1 Python Scripting für Physiker - Handout W5

Zwei der nützlichsten Module für das wissenschaftliche Arbeiten mit Python sind NumPy und SciPy. NumPy stellt dem Benutzer als wichtigstes Feature leistungsstarke multidimensionale Arrays und Matrizen und dazugehörige Operationen zur Verfügung, während SciPy unter anderem Module zur wissenschaftlichen Analyse und Simulation bietet. Für den Gebrauch von NumPy und SciPy is es empfehlenswert, die jeweiligen Online-Referenzen zu den Paketen anzuschauen und griffbereit zu halten:

- http://docs.scipy.org/doc/numpy/reference/
- http://docs.scipy.org/doc/scipy/reference/

Vorsicht ist geboten, da diese Dokumentation in der Regel den aktuellen Entwicklungsstand von NumPy/SciPy widerspiegelt und nicht zwangsläufig die auf dem Rechner installierte Version. Sollten beim Arbeiten mit NumPy/SciPy Diskrepanzen gegenüber der Referenz auftreten, findet man auf http://docs.scipy.org/doc/ Links zu den Referenzen vorheriger Versionen.

1.1 NumPy Arrays

1.1.1 Initialisierung

Das NumPy Arrays ist das wichtigste Konstrukt in NumPy. NumPy Arrays sind im wesentlichen ähnlich zu den bereits bekannten Listen, jedoch mit einigen Einschränkungen und Vorzügen. Nachteilig im Vergleich zu Listen ist die Einschränkung, dass die Elemente eines NumPy Arrays alle denselben Datentypen haben müssen. Dies wird aber durch die deutlich höhere Zugriffs- und Verarbeitungsgeschwindigkeit bei Arrayoperationen gegenüber Listen wettgemacht. Um Arrays benutzen zu können müssen wir wie in W4 gezeigt erst das Modul *numpy* importieren, also den Namensraum bekanntmachen.

```
>>> import numpy                    # Import des Modules numpy
```

Um jetzt ein NumPy-Array aus z.B. einer Liste von Ganzzahlen zu erzeugen kann man man die Funktion *array()* aus dem Modul *numpy* benutzen:

```
>>> myList = range(5)                       # Liste der Ganzzahlen 0 bis 4
>>> myArr = numpy.array(myList)             # Array der Ganzzahlen 0 bis 4
>>> print myArr, type(myArr)
[0 1 2 3 4] <type 'numpy.ndarray'>
```

Die Konvertierung von Listen mit gemischten Typen schlägt unter Umständen fehl, wenn keine einheitliche Konvertierung der Datentypen möglich ist:

```
>>> myList = [4,'spam',4.5,(4,6)]
>>> myArr = numpy.array(myList)
(...)
ValueError: setting an array element with a sequence
>>> myList = myList[:-1]            # loesche Tupel
>>> myArr = numpy.array(myList)
>>> print myArr
['4' 'spam' '4.5']
```

Im ersten Fall schlug die Konvertierung fehl, da kein gemeinsamer Datentyp für alle Elemente gefunden werden konnte (wegen des Tupels). Im zweiten Fall wurde als gemeinsamer Datentyp von int, string und float der string-Datentyp gewählt und für das Array festgelegt.

Natürlich muss man nicht vorher Tupel oder Listen definieren, um Arrays zu erzeugen, es gibt eine Reihe nützlicher Funktionen zur Erzeugung von Arrays. Ähnlich zu *range()* für Listen gibt es *arange()* zur Erzeugung von auf- oder absteigenden Arrays von Zahlen. Praktischer an *arange()* ist hierbei, dass sich auch Ranges von Float-Zahlen generieren lassen:

```
>>> myArr = numpy.arange(5)
>>> print myArr
[0  1  2  3  4]
>>> myArr = numpy.arange(-1,-0.5,0.1)    # drittes Argument ist Schrittweite
>>> print myArr
[-1.  -0.9  -0.8  -0.7  -0.6]
```

Dazu ähnlich ist die Funktion *linspace()*, die jedoch als drittes Argument die Anzahl an Elementen nimmt, und standardmäßig den Endwert enthlt.

```
>>> myArr = numpy.linspace(10,20,5)      # drittes Argument ist Anzahl Elemente
>>> print myArr
[ 10. ,  12.5,  15. ,  17.5,  20. ]
```

Weitere praktische Funktionen zur Generierung von Arrays sind *zeros(N)*, *ones(N)* und *empty(N)*, die jeweils Arrays aus N zu null, eins oder uninitialisierten Elementen erzeugen.

