Dokumentation Aufgabe 5

1. Lösungsidee

Als Grundidee dachte ich mir: „Ich brauche ein Programm, welches alle Möglichen Kombinationen zwischen Schülern und Geschenken berechnet, und diese Kombinationen bewertet, um später die beste Kombination herauszufinden.“

2. Umsetzung

Das Programm ist in drei Hauptschritte unterteilt: Datei auslesen, Die beste Kombination herausfinden und zum Schluss das Ergebnis ausgeben.

```
if __name__ == "__main__":
    read_file("wichteln1.txt")
    best = calc_highscore()
    print_results(best)
```

Abb. 1

Datei auslesen

```
11  def read_file(filename):
12      try:
13          with open(filename, "r") as file:
14              i = -1
15              for line in file:
16                  i += 1
17                  if i != 0:
18                      wish = [int(char) for char in line.split() if char.isdigit()]
19                      current_student = Student(i, wish[0], wish[1], wish[2])
20                      all_students.append(current_student)
21
22
23      except IOError:
24          print("Konnte die Datei nicht lesen!")
25      except:
26          print("Fehler.")
```

Abb. 2

Ich habe die *„try – except“* Syntax verwendet um eventuelle Fehler beim Auslesen abzufangen. Wenn ein solcher auftritt kann ich so eine Angepasste Fehlermeldung ausgeben.

Dann geht er Zeile für Zeile die Datei durch, wobei die erste Zeile übersprungen wird (da nur Schüleranzahl). Er erstellt eine Liste **wish** und fügt dieser alle Zahlen der aktuellen Zeile als neues Element an. Bei dem ganzen Prozess zählt er die Zeilennummer mithilfe der Variable **i**. Diese

Nummer wird auch gleichzeitig als einzigartiger Name von den Schülern verwendet (siehe Zeile 18).

Jeder Schüler wird durch ein Klassenobjekt repräsentiert:

```
4  class Student:
5      def __init__(self, nr, w1, w2, w3):
6          self.nr = nr
7          self.whishes = [w1 - 1 , w2 - 1, w3 - 1]
```

Abb. 3

Dabei hat er eine Nummer **nr**, die, wie oben erwähnt, den Namen darstellt. Laut Aufgabenstellung hat jeder Schüler einen Erst- , Zweit- und Drittwunsch, hier im Programm gespeichert durch die Variablen **w1**, **w2** und **w3**.

```
1  all_students = []
2  highscore : int = 0
```

Abb. 4 „Globale Variablen“

Er fügt der globalen Liste (siehe Abb. 4) **all_students** (siehe Abb. 2, Zeile 19) eine Instanz der Klasse **Student** hinzu, übergeben werden die Zahlen, die in **wish** gespeichert sind.

Kombinationen bewerten

```
41 def calc_highscore():
42     # Definiert die Schwerpunkte der Punkteverteilung
43     # STellt sicher, dass die 'Erstwunscherfüllung' sehr stark gewichtet wird
44     scores = [len(all_students) * len(all_students), len(all_students), 1]
45     highscore = 0
46     best_perm = None
47     score = 0
48 
49     for p in perm(all_students):
50         score = 0
51         for i, person in enumerate(p):
52             if person.whishes[0] == i:
53                 score += scores[0]
54 
55             elif person.whishes[1] == i:
56                 score += scores[1]
57 
58             elif person.whishes[2] == i:
59                 score += scores[2]
60         if score > highscore:
61             highscore = score
62             best_perm = p
63     return best_perm
```

Abb. 5

Diese Funktion ist „das Herz des Programms“. Als erstes erstelle ich eine Liste **scores**, die die Punkteverteilung repräsentiert. Dabei ist das erste Element die Punktzahl für die Erfüllung des Erstwunsches, das zweite Element die Punktzahl für die Erfüllung des Zweitwunsches und das dritte Element die Punktzahl für die Erfüllung des Drittwunsches.

Der **highscore** (siehe Abb. 4) ist ein Integer, welcher die maximal erreichte Punktzahl von allen Kombinationen darstellt. Sobald die Punktzahl (**score**) der aktuellen Permutation höher ist als der **highscore** wird **highscore** auf diesen Wert gesetzt und die Kombination in der Variable **best_perm** gespeichert. Siehe dazu Zeile 60-62.

Um alle möglichen Permutationen zu bekommen nutze ich eine Hilfsfunktion namens **perm**:

```
34  def perm(l):
35      if len(l) == 0:
36          return [[]]
37
38      return [x for y in perm(l[1:]) for x in addperm(l[0],y) ]
```

Abb. 6

Diese nimmt als Argument eine Liste **l**. Als erstes wird geschaut, ob diese Liste nicht leer ist, ansonsten gibt die Funktion eine Leere Liste zurück (Zeile 35-36).

