Stefan Kurtz **Universität Hamburg**
Jeanine Liebold **Zentrum für Bioinformatik**

Programmierung für Naturwissenschaften 1
Wintersemester 2022/2023
Übungen zur Vorlesung: Ausgabe am 21.12.2022

Lehrevaluation

URLs zur Lehrevaluation:

Vorlesung: `https://evasys-online.uni-hamburg.de/evasys/online.php?p=N2GCA`

Übung: `https://evasys-online.uni-hamburg.de/evasys/online.php?p=G3LCZ`

Die Evaluation bezieht sich auf die gesamten PfN1-Übungen und alle drei Übungsgruppen, auch wenn auf der Web-Seite nur der Name von Frau Liebold, der Termin 14-16 für Gruppe 1 sowie ggf. noch PfN II angegeben wird.

Hinweis zu `assert`

In Python kann man Eigenschaften überprüfen, indem man diese in Form von `assert`-Anweisungen formuliert. Dabei folgt nach dem Schlüsselwort `assert` ein boolscher Ausdruck, der die Eigenschaft spezifiziert. Beim Ablauf des Programms wird jeweils verifiziert, ob dieser Ausdruck wahr ist und damit die Eigenschaft erfüllt ist. Wenn nicht, dann wird das Programm mit einer Fehlermeldung abgebrochen. Z.B. wird durch die `assert`-Anweisung

```
assert len(x_vector)== len(y_vector)and len(x_vector)> 0
```

verifiziert, dass `x_vector` und `y_vector` gleich lang sind und mindestens ein Element enthalten.

Selbsttest-Aufgaben zum Thema Funktionen in Python

Um ihre eigenen Fähigkeiten in der Python-Programmierung zu testen, lösen Sie bitte selbständig die folgenden Aufgaben. Nutzen Sie zunächst möglichst nicht den Python-Interpreter, sondern nur Papier und Stift. Sie können Ihre Lösung natürlich später mit dem Python-Interpreter verifizieren. Ihre Lösungen geben Sie nicht ab. Nutzen Sie die ersten 15 Minuten der Übung für diese Aufgaben. Es wird nicht erwartet, dass Sie alle Aufgabe in dieser Zeit lösen können.

1. Geben Sie bitte an, was das folgende Programmstück ausgibt. An welcher Stelle in der Liste wird durch `insert` ein Element eingefügt? Wozu dient in hier die Methode `re.sub`.

```
for n in [347304305,994244,1034,1,32442344]:
  parts = list()
  while n > 0:
    parts.insert(0,'{:03d}'.format(n % 1000))
    n //= 1000
  output = re.sub('^0+', '', ' '.join(parts))
  print('"{:>11s}"'.format(output))
```

2. Geben Sie bitte an, was die folgende Funktion generiert und was die `print`-Anweisung ausgibt. Geben Sie eine Gleichung an, nach der der Wert c_k der Variable `c_k` in Abhängigkeit von `k`≥ 2 berechnet werden kann.

```
def enumerste_series(n):
  a, b = 0, 4
  for k in range(2,n+1):
    c_k = 6 * b - 5 * a
    yield k, c_k
    a = b
    b = c_k
print('k c_k\n{}'.format('\n'.join(map(str,enumerste_series(8)))))
```

3. Welche Tests werden durch die folgende Funktion durchgeführt? Beschreiben Sie die Vorgehensweise.

```
def check_consistency(filename):
  d = {'sh' : '/bin/sh', 'py' : '/usr/bin/env python3'}
  try:
    stream = open(filename)
  except IOError as err:
    sys.stderr.write('{}: {}\n'.format(sys.argv[0],err))
    exit(1)
  for line in stream:
    mo = re.search(r'^#!(.*)$',line.rstrip())
    if mo:
      executer = mo.group(1).lstrip()
      for suffix, expected_executer in d.items():
        if re.search(r'\.{}$'.format(suffix),filename) and \
           executer != expected_executer:
          sys.stderr.write('{}: inconsistencies for {}\n'
                           .format(sys.argv[0],filename))
          exit(1)
      break
  stream.close()
```

