Geht es um numerische Problemstellungen aller Art, ist das Modul Numpy in Python die erste Wahl. Es erlaubt Matrix- und Vektorberechnungen, und ist damit gerade bei größeren Datenmengen ein gutes Stück schneller als handgeschriebener Python-Code.

Erste Schritte

Erstellen von Numpy-Arrays

Die meiste Zeit arbeitet man in Numpy mit den sogenannten Numpy-Arrays. Diese verhalten sich ähnlich wie Listen, es lassen sich aber nachträglich keine neuen Elemente hinzufügen.

Erzeugen eines Arrays aus einer Liste

Ein Numpy-Array lässt sich leicht aus einer Liste erstellen:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print type(arr)
```

Ausgabe:

```
numpy.ndarray
```

Shape - Die Dimensionen eines Arrays

Wir haben im obigen Beispiel gesehen, dass das erzeugte Objekt vom Typ ndarray ist. Ein ndarray kann, wie gezeigt, eine Dimension haben (entspricht einer einfache Liste), durch Schachtelung von Listen können wir aber auch höherdimensionierte Arrays erhalten

```
import numpy as np

arr_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9.0, 10, 11, 12]]) # Liste von Listen -> 2 D
shape = arr_2d.shape #
print shape
print "Anzahl Dimensionen:", len(shape)
for i, dim_len in enumerate(shape):
    print "Dimension", i, "hat Länge", dim_len
```

Ausgabe:

```
(3, 4)
Anzahl Dimensionen: 2
Dimension 0 hat Länge 3
Dimension 1 hat Länge 4
```

Der Tupel shape kann uns für ndarrays anzeigen, wieviele Dimensionen es gibt.

Zu beachten ist übrigens immer, dass alle Listen einer Dimension die selbe Länge haben müssen!

Erzeugen eines mit Nullen gefüllten Arrays

Um einen mit Nullen gefüllten Array zu erstellen, benutzen wir die Funktion *zeros*:

```
shape = (3, 4, 5)
arr_zeros = np.zeros(shape)
```

Erzeugen eines mit Einsen gefüllten Arrays

```
shape = (4, 5, 6)
arr_ones = np.ones(shape)
```

Erzeugen einer Zahlenreihe

Um einen Floatarray zu erzeugen, der zwischen Start- und Endpunkt Werte in regelmäßigen Abständen voneinander enthält, benutzt man *linspace*:

```
start, end = 0, 2*np.pi
size = 100
arr = np.linspace(start, end, size)
print arr
```

Erzeugen von zufälligen Arrays

Für Integers benutzt man:

```
start = 10
end = 25 # Achtung, end ist nicht enthalten!
shape = (5, 5)
np.random.randint(start, end, size=shape)
```

Für Float:

```
start = 3.7
end = 10.0 # nicht enthalten!
shape = (3, 4)
np.random.uniform(start, end, size=shape)
```

Datentyp eines Arrays

Bisher haben wir Numpy-Arrays erzeugt, ohne uns Gedanken darüber zu machen, welchen Datentyp sie eigentlich besitzen.

Den Datentyp können wir uns ansehen, wenn wir das Attribut *dtype* eines Arrays aufrufen

```
print arr.dtype
print arr_2d.dtype
print arr_zeros.dtype
```

Ausgabe:

```
int64
float64
float64
```

Der Grund für diese Ausgabe: im ersten Beispiel wurde ein Array aus einer Liste mit Integers erzeugt. Daher ist auch der resultierende Datentyp vom Typ Integer. Im zweiten Beispiel enthält die zweite Liste am Anfang einen Float, daher bekommt der gesamte Array den Typ Float (die 64 steht dabei für 64 bit, der Größe eines einzelnen Elements des Arrays). Und auch *zeros* erstellt standardmäßig einen Array vom Typen Float.

Möchten wir dieses Verhalten beeinflussen, können wir einfach beim erstellen des Arrays das Keyword *dtype=* mitgeben:

```
arr = np.array([1, 2, 3, 4, 5], dtype=float)
print arr.dtype
```

Ebenso funktioniert das bei den anderen Funktionen zur Erzeugung eines Arrays.

IO mit Numpy

Numpy macht es uns sehr einfach, numerische Daten einzulesen. In der Regel (bei gut formatierten Dateien) reicht die Funktion *loadtxt* aus:

```
data = np.loadtxt("dateiname")
```

Genauso kann man *savetxt* benutzen, um Daten auf die Fesplatte zu schreiben:

```
arr1 = np.random.randint(10, size=(25, 25))
np.savetxt("random_data", arr1)
```

Rechnen mit Arrays

Wir fangen in einer Dimension an:

Addition

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6], dtype=float)

result = arr1 + arr2
print result
print result.dtype
```

Ausgabe:

```
[ 5.  7.  9.]
float64
```

Hier sehen wir, dass die Addition zweier Arrays zu einer elementweisen Addition führt. Da arr2 vom Type float64 ist, ist auch der Array *result* vom Typ float64.

