1. **Einlesen des Daten**

Das Programm wird in Python implementiert.

Z.1 - 5: Hier werden ähnlich wie bei den anderen Aufgaben die Input-Daten eingelesen und in eine Liste eingetragen. Danach wird bei jedem Element das Zeilensprung-Zeichen (\n) sowie die erste beide Elemente entfernt. In den nächsten beiden Zeilen werden nun das String-Format zum int konvertiert.

Jetzt haben wir eine Liste von Listen, die die dreieckigen Teile des Puzzles sind.

2. **Methoden**

Meine Methode basiert im Grunde darauf, jede Rotation eines Teilchens mit der des anderen zu verknüpfen und zu überprüfen, ob diese zusammenpassen (anhand der Voraussetzung, dass die Summe der berührenden Seiten sich 0 ergibt).

- permutate(), erzeugt die 3 Rotation eines dreieckigen Teilchens, welche 3 Listen sind und werden in einer tuple gelagert. Tuple wird hier aufgrund der Ausführungszeit bevorzugt.

  Beispiel: `[-3, -2, -1]` → `([-3, -2, -1], [-2, -1, -3], [-1, -3, -2])`

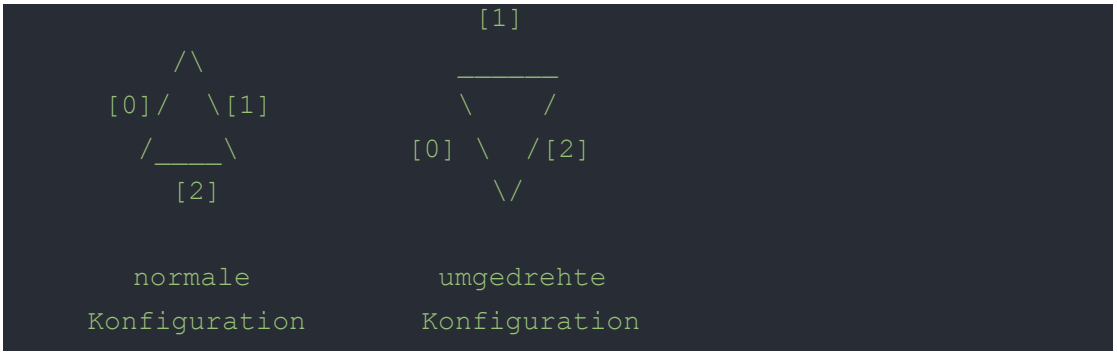
  `[-2, -4, 1]` → `([-2, -4, 1], [-4, 1, -2], [1, -2, -4])`

  Nun wird eine Liste von Tuples von Listen erstellt (Z.30)

  nämlich eine Liste von 9 Teilchen, das jeweils 3 Rotationen enthält

  Ich nenne die puzzles.

  Nun notieren wir die Position der Seiten des Teilchens wie folgt:



- Z. 23 Die nächste Submethode, combine, nimmt jetzt 4 Parameter als Übergabe:
    - list1: eine Rotation (z.B. [-2, -4, 1]). Wichtig ist, dass diese Rotation einem anderen Teilchen als list_list2 gehört.
    - list_list2: ein Teilchen (z.B. ([-3, -2, -1], [-2, -1, -3], [-1, -3, -2]))
    - pos1, pos2: jeweils die Position der Seiten der dreieckigen Teilchen, die aneinander verbunden werden.

  Die Methode überprüft dann, ob list1 zu irgendeiner Rotation in list_list2 zusammenpasst, natürlich in ausgewählten Positionen, gibt die Rotation in list_list2 wieder.
  Beispiel:

```
>> combine([-2, -4, 1],([-3, -2, -1], [-2, -1, -3],[-1, -3, -2]),2,1)
>> [[-2, -1, -3]]     # da[-2, -4, 1][2]+[-2, -1, -3][1] = 0
```

  aber:

  >> combine([-2, -4, 1], ([-3, -2, -1], [-2, -1, -3], [-1, -3, -2]), 0, 2)
  >> [ ]

- Z. 47: Die Methode dreieck_puzzle: nimmt puzzles als Übergabe
    - in dieser Methode bezieht sich piece auf eine einzige Liste (z.B. [-2, -4, 1]), und dessen Plural, pieces, bedeutet eine Liste von mehreren piece (z.B. [[-3, -2, -1], [-2, -1, -3], [-1, -3, -2]]).

    1. Schritt: in der ersten for-Schleife werden die 9 tuples in puzzles iteriert.
       first_pieces ist beispielsweise ([-3, -2, -1], [-2, -1, -3], [-1, -3, -2]).
       sub_list1 enthält die 8 übrig gebliebenen tuples.
       Nun werden die Elemente in first_pieces iteriert: das erste piece ist fixiert.

2. Schritt: wie schon angedeutet wurde, enthält sub_list1 die 8 übrig gebliebenen tuples. Hier werden sie iteriert.
   Nun verwenden wir die combine Methode und bekommen second_pieces, welche enthält alle mögliche Teilchen, die zum fixierten first_piece zusammenpasst.
   sub_list2 sind wiederum die 7 übrig gebliebenen tuples.

3. diese Vorgehensweise wird solange wiederholt, bis wir 8 pieces haben.
   Das letzte Teil soll jetzt nur noch an zwei fixierten Seiten noch übereinstimmen.

- Ich erkläre folgendermaßen, wie die Teilchen aneinander angehängt werden sollen, und warum die Position der Seiten wichtig sind.

```
Die Teilchen(1st_piece - last_piece) sind in folgender
Reihenfolge angeordnet:

            /\
           / 1\
          /____\
         /\    /\
        /3 \ 2/ 4\
       /____\/____\
      /\    /\    /\
     /7 \ 5/ 9\ 6/ 8\
    /____\/____\/____\
```

Wir haben zuerst das first_piece:

```
            /\
           / 1\
          /____\
```

an dessen Unterseite soll second_piece anknüpfen, also an first_piece[2], wiederum ist die Verbindung an der Oberseite des second_piece, daher die Position 1 des second_piece.

Dementsprechend, als second_piece fixiert ist, soll second_piece[0] third_piece angehängt werden, und zwar bei third_piece[1].

Anmerkung:

Diese Aufgabe hat mich am längsten gekostet und zugleich auch am meisten Spaß gemacht. Es fällt natürlich auf, dass irgendeine Methode mit 18 for-Schleifen weder schön noch effizient noch überschaubar noch debug-leicht ist. Das Problem liegt wesentlich darin, dass ich die Daten unter Sublisten und Listen untergeordnet habe und dies verursacht eine komplizierte Verschachtelungen, als ich die Elemente miteinander vergleichen möchte. Als Verbesserungen kann selbstverständlich eine spontane permutate-Liste vor Ort aufgerufen werden, anstatt alles vorher in einer reinzustopfen, oder ich könnte eine elegantere combine-Methode schreiben, in der gleichzeitig 2 for-Schleifen durchlaufen. Dies könnte hoffentlich die Anzahl der for-Schleifen in der Hauptmethode bis zum 9 reduzieren, da immerhin 9 Teilchen vorhanden sind. Es wäre allerdings feiner, ganz ohne for Schleife zu arbeiten und dazu ungefähr 20 Parameter in die Methode übergeben, oder mit while-Schleifen die Arbeit zu vereinfachen, obwohl ich noch keine klare Vorstellung davon habe. Da ich zeitlich mit der Schule sehr gebunden bin, konnte ich leider wenige Zeit dafür investieren, was ich unendlich schade finde.