Programmieraufgaben rund um Matrizen

Aufgaben zu LE2 (+ LE3 + LE5)

Diese Aufgaben sollen helfen, etwas praktische Erfahrung mit dem imperativen Programmieren zu sammeln – konkret mit dem Umgang mit Variablen, Schleifen, Listen, *Dictionaries* und *Comprehensions*. Je nach Gliederung der Lösung in Funktionen bzw. Prozeduren lassen sich damit auch Themen von LE3 explorieren. Testdaten werden in Dateien zur Verfügung gestellt (LE5).

Teil des Programmierens ist nicht nur das Aufschreiben von Python-Anweisungen, sondern es gehört auch dazu, eine Fragestellung zu verstehen, einen Plan für die nötigen Berechnungen zu machen und das fertige Programm daraufhin zu untersuchen, ob es richtig funktioniert.

Der Sinn der folgenden Aufgaben liegt daher darin, selbst ein Programm zu planen, aufzuschreiben und auszuprobieren. Sicher lassen sich auch fertige Programme finden, die man einfach abschreiben kann, aber dabei lernt man eben nichts.

Die Aufgaben sind nicht als «Übungen» gedacht, von Dingen, die Ihr eigentlich können solltet, sondern als «Lernaufgaben», die Euch anregen sollen, Dinge auszuprobieren und Fragen aufzuwerfen, woraus Ihr etwas lernen könnt. Also verzweifelt nicht, wenn nicht alles sofort klar ist.

Wenn Ideen fehlen, Lösungen nicht funktionieren, oder nicht klar ist, ob sich etwas verbessern liesse: Nicht einfach irgendeine fertige Lösung anschauen, sondern mit dem eigenen Lösungsansatz bzw. der individuell spezifischen Frage gezielt das Verständnis erweitern. Fragen stellen kann man:

- im Space von gpr auf DS-Spaces
- mir, als Fachexperten im Büro oder per E-Mail oder via Webex oder in der Sprechstunde etc.
- Mitstudierenden
- allen Menschen, von denen man denkt, dass sie vielleicht helfen könnten (Onkel, Tante, Grosseltern, Kindern, Nachbarn, …)

Aufgabe 1 – Matrix als Liste von Listen

Bevor man mit Matrizen rechnen kann, braucht man einen Plan, wie sie im Computer repräsentiert werden sollen. Eine offensichtliche und direkte Übersetzung von der Darstellung auf Papier in ein Speichermodell ist eine *Liste von* Zeilen (oder von Spalten) ebenfalls repräsentiert als *Listen*.

Betrachten wir beispielsweise die Matrix $\begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 4 \end{bmatrix}$. Diese kann als Liste von Zeilen repräsentiert werden:

```
matrix = [[1, 0, 3], [0, 2, 4]]
```

oder – quasi transponiert – als Liste von Spalten:

```
matrixT = [[1, 0], [0, 2], [3, 4]]
```

Eine beliebige *Liste von Listen* mit Zahlen braucht aber nicht in jedem Fall als Matrix interpretierbar zu sein. Das geht nur, wenn die Liste selbst nicht leer ist und die Listen in der Liste alle dieselbe Länge grösser als 0 haben.

Programmiere eine Funktion *well_formed_matrix(m)*, welche die genannten Eigenschaften prüft:

```
>>> well_formed_matrix([])
False
>>> well_formed_matrix([[], []])
False
>>> well_formed_matrix([[1], [1,2]])
False
>>> well_formed_matrix([[0,0], [1,2]])
True
```

Aufgabe 2 – Matrix als Liste von Listen transponieren

Programmiere eine Funktion *transpose(m)*, die eine Matrix als Liste von Listen transponiert. (Falls *m* keine *wohlgeformte* Matrix im Sinne von Aufgabe 1 ist, soll ein *ValueError* ausgelöst werden.)

```
>>> transpose([[1, 0, 3], [0, 2, 4]])
[[1, 0], [0, 2], [3, 4]]
>>> transpose([[1, 0], [0, 2], [3, 4]])
[[1, 0, 3], [0, 2, 4]]
```

Aufgabe 3 – Matrix flach in einer Liste

Eine andere Möglichkeit Matrizen zu speichern, ist alle Zeilen direkt hintereinander in einer einzigen Liste abzulegen. Technisch greift das Programm dann auf ein Element nur mit einem einzigen Index zu, der zuvor aus Zeilen- und Spaltennummer des Matrixelements berechnet werden muss.

