Python – Übung 14

Bei den folgenden Aufgaben wurden bereits Lösungen implementiert, welche aber nicht die pythonische Art repräsentieren. Die wichtigsten Punkte, die beachtet werden sollen, um guten Python-Code zu schreiben, sind:

- Die Lesbarkeit des Codes ist wichtig. Der Code muss so geschrieben werden, dass man es einfach versteht. Python benutzt eine einfache Syntax, die sich an die natürliche englische Sprache anlehnt. Nutzen Sie diese Eigenschaft aus. Zudem sollen die Codestil-Empfehlungen aus PEP8 befolgt werden, um die Leserlichkeit des Codes zu erhöhen.
- Das Rad nicht neu erfinden. Python besitzt viele eingebaute Funktionen und Methoden. Benutzen Sie die bestehenden Funktionen, um Ihren Code schlank und lesbar zu halten.
- `for`-Schleifen sind überbewertet. Eine gute Übung ist es, sich zu fragen, ob man wirklich eine Schleife braucht oder ob sie durch einen idiomatischen Code ersetzt werden kann, z.B. einer List-Comprehension.

Lösen Sie die folgenden Aufgaben auf pythonische Art, indem Sie unter anderem die obigen Tipps befolgen.

a) Gegeben sind $m \times n$ Buchstaben, die in einem String mit m Zeilen und n Spalten in Matrix-Form angeordnet sind. Es wurde die Funktion `transpose(s)` implementiert, welche die Zeichenanordnung im String `s` transponiert, d.h. die Rollen von Zeilen und Spalten vertauscht. Das Resultat wird als String zurückgegeben, z.B.:

```
>>> matrix = "ABC\nDEF\nGHI\nJKL"
>>> transpose(matrix)
ADGJ
BEHK
CFIL
```

Die folgende Funktion:

```
def transpose(s):
    p = s.split("\n")
    m = len(p)
    n = len(p[0])
    o = []
    for i in range(n):
        l = ""
        for j in range(m):
            l += p[j][i]
        o.append(l)
    return "\n".join(o)
```

soll auf pythonische Art implementiert werden. Hinweis: Eine Zeile genügt.

```
def transpose(s):
    return ...
```

☞ Implementieren Sie die obige Funktion auf pythonische Art, indem Sie u.a. die Stringmethode `join()` und die eingebaute Funktion `zip()` sinnvoll nutzen.

b) Gegeben sei eine Textdatei `data.txt`, welche zeilenweise Zahlen enthält, z.B.:

```
8
-1.5e-2
42.66
...
```

Es wurde folgende Lösung implementiert, welche die Zahlen aus der Datei liest und deren Betrag in der Liste `abs_values` (kein NumPy-Array) abspeichert, z.B. `abs_values = [8.0, 0.015, 42.66, ... ]`.

```
abs_values = []
f = open("data.txt")
lines = f.readlines()
f.close()
for x in lines:
    x = float(x)
    if x < 0:
        abs_values.append((-1)*x)
    else:
        abs_values.append(x)
```

☞ Implementieren Sie die obige Funktionalität, indem Sie geeignete NumPy-Funktionen nutzen. Beachten Sie, dass das Resultat eine Liste sein soll, kein NumPy-Array. Hinweis: Eine Zeile genügt.

```
abs_values = ...
```

☞ Implementieren Sie die obige Funktionalität, ohne jegliche NumPy-Funktionen zu benutzen, weil z.B. die `numpy`-Bibliothek nicht zur Verfügung steht. Hinweis für eine 1-Zeilen-Lösung: setzen Sie eine List-Comprehension ein und nutzen Sie die Methode `read_text()` aus der `pathlib`-Bibliothek für das Lesen der Textdatei.

c) Gegeben ist ein Vektor der Form $\boldsymbol{x} = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}^T$, z.B. `x = np.array([3, 4, 5])`. Es soll ein Code implementiert werden, welcher aus diesem Vektor eine Matrix $\boldsymbol{M}$ in folgender Form erstellt:

$$\boldsymbol{M} = \begin{bmatrix} 1 & x_1 & (x_1)^2 \\ 1 & x_2 & (x_2)^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & (x_n)^2 \end{bmatrix}$$

Folgende Lösung wurde implementiert:

```
M = np.zeros((len(x), 3))
for i in range(len(x)):
    M[i, :] = [1, x[i], x[i]**2]
```

☞ Implementieren Sie eine Lösung ohne jegliche `for`-Schleifen einzusetzen. Benutzen Sie geeignete NumPy-Funktionen. Hinweis: Eine Zeile genügt.

```
M = ...
```

d) Gegeben ist eine Liste, welche Email-Adressen enthält, z.B.

```
mailing_list = ["peter@ost.ch", "laura@gmail.com",
                "linda@ost.ch", "carlos@gmx.com",
                "ian@ost.ch"]
```

Diese Liste soll nach allen Email-Adressen, welche auf `"@ost.ch"` lauten, gefiltert werden. Folgende Funktion wurde implementiert:

```
def ost_only(addresses):
    mail_ost = []
    for addr in addresses:
        if "@ost.ch" == addr[-7:]:
            mail_ost.append(addr)
    return mail_ost
```

☞ Implementieren Sie die Funktion, indem Sie eine List-Comprehension einsetzen und eine geeignete `str`-Methode als Bedingung nutzen. Hinweis: Eine Zeile genügt.

```
def ost_only(addresses):
    return ...
```

e) Es soll die Funktion `is_hex(s)` implementiert werden, welche `True` zurückgibt, falls der String `s` eine hexadezimale Zahl (z.B. `"0xF3"`, `"0XF3"`, `"aa55"`) darstellt, `False` sonst. Die folgende Funktion wurde implementiert:

```
def is_hex(s):
    if s.startswith("0x") or s.startswith("0X"):
        s = s[2:]
    for character in s:
        if character not in "0123456789ABCDEFabcdef":
            break
    else:
        return True
    return False
```

