vorlesung_numpy

November 23, 2022

1 Numpy

Numpy ist ein grundlegendes Python-Modul für wissenschaftliche Berechnungen das sehr verbreitet eingesetzt wird. Viele weitere Module verwenden dieses Modul auch.

Wie immer binden wir das Modul mit `import` ein. Als Abkürzung hat sich np etabliert.

[1]:
```
import numpy as np
```

Jedes Modul definiert eine Variables `__version__` in der die Versionsnummer des Moduls steht. Dies ist, bei sich schnell entwickelnden Modulen, hilfreich um die passende Dokumentation zu lesen.

[2]:
```
np.__version__
```

[2]:
```
'1.21.0'
```

1.1 Numpy-Arrays

Die wichtigste Neuerung bei `Numpy` ist ein neuer Datentyp, das Numpy-Array (ndarray). Über die Funktion `np.array()` lassen sich Python-Listen in Numpy-Arrays umwandeln.

[3]:
```
a = np.array([1,2,3,4])
a
```

[3]:
```
array([1, 2, 3, 4])
```

[4]:
```
type(a)
```

[4]:
```
numpy.ndarray
```

Im Gegensatz zu Python-Listen, enthalten Numpy-Arrays nur Elemente **eines** Datentyps. Die anderen Elemente werden zur Not enstrechend umgewandelt.

[5]:
```
np.array([1,2,3.])  # Alle Elemente werden zu Fließkommazahlen umgewandelt.
```

[5]:
```
array([1., 2., 3.])
```

[6]:
```
np.array([1,2,3.,'c']) # Alle Elemente werden zu Strings umgewandelt.
```

[6]:
```
array(['1', '2', '3.0', 'c'], dtype='<U32')
```

Analog zu der Python-Funktion `range()` gibt es in Numpy eine Funktion `arange()` um direkt einen Numpy-Array zu generieren.

[7]:
```
b = np.arange(2,10,2)
b
```

[7]:
```
array([2, 4, 6, 8])
```

Auch bei Numpy-Arrays kann man auf einzelne Elemente über den Index zugreifen. **Nicht vergessen:** Python zählt ab 0!

[8]:
```
b[2]
```

[8]:
```
6
```

1.2 Teilbereiche von Numpy-Arrays ("slices")

Man kann auch auf Teile des Arrays zugreifen. Dabei gibt man als Index Bereiche an. *name*[*start*:*stop*], auch hier ist *stop* wieder nicht Teil der Liste!

[9]:
```
a
```

[9]:
```
array([1, 2, 3, 4])
```

[10]:
```
c = a[0:3]
c
```

[10]:
```
array([1, 2, 3])
```

Läßt man den Startwert weg, wird beim 0ten Element begonnen.

[11]:
```
a[:3]
```

[11]:
```
array([1, 2, 3])
```

Läßt man den Stopwert weg, geht der Bereich bis zum Ende (inkl. letzem Element).

[12]:
```
d = a[1:]
d
```

[12]:
```
array([2, 3, 4])
```

Ist der obere Index negativ, so wird von hinten gezählt. So bedeutet `-1` z.b. bis zum vorletzten Element (inklusiv).

[13]:
```
a[:-1]
```

[13]:
```
array([1, 2, 3])
```

1.3 Mathematische Operatoren

Der große Vorteil von Numpy-Arrays ist ihr Verhalten unter numerischen Operationen. Sie verhalten sich hier ähnlich wie Vektoren, die Operation wird auf alle Elemente parallel angewendet. Im Gegensatz zu den meisten Programmiersprachen, spart man dadurch viele Schleifen und erhält sehr übersichtlichen Programmtext.

[14]:
```
a
```

[14]:
```
array([1, 2, 3, 4])
```

[15]:
```
a + 1
```

[15]:
```
array([2, 3, 4, 5])
```

[16]:
```
print(a)
print(b)
a + b
```

```
[1 2 3 4]
[2 4 6 8]
```

[16]:
```
array([ 3,  6,  9, 12])
```

[17]:
```
2*a
```

[17]:
```
array([2, 4, 6, 8])
```

Allerdings gilt dies auch für die Multiplikation zweier Arrays,],die auch elementweise durchgeführt wird. Hier hinkt der Vektorvergleich schon etwas.

