

Junioraufgabe 1: Auf und Ab

Team-ID: 00133

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker (15 Jahre)

6. Oktober 2018

Inhalt

Lösungsidee	2
Umsetzung.....	2
Anmerkung zur Komplexität dieser Lösung	3
Beispiel	3
Quellcode	4
Bedeutung des Skripts.....	4
class LadderGameSimulator.....	5
enum GameState	5
LADDERS.....	5
__init__(players)	5
self.players	5
self.is_finished.....	5
reset().....	5
do_move(player_num, fields).....	5
won(player)	6
class LadderGamePlayer	6
self.name.....	6
self.position	6
move(fields)	6
set_position(value).....	6
reset(game_manager).....	7
Das Hauptprogramm.....	7

Lösungsidee

Immer, wenn ein Spieler bewegt wird, muss seine Position gemäß zwei Dingen verändert werden: Endet sein Zug auf einem Leiterfeld, wird die Position entsprechend verändert und wenn er die 100 überschreitet, muss er die verbleibende Anzahl der Felder wieder zurückgehen.

Sobald einer der Spieler, die immer um die gleiche Augenzahl laufen, auf ein Feld kommt, das dieser bereits besucht hat, so kann er das Spiel nicht gewinnen, da nun alles wieder von diesem Feld aus von vorne beginnen würde, solange bis er wieder auf diesem Feld steht. Also wird nach dem Zug geprüft, ob der Spieler dieses Feld bereits betreten hat. Wenn ja, dann kann er nicht ans Ziel kommen. Ansonsten läuft das Ausprobieren für einen Spieler solange, bis er das Feld 100 erreicht.

Umsetzung

Das Programm ist in Python 3 geschrieben.

Die Datei `laddergame.py` enthält eine Klasse `LadderGameSimulator` und eine Klasse `LadderGamePlayer`. `LadderGameSimulator` speichert alle Leitern in Form eines `dicts`, in dem alle Paare in beide Richtungen gespeichert werden. Um diesen Prozess beim Programmieren zu vereinfachen, existiert das Klassenattribut `ladder_pairs`, einem `tuple`, der aus weiteren zweielementigen Tupeln besteht, die je eine Leiter speichern. Das eigentliche `dict` als Klassenattribut `LADDERS` wird durch eine Dict Comprehension erzeugt (siehe Quellcode). Die Klasse `LadderGameSimulator` bewegt mit der Methode `do_move` einen Spieler um eine gegebene Augenzahl, indem die Methode `LadderGamePlayer.set_position` mit der theoretischen neuen Position aufgerufen wird.

In dieser Methode spielt sich der eigentlich interessante Teil ab: Sollte die theoretische Position größer als 100 sein, wird die Position zuerst korrigiert, indem sie auf die Differenz von 200 und der theoretischen Position gesetzt wird. Damit steht der Spieler soweit schon mal auf dem richtigen Feld und die neue Position wird übernommen und der Zähler `LadderGamePlayer.moves`, der die Züge zählt, um 1 erhöht. Jeder Spieler speichert im Attribut `done_fields` ein `set`, das alle bereits betretenen Felder enthält. Wenn der Spieler auf einem Leiterfeld steht, wird die Position nochmal verändert und er steht jetzt auf dem anderen Leiterende. Erst jetzt wird überprüft, ob sich der Spieler auf einem bereits betretenen Feld befindet. Wenn ja, wird dies per Rückgabewert an die Methode `do_move` weitergeleitet. Bei einem solchen Leitersprung darf natürlich immer nur das eine Ende als bereits betretenes Feld gespeichert werden, weil in beide Richtungen ja unterschiedliche Wege entstehen. Also wird auch nur das Feld hinzugefügt, auf dem der Spieler sich nach dem Leitersprung befindet und umgekehrt auch nur das Feld überprüft.

Sollte die Position nun 100 betragen, wird das auch per Rückgabewert an die Methode `do_move` weitergegeben, die dann alles Weitere tun kann.

Um eine Ausgabe zu erzeugen, wenn ein Spieler gewonnen hat, ruft `do_move` noch die Methode `LadderGameSimulator.won` auf, die eine entsprechende Nachricht anzeigt.

