ASE Übung 4 - Python

Einführung

Diese Übung ist eine kurze Einführung in die Programmiersprache **Python** (`https://www.python.org/`). Hierbei handelt es sich um eine einfach zu lernende, *höhere* Programmiersprache (d.h. von den Details des Computers stark abstrahiert), die wegen ihrer Einfachheit und der großen Anzahl verfügbarer Programmbibliotheken im Bereich der Wissenschaften sehr beliebt ist.

Das Ziel ist es den allgemeinen Programmablauf und grundlegende Elemente der Syntax zu verstehen, um damit anschließend ein einfaches Problem im Kontext biologischer Sequenzen zu lösen.

Programmausführung und -ablauf

Python lässt sich als *interaktive* Session, in welcher eingegebene Anweisungen direkt ausgeführt werden, mit dem Aufruf des Python-Interpreters `python` in der Kommandozeile starten (und mit dem Aufruf der Funktion `exit()` oder `CTRL-D` beenden).

Darüber hinaus lassen sich mehrere Anweisungen (und entsprechend ganze Programme) in Python-*Skripten* (einfache Textdateien mit dem Dateisuffix `.py`) für einfache wiederholte Ausführung zusammenfassen. Hierbei werden alle Anweisungen in der Dateien zeilweise ausgeführt, sofern der *Kontrollfluss*-Anweisungen es nicht anders vorgeben. Der Code kann durch *Kommentare* dokumentiert werden. Hierbei handelt es ich um freien Text welcher dem '`#`' Zeichen folgt und vom Programm ignoriert wird.

Syntax Grundlagen

Variablen

In Python bestehen Variablen aus einem *Bezeichner* (beliebige zusammenhängende Folge von Buchstaben, Zahlen und '_' — darf kein Kennwort der Programmiersprache sein und nicht mit einer Zahl beginnen) und können eine Vielzahl unterschiedlicher *Typen* von *Werten* mit dem Zuweisungsoperator '=' zugewiesen werden. Nützlich sind dabei *literale* Konstanten. Dies sind Ausdrücke, die direkt einen Wert repräsentieren und einer Variable zugewiesen werden können. Häufig verwendete Typen sind hierbei *Ganzzahlen* (*integer*), *Gleitkommazahlen* (*floating point*), *Zeichenketten* (*string*) und *Wahrheitswerte* (*boolean*):

```
some_integer_variable = 12
some_floating_point_variable = 12.345
some_string_variable = "This is free text."
some_boolean_variable = True
another_boolean_variable = False
```

Die Werte der Variablen können durch erneute Zuweisung jederzeit überschrieben werden.

Arithmetik & Relationen

Wie jede Programmiersprache erlaubt Python grundlegende Arithmetik auf Variablen und Literalen mit den allgemein bekannten Operatoren:

```
>>> a = 2
>>> b = 3
>>> a + b
5
>>> a * 4
8
>>> 12 / b
4.0
```

Die Vergleichsoperatoren '==' (*gleich*), '!=' (*ungleich*), '<=' (*kleiner-gleich*), '>=' (*größer-gleich*), '<' (*kleiner*) und '>' (*größer*) erzeugen jeweils einen Wahrheitswert als Ergebnis:

```
>>> a == b
False
>>> (a + 1) != b
False
>>> a < b
True
>>> b >= 3
True
>>> b > 3
False
```

Die Operatoren lassen sich nicht nur auf Zahlen, sondern auch auf andere einfache Datentypen anwenden, wie z.B. Zeichen und Zeichenketten:

```
>>> "text" == "text"
True
>>> "text" == "test"
False
```

Aufrufen von Funktionen

Mehrere Anweisungen können zur wiederholten Ausführung in *Funktionen* gekapselt werden. Python verfügt über eine Anzahl nützlicher vordefinierter Funktionen. Eine Funktion kann eine Anzahl von Parametern über die Parameterklammern `()` nehmen (Reihenfolge der Parameter wird beachtet) und einen Wert zurückgeben. Die `print()` Funktion nimmt eine Zeichenkette als Parameter, welche in die Konsole ausgegeben wird, erzeugt jedoch keinen sinnvollen Rückgabewert. Eine Funktion mit Rückgabewert ist z.B. `len()`, welche als Parameter eine Zeichenkette (oder andere Datentypen, welche mehrere Daten zusammenfassen) akzeptiert und die Anzahl der Elemente (d.h. Länge) zurück gibt:

```
>>> some_string = "abcde"
>>> length = len(some_string)
>>> length == 5
True
>>> print(some_string)
abcde
```

Indexierung

Datentypen, die wie Zeichenketten aus einer Reihe von Elementen bestehen, erlauben es auf einzelne Elemente zuzugreifen. Dabei wird der *Index* des gewünschten Elements der Variable über die *Indexklammern* `[]` übergeben:

```
>>> some_string = "abcde"
>>> print(some_string[2])
c
```

Dabei ist zu beachten, dass in Python die Indizes bei `0` beginnen, d.h. `[0]` gibt das erste Element, `[1]` das zweite, usw.

Bedingte Anweisungen und Verzweigungen

Häufig ist es notwendig einige Anweisungen nur unter bestimmten Bedingungen auszuführen. Dies geschieht mit dem Kennwort `if` gefolgt von einem Ausdruck, welcher einen Wahrheitswert ergibt, und einem '`:`'. Anweisungen in den folgenden Zeilen, **welche um 4 Leerzeichen eingerückt sind**, werden ausgeführt, wenn der Wahrheitswert `True` ergab. Ist es notwendig im gegenteiligen Fall (d.h. wenn der Ausdruck `False` ergab) ebenfalls Anweisungen auszuführen, so kann dies nach dem `if`-Block mit dem Kennwort `else:` erzeugt werden:

```
>>> a = 10
>>> b = 20
>>> if a < b:
...     print("a is smaller than b.")
... else:
...     print("a is larger than b.")
...
a is smaller than b.
```

Bedingte Anweisungen können beliebig geschachtelt werden, wobei die Einrückung eines inneren Blocks entsprechend 4 Leerzeichen weiter sein muss, als die des äußeren.

