Institut für Theoretische Physik
Prof. Dr. A. Bäcker
Prof. Dr. R. Ketzmerick

Einführung in Python

1 Programmierumgebung

Diese Einführung soll mit der in den Übungen benutzten Programmiersprache PYTHON vertraut machen. Verwendet wird PYTHON3.
PYTHON ist eine Skriptsprache, die sowohl interaktiv über den Kommandozeileninterpreter als auch durch die Ausführung von Programmen benutzt werden kann. Für die interaktive Verwendung wird IPYTHON empfohlen.

Mittels der Eingabe

```
ipython3
```

in einem Terminalfenster wird IPYTHON gestartet. Es erscheint die Eingabeaufforderung

```
In [1]:
```

2 Grundfunktionen

Alle Funktionen der Programmiersprache können sofort an der Eingabeaufforderung ausprobiert werden.

2.1 Variablenzuweisungen

Einer Variablen kann auf folgende Art ein Wert zugewiesen werden:

```
x = 5
```

Es sind keine vorherigen Vereinbarungen (d.h. Typdeklarationen) notwendig, `x` ist eine ganze Zahl (Typ "int" vom englischen Wort "integer"). Eine weitere Variablenzuweisung ist

```
y = 4.2
```

Hierbei ist `y` eine reelle Zahl (doppelte Genauigkeit). Mittels

```
z = x + y
```

wird `z` die Summe von `x` und `y` zugewiesen. Hierbei ist `z` automatisch eine reelle Zahl (die Summe zweier ganzer Zahlen hingegen ist wieder eine ganze Zahl).
Analoge Rechenoperationen sind Subtraktion, Multiplikation, Division und Potenzierung:

```
-    *    /    **
```

Durch

```
print(z)
```

wird der Wert der Variablen `z` ausgegeben. Der Typ der Variablen `z` kann durch

```
type(z)
```

angezeigt werden.
Wenn Sie bisher PYTHON2 verwendet haben sollten, probieren Sie unbedingt aus, welche Ergebnisse der Divisionsoperator liefert. Berechnen Sie dazu `5/7`, `5.0/7.0`, `5/7.0` und `5.0/7`!

2.2 Relationen

Relationen können direkt mit dem Interpreter ausprobiert werden.
Für die Werte `x = 5` und `y = 4.2` gibt
`x > y` : `True`
`x == y`: `False` aus.
Man beachte den Unterschied zwischen dem vergleichenden `==` und dem zuweisenden `=`.
Weitere Relationen sind `<=`, `>=` und `!=` (ungleich).

2.3 Bedingte Ausführung (if)

Relationen werden zum Aufstellen von Bedingungen benutzt. Eine Bedingung hat die allgemeine Form:

```
1 if i < 5:              # Bedingung
2     print(i)           # Block von Befehlen, die ausgefuehrt werden
3     print(i**i)        # sollen, wenn die Bedingung erfuellt ist.
```

Die Zeile, die den Block einleitet, muss mit einem Doppelpunkt enden. Die auszuführenden Befehle werden in IPython automatisch um 4 Leerzeichen eingerückt. Eine weitere Kennzeichnung erfolgt nicht. Der Block wird durch zweimaliges Drücken der "Enter"-Taste verlassen. Die Raute `#` leitet einen Kommentar ein.
Die Ausführung von Befehlen für den Fall, dass die Bedingung nicht erfüllt ist, lässt sich mittels `else` realisieren (Um die Einrückung für die nachfolgende `else`-Anweisung wieder aufzuheben, drücken Sie in `IPython` viermal "Backspace".):

```
1 if i < 5:                # Bedingung
2     print(i)             # Block von Befehlen, die ausgefuehrt werden
3     print(i**i)          # sollen, wenn die Bedingung erfuellt ist.
4 else:
5     print("i ist >= 5") # ..., wenn die Bedingung nicht erfuellt ist
```

Die Unterscheidung mehrerer Fälle kann man mit `elif` erzielen,

```
1 if i < 5:                # Bedingung
2     print(i)             # Block von Befehlen, die ausgefuehrt werden
3     print(i**i)          # sollen, wenn die Bedingung erfuellt ist.
4 elif i < 10:
5     print(" 5 <= i < 10")
6 elif i < 20:
7     print(" 10 <= i < 20")
8 else:
9     print(" i >= 20")
```

2.4 Schleifen

2.4.1 while-Schleife

Bei einer `while`-Schleife ist die Ausführung der eingerückten Befehle an eine Bedingung geknüpft:

```
k, l = 1, 1              # Anfangswerte fuer k und l
while k < 100:           # Bedingung f. Ausfuehrung des Unterabschnittes
    k, l = k+l, k        # Dies ist der
    print(k, l, 1.0*k/l) # Unterabschnitt

print("Dies wird nicht mehr innerhalb der Schleife ausgegeben!")
```

Welche Zahlen werden hier berechnet? Welches Verhältnis ergibt sich?
Schleifen mittels `while` werden typischerweise dann verwendet, wenn die Anzahl der Durchläufe nicht bekannt ist.

