MATHEMATIK MIT PYTHON, TEIL 1

Arrays mit Schleife

Die leistungsfähigen Arrays der Mathe-Bibliothek numpy im praktischen Einsatz.

Die Lösungen wissenschaftlich oder technisch orientierter Probleme findet man oft schon vorgefertigt in Python-Bibliotheken. In dieser kleinen Artikelserie werden insbesondere die Python-Bibliotheken numpy, scipy und sympy vorgestellt. Dieser erste Teil der Serie zeigt die Bordmittel von Python, stellt die Arrayfunktionen von numpy vor und zeigt, wie Sie das Modul *matplotlib* zur Anzeige von Daten benutzen. Sollten Sie jetzt die Nase rümpfen, weil Sie glauben, Python sei eine langsame Skriptsprache, dann sollten Sie bedenken, dass die genannten Bibliotheken hocheffizient in C/C++ implementiert wurden und vorkompiliert in Python geladen werden. In Sachen Performance können sie sich deshalb durchaus mit Eigenimplementierungen messen.

Die allererste Entscheidung, die Python-Entwickler treffen müssen, ist, welche Version sie nutzen wollen: Python 2.7 oder 3.x. Ich benutze wenn möglich Python 3, da dort einige wichtige Neuerungen eingeführt wurden, die das Programmieren erleichtern [1][2]. Da eine der Änderungen den hier abgedruckten Code betrifft, sei sie kurz vorgestellt: In Python 2 lassen sich Ausgaben mit einem *print*-Befehl mit oder ohne Klammer erstellen, in Python 3 muss man dagegen die Klammer-Schreibweise benutzen:

```
# Python 2
print 'Hello, World!'
# Python 3
print ('Hello, World!')
```

Selbstverständlich gibt es in Python 3 viele weitere Erweiterungen, die aber nicht Thema dieses Artikels sind.

Listing 1: Einfaches Rechnen

```
import math

a = ((3 + 5) / 2 + 3) * (2 + 1)**2
print ("Klammern:", a)
b = math.sqrt(a)
print ("Wurzel:",b)
c = math.fsum([ 1.5, 2.3, 1.9, -3.0, 2.8])
print ("Summe exakt:", c)
c1 = sum([ 1.5, 2.3, 1.9, -3.0, 2.8])
print ("Summe:", c1)
d = math.exp(-c)
print ("Exp.:", d)
```

Python-Programmierumgebungen (IDEs) gibt es sowohl für Linux und Mac als auch für Windows. Ist eine IDE installiert, finden Sie mithilfe der folgenden Zeilen heraus, welche Python-Version Sie gerade benutzen:

```
from platform import python_version
print ("Python: ", python_version())
```

Auch wenn diese ersten Codezeilen den Eindruck vermitteln mögen, so ist diese Artikelserie dennoch kein Einführungskurs in Python, sondern sie dreht sich um das Thema Mathematik mit Python. Suchen Sie eine Einführung, so werden Sie unter [3] und [4] fündig.

Einfache Rechenaufgaben

Selbstverständlich kann man mit Python auch Rechenoperationen durchführen, ohne eine spezielle Bibliothek dafür zu benutzen. Dazu zählen die vier Grundrechenarten sowie das Setzen von Klammern, um die Ausführungsreihenfolge zu kontrollieren. Für leistungsfähigere mathematische Funktionen müssen Sie jedoch das Modul *math* importieren. Listing 1 zeigt einige Beispiele.