1.1.2 Elementweise Operationen

Ein wichtiger Vorteil von Arrays gegenber Listen ist es, dass arithmetische und Vergleichs-Operationen elementweise angewandt werden. Somit hat das Array eher Ähnlichkeiten zu einem Vektor als zu einer Liste, bei der dies nur ber Listen oder die noch nicht vorgestellen List Comprehensions möglich ist. Zu beachten ist, dass die elementweisen Funktionen aus dem Namensraum *numpy* statt aus *math* importiert werden müssen, um richtig auf Arrays zu funktionieren:

```
>>> myArr = numpy.arange(5)
>>> print myArr*2
[0  2  4  6  8]
>>> print numpy.sqrt(myArr) # nicht aus math
[ 0.          1.          1.41421356  1.73205081  2.        ]
```

Mathematische Verknüpfungen zweier Arrays werden im Gegensatz zu Listen ebenfalls elementweise abgehandelt:

```
>>> myArr = numpy.arange(5)
>>> myBrr = numpy.arange(0,1,0.2)
>>> print myArr ** myBrr
[ 1.          1.          1.31950791  1.93318204  3.03143313]
>>> myBrr = numpy.arange(0,1,0.4)
>>> print myArr ** myBrr # falsche Anzahl Elemente
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Dies bedeutet aber auch, dass sich Arrays im Gegensatz zu Listen nicht mit dem Additionsoperator vergrößern lassen. Hierzu muss das Array vergrößert werden, dies kann mit den im nächsten Abschnitt vorgestellten Funktionen geschehen.

Auch Vergleichsoperatoren arbeiten elementweise auf Arrays und geben sogenannte Masken zurück (dazu mehr im nächsten Abschnitt):

```
>>> myMask = myArr > 2
>>> print myMask
[False  False  False  True  True]
```

1.1.3 Multidimensionalität und Indizierung

Wie eingangs erwähnt ist das NumPy Array ein multidimensionaler Container. Zur Erzeugung von mehrdimensionalen Arrays gibt es etliche Möglichkeiten, eine einfache ist es z.B. die Funktionalität der von *numpy* bereitgestellten Erzeugungsfunktionen zu nutzen, ein sogenanntes shape-Tupel als Argument zu nehmen:

```
>>> mdArray = numpy.zeros((2,3))          # (2,3) gibt die shape an
>>> print mdArray
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Diese Funktion erzeugt ein 2x3-Array und initialisiert jedes Element zu 0. Das Argument hierbei ist ein Tupel, der jeweils die Anzahl Elemente jeder Dimension angibt. Die Länge des Tupels gibt somit die Dimensionalität des Arrays an. Überprüfen kann man die Form (shape) des Arrays mit dem Attribut *shape* des Arrays, die Dimensionalität mit *ndim*, und die Gesamtzahl an Elementen mit *size*.

```
>>> print mdArr.shape
(2,3)
>>> print mdArr.ndim
2
>>> print mdArr.size
6
```

Die Indizierung von Arrays funktioniert genau wie bei Listen:

```
>>> myArr = numpy.arange(5)
>>> print myArr[2]
2.
```

Genauso ist das Slicen von Arrays möglich:

```
>>> print myArr[1:-1]
[1 2 3]
```

Indizierung von multidimensionale Arrays lässt sich entweder über mehrere Indizierungsoperatorklammern oder ber Komma-getrennte Indizierungsanweisungen in einer Operatorklammer realisieren:

```
>>> mdArr = numpy.arange(6)
>>> mdArr = mdArr.reshape((2,3))
>>> print mdArr
```

```
[[0  1  2]
 [3  4  5]]
>>> print mdArr[1][0]
3
>>> print mdArr[0,2]
2
```

Die Methode *reshape()* konvertiert das Array in die angegebene Form. Eine ähnliche Methode ist *resize()*, die darüber hinaus as Array auch vergrößern kann, wenn die im Tupel angegebene Anzahl Elemente größer als das Ausgangsarray ist.