2.4.2 for-Schleife

Schleifen mittels `for` werden verwendet, wenn die Anzahl der Durchläufe von Anfang an bekannt ist, z.B:

```
for i in [1, 3, 7]:
    j = i**2
    print("Das Quadrat von", i, "ist", j)
```

2.5 Programme

Man kann dem Interpreter mehrere Programmzeilen auf einmal übergeben, indem man ein Programm in eine Datei schreibt. Als Editor kann beispielsweise GEANY durch die Ausführung folgenden Befehls in einem Terminalfenster (nicht in IPYTHON, ggf. neues Terminalfenster öffnen) gestartet werden

```
geany filename.py &
```

Für Hinweise auf andere Editoren siehe Seite 15. PYTHON-Programme sollten immer die Endung `.py` haben.
Nehmen Sie als Beispiel obiges Programm zur Bestimmung der Fibonacci-Zahlen und des Goldenen Schnittes. Das mit dem Editor erstellte und gespeicherte Programm wird in IPYTHON durch den Befehl

```
run filename.py
```

ausgeführt.
Alternativ kann man ein PYTHON-Programm auch direkt im Terminalfenster mit dem Befehl

```
python filename.py
```

ausführen.

2.5.1 Funktionen

Um Programme besser zu gliedern oder Wiederholungen von Befehlen zu vermeiden, definiert man sogenannte "Funktionen" (auch "Unterroutine"/"Unterprogramm"/... genannt):

```
def sum_prod(x, y):
    """Berechne Summe und Produkt von x, y."""
    return x + y, x * y
```

Die Befehle der Funktion müssen ebenfalls um 4 Stellen eingerückt werden. Durch """..."""
wird die Dokumentation (der sogenannte "Docstring") für die Funktion festgelegt. Probieren
Sie beispielsweise

```
print(sum_prod(4, 6))
a, b = sum_prod(4, 6)
print(a)
print(b)
```

Mittels `help(sum_prod)` erhalten Sie die Dokumentation zur Routine. Diese wird durch
Drücken der Taste "q" verlassen.
In einem PYTHON-Programm werden Funktionen vor dem Hauptprogramm definiert.

2.5.2 Programmaufbau und Dokumentation

Bei etwas längerem Kode, der eventuell auch wieder verwendet werden soll, ist eine gute Strukturierung eines Programmes hilfreich. Ein weiterer sehr wichtiger Aspekt bei der numerischen Umsetzung physikalischer Probleme ist eine sorgfältige, aussagekräftige Dokumentation.
Beides soll an folgendem Beispiel illustriert werden:

```
"""Kurzbeschreibung des Programms ("Titel")

An dieser Stelle soll eine Beschreibung des Programms,
der Methoden, offener Fragen etc. kommen.

Programm unter:
wwwpub.zih.tu-dresden.de/~baecker/teaching/cp2017/newton_beispiel.py
"""

from math import sqrt


def newton_iteration(zahl, anz_iter=5):
    """Berechne Quadratwurzel von ‘zahl‘ mittels Newton-Iteration.

    Die Newton Iteration
        x_k =  1/2 (x_{k-1} + zahl/x_{k-1})
    (mit x_0=1) konvergiert zur gesuchten Wurzel.

    Der Parameter anz_iter (mit dem Default-Wert 5) bestimmt die
    Anzahl der durchgefuehrten Iterationen.
    """
    x = 1.0                         # Startwert der Newton-Iteration
    for ctr in range(anz_iter):     # Fuehre anz_iter Iterationen durch
        x = 0.5 * (x + zahl/x)      # Newton-Iterationsschritt
    return x


def main():
    """Hauptprogramm. Aufruf fuer verschiedene Parameter."""
    print("Newton Iteration")
    print("sqrt(2)      : %16.14f" % (sqrt(2.0)))
```

```
33      print("5 Iterationen: %16.14f" % (newton_iteration(2.0)))
34      print("3 Iterationen: %16.14f" % (newton_iteration(2.0, 3)))
35      print("2 Iterationen: %16.14f" % (newton_iteration(2.0, 2)))
36
37
38  if __name__ == "__main__":
39      main()
```

