

Vorlesung 5: Wichtige Datentypen

Till Bargheer, Hendrik Weimer

Institut für Theoretische Physik, Leibniz Universität Hannover

Programmieren für Physikerinnen und Physiker, 11.11.2019

Wiederholung Vorlesung 4

- ▶ Module importieren und benutzen
- ▶ Namensräume
- ▶ Externe Module installieren
- ▶ NumPy und SciPy

1. Einführung
2. Erste Schritte in Python
3. Den Ablauf eines Programms steuern
4. Zusatzfunktionen durch Python-Module
5. **Wichtige Datentypen**
6. Daten eingeben und ausgeben
7. Datenvisualisierung mit Matplotlib
8. SciPy: Module für die Wissenschaft
9. Analytisches Rechnen mit SymPy
10. Objektorientiertes Programmieren
11. Versionskontrolle
12. Bonusthema

Bereits bekannte Datentypen

- ▶ Ganzzahlen

```
>>> print(type(1))  
<class 'int'>
```

- ▶ Fließkommazahlen

```
>>> print(type(0.1))  
<class 'float'>
```

- ▶ Strings

```
>>> print(type("foo"))  
<class 'str'>
```

- ▶ Wahrheitswerte

```
>>> print(type(True))  
<class 'bool'>
```

Klassen können Funktionen haben!

```
>>> str = "foomatic"  
>>> str.find("bar")  
-1  
>>> str.find("oma")  
2  
>>> "foobاربaz".strip("baz")  
'foobar'
```

Listen von Variablen

- ▶ Definition: Durch Kommas getrennte Folge, eingeschlossen in eckigen Klammern

```
>>> a = [1, 1, 2, 3, 5, 8]
```

- ▶ Einzelne Elemente mit eckigen Klammern (Start bei 0)

```
>>> a[2]  
2
```

- ▶ Mehrere Elemente durch Doppelpunkt für Anfang und Ende

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

- ▶ Schnelle Initialisierung: Multiplikations-Operator

```
>>> [0]*4  
[0, 0, 0, 0]
```

- ▶ Elemente hinzufügen und entfernen:

```
>>> a = [1, 1, 2, 3, 5, 8]
>>> a.append(13)
>>> a
[1, 1, 2, 3, 5, 8, 13]
>>> a.pop()
13
>>> a
[1, 1, 2, 3, 5, 8]
>> a.append([13, 21])
>> a
[1, 1, 2, 3, 5, 8, [13, 21]]
>> a.pop()
[13, 21]
>> a.extend([13, 21])
>> a
[1, 1, 2, 3, 5, 8, 13, 21]
```

Noch mehr zu Listen

- ▶ Listen kombinieren:

```
>>> [1, 1, 2, 3] + [5, 8]
[1, 1, 2, 3, 5, 8]
```

- ▶ Listen in for-Schleifen:

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

- ▶ Vorsicht beim Kopieren von Listen!

```
>>> a = [1, 1, 2]
>>> b = a
>>> a.append(3)
>>> b
[1, 1, 2, 3]
```

- ▶ Lösung: Verwende `copy()`-Funktion

```
>>> a = [1, 1, 2]
>>> b = a.copy()
>>> a.append(3)
>>> b
[1, 1, 2]
```


- ▶ Listen als Argument

```
def foo(bar):  
    print(bar)
```

```
foo([1, 2, 3])
```

- ▶ Listen als Rückgabewert

```
def foo():  
    return [1, 2]
```

```
x, y = foo()
```

- ▶ Wie Listen, mit ein paar kleinen Abweichungen
- ▶ Definition: Mit Kommas getrennt, mit oder ohne runde Klammern

```
>>> 1, 2, 3
(1, 2, 3)
```
- ▶ Tuple mit einem Element: abschließendes Komma
- ▶ Tuples können nicht nachträglich nicht verändert werden
- ▶ Es gibt auch “named tuples” (vgl. `time.localtime()`)

- ▶ Datentyp für Sets (Mengen) in geschweiften Klammern:

```
>>> {1, 2, 3, 1}
{1, 2, 3}
```

- ▶ Operationen mit Sets

```
>>> 1 in {1, 2, 3}
True
>>> {1, 2, 3} - {1, 3}
{2}
>>> {1, 2, 3} | {2.5}
{1, 2, 3, 2.5}
>>> {1, 2, 3} & {1, 3}
{1, 3}
```

- ▶ Ähnlich wie Listen, nur mit Keys (Schlüsseln) als Index:

```
>>> noten = {"Peter": 1.3, "Paul": 1.7, "Maria": 1.0}  
>>> noten['Maria']  
1.0
```

- ▶ Neue Einträge:

```
noten['Bernd'] = 1.3
```

- ▶ Liste aller Keys:

```
>>> list(noten.keys())  
['Maria', 'Paul', 'Bernd', 'Peter']
```

NumPy-Arrays

- ▶ Listen sind keine Vektoren:

```
>>> [0, 1] + [1, 0]  
[0, 1, 1, 0]
```

- ▶ NumPy-Arrays verhalten sich wie gewünscht:

```
>>> import numpy as np  
>>> np.array([0, 1]) + np.array([1, 0])  
array([1, 1])
```

Matrizen als 2D-Arrays

- ▶ Matrizen sind zweidimensionale NumPy-Arrays

```
>>> A = np.array([[1, 2, 3],  
...               [4, 5, 6],  
...               [7, 8, 9]])  
>>> A[1,0]  
4  
>>> A[:,0]  
array([1, 4, 7])  
>>> A.trace()  
15
```

Matrix-Multiplikation

- ▶ $A \cdot A$ ist elementweise Multiplikation!

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

```
>>> A*A  
array([[ 1,  4,  9],  
       [16, 25, 36],  
       [49, 64, 81]])
```

- ▶ Echte Matrix-Multiplikation mit `matmul`:

```
>>> np.matmul(A, A)  
array([[ 30,  36,  42],  
       [ 66,  81,  96],  
       [102, 126, 150]])
```

- ▶ Ab Python 3.6: $A @ A$ funktioniert ebenfalls

Beispiel: Lineares Gleichungssystem lösen

- Lösung von $Ax = b$

$$A = \begin{pmatrix} 2 & 1 & 1 & -1 \\ 1 & -2 & 1 & -1 \\ 1 & 1 & -2 & -1 \\ -1 & -1 & -1 & 2 \end{pmatrix} \quad b = \sqrt{2} \begin{pmatrix} 3/2 \\ 1 \\ 1 \\ -3/2 \end{pmatrix}$$

- Anwendung eines Sprung-Operators A auf einen Quantenzustand x (z.B. Fehlerquelle in einem Quantencomputer)

Diskretes Gitter mit NumPy-Arrays

- ▶ `arange` ist eine Erweiterung von `range` für Fließkommazahlen

```
>>> np.arange(0.1,0.4,0.1)
array([ 0.1,  0.2,  0.3,  0.4])
```

- ▶ Endpunkt hängt von Fließkomma-Rundung ab:

```
>>> np.arange(0.1,0.3,0.1)
array([ 0.1,  0.2])
```

- ▶ Bessere Kontrolle mit `linspace`

```
>>> np.linspace(0.1,0.4,3,endpoint=False)
array([ 0.1,  0.2,  0.3])
```