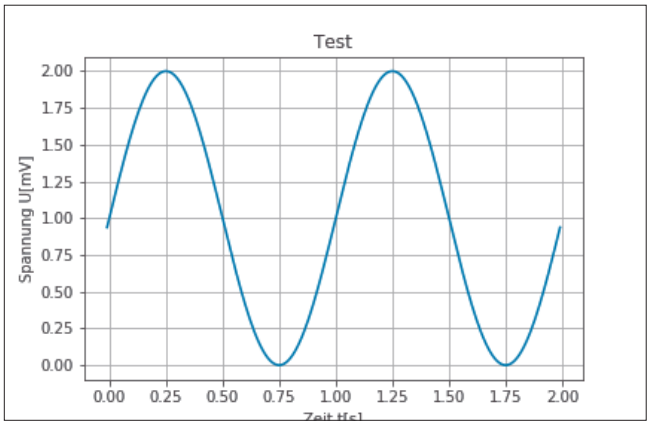
Die Funktion *fsum* im Modul *math* garantiert übrigens Arithmetikgenauigkeit nach IEEE-754. Der Unterschied zur Standardfunktion *sum* ist zur Laufzeit erkennbar. Das Modul *math* stellt folgende Funktionsgruppen zur Verfügung:

Listing 2: Rechnen mit komplexen Zahlen

```
import cmath

a = 3.5 + 4j
print ("Complex:", a)
print ("Real: ", a.real)
print ("Imag.:", a.imag)

b = -1.2 - 1j
c = a + b

print ("Addition:", c)
d = cmath.sqrt(a)
print ("Komplexe Wurzel:", d)

e = cmath.tan(a)
print ("Komplexer Tangens:", e)
```

Listing 3: Grafik ausgeben

```
import matplotlib.pyplot as plt
import numpy as np
import math

t = np.empty(201)
s = np.empty(201)
# Schleife von 0 bis einschließlich 200
for i in range(201):
    t[i] = (i - 1) * 0.01
    s[i] = 1.0 + math.sin(2 * math.pi * t[i])

# Grafik erstellen und ausgeben
plt.plot(t, s)
plt.xlabel('Zeit t[s]')
plt.ylabel('Spannung U[mV]')
plt.title('Test')
plt.grid(True)
plt.savefig("test.png")  # Speichern
plt.show()               # Darstellen
```



Plotten mit Python (Bild 1)

- Zahlentheoretische Funktionen
- Darstellungsfunktionen
- Logarithmische Funktionen
- Trigonometrische Funktionen
- Hyperbolische Funktionen
- Konvertierungen
- Spezielle Funktionen
- Konstanten

Zu den speziellen Funktionen gehören etwa die Fehlerfunktion *erf(x)* und die Gamma-Funktion *gamma(x)*. Unter den Konstanten finden Sie zum Beispiel die Zahlen *pi* und *e*.

Die meisten Programme rechnen mit ganzen Zahlen oder Fließkommazahlen. Python unterstützt beide Typen. In Python 3 können Sie sogar mit beliebig langen Integer-Zahlen rechnen. Darüber hinaus gibt es den Typ der komplexen Zahlen, die sich aus einem realen und einem imaginären Anteil zusammensetzen. Die Rechenregeln für die Addition und Subtraktion komplexer Zahlen sind relativ einfach, siehe Listing 2. Das Python-Modul *cmath* stellt zudem erweiterte Funktionen für komplexe Zahlen zur Verfügung, etwa *sqrt*, *sin*, *asin* oder *exp*, deren mathematische Grundlagen hier nicht erläutert werden.

Noch bevor es um weitere technische und wissenschaftliche Möglichkeiten von Python geht, soll betrachtet werden, wie sich Daten mit Python visualisieren lassen. Dabei hilft das Modul *matplotlib.pyplot*, das Methoden zum Zeichnen von 2D- und 3D-Grafiken enthält.

Der Code in Listing 3 berechnet die darzustellenden Daten sehr konservativ in einer Schleife. Dazu werden zwei leere Arrays angelegt und in einer *for*-Schleife mit Daten gefüllt. Das Ergebnis soll in einem XY-Diagramm ausgegeben werden. Im Beispiel kommen dafür drei Module zum Einsatz: *matplotlib* für die grafische Darstellung, *math* für die Sinus-Berechnung und *numpy* zum Erzeugen der leeren Arrays.

