Name, Vorname:	Matrikelnr.:
Unterschrift	Raum, Sitzplatz:

Einführung in die wissenschaftliche Programmierung

16.02.2018, Wintersemester 2017/18

Allgemeine Hinweise zur Klausur

- **Bearbeitungszeit**: 60 Minuten.
- **Hilfsmittel**: Ein Hilfsblatt **(wird in der Klausur ausgeteilt)**.
- **Hinweise**:
  - Verwenden Sie für Programmieraufgaben die Programmiersprache `Python`!
  - Trennen Sie unter keinen Umständen die Heftung der Blätter auf!
  - Falls der Platz zur Bearbeitung einer Teilaufgabe nicht ausreicht, markieren Sie dies bitte und benutzen die Rückseite des vorhergehenden Blattes!
  - Beschriften Sie den Kopf **dieses** Blattes mit Ihrem vollständigen Namen und Ihrer Matrikelnummer. **Unterschreiben** Sie im dafür vorgesehenen Kästchen.
  - Beschriften Sie den Kopf **jedes anderen** Blattes mit Ihrem Kürzel und Ihrer Matrikelnummer für den Fall, dass sich die Heftung löst.
- **Punkteverteilung**:
  - Für die 4 Klausuraufgaben werden insgesamt ca. 42 Punkte vergeben.
  - Die ungefähre Punkteverteilung innerhalb einer Aufgabe ist als Orientierungshilfe jeweils zu Beginn der Aufgabe angegeben. Es handelt sich **nicht** unbedingt um die endgültige Punkteverteilung.

Bemerkungen der Aufsicht/Korrigierenden

Aufgabe:	1	2	3	4	Σ
Korrektur					

1 Funktionen: Pascalsches Dreieck (≈ 3 + 2.5 + 3.5 = 9 Punkte)

Im Pascalschen Dreieck ist jede Zahl die Summe der beiden Zahlen, die oberhalb von ihr stehen:

$$\forall n \in \mathbb{N} : P_{n,0} = 1 \text{ und } P_{n,n} = 1 \tag{1}$$

$$\forall n \in \mathbb{N}, k = 1, 2, \ldots n : P_{n+1,k} = P_{n,k-1} + P_{n,k} \tag{2}$$

Zum Beispiel: $P_{4,2} = 6 = P_{3,1} + P_{3,2}$

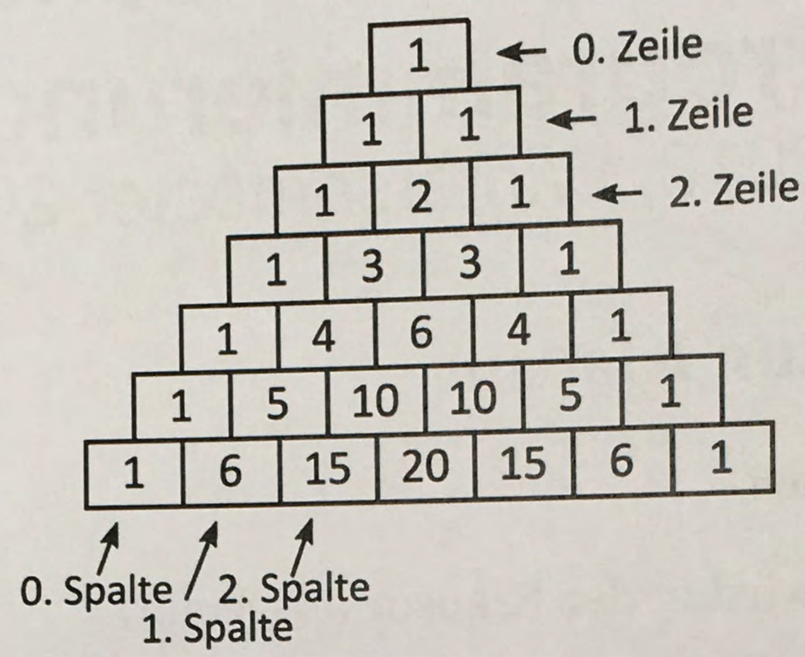

Abbildung 1: Pascalsches Dreieck.

