Einführung in Python II

Inhalt

1 2D-Arrays

1.1 Erzeugung und Zugriff auf Elemente, Zeilen und Spalten

Zur Erzeugung zweidimensionaler Arrays der Größe $M \times N$ gibt es im Modul `numpy` viele Möglichkeiten, z.B.

```
z = np.zeros((M, N))
```

Hierbei bezeichnet M die Anzahl der Zeilen und N die Anzahl der Spalten.
Probieren Sie

```
import numpy as np
matr = np.zeros((4, 3))
print(matr)
```

Weitere nützliche Befehle sind

```
mat1 = np.ones((M, N))          # ueberall 1
mat2 = np.eye(N)                # Einheitsmatrix
mat3 = np.diag(np.arange(N))    # Matrix mit arange(N) auf der Diagonalen
```

Einzelne Elemente spricht man wie folgt an

```
matr[0, 0] = 2.0
matr[3, 0] = 4.0
matr[0, 2] = 1.0
matr[1, 2] = 5.0
print(matr)
```

Einzelne Zeilen oder Spalten einer Matrix erhält man mittels der : Notation

```
print(matr[2, :])       # 3. Zeile
print(matr[:, 1])       # 2. Spalte
```

Hinweis: wie bei eindimensionalen Arrays beginnt die Indexzählung jeweils bei 0.
Zuweisungen an ganze Zeilen oder Spalten sind ebenfalls möglich

```
print(matr)
matr[:, 1] = -2.0      # oder: matr[:, 1] = -2.0 * np.ones(4)
print(matr)
```

1.2 Achsen und Summation

Die Form des Arrays erhält man mittels des Attributes `shape`

```
print(matr.shape)
```

Will man die Summe über alle Elemente von `matr` bilden, so geschieht dies mit der Funktion `np.sum`:

```
print(np.sum(matr))
4.0
```

Ein wichtiges Konzept für mehrdimensionale Arrays ist die Angabe einer Koordinatenachse (`axis`), um z.B. nur über eine Achsenrichtung zu summieren.

```
          [[ 2.  -2.  1.]       |
matr=      [ 0.  -2.  5.]       |
           [ 0.  -2.  0.]     axis=0   (vertikale Achsenrichtung)
           [ 4.  -2.  0.]]      |
                                V
            --  axis=1 -->
          (horizontale Achsenrichtung)


print(np.sum(matr, axis=0))
[ 6.  -8.  6.]


print(np.sum(matr, axis=1))
[ 1.  3.  -2.  2.]
```

1.3 Skalarprodukt und Matrix-Vektor Produkt

Das Skalarprodukt erhält man durch `dot`

```
v1 = np.array([1.0, 2.0, 0.0, 1.0])
v2 = np.array([3.0, 1.0, 5.0, 2.0])
print(np.dot(v1, v2))
```

Die Norm eines Vektors kann man durch

```
print(np.sqrt(np.dot(v1, v1)))
```

berechnen.
Das Matrix-Vektor Produkt erhält man ebenfalls durch `dot`

```
w = np.array([1.0, 2.0, -1.0])
print(np.dot(matr, w))
```

1.4 Konstruktion zweidimensionaler Arrays

Oft braucht man zweidimensionale Arrays mit aufsteigenden Elementen in x- oder in y-Richtung, z.B. zur Berechnung von 2D Funktionen oder für Konturdiagramme (siehe unten). Hierzu kann man die `np.meshgrid` Funktion verwenden:

```
x = np.linspace(0.0, 10.0, 3)
y = np.linspace(0.0, 5.0, 6)
print(x)
print(y)

x2d, y2d = np.meshgrid(x, y)
print(x2d)
print(y2d)
print(x2d**2 + y2d**2)
```

1.5 Warnung

Wie bereits für eindimensionale Arrays illustriert, werden Arrays bei der Zuweisung **nicht** kopiert:

```
x = np.zeros((4, 3))
y = x                   # Zuweisung macht keine Kopie
print(y)
y[2:4, 0] = 1           # veraendere y
print(y)
print(x)                # auch x hat sich geaendert
```

Auch bei der Übergabe an Funktionen werden Arrays **nicht** kopiert:

```
def funktion_mit_nebenwirkung(arr):
    """Element 0 des Arrays ‘arr‘ wird auf 5 gesetzt."""
    arr[0] = 5


x = np.arange(10)
funktion_mit_nebenwirkung(x)
print(x)                # Element 0 von x hat sich geaendert
```

2 Grafik II

`Matplotlib` bietet die Möglichkeit, zweidimensionale Arrays mit dem Befehl `plt.imshow` darzustellen:

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

x = np.linspace(0.0, 10.0, 60)
y = np.linspace(0.0, 10, 101)
x2d, y2d = np.meshgrid(x, y)
f_xy = np.sin(x2d) * np.exp(-y2d**2 / 50)
plt.figure(1)
plt.subplot(111)
plt.imshow(f_xy)
plt.show()
```

Die Darstellung mittels `plt.imshow` hat dabei dieselbe x– und y–Koordinatenabhängigkeit wie bei der Ausgabe mit `print`.

