complex`

Logische Variablen: `bool` mit den Werten `True` und `False`

Zeichenketten: `str, bytes, bytearray`

Container: `list, tuple, dictionary, set, ...`

Zusätzliche Bibliotheken/Module definieren zahlreiche zusätzliche Klassen/Typen, z.B.:

`numpy`: `int32, int64, float32, float64,...`

`matplotlib`: `plot, axes,...`

`datetime`: `date, time, tzinfo,...`

Strings (Zeichenketten)

```
s  = 'Ein String'
s1 = "Auch ein String"
s2 = "Bei uns tag's und nacht's geöffnet"
s3 = """Ein Text über
mehrere Zeilen mit ' und " im
Text"""
#
# backslashes \ haben eine Spezialbedeutung
s4 = "Auch ein\nText über 2 Zeilen"
#
# in 'raw strings' entfällt die Spezialbedeutung:
s5 = r"Das ist nur\n eine Zeile mit einem backslash"
# strings sind Sequenzen von Unicode-Zeichen
s6="ßøéäcë"
```

Das ’ ist kein Akzent sondern das Zeichen, welches auf deutschen Tastaturen rechts neben dem ä und über dem # liegt.
Das " ist die Doppelquote über der Ziffer 2.

Einige Stringfunktionen

```
>>> s = "Automobil"
>>> s.startswith("Au")
True
>>> s + " Fahrrad"
'Automobil Fahrrad'
>>> 2 * s
'AutomobilAutomobil'
>>> s[1:5]
'utom'
>>> s[-1]
'l'
>>> len(s)
9
```

```
>>> s.replace("Auto","Perpetuum") + "e"
'Perpetuummobile'
>>> s.find("om")
3
>>> s[3]
'o'
>>> "om" in s
True
>>> s
Automobil
```

Strings sind *immutable*. Funktionen wie `str.replace()` ändern das Stringobjekt nicht sondern produzieren einen neuen String.

Listen

```
l = []                      # leere Liste
l = [42]                    # Liste mit einem Element
l = [1, 2, 33 ]             # Liste mit 3 Elementen
l = [1, 2, 33, ]            # dieselbe Liste
l = [2, "bla", 2.4 ]        # Elemente können verschiedenen Typ haben
l = [[1, 2], [3, 4], [5, 6]]   # list of lists
```


Einige Listenfunktionen

```
>>> l = [3, 2]
>>> l.append(7)
>>> l
[3, 2, 7]
>>> l = l + [1,9]
>>> l.reverse()
>>> l
[9, 1, 7, 2, 3]
>>> l.sort()
>>> l
[1, 2, 3, 7, 9]
>>> len(l)
5
>>> l[3:5]
[7, 9]
>>> 7 in l
True
```

```
>>> l
[1, 2, 3, 7, 9]
>>> l.insert(3, "a")
>>> l
[1, 2, 3, 'a', 7, 9]
>>> del l[2:4]
>>> l
[1, 2, 7, 9]
>>> l += [2, 3, 4, 3, 4]
>>> l
[1, 2, 7, 9, 2, 3, 4, 3, 4]
>>> l.count(4)
2
>>> l.remove(4)
>>> l
[1, 2, 7, 9, 2, 3, 3, 4]
>>>
```

Listen sind *mutable*. `list.append(), .sort()` etc. ändern das Objekt, auf das sie wirken.

Tupeln

- **Tupeln** ähneln Listen, sind aber *immutable*/unveränderlich.

```
t = (13, 4, "ABC")
```

- Tupeln werden üblicherweise mit runden Klammern geschrieben, die Klammern sind aber optional:

```
t = 13, 4, "ABC"
```

- Tupeln erlauben Mehrfachzuweisungen *(implicit tuple packing & unpacking)*:

```
x, y, z = 0, 0, 1
```

- Funktionen, die mehrere Werte zurückgeben, geben Tupel zurück:

```
def f(x):
    return (sin(x), cos(x))

s, c = f(z)
```

(**nicht** für große Vektoren, Felder, Matrizen,... geeignet)

Strings, Listen, Tupeln sind Sequenzen

Alle Sequenz-Typen erlauben die folgenden Operationen:

`x in S, x not in S`	Element in Sequenz enthalten?
`S[i]`	gibt i-tes Element von S zurück (Zählung ab 0)
`S[i:j]`	gibt Subsequenz mit Elementen $i, \ldots, j-1$ zurück
`S[i:j:k]`	Subsequenz mit El. $i, i+k, i+2k, \ldots, i+n*k < j$
`S[-1]`	letztes Element, negative Indizes zählen von hinten
`len(S)`	Länge der Sequenz
`min(S), max(S)`	kleinstes/größtes Element der Sequenz
`S + T`	Zusammenfügen von Sequenzen gleichen Typs
`n * S`	Zusammenfügen von n Kopien

Logische Variablen, Vergleiche, Tests

- Vergleichsoperatoren: `==, !=, <, >, <=, >=`
- Tests: `a in L, a not in L`

liefern einen Wahrheitswert vom Typ `bool`, entweder `True` oder `False`

```
>>> x = 4 > 5
>>> type(x)
<class 'bool'>
>>> x
False
```


- Boolsche Variablen und Ausdrücke können durch die logische Operatoren `and, or, not` verknüpft werden.

```
>>> x = 4 <= 6 and 'a' in 'kaa'
>>> x
True
>>> if x:
...     print("hurra!")
...
hurra!
```

- Vergleiche können verkettet werden:
  `a < b <= c` ist eine Kurzform für `a < b and b <= c`

Ablaufsteuerung - If

```
if x > 3 and 2 < z < 5:
    y = - 1
elif x < 10:
    y = x + 2
    z = 5
else:
    y = 25
    z = 3
print(x, y, z)
```

- `elif` steht für *else if*
- Die `elif` und `else`-Blöcke sind optional.
- Es kann beliebig viele `elif`-Blöcke geben.

Ablaufsteuerung - Schleifen

Python kennt zwei Schleifenkonstrukte: `while` und `for`.

Die `while test`-Schleife wird abgearbeitet, wenn und solange `test` wahr ist:

```
while x > 0:
   y = x/3
   x = x/2 - y
   ...
print(x, y)
```

Ablaufsteuerung - Schleifen

Der Schleifenblock von `while`- und `for`-Schleifen kann die Anweisungen `break` und `continue` enthalten:

```
while x > 0:
   y = x/3
   z = func( x, 2*y)
   if(z == 0):
      continue   # beginne sofort den nächsten Durchlauf
   if y < 1:
      break      # breche Schleife  ab
   x = x/2 - y
print(x, y)
```


while-Schleife

```
eps = 1
while 1 + eps != 1:
   eps = eps/2
print(2 * eps)
```

ist identisch zu:

```
eps = 1
while True:
   if eps + 1 == 1:
      break
   eps = eps/2
print(2 * eps)
```

for-Schleifen

Allgemeine Form

```
for x in <iterierbares Objekt>:
```

Beispiel: Iterieren über eine Liste:

```
>>> for i in ["Vater", "Mutter", "Tochter"]:
...     print (i)
...
Vater
Mutter
Tochter
>>>
```


for-Schleifen

Das Wichtigste (für Numerik) zum Schluss:

range

- `range(n)` erzeugt ein iterierbares Objekt (einen `iterator`) mit den n Elementen $0, .., n-1$

```
for i in range(4):
    print(i**2)
```

druckt die 4 Zahlen: `0, 1, 4` und `9`.

- Um den Inhalt eines `range`-Objekts zu sehen, kann man es in eine Liste umwandeln:

```
>>>list(range(4))
[0, 1, 2, 3]
```

range

- range hat die 3 Formen
  - range(Stopwert)
  - range(Startwert, Stopwert)
  - range(Startwert, Stopwert, Zuwachs)

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


Mit Hilfe eines Index über eine Liste L iterieren:

```
def sum(L):
    s = 0
    for i in range(len(L)):
        s += L[i]
    return s
```

oder besser: direkt über die Elemente der Liste iterieren:

```
def sum(L):
    s = 0
    for x in L:
        s += x
    return s
```

Interaktive Hilfe

`print(x)`	Zeige Objekt x
`type(x)`	Zeige Typ (Klasse) von x
`help(x)`	Zeige Docstrings/Hilfe zu x
`dir(x)`	Zeige Methoden, *members, member functions*, etc von x

Beispiele (bitte mal ausprobieren!):

```
dir(list)
help(list)
dir(str)
help(str)
help()              #  startet interaktive Hilfe
import math; dir(math)
import math; help(math)
import datetime; help(datetime.time)
import fractions; help(fractions)
```