(a) Nutzen Sie Gleichungen (1) und (2) und schreiben damit eine Funktion `computeChild(parentLine, i)`, die als Argument `parentLine` eine Zeile des Pascalschen Dreiecks und als Argument `i` eine Zahl $0 \leq$ `i` $\leq$ länge(`parentline`) übergeben werden. Der Rückgabewert soll die Zahl in der Zeile unter `parentLine` in der `i`'ten Spalte sein.

Zum Beispiel:

- `computeChild([1,3,3,1], 2)` → `3 + 3` → `6`
- `computeChild([1,2,1], 0)` → `1`
- `computeChild([1,2,1], 3)` → `1`

```
def computeChild(parentLine, i):
    if i == 0 or i == len(parentLine):
        return 1
    return parentLine[i-1] + parentLine[i]
```

- 0.5 for len(parentLine)
- 1 for cases 0 and len(parentLine)
- 1 for default case
- 0.5 for efficient if (either the "or", or the lack of else)

(b) Verwenden Sie `computeChild(...)`, um eine Funktion `buildNextLine(parentLine)` zu schreiben. Die Funktion bekommt eine Zeile `parentLine` des Pascalschen Dreieck als Argument und gibt die nächste Zeile zurück.

Zum Beispiel: `buildNextLine([1,4,6,4,1])` → `[1,5,10,10,5,1]`

```
def buildNextLine(parentLine):
    return [computeChild(parentLine,i) for i in range(len(parentLine)+1)]
```

```
def buildNextLine(parentLine):
    nz = []
    for i in range(len(parentLine)+1):
        nz.append(computeChild(parentLine,i))
    return nz
```

- 0.5 for correct range
- 0.5 for correct call to computeChild
- 0.5 for loop
- 1 for empty list + append or compact allocation

(c) Das Pascalsche Dreieck ist mit dem Binomialkoeffizienten verknüpft: $P_{n,k} = \binom{n}{k}$. Schreiben Sie mithilfe von Funktion `buildNextLine(...)`, eine Funktion `binomial(n,k)`, die $\binom{n}{k}$ mit dem Pascalschen Dreieck berechnet.

```
def binomial(n,k):
    if k < 0 or k > n: # BONUS
        return 0       # BONUS
    l = [1] # 0-th Zeile
    for i in range(n):
        l = buildNextLine(l)
    return l[k]
```

- 1 for correct initialization
- 0.5 for loop
- 1 call
- 1 return
- BONUS 1 for checking $0 \leq k \leq n$ and returning 0 if not

2 Objektorientierte Programmierung: Möbel (≈ 5 + 2 + 2 + 2 = 11 Punkte)

Die Möbelhäuser XXXLatz und Sägmühler lassen ein gemeinsames Programm zur Verwaltung von Möbelangeboten entwickeln.

(a) Schreiben Sie eine Klasse `Moebel`, die zwei Objektvariablen `preis` und `rabatt` Werte aus Parametern im Konstruktor zuweist. Ferner soll die Klasse eine Methode `existiertRabatt()` enthalten, die genau dann einen wahren Wert (`True`) zurückliefert, wenn der Rabatt einen positiven Wert hat, und ansonsten `False`. Schließlich soll die Klasse eine Methode `berechneFinalenPreis()` implementieren, in der der endgültige Preis abzüglich des Rabatts (angegeben in %) berechnet und zurückgegeben wird.

```
class Moebel:                                          0.5
  def __init__(self, preis, rabatt):                   1.5
    self.preis  = preis                                0.5
    self.rabatt = rabatt
  def existiertRabatt(self):                           0.5
    if (self.rabatt > 0):                              0.5
      return True                                      0.5
    else:
      return False
  def berechneFinalenPreis(self):
    return self.preis*(100-self.rabatt)/100            1.0
```

Bewertung:

- Keyword class (0.5), Konstruktor Syntax ok (1.5), Objektvariablen ok (0.5)
- Fkt. `existiertRabatt()`: Funktionsdef. ok (0.5), if-Bed. (0.5), return value ok (0.5)
- Fkt. `berechneFinalenPreis()`: Formel korrekt berechnet + zurückgegeben (1.0)

Total: 5 Punkte

(b) Wie lautet der Code, um ein Möbelstück mit einem Preis von 100 (Euro) und einem Rabatt von 19 (%) zu instanziieren und den Gesamtpreis auf die Kommandozeile auszugeben?

```
myMoebel = Moebel(100, 19)                             1.0
print "Preis mit Rabatt ist: ", \                     0.5
myMoebel.berechneFinalenPreis()                        0.5
```

