

# Algorithmik kontinuierlicher Systeme

Einführung in Python



# Organisatorisches

- Bitte im EST **sowohl** für AlgoKS Theorieübung, **als auch** für AlgoKS Programmierübung anmelden!
- Die erste Programmierübung beginnt Montag 16.04. um 10:00 Uhr
- Programmieraufgaben sind Einzelabgaben. Wir prüfen auf Plagiate (auch gegenüber dem letzten Semester).

# Videoaufzeichnung

Die Vorlesung wurde letztes Jahr aufgezeichnet.

Die Aufzeichnungen findet man unter  
[video.cs.fau.de/by-lecture/AlgoKS/2017s](https://video.cs.fau.de/by-lecture/AlgoKS/2017s)

Nutzername: AlgoKS

Passwort: ALGOKSFAU

**Wichtig:** Der Stoff ändert sich jedes Jahr geringfügig.

# Python

- Eine freie Skriptsprache
- BDFL: Guido van Rossum
- Tausende Bibliotheken, z.B. NumPy für Numerik
- Das bessere Matlab
- Webseite und Dokumentation:  
[www.python.org](http://www.python.org)



# Wieso Python?

- Eine der 10 populärsten Programmiersprachen der Welt.
- Python ist einfach zu verwenden und zu lesen.
- Python ist freie Software.
- Es gibt über 15000 Python-Pakete.
- Moderne Forschungsergebnisse werden oft in interaktivem Python 'publiziert'.

# Python ist super!

## Wieso verwenden nicht alle Python?

- Python Programme sind nicht schnell\*.
- Python Programme sind tendenziell etwas fehleranfällig.  
Bitte keine Atomkraftwerke mit Python programmieren.

\*Man kann allerdings Python gut mit C/C++ koppeln.

# Python auf einen Blick

- Einrückung ist Pflicht
- Automatische Speicherverwaltung
- Alles ist ein Objekt
- Ententypisierung
- Gutes Modulsystem



# Python verwenden

Unter Linux:

```
$ apt-get install python3-scipy python3-matplotlib
```

```
$ python3      # WICHTIG Python 3
```

```
>>> 5+5
```

```
10
```

```
$ python3 mein_programm.py
```

**Auf anderen Systemen (Windows, MacOS)**

Anaconda installieren: [www.continuum.io/downloads](http://www.continuum.io/downloads)

Den Instruktionen auf der Website folgen





# Python Entwicklungsumgebungen

## Im CIP installiert:

- PyCharm [www.jetbrains.com/pycharm](http://www.jetbrains.com/pycharm)  
**addpackage pycharm**  
**pycharm**
- Vim [www.vim.org](http://www.vim.org)
- Emacs [www.gnu.org/software/emacs](http://www.gnu.org/software/emacs)

## Oder ein Editor aus der folgenden Übersicht:

[wiki.python.org/moin/PythonEditors](http://wiki.python.org/moin/PythonEditors)



# 1. Python Grundlagen



# Zahlen

- Die Operatoren +, -, \*, % und \*\* verhalten sich „wie gewohnt“

```
>>> 2 + 2
```

```
4
```

```
>>> ((2 - 3) * 5) ** 2
```

```
25
```

- Der / Operator verhält sich anders als in C und Java

```
>>> -9 / 5
```

```
-1.8
```

- Der // Operator rundet zur nächst niedrigeren Zahl

```
>>> -9 // 5
```

```
-2
```



# Zuweisungen

- Zugewiesen wird mit =

```
>>> a = 5
```

```
>>> a
```

```
5
```

- Variablen werden automatisch bei Zuweisung angelegt

```
>>> foo
```

```
NameError: name 'foo' is not defined
```

```
>>> foo = 42
```

```
>>> foo
```

```
42
```



# Zuweisungen

Mehrfachzuweisungen sind möglich

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

```
>>> b
```

6

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

```
>>> b
```

5

```
>>> a
```

6



# Zeichenketten

```
>>> 'Zeichenkette'  
'Zeichenkette'
```

```
>>> "foo" == ""foo"" == 'foo' == "'foo'"  
True
```

Sonderzeichen wie gewohnt:

```
>>> print("foo\nbar")  
foo  
bar
```



## Zeichenketten (2)

Funktionen in Python sind überraschend generisch

```
>>> "Na " * 8 + "Batman!"
```

```
'Na Na Na Na Na Na Na Na Batman!'
```

Zeichenketten können über mehrere Zeilen gehen

```
"""foo
```

```
bar
```

```
baz
```

```
""" # ... aber nur mit ' oder """
```



# Boolsche Ausdrücke

```
>>> 5 == 3 + 2
```

True

```
>>> 2 <= 5 <= 7
```

True

Boolsche Funktionen werden ausgeschrieben (and, not, or)

```
>>> x = 2
```

```
>>> x < 3 and not x > 7
```