Easier to ask for forgiveness than permission (EAFP), auf Deutsch: leichter um Vergebung zu bitten, als um Erlaubnis. Dieser geläufige Python-Programmierstil setzt die Existenz von validen Elementen voraus und fängt Ausnahmen ab, wenn die Voraussetzung nicht erfüllt wurde. Dies führt zu einem allgemein sauberen und prägnanten Stil, welcher viele try- und except-Anweisungen benutzt. Diese Technik hebt sich vom *Look before you leap* (LBYL)-Stil ab, der in vielen anderen Sprachen wie beispielsweise C geläufig ist, wo explizit auf Vorbedingungen getestet wird und sich durch das Vorhandensein vieler `if`-Anweisungen auszeichnet.

☞ Gehen Sie davon aus, dass der String normalerweise eine gültige Hex-Zahl beinhaltet und versuchen Sie mittels einer geeigneten eingebauten Funktion den String in eine Ganzzahl umzuwandeln. Fangen Sie die entsprechende Exception im Fehlerfall ab und geben Sie in diesem Fall ein `False` zurück.

f) Wie in Abb. 1 dargestellt, sei ein Punkt $\vec{P}$ und eine rechteckige Fläche im dreidimensionalen Raum gegeben. Die Fläche liegt in der xy-Ebene, ist $a \times b$ gross und wird in $N \times N$ kleine vektorielle Flächenelemente $\Delta\vec{S}_n$ unterteilt. Es soll die folgende Grösse

$$S = \sum_n \left\| \frac{\vec{R}_n}{\left\|\vec{R}_n\right\|} \cdot \Delta\vec{S}_n \right\| \tag{1}$$

berechnet werden, wobei $\vec{R}_n$ der Distanzvektor zwischen dem Punkt $\vec{P}$ und die Mitte der Teilflächen $\Delta\vec{S}_n$ ist.

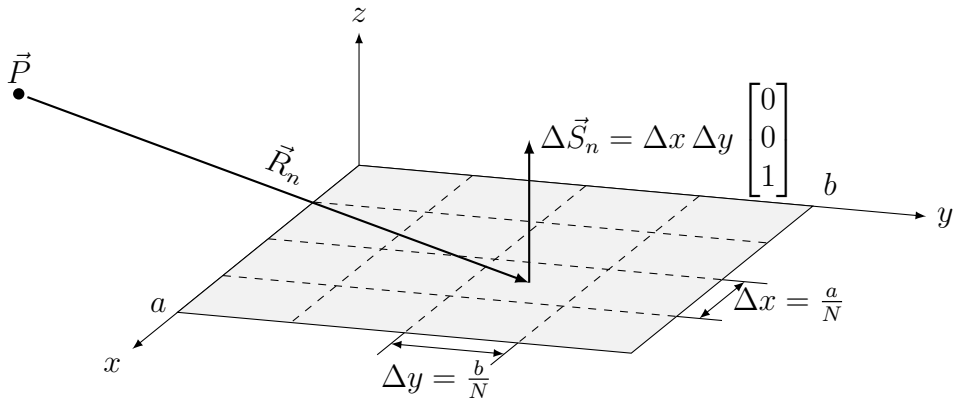

Abbildung 1: Der Punkt $\vec{P}$ und die Fläche in der xy-Ebene.

Die folgende Funktion wurde implementiert:

```
def summation(a, b, p, N):
    p = np.asarray(p)
    dx = a/N
    dy = b/N
    x = np.linspace(0, a, N, endpoint=False) + dx/2
    y = np.linspace(0, b, N, endpoint=False) + dy/2
    z = 0

    dS = dx*dy*np.array([0, 0, 1])
    S = 0
    for xi in range(N):
        for yi in range(N):
            R = np.array([x[xi], y[yi], z]) - p
            S += np.abs(np.inner(R/np.linalg.norm(R), dS))
    return S
```

aber deren Ausführungszeit dauert viel zu lange, wie dieser Test zeigt:

```
>>> import time
>>> a = 3; b = 4; p = [1.5, 2, 100]; N = 1000
>>> tic = time.time()
>>> summation(a=a, b=b, p=p, N=N)
>>> toc = time.time()
>>> print(f"elapsed time: {toc - tic:.3f} s")
elapsed time: 9.078 s
```

Die allermeisten NumPy-Funktionen (wie auch `np.inner()` und `np.linalg.norm()`) unterstützen mehrdimensionale Arrays, um mehrere Vektoren zu gruppieren (Stack). Dadurch wird die Iteration über die einzelnen Vektoren innerhalb der NumPy-Funktion durchgeführt und man erspart sich so den relativ grossen Overhead der Schleifen und Funktionsaufrufe. Bei einigen Funktionen müssen die Vektorenwerte entlang der letzten Array-Dimension angeordnet sein, wie dies bei `np.inner()` der Fall ist. Bei anderen Funktionen (z.B. `np.linalg.norm()`) kann man die Dimension mit dem `axis`-Argument angeben.

☞ Implementieren Sie die Funktion ohne jegliche `for`-Schleifen einzusetzen. Bauen Sie ein mehrdimensionales Array `R`, welches alle Vektoren $\vec{R}_n$ beinhaltet. Benutzen Sie dafür die Funktionen `np.meshgrid()` und `np.stack()`. Wenden Sie dann die NumPy-Funktionen auf das gesamte Array `R` an.
Hinweis: Die Ausführungszeit sollte ca. 100 Mal kürzer sein.