Bewertung:

- Konstruktoraufruf (1.0), print mit Text (0.5), Fkt.-Aufruf ok (0.5)

Total: 2 Punkte

(c) Implementieren Sie eine Klasse `Schrank`, die von `Moebel` erbt und als Parameter im Konstruktor zusätzlich `anzahlFuesse` bekommt. Der Rabatt für `Schrank` soll initial Null sein. Außerdem erhält die Klasse eine Objektvariable `istHaengeschrank`, die nur dann den Wert `True` speichern soll, wenn es keine Füße für den Schrank gibt.

```
class Schrank(Moebel):                                          0.5
  def __init__(self, preis, anzahlFuesse):                      1.0
    Moebel.__init__(self,preis,0)
    self.anzahlFuesse = anzahlFuesse                            0.5
    self.istHaengeschrank = False                               0.5
    if (anzahlFuesse==0):
      self.istHaengeschrank = True
```

Bewertung:

- Klasse: Vererb. (0.5)
- Basis-Konstruktor: existent + Null ok (1.0)
- Flag Haenge initial (0.5), set True (0.5)

Total: 2 Punkte

(d) Schreiben Sie eine weitere Kindklasse `Stuhl`, die von `Moebel` erbt und die Existenz eines Polsters als Booleschen Wert in dem Parameter `polster` im Konstruktor erhält, um den Wert als **private** Objektvariable zu speichern. Außerdem soll die Klasse eine Klassenvariable `counter` besitzen, die die Anzahl der erzeugten Objekte der Klasse `Stuhl` enthält. Implementieren Sie ferner eine Methode `getAnzahlStuehle()`, die den Wert dieser Klassenvariable zurückgibt.

```
class Stuhl(Moebel):                                            0.5
  counter = 0
  def __init__(self, preis, polster):                           0.5
    Stuhl.counter += 1
    Moebel.__init__(self,preis,0)                               0.5
    self.__polsterExistent = polster                            0.5
  @staticmethod                                                 0.5
  def getAnzahlStuehle():                                       0.5
    return Stuhl.counter
```

Bewertung:

- Klassenvariable (0.5), Inkrementierung counter (0.5)
- Syntax private Objektvariable (0.5)
- return ok (0.5)
- BONUS: Keyword staticmethod (0.5)

Total: 2 Punkte

3 Reguläre Ausdrücke & Dictionaries: Kfz-Kennzeichen (3 + 2 + 3 = 11 Punkte)

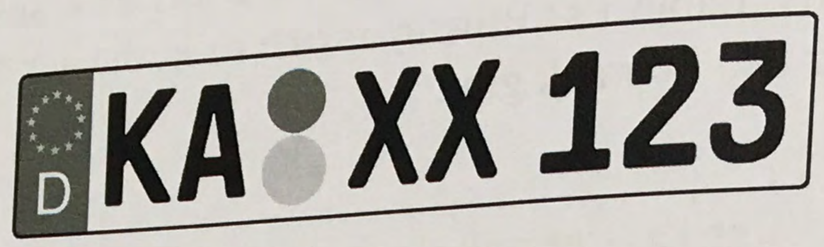

Abbildung 2: Kfz-Kennzeichen `KA-XX123`

Ein Kfz-Kennzeichen besteht aus 4 Teilen:

- das Unterscheidungszeichen (Stadt): 1 bis 3 Großbuchstaben (für diese Aufgabe ohne Umlaute),
- ein Bindestrich: '-'
- 1 oder 2 Großbuchstaben, ohne Umlaut
- 1 bis 4 Ziffern ohne führende Null

Zum Beispiel ist "`KA-XX123`" ein gültiges Kfz-Kennzeichen: "`KA`" + "`-`" + "`XX`" + "`123`"

(a) Schreiben Sie mithilfe von regulären Ausdrücken eine Funktion `validate(s)`, der als Argument `s` eine String übergeben wird. Der Rückgabewert soll `True` sein, wenn `s` ein gültiges Kfz-Kennzeichen ist, ansonsten `False`.