Beim Aufruf des Programms mittels `python newton_beispiel.py` erhält man als Ausgabe

```
Newton Iteration
sqrt(2)      : 1.41421356237310
5 Iterationen: 1.41421356237309
3 Iterationen: 1.41421568627451
2 Iterationen: 1.41666666666667
```

Neben der Funktion `newton_iteration` zur Durchführung der Schritte des Newton-Verfahrens wird eine weitere Funktion `main` als Hauptprogramm definiert.
Beim Aufruf des Programms mittels `python3 newton_beispiel.py` (oder in IPython mittels `run newton_beispiel.py`) ist die Bedingung in Zeile 37 erfüllt, so dass die Funktion `main` aufgerufen wird.
Der Sinn dieses Konstrukts liegt darin, dass man die Funktion `newton_iteration` auch in anderen Programmen nutzen kann, ohne dass `main` aufgerufen wird:

```
1  import newton_beispiel
2
3  ergebnis = newton_beispiel.newton_iteration(2.0)
```

2.5.3 Zeilenverlängerung

Um zu verhindern, dass Zeilen länger als 79 Zeichen werden, gibt es verschiedene Möglichkeiten.
Klammern ermöglichen eine automatische Fortführung in der nächsten Zeile:

```
a = 400 * (50*eine_lange_variable + und_noch_eine_zu_lange_variable +
           und_weiter_gehts_in_der_naechsten_zeile )
```

Ebenso kann mittels \ eine Zeile verlängert werden

```
a = 400 * (50*eine_lange_variable + und_noch_eine_lange_variable) + \
           und_weiter_gehts_in_der_naechsten_zeile
```

Hierbei ist zu beachten, dass \ direkt vom Zeilenumbruch gefolgt werden muss, also insbesondere auch keine folgenden Leerzeichen erlaubt sind.

2.6 NumPy-Modul und Arrays

Python stellt einen Satz von Standardbefehlen zur Verfügung, der mit einer Vielzahl zusätzlicher Module erweitert werden kann. Durch

```
import numpy as np
```

werden alle Befehle, die im Modul `numpy` enthalten sind, über die Variable `np` zugänglich gemacht[1]. Auf die einzelnen Funktionen kann dann mittels `np.<befehl>` zugegriffen werden. Das `numpy`-Modul stellt u.a. mathematische Funktionen und Arrays bereit. Laden Sie das `numpy`-Modul für die folgenden Aufgaben zu eindimensionalen Arrays. (Mehrdimensionale Arrays werden später behandelt.) Probieren Sie folgende Befehle aus

```
x = np.arange(10)
print(x)
y = np.zeros(10)
print(y)
z = 2*np.ones(10, dtype=np.int)  # ganzzahlige Werte anlegen
print(z)
z2 = 2.0*np.ones(10, dtype=np.int)
print(z2)
```

Was ist der Unterschied zwischen `z` und `z2`? Beachten Sie was passiert, wenn Sie `z[3] = 7.3` setzen. Hätten Sie das erwartet?

Während `np.zeros` und `np.ones` Fließkommazahlen zurückgeben (oder ganze Zahlen, wenn das Schlüsselwort `dtype=np.int` angegeben wird), liefert `np.arange` nur dann ganze Zahlen, wenn es ausschließlich mit ganzen Zahlen aufgerufen wird.

Elementare Operationen:

Neben der Multiplikation mit Skalaren ist auch die Addition und Division möglich,

```
x = np.arange(10)
print(4*x)
print(4.0*x)
print(x + 1)
print(x/10.0)
print(x - np.pi)
print(1.0*(x < 3.5))
print(1.0*((3 < x)*(x <= 7)))
```

Arrays kann man elementweise addieren, subtrahieren, multiplizieren und auch potenzieren:

```
w = np.arange(1, 20, 2)
x = 2*np.ones(10)
print(x + w)
print(x - w)
print(x*w)
print(x/w)
print(x**2)
print(x**w)
```

Für die oben genannten Operationen ist es wichtig, dass `x` und `w` die gleiche Anzahl von Elementen haben. Probieren Sie

[1] Alternative: am Anfang des Programms `from numpy import *` und dann direkt auf die einzelnen Funktionen zugreifen, z.B.: `linspace(0.0, 1.0, 10)`. Auch wenn dies etwas Tipp-Arbeit erspart, so wird im Rahmen dieses Kurses auf diese Variante verzichtet, da sie — insbesondere für größere Projekte — generell nicht zu empfehlen ist.

```
v1 = np.arange(10)
v2 = np.zeros(9)
print(v1 + v2)                # Fehler
```

