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

Programmierung für Naturwissenschaften 1
Wintersemester 2022/2023
Übungen zur Vorlesung: Ausgabe am 09.11.2022

Bitte beachten Sie auch die Hinweise am Ende des Übungsblatts.

Definition von Abkürzungen von cd-Befehlen

In den Übungen fällt immer wieder auf, dass beim Wechsel in Unterverzeichnisse die kompletten Pfade angegeben werden. Um das für häufig besuchte Unterverzeichnisse zu vereinfachen, definiert jeder Studierende in der Datei `.zshrc` (bei Verwendung der zsh) bzw `.bashrc` (bei Verwendung der bash) die folgenden Abkürzungen :

```
alias cdpfn1u='cd ${HOME}/pfn1_2022/Uebungen'
alias cdpfn1s='cd ${HOME}/my_pfn1/Uebungen'
```

Dabei wird vorrausgesetzt, dass das PfN1-Repository im Home-Verzeichnis abgelegt wird und das Verzeichnis mit den eigenen Lösungen im Verzeichnis `my_pfn1` liegt. Falls das nicht der Fall ist, müssen Sie diese Pfade entsprechend anpassen. In einem neuen Terminal können Sie dann durch den Befehl `pfn1u` in das Unterverzeichnis des Repositories mit den PfN1-Uebungen wechseln. Entsprechendes gilt für den Befehl `pfn1s`.

Selbsttest-Aufgaben zum Thema Listen in Python

Um ihre eigenen Fähigkeiten in der Python-Programmierung zu testen, lösen Sie bitte selbständig die folgenden Aufgaben. Nutzen Sie dabei zunächst nicht den Python-Interpreter, sondern nur Papier und Stift. Sie können Ihre Lösung natürlich später mit dem Python-Interpreter verifizieren. Um das zu vereinfachen, werde ich nach der Übung eine Datei `Uebungen/self_test_lists_code.py` mit den Progammstücken in das Repository kopieren. Ihre Lösungen geben Sie nicht ab. Nutzen Sie die ersten 15-20 Minuten der Übung für diese Aufgaben.

1. Charakterisieren Sie die einzelnen Teile der folgenden Schleife in Python3:

   ```
   for a in s:
     print('a={}'.format(a))
   ```

2. Notieren Sie jeweils nach jeder Anweisung (ab Zeile 2) den Wert der Liste `char_list` und geben Sie an, was das Program ausgibt.

   ```
   char_list = ['r','o','t','o','r']
   cc = char_list.pop(0)
   char_list.append(cc)
   cc = char_list.pop(3)
   char_list.insert(0,cc)
   print(''.join(char_list))
   ```

3. Notieren Sie jeweils nach jeder `insert`-Anweisung die Elemente der Liste `numbers` und geben Sie an, was das Program ausgibt.

```
numbers = [1,3,5]
for i in [0,2,4,6]:
  numbers.insert(i,i)
print(numbers)
```

4. Geben Sie die Ausgabe des folgenden Programmstücks an:

```
print(sum([0,1,2,3,4]))
print(sum([1,1,2,3,4][1:]))
print(sum([0,1,2,3,5,4,6,7][2:5]))
```

5. Geben Sie die Ausgabe des folgenden Programmstücks an:

```
j = 1
i = 0
while i <= 4:
  print(i * j)
  j = -1 * j
  i = i + 1
```

6. Sei `ls` eine Liste. Verändern Sie das folgende Programmstück so, dass die erste und letzte Zeile wegfällt und die gleiche Ausgabe entsteht:

```
idx = 0
for elem in ls:
  print('{} is at index {}'.format(elem,idx))
  idx = idx + 1
```

7. Geben Sie die Ausgabe des folgenden Programmstücks an:

```
for exponent in [2,3,4,5,6]:
  value = 2 ** exponent
  print('{} {}'.format(value,value % 3))
```

8. Geben Sie die Ausgabe des folgenden Programmstücks an:

```
ls = [0,1,2,3]
for l in ls:
  print(len(ls[l:]))
```

9. Geben Sie die Ausgabe des folgenden Programmstücks an:

```
ls = list()
for i in [1,2,3,4]:
  ls.append('{}'.format(i))
print(' + '.join(ls))
```

10. Geben Sie die Ausgabe des folgenden Programmstücks an:

```
for n in [2,3,5,7,11,13,17]:
  print('{} = 7 * {} + {}'.format(n,n // 7,n % 7))
```

11. Geben Sie die Ausgabe des folgenden Programmstücks an:

```
    ls = list()
    distance = ord('a') - ord('A')
    for cc in ['h','u','g','e']:
      ls.append(chr(ord(cc) - distance))
    print(''.join(ls))
```