Der *plot*-Funktion werden die beiden Daten-Arrays übergeben. Danach bietet die Funktion mehrere Optionen, um die Grafikausgabe zu erweitern oder zu formatieren. Hier wird ein Text für die beiden Achsen und den Titel definiert und mit *grid* die Darstellung eines Rasters bestimmt. Der Aufruf von *savefig(„test.png“)* speichert die Grafik als PNG-Datei. Die *show*-Funktion zeigt die Grafik in einem separaten Fenster beziehungsweise in der Python-Konsole an, siehe Bild 1.

Wer das Modul *numpy* bereits kennt, wird sich über das obige Beispiel gewundert haben, denn *numpy* kann viel mehr, als leere Arrays bereitstellen. Listing 4 zeigt einige der Tricks, die *numpy* in Sachen Arrays beherrscht. Es hat sich übrigens als Quasi-Standard etabliert, das *numpy*-Modul unter dem Alias *np* zu importieren.

Auf die Import-Befehle folgen die interessantesten Zeilen des Listings: Das Erzeugen der Arrays sowie die explizit programmierte Schleife wurden durch Aufrufe im *numpy*-Modul ersetzt. Zunächst wird mit der *arange*-Funktion ein Array *t* erzeugt, welches die Werte von 0.0 bis 2.0 mit einer Schrittweite von 0.01 enthält. Im zweiten Schritt wird das Array *s* erzeugt und für jeden Wert im Array *t* ein Wert im Array *s* berechnet, ganz ohne Schleife. Allerdings ist zu beachten, dass nun nicht mehr die Funktion *math.sin* benutzt werden darf, sondern die Sinus-Funktion *np.sin* zum Einsatz kommt. Die Ausgabe der Grafik gleicht hier dem vorherigen Beispiel.

Die Varianten in Listing 3 und 4 benötigen sehr unterschiedliche Rechenzeiten. Bei der zweiten Variante wird nicht intern parallelisiert, sondern es werden die Vektoreigenschaften der Prozessoren – sofern vorhanden – ausgenutzt. Diese Verfahren sind unter den Namen *SSE* (Streaming SIMD Extensions) oder *AVX* (Advanced Vector Extensions) bekannt.

Tabelle 1 zeigt die Rechenzeiten der beiden Python-Varianten und eines entsprechenden C#- beziehungsweise C++-Programms. Die Anzahl der Schleifendurchläufe wurde bei der Messung auf 2.000.000 erhöht. In allen Beispielen läuft jeweils nur ein Thread.

Das Python-Modul *matplotlib* ist sehr leistungsfähig und enthält zugleich Funktionen für andere Darstellungsarten ▶

Listing 4: Vereinfachte Arrays

```
import matplotlib.pyplot as plt
import numpy as np

# Berechnung über Arrays
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

# Grafik erstellen und ausgeben
plt.plot(t,s)
plt.xlabel('Zeit t[s]')
plt.ylabel('Spannung U[mV]')
plt.title('Test')
plt.grid(True)
plt.savefig("test.png")  # Speichern
plt.show()               # Darstellen
```

der Daten. Dies soll ein weiteres einfaches Beispiel zeigen. Dafür wird eine Funktion *f(x, y)* definiert, die einen Wertebereich aufspannt. Diese Daten sollen in einem sogenannten Höhenliniendiagramm dargestellt werden. Listing 5 zeigt den Quellcode dazu.