Ganze Zahlen

- übliche Arithmetik `+,-,*,/` Potenz `**`, `//, %`
- Zahlen vom Typ `int` können beliebig groß werden:

  `3**55 => 17444921100912017907117050 7`
- Division erzeugt immer Fließkommazahlen `3/3 => 1.0`
  Integer-Division `//` rundet nach $-\infty$

  `25//7 => 3,  -25//7 => -4`
- Der Operator `%` liefert Rest der Ganzzahl-Division `25%7 => 4`
  so dass immer

  `(a//b)*b + a%b = a`

  gilt

Ganze Zahlen in anderen Sprachen

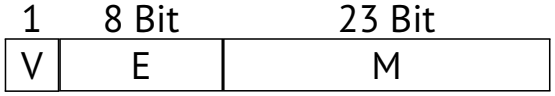
- Die meisten Programmiersprachen haben Integers mit fester Länge von n Bits ($n = 32, 64$) $\Longrightarrow$ schneller
- Wertebereich: $-2^{n-1} \leq n \leq 2^{n-1} - 1$
- Arithmetik ist modulo 2^n, keine Fehlermeldung beim Überlauf

```
>>> import numpy as np
>>> np.iinfo(np.int32)
iinfo(min=-2147483648, max=2147483647, dtype=int32)
>>> 2**31
2147483648
>>> np.iinfo(np.int64)
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
>>> 2**63
9223372036854775808
>>> a=np.int32(2147483600)
>>> b=np.int32(200)
>>> a+b
-2147483496
```

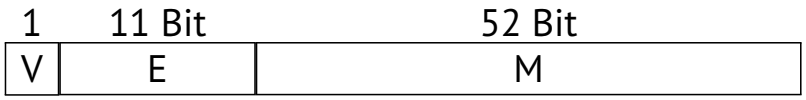

Fließkommazahlen

- viel größerer Zahlenbereich als ganze Zahlen begrenzter Länge
- mit begrenzter Genauigkeit (“Anzahl der gültigen Ziffern”)
- $x = (-1)^V \times 2^E \times M$
  E Exponent, M Mantisse ($0 \leq M < 1$), V Vorzeichenbit
- Standard IEEE 754

Einfache Genauigkeit: 32 Bit (`'float'` in C)



Doppelte Genauigkeit: 64 Bit (`'double'` in C)



Fließkommazahlen

	einfach	doppelt
Exponentenlänge E (Bits)	8	11
kleinster Exponent e_{min}	-125	-1021
größter Exponent e_{max}	128	1024
Mantissenlänge	24	53
betragsmäßig kleinste normalisierte Zahl größer Null	1.18×10^{-38}	2.23×10^{-308}
größte darstellbare Zahl	3.40×10^{38}	1.79×10^{308}
kleinste Zahl ϵ, für die $1 + \epsilon \neq 1$	5.96×10^{-8}	1.11×10^{-16}
Abstand $\epsilon' = 2\epsilon$ zwischen 1.0 und der nächstgrößeren Maschinenzahl	1.19×10^{-7}	2.22×10^{-16}

Fließkommazahlen in Python

- Pythons `float` sind auf 64Bit-Betriebssytemen auch 64Bit (doppelt genaue) Fließkommazahlen.
- Das Modul `numpy` stellt auch `float32, float64` zur Verfügung.

```
>>> 0.1 + 0.1 == 0.2
True
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> format(0.1, '.25g')
'0.1000000000000000055511151'
>>>
```

Test auf näherungsweise Gleichheit bei Fließkommazahlen:

```
eps=1.e-8
if abs(x-y) < eps:
    ...
```

Das `math` Modul

- Winkel im Bogenmaß
- `sin, cos, tan, asin, acos, atan, atan2(y,x), sinh, cosh, tanh, asinh, acosh, atanh, sqrt, exp, log, log2, log10`
- `ceil(a)` kleinste ganze Zahl $\geq a$.
- `floor(a)` größte ganze Zahl $\leq a$
- `gcd(m, n)`
- `erf(x), factorial(n), gamma(x), ...`
- `e, pi`

Bereits in Python eingebaut:

- `abs`
- `max(a1, a2 [,a3,...]),  min(a1,a2 [,a3,...])`
- Typumwandlungen: `int, float, complex, str`
- Integers in anderen Basen als Strings: `bin, hex, oct, int`
  `bin(23) -> '0b10111', int('10111', 2) -> 23`