Um das Spiel für alle Augenzahlen einmal durchlaufen zu lassen, wird mit einer `for`-Schleife durch alle Zahlen von 1 bis 6 iteriert, um für jede ein neues Spiel zu beginnen, solange, bis der Spieler entweder gewonnen hat oder nicht ans Ziel kommen kann. Nach zurücksetzen des Spiels mittels `LadderGameSimulator.reset` ist dann die nächste Augenzahl dran.

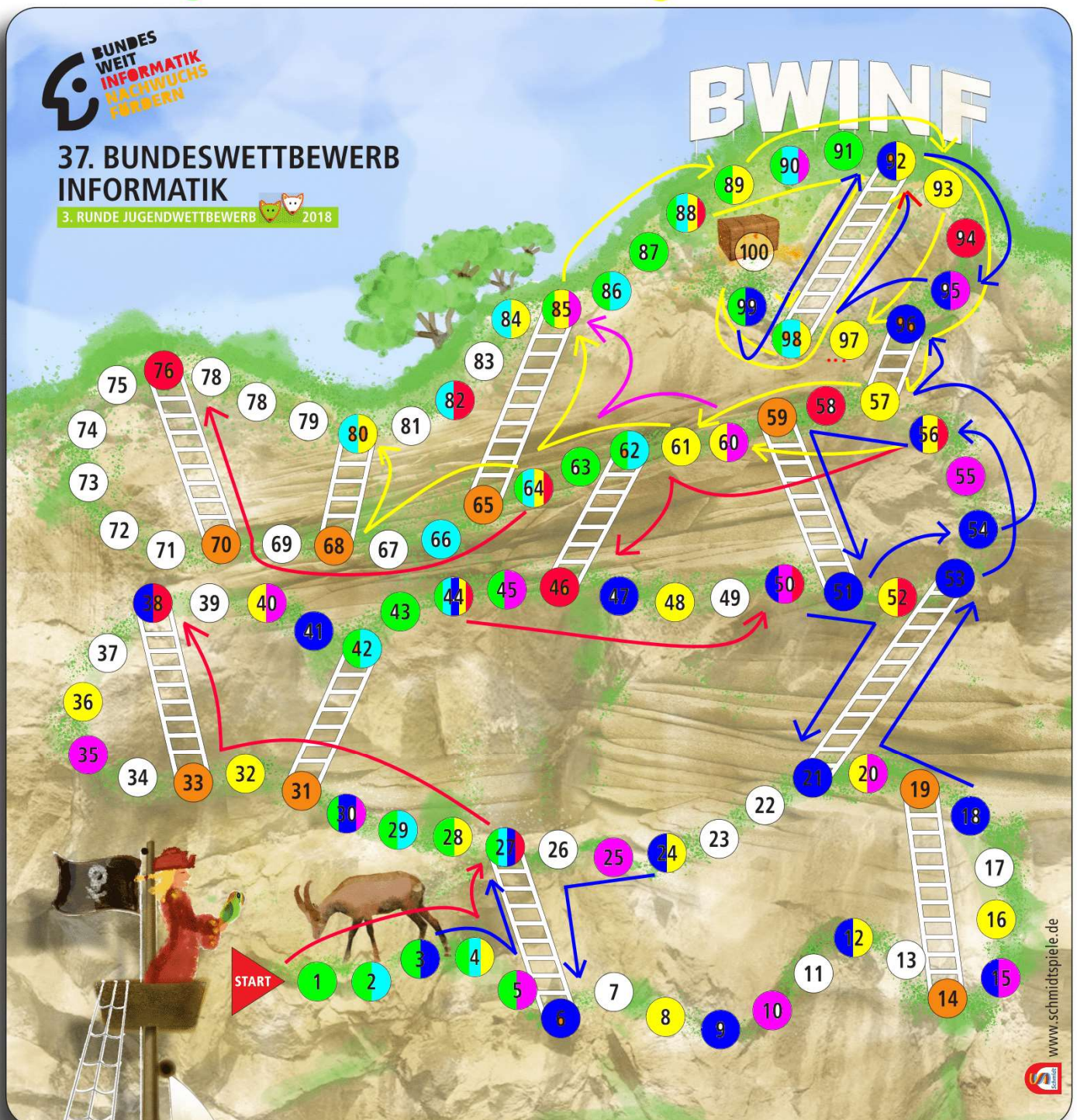
Anmerkung zur Komplexität dieser Lösung

Dies ist eine sehr komplexe Lösung. Mir ist natürlich bewusst, dass man das Programm auch ohne Klassen in einer einzelnen Funktion (oder vielleicht noch nicht mal das) schreiben könnte. Ich wollte möglichst erweiterbare Klassen bauen, die auch zur Modularisierung geeignet sind. Dies ist ein wichtiges Konzept in der Programmierung, und, um ehrlich zu sein, hatte ich auch Spaß daran, das Ganze ein bisschen komplexer und strukturierter zu gestalten.

Beispiel








Zum Beispiel gibt es eigentlich nicht viel zu sagen. Wie erwartet erreicht man mit Augenzahl 1 zwar das Ziel, aber auch am langsamsten. Hier ist ein Bild, in dem die Wege mit den jeweiligen Augenzahlen veranschaulicht. Pfeile werden verwendet, um komplexe Situationen darzustellen, es wird immer nur das Feld nach einem Leitersprung (also am Ende des Zuges) markiert

1 grün 2 hellblau 3 blau 4 gelb 5 rot 6 lila



Die Wege mit den Augenzahlen 3 und 4 erreichen nicht das Ziel, daher sind sie am Ende auch entsprechen komplex mit vielen Feldern. Bei Augenzahl 4 hört der gelbe Weg an der 97 mit drei roten Punkten auf, eigentlich gäbe es noch 2 Züge, die dann wieder auf der 97 enden. Der Pfeil mit dem roten Ende zur 92 kennzeichnet bei 3 Augenzahlen das Ende, da dieses Feld bereits betreten wurde.

Die folgende Tabelle gibt noch einen Überblick und einige Informationen, was bei den Augenzahlen auffällig ist.