[18]:
```
print(a)
print(b)
a * b
```

```
[1 2 3 4]
[2 4 6 8]
```

[18]:
```
array([ 2,  8, 18, 32])
```

Will man statt elementweiser Multiplikation, das Skalarprodukt der Vektoren, so muss man die Funktion `np.dot()` verwenden.

[19]:
```
np.dot(a, b)
```

[19]:
```
60
```

Analog gibt es mit `np.cross()` auch das Kreuzprodukt, allerdings ist dies nur in 2- und 3-Dimensionen definiert.

[20]:
```
np.cross(a, b)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_2894/3422562784.py in <cell line: 1>()
----> 1 np.cross(a, b)

<__array_function__ internals> in cross(*args, **kwargs)

/opt/conda/lib/python3.9/site-packages/numpy/core/numeric.py in cross(a, b,␣
 ↪axisa, axisb, axisc, axis)
   1604             "(dimension must be 2 or 3)")
   1605     if a.shape[-1] not in (2, 3) or b.shape[-1] not in (2, 3):
-> 1606         raise ValueError(msg)
   1607
   1608     # Create the output array

ValueError: incompatible dimensions for cross product
(dimension must be 2 or 3)
```

[ ]:
```
print(f'c={c} und d={d}')
print()
print("c x d =", np.cross(c, d))
```

Potenzieren funktioniert elementweise, wieder wie erwartet

[ ]:
```
a**3
```

[ ]:
```
a * a * a
```

1.4 Weitere wichtige Funktionen:

Numpy implementiert auch eine Reihe weitere Funktionen, die einem die Programmierung leichter machen (und Schleifen ersparen).

So erhält man z.B. die Summe aller Elemente eine Numpy-Arrays mit der Funktion `np.sum()`,

[ ]:
```
print(a)
print(np.sum(a))
```

und den Mittelwert mit `np.mean()`:

[ ]:
```
np.mean(a)
```

Der Mittelwert ist natürlich die Summe der Elemente, geteilt durch die Anzahl der Elemente. Letztere erhält man über die Python-Funktion len().

[ ]:
```
np.sum(a) / len(a)
```

[ ]:
```
len(a)
```

1.5 Numpy-Arrays initialisieren

Neben `np.array()` und `np.arrange()` gibt es weitere Funktionen um Numpy-Arrays zu konstruieren: Will man einen Array bestimmter Größe bei dem alle Elemente den Wert `0` haben, kommt `np.zeros()` zum Einsatz. Hier gebe ich nur an, wieviele Elemente der Array haben soll.

[ ]:
```
np.zeros(10)
```

Analog kann ich auch einen Array nur mit dem Wert `1` über `np.ones()` erhalten.

[ ]:
```
np.ones(10)
```

Für andere Zahlenwerte gibt es keine entsprechende Funktionen…

[ ]:
```
np.twos(10)
```

… diese erhalte ich aber einfach über Multiplikation mit der entsprechenden Konstante.

[ ]:
```
2 * np.ones(10)
```

1.5.1 linspace()

Mit `np.linspace(*start*,*stop*,*num*)` gibt es eine weitere Funktion um Werte in einem Intervall zu erzeugen. Allerdings gibt man hier nicht wie bei `np.arange()` die Schrittweite, sondern die Anzahl (*num*) der Elemente an. **Wichtig**:`np.linspace()` ist eine der wenigen Funktionen, bei der die **obere Grenze Teil der Liste** ist!

[ ]:
```
x = np.linspace(0, np.pi, 7)
print(x)
```

1.6 Mathematische Funktionen

Alle Funktionen, die es im Modul `math` gibt, findet man auch in `numpy`. Praktische ist hier, dass auch diese dann elementweise berechnet werden:

[ ]:
```
np.cos(x)
```

Wieder wird alles parallel berechnet und man spart die Programmierung von Schleifen. Besonders häufig verwendet man dies beim grafischen Darstellen von Funktionen, wie im nächsten Teil *matplotlib* gezeigt wird.