**Stefan Kurtz** **Universität Hamburg**
**Emanuel Ehmki** **Zentrum für Bioinformatik**

# Programmierung für Naturwissenschaften 1
# Wintersemester 2019/2020
# Übungen zur Vorlesung: Ausgabe am 08.01.2020

...und immer wieder am Mittwoch verfügbar:
`https://feedback.informatik.uni-hamburg.de/PfN1/wise2019-2020`

**Aufgabe 10.1** (2 Punkte) Aus Zeitgründen können wir in der Vorlesung das Thema „Anwendung der in Python verfügbaren Sortiermethoden“ (siehe Folien Seite 325–332) nicht behandeln. Dieser Abschnitt wird daher in dieser Aufgabe behandelt. Dazu müssen müssen sich einige Studierende vor der Übung anhand der Vorlesungsfolien auf dieses Thema vorbereiten und das erworbene Wissen in Kleingruppen in den ersten 20-30 Minuten am Anfang der Übung an die anderen Studiererenden weitergeben.

Die Kleingruppen setzen sich wie folgt zusammen:

1. Jegminat, Breiholz, Franke, Hauschild, Grosse
2. Schuett, Stahl, Paulsen, Flotow, Jakobi
3. Ehlers, Kaemmle, Molkentin, Schenk
4. Podolskiy, Jochens, Loewenberg, Myronovych
5. Gubernator, Kniep, Dao, Music, Gruber-Roet
6. Moeller, Kuhn, Carlsen, Leege
7. Biegemann, Harkov, Breker, David
8. Tang, Quante, Lehmann, Scheele
9. Eckmann, Plesch, Gruetzmacher, LFranke
10. Liessmann, Froechtling, Scheu, Paffenholz
11. Kaether, Witte, Pluemer, Lohmann
12. Radtke, Ebbing, Lindemann, Block
13. Klemm, Rahlf, Fender

Die Namen der Studierenden, die sich auf das Thema vorbereiten müssen, sind jeweils am Beginn jeder Zeile aufgeführt. Falls jemand von diesen Studierenden nicht zur Übung erscheint, verteilen sich die übrigen Mitglieder der Kleingruppe auf die anderen Gruppen.

Nach der Übung dokumentiert jede Kleingruppe in einer E-mail an `kurtz@zbh.uni-hamburg.de` das Vorgehen bei der Erarbeitung des Themas und ggf. noch bestehende Verständnisfragen oder Hinweise zu Unklarheiten in den Folien. Willkommen sind natürlich auch Bemerkungen zur Lehrform selbst. Dabei soll nicht der Inhalt der Folien repliziert werden. Die E-mail soll die folgenden Eigenschaften haben:

- abgesendet bis zum Abgabetermin der entsprechenden Übung,
- maximal 15 Zeilen mit maximal 80 Zeichen pro Zeile,
- Angabe der Nachnamen aller Mitglieder der Kleingruppe, die teilgenommen haben (alle genannten Personen erhalten die zwei Punkte).

Die E-mail soll von einer/einem Studierenden erstellt werden, die/der sich nicht auf das Thema vorbereitet hat.

**Aufgabe 10.2** (3 Punkte) In dieser Aufgabe geht es noch einmal um die vollständige Zerlegung einer positiven ganzen Zahl in Summanden, wie in Aufgabe 8.1. Hier ist die entsprechende Musterlösung mit der Funktion `split_number`, die eine lokale rekursive Funktion `split_number_rec` aufruft.

```
def split_number(number,terms_of_sum):
  def split_number_rec(terms_of_sum,best_split,remain,terms_idx,l):
    if remain == 0:
      quality = quality_function(l)
      if best_split[0] is None or quality < best_split[1]:
        best_split[0] = l
        best_split[1] = quality
    else:
      for idx, this_num in enumerate(terms_of_sum[terms_idx:]):
        if this_num > remain:
          break
        new_l = l.copy()
        new_l.append(this_num)
        split_number_rec(terms_of_sum,best_split,remain - this_num,idx,new_l)
  best_split = [None,None]
  split_number_rec(terms_of_sum,best_split,number,0,list())
  return best_split
```

Ihre Aufgabe ist es nun, aus der rekursiven Lösung eine iterative Lösung zu entwickeln, also eine Lösung, in der es keine rekursiven Aufrufe, weder direkt noch indirekt, gibt. Dazu sollen Sie zwei Funktionen implementieren.

