1. Python aufrufen

Weiter unten werden Grundstrukturen der Programmiersprache Python erläutert. Ein Python-Programm ist nichts anderes als ein Text, der gewisse syntaktische Regeln beachtet. Ein so genannter **Python-Interpreter** kann ein solches Programm ausführen.

Python-Interpreter können Sie auf zwei Weisen verwenden.

Entweder Sie erstellen den Programmcode als Textdatei mit der Endung `.py`, als beispielsweise `programm.py` und geben in einem Terminal-Fenster (Doppelklick auf das Terminalsymbol)

```
python programm.py
```

ein (alle Zeilen, hier und im Folgenden, sind mit der Eingabetaste abzuschließen). Anschließend führt der Python-Interpreter den Code aus.

Oder aber Sie starten durch `python` den interaktiven Python-Intepreter. Dann können Sie Python-Befehle oder -Programme direkt eingeben. Dieser interaktive Modus ist besonders gut dazu geeignet, Befehle auszuprobieren, ist aber bereits nützlich als 'Taschenrechner'.

Bemerkung: Wenn Sie Ihren eigenen Computer dabei haben und eine Internetverbindung haben, können Sie die manche der folgenden Übungen mit Hilfe der Website
`http://labs.codecademy.com`
machen. Wenn Sie dort 'Python' wählen, erscheint rechts ein interaktiver Python-Interpreter und links ein Editor für Programme. Mit Hilfe des 'Run'-Knopfes können Sie die Programme, die Sie im Editor erstellen, ausführen.

Versuchen Sie, das folgende Beispiel zu verstehen:

```
>>> a=5.2
```

```
>>> b=a*a
>>> b
27.04000000
>>> 2**100
1267650600228229401496703205376L
>>>z=1+1.j
>>>z*z
2j
>>>z**3
-2+2j
>>> 1/z
0.5-0.5j
```

Aufgabe: Berechnen Sie `(1/2**0.5*(1+1j))**2`. Wie ist die Multiplikation komplexer Zahlen definiert und was bedeutet sie geometrisch? Welches Ergebnis hätten Sie erwartet?

Sie können im interaktiven Python-Interpreter auch eine externe Programm-Datei ausführen

```
>>>execfile("programm.py")
```

Wenn Sie beim Starten des interaktiven Python-Interpreters bereits ein Programm ausführen wollen, geht das so:

```
python -i programm.py
```

Steht in einem solchen Programm die Zeile

```
2**10
```

so wird zwar etwas berechnet, man sieht aber nichts. Um das Ergebnis der Berechnung anzuzeigen muss man das durch eine `print`-Anweisung machen, etwa so:

```
print "Das Ergebnis ist : ", 2**10
```

In eine `print`-Anweisung können Sie alle möglichen durch Kommata getrennten Objekte schreiben. Der Python-Interpreter weiß, wie diese darzustellen sind. (Man kann die Formatierung etwa von Zahlen genauer bestimmen, aber das brauchen Sie einstweilen nicht.)

Aufgabe:
Um das Ausführen von Programmen zu testen, sollen Sie ein kleines Programm erstellen. Geben Sie ihm einen Namen, der mit ihren Namen anfängt, also etwa in meinem Fall `stefanbornprog.py`. Öffnen Sie ein so genanntes Terminal und geben Sie in der Kommandozeile `geany`, Leerzeichen, Programmname ein, also etwa in meinem Fall:

```
geany  stefanbornprog.py
```

Schreiben Sie nun ein kleines Programm, das mit Hilfe von `print`-Anweisungen etwas ausgibt. Speichern Sie die Datei mit Strg+S und verlassen Sie `geany` anschließend mit Strg+Q.
Probieren Sie die beiden Weisen aus, ihr Programm aufzurufen. Was passiert, wenn Sie von der Kommandozeile anstelle von `python -i Programmname` nur `python Programmname` eingeben?

2. Module

Für alle möglichen Zwecke gibt es Sammlungen von nützlichen Funktionen und Strukturen, die in so genannten **Modulen** gebündelt werden. Durch

```
import numpy
```

wird zum Beispiel das Modul numpy geladen, das Funktionen zum numerischen Umgang mit Matrizen, Gleichungssystemen, Differentialgleichung und vieles mehr enthält. Anschließend kann man auf die Funktionen des Moduls so zugreifen:

```
numpy.sin(numpy.pi)
```

berechnet den Wert der von **numpy** zur Verfügung gestellen Sinus-Funktion an der Stelle π, wie sie von **numpy** zu Verfügung gestellt wird. Da die Funktionswerte wie auch π hier nur durch Fließkommazahlen genähert werden, ist das Ergebnis übrigens nicht 0, sondern nur *sehr klein*: $1.22464e-16$, d.h. $1.22464 \cdot 10^{-16}$.