	Ziel erreicht?	Züge	Informationen
	Ja	26	Mit nur Einsen als Augenzahl erreicht man immer das Ziel, weil es durch das Überspringen nie Leitern gibt, die wieder zurückführen. Mit Leitern, die zurückführen, würde das nicht funktionieren
	Ja	17	100 lässt sich glatt durch 2 teilen, deshalb dürfen alle Leitern auch insgesamt nur eine durch 2 glatt teilbare Zahl als Summe der übersprungenen Felder haben, was hier der Fall ist: $(27 - 6) + (42 - 31) + (62 - 46) + (80 - 68) + (98 - 92) = 66$
	Nein	24	Die 3 verfängt sich in einer Schleife, die von der 92 zur 98 und wieder zur 92 geht. Ähnlich den Augenzahlen 4, 5 und 6 passt die Differenz der Leitern hier aber nicht.
	Nein	29	Die 4 verfängt sich, sobald die Figur auf der 97 steht. Innerhalb von zwei Zügen (nach dem „Abprallen“ von der 100) steht der Spieler wieder auf der 97. Auf die 97 kommt der Spieler aber auch nur, weil er durch die Leitern eine gewisse Differenz hinzuerhält. Ansonsten würde er auf durch 4 teilbaren Zahlen bleiben.
	Ja	16	Die 100 ist durch 5 glatt teilbar, und deshalb erreicht man sie auch ohne irgendwelches „Abprallen“ von der 100. Auch der einzige Leitersprung ändert nichts, weil dieser eine Differenz von $85 - 65 = 20$ ausmacht, was durch 5 glatt teilbar ist.
	Ja	14	Die 100 ist eigentlich nicht glatt durch 6 teilbar, aber durch die Leitersprünge, die sich zu $(27 - 6) + (38 - 33) + (46 - 62) + (76 - 70) = 16$ addieren, was Modulo 6 gleich 4 ist, ergibt das nur noch 96, was durch 6 glatt teilbar ist ($96 \div 6 = 16$). Man beachte, dass bei der Leiter von 46 zu 62 nicht die kleinere von der größeren Zahl abgezogen wurde, da die Leiter hier in die andere Richtung genommen wird und sich die Leiterdifferenz aus 46 und 62 so subtrahieren muss.