- Die Funktion `split_number_enumerate(number,terms_of_sum)` implementiert einen Generator, der die additiven Zerlegungen der positiven ganzen Zahl `number` bzgl. der möglichen Summanden in der sortierten Liste `terms_of_sum` aufzählt, und zwar in lexikographischer Reihenfolge. Die Funktion hat also keine `return`-Anweisung, sondern verwendet `yield`. Zur Speicherung der zu lösenden Teilaufgaben verwenden Sie einen Stack, d.h. eine Liste, an die Element am Ende mit `append()` angehängt und mit `pop()` entfernt werden.
- Die Funktion `split_number_itrv(number,terms_of_sum)` benutzt den Generator zum Aufzählen der additiven Zerlegungen von `number` bzgl. `terms_of_sum` und liefert durch eine `return`-Anweisung die Zerlegung mit dem kleinsten quality-Wert. Dieser wird durch die Funktion `quality_function` bestimmt, die Sie aus Ihrer vorherigen Lösung von Aufgabe 8.1 wiederverwenden können. Falls es mehrere Zerlegungen mit dem gleichen minimalen quality-Wert gibt, dann soll die lexikographisch kleinste Zerlegung mit minimalem quality-Wert geliefert werden.

In den Materialien finden Sie ein Hauptprogramm in der Datei `splitnumber_mn.py` sowie Testfälle mit den erwarteten Ergebnissen. Durch `make test` verifizieren Sie, dass Ihr Programm korrekt funktioniert.

**Aufgabe 10.3** (5 Punkte) In dieser Aufgabe geht es um die Entwicklung von Klassen zur Speicherung und Verarbeitung von Moleküldaten.

Bei der computerbasierten Verarbeitung von Molekülen wird oft das mol2-Format verwendet. Eine Datei im mol2-Format kann mehrere Molekül-Einträge enthalten. Jeder Eintrag beginnt dabei mit der Header-Zeile `@<TRIPOS>MOLECULE`, gefolgt von einer Zeile mit dem Namen des Moleküls. Danach findet man mindestens drei Zeilen mit Informationen zum Molekül, die aber für diese Aufgabe nicht wichtig sind. Die erste Zeile enthält die Anzahl der Atome, Bindungen, Unterstrukturen, Features und Sets, wobei nur die Anzahl der Atome Pflicht ist. Die folgenden Zeilen enthalten Informationen zur Klassifizierung des Moleküls und seiner Ladungen.

Der mit `@<TRIPOS>Atom` beginnende Abschnitt listet die Atome auf, die zum Molekül gehören.

Diese sind wichtig für diese Aufgabe. Ein Atom-Eintrag besteht aus mindestens 6 Werten: einer ID, einem Namen, den Koordinaten $(x, y, z)$ und dem Atomtyp. Es folgen weitere optionale Werte.

Ein weiterer Abschnitt nach `@<TRIPOS>Bond` listet die Bindungen zwischen Atomen im Molekül auf. Ein Eintrag enthält die ID der Bindung, zwei IDs Atom-ID$_1$ und Atom-ID$_2$ der gebundenen Atome und die Art der Bindung (`1` = einfach, `2` = zweifach, `ar` = aromatisch).

Hier ein Beispiel eines mol2-Eintrags für ein Molekül mit dem Namen `ADE`:

```
@<TRIPOS>MOLECULE
ADE
   10    11     1     1     0
SMALL
USER_CHARGES

@<TRIPOS>ATOM
      1 N9          61.9022   91.9485    4.2480 N.2       1 ADE1        0.0000
      2 C8          61.2132   92.8472    3.6764 C.2       1 ADE1        0.0000
      3 N7          60.4414   93.4649    4.5274 N.pl3     1 ADE1        0.0000
      4 C5          60.6732   92.8975    5.7357 C.ar      1 ADE1        0.0000
      5 C6          60.1367   93.1516    6.9991 C.ar      1 ADE1        0.0000
      6 N6          59.2047   94.1057    7.2285 N.pl3     1 ADE1        0.0000
      7 N1          60.6045   92.3797    8.0079 N.ar      1 ADE1        0.0000
      8 C2          61.5377   91.4151    7.8135 C.ar      1 ADE1        0.0000
      9 N3          62.0630   91.1619    6.5931 N.ar      1 ADE1        0.0000
     10 C4          61.6277   91.9049    5.5602 C.ar      1 ADE1        0.0000

@<TRIPOS>BOND
     1    1    2 2
     2    1   10 1
     3    2    3 1
     4    3    4 1
     5    4    5 ar
     6    4   10 ar
     7    5    6 1
     8    5    7 ar
     9    7    8 ar
    10    8    9 ar
    11    9   10 ar
```

In den Materialien finden Sie eine Datei `molecule_template.py`. Bitte benennen Sie diese in `molecule.py` um. Sie sollen das in der Datei vorhandene Gerüst der Klassen `Molecule`, `Atom` und `Bond` so vervollständigen, dass das Programm `mol2iter.py` mol2-Dateien verarbeitet und die oben beschriebene Information im gleichen Format (abgesehen von der Anzahl der Leerzeichen) wieder ausgibt. Die Methoden, die mit einem Kommentar der Form `required only for` versehen sind, brauchen Sie nicht zu implementieren.