Zum Beispiel:

- `validate('KA-XX123')` → `True`
- `validate('ka-xx123')` → `False`
- `validate('NotValid')` → `False`
- `validate('lorem KA-XX123 ipsum')` → `False`

```
import re
def validate(s):
    if re.match('[A-Z]{1,3}-[A-Z]{1,2}[1-9][0-9]{0,3}$', s):
        return True
    else:
        return False
```

- 0.5 for import re
- 1.5 for approximate correctness
- 0.5 for using match or search with ˆ and $ at the end
- 0.5 for testing not 0 at the start of the number part
- BONUS 0.5 for the if/else as regex doesn't return True/False but Object/None

(b) Schreiben Sie eine Funktion `getCityId(s)`, der als Argument `s` eine gültiges Kfz-Kennzeichen (String) übergeben wird. Der Rückgabewert soll das Unterscheidungszeichen sein.

Zum Beispiel: `getCityId('KA-XX123')` → `'KA'`

```
def getCityId(s):
    return re.match('([A-Z]{1,3})', s).group(1)

def getCityId(s):
    i = 0
    while s[i] != '-':
        i += 1
    return s[:i]
```

- REGEX: 1 for regex with catch + 1 for return group1
- LOOP: 1 for loop with catch + 1 for return slice (or accumulate on an variable initialized as empty string and return it)

(c) Schreiben Sie (mithilfe der beiden vorherigen Funktionen) eine Funktion `matchToCity(ls,d)`, die eine Liste von Strings `ls` und ein Dictionary `d` (welches einem Unterscheidungszeichen eine Stadt zuordnet, siehe Beispiel) als Argument übergeben bekommt. Die Funktion `matchToCity(ls,d)` soll ein Dictionary zurückgeben, die ein bestimmtes (gültiges) Kennzeichen, zu ihrem Unterscheidungzeichen zuordnet (siehe Beispiel). Alle ungültigen Kennzeichen, und diejenigen, dessen Unterscheidungszeichen nicht in `d` gelistet ist, sollen ignoriert werden.

Zum Beispiel:

```
ls = ['M-A12', 'BN-U42', 'K-XX638', 'notValid', 'BN-ZZ999']
d  = {'M':'Muenchen', 'BN':'Bonn'}
print matchToCity(ls,d)

→ {'BN-ZZ999':'Bonn', 'BN-U42':'Bonn', 'M-AB123':'Muenchen'}
```

```
def matchToCity(ls,d):
    r = {}
    for s in ls:
        if validate(s):
            c = getCityId(s)
            if c in d:
                r[s] = d[c]
    return r
```

- 0.5 for initialize empty dict
- 0.5 for loop over list

- 0.5 for test if valid
- 0.5 for call to previous function
- 0.5 for test if key exists
- 0.5 for correct allocation (can also use update)

(d) Schreiben Sie eine Funktion `invert_dictionary(d)`, die ein dictionary `d` (das Resultat der vorherigen Aufgabe) als Argument bekommt, und ebenfalls ein Dictionary zurückgibt. Dieses Rückgabe-Dictionary soll Städte mit einer Liste aller ihr zugehörigen Kennzeichen verbinden.

Zum Beispiel:

```
d = {'BN-ZZ999':'Bonn', 'BN-U42':'Bonn', 'M-AB123':'Muenchen'}
print invert_dictionary(d)

→ {'Bonn':['BN-ZZ999', 'BN-U42'], 'Muenchen':['M-AB123']}
```

```
def TODO_FINDNAME_invert(d):
    r = {}
    for key, value in d.iteritems():
    # or for key in d: | value = d[key]
        if value in r:
            r[value].append(key)
        else:
            r[value] = [key]
    return r
```

- 0.5 for initialize empty dict
- 1 for loop over key and value or loop over key and getting value
- 1 for correct test if value already in result
- 0.5 for append

4 Numpy & Scipy: Coulomb-Potential mit Matrizen (3 + 2 + 2 + 2 + 2 = 11 Punkte)

Im Nachfolgenden wollen wir ein System von positiven Punktladungen ($+e$) in einem zwei-dimensionalen Koordinatensystems modellieren. Das Coulomb-Potential für eine punktförmige Ladung X am Ort Y ist (in Gauss-Einheiten) gegeben durch

$$\Phi(X,Y) = \frac{e}{||X-Y||} = \frac{e}{\texttt{distance(X,Y)}}, \tag{3}$$

wobei e den Wert `e = 1.602E-19` hat und $||X-Y||$ die euklidische Norm (=geometrische Distanz von X nach Y) darstellt.

