

Algorithmik kontinuierlicher Systeme

Einführung in Python - Teil 2



Übungsbetrieb

Das 1. Übungsblatt ist da!

- Nicht vergessen:
 - Die Vorlagen finden Sie auf der LSS Homepage
 - Python 3.5 verwenden!
 - Selber testen mit **python3.5 test.py**
 - Keine Plagiate, kein Betrug!
 - Abgabe über das EST
 - Bei Unklarheiten: FSI Informatik Forum



Organisatorisches

Wer öfters in die Rechnerübung möchte, kann das gerne tun
(oder wessen Übung auf einen Feiertag fällt)

Übungseinteilungen sind (im Moment) nicht verbindlich, wir
vertrauen auf Ihren gesunden Menschenverstand!



Tafelübungen

- Das Ergebnis gibt es heute im EST



Kurze Wiederholung

Zahlen

- Beliebig genaue Ganzzahlen **167896291**
- Gleitkommazahlen **1.5**
- Komplexe Zahlen, Dezimalbrüche, ...

Datenstrukturen

- Listen **[X,Y,Z]**
- Tupel **(X,Y,Z)**
- Zeichenketten **"XYZ"**
- Wörterbücher **{X:1, Y:2, Z:6}**
- Mengen **{X, Y, Z}**



Kontrollfluss

```
>>> x = 0
```

```
>>> while True:
```

```
    x += 1
```

```
    if x == 100:
```

```
        break
```

```
    elif x%2 == 0:
```

```
        continue
```

```
    print(x)
```

```
>>> for x in range(1,100,2):
```

```
    print(x)
```



Das Modulsystem

```
>>> import sys
```

```
>>> sys.getsizeof(1.5)
```

```
24
```

```
>>> import numpy as np
```

```
>>> np.array([1,2])
```

```
array([1, 2])
```

```
>>> from math import sqrt
```

```
>>> sqrt(25)
```

```
5.0
```



Das Modulsystem

Schlechter Stil:

```
>>> from numpy import *  
>>> array((2,3))
```

- Tiefer im Code weiß man nicht mehr woher array kommt
- Lädt alles mögliche ins globale namespace



Zuweisungen in Python

```
>>> a, b = 5,6
```

```
>>> a, b = b, a
```

```
>>> (a, b) = b, a
```

```
>>> (a, b) = [b, a]
```

```
>>> (a, b)
```

```
(6,5)
```

```
>>> [a, [b, c], d] = 1, (2, 3), 4
```

```
>>> (a, b, c, d)
```

```
(1, 2, 3, 4)
```

Fast alles kann zugewiesen werden, solange die „Struktur“
auf beiden Seiten gleich ist



Semantik von Referenzen und Zuweisungen



Zuweisungen kopieren nicht!

```
>>> a = [1, 2, 3]
```

```
>>> b = a
```

```
>>> a.append(4)
```

```
>>> b
```

[1,2,3,4] # Überraschung!

Fazit:

- Konstanten (Zahlen, Zeichenketten) verhalten sich wie erwartet
- Vorsicht mit veränderlichen Daten, z.B. Listen



Was passiert bei einer Zuweisung?

```
>>> a, b[i], c = DING1, DING2, DING3
```

1. Rechte Seite wird von links nach rechts ausgewertet
2. Linke Seite, von links nach rechts:
 1. Der angegebene Ort wird nachgeschlagen, oder neu im Speicher angelegt
 2. Der zugehörige Wert der rechten Seite wird an den Ort geschrieben

```
>>> a = [0, 0, 0, 0]
```

```
>>> i, a[i], i, a[i] = [0, 1, 2, 3]
```

```
>>> a # ???
```



Was passiert bei einer Zuweisung? (2)

```
>>> a = [0, 0, 0, 0]
```

```
>>> i, a[i], i, a[i] = [0, 1, 2, 3]
```

```
>>> a # ???
```

Auflösung:

```
[1, 0, 3, 0]
```

```
>>> i
```

```
2
```



Was passiert bei einer Zuweisung? (3)

```
>>> x = 'foo'
```

1. Eine Zeichenkette wird angelegt oder gesucht
2. Die Variable x wird gesucht und evtl. erzeugt
3. In der Variable x wird eine Referenz auf 'foo' gespeichert

```
>>> a = 'foo'
```

```
>>> b = 'foo'
```

```
>>> id(a)
```

```
140694280407760
```

```
>>> id(b)
```

```
140694280407760
```



