Aufgabenblatt 5

Allgemeine Informationen zum Aufgabenblatt:

- Die Abgabe erfolgt in TUWEL. Bitte laden Sie Ihre Python-Datei bis spätestens **Donnerstag, 15.05. 16:00 Uhr** in TUWEL hoch.
- Beachten Sie bitte folgende Punkte:
  - Die Programme müssen syntaktisch korrekt sein. Achten Sie außerdem darauf, dass die Beispiele aus der Angabe zu keinen Abstürzen führen.
  - Wenn Fehler auftreten ziehen wir abhängig von der Schwere der Fehler Punkte ab. Bei entsprechend kleinen Fehlern können auch gar keine Punkte abgezogen werden. Geben Sie daher die bestmögliche eigene Lösung ab, auch wenn diese nicht vollständig richtig ist.
  - Ihre Programme sollen demonstrieren, dass Sie die Themen des Aufgabenblattes beherrschen. Verwenden sie daher keine "Abkürzungen" und beschränken Sie sich auf die Konstrukte die in der Vorlesung vorgestellt wurden.
  - Um das Testen einzelner Aufgaben zu erleichtern sollen alle Aufgaben als entsprechende Funktionen implementiert werden. Bitte achten Sie darauf, dass Sie in der Abgabedatei keinen Code außerhalb der in den jeweiligen Aufgaben definierten Funktionen erstellen. Sie finden entsprechende Kommentare in der Angabedatei.
  - Bei den meisten Angaben sind explizite Annahmen definiert. Sie können davon ausgehen, dass diese Annahmen stets eingehalten werden. D.h. Sie müssen deren Gültigkeit weder überprüfen, noch muss Ihre Implementierung für Werte welche diese Annahmen nicht erfüllen funktionieren.
  - Type hints: Zur besseren Beschreibung und Dokumentation der verwendeten Funktionen verwenden wir ab diesem Aufgabenblatt sogenannte Type-Hints in den Funktionsköpfen. Diese beschreiben die intendierten Typen der Parameter und Rückgabewerte. Die von uns verwendeten Type Hints stehen erst ab **Python 3.10** zur Verfügung.
  - Das Aufgabenblatt enthält 4 Aufgaben, auf welche Sie insgesamt 5 Punkte erhalten können.

In diesem Aufgabenblatt werden folgende Themen behandelt:

- Rekursion (insbesondere über Listen)
- Kollektionen (Dictionaries, (verschachtelte) Listen)
- Testen
- Ausnahmen (Exceptions)

Aufgabe 1 (Rekursion (über Sequenzen)) [1 Punkt]

In dieser Aufgabe trainieren Sie die Verwendung rekursiver Lösungsstrategien mit Sequenzen.
Verwenden Sie zur Umsetzung der angegebenen Funktionen **keine** Schleifen, globalen Variablen, List/Tuple/...-Comprehensions oder zusätzliche Funktionen. Dies betrifft sowohl in Python verfügbare Funktionen als auch eigene Funktionen. Auch die Funktionsköpfe dürfen nicht erweitert oder verändert werden.
Ausgenommen von diesen Verboten sind `len` und `in`, welche beide verwendet werden dürfen. Auch die Verwendung von Slicing ist erlaubt.
Testen Sie Ihre Implementierung zumindest mit den angegebenen Aufrufen, indem Sie die Ergebnisse der Aufrufe in der Konsole ausgeben. Verwenden Sie dazu die Funktion `exercise1()`, d.h. geben Sie die Aufrufe und Ausgaben in dieser Funktion an.
Implementieren Sie die folgenden Funktionen:

1. `delete_direct_repeats(values: list) -> list` gibt eine neue Liste zurück, deren Inhalt aus `values` erzeugt wird, indem die Werte aus `values` in der bestehenden Reihenfolge übernommen werden. Von direkt hintereinander stehenden Kopien des selben Wertes wird immer nur ein einzelnes Vorkommen des Wertes in die neue Liste übernommen. So wird aus `[1, 2, 2, 3, 3, 3, 2, 2]` z.B. die Liste `[1, 2, 3, 2]`.

   Annahmen: `values` ist eine Liste.