4. Beschreiben Sie, was die folgende Funktion berechnet, wenn `n` eine positive ganze Zahl ist und `factors` eine Liste von positiven ganzen Zahlen. Was wird durch die `for`-Schleife nach der Funktionsdefinition berechnet und ausgegeben? Dabei ist `primes_up_to_100` eine Liste der Primzahlen ≤ 100.

```
def count_products(n,factors):
  if n == 1:
    return 1
  count = 0
  for i in factors:
    if n % i == 0:
      count += count_products(n // i, factors)
  return count
for n in range(100,150+1):
  if count_products(n,primes_up_to_100) == 0:
    print(n)
```

Aufgabe 9.1 (3 Punkte)

In der Datei `redundant.py` finden Sie ein Python-Programm mit vielen Redundanzen und einigen Programmteilen, die vereinfacht werden können.

Implementieren Sie in einer Datei `structured.py` eine strukturierte Version des Programms aus `redundant.py`. Die Strukturierung erfolgt durch die Deklaration einer Funktion `compare` mit folgenden Eigenschaften:

- Durch diese Funktion sollen möglichst viele Redundanzen im Programmcode der Datei `redundant.py` vermieden werden.
- Die Berechnung der Werte in den Variablen mit dem Suffix `_m` soll mit Hilfe von existierenden Python-Funktionen (siehe `python-slides.pdf`, Abschnitt Lists, frame 13) erfolgen. Dazu müssen Sie sich überlegen, welche Werte in diesen Variablen gespeichert werden.

Ihr Programm `structured.py` soll durch drei Aufrufe der Funktion `compare` mit den passenden Argumenten das gleiche Ergebnis liefern wie das Programm `redundant.py`. Das verifizieren Sie durch `make test`. Entwickeln Sie den Programmcode von `compare` schrittweise und geben Sie ggf. Zwischenergebnisse aus, die Sie mit den entsprechenden Zwischenergebnissen aus `redundant.py` vergleichen. Sie dürfen in Ihrem Programm nur das Modul `math` importieren.

Die Musterlösung umfasst 24 Zeilen Programmcode, einschließlich der Definition der drei Listen nach der Deklaration von `compare`. Ihre Lösung sollte nicht mehr als 30 Zeilen umfassen.

Punkteverteilung:

- 2 Punkte für die strukturierte Implementierung
- 1 Punkt für den bestandenen Test

Aufgabe 9.2 (4 Punkte)

In dieser Aufgabe geht es nochmals um das Problem der vollständigen Zerlegung einer positiven ganzen Zahl in Summanden. Diesmal soll nicht nur berechnet werden, ob das Problem eine Lösung hat, sondern es sollen alle additiven Zerlegungen berechnet werden. Hier nochmals die Problemdefinition:

Sei n eine positive ganze Zahl und S eine Menge positiver ganzer Zahlen. Eine additive Zerlegung von n bzgl. S ist eine Liste $[s_1, s_2, \ldots, s_k]$ von k Zahlen aus S, mit $k \geq 1$, so dass gilt:

- $s_i \leq s_{i+1}$ für alle i, $1 \leq i \leq k-1$
- $n = \sum_{i=1}^{k} s_i$

D.h. alle Zahlen einer additiven Zerlegung stammen aus S und die Zerlegung enthält mindestens ein Element. Außerdem ist die additive Zerlegung aufsteigend sortiert und ihre Summe ist n. Zahlen dürfen mehrfach vorkommen.