Aufgabe 3.1 (2 Punkte) Wir betrachten das Python-Programm in der Datei

`false_statements.py`.

```
num1 = 7
num2 = 8
ls1 = []
ls2 = [1,2,3]

if num1 + num2 > 15:
  print('1')

if (num1 + num2) % 2 == 0:
  print('2')

if (num1 * num2) % 16 == 0:
  print('3')

if len(ls2) == 2:
  print('4')

if len(ls1) + len(ls2) == 5:
  print('5')

if ls1 != [] and ls1[0] == ls1[len(ls1)-1]:
  print('6')
```

Dabei ist `%` der Modulo-Operator (Restwert-Operator), d.h. für zwei ganze Zahlen `i` und `j` liefert `i % j` den ganzzahligen Rest beim Teilen von `i` durch `j`. Der Operator `==` vergleicht zwei Werte und liefert genau dann `True`, wenn die Werte gleich sind. Der Operator `!=` vergleicht zwei Werte und liefert genau dann `True`, wenn die Werte ungleich sind. Die eckigen Klammern bei einer Liste stehen für den Indexzugriff, d.h. wenn `ls` eine Liste der Länge n mit $n > 0$ ist, bezeichnet `ls[i]` mit $0 \leq i \leq n-1$ das $(i+1)$te Element der Liste. Z.B. ist `s[0]` das erste Element des Arrays und `s[n-1]` das $n-$te. Die Funktion `len`, angewandt auf eine Liste, liefert deren Länge. Die Anweisungen, die relativ zu einer `if`-Anweisung eingerückt sind, heißen Blöcke und werden nur dann ausgeführt, wenn die Bedingung in der `if`-Anweisung wahr ist.

Wenn man das Programm ausführt, liefert es keine Ausgabe im Terminal. Überlegen Sie, warum das so ist.

Kopieren Sie das Programm in eine Datei `true_statements.py` und ändern Sie nur die Werte der 4 Variablen `num1`, `num2`, `ls1`, `ls2` so, dass alle `if`-Anweisungen zu `True` auswerten und 1 bis 6 zeilenweise ausgegeben wird. Durch `make test` verifizieren Sie, dass Ihr Programm korrekt funktioniert.

Hinweise zur Lösung: `python-slides.pdf`, Abschnitt *Flow of control*, frame 3-13

Aufgabe 3.2 (2 Punkte)

In der Vorlesung haben Sie `while`- und `for`-Schleifen kennengelernt.

In den Materialien zur Übung finden Sie eine Datei `loops_orig.py`. Diese enthält 2 `for`-Schleifen und 2 `while`-Schleifen. Erstellen Sie eine Kopie der Datei mit dem Namen `loops.py`. Ersetzen Sie darin alle `for`-Schleifen durch `while`-Schleifen und umgekehrt. Bei den `while`-Schleifen darf nicht der `range`-Operator verwendet werden.

In den Materialien zur Übung finden Sie eine Testdatei und ein `Makefile`. Darin ist ein Test implementiert, der Ihr Programm aufruft und das Ergebnis mit Hilfe des Linux-Tools `diff` mit dem erwarteten Ergebnis vergleicht. Diesen Test können Sie mit dem Befehl `make test` ausführen (Achtung: Ein erfolgreicher Test bedeutet nicht unbedingt, dass die Aufgabe korrekt gelöst ist. Die Aufgabe gilt selbstverständlich erst dann als gelöst, wenn die Umwandlung bzgl. der Form der Schleifen auch durchgeführt wurde.)

Punktevergabe: Je 0.5 Punkte für die korrekte Konvertierung der Schleifentypen.

Aufgabe 3.3 (2 Punkte)

In den Materialien zu dieser Aufgabe finden Sie eine Datei `celsius_template.py`, die Sie in `celsius.py` umbenennen. In der Datei `celsius.py` implementieren Sie ein Python-Programm das für alle ganzzahligen Temperaturen von 1 bis 100 Grad Celsius die entsprechende Temperatur in Fahrenheit ausgibt. Zur Erinnerung: Wenn t_C die Temperatur in Celsius ist, dann ist $t_F = (t_C \cdot 9)/5 + 32$ die entsprechende Temperatur in Fahrenheit. Verwenden Sie die Methode `range()`, um die Werte im genannten Temperaturbereich aufzuzählen.