Im Material finden Sie eine mol2-Datei, die erwartete Ausgabe sowie die beiden genannten Python-Dateien. `mol2iter.py` enthält bereits eine vollständige Funktion `mol2Iterator` zum Einlesen einer mol2-Datei, die Sie nicht verändern dürfen. In den Zeilen

```
for molecule_name, atom_list, bond_list in mol2Iterator(mol2file):
  molecule_list.append(Molecule(molecule_name,atom_list,bond_list))
```

wird ein mol2-Eintrag aus der angegebenen Datei gelesen und der Name des Moleküls sowie die Liste der Atome und die Liste der Bindungen zurückgeliefert. Jedes Element aus `atom_list` ist

selbst wieder eine Liste mit mindestens 6 Werten, nämlich den Werten einer Atomzeile. Jedes Element aus `bond_list` ist selbst wieder eine Liste mit mindestens 4 Werten, nämlich den Werten einer Bondzeile.

Die Ausgabe der Moleküle erfolgt in den folgenden Zeilen:

```
for molecule in molecule_list:
  print('{}'.format(molecule))
```

Die Klasse `Molecule` soll den Namen eines Moleküls sowie die Liste von Atomen und die Liste von Bindungen speichern. Die Atome und Bindungen sind jeweils Instanzen der Klasse `Atom` bzw. `Bond`. Die `__init__`-Methode der Klasse erhält dazu (nach dem Parameter `self`) den Molekülnamen sowie die Listen aller Atome und Bindungen des Moleküls (siehe oben) und muss diese in entsprechenden Instanzvariablen `self._molecule_name`, `self._atom_list`, `self._bond_list` speichern. Die letzten beiden Instanzvariablen sind Listen.

Die `__str__`-Methode der Klasse `Molecule` soll das Molekül Format (siehe oben) als String (inklusive der verschiedenen Headerzeilen der Form `@<TRIPOS>...`) zurückliefern.

Neben der Klasse `Molecule` müssen noch die Klassen `Atom` und `Bond` implementiert werden, um die Werte für einzelne Atome des Moleküls sowie für Bindungen dieser Atome zu speichern.

Für ein Atom muss es Instanzvariablen für eine Atom-ID, einen Namen, drei Koordinaten als Fließkommawerte, einen Atomtyp sowie eine Liste weiterer optionaler Werte geben. Entsprechend hat die Methode `__init__` der Klasse `Atom` nach `self` noch 7 weitere Parameter und entsprechende Instanzvariablen. Die `__str__`-Methode soll eine Liste der Zeilen, die die Atome beschreiben, in einem String zurückliefern.

Für eine Bindung aus der Klasse `Bond` muss es Instanzvariablen für die ID der Bindung, die Atom-IDs der an der Bindung beteiligten Atome sowie den Bindungstyp und optionale Angaben geben. Die optionalen Werte einer Zeile werden als Liste übergeben. Entsprechend hat die `__init__`-Methode der Klasse nach `self` noch 5 Parameter, und es gibt mindestens 5 Instanz-Variablen in der Klasse. Die `__str__`-Methode der Klasse soll eine Liste der Zeilen, die die Bindungen beschreiben, in einem String zurückliefern.

Beachten Sie, dass die `__str__`-Methoden der drei zu implementierenden Klassen keine Ausgabe-Funktionen wie `print` oder `write` aufrufen, sondern jeweils genau einen String zurückliefern, so dass man z.B. ein Molekül durch eine Anweisung `print('{}'.format(molecule))` formatiert ausgeben kann. Der genannte String enthält das Zeichen \n als Zeilentrenner, so dass der String aus mehreren Zeilen besteht. Innerhalb einer Zeile werden aufeinanderfolgende Werte jeweils durch genau ein Leerzeichen getrennt. Die Reihenfolge der Zeilen und der Werte innerhalb einer Zeile entspricht der Reihenfolge in der Eingabedatei. Dadurch lassen sich die `__str__`-Funktionen auf einfache Weise testen.

Diese Beschreibung ist so ausführlich geworden, damit Sie selbst nicht das Design der Klassen entwickeln müssen. Ihre Aufgabe ist es, diese Spezifikation in lauffähigen Python-Code zu übertragen und zwar in der Datei `molecule.py`.

In späteren Übungsaufgaben werden Sie die hier genannten Klassen um weitere Methoden und weitere Klassen ergänzen.

**Bitte die Lösungen zu diesen Aufgaben bis zum 13.01.2020 um 18:00 Uhr an pfn1@zbh.uni-hamburg.de schicken. Die Besprechung der Lösungen erfolgt am 15.01.2020.**