Hier ist eine Tabelle mit der Menge der additiven Zerlegungen $L(n, S)$ für gegebene n und S.

n	S	$L(n, S)$
11	$\{2, 3, 5\}$	$\{[2, 2, 2, 2, 3], [2, 2, 2, 5], [2, 3, 3, 3], [3, 3, 5]\}$
32	$\{7, 11, 13\}$	$\{[7, 7, 7, 11]\}$
38	$\{7, 11, 13\}$	$\{[7, 7, 11, 13]\}$
45	$\{8, 9\}$	$\{[9, 9, 9, 9, 9]\}$
47	$\{11, 12, 13, 14\}$	$\{[11, 11, 11, 14], [11, 11, 12, 13], [11, 12, 12, 12]\}$
47	$\{13, 14\}$	$\emptyset$

Hinweis: Die additiven Zerlegungen lassen sich wie folgt rekursiv berechnen: Man probiert von links nach rechts Zahlen s aus `terms_of_sum` (der aufsteigend sortierte Liste der Zahlen aus S) aus, subtrahiert s von dem aktuellen Restwert `remain`, falls $s \leq$`remain`, und löst dann für alle noch zulässigen Elemente aus `terms_of_sum` rekursiv ein kleineres Teilproblem der additiven Zerlegung.

Implementieren Sie in einer Datei Datei `splitnumber_sol.py` die folgenden zwei Funktionen:

Die Funktion `splitnumber_sol(number,terms_of_sum)` liefert mit einer `return`-Anweisung die Liste der additiven Zerlegungen von `number` bzgl. `terms_of_sum` zurück. Dabei ist `number` die zu zerlegende Zahl und `terms_of_sum` eine aufsteigend sortierte Liste, die die Menge S in der obigen Definition repräsentiert. Z.B. muss `splitnumber_sol(47,[11,12,13,14])` die Liste `[[11,11,11,14],[11,11,12,13],[11,12,12,12]]` zurückliefern.

Diese Funktion ruft eine rekursive Funktion `splitnumber_sol_rec` mit vier Argumenten auf:

- das Argument `solutions` ist die Liste der bisher gefundenen additiven Zerlegungen des Ausgangswertes n bzgl. S. An diese Liste muss also jeweils eine gefundene Zerlegung angehängt werden.
- `terms_of_sum_remain` sind die übrigen Elemente aus S, die noch ausgewählt werden können
- `remain` ist eine ganze Zahl $\leq n$, für die noch eine additive Zerlegung der Zahlen aus der Liste `terms_of_sum_remain` berechnet werden muss.
- `l` ist eine Liste, die die in den bisherigen rekursiven Aufrufen aufgesammelten Summanden aus `terms_of_sum` enthält, d.h. die Summe von `remain` und der Werte aus `l` ist n. Wenn also `remain` 0 ist, dann ist `l` eine additive Zerlegung von n. Beachten Sie, dass Sie für jeden rekursiven Aufruf zunächst durch `new_l = l.copy()` eine Kopie von `l` erzeugen müssen, an die man den nächsten Summanden anhängt.

Die Funktion `splitnumber_sol_rec` hat keinen Rückgabewert.

Beantworten Sie in der Datei `terms_of_sum_remain.txt` die folgende Frage.

Warum ist es notwendig, dass man in der rekursiven Funktion einen Parameter `terms_of_sum_remain` verwendet, in dem nicht immer alle Elemente von S stehen? 1 Pkt

Im Material zu dieser Übung finden Sie

- die Musterlösung der SplitnumberExists-Aufgabe, die Sie als Grundlage Ihrer Lösung nutzen können,
- eine Datei `splitnumber_sol_unit_test.py`, die Ihr Modul `splitnumber_sol.py` importiert und die Funktion `splitnumber_sol` aufruft und testet. Sie müssen sich also in dieser Aufgabe nicht um die Behandlung von Benutzerfehlern kümmern. Durch `make test` werden Tests durchgeführt, die verifizieren, dass Ihre Funktion für die zahlreichen Testfälle die richtigen Ergebnisse liefert.

Punkteverteilung:

- 1.5 Punkte für die korrekte Implementierung von `splitnumber_sol_rec`.
- 0.5 Punkte für die korrekte Implementierung von `splitnumber_sol`.

- 1 Punkt für erfolgreiche Tests.
- 1 Punkt für die korrekte Antwort auf die gestellte Frage.