Multiplikation

Was passiert wenn wir die beiden Arrays multiplizieren?

```
result = arr1 * arr2
print result
```

Ausgabe:

```
[  4.  10.  18.]
```

Man sieht, dass auch hier wieder elementweise multipliziert wurde! Wollen wir stattdessen das Skalarprodukt berechnen, müssen wir auf die Funktion *dot* zurückgreifen:

```
result = np.dot(arr1, arr2)
print result
```

Ausgabe:

```
32.0
```

Boolsche Operationen mit Numpy-Arrays

Auch boolsche Operationen werden elementweise angewandt.

```
arr1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2 = np.array([[2, 3, 4], [4, 5, 6], [9, 8, 7]])

print arr2 > arr1
```

Ausgabe:

```
[[ True  True  True]
 [False False False]
 [ True False False]]
```

Das Ergebnis ist in diesem Beispiel also wieder ein zweidimensionaler Array.

Weitere elementweise Operationen

Alle bekannten Funktionen wie cos, sin, exp, ... lassen sich auch in Numpy elementweise auf einen Array anwenden:

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print np.sin(arr)
print np.exp(arr)
print np.sqrt(arr)
```

Ausgabe:

viele Zahlen

Höhere Dimensionen

Arbeiten wir mit Arrays in höheren Dimensionen, funktioniert bei elementweisen Operationen alles wie in einer Dimension.

Die Funktion *dot* ermöglicht für zwei Dimensionen Matrix-Multiplikation. Bei Kombination von 2D- und 1D-Arrays wird Matrix-Vektor-Multiplikation angewandt.

sum, mean und var

Numpy bringt eigene Methoden mit, um über Arrays zu summieren (sum), sie zu mitteln (mean), oder die Varianz auszurechnen (var).

Einfachstes Beispiel:

```
arr = np.array([1, 2, 3])
print arr.sum()
print arr.mean()
print arr.var()
```

mit der Ausgabe

```
6
2.0
0.666666666667
```

Liegt der Array in höheren Dimensionen vor, so wird standardmäßig über alle Elemente summiert/gemittelt/die Varianz gebildet.

In manchen Fällen ist das jedoch nicht das, was wir möchten. Wenn wir beispielsweise einen Array mit mehreren 3D-Koordinaten haben, und davon den geometrischen Schwerpunkt berechnen wollen, so müssen wir die x-, y- und z-Koordinaten aller Atome separat mitteln.

```
atoms = np.array([[5, 9, 4],
                  [8, 8, 8],
                  [3, 0, 4],
                  [1, 7, 8]])
```

Dafür geben wir der mean-Methode noch den zusätzlichen Parameter *axis* mit. In diesem Fall wäre das die 0. Dimension. Numpy mittelt dann über alle Zeilen des Arrays

```
geometric_center = atoms.mean(axis=0)
print geometric_center
```

Damit erhalten wir aus dem Array atoms mit der shape (4, 3) einen neuen eindimensionalen Array der shape (3,).

```
array([ 4.25,  6.  ,  6.  ])
```

Für axis=1 würde dementsprechend über alle Spalten jeder Zeile gemittelt:

```
array([ 6.        ,  8.        ,  2.33333333,  5.33333333])
```

all und any

Ähnlich wie sum, mean und var, gibt es für boolsche Arrays die Methoden all und any. Dabei gibt all True zurück wenn alle Einträge des Arrays True sind. Any gibt True zurück, falls einer der Einträge True ist.

Auch hier können wir wieder das keyword axis angeben, was zur Folge hat, dass in der angegebenen Dimension überprüft wird, ob alle Einträge/mindestens ein Eintrag True sind, und die Dimension dann auf True oder False reduziert wird.

Beispiel:

```
bool_array = np.array([[True, False, False],
                       [True, True, True],
                       [False, False, False]])
```

```
print bool_array.any() # True, weil es Einträge im gesamten Array gibt, die True sind
print bool_array.all() # False, weil es Einträge gibt, die False sind
print bool_array.any(axis=0) # any wird auf jede Spalte angewendet
print bool_array.any(axis=1) # any wird auf jede Zeile angewendet
print bool_array.all(axis=0) # all wird auf jede Spalte angewendet
print bool_array.all(axis=1) # all wird auf jede Zeile angewendet
```

Ausgabe:

```
True
False
[ True  True  True]
[ True  True False]
[False False False]
[False  True False]
```

Broadcasting

Was passiert wenn wir versuchen, zwei Arrays unterschiedlicher Dimensionen miteinander zu addieren/multiplizieren?

Der einfachste Fall ist, wenn ein Array nur ein einziges Element enthält:

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([2])

print arr1 * arr2
```

Ausgabe:

```
[[ 2  4  6]
 [ 8 10 12]]
```

Hier wird jedes Element von arr1 mit 2 multipliziert. Wir können uns vorstellen, dass arr2 auf die Dimension von arr1 vergrößert wurde, um dann schließlich eine Multiplikation zweier Arrays mit gleichen Dimensionen durchzuführen.

arr2 kann aber auch eine komplexere shape haben:

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([[3, 4, 5]])
print arr1.shape
print arr2.shape
print arr1*arr2
```

Ausgabe:

```
(2, 3)
(1, 3)
[[ 3  8 15]
```

```
 [12 20 30]]
```

Was ist hier passiert? Wir sehen, dass beide Arrays zwei Dimensionen haben, wobei die letzten Dimensionen beide die Länge 3 haben. Die nullten Dimensionen haben jedoch unterschiedliche Größen. Numpy kopiert nun einfach die nullte Dimension von arr2 einmal, so dass arr2 identisch ist zu

```
np.array([[3, 4, 5], [3, 4, 5]])
```

Nun kann wieder elementweise gerechnet werden.

Hinweis:

Wir haben arr2 schon direkt mit zwei Dimensionen (zwei geschachtelte Klammern) initialisiert:

```
arr2 = np.array([[3, 4, 5]])
```

Tatsächlich wäre das gar nicht notwendig. Es reicht auch nur eine Dimension:

```
arr2 = np.array([3, 4, 5])
```

Numpy kann dann immer noch broadcasten. Es fügt einfach solange neue Dimensionen (der Länge 1) hinzu, bis die Anzahl der Dimensionen der beiden Arrays gleich ist.

Allgemein: Damit zwei Arrays gebroadcastet werden können, muss folgendes gelten (man fängt dabei an, die Dimensionen von hinten paarweise zu vergleichen): beide Dimensionen haben jeweils die selbe Länge oder eine davon hat die Länge 1.

Beispiel

arr1 hat shape (2, 3, 4, 1, 7, 5) arr2 hat shape (1, 3, 4, 8, 7, 1)

Diese beiden Arrays können gebroadcastet werden. Der resultierende Array hat dann die Shape (2, 3, 4, 8, 7, 5).

Folgendes Beispiel funktioniert aber nicht:

arr1 hat shape (2, 3, 4, 1, 7, 5) arr2 hat shape (1, 3, 4, 8, 7, 3)

da die letzten Dimensionen (3 und 5) nicht miteinander kompatibel sind.

Praktisches Beispiel

Eine Anwendung ist z.B. das dyadische Produkt (= Produkt aus Spalten- und Zeilenvektor, bei dem eine Matrix entsteht):

```
arr1 = np.array([[1], [3], [2]])
arr2 = np.array([2, 1, 0, 3])

mat = arr1 * arr2
```

Was passiert genau? Schauen wir uns die shapes an:

arr1.shape -> (3, 1)

arr2.shape -> (4,)

1. Numpy betrachtet die letzten Dimensionen der beiden Arrays. arr1 hat als letzte Dimensionsgröße 1, arr2 hat 4. Numpy macht daraus: arr1: (3, 4) # kopiere die letzte Dimension 4 mal -> arr1 wird zu `python np.array([[1, 1, 1, 1], [3, 3, 3, 3], [2, 2, 2, 2]])`
2. Numpy betrachtet die nächste Dimension: arr1 hat als vorletzte Dimensionsgröße 3, arr2 hat keine weitere Dimension. -> Numpy erweitert die shape von arr2 zu (1, 4) Jetzt können die Werte wieder kopiert werden -> arr2 wird zu `python np.array([[2, 1, 0, 3], [2, 1, 0, 3], [2, 1, 0, 3]])`
3. Jetzt wo beide Arrays die selbe shape haben, kann einfach elementweise multipliziert werden:

```
array([[2, 1, 0, 3],
       [6, 3, 0, 9],
       [4, 2, 0, 6]])
```

Neue Dimensionen zu einem Array hinzufügen

Wie wir gesehen haben, fügt Numpy beim Broadcasten oft neue Dimensionen ein, so dass elementweise Rechenoperationen ausgeführt werden können. In manchen Fällen müssen zusätzliche Dimensionen jedoch auch manuell eingefügt werden. Dazu folgendes Beispiel:

```
atoms = np.array([[0, 4, 8],
                  [6, 0, 2],
                  [3, 0, 8]])
masses = np.array([1.0, 1.5, 2.5])
```

Wir haben einen Array, in dem zeilenweise die Koordinaten von 3 Atomen gespeichert sind, sowie einen Array, der ihre Massen speichert. Da wir den Schwerpunkt berechnen wollen, wollen wir erst jede Atomkoordinate mit dem jeweiligen Gewicht des Atoms multiplizieren und anschließend durch die Gesamtmasse teilen. Berechnen wir jetzt aber naiv atoms*masses, tut Numpy leider nicht das, was wir wollen. Es multipliziert nun die x-Koordinaten mit der Masse von Atom 1, die y-Koordinaten mit der Masse von Atom 2 und die z-Koordinaten mit der Masse von Atom 3.

Der Grund: masses und atoms werden falsch gebroadcastet.

Shape-Betrachtung:

atoms: (3, 3)

masses: (3,)

Daraus wird:

atoms: (3, 3)

masses: (1, 3) # Numpy fügt eine Dimension am Anfang ein!

Und nun wird der Inhalt der hinteren Dimension von masses dreimal kopiert:

```
masses = np.array([[1.0, 1.5, 2.5],
                   [1.0, 1.5, 2.5],
                   [1.0, 1.5, 2.5]])
```

Um das zu verhindern, machen wir folgendes: Wir fügen eine zusätzliche Dimension hinten an masses an:

```
masses = masses[:, np.newaxis]
print masses
```

masses sieht dann so aus:

```
array([[ 1. ],
       [ 1.5],
       [ 2.5]])
```

Wir haben einen Spaltenvektor daraus gemacht!

Und diesmal broadcastet Numpy dann auch richtig: jede Atomkoordinate wird mit der entprechenden Atommasse multipliziert.

Indexing und Slicing

Indexing und Slicing funktioniert mit Numpy-Objekten ganz ähnlich wie mit Listen. Der Unterschied ist nun aber, dass auch höhere Dimensionen berücksichtigt werden müssen.

Indexing

```
mat = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

print mat[0, 0]
print mat[-1, 2]
```

Hier wird also für jede Dimension ein Index angegeben.

Alternativ kann man auch für jede Dimension eine Liste mit Indizes angeben:

```
print mat[[0, 0, 2], [-1, 0, 2]]
```

Ausgabe:

```
[ 4  1 11]
```

Diese Operation könnten wir auch in mehreren einzelnen Schritten ausführen:

```
print mat[0, -1],
print mat[0, 0],
print mat[2, 2]
```

Damit konstruieren wir einen neuen Array mit den Werten mat[0, -1], mat[0, 0] und mat[2, 2]

Slicing

```
mat = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print mat[:, 0] # erste Spalte für jede Zeile
print ""
print mat[:2, :3] # die ersten beiden Zeilen, und davon jeweils die ersten 3 Spalten
```

Ausgabe:

```
[1 5 9]

[[1 2 3]
 [5 6 7]]
```

Praktisches Beispiel: Bildbearbeitung

Das Modul PIL erlaubt uns, Bilder einzulesen, und als Numpy-Array zu bearbeiten.

```
from PIL import Image

pic_array = np.array(Image.open("blume.JPG")) # andere Dateiformate funktionieren auch
```

pic_array ist ein Numpy-Array mit den Dimensionen (höhe, breite, farbkanäle). Bei einem Farbbild haben wir dann 3 Farbkanäle: rot, grün und blau.

Als einfachstes Beispiel können wir nun aus einem Farbbild ein Schwarz-Weiß-Bild machen. Dazu mitteln wir die Werte der 3 Farbkanäle

```
black_white = pic_array.mean(axis=2)
```

Da die Kanäle in der zweiten Dimension gespeichert sind, müssen wir das auch der mean-Methode über das axis keyword mitteilen.

Um das neuerhaltene Schwarz-Weiß-Bild speichern zu können, machen wir folgendes:

```
black_white = np.array(black_white, dtype=np.uint8) # Um den Array umwandeln zu können, muss
bw_image = Image.fromarray(black_white)
bw_image.save("bw_img.jpg")
```

Ein schöner Effekt kann auch erzielt werden, indem wir die Intensität jedes Farbwertes umdrehen.