Zunächst wird die Python-Funktion *f(x, y)* definiert, welche die Höhenwerte in Abhängigkeit von den Koordinaten *x* und *y* berechnet. Danach wird mit *meshgrid* ein passendes Gitternetz der Berechnungspunkte erstellt. Mit der *axes*-Funktion kann man festlegen, welcher Bereich des Fensters mit der eigentlichen Grafik auszufüllen ist. Im Beispiel sollen links und unten jeweils fünf Prozent freier Platz bleiben. Die übrigen 90 Prozent der Fensterbreite und -höhe darf die Grafik einnehmen. In drei Schritten wird nun der Höhenlinien-Plot vorbereitet und gezeichnet: Die *contourf*-Funktion stellt zunächst die einzelnen Höhenflächen farblich dar, danach werden die schwarzen Trennlinien für die Höhenbereiche eingezeichnet und im dritten Schritt die Beschriftungen an den Höhenlinien angebracht. Es folgen das Raster (*grid*) und die Achsenbeschriftungen (*xticks* und *yticks*), bevor die Grafik in einem Fenster ausgegeben beziehungsweise gespeichert wird (Bild 2).

Tabelle 1: Rechenzeiten

Variante	Rechenzeit (Sekunden)
Normale for-Schleife	1,83
numpy-Array	0,11
.NET/C#	0,094
C++	0,074

Noch einmal Python-Arrays

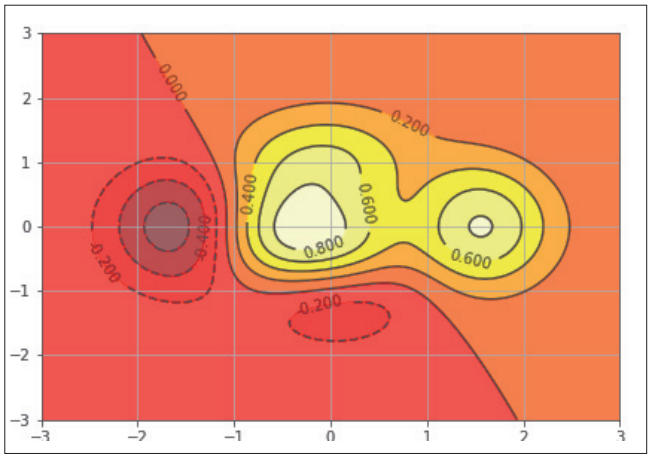
In Listing 4 wurden die Arrays aus dem Modul *numpy* bereits benutzt. Diese Arrays sind jedoch ein wesentlicher Teil von Python, der nun genauer behandelt werden soll. Grundsätzlich wird durch diese Arrays folgende generelle Vereinfachung eingeführt:

```
# Aus der Schleife:
c = []   # Leeres Array
for i in range(n):
    c.append(a[i] * b[i])

# wird die einfache Zeile:
c = a * b
```

Man spart also die Schleife und die Indizierung der Variablen. Außerdem lässt sich der Code für Vektoroperationen op-

Listing 5: Höhenlinien

```
import matplotlib.pyplot as plt
import numpy as np

# Funktion
def f(x, y):
    return (0.9 - x/2 +
            x**5 + y**3) *
            np.exp(-x**2 - y**2)

# Anzahl der Punkte in X und Y
n = 256
xx = np.linspace(-3, 3, n)
yy = np.linspace(-3, 3, n)
X, Y = np.meshgrid(xx, yy)

# Grafikausgabe
plt.axes([0.05, 0.05, 0.9, 0.9])

plt.contourf(X, Y, f(X, Y), 8, alpha = 0.75,
             cmap = plt.cm.hot)
C=plt.contour(X, Y, f(X, Y), 8, colors = 'black',
              linewidth = 0.5)
plt.clabel(C, inline = 1, fontsize = 10)

plt.grid()
plt.xticks()
plt.yticks()

plt.savefig("test.png") # Speichern
plt.show()              # Darstellung Fenster
```



Ein Höhenlinien-Plot mit Python (Bild 2)

timieren, die wesentlich schneller als normale Schleifendurchläufe sind, wie die Laufzeiten in Tabelle 1 belegen.

