NumPy Funktionen

by Woche 3

Da wir nun wissen wie wir Numpy-Arrays erstellen können, wollen wir uns nun ansehen, was wir mit diesen Arrays alles machen können.

```
import numpy as np
```

Slicen & Filtern

Wie Liste, können Arrays auch gesliced werden, indem die eckigen Klammern und Doppelpunkte genutzt werden (siehe Kapitel 2.4.1):

```
x = np.array([50, 60, 70])
x[1:]
```

```
array([60, 70])
```

```
x = np.arange(10, 20)
x[-6:-1:2]
```

```
array([14, 16, 18])
```

Es ist auch möglich, Arrays zu filtern. Im Prinzip muss also für jedes Element geprüft werden ob eine bestimmte Filter-Bedingung erfüllt ist. Umgesetzt wird das in NumPy mit sogenannten *Boolean Arrays*, also Arrays, die nur `True` und `False` Werte enthalten. Hier erstmal ein Beispiel:

```
x = np.arange(1, 6)
y = x>2

z = x[y]

print(x)
print(y)
print(z)
```

```
[1 2 3 4 5]
[False False  True  True  True]
[3 4 5]
```

Das Array `y` ist hier das *Boolean Array*, das angibt, ob die Werte im Array `x` größer als 2 sind. Damit das Filtern mit `x` und `y` funktioniert, müssen beide unbedingt gleich lang sein - das ist hier aber automatisch gegeben, da `y = x>2` genutzt wird um `y` zu erzeugen. Wie schon bei den Rechenoperationen oben wird die Bedingung `>2` einfach auf das Array angewendet (`x`) und das Ergebnis ist dann ein Array mit `True` und `False` Werten. Da es sich in `x` um die Zahlen 1-5 handelt, sind demnach die ersten zwei Elemente in `y` `False` und die letzten drei `True`.

Nun kann mittles der Index-Schreibweise, also den eckigen Klammern, das Array `x` gefiltert werden. Alle Werte in `x` an deren entsprechender Stelle in `y` ein `True` steht, werden in das neue Array `z` übernommen. Das Ergebnis ist also das Array `z`, das nur die Werte 3, 4 und 5 enthält.

Tatsächlich kann die Bedingung auch direkt in den eckigen Klammern angegeben werden. Das spart eine Zeile und ist auch intuitiver, da die Bedingung direkt dort steht, wo das Array gefiltert wird. Es passiert aber im Endeffekt dasselbe, nur dass das *Boolean Array* nicht als Objekt zwischengespeichert wird.

```
x = np.arange(1, 6)

z = x[x>2]

print(x)
print(z)
```

```
[1 2 3 4 5]
[3 4 5]
```

```
x = np.arange(1, 6)

z = x[(x>2) & (x<5)]

print(x)
print(z)
```

```
[1 2 3 4 5]
[3 4]
```

Funktionen

Wir können das Beispiel aus dem Showcase im letzten Kapitel aufgreifen und sehen, dass man eine Operation auf ein Array prinzipiell genauso anwenden kann, wie auf eine einzelne Zahl, mit dem Unterschied, dass die Operation auf jede Zahl im Array angewendet wird. Anstatt nur zu quadrieren, kann die Operation auch komplexer sein. Multipliziert man zwei Arrays derselben Länge miteinander, so wird das Produkt von den jeweiligen Elementen gebildet.

```
x = np.arange(1, 6)
y = 0.5 - (x**2 + 2)

print(x)
print(y)
```

```
[1 2 3 4 5]
[ -2.5  -5.5 -10.5 -17.5 -26.5]
```

```
x = np.arange(1, 4)
y = np.arange(2, 5)
z = x*y

print(x)
print(y)
print(z)
```

```
[1 2 3]
[2 3 4]
[ 2  6 12]
```

Es kann allerdings nicht wie in Kapite 2.1 die Wurzel mittels `math.sqrt()` gezogen werden, da `math.sqrt()` nur auf einzelne Zahlen angewendet werden kann. Stattdessen gibt es in NumPy die Funktion `np.sqrt()`, die auf Arrays angewendet werden kann.

```
import math

x = np.arange(4, 6)
y = math.sqrt(x)
```

```
TypeError: only length-1 arrays can be converted to Python scalars
```

```
#
```