True





# Funktionen



```
>>> def praise(name):  
    print(name + " ist super!\n")
```

```
>>> praise("Python")  
Python ist super!
```

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

```
>>> square(5)  
25
```



# Funktionen

```
>>> def factorial(n):  
    if n == 0:  
        return 1  
    elif n > 0:  
        return n * factorial(n-1)
```

```
>>> factorial(5)  
120  
>>> factorial(-2)  
>>>
```

Alle Funktionen geben implizit **None** zurück



# Dokumentieren von Funktionen

- Zeichenketten am Anfang der Funktion werden als Dokumentation erkannt
- **Gute** Dokumentation ist sehr wertvoll

```
>>> def praise(name):  
    """Praise the Entity specified by the  
    string ``name`` by printing a  
    statement to the standard output.  
    """"  
  
    print(name + " ist super!\n")
```



# Einrückung

Vorteil: Code braucht (normalerweise) keine Semikolons

Nachteil: Einrückung ist wichtig!

```
def foo(x):  
    a = 5  
    def square(n):  
        return n * n  
    b = square(x)  
    return b - a
```



# Einrückung (2)

Goldene Regel: Nach Doppelpunkt eine Ebene tiefer

```
def positive(x):  
    if x > 0:  
        return True  
    else:  
        return False  
return "w00t?!" # wird nie erreicht
```



# Datenstrukturen



# Listen und Tupel

```
>>> [3, 2, 1, 'foo'] # eine Liste
```

```
[3, 2, 1, 'foo']
```

```
>>> (3, 2, 1, 'foo') # ein Tupel
```

```
(3, 2, 1, 'foo')
```

```
>>> (5)          # KEIN Tupel
```

```
5
```

```
>>> (5,)         # ein Tupel
```

```
(5,)
```

- Listenelemente können modifiziert werden
- Tupel sind immer konstant



## Listen und Tupel (2)

```
>>> a = ('foo', 'bar')
```

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

```
'foo'
```

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

```
'f'
```

```
>>> a[0] = 'baz'
```

```
TypeError: 'tuple' object not mutable
```

```
>>> b = list(a)    # opposite: tuple(a)
```

```
>>> b[0] = 'baz'
```

```
>>> b
```

```
['baz', 'bar']
```





# Elemente Auswählen

- Nummerierung beginnt immer bei 0

```
>>> 'abc'[0]
```

```
'a'
```

```
>>> 'abc'[2]
```

```
'c'
```

- Negative Indizes beginnen beim letzten Element

```
>>> 'abc'[-1]
```

```
'c'
```

```
>>> 'abc'[-3]
```

```
'a'
```



# Teilmengen auswählen

- Die Syntax ist START : ENDE
- START ist eingeschlossen, ENDE nicht
- START und ENDE können auch weggelassen werden,

```
>>> [0,1,2,3,4][1:3]
```

```
[1, 2]
```

```
>>> [0,1,2,3,4][1:]
```

```
[1, 2, 3, 4]
```

```
>>> 'last six digits'[-6:]
```

```
'digits'
```

```
>>> [][:]
```

```
[]
```



## Teilmengen auswählen (2)

Zusätzlich kann die Schrittweite angegeben werden

```
>>> a = list(range(9))
```

```
>>> a
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> a[1:7:2]
```

```
[1, 3, 5]
```

Natürlich auch negativ!

```
>>> a[::-1]
```

```
[8, 7, 6, 5, 4, 3, 2, 1, 0]
```



# Datenstrukturen vereinigen

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

```
>>> a + [3,4]
```

```
[1, 2, 3, 4]
```

```
>>> a
```

```
[1, 2]
```

```
>>> a.extend([3,4])
```

```
[1, 2, 3, 4]
```

```
>>> a
```

```
[1, 2, 3, 4]
```



# Wörterbücher (dictionaries)

Verwaltung von (Schlüssel : Wert) Paaren

```
>>> {'foo' : 2+2, 'bar' : 99}
```

```
{'foo' : 4, 'bar' : 99}
```

Generische Schlüssel

```
>>> dict = {'a' : 5, (5,7) : 9, 'w00t' : '?'}
```

```
>>> dict['a']
```

```
5
```

```
>>> dict[(5,7)]
```

```
9
```



# Wörterbücher (2)

Einträge löschen

```
>>> dict = {1:'Hallo', 2:'Welt', 3:'!'}
```

```
>>> del dict[2]
```

```
>>> dict
```

```
{1:'Hallo', 3:'!'}
```

```
>>> dict.clear()
```

```
>>> dict
```

```
{}
```

Anmerkung: Wörterbücher sind als Hashtabellen implementiert, d.h. der Zugriff ist  $O(1)$