Aufgabe 9.3 (2 Punkte)

In der Vorlesung wurde im Abschnitt *Reading and representing data matrices* gezeigt, wie man eine Matrix in ein Dictionary von Dictionaries konvertiert und dieses wieder ausgibt. Den entsprechenden Programmcode finden Sie in den beiden Dateien `data_matrix.py` und `data_matrix_main.py` im Verzeichnis `pfn1_2022/Vorlesung/src/Chemistry`.

Implementieren Sie nun in einer Datei `data_matrix_class.py` eine Klasse `DataMatrix`, die die gleiche Funktionalität bietet, wie die drei Funktionen aus `data_matrix.py`. Im Einzelnen müssen Sie die genannte Klasse mit zwei Member-Variablen `_matrix` und `_attribute_list` und den folgenden Methoden implementieren:

- `__init__(self,lines,key_col=1,sep='\t')` entspricht der Methode `data_matrix_new` aus `data_matrix.py`. `__init__` liefert natürlich keine Werte über eine `return`-Anweisung, sondern speichert die Matrix und die Attributliste in den genannten Member-Variablen.
- Die Methode `show(self,sep,attributes,keys)` soll das Gleiche leisten, wie die Funktion `data_matrix_show`.
- Die Methode `show_orig(self,sep,attributes,keys)` soll das Gleiche leisten, wie die Funktion `data_matrix_show_orig`.
- Die Methode `keys(self)` liefert `self._matrix.keys()` über eine `return`-Anweisung.
- Die Methode `attribute_list(self)` liefert `self._attribute_list` über eine `return` -Anweisung.

Die beiden Member-Variablen sind privat und dürfen nicht außerhalb der Klassendefinition verwendet werden.

Nach der Implementierung der Klasse kopieren Sie den Programmcode aus `data_matrix_main.py` in die Datei `data_matrix_class.py` und modifizieren ihn so, dass die Funktionalität bzgl. der Datenmatrizen durch die Methoden der Klasse `DataMatrix` realisiert wird. Das Hauptprogramm soll ausgeführt werden, wenn `__name__ == '__main__'` gilt. Dadurch kann die Klasse in Zukunft weiterverwendet werden.

In den Materialien finden Sie zwei Testdateien `atom-data-mini.tsv` und `atom-data.tsv` und ein Makefile. Durch `make test` verifizieren Sie, dass Ihr Programm für diese Testdaten korrekt funktioniert.

Punkteverteilung: 1.5 Punkte für eine nachvollziehbare Implementierung der Klasse; 0.5 Punkte für den bestandenen Test

Aufgabe 9.4 (5 Punkte)

In dieser Aufgabe geht es darum, in Python3 eine Klasse `Morse` zur Codierung und Decodierung eines Textes durch Morse-Zeichen zu implementieren. Für jedes alphanumerische Zeichen sowie die Zeichen `.` und `,` ist der Morse-Code eine Folge zweier Signallängen (kurz und lang, dit und dah im Englischen). Diese Signallängen werden durch die Zeichen `.` und − beschrieben. Einen String, der nur aus den Zeichen `.` und − besteht, nennen wir *Morse-String*.

In der Datei `morseClass_template.py` finden Sie eine Basis-Implementierung der Klasse `Morse` mit einem Dictionary `morse_code`, das die Codierung der genannten Zeichen in einen

Morse-String definiert. `morse_code` enthält zusätzlich eine Codierung für das Leerzeichen. Benennen Sie die Datei `morseClass_template.py` in `morseClass.py` um.

Ihre Aufgabe besteht aus den folgenden Teilaufgaben.