Was passiert bei einer Zuweisung (4)

```
>>> x = []
```

1. Eine leere Liste wird im Speicher angelegt
2. Die Variable x wird gesucht und evtl. erzeugt
3. In der Variable x wird eine Referenz auf die neu erzeugte Liste gespeichert

```
>>> a, b = [], []
```

```
>>> id(a)
```

```
140455607120648
```

```
>>> id(b)
```

```
120455607074120
```



Verständnisfrage

```
>>> a = [[]] * 5
```

```
>>> a
```

```
[[], [], [], [], []]
```

```
>>> a[0].append(42)
```

Was ist der Wert von A?

```
>>> a
```



Antwort

```
>>> a = [[]] * 5
```

```
>>> a
```

```
[[], [], [], [], []]
```

```
>>> a[0].append(42)
```

Was ist der Wert von A?

```
>>> a
```

```
[[42], [42], [42], [42], [42]]
```

Zusammenfassung

- Solange nur konstante Datentypen verwendet werden ist alles gut
- Veränderliche Daten wie Listen werden erst zum Problem, wenn sie tatsächlich verändert werden
- Sobald Daten verändert werden:
 - Wer hat alles Referenzen auf die Daten?
 - Sollten die Änderungen gesehen werden?
 - Im Zweifel kopieren!
 - Siehe copy/deepcopy

```
>>> b = [1, 2, 3]
```

```
>>> a = b[:]
```



Funktionen - Teil 2



Klassische Funktionen

```
>>> def f(foo, bar, baz):  
    """Kommentare sind oft Hilfreich!"""  
    return (foo, bar, baz)
```

```
>>> foo(3, 4, 5)  
(3, 4, 5)
```

- Fast wie in C/Java
- in Python müssen keine Typen angegeben werden

Python kann aber noch mehr...



Standardargumente

```
>>> def f(foo, bar=2, baz=3):  
    return (foo, bar, baz)
```

```
>>> f(1, 2)
```

```
(1, 2, 3)
```

```
>>> f(1)
```

```
(1, 2, 3)
```

Es dürfen keine normalen Argumente auf Standardargumente folgen!

```
>>> def f(foo=1, bar=2, baz): pass
```

SyntaxError: non-default argument follows default argument



Schlüsselwörter

```
>>> def f(foo, bar, baz)
      return (foo, bar, baz)
```

Die Reihenfolge der Argumente ist oft schwer zu Merken
Die Lösung in Python: Schlüsselwörter

```
>>> f(3, baz=5, bar=4)
(3, 4, 5)
```




Variable Anzahl von Argumenten

```
>>> def sum2(a, b): return a + b
```

```
>>> def sum3(a, b, c): return a + b + c
```

Eleganter:



```
>>> def sum(*args):
    sum = 0
    for arg in args:
        sum += arg
    return sum
```

```
>>> sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
55
```



Variable Anzahl von Schlüsselwörtern

```
>>> def shopping_list(**kwargs):  
    for key, value in kwargs.items():  
        print(key + ": " + str(value))
```

```
>>> shopping_list(apples=3, bananas=5)  
apples: 3  
bananas: 5
```

kwargs ist ein normales Wörterbuch, hier z.B.
{'bananas' : 5, 'apples' : 3}



Zusammenfassung

Standardwerte für einzelne Argumente

```
>>> def f(x, y=2): pass
```

Argumente spezifizieren mit Schlüsselwörtern

```
>>> f(y=9, x=12)
```

Variable Anzahl von Argumenten

```
>>> def f(x, y, *rest): pass
```

Variable Anzahl von Schlüsselwörtern

```
>>> def f(x, y, **dictionary): pass
```



Argumente Entpacken

Umgekehrt können mit * und ** auch Funktionen aufgerufen werden

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

```
>>> sum(*l)
```

```
15
```

```
>>> d = {'bananas' : 5}
```

```
>>> shopping_list(**d)
```

```
bananas: 5
```



Lambda Funktionen

Wieso müssen Funktionen immer einen Namen haben?

```
>>> def square(x): return x*x
```

```
>>> square
```

```
<function square at 0x7fc0e1a037b8>
```

```
>>> lambda x: x*x
```

```
<function <lambda> at 0x7fc0e1a03730>
```

Wichtig für funktionale Programmierung!

```
>>> import functools
```



OOP - Objektorientiertes Programmieren



Alan Kay - Begründer von OOP

- 1973 Entwickler der GUI
- 1980 Smalltalk + OOP
- 2003 ACM Turing Award
- 2005 One Laptop per Child Projekt