   Beispielaufrufe und erwartete Rückgaben:

   `delete_direct_repeats([1, 2, 2, 4, 3, 3, 3, 2])` gibt `[1, 2, 4, 3, 2]` zurück

   `delete_direct_repeats(['a', 'a', 'a'])` gibt `['a']` zurück

   `delete_direct_repeats([7, 7, 4, 7, 7, 7])` gibt `[7, 4, 7]` zurück

   `delete_direct_repeats([4.0])` gibt `[4.0]` zurück

   `delete_direct_repeats([[1], [], []])` gibt `[[1], []]` zurück

2. `balance(load: list[int|float], begin: int, end: int) -> None` ändert in `load` die Werte an den Indizes im Intervall `[begin, end]`:

   Es werden die Indizes von `load` im Intervall `[begin, end]` betrachtet, und jeweils die erste mit der letzten Stelle verglichen, die zweite mit der vorletzten Stelle, usw. Beiden Stellen wird jeweils der Durchschnitt der beiden Werte an diesen Stellen zugewiesen. Anders ausgedrückt: um die Werte an diesen Stellen auszugleichen, wird immer die Hälfte der Differenz zwischen den beiden Werten vom größeren Wert abgezogen und zum kleineren Wert dazuaddiert.

   Die Funktion hat keine Rückgabe.

   Hinweis: Je nach Implementierung kann bei Teillisten ungerader Länge ein `int`-Wert in der Mitte der Liste ein `int` bleiben oder zu einem äquivalenten `float` werden.

   Annahmen: `load` ist eine Liste von Zahlen (`float` sowie `int`), und es gilt `0 <= begin < end < len(load)`.

   Beispielaufrufe und erwartete Rückgaben:

   Für `load1=[0, 4]`: Nach `balance(load1, 0, 1)` gilt `load1 == [2.0, 2.0]`

   Für `load2=[0, -3, 1]`: Nach `balance(load2, 0, 2)` gilt `load2 == [0.5, -3, 0.5]`

Für `load3=[1, 2, 9, 10]`: Nach `balance(load3, 0, 3)` gilt `load3 == [5.5, 5.5, 5.5, 5.5]`

Für `load4=[1, 2, 9, 10]`: Nach `balance(load4, 0, 2)` gilt `load4 == [5.0, 2, 5.0, 10]`

Für `load5=[0, 9, 9]`: Nach `balance(load5, 1, 1)` gilt `load5 == [0, 9, 9]`

3. `collect(elements: list[tuple[str, int]], position: int, forbidden: set[int]) -> tuple[str, int]` durchläuft die Liste `elements` nach folgender Regel:

   Die Einträge in `elements` sind Paare (*value*, *next*) aus einem String *value* und einer ganzen Zahl *next*. Vom Index `position` ausgehend wird jeweils der Index *next* als nächstes besucht. Das Durchlaufen der Liste endet, sobald versucht wird einen nicht existierenden Index zu besuchen, ein Index zum zweiten mal besucht wird, oder ein Index in `forbidden` besucht wird.

   Die Funktion gibt ein zweistelliges Tupel (*text*, *count*) zurück. Dabei ist *text* die durch Beistrich getrennte Verkettung der Strings *value* in den besuchten Einträgen – in der Reihenfolge, in der sie besucht wurden – und *count* die Anzahl der verketteten Einträge.

   Die Liste `elements` darf **nicht verändert werden**!

   Annahmen: `elements` ist eine Liste von Paaren von ganzen Zahlen, `position` ist eine ganze Zahl, und `forbidden` eine Menge.

   Beispielaufrufe und erwartete Rückgaben:

   `collect([('Welt', 1), ('Hallo', 2), ('schnöde', 0), ('!', 4)], 1, set())`
   gibt `('Hallo,schnöde,Welt', 3)` zurück

   `collect([('Welt', 1), ('Hallo', 2), ('schnöde', 0), ('!', 4)], 1, {0})`
   gibt `('Hallo,schnöde', 2)` zurück

   `collect([('Welt', 1), ('Hallo', 2), ('schnöde', 0), ('!', 4)], 0, {0})`
   gibt `('', 0)` zurück

   `collect([('a,b,c', 0), ('b', 3), ('a', 1), ('c', 4)], 2, set())`
   gibt `('a,b,c', 3)` zurück