Um nicht immer *numpy.*... schreiben zu müssen, kann man die von Numpy zur Verfügung gestellten Befehle auch direkt importieren

```
from numpy import *
a=sin(pi)
```

Das birgt allerdings Risiken, wenn Sie noch andere Module importieren, die möglicherweise Funktionen mit denselben Namen enthalten

Die dritte Form des Imports ermöglicht ihnen, den Modulnamen abzukürzen

```
import numpy as np
a=np.sin(np.pi)
```

Wenn Sie herausfinden wollen, ob es für irgendeine Aufgabe, die Sie lösen wollen, geeignete Module gibt, können Sie im Python Package Index nachsehen. (`pypi.python.org`).

3. Variablen und Zuweisungen

3.1 Einfache Variablentypen

Eine Variable belegt einen gewissen Speicherplatz. Sie kann für eine Zeichenkette, für eine ganze Zahl, für eine Fließkommazahl, für eine Tabelle solcher Objekte und manches andere stehen.

Durch

```
a=2.0
Hausnummer=333
c="Katze"
```

wird eine Fließkommavariable mit Namen `a` mit Wert 2.0, eine Ganzzahlvariable `Hausnummer` mit Wert 3 und eine Zeichenkettenvariable `c` mit Wert „Katze“ angelegt. (Achtung! Groß- und Kleinbuchstaben werden unterschieden. `hausnummer` wäre eine andere Variable.)

Wenn Sie anschließend

```
a=4
```

eingeben, wird die alte Fließkommavariable `a` gelöscht und stattdessen eine Ganzzahlvariable angelegt.

Durch

```
d=float(b)
```

wird die ganze Zahl `b` in eine Fließkommazahl umgewandelt und deren Wert in der Variable `d` gespeichert.

Für Zahlen sind die üblichen Rechenoperationen definiert. Für spezielle Funktionen müssen Sie ein geeignetes Modul importieren. Sehen Sie sich die folgenden Beispiele genau an. Fragen Sie bei allem, was unklar ist. Was fällt Ihnen auf?

```
>>>a=3
>>>b=5
>>>c=3.
>>>b/a
1
>>>b/float(a)
1.6666666666666667
>>>b/c
1.6666666666666667

>>>import numpy as np
>>>np.sin(a)
0.14112000805986721

>>>import math
>>>math.sin(a)
0.1411200080598672
```

Eine Warnung: Wenn zwei ganze Zahlen dividiert werden, so wird in Python 2.7 nicht der Dezimalbruch, sondern das Ergebnis bei ganzzahliger Division mit Rest berechnet. Wollen Sie den Dezimalbruch, so müssen Sie eine der beiden Zahlen explizit zur Fließkommazahl umwandeln. – Dieses Verhalten ist ärgerlich und eine häufige Fehlerquelle. In Python 3.0 ist es behoben, aber viele wissenschaftliche Pakete gibt es bisher nur für Python 2.7., so dass wir dieses verwenden. Um aber doch dieses Verhalten zu erzwingen, können Sie in die erste Zeile eines Programms das Folgende schreiben:

```
from __future__ import division
```

Danach verhält sich / wie die gewöhnliche Division, während das Symbol // die Division mit Rest bezeichnet.

Aufgabe:

1. Berechnen Sie 2^{1000}. Wieviele Stellen hat 2^{1000}?

2. Bestimmen Sie das Volumen einer Kugel mit Radius 5 ($\frac{4}{3}\pi r^3$). (Achtung: Das Ergebnis ist *nicht* 392.7.)

3.2 Zeichenketten (Strings)

```
>>>c='Katze'
```

Durch `c[0]` greift man auf den ersten Buchstaben von „Katze“zu, `c[1]` auf den zweiten, etc. Die Länge der Zeichenkette erhält man durch `len(c)`. Das letzte Zeichen von *c* hat den Index `len(c)-1`, also liefert `x[len(c)-1]` den Wert “e”. Sie können aber das letzte Zeichen auch so erhalten: `x[-1]`, das vorletzte `x[-2]`, etc. Ein negativer Index wird so interpretiert, dass vom Ende der Zeichenkette zu zählen ist.

Für Zeichenketten bedeutet + das Aneinanderhängen der Zeichenketten. Weil das Wort *Strings* gebräuchlicher und kürzer ist, nenne ich die Zeichenketten von jetzt an Strings.

```
>>>wort='futter'
>>>c+'n'+wort
'Katzenfutter'
```

Aufgabe: Probieren Sie aus, was passiert, wenn man eine ganze Zahl und einen String durch das Multiplikationszeichen $*$ verbindet.

Es gibt natürlich viel mehr Stringoperationen, die man nachschlagen kann, sobald man sie braucht. Es sollen noch einige erwähnt werden, weil wir sie bald gebrauchen können.