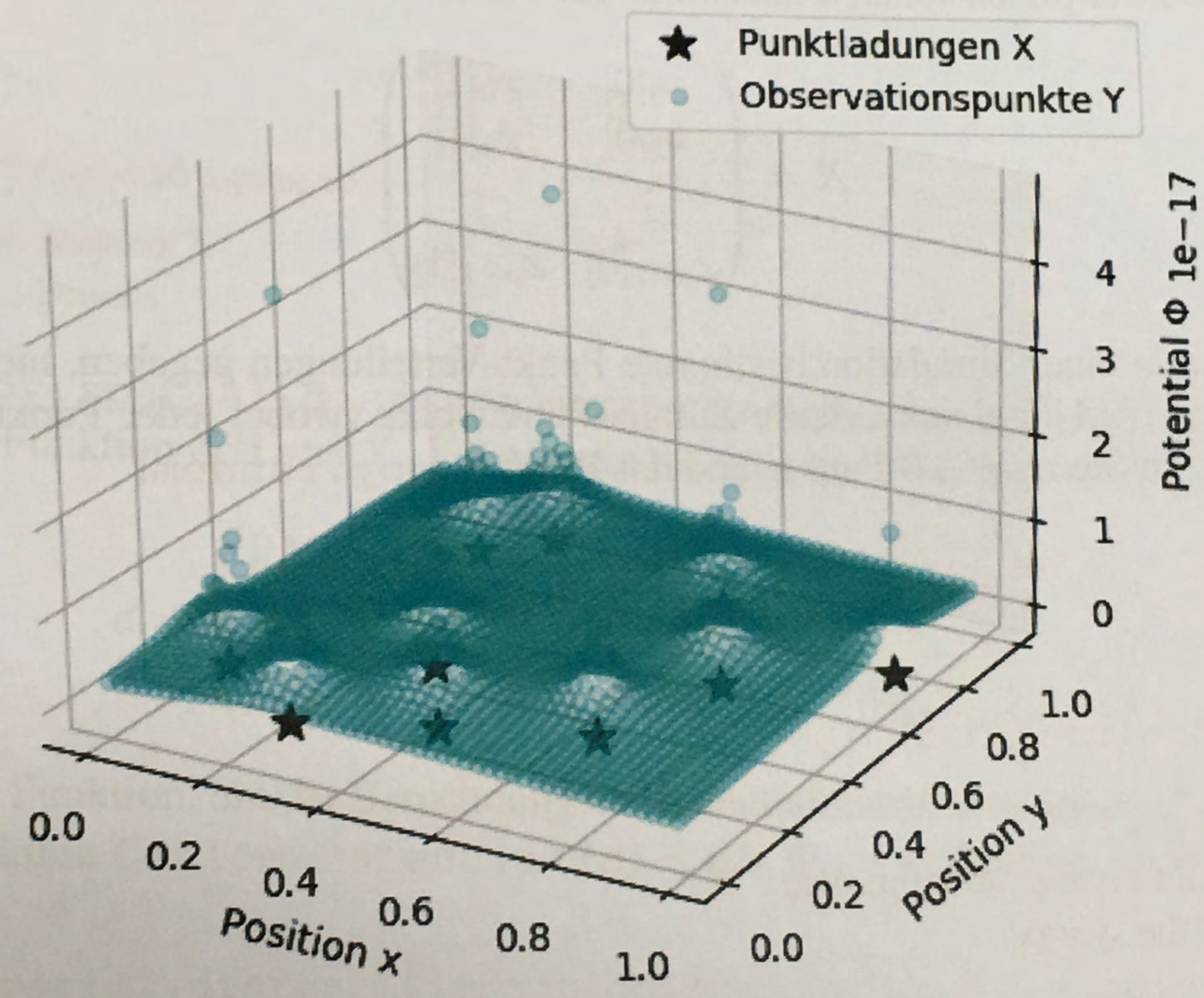

Abbildung 3: In dieser Aufgabe erstellen wir Punktladungen X und Punkte Y, und berechnen d Coulomb-Potential Φ an Punkten Y (resultierend aus Punktladungen an Position X).

Importieren Sie benötigte Pakete explizit!