Die Arrays aus dem Modul *numpy* werden über die Klasse *ndarray* implementiert. Darum hört man oft auch den Begriff *ndarray*-Objekt. Die *ndarray*-Objekte lassen sich in einer vorgegebenen Größe erstellen, sie können einen Rang (Dimensionalität) und einen Typ haben und in unterschiedlicher Weise initialisiert werden. Eine interessante Vorgehensweise beim Initialisieren eines Arrays ist der Aufruf einer Funktion, die einen Wert zurückgibt:

```
import numpy as np
def f(i, j):
    return 2 * i * j
```

▶

Listing 6: Array-Operationen

```
import numpy as np

# Matrix transponieren
a = np.linspace(1,6,6).reshape(3,2)
print ("Ursprung:\n", a)
b = a.transpose()
print ("Transponiert:\n", b)

# Arrays zusammenfassen
c1 = np.array([0, 0, 0, 0])
c2 = np.array([1, 1, 1, 1])
c = np.vstack((c1, c2))

print ("Vertikal zusammen:\n", c)
d = np.hstack((c1, c2))
print ("Horizontal zusammen:\n", d)

# Arrays splitten
e = np.hsplit(d, 4)
print ("Aufgesplittet:\n", e)
```

```
a = np.fromfunction(f, (4, 3))
print (a)
```

Mit diesen Zeilen erstellen Sie ein Array *a*, welches aus vier Zeilen und drei Spalten besteht. Die Funktion *f(i, j)* wird mit den Werten für *i* und *j* aufgerufen und liefert das Ergebnis zurück, mit welchem das jeweilige Array-Element initialisiert wird. Die Ausgabe sieht also so aus:

```
[[  0.   0.   0.]
 [  0.   2.   4.]
 [  0.   4.   8.]
 [  0.   6.  12.]]
```

Auf diese Weise lassen sich Arrays sehr schnell und einfach mit beliebigen Daten initialisieren. Im Speicher werden *ndarray*-Objekte in der Reihenfolge „Zeile zuerst" abgelegt. Das heißt, in einem zweidimensionalen Array werden die Daten Zeile für Zeile hintereinander abgelegt:

```
import numpy as np
b = np.array( ((1,2), (3,4)) )
print (b)
print (b.flatten())
```

Der erste *print*-Befehl zeigt das normale Array in zwei Dimensionen, mit dem zweiten Befehl werden die Daten sequentiell ausgegeben, so wie sie im Speicher stehen:

```
[[1 2]    # Erstes print
 [3 4]]

[1 2 3 4] # Zweites print
```

Selbstverständlich lassen sich diese Arrays auch in der Größe oder in ihrer Anordnung (Dimensionalität) ändern. Hier müssen Sie jedoch beachten, dass die Reihenfolge der Daten im Speicher nicht verändert wird. Neu hinzukommende Array-Elemente werden mit *Null* initialisiert.

Einige Array-Operationen, die in der mathematischen Praxis häufig auftreten, sollen hier noch an einem weiteren Beispiel vorgestellt werden: das Transponieren einer Matrix sowie das Zusammenfassen und Aufsplitten von Arrays, vergleiche Listing 6).

Listing 7: Array-Indizierungen

```
import numpy as np

# Einfache Indizierung
a = np.linspace(1,6,6)
print (a)
# Ab a[1] bis a[4] jedes zweite Element
print (a[1:4:2])
# Ab a[2] bis zum vorletzten Element
print (a[2:-1])
# Ab a[3] jedes zweite Element rückwärts
print (a[3::-2])
print ()

# Etwas komplexere Indizierung
b = np.linspace(1,12,12).reshape(4,3)
print (b)
# Die komplette dritte Zeile
print (b[2,:])
# Die komplette zweite Spalte
print (b[:,1])
# Von der zweiten Zeile bis zur vorletzten Zeile
# und von der zweiten Spalte bis zum Ende
print (b[1:-1,1:])
# Vom Anfang jede zweite Zeile
print (b[::2,:])
# Zwei Zeilen und Spalten unten links
print (b[2:,:2])
# Jedes zweite Element in jeder zweiten Zeile
print (b[1::2,::2])
```