Die Anzahl der Elemente eines Arrays (= Arraylänge) erhalten Sie durch die Funktion `len`,

```
print(len(x))
```

Die Summe der Elemente eines Arrays erhält man mittels `np.sum`,

```
print(np.sum(x))
```

Zugriff auf einzelne Elemente und Slicing:

Auf einzelne Elemente eines Arrays kann man mittels

```
print(x[5])
```

zugreifen. Beachten Sie, dass das erste Element durch `x[0]` und das letzte durch `x[len(x)-1]` oder `x[-1]` angesprochen wird.
Teilbereiche eines Arrays können mit der : Notation ("slicing") angesprochen werden:

```
x = np.arange(10)
print(x)
print(x[0:4])
print(x[5:7])
print(x[4:])
print(x[:6])
print(x[1:6:2])
print(x[:6:2])
print(x[::2])
```

Beachten Sie, dass (analog zum `np.arange` Befehl) die obere Grenze **nicht** mit eingeschlossen ist.

To copy or not to copy:

Wichtig ist folgendes Beispiel:

```
k = 10
l = k
print(k, l)
l = 4
print(k, l)

# Aber:
q = np.arange(5)
r = q
print(q, r)
r[2] = -100
print(q, r)
```

Dies ist eine häufige Ursache für Fehler – können Sie sich vorstellen, warum für Arrays das Verhalten anders ist?
Zum Kopieren eines Arrays verwendet man daher beispielsweise[2]

[2] Alternativ kann eine Kopie eines Arrays auch durch eine Operation mit dem Array, z.B. `r = 1*q`, erreicht werden. Beachten Sie dabei mögliche Typkonversionen!

```
q = np.arange(5)
r = q.copy()                # Kopie-Erzeugung mittels der Methode copy
print(q, r)
r[2] = -100
print(q, r)
```

Erzeugung von Arrays:

Ein Array reeller Zahlen (doppelte Genauigkeit) kann mittels

```
a = np.zeros(10)
print(a)
print("Arraytyp", a.dtype.name)     # liefert: Arraytyp float64
```

erzeugt werden.

Arrays mit reellen, linear ansteigenden Einträgen kann man wie folgt erzeugen

```
print(np.arange(0.0, 7.2, 0.8))     # Start, Ende, Inkrement
```

ACHTUNG: Vergleichen Sie mit

```
print(np.arange(0.0, 5.4, 0.6))
```

Was ist hier unterschiedlich? Hätten Sie das erwartet? Ist Python schlecht? Schauen Sie sich dazu die Ausgaben der folgenden Zeilen an.

```
print(repr(9.0*0.8))
print(repr(9.0*0.6))
```

Um bei reellen Arrays sicherzustellen, dass ein Array eine vorgegebene Anzahl von Elementen hat, sollte man **immer** den Befehl `np.linspace` verwenden

```
print(np.linspace(0.0, 5.0, 10))    # Start, Ende, Anzahl
```

Möchte man hierbei den Endpunkt ausschließen, so ist dies mittels

```
print(np.linspace(0.0, 5.0, 10, endpoint=False))
```

möglich.

Weitere Array Operationen:

Neben den oben aufgeführten kann man eine Vielzahl weiterer Operationen auf Arrays anwenden, z.B.:

```
N = 100000
x = np.linspace(0.0, 1.0, N)    # Array mit Werten in [0.0, 1.0]
y = np.cos(x)
z = np.abs(y)                   # Betrag der Elemente
```

Vergleichen Sie die Geschwindigkeit der Berechung von

```
np.sum(np.cos(x))
```

mit

```
summe = 0.0
for i in np.arange(N):
    summe = summe + np.cos(x[i])
print(summe)
```

Dieses Beispiel illustriert zwei wesentliche Aspekte

- Verwendung von Arrays führt zu kürzeren und damit übersichtlicheren Programmen.
- `for`-Schleifen sind deutlich langsamer.

Berechnen Sie mittels einer `for`-Schleife:

$$\sum_{n=1}^{100} \frac{1}{n} \approx 5.1873...$$

Probieren Sie auch:

```
print(np.sum(1.0/np.arange(1, 101)))
```

Hilfe zum NumPy Modul (siehe Seite 14 zu allgemeinen Hinweisen zur Hilfe):

- am Python Prompt: `help("numpy")`, Beenden durch Taste "q"
- am Python Prompt: `help("numpy.linspace")`

3 Graphik mit matplotlib

Ein wesentlicher Aspekt numerischer Untersuchungen in der Physik ist die graphische Darstellung von Daten. Hierfür verwenden wir `matplotlib`[3].
Im Folgenden werden einige grundlegende Beispiele und die wichtigsten Befehle aufgeführt, eine ausführliche Dokumentation findet sich unter `http://matplotlib.org/`.