```
pic_array = np.array(Image.open("blume.JPG"))
pic_array = 255 - pic_array
Image.fromarray(pic_array).save("komplementär.jpg")
```

Fortgeschrittenes Beispiel: Blume umfärben

In diesem etwas schwierigeren Beispiel wollen wir jetzt die Blume umfärben. Nämlich von gelb zu rot. Als erstes müssen wir dazu die gelben Stellen im Bild finden. Eine kurze <beliebige Suchmaschine>-Suche wird uns zeigen, dass Gelb im RGB-Raum den Wert (255, 255, 0) hat. Wir bekommen also Gelb wenn wir Rot und Grün mischen. Es reicht nun aber nicht, den Bild-Array nur nach Pixeln zu durchsuchen, die hohe Rot- und Gelbwerte haben. Denn dann wäre auch Weiß dabei (255, 255, 255).

Wir machen also folgendes: wir suchen nach all den Pixeln, deren Rot- und Gelbwerte über dem Durchschnitt liegen, deren Weißwerte aber unter dem Durchschnitt liegen.

1. Array aus Bild erzeugen:

“‘python
from PIL import Image import numpy as np

img_arr = np.array(Image.open(“blume.JPG”)) “‘

Unser Array hat die shape (600, 800, 3), denn das Bild ist 600 Pixel hoch und 800 Pixel breit, und jeder Pixel hat 3 Farbwerte.

2. Durchschnitts-RGB-Werte berechnen:

```
python    mean_rgb = img_arr.mean(axis=(0, 1))
```

Hier mitteln wir die RGB-Werte über alle Bildpunkte, d.h. über die 0. und 1. Dimension. Das Resultat ist ein Array der shape (3,), der einen mittleren RGB-Wert des gesamten Bildes enthält.

3. Alle Bildpunkte finden, deren Rot- und Grünwerte über den Durchschnittswerten liegen.

Da wir nur an den Rot- und Grünwerten interessiert sind, müssen wir nur den Teilarray img_arr[:, :, :2] betrachten. D.h. in diesem Array fehlen die Blauwerte.

Schreiben wir nun

```
python    img_arr[:, :, :2] > mean_rgb[:2]
```

bekommen wir einen Array mit boolschen Werten zurück (shape: (600, 800, 2)). Diese sind über all dort True, wo entweder der Rot- **oder** der Grünwert über dem Durchschnittswert liegen. Das ist aber nicht was wir wollen! Wir möchten alle Pixel finden, bei denen sowohl Rot- als auch Grünwert über dem Durchschnitt liegen. Anders gesagt suchen wir in dem oberen boolschen Array alle Pixel, bei denen in der letzten Dimension nur Trues vorkommen.

Daher:

```
python    where_is_yellow = (img_arr[:, :, :2] > mean_rgb[:2]).all(axis=2)
```

Der resultierende boolsche Array mit der Shape (600, 800) sagt uns, an welchen Stellen des Bildes die Rot- und Grünwerte über dem Mittel liegen.

Darüberhinaus wollen wir aber auch nur die Stellen, bei denen der Blauwert nicht über dem Durchschnitt liegt.

Das können wir so ausdrücken:

```
python    only_little_blue = img_arr[:, :, 2] < mean_rgb[2]
```

Was uns jetzt noch fehlt ist eine komponentenweise *and* Verknüpfung, um nämlich einen Array zurückzubekommen, der uns für jeden Pixel sagt, ob es dort Gelb gibt **und** ob dort wenig blau vorkommt.

Selbstverständlich gibt es diese Funktion in numpy. Sie heißt dort *logical_and*:

```
python    yellow_and_little_blue = np.logical_and(where_is_yellow,
only_little_blue)
```

Nun sind wir fast fertig. Mittels *np.where* lassen wir uns die Indizes des obigen Arrays ausgeben, wo True gespeichert ist.

```
indices = np.where(yellow_and_little_blue)
```

und ändern die RGB Werte an diesen Stellen zu [255, 0, 0] (was der Farbe Rot entspricht).

```
img_arr[indices] = [255, 0, 0]
```

Schließlich speichern wir noch alles:

```
Image.fromarray(img_arr).save("blume_rot.jpg")
```

Und das Ergebnis ist...

... noch nicht ganz das, was wir uns erhofft hatten. Hier kommt nun ein bisschen Heuristik ins Spiel. Wenn wir die Zeile

```
where_is_yellow = (img_arr[:, :, :2] > mean_rgb[:2]).all(axis=2)
```

umändern zu

```
where_is_yellow = (img_arr[:, :, :2] > 0.6*mean_rgb[:2]).all(axis=2)
```

Sieht das Ergebnis...

...schon wesentlich besser aus. Durch das Heruntersetzen des benötigten Gelbwertes werden nun zwar auch die Schatten rot gefärbt, aber das kann auch als Feature statt als Bug betrachtet werden...