Erklärung Zeile 41: In der einen Zeile „verstecken“ sich zwei ineinander verschachtelte „*For-Schleifen*“.
In der ersten `for y in perm(l[1:])` Schleife ruft die Funktion sich selber auf. Als Argument übergibt sie sich eine Liste, die alle Elemente von **l** enthält, bis auf das Erste.
In der zweiten Schleife `for x in addperm(l[0],y)` ruft er eine Funktion **addperm** mit den Argumenten **l[0]** und **y** auf**.** Dies bedeutet, dass der Funktion **addperm** das erste Element der Liste **l** und als zweites Argument der Rest der Liste **l**, gespeichert in der Variable **y** (siehe erste Schleife) übergeben wird.

```
28  def addperm(x,l):
29      # Verscheibt das element 'x' in der Liste 'l'.
30      # Am Anfang ist 'x' an 1. Stelle in der Liste, dann an 2. Stelle usw.
31      # Solange um 1 nach hinten verschoben, bis 'x' das letzte Element in der Liste ist.
32      return [ l[0:i] + [x] + l[i:]  for i in range(len(l)+1) ]
```

Abb. 7

Die Funktion **addperm** (siehe Abb. 7) nimmt als Argumente **x** und **l**. Innerhalb der Funktion gibt es wieder eine Ausgabe (Zeile 32). Diese setzt sich zusammen aus: a) einer Liste aller Elemente der Liste **l** (Argument); b) Der Variable **x** als Liste und c) den Rest der Liste **l** ab dem Element mit dem Index **i**. Das heißt, die Ausgabe ist eine Liste, mit einer bestimmten Anzahl von Listen als Elemente. Diese „Unterlisten“ sind an sich gleich der Liste **l** , nur dass der Wert für **x** immer „in der Liste um eins nach hinten rutscht“.

```
[['Peter', 'Johanna', 'Alfred'], ['Johanna', 'Peter', 'Alfred'], ['Johanna', 'Alfred', 'Peter']]
```

Beispielausgabe, bei `addperm("Peter", ["Johanna", "Alfred"])`

Bezüglich der Beispielausgabe: `"Peter"` repräsentiert in dem Fall die Variable **x**.
`["Johanna", "Alfred"]` das Argument **l.**

Diese Ausgabe wird nun so oft ausgeführt, wie die Liste **l** Elemente hat.

Aufgrund der ersten *„For-Schleife“* in **perm** (siehe Abb. 6) werden so alle möglichen Kombinationen ermittelt.

```
49      for p in perm(all_students):
50          score = 0
51          for i, person in enumerate(p):
52              print(str(i))
53              #print(person)
54              if person.whishes[0] == i:
55                  score += scores[0]
56
57              elif person.whishes[1] == i:
58                  score += scores[1]
59
60              elif person.whishes[2] == i:
61                  score += scores[2]
62          if score > highscore:
63              highscore = score
64              best_perm = p
65      return best_perm
```

Abb. 8

Für jede dieser Kombinationen prüft das Programm dann welche Wünsche der Schüler erfüllt sind und verteilt dementsprechend die Punkte. Am Ende gibt die Funktion eine Liste mit Schülern zurück. Diese Liste ist, so sortiert, dass der erste Schüler in der Liste Geschenk 1 bekommt usw..

Ergebnis ausgeben

```
68  def print_results(best_permutation):
69      i = 0
70      for s in best_permutation:
71          i += 1
72          print("Schüler: ", s.nr, " bekommt Geschenk ", i)
```

Abb. 9

Die letzte Funktion gibt die beste Verteilung in die Kommandozeile aus. Dazu zählt sie eine Variable **i** hoch. Dies ist die Geschenknummer. Danach gibt sie den Schüler mit seiner Nummer aus, und anschließend **i**.

3. Limitierungen

Bei meinen Tests mit den Beispieldateien mit mehr als ~11 Schülern bin ich auf extrem hohen Arbeitsspeicherverbrauch gestoßen, der benötigt wird, um die Liste, die alle Permutationen von **all_students** enthält, aufzubauen. Zusätzlich würde die Bewertung auszurechnen auch sehr lange dauern.

Eine weitere Limitierung ist der Zeitaufwand, um die ganzen Permutationen zu berechnen.

Beide Probleme würden sich zumindest vermindern lassen, indem man nur alle Permutationen für die ersten 3 Wünsche berechnet. Somit wären das bei 1000 Schülern nur 997.002.000 mögliche Kombinationen, was für einen Computer möglich ist.