Natrlich sind auch hierbei in jeder Dimension Slices möglich:

```
>>> print mdArr[0,:] # erste Zeile
[0 1 2]
>>> print mdArr[:,-1] # letzte Spalte
[2 5]
print mdArr[:-1,1:]
[[1, 2]]
```

Ein entscheidender Vorteil von NumPy Arrays ist es, dass zur Indizierung auch Listen/Arrays (allg. Sequenzen) oder Masken (True/False Arrays gleicher Form) benutzt werden können. Im ersten Fall besitzt die Index-Liste weniger Elemente als die Ursprugsliste, und jedes Element gibt an welche Indizes des Ursprungsarrays in das Ausgabearrays übernommen werden.

```
>>> myArr = numpy.arange(10,20)
>>> indArr = numpy.arange(0,10,2) # Indiziere jedes zweite Element
>>> print myArr[indArr]
[10 12 14 16 18]
```

Die bereits oben erwähnten Masken können ebenfalls zur Indizierung genutzt werden, und bieten eine natürliche Schreibweise zur Reduktion von Arrays. Zu beachten ist jedoch, dass die Masken die gleiche Form wie die zu indizierenden Arrays haben müssen. Daraus folgt auch, dass die Masken auf alle Arrays dieser Form angewandt werden können. Dies ist z.B. sehr nützlich zum gleichzeitigen Slicen mehrerer Spalten eines Datensatzes

```
>>> myArr = numpy.arange(10,20)
>>> myBrr = numpy.arange(40,50)
>>> myMask = myArr > 15
>>> print myArr[myMask], myBrr[myMask]
[16 17 18 19] [46 47 48 49]
```

Masken lassen sich ebenfalls einfach über die Multiplikations- und Additionsoperatoren schneiden oder vereinigen:

```
>>> myMask = (myArr > 15) * (myBrr < 49)
>>> print myArr[myMask], myBrr[myMask]
[16 17] [46 47]
```

Masken können auch zur Erzeugung von Indexlisten genutzt werden, indem sie an die Funktion *where()* übergeben werden:

```
>>> myInd = numpy.where(myMask)
>>> print myArr[myInd], myBrr[myInd]
[16 17] [46 47]
```

2 Übungen

Für die Durchführung der Übungen nutzen Sie bitte die Online-Referenzen zu NumPy oder die *help()*-Funktion, um sich mit der Funktionsweise der genannten Funktionen vertraut zu machen

1. Bestimmen Sie wie in Übung 4 mit Hilfe des Monte-Carlo-Integrations Verfahrens den Wert von π. Nutzen Sie hierfür die Möglichkeiten, die NumPy Arrays bieten (elementweise Operationen, Masken).

2. Lesen Sie den Datensatz "data5.dat" mit Hilfe der Routine *loadtxt()* ein und generieren Sie die Häufigkeitsverteilungen der einzelnen Spalten mit Hilfe der Funktion *histogram()*. Stellen Sie die Ergebnisse grafisch dar (Siehe Übung 3).

3. Schreiben sie eine Funktion, die eine Anzahl von K Punkten des Polynoms $P(x) = x^0 + x^1 + ... + x^{N-1} + x^N$ im Wertebereich $x_s < x < x_e$ berechnet. N als Parameter an die Funtion übergeben werden. Hilfreich für die Durchführung dieser Aufgabe ist die Funktion *meshgrid()* (Siehe Übung 2)

```
Beispiel:
>>> x,y = poly(0,10,5,2)   # xs=0; xe=10; K=5; N=2
>>> print x,y
[0,2,4,6,8] , [1,7,21,43,73]
```

4. (Advanced) Die Datei "exercise5_adv.py" enthält eine Routine zur zufälligen Erzeugung von "Spektren". Finden Sie einen Algorithmus, der für erzeugte Spektren die Positionen und Höhen der Peaks findet, und wenn möglich mit einer passenden Funktion fittet.