Ist `s` ein String, der am Ende Leerzeichen enthält oder `\n` (’newline’), bzw. `\r`(’carriage return’), so liefert die Funktion `s.rstrip()` einen String zurück, aus dem diese unsichtbaren Zeichen entfernt wurden. (Die Zeichen `\r` und `\n` trennen in gewöhnlichen Textdateien die Zeilen voneinander. Deswegen treten diese Zeichen beim Lesen solcher Dateien für gewöhnlich auf.)

Die Funktion `s.replace(alt,neu)` liefert einen String, in dem alle Vorkommen des Strings `alt` im String `s` durch den String `neu` ersetzt

wurden. Die Funktionen `s.upper` und `s.lower` liefern den String in Groß- bzw. Kleinbuchstaben umgewandelt zurück:

```
>>>satz='Kaiser Wilhelm reitet durch den Tiergarten.'
>>>satz2=satz.replace('Kaiser Wilhelm','Kanzlerin Merkel')
>>>satz3=satz2.replace('e','u')
>>>satz
'Kaiser Wilhelm reitet durch den Tiergarten.'
>>>satz2
'Kanzlerin Merkel reitet durch den Tiergarten.'
>>>satz3
'Kanzlurin Murkul ruitut durch dun Tiurgartun.'
>>>satz3.upper()
'KANZLURIN MURKUL RUITUT DURCH DUN TIURGURTUN.'
```

Aufgabe: Schreiben Sie ein kleines Programm, dass eine Zeichenkette einliest, alle Vokale entfernt und die Zeichenkette anschließend wieder ausgibt. Dazu müssen Sie noch wissen, wie Sie eine Zeichenkette einlesen: `s=raw_input('Geben Sie einen Text ein: ')`

3.3 Listen

Andere Variablentypen, die wir benötigen werden, sind Listen. Auf Elemente von Listen kann man wie bei Strings zugreifen, aber die Elemente von Listen können einen beliebigen Typ haben:

```
>>>kohl=["Weißkohl", "Rotkohl", "Wirsing"]
>>>kohl[0]
"Weißkohl"
>>>kohl[2]
"Wirsing"
>>>kohl[-1] #  von hinten gezählt!
"Wirsing"
```

Sie können auch Listen von Listen definieren, Elemente einfügen und entfernen, etc. Darüber wäre an anderer Stelle zu reden, hier nur einige Beispiele, zunächst ein Beispiel, in dem wir als neues Element der Liste `kohl` die Liste `[1,2,3]` anhängen.

```
>>>l=[1,2,3]
>>>kohl.append(l)
["Weißkohl", "Rotkohl", "Wirsing",[1,2,3]]
```

Wir können aber eine Liste um eine weitere Liste erweitern, d.h. deren Elemente einzeln anhängen:

```
>>>kohl.extend(l)
["Weißkohl", "Rotkohl", "Wirsing",[1,2,3],1,2,3]
```

Die Zugehörigkeit zu einer Liste prüft man mit `in`:

```
>>>"Rotkohl" in kohl
True
>>>"Kartoffel" in kohl
```

```
False
>>> 1 in kohl
False
>>> [1,2,3] in kohl
True
```

Durch `range(a,b)` wird die Liste aller ganzen Zahlen i mit $a \leq i < b$ erzeugt, also wird etwa durch `l=range(0,10)` die Liste `[0,1,2,3,4,5,6,7,8,9]` erzeugt. Wichtig sind noch Ausdrücke, die einen Teil einer Liste liefern. Ist l eine Liste, so liefert `l[a:e]` die Liste aller Einträge von l mit Index i, wobei $a \leq i < e$. Beachten Sie die Zeichen $\leq$ und $<$! Lässt man a oder e weg, so fällt die enstprechende Bedingung weg, also liefert `l[:e]` alle Einträge von l mit Index kleiner e, sowie `l[a:]` alle Einträge von l mit Index $\geq a$. Wollen Sie in einem solchen Indexbereich nur jedes s. Zeichen, so schreiben Sie `l[a:e:s]`.

Ein Beispiel mit der Liste `range(0,10)`:

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

Aufgabe: Erzeugen Sie eine Liste, die alle durch 13 teilbaren Zahlen, die kleiner als 10000 sind, enthält. Sehen Sie sich die ersten 100 und die letzten 100 Elemente dieser Liste an.

Eine Operation, die einen String in eine Liste von Strings umwandelt, soll noch erwähnt werden, weil wir sie gleich brauchen. So bricht man einen String an gewissen Trennzeichen in eine Liste von Strings auf:

```
>>>s='Hund,Katze,Maus'
>>>s.split(',')
['Hund','Katze','Maus']
```

Aufgabe:

1. Öffnen Sie den Ordner `ss2013/Einfuehrung`. Dort befindet sich die Datei `wortliste1_template.py`. Öffnen Sie diese mit dem Editor `geany` und speichern Sie sie unter einem neuen Namen, der Ihren Namen oder Initialen enthält, etwa für mich `wortliste1_stefanborn.py`.