3.1 Statische Plots

Zum Erzeugen eines statischen Plots werden zunächst die zu zeigenden Daten berechnet und dann der `plot`-Befehl verwendet, welcher die Daten für die Ausgabe vorbereitet. Um die Daten tatsächlich am Bildschirm anzeigen zu lassen, ist am Ende des Programms der `show()`-Befehl nötig. Dieser stellt die Daten dar und ermöglicht das Zoomen, Verschieben und Abspeichern des Plots.
Folgendes Programm stellt die Sinusfunktion und ihre zweite Harmonische graphisch dar:

```
1 import numpy as np
2 import matplotlib.pyplot as plt        # Graphikbefehle
3
4 x = np.linspace(0.0, 2.0*np.pi, 100)   # x-Werte erzeugen
```

[3] Zusatzinformation: `matplotlib` kann verschiedene sogenannte Backends für die graphische Darstellung verwenden. Hierbei ist unter Linux `GtkAgg` aus Geschwindigkeitsgründen zu bevorzugen. Dies kann in der Datei `~/.config/matplotlib/matplotlibrc` mit der Zeile `backend:GtkAgg` eingetragen werden. Für Windows ist das Backend `TkAgg` zu bevorzugen. Sie können Ihre Installation manuell auf `TkAgg` umstellen, indem Sie in der Datei `C:\Anaconda3\Lib\site-packages\matplotlib\mpl-data\matplotlibrc` mit einem Texteditor die Zeile `backend:Qt5Agg:` durch `backend:TkAgg` ersetzen. Das Backend `GtkAgg` (benötigt `pygtk`) kann für flüssigere Animationen sorgen, läuft auf manchen Windows-Rechnern jedoch nicht so stabil wie `TkAgg`.

```
5
6 plt.plot(x, np.sin(x))                    # Werte darstellen
7 plt.plot(x, np.sin(2*x))                  # weitere Funktion hinzufuegen
8 plt.show()                                # graphische Darstellung
```

Durch `import matplotlib.pyplot as plt` werden die Plotbefehle unter `plt.` bereitgestellt.
Der Stil der Kurve läßt sich direkt beim `plot`-Befehl angeben:

```
plt.plot(x, np.sin(x), linestyle='-.', linewidth=3, color='g')
```

Es ist aber auch möglich, die Eigenschaften später anzupassen. Hierzu muss man sich allerdings merken, was bisher geplottet wurde:

```
1  import numpy as np
2  import matplotlib.pyplot as plt          # Graphikbefehle
3
4  x = np.linspace(0.0, 2.0*np.pi, 100)     # x-Werte erzeugen
5
6  sinus = plt.plot(x, np.sin(x))           # plotten - plot(xdata, ydata)
7  sinus2 = plt.plot(x, np.sin(2*x))        # weitere Funktion hinzuplotten
8
9  plt.setp(sinus[0], linewidth=3)          # Linienbreite setzen
10
11 plt.show()
```

Hier eine Auswahl anderer Schlüsselwörter[4]:

```
plt.setp(sinus[0], linestyle='-.')       # Linientyp: Strich-Punkt
plt.setp(sinus[0], color='g')            # gruen
plt.setp(sinus2[0], linestyle='None')    # Linientyp: keine Linie
plt.setp(sinus2[0], marker='s')          # kleine Rechtecke als Marker
plt.setp(sinus2[0], markerfacecolor='r') # rote Marker
plt.setp(sinus2[0], markeredgecolor='k') #  mit schwarzem Rand
plt.setp(sinus2[0], markersize=4)        # Groesse der Marker
plt.setp(sinus2[0], markeredgewidth=2)   # Breite ihres Randes
```

3.1.1 Plotbereich, Überschrift und Achsenbeschriftungen

Folgendes Beispiel zeigt, wie man den Plotbereich festlegt und eine Überschrift und Achsenbeschriftungen anbringt:

```
1  import numpy as np
2  import matplotlib.pyplot as plt          # Graphikbefehle
3
4  plt.title("Ueberschrift")
5  plt.xlabel("x Beschriftung")
6  plt.ylabel("y Beschriftung")
7  x = np.linspace(0.0, 1.0, 100)
```

[4]Wenn Sie diese Zeilen selber ausprobieren möchten, empfehlen wir, ein kleines Programm zu schreiben - interaktiv ist dies umständlicher.

```
plt.plot(x, 2.0*x*x)
plt.axis([0.0, 1.0, 0.0, 2.0])          # Plotbereich [x0, x1, y0, y1]
plt.show()
```