   `collect([('a,b,c', 0), ('b', 3), ('a', 1), ('c', 4)], 0, set())`
   gibt `('a,b,c', 1)` zurück

Aufgabe 2 (Kollektionen (Dictionaries)) [1 Punkt]

Ziel dieser Aufgabe ist es, die Struktur von und das Arbeiten mit Dictionaries kennenzulernen, und vertrauter im Umgang mit grundlegenden Funktionen rund um Dictionaries zu werden.
Sie dürfen zur Lösung der Aufgaben alle in der Vorlesung behandelten Sprachkonstrukte verwenden (beachten Sie vor allem die Folien "Wichtige Operationen auf Dictionaries"), sowie die in Python verfügbaren Funktionen für Kollektionen. Achten Sie darauf, dass Ihre Erweiterungen nicht nur für die konkreten gegebenen Dictionaries funktionieren, sondern auch für andere Dictionaries mit derselben Struktur, aber anderen Werten.

1. In der Angabedatei `blatt5.py` sind in der Funktion `fun_with_rivers()` zwei Dictionaries `country_codes` und `rivers` definiert. Das Dictionary `country_codes` enthält zu einigen dreistelligen ISO-Ländercodes gängige Bezeichnungen des jeweiligen Landes. Das Dictionary `rivers` enthält zu einer Auswahl an Flüssen jeweils eine Liste der Länder (als dreistellige Ländercodes) durch die der Fluss fließt.

   Erzeugen Sie aus diesen Datenstrukturen folgende Informationen (die Dictionaries dürfen nicht verändert werden). Implementieren Sie Ihre Lösungen innerhalb der Funktion `fun_with_rivers()`.

   - Ermitteln Sie die Namen aller Länder in `country_codes`, die auf `'en'` enden. Geben Sie die Namen alphabetisch aufsteigend sortiert in einer Zeile durch `', '` getrennt aus.

     Erwartete Ausgabe:

     ```
     Bulgarien, Italien, Polen
     ```

   - Gesucht sind alle Flüsse in `rivers`, die durch mehr als drei Länder fließen. Geben Sie jeden solchen Fluss in einer eigenen Zeile aus. Die Zeile soll den Namen des Flusses enthalten, sowie nach einem Abstand in runden Klammern die Anzahl der Länder durch die der Fluss fließt.

     Erwartete Ausgabe (Reihenfolge beliebig):

     ```
     Donau (10)
     Rhein (6)
     ```

   - Geben Sie eine Auflistung aller Länder aus, durch welche die Donau fließt. Falls für ein Land der Name in `country_codes` definiert ist, verwenden Sie diesen Namen. Geben Sie ansonsten `'(???)'` statt des Namens aus.

     Geben Sie die Länder in einer Zeile durch `' -> '` getrennt in der Reihenfolge aus, wie sie in `rivers` angegeben sind.

     Erwartete Ausgabe (in einer Zeile, hier aus Platzgründen mehrzeilig):

     ```
     Deutschland -> Österreich -> Slowakei -> (???) -> (???) -> (???) ->
     Bulgarien -> (???) -> (???) -> Ukraine
     ```

   - Geben Sie an, durch wie viele *verschiedene* Länder die in `rivers` genannten Flüsse insgesamt fließen, sowie wie viele dieser Länder in `country_codes` definiert werden.

     Erwartete Ausgabe:

     ```
     Die Flüsse fließen durch 17 verschiedene Länder.
     9 davon kennen wir.
     ```

- Geben Sie alle Flüsse in `rivers` an, welche durch Österreich fließen. Geben Sie die Flüsse durch `~~` getrennt in einer Zeile aus.

  Erwartete Ausgabe (Reihenfolge beliebig):

  `Donau~~Rhein~~Inn`

2. Für diese Unteraufgabe gehen wir zurück zur Funktion `top_donors(donations)` aus Aufgabenblatt 4 (Aufgabe 3.4). Implementieren Sie die beiden folgenden Funktionen.

   - Implementieren Sie die Funktion
     `top_donors(donations: list[tuple[str, tuple[int, ...]]]) -> dict[str, list[int]]`
     mit einem alternativen (vielleicht auch intuitiveren) Verhalten als in Aufgabenblatt 4:

     `donations` ist eine Liste[1] von zweistelligen Tupeln, welche an erster Stelle einen Namen (als String) und an zweiter Stelle wiederum ein Tupel enthalten. Dieses Tupel enthält eine unbestimmte Anzahl an Zahlen, welche Spendensummen darstellen sollen.