Komplexe Zahlen und das Modul cmath

```
>>> x=2+3j
>>> x.real
2.0
>>> x.imag
3.0
>>> x.conjugate()
(2-3j)
>>> abs(x)
3.605551275463989
>>> import cmath
>>> cmath.sin(x)
(9.15449914691143-4.168906959966565j)
>>> cmath.exp(-cmath.pi * 1j)
(-1-1.2246467991473532e-16j)
>>> cmath.log(-2).imag
3.141592653589793
>>> cmath.polar(x)
(3.605551275463989, 0.982793723247329)
>>> (r,phi)=cmath.polar(x)
>>> cmath.rect(r,phi)
(2+3j)
>>>
```

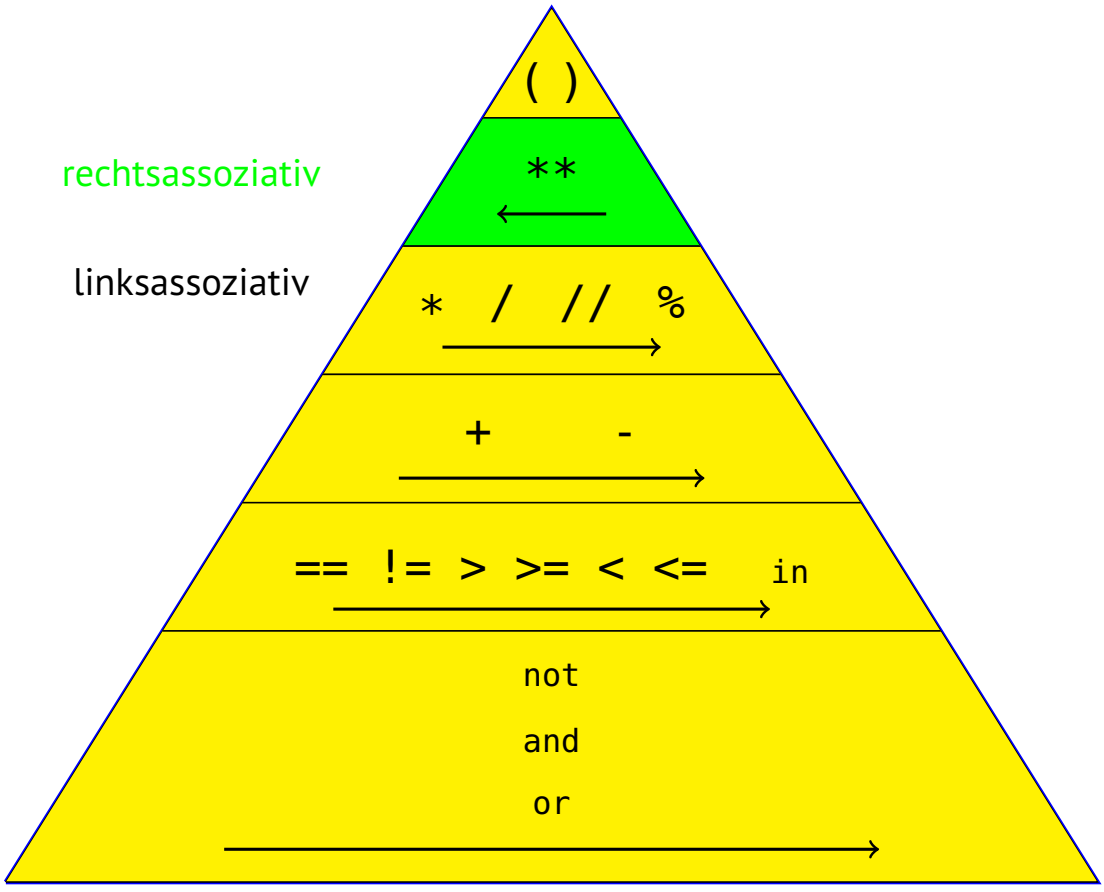
Zuweisungen

- Auf der linken Seite muß ein Objekt stehen: ~~`x + y = sin(z)`~~
- Zuweisungen sind keine Werte: ~~`if (x=sin(y)) < 0:`~~
  (aber `x = y = z = 3` ist erlaubt)

Arithmetische Operationen

- Division ist `a/b`, **nicht** `a:b`. Multiplikationsoperator ist nötig:
  ~~`2(x + 3sqrt(y))`~~ `2*(x + 3*sqrt(y))`
- `*,/` bindet stärker als +,− `a + b * c = a + (b * c)`
- `*,/,+,-` sind linksassoziativ `a / b / c = (a / b) / c`
  `a/b*c = (a/b) * c`
- `a**b` ist der Potenzoperator a^b.