1. Implementieren Sie in der Klasse `Morse` eine Methode `encode(self,text)`, die einen String `text` als Argument erhält und mit einer `return`-Anweisung den entsprechenden Morse-String zurückliefert. Bei der Codierung sollen Kleinbuchstaben wie die entsprechenden Großbuchstaben behandelt werden. Während bei der traditionellen Anwendung von Morse-Codes zwischen der Codierung von zwei aufeinanderfolgenden Zeichen eine kurze Pause erfolgt, die man in einem String typischerweise durch ein Leerzeichen codiert, soll in Ihrer Implementierung kein Leerzeichen eingefügt werden. Beispiel: Für den String `SOS` liefert die Funktion den Morse-String `...---...` 1 Punkt
2. Wenn das Programm `morseClass.py` mit der Option `--text` aufgerufen wird, wird der Morse-String angezeigt. Im Makefile sind einige Tests implementiert, die testen, ob Ihre Funktion `encode` korrekt funktioniert.
3. Wenn das Programm `morseClass.py` nicht mit der Option `--text` oder `--decode` aufgerufen wird, dann wird ein Shell-Skript generiert, das entsprechend der Zeichen des Morse-Strings ein Programm zum Abspielen der Dateien `dit.wav` und `dah.wav` aufruft. Wenn man dieses Shell-Skript über eine Pipe mit dem Befehl `sh -s` verbindet (siehe Makefile), kann man den Morse-String hören. Dafür muss unter macOS das Programm `afplay` und unter Linux das Programm `aplay` verfügbar sein.[1] Wenn das bei Ihnen der Fall ist, können Sie sich die Morsezeichen anhören (falls der Lautsprecher an ist). Testen Sie die Funktionalität durch den Aufruf von `make test_sound`.[2]
4. Auch wenn man den Morse-Code kennt, kann man den durch die Funktion `encode` gelieferten Morse-String nicht immer eindeutig dekodieren. Geben Sie in einer Textdatei `decoding_morse.txt` eine Erklärung hierfür anhand eines Beispiels, an dem Sie das Problem verdeutlichen. Zu Beantwortung dieser Fragen müssen Sie sich in `morseClass.py` das Dictionary `morse_code` ansehen. 1 Punkt
5. Um eine eindeutige Decodierung eines Morse-Strings zu ermöglichen, muss man eine andere Codierung verwenden, wie z.B. die Codierung entsprechend dem Dictionary `morse_code2_0` . Wenn man die Option `--mc2_0` wählt, erfolgt die Codierung entsprechend diesem Dictionary. Beispiel: Der Code von `SOS` entsprechend `morse_code2_0` ist `..-.-.....-.`
6. Implementieren Sie in der Klasse `MorseClass` eine Methode `decode`, die einen Morse-String (entsprechend der Codierung mit `morse_code2_0`) als Argument erhält und den hierdurch codierten Text mit einer `return`-Anweisung als Ergebnis liefert. Beispiel: Für den obigen Morse-String `..-.-.....-.` liefert diese Funktion den dekodierten String `SOS`. 2 Punkte
7. In den Materialien finden Sie 2 Dateien `unknown_short.code2_0` und `unknown.code2_0` mit Morsestrings (Version 2.0) für zwei unbekannte Texte. Die Originaltexte sind nicht in den Testdaten vorhanden. Trotzdem kann durch den Aufruf von `make test_decode` getestet werden, ob Ihre Implementierung der Methode `decode` korrekt funktioniert. Beschreiben Sie

[1] Unter Linux muss ggf. das Package `alsa-utils` installiert werden.

[2] Leider funktioniert dieser Test nicht in der virtuellen Linux-Version unter MS-Windows. Aber man kann sich dann einfach die Datei `SOS_sound.mp4` direkt unter MS-Windows anhören oder `make --MS_WINDOWS SOS` aufrufen und die im Terminal angezeigte Folge von Anweisungen unter MS-Windows ausführen. Diese Lösung wurde im Wintersemester 2020/2021 von Tim Wacke vorgeschlagen.

in einer Textdatei `test_decode.txt`, warum das möglich ist. Dafür müssen Sie sich das Makefile und den Optionsparser ansehen und ausprobieren, was die einzelnen Kommandos beim Aufruf von `make test_decode` bewirken. Schauen Sie sich außerdem den Text an, der durch `./morseClass.py -d unknown.code2_0` ausgegeben wird. Der Text erläutert eine wichtige Eigenschaft von Codierungen. 1 Punkt

Bitte die Lösungen zu diesen Aufgaben bis zum 09.01.2023 um 18:00 Uhr an pfn1@zbh.uni-hamburg.de schicken.