# Der 'in' Operator

Testen auf Zugehörigkeit

```
>>> 5 in [1,2,3,4,5,6,7]
```

**True**

```
>>> vowels = 'aeiou'
```

```
>>> 'f' not in vowels
```

**True**



# Zusammenfassung Datenstrukturen

## Die wichtigsten Datenstrukturen:

- Listen `[]`
- Tupel `()`
- Zeichenketten `" "`
- Wörterbücher `{ }`

## Nächste Woche noch

- Klassen
- Arrays





# Kontrollfluss



# Bedingungen

```
>>> if weight(person) == weight(duck):  
    person.name + "is a witch!"  
else:  
    "She is innocent!"
```

```
>>> if x == 0:  
    "zero"  
elif x < 0:  
    "negative"  
elif x > 0:  
    "positive"
```



# Schleifen

```
>>> while not lecture_end():  
    slide = next_slide()  
    talk_about(slide)
```

```
>>> for i in [1, 2, 3]:  
    print(i)
```

1

2

3



# for-Schleifen

Das elementare Werkzeug des Python-Programmierers

```
>>> a = []  
>>> for x in range(5):  
    a.append(x*x)  
>>> a  
[0, 1, 4, 9, 16]
```

**Wichtig:** Eine **range** geht von 0 bis  $n - 1$



## for-Schleifen (2)

For-Schleifen können über fast alles iterieren

```
>>> for c in 'abc': ...
```

```
>>> for n in range(5): ...
```

```
>>> for elt in (1, 'foo', 1.5, []): ...
```

```
>>> for (a, b) in [(1,8), (2,7)]: ...
```

**break** und **continue** wie in C und Java

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

```
    print(i)
```

```
    if i == 2: break
```

0

1

2



# Nichts tun

```
>>> pass
```

```
>>>
```

- Macht die Absicht klarer
- Bisweilen sogar unerlässlich

```
>>> if all_good():  
    pass  
    else:  
        try_to_fix_it()
```



# Zusammenfassung

- Python hat sehr mächtige for-Schleifen
- if, break, continue und while fast wie in C und Java
- Einrückungen nicht vergessen!

Etablierte Regel:

4 Leerzeichen pro Stufe Einrückung (nie Tabs)

Weitere Stilregeln zur Einrückung etc in [PEP 8](#)



# Das Modulsystem von Python





# Module in Python

## Aufgaben eines Modulsystems:

- Namenskollisionen vermeiden
- Projekte strukturieren
- Fremde Bibliotheken verwenden

## Nutzung in Python:

```
>>> import foo
```

```
ImportError: No module named foo
```

```
>>> import sys
```

```
>>>
```



# Was passiert bei 'import x'

```
>>> import sys
```

1. Das momentane Verzeichnis wird nach der Datei **sys.py** durchsucht, ansonsten eine folge von Standard-Verzeichnissen
2. Der Inhalt der Datei wird Ausdruck für Ausdruck in einen neuen Namensraum geladen
3. Ein Modul mit allen Symbolen des neuen Namensraums wird erstellt
4. Die Variable **sys** wird auf dieses Modul gesetzt

```
>>> sys
```

```
<module 'sys' (built-in)>
```



# Importierte Module verwenden

```
>>> import sys
```

```
>>> sys.version
```

```
'3.5.3 (default, Apr 21 2017, 11:52:50) \n[GCC 4.2.1  
Compatible Apple LLVM 8.1.0 (clang-802.0.42)]'
```

```
>>> import math
```

```
>>> math.sqrt(5)
```

```
2.23606797749979
```



# Arten des Importierens

- import ...  
**import sys**
- import ... as ...  
**import numpy as np**
- from ... import ...  
**from math import sqrt, pow**
- from ... import \*  
**from math import \***



# Ein eigenes Python Modul erstellen

- Schreibt euren Code in eine Datei XYZ.py
- Legt die Datei in den Python Suchpfad (z.B. der Ordner in dem ihr arbeitet)
- Fertig ist das Modul XYZ!



# Ausführbare Dateien "Wie in C"

In der ersten Zeile:

```
#!/usr/bin/env python3
```

Gefolgt von

```
import foo
```

```
from bar import f1, f2
```

Irgendwo am Ende:

```
if __name__ == "__main__":
```

```
    my_code()
```

```
    more_of_my_code()
```



# Abschluss: Der Übungsbetrieb



# Wie sieht eine Programmierübung aus?

Alles Material finden Sie unter

[www10.cs.fau.de/lehre/byName/algoks](http://www10.cs.fau.de/lehre/byName/algoks)

Jede Programmierübung besteht aus

- Einem Aufgabenblatt (als PDF)
- Mehreren Dateien zu den Aufgaben auf dem Blatt
- Zugehörige Testsuiten **\*\_test.py**





# Selber testen mit test.py

## Vorteile:

- Punkte idR bereits vor Abgabe bekannt
- Weniger Zeit bei der Fehlersuche
- Lernen wie toll Tests sind

## Nachteile:

- Für jede Teilaufgabe gibt es alle oder keine Punkte
- Testsuite kann unterwandert werden\*

\* Wir haben zahlreiche versteckte Tests



# Live Demo!