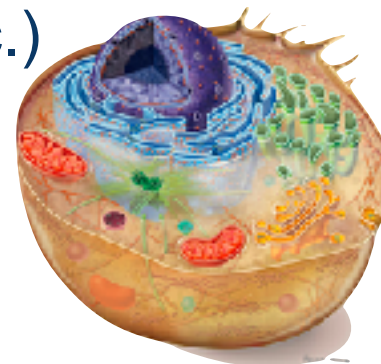
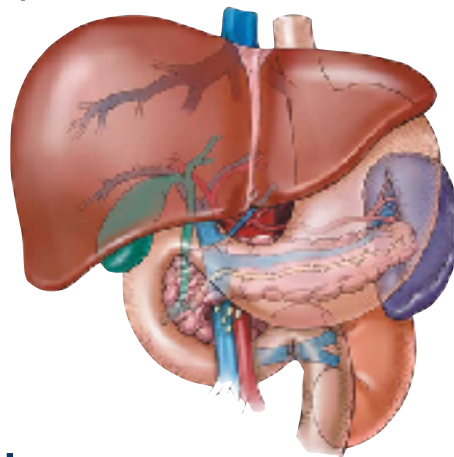
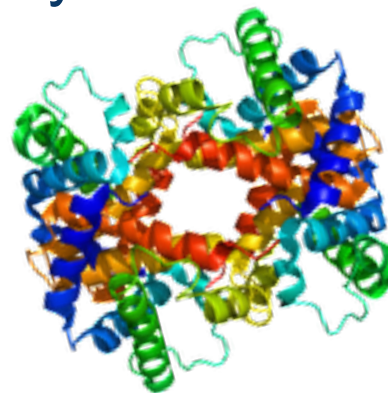
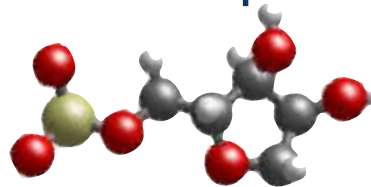


"The Computer Revolution Hasn't Happened Yet"

OOP und Biologie

Wie geht die Natur mit komplexen Systemen um?

- Moleküle
- Proteine
- Zellbausteine (Mitochondrien etc.)
- Zellen
- Organe
- Organismen...



OOP und Biologie (2)

Wie können wir von der Natur lernen

- (weitestgehend) unabhängige Bausteine
- Wohldefinierte Kommunikation
- Flexibilität
- Robustheit

In der Programmierpraxis

- Verlangt Intuition und Erfahrung
- Es gibt kein eindeutiges Richtig oder Falsch



Klassen und Instanzen

- Instanz = Ein Stück Speicher mit
 - Einer Referenz auf die Klasse
 - Dem Zustand der Instanz
- Klasse legt das Verhalten der Instanz fest

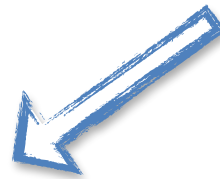
In der Biologie

- Klasse: Mitochondrium
- Instanz: Das Ding unter dem Mikroskop da



Klassen in Python

```
>>> class Cell:  
    position = [0.0, 0.0, 0.0]  
    radius = 1.0
```



```
    def grow(self):  
        self.radius += 0.1
```

Neue Konzepte:

- Attribute (position und radius)
- Methoden (grow)

Klassen verwenden

```
>>> class Cell:  
    position = [0.0, 0.0, 0.0]  
    radius = 1.0
```

```
    def grow(self):  
        self.radius += 0.1
```

```
>>> x = Cell()  
>>> x.grow()  
>>> x.radius  
1.1
```



Klassenmethoden

```
>>> class Cell:  
    def grow(self): pass
```

Wieso braucht es self?

- In Python gibt es eigentlich keine Klassenmethoden
- Klassenmethoden sind normale Funktionen, die von der Klasse verwaltet werden

x.m(a,b) -> getattr(x, 'm')(x, a, b)

cell.grow() -> getattr(cell, 'grow')(cell)



Klassen Initialisieren

Zellen sollten nicht alle bei [0.0, 0.0, 0.0] beginnen

```
>>> class Cell:
```

```
    position = [0.0, 0.0, 0.0]
```

```
    radius = 1.0
```

```
    def __init__(self, pos):
```

```
        self.position = pos
```

```
>>> x = Cell([1.0, 1.0, 4.0])
```

```
>>> x.position
```

```
[1.0, 1.0, 4.0]
```