   Der bereits geschriebene Teil des Programms liest die ganzen Buddenbrooks in einen String. Vervollständigen Sie das Programm, so dass es eine Wortliste erstellt. Das Programm können Sie mit der Taste F5 (oder über das Menu) ausführen.

2. Anschließend bleiben Sie im interaktiven Python-Interpreter, wo Sie sich die Variablen ansehen können. Sehen Sie sich beispielsweise die Wörter 1000 bis 1030 an oder 2000 bis 2030. Sind Sie mit der Wortliste zufrieden? Wenn nein, überlegen Sie sich Verbesserungen und testen Sie diese.

3.4 Wörterbücher

Wörterbücher oder *Dictionaries* sind ein praktischer Datentyp, um irgendwelchen Daten ('keys') irgendwelche anderen Daten zuzuordnen ('values'). Ein Wörterbuch wird durch `w={}` angelegt, ist aber zunächst leer. Nun kann man durch `w[key1]=value1` ein Paar `key1:value1` dem Wörterbuch hinzufügen, durch `w[key2]=value2` ein weiteres Paar `key2:value2`, etc. Durch `w[key1]` erhält man anschließend den Wert `value1`, etc. Ein Beispiel:

```
>>>w={}
>>>w[11]=[2.444]
>>>w['Katze']=75
>>>w['Hund']=['Maus']
>>>w[1.23]=[44]
>>>w
{11: [2.444], 'Katze': 75, 'Hund': ['Maus'], 1.23: [44]}
>>>w['Hund']
['Maus']
```

Als *Keys* kommen Strings, ganze Zahlen, Bruchzahlen und Objekte anderer grundlegender Datentypen in Frage (aber beispielsweise keine Listen.)

Ob ein gewisser *key* im Wörterbuch vorkommt, lässt sich so abfragen:

```
>>> 11 in w
True
>>> 'Maus' in w
False
```

denn 'Maus' ist kein *key*, sondern ein *value*.

3.5 Vektoren, Matrizen

Für numerische Berechnungen werden wir meistens die Module **numpy** (Numerical Python) und **scipy** (Scientific Python) benutzen, die uns ermöglichen, Matrizen, Vektoren, Tensoren als so genannte „Arrays “ zu definieren:

```
>>>import numpy as np
>>>A=np.zeros([2,3])
>>>A
array([[0.,0.,0.],[0.,0.,0.]])
>>>B=np.array([[1,2,3],[4,5,6]])
>>>v=np.array([1,4,2.3])
>>>
```

`A` ist dann eine 2×3-Matrix, die Nullen (als Fließkommazahlen) enthält, `B` ist eine solche Matrix, die in der ersten Zeile $1, 2, 3$, sowie in der zweiten $4, 5, 6$ enthält. `v` ist ein Vektor, dessen 3 Einträge 1.0, 4.0 und 2.3 sind.

Aufgabe: Probieren Sie beispielsweise die Matrizenmultiplikation in numpy aus. Starten sie dazu in einem Terminal 'python' und laden Sie das Modul numpy.

```
import numpy as np
```

Erzeugen Sie Matrizen oder Vektoren mit passenden Dimensionen. Mit `np.dot(A,B)` werden zwei Matrizen multipliziert. (Ein Vektor ist ein Spezialfall einer Matrix mit nur einer Spalte.)
Suchen Sie die Seite

`http://docs.scipy.org/doc/numpy/reference/routines.linalg.html`

auf, die eine Auflistung der in Numpy eingebauten Funktionen aus dem Gebiet der linearen Algebra enthält. Wollen sie noch etwas ausprobieren?

An dieser Stelle scheint es angebracht, ein Beispiel dafür zu geben, wie man Daten graphisch darstellt. Wir können N verschiedene x-Werte in

einem Numpy-Vektor `x` speichern, die zugehörigen y-Werte in einem Numpy-Vektor `y`. Das Modul `matplotlib` stellt eine Methode zur Verfügung, mit der man die zugehörigen Punkte in einem Koordinatensystem plotten kann. Sehen Sie sich das Beispiel `plottest.py` an und lassen Sie es laufen.

4. Programmieren

Fürs Programmieren ist es wesentlich, das Programm in übersichtliche Stücke zu unterteilen und den **Programmfluss** zu steuern.

Eine in diesem Zusammenhang wichtige Struktur von Python ist ein so genannter **Block**. Ein Block ist, anders als in vielen anderen Programmiersprachen, durch eine gemeinsame Einrückungstiefe definiert

```
....
....
    .......
    .......    Diese vier Zeilen
    .......    bilden einen Block
    .......
...
...
```

Auch die ganzen acht Zeilen bilden einen Block, in dem eben der kleiner Block enthalten ist. Einrücken können Sie entweder mit der Tabulator-Taste oder mit Leerzeichen. Mischen Sie auf keinen Fall beides, das kann zu Problemen führen.