Schleifen

Ist es notwendig Anweisungen einige Male zu wiederholen, so stehen Schleifenausdrücke zur Verfügung, wie z.B. die `while` Schleife, welche Anweisungen solange wiederholt, wie eine bestimmte Bedingung zutrifft. Wie bei dem `if` Ausdruck (und in Python generell, wann immer ein bestimmter Kontext von seiner Umgebung unterschieden werden soll), werden die zu wiederholenden Anweisungen um 4 Leerzeichen eingerückt. Der folgende Code nutzt die `while` Schleife um alle Buchstaben einer Zeichenkette einzeln in der Konsole auszugeben:

```
>>> some_string = "abcde"
>>> i = 0
>>> length = len(some_string)

>>> while i < length:
...     print(some_string[i])
...     i = i + 1
...
a
b
c
d
e
```

Dabei wird zunächst eine Indexvariable `i` mit dem Wert `0` erzeugt und die Länge der Zeichenkette gespeichert. Die Schleife läuft dann so lange, wie die Indexvariable kleiner als die Länge ist. In jeder Iteration wird der Buchstabe auf Position `i` ausgegeben und anschließend `i` um `1` erhöht.

Die Bedingung der folgenden Schleife wird immer zutreffen und die Schleife läuft endlos:

```
>>> i = 0
>>> while True:
...     i = i + 1
```

Ein Skript mit dieser Schleife wird nicht von selbst enden, da das Ende der Datei (und somit der Anweisungen) nie erreicht wird. Es lässt sich nur manuell mit dem *Interrupt*-Befehl `CTRL-C` beenden.

Definition eigener Funktionen

Es ist möglich neue Funktionen zu definieren. Hierzu verwendet man den Ausdruck `def` gefolgt von dem Funktionsnamen und in klammern aufgelisteten Parametern (mit gleichen Restriktionen wie bei Variablen). Die Anweisungen, welche die Funktion ausführen soll, werden wie bei Schleifen oder Konditionen durch Einrückung zugeordnet. Dabei werden die Parameter wie definierte Variablen behandelt, deren Werte beim Aufruf der Funktion bestimmt werden. Funktionen können Werte mittels des `return` Stichwortes zurück geben. Die Folgende Funktion ermittelt die kleinere der als Parameter übergebenen Zahlen:

```
>>> def get_smaller_number(a, b):
...     if a < b:
...             return a
...     else:
...             return b
...
>>> get_smaller_number(10, 20)
10
```

Sequenzidentität

Implementieren Sie eine Funktion zur Berechnung der Sequenzidentität zweier Sequenzen gleicher Länge. Nutzen Sie hierfür ein bereitgestelltes Skript, was Sie sich zunächst in Ihr Arbeitsverzeichniss mit folgenden Kommandozeilenbefehlen kopieren:

```
mkdir ase_uebung_04
cd ase_uebung_04
cp /home/olzhabaev/Public/seq_identity.py .
```

Öffnen Sie die kopierte Datei mit einem Texteditor (z.B. `kate`). In diesem Skript ist eine Liste mit Sequenzen definiert (Z. 35). Für ein Paar dieser Sequenzen wird die unvollständige

Funktion zur Berechnung der Sequenzidentität aufgerufen (Z. 41). Anschließend wird das Ergebnis in die Konsole ausgegeben (Z. 44).

Vervollständigen Sie die Funktion `compute_sequence_identity(...)`, so dass diese die Sequenzidentität zweier gleich langer Sequenzen berechnet. Setzen Sie hierfür die folgenden Schritte in Code um:

1. Ermitteln Sie die Längen der beiden Eingabesequenzen.
2. Vergleichen Sie die Längen.
3. Sind die Längen nicht gleich, so soll ein entsprechender Text in der Konsole erscheinen. Die Funktion führt danach keine weitere Anweisung mehr durch.
4. Anderenfalls sollen übereinstimmende Positionen gezählt werden.
   (a) Legen Sie eine Match-Zählervariable (z.B. `num_matches`) an mit entsprechendem Initialwert.
   (b) Erzeugen Sie eine `while` Schleife mit entsprechender Indexvariable (z.B. `i`) zur Iteration durch die Sequenzpositionen.
   (c) In jeder Iteration `i` soll geprüft werden, ob die Sequenzen an Position `i` gleich sind. Ist das der Fall, so soll die Zählervariable um 1 erhöht werden.
   (d) Am Ende der Schleife soll die Indexvariable `i` um 1 erhöht werden.
5. Rechnen Sie die Anzahl der Matches zu einer Prozentzahl um, und lassen Sie die Funktion diesen Wert zurück geben.

Sie können das Skript nun ausführen, indem sie in der Kommandozeile den Python-Interpreter mit dem Dateinamen des Skripts als Argument starten:

```
python seq_identity.py
```

Überprüfen Sie Ihre Implementation, indem Sie die Sequenzidentität für unterschiedliche Paare der definierten Sequenzen berechnen. Wählen Sie hierzu andere Indizes der Sequenzliste beim Aufruf der Funktion in Zeile 41. Die erwarteten Werte für alle Paare der definierten Sequenzen finden Sie zum Vergleich in folgender Matrix:

```
  |   0   1   2   3
--+----------------
0 | 100
1 |  45 100
2 |  20  25 100
3 |  15  30  25  100
```