Nehmen wir an, die Zeilenlänge – also die Anzahl der Spalten – sei *N*. Dann finden sich die Elemente der Zeile mit Index 0 gerade an den Listpositionen *0 bis N-1*, die der Zeile mit Index 1 an den Listpositionen *N bis 2N-1* und so weiter.

Damit das funktioniert, müssen ausser der Liste von Werten in der Matrix auch die Dimensionen gespeichert werden (oder zumindest die Anzahl der Spalten, denn die Anzahl der Zeilen könnte man berechnen als Länge der Liste geteilt durch die Anzahl der Spalten.) Dafür eignet sich beispielsweise ein Tupel der folgenden Form:

(*Anzahl Zeilen*, *Anzahl Spalten*, *Liste der Werte*)

Auch wenn das Tupel als solches unveränderbar ist, können trotzdem die Werte in der Matrix geändert werden, denn das Tupel enthält ja nur eine Referenz auf die Liste als Ganzes – und diese ist selbst ja veränderbar.

Programmiere eine Funktion *flatten(m)*, die eine Matrix *m* als *Liste von Listen* in eine *flache Matrix* der oben beschrieben Form umwandelt:

```
>>> flatten([[1, 0, 3], [0, 2, 4]])
(2, 3, [1, 0, 3, 0, 2, 4])
>>> flatten([[1, 0], [0, 2], [3, 4]])
(3, 2, [1, 0, 0, 2, 3, 4])
```

Zusatz-Aufgabe für Interessierte: Man kann einfach Tupel verwenden und regelt dann quasi «per Konvention» wie die Werte mit Index 0 bis 2 zu interpretieren sind. Eine interessante Alternative bieten *named tuples* aus dem *collections*-Modul.[1] Hier besteht die Gelegenheit, das einmal auszuprobieren.

Aufgabe 4 – Transponier-Funktion erweitern

Um zu unterscheiden, in welcher Datenstruktur eine Matrix gespeichert wurde, kann die Funktion `type(m)` der Sprache Python helfen, wie folgende Aufrufbeispiele illustrieren:

```
>>> m_2d = [[1, 0, 3], [0, 2, 4]]
>>> m_flat = flatten(m_2d)
>>> type(m_2d)
<class 'list'>
>>> type(m_flat)
<class 'tuple'>
>>> type(m_flat) == tuple
True
>>> type(m_flat) == list
False
```

Falls *named tuples* verwendet werden, muss anstelle von `tuple` im Typ-Test der entsprechende Name benutzt werden:

```
>>> m_flat = flatten1(m_2d)
>>> type(m_flat)
<class '__main__.Matrix'>
>>> type(m_flat) == Matrix
True
>>> type(m_flat) == tuple
False
```

Dieses Unterscheidungsmerkmal lässt sich benutzen, um am Anfang der Funktion *transpose(m)* aus Aufgabe 2 zu unterscheiden, mit welcher Daten-Struktur die als Argument angegebene Matrix gespeichert ist. Nach einer entsprechenden Fallunterscheidung kann die Matrix dann entsprechend interpretiert und transponiert werden.

Erweitere die Funktion *transpose(m)* aus Aufgabe 2 so, dass sie auch flach in einer Liste (das heisst eigentlich eben als Tupel mit u.a. einer Liste) gespeicherte Matrizen (wie in Aufgabe 3 erzeugt) transponieren kann. Das Ergebnis soll das gleiche Daten-Format haben, wie das jeweilige Argument *m*.