Rangfolge und Assoziativität



Beispiel:

```
2*a**3 + 5 != 0 and not x == 0 or y > 1
```

wird interpretiert als:

```
((((2*(a**3)) + 5) != 0) and (not (x == 0))) or (y > 1)
```

Natürlich ist auch niemand böse, wenn man einige der Klammern hinschreibt:

```
((2*a**3 + 5 != 0) and (not x == 0)) or y > 1
```


Noch etwas Syntax

- Operationen der Form `a = a ⊙ b` können als `a ⊙= b` geschrieben werden:
  `x /= 2; L += [x]; i += 1; str += '\n'`
- Die Tue-nix-Anweisung: `pass`

```
if x == 0:
   pass
else:
   ...
```

- List comprehension
  - Erzeugt aus einer Liste/Generator eine neue Liste ohne explizite Iteration
  - ähnelt der mengentheoretischen Notation $M = \{2x \mid x \in L, x^2 > 55\}$

List comprehension

```
>>> [ 2**i for i in range(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> L = [ 2**i for i in range(10)]
>>> L
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> [ 3*x for x in L ]
[3, 6, 12, 24, 48, 96, 192, 384, 768, 1536]
>>> [ 3*x for x in L if x > 30 ]
[96, 192, 384, 768, 1536]
>>> from math import sqrt
>>> [sqrt(x) for x in range(8) if x%2 == 0]
[0.0, 1.4142135623730951, 2.0, 2.449489742783178]
>>>
```


Funktionen

- startet mit `def`
- im Allgemeinen mindestens eine `return`-Anweisung
- eigener *namespace*: Variablen sind lokal

```
def posdiff(a, b):
    z = a - b
    if z > 0:
       return z
    else:
       return -z

x = posdiff(44, 55)     # hier ist z unbekannt
```

Funktionen

- Eine Funktion, die ohne `return` endet, gibt das spezielle Objekt `None` zurück:

```
>>> def g(z):
...     print("hihi")
...
>>> x = g(33)
hihi
>>> x
>>> type(x)
<class NoneType>
>>> x==None
True
>>> print(x)
None
>>>
```


Funktionen

Bei der Definition einer Funktion...

- Funktionsargumente können *default*-Werte haben. Argumente mit *default*-Wert müssen nach den Argumenten ohne *default* stehen.

```
def Nullstellensuche(x0, x1, eps=1.e-8, MaxIter=1000):
    ...
    ...
z = Nullstellensuche(a, b)
z = Nullstellensuche(a, b, 1.e-10)
z = Nullstellensuche(a, b, 1.e-10, 2000)
```

Funktionen

Beim Aufruf einer Funktion...

- Funktionsargumente können statt durch ihre Position in der Argumentliste durch ihren Namen identifiziert werden.
  Benannte Argumente müssen nach den Argumenten ohne Namen stehen.

```
def func(x0, x1, x2, eps=1.e-8, MaxIter=1000):
    ...
# die folgenden Aufrufe sind alle äquivalent:
z = func(a, b, b2, 1.e-10 )
z = func(eps=1.e-10, x1=b, x2=b2, x0=a)
z = func(a, eps=1.e-10, x1=b, x2=b1)
```


Funktionen

Funktionen sind ganz normale Objekte

- Sie können von Funktionen erzeugt und als Ergebnis `return`t werden.

```
>>> def generate_add_func(x):
...     def addx(y):
...         return x+y
...     return addx
...
>>> h = generate_add_func(4)
>>> h(1)
5
>>> h(2)
6
>>> h(10)
14
>>>
```

Funktionen

Funktionen sind ganz normale Objekte

- Sie können als Argumente an Funktionen übergeben werden.

```
>>> def Riemann_integrate(f, a, b, NInter=1000):
...     delta = (b-a)/NInter
...     s = 0
...     for i in range(NInter):
...         s += delta * f(a + delta/2 + i * delta)
...     return s
...
>>> from math import sin, pi
>>> Riemann_integrate(sin, 0, pi)
2.0000008224672694
>>> Riemann_integrate(sin, 0, pi, NInter=3000)
2.00000009138523
>>> Riemann_integrate(sin, 0, pi, NInter=300)
2.000009138551823
>>> Riemann_integrate(sin, 0, 2*pi)
-3.1508609198323267e-16
>>>
```