4.1 Bedingungen

Wenn Sie eine Folge von Befehlen nur dann ausführen wollen, wenn eine gewisse Bedingung erfüllt ist, und sonst eine andere Folge von Befehlen, so geht das so:

```
if Alter>120:
   print "Da stimmt was nicht, so alt kann man nicht werden."
elif Alter >100:
   print "Gratuliere!"
   print str(Alter)+" Jahre ist ein biblisches Alter"
else:
   print "Gratuliere zum "+str(Alter)+ ". Geburtstag"
```

Nach dem Doppelpunkt beginnt in der nächsten Zeile jeweils ein tiefer eingerückter Block. Genau dieser Block wird ausgeführt, wenn die Bedingung vor dem Doppelpunkt erfüllt ist.

Als Bedingung kann jeder Ausdruck dienen, der wahr oder falsch sein kann, also etwa Gleichungen, Ungleichungen etc. Wahrheitswerte haben einen besondere Typs 'bool', der nur die Werte 'True' und 'False' zulässt.

```
>>> 1==2
False
>>> 1==1
True
>>> 1<=2
True
>>> 1>2
False
>>> 'a' in ['a','b','c']
True
>>> x=(1==2)
>>> x
False
# x ist eine Variable des Typs 'bool':
>>>type(x)
<type 'bool'>
```

Achtung: Da das Gleichheitszeichen „=“für Zuweisungen von Variablen verwendet wird, gibt es ein anderes Symbol, um die Gleichheit von zwei Ausdrücken abzufragen: zwei Gleichheitszeichen ==. Diese beiden darf man nicht verwechseln (kann es aber auch nicht, das es üblicherweise eine Fehlermeldung gibt, wenn man sie verwechselt hat.)

4.2 for-Schleifen

Häufig ist es nötig, mit allen Einträgen einer Liste oder eines Vektors eine

Operation vorzunehmen. Dafür sind so genannte `for`-Schleifen geeignet. Ist `liste` eine Liste, so führt

```
for x in liste:
    ....
    ....
    ....
```

eine Schleife über alle Elemente `x` der Liste `liste` aus.

Indem wir die Liste `range(a,b)` aller ganzen Zahlen n mit $a \leq n < b$ verwenden, können wir eine Schleife über diese Zahlen durchführen:

```
import numpy
x=numpy.array([1.0,2.0,3.0,5.0])
y=numpy.zeros(4)
for i in range(0,len(x)):
    y[i]=x[i]*x[i]
    print y[i], " ist das Quadrat von ", x[i]
.
.
.
```

Wir berechnen die Quadrate der Einträge des Vektors `x` und bilden daraus den Vektor `y`.

Ein Beispiel mit der Liste `kohl` vom Anfang, zusammen mit der Ausgabe, die es hervorbringt:

```
>>>for sorte in kohl:
>>>    print sorte+ " ist wohlschmeckend und gesund"
"Weißkohl ist wohlschmeckend und gesund",
"Rotkohl ist wohlschmeckend und gesund",
"Wirsing ist wohlschmeckend und gesund"
```

Aufgabe: Zählen Sie die Häufigkeit der Wörter in der weiter oben erzeugten Wortliste mit Hilfe eines Wörterbuchs. Falls Ihr eigenes Programm noch nicht funktioniert, können Sie auch die Datei `wortliste_wb_template.py` zur Grundlage nehmen – bitte wieder mit Ihren Namen oder Initialen umbenennen, also etwa `wortliste_wb_hanspeter.py`

4.3 while-Schleifen

Man kann auch einen Block von Anweisungen solange durchführen, wie eine gewisse Bedingung erfüllt ist. Wir realisieren dieselbe Schleife wie oben in dieser Weise:

```
import numpy
x=numpy.array([1.0,2.0,3.0,5.0])
y=numpy.zeros(4)
i=0
while i<4:
    y[i]=x[i]*x[i]
    print y[i], " ist das Quadrat von ", x[i]
    i=i+1
.
.
.
```

Die Variable `i` durchläuft den Block in der Schleife mit den Werten 0,1,2,3. Zuletzt wird `i` auf 4 erhöht, die Bedingung ist nicht mehr erfüllt, und das Programm fährt mit dem Code fort, der nach dem Block kommt.

Aufgabe: Schreiben Sie ein Programm, das den Benutzer bittet, eine natürliche Zahl n einzugeben und anschließend die zugehörige Collatz-Folge berechnet und die ersten 50 Glieder auszugeben. Die Folge ist rekursiv definiert: $a_0 = n$,

$$a_{k+1} = \begin{cases} \frac{1}{2}a_k, & \text{falls } a_k \text{ gerade} \\ 3a_k + 1 & \text{sonst.} \end{cases}$$

Zum Eingeben der ganzen Zahl verwenden Sie:
`n=int(raw_input('Geben Sie eine natürliche Zahl ein: '))` Durch int() wird beispielsweise die Zeichenkette '1231' in die ganze Zahl 1231 umgewandelt. Diese Umwandlung brauchen wir, da `raw_input` nur eine Zeichenkette liefert. Sie könnten stattdessen auch `input` verwenden.
Testen Sie das Programm für einige Anfangswerte. Was fällt Ihnen auf?