Am Anfang der Ausgabe soll eine Zeile der Form `# celsius fahrenheit` ausgegeben werden, wobei die beiden Bezeichner für die Maßeinheiten durch einen Tabulator getrennt werden. Formatieren Sie die Ausgabe mit `print`. Verwenden Sie dazu die Anweisung

```
print('{:.2f}\t{:.2f}'.format(celsius, fahrenheit))
```

wobei `celsius` und `fahrenheit` Variablen sind, in denen die auszugebenden Temperaturwerte in den entsprechenden Einheiten gespeichert sind. Im Material zu dieser Übungsaufgabe (Unterverzeichnis `Celsius`) finden Sie ein Makefile, das Tests durchführt. Durch `make test` verifizieren Sie, dass Ihr Programm korrekt funktioniert.

Hinweise zur Lösung: `python-slides.pdf`, Abschnitt *Flow of control*, frame 26-33

Aufgabe 3.4 (3 Punkte)

Die 20 Aminosäuren lassen sich in zwei Gruppen einteilen, nämlich in die Gruppe der hydrophoben und in die Gruppe der hydrophilen Aminosäuren:

hydrophob	`A`, `F`, `I`, `L`, `M`, `P`, `V`, `W`
hydrophil	`C`, `D`, `E`, `G`, `H`, `K`, `N`, `Q`, `R`, `S`, `T`, `Y`

Benennen Sie die Datei `hydrocount_template.py` in `hydrocount.py`. Implementieren Sie hierin ein Python-Script, das in `sys.argv` genau einen Dateinamen erwartet. Sie können davon ausgehen, dass die Eingabedatei Strings über dem Alphabet der Einbuchstaben-Codes für Aminosäuren (siehe `python-slides.pdf`, Abschnitt 6, frame 3) enthält.

Die Datei soll geöffnet und zeilenweise gelesen werden. Es soll für jede Zeile der Eingabedatei die Anzahl der hydrophoben und hydrophilen Aminosäuren bestimmt und ausgegeben werden. Zeichen im Eingabe-String, die keine Aminosäuren bezeichnen, können ignoriert werden.

Ihr Programm soll zunächst eine Tabulator-separierte Kopfzeile

```
# hydrophobic hydrophilic
```

ausgeben. Für jede Zeile der Eingabedatei soll eine Tabulator-separierte Zeile mit zwei ganzen Zahlen ausgegeben werden, nämlich die Anzahl der hydrophoben und die Anzahl der hydrophilen Aminosäuren der aktuellen Zeile. Beispiel: Für die Zeile `ISDKDLYVAALTNADLN` soll das Programm die folgende Ausgabe liefern:

```
8       9
```

Um das Programm robust zu machen und dem Benutzer einen Hinweis zu geben, wie es verwendet wird, testen Sie bitte mit einer `if`-Anweisung am Anfang des Programms, ob die Liste `sys.argv` genau zwei Elemente enthält. Wenn das nicht der Fall ist, dann soll eine Fehlermeldung der Form

```
Usage: ./hydrocount.py <inputfile>
```

mit `sys.stderr.write` auf den Standard-Fehlerkanal ausgegeben und das Programm mit `exit(1)` abgebrochen werden. Die Funktion `write` kann dabei so wie `print` verwendet werden. Allerdings muss am Ende noch ein \n eingefügt werden, damit nach der Fehlermeldung ein Zeilenvorschub erfolgt. Der Pfad des Programms steht in `sys.argv[0]`. Dieser muss in der obigen `Usage`-Ausgabe mit Hilfe eines Platzhalters im formatierten String ausgegeben werden.

Wenn `sys.argv` genau zwei Elemente enthält, dann ist `sys.argv[1]` der Name der Datei, die Sie in Ihrem Programm öffnen und zeilenweise lesen. Falls `line` die aktuelle Zeile ist, dann können Sie mit einer `for`-Schleife die einzelnen Zeichen der Zeile lesen und wie oben beschrieben die Anzahl der hydrophoben und der hydrophilen Aminosäuren in der aktuellen Zeile zählen. Verwenden Sie keinen indexbasierten Zugriff auf die Zeichen der Zeile.

Im Material zu dieser Übungsaufgabe gibt es eine Datei mit Proteinsequenzen und den dafür erwarteten Ergebnissen. Durch `make test` verifizieren Sie, dass Ihr Programm korrekt funktioniert.

Hinweise zur Lösung: `python-slides.pdf`, Abschnitt *Lists*, frame 15

Hinweise zur Lösung: `python-slides.pdf`, Abschnitt *Flow of control*, frame 3-13

Punktevergabe:

- 1 Punkt für die korrekte Behandlung möglicher Fehler.
- 1 Punkt für die korrekte Berechnung
- 1 Punkt für bestandene Tests

Bitte die Lösungen zu diesen Aufgaben bis zum 14.11.2022 um 18:00 Uhr an pfn1@zbh.uni-hamburg.de schicken.

Hinweise

In `Vorlesung/synopsis.pdf` finden Sie nun alle Folien mit Zusammenfassungen von Inhalten und Übersichtstabellen mit Methoden einiger wichtiger Klassen.