[1] Die Dokumentation zu «named tuples»: https://docs.python.org/3/library/collections.html#collections.namedtuple

```
>>> m_2d = [[1, 0, 3], [0, 2, 4]]
>>> transpose(m_2d)
[[1, 0], [0, 2], [3, 4]]
>>> transpose(flatten(m_2d))
(3, 2, [1, 0, 0, 2, 3, 4])
```

Aufgabe 5 – Matrix als Dictionary

Anstatt die Struktur einer Matrix auf das Speicher-Layout abzubilden, was Listen letztlich tun, könnte man eine Matrix auch als Abbildung eines Tupels von Zeilen- und Spaltennummer auf einen Wert begreifen. Als Datenstruktur für solch eine Abbildung wäre dann ein *Dictionary* denkbar, bei dem Tupel mit Zeilen- und Spaltennummern als Schlüssel verwendet werden.

Eine solche Art der Datenspeicherung ist vor allem dann interessant, wenn es viele Elemente gibt, die den Wert 0 haben und deswegen gar nicht gespeichert werden müssen. Unsere Beispielmatrix vom Anfang $\begin{bmatrix}1 & 0 & 3\\0 & 2 & 4\end{bmatrix}$ in einem Dictionary gespeichert, ergäbe *{(0, 0): 1, (0, 2): 3, (1, 1): 2, (1, 2): 4}*.

Wie bei der Speicherung in einer flachen Liste ist es auch hier vielleicht sinnvoll, zusätzlich die Dimensionen der Matrix zu speichern, und eine entsprechende Tupel-Struktur zu verwenden.

Programmiere eine Funktion *as_dict(m)*, die eine als *Liste von Listen* (und allenfalls auch eine als *flache Liste*) gespeicherte Matrix in eine Matrix konvertiert, die in einem *Dictionary* gespeichert ist:

```
>>> as_dict([[1, 0, 3], [0, 2, 4]])
(2, 3, {(0, 0): 1, (0, 2): 3, (1, 1): 2, (1, 2): 4})
>>> as_dict([[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])
(4, 4, {})
>>> as_dict([[5]])
(1, 1, {(0, 0): 5})
```

Aufgabe 6 – Transponier-Funktion nochmals erweitern

Erweitere die Funktion *transpose(m)* aus den Aufgabe 2 und 4 so, dass sie auch für mit einem Dictionary gespeicherte Matrizen funktioniert:

```
>>> m_2d = [[1, 0, 3], [0, 2, 4]]
>>> transpose(m_2d)
[[1, 0], [0, 2], [3, 4]]
>>> transpose(flatten(m_2d))
(3, 2, [1, 0, 0, 2, 3, 4])
>>> transpose(as_dict(m_2d))
(3, 2, {(0, 0): 1, (2, 0): 3, (1, 1): 2, (2, 1): 4})
```

Zur Unterscheidung der beiden Tupel-basierten Datenformate (*flache Liste* und *Dictionary*) kann der Typ der dritten Komponente im Tupel dienen:

```
>>> type(flatten2(m_2d)[2])
<class 'list'>
>>> type(as_dict(m_2d)[2])
<class 'dict'>
```

Aufgabe 7 – Addition von Matrizen

Die Addition von Matrizen ist komponentenweise definiert. Voraussetzung ist, dass zwei zu addierende Matrizen dieselben Dimensionen (Anzahl Zeilen und Anzahl Spalten) haben. Für Matrizen M_1 und M_2 mit gleicher Anzahl Zeilen und gleicher Anzahl Spalten gilt: $M_S = M_1 + M_2$, wobei $M_S[i,j] = M_1[i,j] + M_2[i,j]$.

Programmiere eine Funktion *mat_add(m1, m2)* die für die drei zuvor diskutierten Matrizen-Datenstrukturen zwei Matrizen addiert, falls die Anzahl der Zeilen und Spalten übereinstimmt. Nehmen wir zur Vereinfachung an, dass beide Matrizen in der gleichen Art gespeichert seien.