3.1.2 Mehrere Plotfenster/ Plotbereiche

Mehrere Plotfenster lassen sich mit dem `figure`-Befehl wie folgt öffnen:

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

x = np.linspace(0.0, 1.0, 100)
plt.figure(0)                            # neues Fenster 0
plt.plot(x, 2.0*x*x)
plt.figure(1)                            # neues Fenster 1
plt.plot(x, np.sqrt(x))
plt.figure(0)                            # wieder Fenster 0 aktivieren
plt.plot(x, 3.0*x*x)
plt.show()
```

Sollte – wie in den vorangegangenen Beispielen – nur ein Fenster nötig sein, so kann der `figure`-Befehl weggelassen werden. Er wird dann implizit mit dem ersten `plot`-Befehl aufgerufen. Allerdings ist es guter Programmierstil, das Fenster explizit durch `figure` zu erstellen. Mit dem `subplot`-Befehl können mehrere Plotbereiche in einem Fenster erzeugt werden. Hierbei selektiert `subplot(nmk)` in einem Fenster mit $n \times m$ Plotbereichen den k-ten Plotbereich. Die verwendete Syntax ist zunächst etwas gewöhnungsbedürftig[5]

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

x = np.linspace(0.0, 1.0, 100)
plt.figure(0)                   # neues Fenster 0
plt.subplot(423)                # der dritte von 4*2=8 Bereichen
                                # (angeordnet in 4 Zeilen und 2 Spalten)
plt.plot(x, 2.0*x*x)
plt.subplot(426)                # Bereich 6
plt.plot(x, np.sqrt(x))
plt.subplot(423)                # wieder Bereich 3 aktivieren
plt.plot(x, 3.0*x*x)
plt.show()
```

Die Zahlen des Beispiels sind absichtlich etwas ungewöhnlich gewählt, um die Wirkungsweise zu demonstrieren. In der Praxis sind Werte wie `111`, `211`, `121`, `122` für den `subplot`-Befehl üblicher.
Hier verhält es sich im Übrigen ähnlich zum `figure`-Befehl: Wird nur ein Plotbereich benötigt, kann auf den `subplot(111)`-Befehl verzichtet werden, da auch der `subplot`-Befehl (wenn der Programmierer ihn nicht explizit benutzt) implizit von `plot` aufgerufen wird.

[5] Alternativ kann man auch den Befehl `plt.subplots` verwenden, um ein Plotfenster mit mehreren Plotbereichen anzulegen.

3.2 Interaktionen in Plotfenstern

Um mittels der Maus Koordinaten auswählen zu können, verwendet man den `connect`-Befehl:

```
1  """Beispiel zur Interaktion in Plotfenstern.
2
3  Beim Klicken innerhalb des Plotfensters wird ein Punkt gezeichnet.
4  Drueckt man 's' wird eine Sinus-Kurve eingezeichnet.
5  """
6
7  import functools
8  import numpy as np
9  import matplotlib.pyplot as plt
10
11
12 def bei_maus_klick(event, amplitude):
13     """Zeichne Sinus-Kurve ausgehend von Mausposition."""
14     xpos = event.xdata
15     ypos = event.ydata
16
17     t = np.linspace(0, 2*np.pi, 300)
18     x = t + xpos
19     y = amplitude * np.sin(t) + ypos
20     plt.plot(x, y, ls='-', lw=1, c='r')   # Kurve hinzufuegen
21     plt.draw()                            # plotten
22
23
24 def main():
25     """Hauptprogramm"""
26     ampl = 0.8
27
28     plt.figure(0)
29     plt.subplot(111, autoscale_on=False)
30     plt.axis([0, 2*np.pi, -1, 1])
31     plt.xlabel("x")
32     plt.ylabel("y")
33
34     klick_funktion = functools.partial(bei_maus_klick, amplitude=ampl)
35     plt.connect('button_press_event', klick_funktion)
36     plt.show()
37
38
39 if __name__ == "__main__":
40     main()
```

Der `plt.draw()`-Befehl (Zeile 21) dient dazu, dass die mit `plt.plot` angegebene Kurve gezeichnet wird.

Durch das Konstrukt

```
1    klick_funktion = functools.partial(bei_maus_klick, amplitude=ampl)
2    plt.connect('button_press_event', klick_funktion)
```

wird festgelegt, dass im Fall eines Mausklicks die Funktion `bei_maus_klick` aufgerufen wird. Hierbei wird für `bei_maus_klick` das Funktionsargument `amplitude` auf den Wert `ampl` gesetzt.
Über `event` lassen sich weitere Informationen über das Ereignis erhalten:
Z.B. gibt `event.button` die gedrückte Maustaste (1=linke, 2=mittlere, 3=rechte) an und `event.inaxes` hat den Wert `True` wenn der Mausklick innerhalb des (weißen) Plotbereichs stattgefunden hat.