     Dieses mal erzeugt die Funktion ein Dictionary, welches einen Eintrag für jeden in `donations` vorkommenden Namen enthält (der Name ist der Key des Eintrags). Der zugehörige Wert ist eine Liste, welche alle diesem Namen zugeordneten Spendensummen enthält – diesesmal ohne Einschränkung der Häufigkeit oder Größe der Beträge. Die Funktion gibt das erzeugte Dictionary zurück.

     Enthält `donations` zum Beispiel die beiden Einträge `('Alice', (2, 3))` und `('Alice', (3, 10))`, dann enthält das erzeugte Dictionary den Eintrag `'Alice': [2, 3, 3, 10]`.

     Annahme: `donations` ist eine Kollektion mit dem zuvor beschriebenen Inhalt.

     Beispielaufrufe:

     ```
     top_donors([('Alice', (200, 300)), ('Bob', (10, 10, 10)), ('Eve', (20, 50)),
                 ('Alice', (1000,)), ('Bob', (300, 300))]) gibt
     {'Alice': [200, 300, 1000], 'Bob': [10, 10, 10, 300, 300], 'Eve': [20, 50]}
     ```
     zurück.

     ```
     top_donors([('GPA', (100, 100, 100)), ('GPA', (100, 100, 100)),
                 ('GPA', (100, 100, 101)), ('GPA', (150, 50, 101))]) gibt
     {'GPA': [100, 100, 100, 100, 100, 100, 100, 100, 101, 150, 50, 101]}
     ```
     zurück.

     `top_donors({('A', (100, 100)), ('B', (100,))}) gibt`
     `{'B': [100], 'A': [100, 100]}` zurück.

     `top_donors([('Dagobert', ())])` gibt `{'Dagobert': []}` zurück.

     `top_donors([])` gibt `{}` zurück.

[1] In Aufgabenblatt 4 konnte es eine beliebige Kollektion sein

- Implementieren Sie die Funktion
  `get_personal_total(aggregated_donations: dict, name: str) -> int`, welcher als Argument `aggregated_donations` Dictionaries der Form übergeben werden, wie sie `top_donors` erzeugt. Falls `name` ein existierender Key in `aggregated_donations` ist, gibt die Funktion die Summe der Spenden dieses Eintrags zurück. Beschreibt `name` keinen existierenden Eintrag wird -1 zurückgegeben.

  Annahme: `aggregated_donations` ist ein Dictionary der zuvor beschriebenen Struktur.

  Beispielaufrufe und erwartete Rückgaben:

  Für
  ```
  dict_a = {'Alice': [200, 300, 1000],
            'Bob': [10, 10, 10, 300, 300],
            'Eve': [20, 50]} und
  dict_b = {'GPA': [100, 100, 100, 100, 100, 100, 100, 100, 101, 150, 50, 101]}
  ```

  ```
  get_personal_total(dict_a, 'Alice')                          gibt 1500 zurück
  get_personal_total(dict_a, 'Bob')                             gibt 630 zurück
  get_personal_total(dict_b, 'Alice')                            gibt -1 zurück
  get_personal_total(top_donors([('Alice', (0, ))]), 'Alice')    gibt 0 zurück
  get_personal_total(dict(), 'Bob')                              gibt -1 zurück
  ```

Aufgabe 3 (Programme testen: Testfälle zur Qualitätssicherung) [1 Punkt]

Ziel dieser Aufgabe ist es, das systematische Entwickeln von Testfällen zu üben.
Anders als die bisherigen Aufgaben, ist die Lösung zu dieser Aufgabe **nicht** in der Datei `blatt5.py` abzugeben, sondern direkt in TUWEL. Im Abschnitt zu diesem Übungsblatt finden Sie die Test-Aktivität *Aufgabenblatt 5 - Aufgabe 3* (direkter Link: `https://tuwel.tuwien.ac.at/mod/quiz/view.php?id=2628152`). Dort finden Sie die weitere Aufgabenstellung.