5. Ein Modellierungsbeispiel: Schall

i) Kurze Einführung: Was ist Schall überhaupt?

ii) `python -i schallwerkzeuge.py`

ein. Geben Sie als nächstes

```
y=recordsnd(None,4)
```

ein. Sie können nun nach einem weiteren Drücken der Eingabetaste ein 4 Sekunden langes Stück Schall aufnehmen. Was Sie erhalten, ist eine Liste von Zahlen (ein „Vektor“, bzw ein „array“).

$$y[0],\ldots,y[ANZAHL-1]$$

aus reellen Zahlen. Pro Sekunde werden RATE viele Werte gemessen. Dabei ist jedes $y[i]$ eine reelle Zahl zwischen -1 und 1.

Um sich ein Bild zu machen, sehen Sie sich die Daten mal an (die x-Koordinate ist die Zeit in Sekunden, die y-Koordinate ist durch unseren Vektor gegeben.)

```
inspectsnd(y)
```

Sie können mit dem Lupensymbol Teile des Graphen vergrößern und mit dem ’Home’-Symbol wieder zur ursprünglichen Darstellung zurückkehren.

Nehmen Sie nun auf diese Weise verschiedene Geräusche auf und inspizieren Sie das Signal. Können Sie verschiedene Geräusche an ihrem Graphen unterscheiden.

Wenn Sie sich das Geräusch noch einmal anhören wollen, können Sie das mit

```
playsnd(y, RATE)
```

tun. Mit

```
playsnd(y, 2*RATE)
playsnd(y, RATE//2)
```

können Sie es sich doppelt, bzw. halb so schnell anhören.

iii) Das im Intervall $[0,T]$ am Lautsprecher eintreffende Schallsignal lässt sich nach Wahl von Einheiten als Funktion $f : [0,T] \to \mathbb{R}$ verstehen. Solche Funktionen kann man addieren und mit reellen Zahlen multiplizieren. Bezüglich dieser Operationen bilden diese Signale einen *Vektorraum* V. Ebenso bilden die diskretisierten Signale $\tilde{f} : \{0,\ldots,ANZAHL-1\} \to \mathbb{R}$ einen Vektorraum V_d. Die Messung ist eine lineare Abbildung $V \to V_d$. Wenn man davon spricht, dass „sich zwei Schallsignale überlagern“, ist nichts anderes gemeint als die Summe.

Da die Übertragung eines Schallsignals von einem Ort P zu einem Ort Q (mit allen Reflexionen, Schwächungen etc.) ebenfalls eine lineare Abbildung ist, lässt sich aus einer Zerlegung des Ausgangssignals als Summe auch eine solche Zerlegung des Endsignals gewinnen.

Versuchen Sie einmal das Folgende:

```
ya=recordsnd{None, 6)
yb=recordsnd(None, 6)
playsnd((ya+yb)/2,RATE)
playsnd((10*ya+yb)/11, RATE)
playsnd((ya+10*yb)/11,RATE)
```

iv) Erzeugen Sie nun ein kurzes Knackgeräusch (wie erzeugen Sie einen besonders kurzen Knall?). Gelingt es Ihnen, an dem Graphen die Zeit ausfindig zu machen, die bis zum ersten Echo vergeht? Schlagen Sie einen ungefähren Wert für die Schallgeschwindigkeit nach, um diese Zeit auf eine Entferung umzurechnen. Sie können den Computer zu

anderen Stellen des Gebäudes tragen, um zu sehen, wie sich das Echo verändert.

v) Nehmen Sie nun eine Stimmgabel auf, die sie kurz vor die kleine Mikrofone, oben über dem Bildschirm der Laptops halten müssen.

Wie sieht das Signal aus? Warum hat es einen Sinn, von der Frequenz zu sprechen? Bestimmen Sie diese.

Wiederholen Sie diese Bestimmung der Frequenz drei Mal und notieren Sie die Frequenz.

vi) Starten Sie das Programm `dualscope.py`, das ein Oszilloskop zur verfügung stellt. (im Terminal: `python dualscope.py`). Experimentieren Sie damit rum.

vii) Programmieren Sie eine Funktion `entrausche(y)`, die ein Signal zurückgibt, das weniger rauscht. (Machen Sie zunächst ein kleines „Brainstorming“ mit Ihrer Gruppe, um auf eine Idee zu kommen, was es bedeuten kann, das Rauschen zu vermindern. Um diese dann umzusetzen, müssen sich dafür erinnern, wie man Funktionen in `python` definiert, und an einiges mehr: Fragen Sie Dozenten und Tutoren, wenn Ihnen was fehlt.)