Funktionen und globale Variablen

Globale Variablen sind innerhalb von Funktionen lesbar:

```
my_pi = 3.14

def umfang(r):
    u = 2 * my_pi * r
    return u
```

Um globale Variablen innerhalb einer Funktion zu modifizieren, müssen sie als global deklariert werden (ansonsten wird eine lokale Variable gleichen Namens angelegt):

```
my_pi = 3.14

def improve_my_pi():
    global my_pi
    my_pi = 3.14159

print(my_pi)      #  druckt 3.14
improve_my_pi()
print(my_pi)      # druckt 3.14159
```

Ein- und Ausgabe - Konsole

Direkte Ein- und Ausgabe: `input()` und `print()`

```
answer = input('Bitte Zahl eingeben!')    # answer ist ein string
a = int(answer)                           # Konvertierung
b = a**3
print('Die Antwort ist ', b)
```

Fehlerbehandlung:

```
from sys import exit
answer = input('Bitte Zahl eingeben!')    # answer ist ein string
try:
    a = int(answer)                  # Versuche, ob konvertierbar
except ValueError:                   # wenn nicht:
    print('Das war keine ganze Zahl!')
    exit()
b = a**3
print('Die Antwort ist ', b)
```


Stringformatierung

Ziel: „schön“ formatierte Ausgaben/Tabellen

```
x = 33
y = 44.3
print( " x= {}   y= {}".format(x, y) )
# druckt:  x= 33   y= 44.3
```

- Strings haben eine `format()`-Funktion, womit Platzhalter der Form `{}` durch Werte/Variablen ersetzt werden.
- Innerhalb des Platzhalters `{}` können Formatierungsanweisungen stehen.

```
>>> x='Klaus'
>>> "Vorname: {}; Name: {}".format(x, "von und zu Münchhausen")
'Vorname: Klaus; Name: von und zu Münchhausen'
```

rechtsbündig, mind. 20 Zeichen:

```
>>> "Vorname: {:>20}; Name: {:>20}".format(x, "von und zu Münchhausen")
'Vorname:                Klaus; Name: von und zu Münchhausen'
```

linksbündig, mind. 20 Zeichen:

```
>>> "Vorname: {:<20}; Name: {:<20}".format(x, "von und zu Münchhausen")
'Vorname: Klaus               ; Name: von und zu Münchhausen'
```

zentriert

```
>>> "Vorname: {:^20}; Name: {:^20}".format(x, "von und zu Münchhausen")
'Vorname:        Klaus        ; Name: von und zu Münchhausen'
```

rechtsbündig, auf 20 Zeichen kürzen

```
>>> "Vorname: {:>.20}; Name: {:>.20}".format(x, "von und zu Münchhausen")
'Vorname: Klaus; Name: von und zu Münchhaus'
```

rechtsbündig, mind. 12 Zeichen, auf 18 Zeichen kürzen

```
>>> "Vorname: {:>12.18}; Name: {:>12.18}".format(x, "von und zu Münchhausen")
'Vorname:        Klaus; Name: von und zu Münchha'
```

linksbündig, 12 Zeichen Platz, auf 4 Zeichen kürzen

```
>>> "Vorname: {:<12.4}; Name: {:<12.4}".format(x, "von und zu Münchhausen")
'Vorname: Klau        ; Name: von         '
```


Integers

```
>>> x=7831678
>>> " x = {}".format(x)
' x = 7831678'
```

```
>>> " x = {:20d}".format(x)        # rechtsbündig mind. 20 Zeichen
' x =              7831678'
>>> " x = {:<20d}".format(x)       # linksbündig
' x = 7831678             '
>>> " x = {:^20d}".format(x)       # zentriert
' x =       7831678       '
>>> " x = {:4d}; y= {:4d}".format(x, 2)  # Ints werden nicht abgeschnitten
' x = 7831678; y=    2'
```

Andere Basen:

```
>>> "Basis 8: {:o}; Basis 16: {:x}; Basis 2: {:b}".format(x,x,x)
'Basis 8: 35700176; Basis 16: 77807e; Basis 2: 11101111000000001111110'
```