Die **`__init__`** Funktion wird implizit aufgerufen



Vererbung

Im Prinzip „nur“ eine Kurzschreibweise

```
>>> class Human:  
    species = 'human'
```

```
>>> class StemCell(Cell):  
    def mitosis(self): ...
```

```
>>> class HumanStemCell(StemCell, Human):  
    pass
```

```
>>> x = HumanStemCell()  
>>> (x.radius, x.species, x.mitosis())  
(1.0, 'human', (<HumanStemCell> ...))
```



Methoden von Elternklassen verwenden

```
>>> class StemCell(Cell):  
    def mitosis(self): ...  
  
    def __init__(self, **kwargs):  
        super().grow()  
        super().__init__(**kwargs)  
  
>>> x = StemCell(pos=[1.0, 0.0, 2.0])  
>>> (x.position, x.radius)  
([1.0, 0.0, 2.0], 1.1)
```

Bei mehreren Eltern siehe MRO



Fehlerbehandlung



Implizite Fehlerbehandlung

Die gute Nachricht:

- Python ist dynamisch typisiert
- Ungültige Funktionsaufrufe werden automatisch signalisiert

```
>>> "foo" < 5
```

```
TypeError: unorderable types: str() < int()
```

```
>>> [].append(3,4)
```

```
TypeError: append() takes exactly one argument (2 given)
```

Python Programme können nicht abstürzen.TM



Fehlerbehandlung

Implizite Fehlerbehandlung hat aber auch Nachteile:

- Die Fehlermeldungen sind oft kryptisch
- Python ist sehr generisch, fehlerhafte Programme funktionieren bisweilen trotzdem

```
>>> table = []
```

```
>>> def add_names(list_of_names):  
    for name in list_of_names:  
        table.append(name)
```

```
>>> add_names("Lisa")
```

```
>>> table
```

```
['L', 'i', 's', 'a']
```



Explizite Fehlerbehandlung

```
>>> def add_names(names):
    if not isinstance(names, list):
        raise TypeError("Want List!")
    for name in list_of_names:
        table.append(name)
```

```
>>> add_names("Lisa")
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in add_names

TypeError: Want List!



Fehler in Python

```
>>> raise TypeError("Want List!")
```

- Werden mit **raise** signalisiert
- Sind normale Objekte
- Sind Unterklassen von **Exception**
- Sie können eigene Fehlerklassen definieren

```
>>> class AlgoKSError(Exception): pass
```

```
>>> raise AlgoKSError()
```



Fehler automatisch behandeln

- Nicht immer soll das Programm bei Fehlern abbrechen
- Manche Fehler können automatisch behoben werden

```
>>> try:  
    42 / 0  
except:  
    print("Oops!")  
Oops!
```



Fehler behandeln mit try und except

```
file = "foo.py"
```

```
try:
```

```
    f = open(file, 'r')
```

```
except IOError:
```

```
    print('cannot open', file)
```

```
else:
```

```
    process_file(f)
```

```
    f.close()
```

Für Details: docs.python.org/3.5/tutorial

Iteratoren und Generatoren



Wie funktionieren for-Schleifen?

```
>>> for element in [1, 2, 3]:  
    print(element)  
>>> for line in open("somefile.txt"):  
    print(line)
```

Was passiert beim iterieren über eine Datenstruktur?

- Die Schleife ruft **iter(X)** auf → Iterator Objekt
- Auf dem Iterator wird solange **next()** aufgerufen, bis eine **StopIteration** Ausnahme signalisiert wird



Iterator Beispiel

```
>>> l = [1, 2]
```

```
>>> it = iter(l)
```

```
>>> next(it)
```

1

```
>>> next(it)
```

2

```
>>> next(it)
```

StopIteration



Iteratoren überall

Nicht nur for-Schleifen verwenden Iteratoren, sondern auch viele Python-Bibliotheksfunktionen

```
>>> list((1,2,3))
```

```
[1, 2, 3]
```

```
>>> sum([1,2,3,4,5])
```

```
15
```

```
>>> dict = {1 : 'foo', 2 : 'bar'}
```

```
>>> tuple(dict.keys())
```

```
(1, 2)
```



Die Funktion zip

- Nimmt eine beliebige Anzahl von Iteratoren
- Erzeugt einen Iterator über Tupel daraus
- Nützlich um Daten zu kombinieren

```
>>> list(zip([1,2,3], 'abc'))
```

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

```
>>> for (a, b) in zip([1,2,3], 'abc'):  
    print(str(a) + b)
```

```
1a
```

```
2b
```

```
3c
```



Magische Funktionen

Wie funktionieren dir, str, +, -, iter, next?

next(x) -> x.__next__()

dir(x) -> x.__dir__()

str(x) -> x.__str__()

float(x) -> x.__float__()

x() -> x.__call__()

x + y -> x.__add__(y)

x and y -> x.__and__(y)



Eigenen Iterator definieren