Stimmen die verwendeten Datenstrukturen oder die Dimensionen der Matrizen *m1* und *m2* nicht überein, oder ist eine der Matrizen nicht wohlgeformt, soll ein *ValueError* ausgelöst werden:

```
>>> mat_add([[1, 0, 3], [0, 2, 4]], [[1, 0, 3], [0, 2, 4]])
[[2, 0, 6], [0, 4, 8]]
>>> mat_add([[2]], [[3]])
[[5]]
```

```
>>> mat_add([[1, 0, 3], [0, 2, 4]], [[3]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in mat_add
ValueError
>>> mat_add((2, 3, [1, 0, 3, 0, 2, 4]), (2, 3, [1, 0, 3, 0, 2, 4]))
(2, 3, [2, 0, 6, 0, 4, 8])
>>> mat_add((2, 3, {(0, 0): 1, (0, 2): 3, (1, 1): 2, (1, 2): 4}),
... (2, 3, {(0, 0): 1, (0, 2): 3, (1, 1): 2, (1, 2): 4}))
(2, 3, {(0, 0): 2, (0, 2): 6, (1, 1): 4, (1, 2): 8})
>>> mat_add((1,1,{(0,0):1}), (1,1,{(0,0):-1}))
(1, 1, {})
```

Aufgabe 8 – Multiplikation von Matrizen

Das Produkt $M_p = M_1 \cdot M_2$ zweier Matrizen M_1 und M_2 kann berechnet werden, falls die Anzahl der Spalten in M_1 mit der Anzahl der Zeilen in M_2 übereinstimmt. Diese Anzahl sei *n*. M_p hat dann so viele Zeilen wie M_1 und so viele Spalten wie M_2. Für die Werte von M_p gilt:

$$M_p[i,j] = \sum_{k=0}^{n} M_1[i,k] \cdot M_2[k,j]$$

Programmiere eine Funktion *mat_mul(m1, m2)*, die für die verschiedenen Matrix-Datenstrukturen eine solche Matrix-Multiplikation implementiert. Nehmen wir zur Vereinfachung an, dass beide Matrizen in der gleichen Art gespeichert seien.

Stimmen die verwendeten Datenstrukturen oder die Dimensionen der Matrizen *m1* und *m2* nicht überein, oder ist eine der Matrizen nicht wohlgeformt, soll ein *ValueError* ausgelöst werden:

```
>>> mat_mul([[1, 0, 3], [0, 2, 4]], [[1, 0], [0, 2], [3, 4]])
[[10, 12], [12, 20]]
>>> mat_mul([[1, 0, 3], [0, 2, 4]], [[1, 0, 3], [0, 2, 4]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in mat_mul
ValueError
>>> mat_mul((2, 3, [1, 0, 3, 0, 2, 4]), (3, 2, [1, 0, 0, 2, 3, 4]))
(2, 2, [10, 12, 12, 20])
>>> mat_mul((2, 3, {(0, 0): 1, (0, 2): 3, (1, 1): 2, (1, 2): 4}),
... (3, 2, {(0, 0): 1, (1, 1): 2, (2, 0): 3, (2, 1): 4})
(2, 2, {(0, 0): 10, (0, 1): 12, (1, 0): 12, (1, 1): 20})
```

Aufgabe 9 – Tests mit grossen Matrizen und Zeitmessungen

Auf *ds-spaces* finden sich bei diesen Aufgaben auch einige Dateien mit unterschiedlich grossen quadratischen Matrizen von Zufallszahlen. In der ersten Zeile jeder Datei steht die Anzahl der Zeilen und Spalten der Matrix. Die weiteren Zeilen enthalten die Matrizen selbst: Pro Matrix-Zeile eine Text-Zeile. Die Zahlen sind jeweils durch ein Leerzeichen getrennt.

Programmiere eine Funktion *read_matrix(file_name)*, die eine solche Datei lesen und in eines der diskutierten Datenformate speichern kann.

Verwende die unterschiedlich grossen Datenmengen um die Laufzeit der verschiedenen programmierten Matrix-Operationen (Transponieren, Addieren, Multiplizieren) zu untersuchen. Um wie viel werden die Operationen jeweils langsamer, wenn man eine Matrix mit 10-mal so vielen Zeilen und Spalten wählt?