Quellcode

Bedeutung des Skripts

```
python3 Junioraufgabe1/laddergame.py
```

Stellt die Klassen `LadderGameSimulator` und `LadderGamePlayer` zur Verfügung. Wenn das Modul direkt aufgerufen wird, wird ein Spiel für jede Augenzahl simuliert und die Schritte werden ausgegeben.

class LadderGameSimulator

Diese Klasse simuliert das Leiterspiel. Die Klasse ist so aufgebaut, dass theoretisch auch eine grundlegende Struktur für komplexere Spiele existiert, unter anderem, damit die Klasse hier komfortabel benutzt werden kann.

enum GameState

Ein Enum, um verschiedene Zustände zu speichern, mit denen die Methoden miteinander kommunizieren können.

```
class GameState(enum.Enum):
    SameField = 0
    Valid = 1
    Won = 2
```

SameField wird zurückgegeben, wenn der Spieler auf ein bereits betretenes Feld kommt, Valid, wenn alles in Ordnung ist, und Won, wenn der Spieler gewonnen hat.

LADDERS

Dieses Klassenattribut speichert alle Leitern in beide Richtungen als dict. Um die Leitern einfacher eingeben zu können, werden sie mit einer Dict Comprehension aus dem Klassenattribut ladder_pairs umgewandelt:

```
ladder_pairs = (( 6, 27), (14, 19), (21, 53), (31, 42), (33, 38), (46, 62),
                (51, 59), (57, 96), (65, 85), (68, 80), (70, 76), (92, 98))
LADDERS = {i: j for i, j in (ladder_pairs +
                             tuple((y, x) for x, y in ladder_pairs))}
```

Die Variable ladder_pairs enthält alle Leitern nur in eine Richtung. Deshalb wird sowohl durch das Tupel in der einen Richtung als auch in der anderen Richtung iteriert. tuple((y, x) for x, y in ladder_pairs) dreht die Elemente des ursprünglichen Tupels alle um. Dadurch enthält das fertige dict beide Varianten in beide Richtungen.

__init__(players)

Erstellt eine neue Klasse mit self.players = players

self.players

Speichert alle Spieler dieses Spiels in einer Liste mit LadderGamePlayer-Objekten.

self.is_finished

True, wenn ein Spieler schon gewonnen hat, ansonsten False.

reset()

Setzt das Spiel und jeden Spieler in self.players mit der Methode LadderGamePlayer.reset zurück.

do_move(player_num, fields)

Führt einen Zug des Spielers mit dem Index player_num in der Liste self.players mit fields als Augenzahl aus, indem die Methode move des Spielerobjekts aufgerufen wird. Gibt einen GameState aus dem Enum zurück, je nach Situation. Die Funktion tut nichts, wenn self.is_finished True ist. Ansonsten wird die move-Methode des entsprechenden LadderGamePlayer-Objekts aufgerufen. Hat der Spieler gewonnen, wird zusätzlich self.won mit dem Spielerobjekt als Argument aufgerufen.

```
def do_move(self, player_num, fields):
    if self.is_finished:
        return
    player = self.players[player_num]
    code = player.move(fields)
    if code == self.SameField:
        # der Spieler steht auf einem bereits betretenen Feld
        return self.SameField
    elif code == self.Won:
        # der Spieler ist am Ziel angekommen
        self.won(player)
        return self.Won
```

won(player)

Teilt dem Simulator mit, dass der Spieler `player` gewonnen hat. `self.is_finished` wird auf `True` gesetzt und eine Nachricht mit dem Spielernamen (`player.name`), der das Ziel erreicht hat, ausgegeben.

class LadderGamePlayer

Stellt einen Spieler des Leiterspiels dar

self.name

Speichert den Namen des Spielers als String.