4 Hilfe, Dokumentation

Dieser Abschnitt soll ein paar Hinweise zur Dokumentation in Python geben. Eine übliche Fragestellung ist: "Was macht z.B. die Funktion `plot`?". Hierzu gibt es verschiedene Möglichkeiten, wie z.B.

- interaktive Hilfe am Python Prompt: `help("<Befehl>")`
- Information über ein Objekt: `object?` (nur IPython)

4.1 `help()`

Diese Funktion wird von Python selbst zur Verfügung gestellt. Man benutzt sie, indem man die gesuchte Funktion als Zeichenkette *inklusive des Modulnamens* übergibt.
Beispiel für den `plot`-Befehl: `help("matplotlib.pyplot.plot")` oder Hilfe zum gesamten Plot-Modul mittels `help("matplotlib.pyplot")`. Die Hilfe funktioniert auch für importierte Module oder Funktionen, z.B.: `help(np)` oder `help(np.arange)`.

4.2 IPythons "?" Hilfe

Möchte man auf die Schnelle von einer bereits erzeugten Variablen Details wissen, z.B. ihren Typ, ist der Befehl "?" nützlich. Er funktioniert nur unter IPython. Man schreibt die gewünschte Variable und hängt ein Fragezeichen an. Es werden verschiedene Informationen ausgegeben:

```
In [1]: i = 1       # i ist eine Integervariable

In [2]: i?          # Ausgabe der Information zu i
Type:           int
Base Class:     <type 'int'>
String Form:    1
Docstring:          # es folgt etwas Dokumentation zur Typumwandlung
```

Am wichtigsten sind die Werte "Type" (welchen Typ hat die Variable) und "String Form" (wie sieht die Variable in einen String umgewandelt aus).
Wenn der Dokumentation etwas länger ist, dann kann diese mittels `q` verlassen werden.

4.3 Dokumentation und Hilfe im WWW

- Grundlegendes und Weiterführendes zu `numpy`, `scipy`, ...: `www.scipy.org`
- Weitere Links unter `wwwpub.zih.tu-dresden.de/~baecker/teaching/python`
- Für Fragen zur Vorlesung gibt es eine Mailingliste (siehe Abschnitt 7.2).

5 Editor

Folgende Einstellungen sind für jeden Editor empfohlen:

- Zeilenlänge <= 79 Zeichen
- `TAB` als 4 Leerzeichen einstellen
- Syntax Highlighting anschalten

Auch wenn die Wahl des Texteditors im Kurs freigegeben ist, sollen für den Anfänger hier drei gängige Editoren vorgestellt werden.

5.1 Geany

Geany ist ein Editor mit Funktionen einer Entwickulungsumgebung, der unter Windows und Linux verwendet werden kann.
Geany stellt viele nützliche Features wie automatische Codevervollständigung, Syntaxhervorhebung und Formatierung sowie einen Symbolbrowser zur Verfügung und lässt mit Plug-ins (z.B. Debugger) erweitern.

5.1.1 Einstellungen

Folgende Einstellungen, die für die Übungen empfohlen werden, können wie folgt vorgenommen werden:

- 1. Menü Erstellen –> Kommandos zum Erstellen konfigurieren
  2. Bei Kompilieren: `python3 -m py_compile "%f"` und bei Ausführen: `python3 "%f"` eintragen
- Automatisches Einrücken unter
  Menü Bearbeiten → Einstellungen → Editor → Einrückung
  - Breite: 4
  - Typ: Leerzeichen

5.1.2 Tastaturbefehle

Folgende Tastenkürzel sind nützlich:

- `F5` run
- `STRG+e` kommentieren von Zeile oder ausgewähltem Block
- `STRG+i` rückt den markierten Block ein
- `STRG+u` macht die Einrückung des markierten Blocks rückgängig
- `STRG+b` bewegt den Cursor zur jeweils zugehörigen Klammer

5.2 Spyder

Spyder ist ein Editor, der für Windows und Linux Systeme geeignet ist. Nach dem Öffnen von Spyder (beispielsweise mittels `spyder3` in einem Terminalfenster) gibt es drei Bereiche: auf der linken Seite befindet sich das Eingabefeld, in dem das Programm geschrieben wird, rechts unten befindet sich ein IPython Interpreter und rechts oben ist ein Feld, in dem die Dokumentation zu dem aktuellen Befehl angezeigt wird.