(a) Schreiben Sie eine Funktion `CoulombPotential(d)`, die als Input eine Distanz `d` zwischen einem Quell- und einem Observations-Punkt bekommt, und als Output das Coulomb-Potential zurückgibt.
Geben Sie Acht, dass sie nicht (numerisch) durch Null teilen! Geben Sie in diesem Fall als Potential `float('Inf')` zurück.

```
def CoulombPotential(d):
  E = 1.602e-19
  if (d > 1e-16):
    return E/d
  return float('Inf')
```

- 1 for edge case
- 1 for defining E
- 1 for correct return

(b) Erstellen sie eine zwei-dimensionale Punktwolke der Form

$$X = \begin{pmatrix} x_0[0] & x_0[1] \\ x_1[0] & x_1[1] \\ \vdots & \vdots \\ x_{n-1}[0] & x_{n-1}[1] \end{pmatrix}.$$

Oft bekommt man in einer Simulation bestimmte Punkt-Verteilungen gegeben, hier genügt uns eine im Einheitsquadrat ($[0,1]^2$) gleichverteilte Zufalls-Punktwolke (wobei jeder Punkt die Punktladung $+e$ darstellt). Nutzen Sie `n = 10` Punkte und das Paket `numpy.random`!

```
import numpy
N = 10
d = 2
X = numpy.random.rand(N,d)
```

- 1 for correct call to numpy.random.rand
- 0.5 for order of the syntax
- 0.5 for numpy import

(c) Um den Potentialverlauf zu analysieren, können Messpunkte Y hilfreich sein (siehe Abbildung 4). Dazu wollen wir den Problemraum equidistant diskretisieren. Schreiben Sie Python Code um Zielpunkte Y zu erstellen, in jeder Raumrichtung $m = 20$ Stück, verteilt auf das Einheitsquadrat ($[0,1]^2$).

Also,

$$Y = \begin{pmatrix} 0 & 0 \\ 0 & 0.05 \\ \vdots & \vdots \\ 0 & 0.95 \\ 0.05 & 0 \\ 0.05 & 0.05 \\ \vdots & \vdots \\ 0.95 & 0.95 \end{pmatrix}.$$

```
m=20
Y=numpy.zeros((m*m,2))
for i in range(m):
  for j in range(m):
    Y[i+m*j,0]=1/m*i
    Y[i+m*j,1]=1/m*j
```

oder mit numpy.arange und meshgrid

- 1 for correct shape of Y
- 1 for correct bounds

(d) Das Paket `scipy.spatial.distance` bietet einige Funktionalitäten um räumliche Distanzen zu berechnen. Die Funktion `cdist(X,Y)` gibt ein `numpy` Array zurück, mit Einträgen

$$d = \begin{pmatrix} ||X_0 - Y_0|| & ||X_1 - Y_0|| & \dots & ||X_{n-1} - Y_0|| \\ \vdots & \vdots & \vdots & \vdots \\ ||X_0, Y_{n-1}|| & ||X_1, Y_{n-1}|| & \dots & ||X_{n-1}, Y_{n-1}|| \end{pmatrix}.$$

Nutzen sie diese Funktion, um die Berechnung der Potentialmatrix Φ zu implementieren, berechenbar mithilfe der Funktion `CoulombPotential(d)` , d.h. $\Phi_{ij} = e/\texttt{distance}(X_j, Y_i)$.

```
from scipy.spatial.distance import cdist

Phi=Coulomb(cdist(X,Y))
```

- 1 for correct call to cdist
- 1 for correct calling of coulomb on all elelments of distance array

(e) Die Potentialmatrix beschreibt an Position $\Phi_{i,j}$ das Potential an Punkt Y_i resultierend aus Ladung X_j. Um das Gesamtpotential (resultierend von allen Ladungen X) an Position Y_i zu bekommen, können

wir eine Zeilensumme bilden. Alternativ ist das in Matrix-Operationen äquivalent mit

$$\texttt{Gesamtpotential} = \Phi * \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}.$$

Nutzen Sie dazu entweder `numpy.dot` oder alternativ `numpy.sum`!

```
Phi_ges=np.dot(Phi,np.ones((m*m,1)))
```

oder:

```
Phi_ges=np.sum(Phi,axis=1)
```

- 1 for correct call to respective functions
- 1 for correct result