Floats

- f-Format: Dezimalzahlen

```
>>> x = 56231.789
>>> "Res= {:>13.5f}".format(x)    # 13 Zeichen Platz, 5 Nachkommastellen
'Res=   56231.78900'
```

- e-Format: Exponentialschreibweise

```
>>> "Res= {:>13.5e}".format(x)    # 13 Zeichen Platz, 5 Nachkommastellen
'Res=   5.62318e+04'
```

- g-Format: Dezimal- oder Exponentialschreibweise, größenabhängig

```
>>> "Res= {:>13.8g}".format(x)    #  13 Z. Platz, 8 Gesamtstellen(!)
'Res=     56231.789'
```


Dateibasierte Ein- und Ausgabe

- Öffnen der Datei `f=open(...)`
- Lesen aus der Datei/Schreiben in die Datei `f.readline()/f.write()`
- Schließen der Datei `f.close()`

I/O ist ein weites Feld. Wir betrachten einige wenige der zur Verfügung stehenden Funktionen.

```
f =  open('datei.txt', 'r')
```

- 1. Argument ist der Dateiname, der auch einen Pfad enthalten kann, z.B. `'../daten/datei2.dat', r'c:\user\max\datei3'`
- 2. Argument:
  - `'r'` read
  - `'w'` write (vorhandenes überschreibend)
  - `'a'` append (fortschreibend)
- Fehlerbehandlung:

```
from sys import exit
try:
    f =  open('datei.txt', 'r')
except IOError as e:
    print(e)  # Fehlermeldung
    exit()
```


Beispiel: Datei schreiben

```
f1 = open('datei.txt','a')
f1.write('     x           y        z\n')
for i in range(100):
    ...
    f1.write( "{:20.10g} {:20.10g} {:20.10g}\n".format(x[i] ,y[i] ,z[i]) )
    ...
f1.close()
```

Bei `.write()` müssen Zeilenumbrüche `'\n'` explizit angegeben werden.

Beispiel: Datei lesen

```
f2 = open('tabelle1.txt','r')
f2.readline()  # skip first line
f2.readline()  # skip second line

               # file objects are sequences of lines
               # so we can iterate over lines
for l in f2:          #  l= '   1    3.44  5.66\n'
    Lst = l.split()  #  Lst =  ['1', '3.44', '5.66']
    n = int(Lst[0])
    x = float(Lst[1])   # type conversion
    y = float(Lst[2])
...
f2.close()
...
```

Datei sehe so aus:

```
Nummer Länge Breite
--------------------
   1    3.44  5.66
   2    6.32  7.08
   3   12.00  9.32
 usw.
```


Klassen

```
class Punkt:
    """simple class of points in the cartesian plane"""

    def __init__(self, x , y):
        self.x = x
        self.y = y

    def translate(self, deltax, deltay):
        self.x += deltax
        self.y += deltay

    def __str__(self):
        return("Punkt bei ({}, {})".format(self.x, self.y) )
```

Da Python keine Variablendeklarationen kennt, werden die Attribute eines Objekts vom Konstruktor `__init__()` angelegt.

Klassen

```
>>> p1 = Punkt(2, 3)        # hier wird __init__ aufgerufen
>>> p1.x
2
>>> print(p1)               # hier wird __str__ aufgerufen
Punkt bei (2, 3)
>>>
>>> p1.translate(-3, 4.5)
>>> print(p1)
Punkt bei (-1, 7.5)
>>>
>>> p2 = Punkt(0, -7)
>>> p2.y = 8
>>> print(p2)
Punkt bei (0, 8)
>>>
```


Vererbung & operator overloading

```
class Vec(Punkt):
    def __add__(self, other):
        return(Vec(self.x + other.x,
                   self.y + other.y ))
    #def __sub__(self, other):
    #def __mul__(self, other):
    #def __neg__(self):
    # usw...

    def __str__(self):
        return("Vektor({}, {})".format(self.x, self.y)  )
```

```
>>> v1 = Vec(7, 8)             # __init__ von Punkt geerbt
>>> v2 = Vec(-11, 2)
>>> v3 = v1 + v2               # hier wird __add__ aufgerufen
>>> print(v3)
Vektor(-4, 10)
>>> v1.translate(3, 3)         # .translate() von Punkt geerbt
>>> print(v1)                  # __str__ umdefiniert
Vektor(10, 11)
```