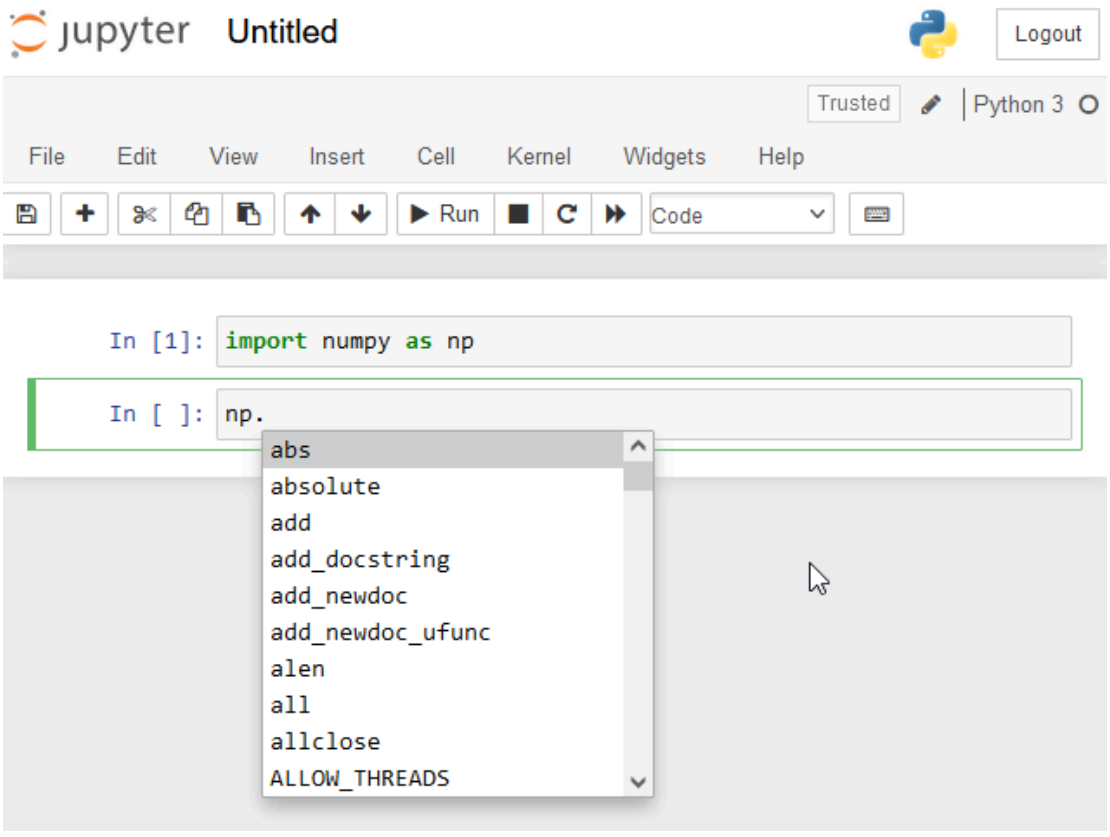
```
x = np.arange(4, 6)
y = np.sqrt(x)

print(x)
print(y)
```

```
[4 5]
[2.         2.23606798]
```

i Hinweis

NumPy bietet neben `np.sqrt()` eine Reihe solcher Funktionen, die auf Arrays angewendet werden können. Als genereller Tipp und um sich einen Überblick zu verschaffen kann in Jupyter Notebooks in einer Code-Zelle `np.` eingegeben werden und dann mittels `TAB` (Tabulator-Taste) die Autovervollständigungsvorschläge angezeigt werden:



Aggregatfunktionen

Es gibt auch sogenannte *Aggregatfunktionen*, die auf Arrays angewendet werden können. Das sind Funktionen, die aus einem Array einen einzelnen Wert berechnen. Ein Beispiel ist die Funktion `np.sum()`, die die Summe aller Elemente in einem Array berechnet. Diese Aggregatfunktionen unterscheiden sich also nicht von den anderen NumPy-Funktionen dahingehend, dass sie sich auf alle Elemente des Arrays bezieht, sondern darin, dass sie nur ein einzelnes, aggregiertes Ergebnis zurückgeben. Solche Aggregatfunktionen wie `np.mean()`, `np.median()`, `np.min()`, `np.max()`, `np.std()` und `np.var()` sind für die Datenanalyse unentbehrlich, sodass wir im nächsten Kapitel noch genauer darauf eingehen werden wie all diese Maße zu nutzen sind.

Für den Moment seien lediglich zwei Beispiele gegeben:

```
temperaturen = np.array([20, 21, 22, 23, 27])

durchschnitt_temp = np.mean(temperaturen)
print(durchschnitt_temp)
```

```
22.6
```

```
einnahmen = np.array([1.99, 2.49, 19.99])

print(np.sum(einnahmen))
print(np.sum(einnahmen[einnahmen < 10]))
```

```
24.47
4.48
```

Weitere Ressourcen

- Learn NUMPY in 5 minutes - BEST Python Library!
- Cheat Sheet: Python for Data Science - Jupyter Notebook

Übungen

Bereche den Mittelwert, Median und das Maximum des folgenden Arrays:

```
mein_array = np.array([1, 2, 3, 3, 3, 4, 12])
```

Mittelwert: _Median: _Maximum: __

Jetzt slice das Array so, dass das erste und letzte Element entfernt werden und berechne erneut Mittelwert, Median und das Maximum.

Mittelwert: _Median: *Maximum:*