Listing 8: Ein vollständiges Beispiel

```
import numpy as np
import matplotlib.pylab as plt

# Anzahl der Werte
N = 100

# Funktion für die Berechnung
def f(i, n):
    x = i / N
    lam = 2 / (n+1)
    return x * (1-x) * np.sin(2 * np.pi * x / lam)

# Daten berechnen
a = np.fromfunction(f, (N+1, 5))
# Wo sind Maxima und Minima
min_i = a.argmin(axis=0)
max_i = a.argmax(axis=0)

# Plotten der Funktionslinien
plt.plot(a, color='k')
# Plotter der Minima und Maxima
plt.plot(min_i, a[min_i, np.arange(5)], 'v',
         color='r', markersize=10)
plt.plot(max_i, a[max_i, np.arange(5)], '^',
         color='b', markersize=10)

# Grafik im Fenster zeigen und speichern
plt.savefig("test.png")
plt.show()
```

Wenn dieses Programm ausgeführt wird, erscheinen folgende Ausgaben auf dem Bildschirm:

```
Ursprung:
 [[ 1.  2.]
  [ 3.  4.]
  [ 5.  6.]]

Transponiert:
 [[ 1.  3.  5.]
  [ 2.  4.  6.]]

Vertikal zusammen:
 [[0 0 0 0]
  [1 1 1 1]]
Horizontal zusammen:
 [0 0 0 0 1 1 1 1]

Aufgesplittet:
 [array([0, 0]), array([0, 0]), array([1, 1]),
  array([1, 1])]
```

Im ersten Teil des Beispiels wird ein zweidimensionales Array mit drei Zeilen und zwei Spalten erstellt. Hierzu findet die Funktion *linspace* Verwendung. Die drei Parameter in diesem Aufruf sind Startwert, Endwert und Anzahl der Werte. Es wird jedoch ein lineares, eindimensionales Array erstellt, welches im zweiten Schritt zu einem Array mit drei Zeilen und zwei Spalten umgeordnet (*reshape*) wird. Dieses Array lässt sich mit der *transpose*-Funktion „stürzen", das heißt, aus den Zeilen werden Spalten und aus den Spalten werden Zeilen. Danach werden zwei eindimensionale Arrays *c1* und *c2* erstellt und zunächst vertikal mit der *vstack*-Funktion und dann horizontal mit der *hstack*-Funktion zusammengefasst. Das Aufspalten eines großen Arrays in mehrere kleinere Arrays klappt mit *hsplit* (horizontal) oder *vsplit* (vertikal). Im Beispiel wird das Array *d*, welches acht Zahlen enthält, in vier Arrays mit jeweils zwei Zahlen aufgeteilt.

Als Nächstes wollen wir die Indizierung von *numpy*-Arrays etwas genauer unter die Lupe nehmen, da es dort sehr viele interessante Möglichkeiten gibt. Das Beispiel aus Listing 7 erzeugt die folgenden Ausgaben:

```
[ 1.  2.  3.  4.  5.  6.]
[ 2.  4.]
[ 3.  4.  5.]
[ 4.  2.]

[[  1.   2.   3.]
 [  4.   5.   6.]
 [  7.   8.   9.]
 [ 10.  11.  12.]]
[ 7.  8.  9.]
[  2.   5.   8.  11.]
[[ 5.  6.]
 [ 8.  9.]]
[[ 1.  2.  3.]
 [ 7.  8.  9.]]
[[  7.   8.]
 [ 10.  11.]]
[[  4.   6.]
 [ 10.  12.]]
```

Im ersten Teil des Beispiels wird ein eindimensionales Array benutzt, welches sechs Elemente enthält. Die Indizierung des Arrays funktioniert so:

```
array[start,ende,schrittweite]
```

Teile der Indizierung können weggelassen werden. Wenn man zum Beispiel die Angabe für das Ende weglässt, so wird das gesamte Array bis zum letzten Element benutzt. Allerdings sind alle Doppelpunkte richtig zu setzen. In Listing 7 ist in Kommentarzeilen angegeben, welche Datenelemente ausgegeben werden.