```
matr = np.reshape(np.arange(20), (5, 4))
print(matr)

plt.figure(1)
plt.subplot(121)
plt.imshow(matr)
# Test - wo ist matr[0, 1] ?
matr[0, 1] = 19
print(matr)

plt.subplot(122)
plt.imshow(matr, interpolation='nearest')       # keine Interpolation
plt.show()
```

Der Achsenbereich kann wie folgt angegeben werden

```
plt.imshow(f_xy, extent=[0.0, 1.0, 0.0, 2.0])   # xmin, xmax, ymin, ymax
```

Die Standard-Farbzuordnung verwendet `jet`, welche besser vermieden werden sollte[1]. Die Farbzuordnung kann mit `cmap` gesetzt werden:

```
plt.imshow(f_xy, cmap=plt.cm.winter)
```

Weitere Optionen[2] sind z.B. `plt.cm.summer`, `plt.cm.cool`, `plt.cm.flag`, `plt.cm.gray` und `plt.cm.autumn`. An der Seite eines Achsenbereichs sollte immer mittels `plt.colorbar()` eine Farbskala erzeugt werden. Der angezeigte Bereich der Farbskala wird mittels der Argumente `vmin` und `vmax` von `plt.imshow` beeinflusst. Für weitere Details siehe die Dokumentation von `plt.imshow`.
Um ein Konturdiagramm zu erzeugen, benutzt man die Routine `plt.contour`:

```
plt.contour(x2d, y2d, f_xy, levels=[-1.5, -0.75, 0.0, 0.75, 1.5])
```

Mit dem (optionalen) Parameter `levels` können die Werte für die Höhenlinien festgelegt werden.

[1] Siehe z.B.: `https://jakevdp.github.io/blog/2014/10/16/how-bad-is-your-colormap`.
[2] Siehe auch `http://matplotlib.org/users/colormaps.html`.

3 Komplexe Arrays

Die imaginäre Zahl i wird in Python3 durch `1j` angesprochen:

```
c1 = 1.0 + 1j
c2 = 1.0 + 5j
print(c1 + c2)
print(c1 - c2)
print(c1*c2)
print(c1/c2)
```

Komplexe Arrays werden beispielsweise mittels

```
v1 = np.zeros(5, dtype=np.complex)
print(v1)
v2 = np.ones(5, dtype=np.complex)
print(v2)
v3 = np.arange(5) + 1j*(5-np.arange(5))
print(v3)
```

erzeugt.
Realteil oder Imaginärteil beschafft man sich mittels des Zusatzes `.real` bzw. `.imag`,

```
print(v3.real)
print(v3.imag)
```

Bei der Bildung des Produktes zweier komplexer Vektoren mittels `dot` wird das erste Argument **nicht** konjugiert,

```
print(np.dot(v3, v2))
```

Dies ist mittels des `conjugate` Befehls möglich,

```
print(np.conjugate(v3))
print(np.dot(np.conjugate(v3), v2))
```

4 Weitere Array Befehle (`sort`, `argsort`)

Mehrere Elemente eines Arrays lassen sich wie folgt auswählen

```
a = np.arange(0, 80, 10)
print(a)
ind = [1, 2, 7]
print(a[ind])
```

Analog kann man mehreren Elementen von `a` gleichzeitig einen Wert zuweisen

```
ind = [1, 2, 5]
a[ind] = [55, 33, 22]
print(a)
```

Sortieren kann man mit `sort`,

```
print(a)
print(np.sort(a))
```

Oft braucht man den Befehl `argsort(a)`, der die Reihenfolge der Indizes von `a` angibt, die das Array `a` sortieren

```
ind = np.argsort(a)
print(ind)
```

In Kombination mit dem Zugriff über Indizes kann man dann das ursprüngliche Array sortieren,

```
print(a[ind])
```

Ein derartiges Vorgehen ist insbesondere dann nützlich, wenn ein weiteres Array `b` gemäß der Sortierung von `a` mitsortiert werden soll,

```
b = np.arange(len(a))**2
print(b[ind])
```

Ebenfalls ist es möglich, nur bestimmte Elemente eines Arrays auszuwählen, indem man mit Booleschen Variablen arbeitet. Ein Beispiel:

```
b = np.arange(4)**2
print(b <= 5)
print(b[b<=5])
```

Beachten Sie bitte, dass die Sortierung komplexer Zahlen mathematisch nicht wohldefiniert ist (Es gibt keine Ordnungsrelation in den komplexen Zahlen!). Das Modul `numpy` stellt dafür **eine** Lösung bereit. Betrachten Sie dazu beispielsweise:

```
In [2]: x = np.array([1.0+3.0j, 1.0+2.0j, 2.0+0.1j])

In [3]: print(np.argsort(x))
[1 0 2]
```

5 Speichern und Laden von Arrays

Es gibt verschiedene Möglichkeiten Arrays dauerhaft (z.B. auf Festplatte) in binärer Form oder als Text zu speichern, unter anderem auch komprimiert. Das Speichern in Form von Textdateien sollte bevorzugt werden, da diese menschenlesbar sind und einfacher von anderen Programmen weiterverarbeitet werden können. Für einfache Ein- und Ausgabe kann `np.savetxt` und `np.loadtxt` verwendet werden:

```
data = np.diag(np.linspace(0, 1, 100))
print(data)
np.savetxt("ausgabe.dat", data)
print(np.loadtxt("ausgabe.dat"))
```

Für weitere Optionen siehe die Dokumentation von `np.loadtxt` und `np.savetxt`.
Diese Möglichkeiten stoßen beim Speichern von großen und komplexen Datenstrukturen schnell an ihre Grenzen. Für solche Anwendungen existieren mächtigere Bibliotheken mit wesentlich mehr Möglichkeiten, wie z.B. `h5py` oder `PyTables`, welche auf `HDF5` basieren.