self.position

Speichert die aktuelle Position auf dem Spielplan als `int`. Kann von außen nur gelesen werden; es existiert aber das Attribut `_position`, das nur intern verwendet wird. Die Position sollte von außen nur über `move` bzw. `set_position` verändert werden.

move(fields)

Bewegt den Spieler mit `self.set_position` um `fields` Felder nach vorne. Gibt einen `GameState` der Klasse `LadderGameSimulator` zurück, um über den aktuellen Status zu informieren.

set_position(value)

Setzt die aktuelle Position auf `value` gemäß den unter „Lösungsidee“ beschriebenen Schritten.

```
def set_position(self, value):
    # Zähler um 1 erhöhen
    self.moves += 1
    print(str(self.moves) + ") move to", value)
    # Prüfen, ob die theoretische Position das Ziel überschreitet
    if value > 100:
        # entsprechend Felder zurückgehen
        self._position = 200 - value
        print(" Ziel nicht exakt erreicht. Gehe zurück auf", self._position)
    else:
        # ansonsten gewöhnlich auf das übergebene Feld gehen
        self._position = value

    # prüfen, ob das aktuelle Feld zu einer Leiter gehört
    # (self.game_manager ist der Simulator, zu dem der Spieler gehört)
    if self._position in self.game_manager.LADDERS:
        # Zum anderen Leiterende gehen
        self._position = self.game_manager.LADDERS[self._position]
        print(" jumping to", self._position)
```

```

# prüfen, ob der Spieler auf einem bereits betretenen Feld steht
if self._position in self.done_fields:
    print("Bereits betreten")
    return LadderGameSimulator.SameField
# prüfen, ob der Spieler gewonnen hat
if self._position == 100:
    return LadderGameSimulator.Won
# Das Spiel ist nicht beendet/abgebrochen,
# also aktuelle Position zu den bereits betretenen hinzufügen
self.done_fields.add(self._position)
return LadderGameSimulator.Valid

```

`reset(game_manager)`

Wird vom Simulator genutzt, um den Spieler zurückzusetzen (Position, Anzahl der Züge, betretene Felder) und um dem Spieler den zugehörigen Simulator als `game_manager` mitzuteilen.

Das Hauptprogramm

Hier wird der Quellcode beschrieben, der alle Spiele für die Augenzahlen von 1 bis 6 einmal simuliert. Hier ist erstmal der Code (er befindet sich am Ende der `laddergame.py` Datei):

```

if __name__ == "__main__":
    player = LadderGamePlayer()
    game = LadderGameSimulator((player,))
    for spots in range(1, 7):
        game.reset()
        player.name = "Spieler" + str(spots)
        print("\n#####\nVersuche Augenzahl", spots)
        while True:
            code = game.do_move(0, spots)
            if code == game.SameField:
                print("Auf diesem Feld war der Spieler schon...")
                print("Spieler kann nicht gewinnen")
                break
            elif code == game.Won:
                break

```

Dieser Teil wird nur ausgeführt, wenn das Skript nicht als Modul benutzt wird (Zeile 1).

Es werden zunächst ein Spielerobjekt und ein Simulator-Objekt, dem der Spieler übergeben wird, erstellt. Der Teil in der folgende `for`-Schleife wird 6-mal aufgerufen, mit den Augenzahlen `spots` von 1 bis 6 jeweils einmal. Nach dem Zurücksetzen des Spiels, also auch der Spielerposition, wird der der Spielernamen gemäß der Augenzahl geändert (nicht unbedingt wichtig). In der folgenden Schleife wird solange der erste (und einzige) Spieler des Simulators um die Augenzahl bewegt (die `0` steht für den ersten Spieler), bis dieser entweder gewonnen hat (dann ist die Ausgabe bereits über `LadderGameSimulator.won` erfolgt) oder sich jetzt auf einem bereits betretenen Feld befindet (dann wird die Ausgabe hier im Hauptprogramm gemacht, dass ein Spieler sich auf einem bereits betretenen Feld befindet, ist für den Simulator ja eigentlich nicht relevant). Danach ist die nächste Augenzahl an der Reihe.