```
>>> from random import randint
```

```
>>> class random:
```

```
    def __iter__(self): return self
```

```
    def __next__(self):
```

```
        r = randint(-1,9)
```

```
        if r == -1: raise StopIteration
```

```
        else: return r
```

```
>>> list(Random())
```

```
[9, 5, 8, 1, 2, 7]
```

```
>>> list(Random())
```

```
[7, 5, 0, 8]
```



Generatoren

- Eigene Iteratoren zu definieren ist umständlich
- Die Lösung: Generatoren

```
>>> def my_generator():  
    yield 'a'  
    yield 'b'  
    yield 'c'
```

```
>>> list(my_generator())  
['a', 'b', 'c']
```

Yield verwandelt eine Funktion in einen Generator



Generatoren (2)

- Statt normaler Rückgabewerte liefert die Funktion einen Iterator
- Beim Iterieren wird jeweils bis zum nächsten **yield** Ausdruck gerechnet
- Wenn das Ende der Funktion erreicht ist, wird automatisch StopIteration signalisiert

```
>>> def myiter():
    for i in [1,2,3]: yield i
>>> myiter()
<generator object myiter at 0x102281db0>
>>> next(myiter())
1
```



NumPy - Numerik mit Python



NumPy Arrays

- Ähnlich zu Python Listen
- Aber: Alle Elemente haben den gleichen Typ
- Beliebig viele Dimensionen
- Sehr effizient

```
>>> np.array([[1,2,3],[4,5,6]])  
array([[1, 2, 3],  
       [4, 5, 6]])
```



Indexoperationen

```
>>> a = np.array([[1,2,3],[4,5,6]])
```

```
>>> a[0]
```

```
[1, 2, 3]
```

```
>>> a[0,0]
```

```
1
```

```
>>> a[0:2,0:2]
```

```
array([[1, 2],  
       [4, 5]])
```

```
>>> a[:,0:3:2]
```

```
array([[1, 3],  
       [4, 6]])
```

NumPy Arrays als Objekte

- NumPy Objekte sind Instanzen der Klasse `numpy.ndarray`
- Wichtige Attribute sind:

```
>>> a.shape
```

```
(2, 3)
```

```
>>> a.dtype
```

```
'int64'
```

Gültige Elementtypen sind z.B.

`int8`, `uint16`, `float32`, `float64`, `complex128`

```
>>> np.array([1,2], float)
```

```
>>> _.dtype
```

```
dtype('float64')
```



Nützliche Funktionen auf Arrays

- **transpose**
- **flatten**
- **concatenate**
- **fill**
- **eye**
- **identity**
- **arange**



Mathematische Operationen in NumPy

- Werden Elementweise angewendet
- Deutlich schneller als normale Python Operationen

```
>>> a = np.array([[1,2,3],[4,5,6]])
```

```
>>> a * a
```

```
array([[ 1,  4,  9],  
       [16, 25, 36]])
```

```
>>> -a
```

```
array([[ -1, -2, -3],  
       [-4, -5, -6]])
```



Automatisches Replizieren

- Arrays müssen nicht gleich groß sein
- Pro Dimension darf nur ein Operand mehrere Werte haben

```
>>> a = np.array([[1,2,3],[4,5,6]])
```

```
>>> a * 2
```

```
array([[ 2,  4,  6],  
       [ 8, 10, 12]])
```

Die 2 wird hier auf ein Array der Form (2, 3) erweitert



SciPy

NumPy liefert Grundfunktionalität, SciPy den ganzen Rest

SciPy Module

- Numerische Integration (`scipy.integrate`)
- Optimierung (`scipy.optimize`)
- Interpolation (`scipy.interpolate`)
- Fast-Fourier Transformationen (`scipy.fftpack`)
- Lineare Algebra (`scipy.linalg`)

Bei Numerik-Problemen immer zuerst SciPy konsultieren!



Das Zen von Python



>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

...



Jan's inoffizielle Tipps für und nicht nur für AlgoKS

- Python spielerisch lernen: codinggame.com
- Python visuell verstehen: pythontutor.com
- Gute Python Talks: Raymond Hettinger
- rosettacode.org
- 3Blue1Brown: Essence of linear algebra



Literatur

Python

docs.python.org/3.5

wiki.python.org/moin

NumPy und SciPy

docs.scipy.org/doc

Matplotlib

matplotlib.org