Noch interessanter sind die Ausgaben im zweiten Teil des Beispiels, da hier ein zweidimensionales Array benutzt wird. Trotzdem bleiben die Regeln dieselben wie eben beschrieben. Sie können ganze Zeilen oder Spalten aus einem Array mit einem einzigen Befehl herauslösen. Bei einer solchen Operation entsteht wieder ein neues Array, mit dem Sie weiterarbeiten können. Stellvertretend soll hier noch das letzte Beispiel aus Listing 7 erklärt werden:

```
print (b[1::2,::2])
```

Da alle Indizes mit 0 beginnen, ergibt sich folgende Operation: Ab der zweiten Zeile soll jede zweite Zeile, ab der ersten Spalte jede zweite Spalte benutzt werden. Die Endwerte wurden nicht angegeben, da jeweils bis zum Ende der Spalten und Zeilen gegangen werden soll. Außerdem ist es nicht nötig, einen Startwert für die Spalten anzugeben, da mit der ersten Spalte begonnen werden soll.

Es sei noch erwähnt, dass Sie die zuletzt gezeigten Möglichkeiten nicht nur bei der Ausgabe von Teil-Arrays nutzen können, vielmehr gelten die gleichen Syntax-Regeln auch beim Setzen oder Ändern von Array-Elementen:

```
v = np.array([1,1,1,1,1,1,1,1])
v[3::2] = 5
print (v)
```

In diesem Beispiel wird ab Array-Element *v[3]* in jedem zweiten Element die Zahl 1 durch eine 5 ersetzt. Als Ausgabe erhalten Sie also:

```
[1 1 1 5 1 5 1 5]
```

Diese Beispiele zeigen, wie leistungsfähig die *numpy*-Arrays sind. Durch geschickten Einsatz dieser Arrays lassen sich viele Zeilen Code einsparen. Trotzdem ist diese Art und Weise, Datenfelder zu erzeugen, zunächst etwas gewöhnungsbedürftig. Eine ebenfalls sehr spannende Array-Syntax zeigt noch folgendes Beispiel:

```
import numpy as np

x = np.array([-5, 3, -4, 7, 0, -5, -3, 5])
x[x < -3] = 0
print (x)
```

Nach Ausführen dieses kleinen Beispiels enthält das Array x die folgenden Zahlenwerte:

```
[ 0  3  0  7  0  0  -3  5]
```

Alle Werte im Array x, die kleiner als -3 sind, wurden durch den Wert 0 ersetzt, wieder ganz ohne explizite *for*-Schleife.

Von Min zu Max

Die Arrays im Modul *numpy* können auch mathematische Operationen ausführen. An dieser Stelle sollen die Berechnung von Maximal- und Minimalwerten und die Sortierung von Arrays gezeigt werden:

```
import numpy as np
# Maximum und Minimum (einfach)
a = np.array([1., 4.5, 0.5, 5.6, 8.9, 3.7, 9., -2.4])

print ("Max.:", a.max())
print ("Min.:", a.min())
```