Bitte denken Sie daran, Ihre Programmcodes korrekt zu formatieren:

- Die Länge einer Zeile darf höchstens 80 Zeichen sein.
- Eingerückt werden muss konsistent mit 2 oder 4 Leerzeichen.
- Sie dürfen keine Tabulatoren verwenden.
- Benutzen Sie `bin/check_code.py`, um die Formatierung zu überprüfen.

Hinweise zu Tests

Ein Studierender hat gefragt, wie Tests und Makefiles funktionieren. Ich beschreibe das im Folgenden kurz. Für die Lösung der Übungsaufgaben ist es aber nicht notwendig, diese Hinweise zu lesen.

Für die meisten Übungsaufgaben finden Sie in den Materialien jeweils eine Datei `Makefile`. In dieser wird beschrieben, welche Tests es gibt und wie die Tests durchgeführt werden. Der Haupttest wird meist durch den Aufruf von

```
make test
```

im Terminal ausgeführt. Dazu muss natürlich die Datei `Makefile` sowie die Testdaten und (falls vorhanden) Testskripte aus dem Material zu Aufgabe vorhanden sein, d.h. Sie müssen diese in das Verzeichnis kopieren, in dem Sie Ihre Lösung entwickeln. Ein erfolgreicher Test sieht z.B. so aus:

```
$ make test
Congratulations. to_utf8 passed
Congratulations. to_latex passed
Congratulations. test passed
```

Hier wurde ein Haupttest mit dem Namen `test` durchgeführt und zwei Teiltests `to_utf8` und `to_latex`, die durch `test` initiiert wurden.

Bei einem erfolglosen Test bekommt man z.B. die folgende Ausgabe:

```
$ make test
1c1
< "Ubermütige Hühner brüten in der
---
> Übermütige Hühner brüten in der
18c18
< die "Uberresten der, über die
---
> die Überresten der, über die
make: *** [to_utf8] Error 1
```

Es gibt also ein Problem beim Test `to_utf8`. Dazu muss man sich in der Datei `Makefile` den Test `to_utf8` ansehen:

```
to_utf8:post_an_den_zwiebelfisch_latex.txt
        @./umlaut_latex_to_utf8.py $< | diff -
           post_an_den_zwiebelfisch_utf8.txt
        @echo "Congratulations. $@ passed"
```

Dort findet man den Aufruf des Kommandos `diff`, das die Ausgabe von `umlaut_latex_to_utf8.py` mit dem Inhalt der Datei `post_an_den_zwiebelfisch_utf8.txt` vergleicht. (Hinweise zu `diff` finden Sie bereits in einem frührern Übungsblatt) Das genannte Python-Skript konvertiert offensichtlich `"U` nicht in Ü, so dass es Unterschiede zur Referenzdatei gibt. Mit dieser Information kann man sich gezielt auf die Suche nach dem Fehler machen.

Hinweise zu Makefiles

In der Datei `Makefile` werden Tests durch Regeln der Form

```
goal:dependencies
  command
```

spezifiziert. Dabei ist `goal` ein frei wählbarer Bezeichner (z.B. ein Dateiname oder das Wort `test`) für ein sogenanntes Ziel, das erreicht werden soll. Wenn man nun `make goal` im Terminal ausführt, sucht `make` nach einer Regel, bei der links vom Doppelpunkt der Bezeichner `goal` steht. Das gilt für die obige Regel und `make` versucht diese Regel anzuwenden. Die Regel besagt, dass es Abhängikeiten gibt, die durch `dependencies` nach dem Doppelpunkt definiert werden. Das sind ein oder mehrere Bezeichner für andere Ziele (z.B. ein Dateiname oder `to_utf8`), die selbst wiederum durch Regeln beschrieben werden können, oder Namen existierender Dateien sind. `make` versucht nun durch Anwendung der entsprechenden Regeln diese Ziele zu erfüllen. Es kann sein, dass dafür andere Ziele erfüllt werden müssen. Dieser Prozess wird von `make` solange durchgeführt, bis alle `dependencies` erfüllt sind. Dann ruft `make` das Kommando `command` auf, das in der obigen Regeln mit einem Tabulator eingrückt wird.

Durch Makefiles lassen sich damit komplexe Abhängigkeit zwischen Programmaufrufen (z.B. beim Kompilieren von Programmcode oder beim Testen) durch einfache elementare Regeln beschreiben und `make` kümmert sich dann darum diese Regeln anzuwenden, d.h. die Kommandos in der richtigen Reihenfolge aufzurufen. Ein großer Teil der Software im akademischen Umfeld wird heute mit Hilfe von Makefiles kompiliert und getestet.