6. Funktionen

6.1 Grundlagen

Ein besonders wichtiges Hilfsmittel zur Strukturierung von Programmen ist die Definition von Funktionen.

```
def f(x,y):
    if x<y:
        return x*x
    else:
        return 0
```

Mit `return ...` wird die Größe definiert, die anschließend als Wert der Funktion zurückgegeben wird. Hier liefert `f(2,3)` den Wert 4 und `f(3,2)` den Wert 0. Funktionen müssen aber nicht unbedingt einen Wert zurückgeben, sie können auch einfach etwas tun.

```
def  lobe(s)
    print s+ " ist besonders wohlschmeckend."

for sorte in kohl:
     lobe(sorte)
```

Alle Variablen, die Sie in einer Funktion verändern oder zuweisen, sind lokal, d.h. die Zuweisung wirkt sich nicht außerhalb der Funktion aus, wenn dort eine Variable denselben Namen trägt. Wenn Sie tatsächlich Variablen von außerhalb verändern wollen, müssen Sie sie dass durch `global variablenname` deklarieren.

Aufgabe:
Öffnen Sie das Programm `test_lokal.py`, führen Sie es aus und interpretieren Sie das Ergebnis.

6.2 Argumente

Wenn einer Funktion Argumente übergeben werden, so werden diese im Falle von Zahlen, Strings und einigen einfachen Typen als Kopie übergeben („call by value“), bei Listen, Arrays, Mengen etc. dagegen als Verweis („call by reference). Wenn Sie also eine übergebene Liste verändern, verändert sich die ursprüngliche Liste. Achtung: Eine Argumentvariable kann man nicht in derselben Funktion als `global` deklarieren.

Aufgabe:
Öffnen Sie das Programm `test_parameter.py`, führen Sie es aus und interpretieren Sie das Ergebnis. Geben sie anschließend `funktion1(3)` ein. Was geschieht?

Da Funktionen sich auch selbst aufrufen können, ermöglichen sie das einfache Berechnen von rekursiv definierten Größen.

Aufgabe:
Die Fibonacci-Folge ist definiert durch $a_0 = 1$, $a_1 = 1$, $a_{n+2} = a_n + a_{n+1}$. Schreiben Sie eine Funktion `fib(n)`, die Ihnen a_n liefert (falls Sie dies mit dem 'virtuellen Computer' des Labors machen, geben Sie dem Programm wie immer einen Namen, der Ihre Namen oder Initialen enthält, z.B. `fibonacci_sb.py`) Berechnen Sie mit dieser Funktion a_{10}, a_{20}, a_{30} und a_{35}. Was fällt Ihnen auf? Analysieren Sie die Ursache des Problems und schreiben Sie eine zweite Funktion, die diesen Fehler nicht hat.

6.3 Weiteres zum Programmieren mit Funktionen

Um wirklich mit Funktionen umgehen zu können, eine komplexe Aufgabe in einfachere Aufgaben zu zerlegen und alles, was mehrfach vorkommt, wieder in eigene Funktionen zu verlagern. Um das zu üben, folgende Aufgabe:

Aufgabe: [nach Allen B. Downing, Programmieren lernen mit Python] Laden Sie das Programm `polygon_template.py` und speichern Sie es wieder unter Ihrem eigenen Namen, also etwa `polygon_sb.py` in meinem Fall. Führen Sie es durch: Es öffnet eine Fenster mit einer 'Schildkröte'. Wenn Sie sich den Code ansehen, so wird ein Objekt des Typs `Turtle` namens `tim` erzeugt. (Zu Klassen und Objekten kommen wir später.) Es wird eingestellt, welche Zeit 'tim' zwischen den einzelnen Aktionen wartet.
Diese Objekt hat nun verschiedene Methoden, also Funktionen, die Sie aufrufen können, indem Sie deren Namen mit einem Punkt an den Namen des Objekts anhängen:

```
tim.fd(x)   #'forward'  bewegt tim um x Einheiten vorwärts
tim.lt()    #'left turn' dreht tim um 90 Grad nach links
tim.rt()    #'right turn' dreht tim um 90 Grad nach rechts
tim.lt(x)   #'left turn' dreht tim um x Grad nach links
tim.rt(x)   #'right turn' dreht tim um x Grad nach rechts
tim.pd()    #'pen down'   versetzt tim in den Schreibmodus
tim.pu()    #'pen up' beendet den Schreibmodus
```

Programmieren Sie

1. eine Funktion `quadrat`, die ein Quadrat der Seitenlänge x zeichnet,
2. eine Funktion `polygon`, die ein Polygon mit n Ecken und der Seitenlänge x zeichnet.
3. eine Funktion `kreis`, die einen (angenäherten) Kreis mit Radius r zeichnet.
4. eine Funktion `bogen`, die einen Kreisbogen mit Radius r und Winkel *winkel* zeichnet.