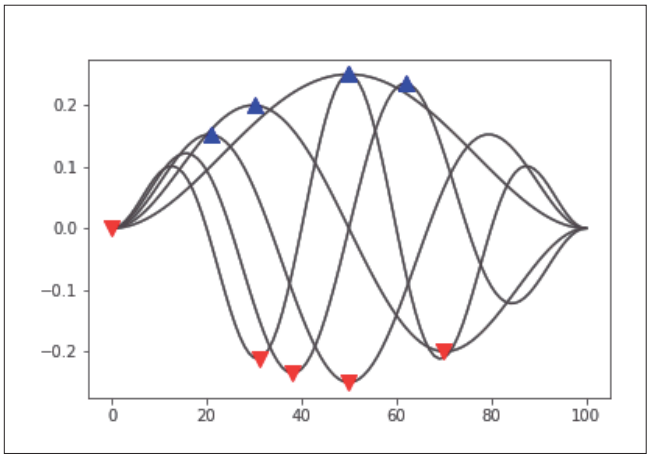
Als Ergebnis liefert dieser Code 9.0 als Maximum und -2.4 als Minimum. Es gibt aber auch noch weitere interessante Aspekte: Zum einen können Sie mehrdimensionale Arrays benutzen und dann beispielsweise die Maxima und Minima der Spalten oder Zeilen eines zweidimensionalen Arrays ermitteln. Als Ergebnis liefert *numpy* in diesem Fall ein Array der entsprechenden Größe. Mit dem *axis*-Parameter können Sie dabei angeben, ob über Spalten oder Zeilen gerechnet werden soll.

```
import numpy as np
# Maximum und Minimum (Array)
b = np.array([[1,2,3], [4,5,6], [7,8,9]])

mi1 = b.min(axis=0)   # [1 2 3]
mi2 = b.min(axis=1)   # [1 4 7]
ma1 = b.max(axis=0)   # [7 8 9]
ma2 = b.max(axis=1)   # [3 6 9]
```

Der Aufruf der *argmin*- oder *argmax*-Funktion liefert den Integer-Index des Array-Elements mit dem jeweils kleinsten oder größten Wert.

Die Sortierung von Arrays funktioniert übrigens in gleicher Weise und kann bei mehrdimensionalen Arrays ebenfalls mit dem *axis*-Parameter gesteuert werden.

Zum Schluss noch ein Beispiel, das Arrays, grafische Ausgabe und Berechnungsfunktionen im Zusammenspiel effizient ausnutzt. Es sollen mehrere Sinus-Funktionen mit unterschiedlicher Frequenz berechnet und gezeichnet werden. Zudem gilt es, die Maximal- und Minimalwerte der einzelnen



Sinus-Kurven, berechnet mit Listing 8 (Bild 3)

Funktionen zu kennzeichnen. Listing 8 zeigt den Python-Code für diese Aufgabe. Der Code ist nahezu selbsterklärend. Anzumerken ist, dass *color='k'* die Zeichenfarbe Schwarz bedeutet. Interessant ist hier, wie einfach sich das Zeichnen der Minima- und Maxima-Punkte gestaltet. Die Arrays *min_i* und *max_i* enthalten jeweils die Array-Elementnummern der kleinsten beziehungsweise größten Funktionswerte. Diese Indizes werden zusammen mit den Daten im Array *a* benutzt, um die Dreiecke an den richtigen Positionen zu zeichnen. Bild 3 zeigt die fertige Grafik.

Fazit

In diesem ersten Teil der Artikelserie wurde gezeigt, wie stark Sie Berechnungen mithilfe des Moduls *numpy* vereinfachen können. Die meisten Schleifen fallen weg, der Code wird besser lesbar und deutlich kürzer. Und weniger Code heißt letztendlich auch weniger Fehler!

Der kommende zweite Teil des Artikels erklimmt etwas höhere Sphären der Mathematik. Vorgestellt werden Statistik-Funktionen sowie Polynome, und die lineare Algebra wird auch nicht zu kurz kommen. ■

[1] Neues in Python 3, www.dotnetpro.de/SL1712Rechnen1
[2] Auswahlhilfe Python 2 oder 3, www.dotnetpro.de/SL1712Rechnen2
[3] Tutorial zu Python 3.3, www.dotnetpro.de/SL1712Rechnen3
[4] Python, Einfach & schnell programmieren, www.dotnetpro.de/SL1712Rechnen4

Bernd Marquardt
ist selbstständiger Berater und freier Autor. Seine Interessen liegen beim Programmieren grafischer und mathematischer Algorithmen, der Parallelprogrammierung und dem .NET Framework.

dnpCode A1712Rechnen