5.2.1 Einstellungen

Folgende Einstellungen, die für die Übungen empfohlen werden, können unter Tools/Preferences vorgenommen werden:

- `TAB` als 4 Leerzeichen:
  - "Editor/Advanced settings/Indentation characters: 4 spaces"
  - "Editor/Advanced settings/Tab always indent"
- Damit Animationen in ausgeführten Skripten funktionieren, muss "Console/External modules/GUI backend: TkAgg" eingestellt werden.
- Zur Vermeidung von Fehlern sollten Skripte nicht im bereits geöffneten IPython-Interpreter, sondern in einem neuem Interpreter ausgeführt werden. Dies wird mittels "Run/Execute in a new dedicated Python3 interpreter" erreicht.
- Das Syntax Highlighting kann unter "Editor/Syntax color scheme" auf einen anderen default gesetzt, oder unter "Syntax coloring" nach Wunsch angepasst werden.

5.2.2 Tastaturbefehle

Auch in Spyder gibt es viele Befehle, die man über die Tastatur ausführen kann, z. B.:

- `STRG`-1 zum Aus-/Einkommentieren des markierten Blocks
- `TAB` bzw. `SHIFT-TAB` zum Einrücken des markierten Blocks
- F5 zum Starten des aktuell offenen Programms

Weitere Kurzbefehle können unter Tools/Preferences/Keyboard shortcuts eingestellt werden.

5.3 Weitere Editoren/Entwicklungsumgebungen

Natürlich gibt es weitere empfehlenswerte Editoren, wie z.B. `emacs`, `eric`, `notepad++`, `eclipse`, `PyCharm`, u.v.m.

6 Thanks

Finally: Vielen Dank an J. Braun, L. Bittrich, J. Brödel, A. Eberlein, S. Günther, N. Hlubek, F. Dressel, J. Kullig, M. Langer, U. Lorenz, S. Löck, C. Löbner, J. Löhnert, M. Michler, M. Richter, T. Rudolf, L. Schilling und A. Schnell!

7 Wichtige Administrativa

7.1 Abgabe der Programme

Wir möchten die abgegebenen Programme testen, Fehler finden, Ausdrucke mit Bemerkungen machen und sie in der nächsten Übung besprechen.
Dies ist nur möglich, wenn einige formale Regeln beachtet werden:

- **Abgabe** der PYTHON-Programme **bis jeweils Montag 8:00 Uhr per E-Mail an**

  `cpuebung@mailbox.tu-dresden.de`

  - Subject: `Blatt<nr>` (d.h. `Blatt1`, `Blatt2`, ...)
  - Der Name des Attachments enthält die Aufgabennummer und den Namen im Format `m_n_vorname_nachname.py` (ohne Sonderzeichen).

    Für Aufgabe 1.1 wäre dies beispielsweise `1_1_vorname_nachname.py`
  - Text: Keiner.

    Die E-Mails werden automatisiert ausgewertet. Alle Bemerkungen müssen daher als Kommentar im Python-Programm stehen (siehe Abschnitt 2.5.2).
  - Es wird automatisiert eine Eingangsbestätigung versendet. Achten Sie daher bitte auf eine gültige Absenderadresse.

- Für die Punktevergabe sind folgende Kriterien entscheidend:
  - **Funktionsfähigkeit** (inkl. Beantwortung von Fragen)
  - ausführliche **Kommentierung**, siehe Abschnitt 2.5.2 und die Musterlösungen
  - **Stil** (Eleganz, Lesbarkeit, ...), insbesondere:
    - optische Gliederung, sinnvolle Variablennamen
    - <= 79 Zeichen pro Zeile und keine Tabulatoren (TABs) wegen des Ausdrucks.

- Im Fall eines Täuschungsversuches gilt die Prüfungsleistung als nicht bestanden.

  Die Feststellung ist auch nach Abgabe der Programmsammlung möglich.

- Für die ersten Programmieraufgaben wird die Musterlösung im Laufe des Montags bereitgestellt unter:

  `wwwpub.zih.tu-dresden.de/~baecker/teaching/cp2017/`

7.2 Mailingliste

Die Mailingliste zum Kurs dient der Diskussion von Fragen zur Vorlesung und den Übungsaufgaben. Ebenfalls dient sie zur Bekanntgabe eventueller Errata oder Hinweise zur Lösung der Übungsaufgaben. Die Adresse der Liste ist:

`phy-cp2017@groups.tu-dresden.de`

Um sich einzutragen, besuchen Sie bitte:

`https://mailman.zih.tu-dresden.de/groups/listinfo/phy-cp2017`