7. Klassen und Objekte

7.1 Methoden und Attribute, Konstruktoren

Die 'Schildkröte' war ein Beispiel eines Objekts, das etwas *tun* kan, vorwärts gehen, zeichnen, etc. ('Methoden'), dass aber auch innere Zustände ('Attribute') hat, auf die es dabei zugreif. Wenn Sie nochmals Ihr `polygon...`-Programm öffnen und durchführen, können Sie mit `tim.x` und `tim.y` den gegenwärtigen Ort abfragen. Diese Kapselung der Daten und der Funktionen in ein Objekt liegt in der Natur vieler Probleme, die wir lösen wollen.

In `toene_template.py` wird die Klasse `ton` definiert. Sie hat zunächst eine Methode `__init__` und eine Methode `abspielen`, die noch nichts tut:

```
class ton():

   skala={'c':0,'cis':1,'d':2,'dis':3,'e':4,'f':5,'fis':6,'g':7}

   grundton=220*2**0.25    # kleines c

   def __init__(self,wert,start,dauer,lautstaerke):
        self.wert=wert
        self.start=start
        self.dauer=dauer
        self.lautstaerke=lautstaerke
        self.frequenz=ton.grundton*2**(ton.skala[self.wert]/12.)

   def abspielen(self):
        pass
```

Wenn Sie nun ein Objekt ('eine Instanz') der Klasse `ton` erzeugen wollen, so geht das so:

```
t=ton('a',0.,2.,1.)
```

Dabei wird Zunächst ein abstraktes Objekt erzeugt und an die Variable `t` gebunden. (Achtung: Für alle Methoden, die wir in der Objektdefinition

geben, heißt das jeweilige Objekt `self`. Für `self` setzt der Interpreter immer das Objekt ein, um dass es gerade geht, also hier `t`.)

Anschließend wird die Methode `ton.__init__(self, 'a',0.,2.,1.)` aufgerufen, der so genannte *Konstruktor*. Nun werden die Attribute des Objekts `self` gemäß den übergebenen Werten zugewiesen. Anschließend liefert `t` das so erzeugte Objekt.

Nun können Sie durch `t.frequenz` etwa diesen Wert abfragen. Die Methoden `wavedaten` und `abspielen` tn noch nichts im Template, d.h. es geschieht nichts, wenn `t.abspielen` aufgerufen wird.

Aufgabe: Implementieren Sie zunächst die Methode `wavedaten`, sie soll eine Funktion sein, die die den Vektor der Schallintensitätsdaten mit der Samplerate `RATE` zurückgibt. Anschließend implementieren Sie die Methode `abspielen`.
Speichern Sie wieder das `toene...`-Template mit Ihrem Namen. Definieren Sie anschließend eine Klasse `lied` mit einem Konstruktor, der Methode `ton_hinzufuegen` und den Methoden `wavedaten` und `abspielen`.

7.2 Ein etwas komplexeres Beispiel

Ein *zellulärer Automat* besteht aus einer n-dimensionalen Anordnung von Zellen, die gewisse Zustände haben können, etwa 0 oder 1, oder auch ein Kontinuum von Zuständen (gegeben durch eine reelle Zahl) und einer Regel, wie sich aus einen gegeben Konfiguration von Zuständen eine neue Konfiguration ergibt. Diese Regel können wir als Beschreibung eines "Zeitschritts" verstehen. Solche zellulären Automaten sind nicht nur eine mathematisch-informatische Spielerei, sondern Sie werden auch in verschiedenen Wissenschaften als Modelle verwendet. Leider gibt es sehr wenige mathematische Sätze, die beschreiben, wie sich ein Automat mit einer gewissen Regel verhält. In den meisten Fällen kann man das Verhalten nur 'empirisch' durch Simulation untersuchen.

Ein berühmter zellulärer Automat, der allerdings nur theoretische Bedeutung hat, ist Conways 'Game of Life'.

Ein zellulärer Automat verlangt förmlich nach einer objektorientierten Umsetzung, denn er hat Zustände ('Attribute') und eine Dynamik (die durch 'Methoden' umgesetzt wird).

Aufgabe: Lesen Sie sich bitte

`http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens` durch, insbesondere die Regel für die Berechnung des neuen Zustands. Laden Sie nun die Datei `Life_template.py` und speichern Sie sie unter Ihrem Namen. Es gibt dort eine Klasse `Life`, die den Automaten beschreibt, und eine Klasse `LifeViewer`, die die Visualisierung leistet und uns hier nicht interessieren soll. Die Methode `step` der Klasse `Life` tut bisher noch nichts. Implementieren Sie dort die Zustandsänderung gemäß der Regel!

Im `main()`-Teil, der die Klassen testet, wird bisher nur ein zufälliges Anfangsmuster verwendet. Hier können Sie auch gezielt ein Anfangsmuster anlegen und das Verhalten beobachten.

Weitere interessante Hinweise zum Game of Life und dieser Implementierung finden Sie unter

`http://greenteapress.com/complexity/html/thinkcomplexity008.html`.