Aufgabe 4 (Ein Spiel) [2 Punkte]

Ziel dieser Aufgabe ist es, Erfahrung im Lösen konkreter Problemstellungen mit Hilfe von Kollektionen (insbesondere verschachtelte Listen) sowie mit deren Anwendung zu sammeln, Programmabläufe zu erstellen, und Exceptions zur Ausnahmebehandlung einzusetzen. Dazu implementieren wir ein Spiel, wobei Teile der Funktionalität in der Angabedatei `blatt5.py` bereits vorgegeben sind.

Spielregeln: Das Spiel ist von Geschicklichkeitsspielen inspiriert, bei denen abwechselnd Spielsteine auf einer meist labilen Struktur platziert werden müssen. Fallen dabei Steine von der Struktur, müssen diese zurückgenommen werden, und das Ziel ist, die eigenen Steine so schnell wie möglich loszuwerden. Um den Umfang der Aufgabe im zeitlichen Rahmen zu halten, werden wir an einigen Stellen einige (teilweise auch unintuitive) Regeln aufstellen.
Das Spiel wird auf einem $m \times n$ großen Feld (m Zeilen, n Spalten) von zwei Personen gespielt. Beide Spieler:innen haben am Anfang die gleiche Anzahl an Spielsteinen. Diese legen sie nun abwechselnd auf ein noch unbesetztes Feld. Dabei reduziert sich die Anzahl der verfügbaren Steine der jeweiligen Person um 1. Nun kann es passieren, dass beim Ablegen des Steins das Brett so wackelt, dass bereits gesetzte Steine herunterfallen. Hier weichen wir vom typischen Ablauf solcher Spiele ab: wir weisen diese Steine nicht alle der Person zu, die gerade am Zug ist, sondern jede Person bekommt die Steine zurück, die sie auch gesetzt hat. Das Spiel endet, wenn eine Person keine Steine mehr auf der Hand hat, oder nach einer festgesetzten Anzahl von Runden – je nachdem was früher eintritt. Es gewinnt, wer zu diesem Zeitpunkt weniger Steine auf der Hand hat.

Umsetzung: Zum Verwalten des Spielfeldes verwenden wir eine Liste von Listen von Strings, wobei unbesetzte Felder den String `' '` enthalten und besetzte Felder die Markierung der jeweiligen Person: `'1'` für Spieler:in 1, und `'2'` für Spieler:in 2. Das "Wackeln" simulieren wir auf folgende Art und Weise: wir entscheiden nach jedem Zug, ob Steine herunterfallen. Wenn Steine herunterfallen, dann können sie das in zwei verschiedenen Mustern machen:

- Entweder es fallen alle Steine in einer Zeile,
- oder es fallen alle Steine in einer Spalte.

Die Idee ist, diese Entscheidungen (fallen überhaupt Steine herunter, und, falls ja, in welchem Muster und auf welchen Feldern) zufällig zu treffen - zum leichteren Testen und Vergleichen werden wir jedoch auch eine Möglichkeit vorsehen, das Verhalten explizit festzulegen.
Alle weiteren relevanten Details zur Implementierung müssen Sie aus der bereitgestellten, teilweisen Implementierung des Spiels ablesen oder selbst festlegen.
Zur besseren Übersicht, Wartbarkeit und Wiederverwendbarkeit wird die Implementierung dieses Spiels in mehrere Funktionen gegliedert. Zwei der Funktionen sind bereits vollständig vorhanden, zwei weitere Funktionen fehlen jedoch noch vollständig (nur ihr Funktionskopf und eine Dummy-Rückgabe sind bereits vorhanden), und zwei Funktionen – darunter die Hauptfunktion, welche das gesamte Spiel kontrolliert – sind teilweise vorhanden, aber noch nicht vollständig.

Vordefinierte Funktionen: Die beiden bereits vollständig implementierten Funktionen dürfen nicht verändert werden und bieten folgende Funktionalität:

- `print_board(board: list[list[str]]) -> None` gibt das in `board` repräsentierte Spielfeld auf der Konsole aus.

- `get_desaster(desasters: list[tuple], board: list[list[str]]) -> tuple` bestimmt, ob Steine herunterfallen – und falls ja, wo. Die Funktion gibt dazu eines der folgenden Tupel zurück:
  - `(False, )`, falls keine Steine herunterfallen.
  - `('row', <rid>)`, wenn alle Steine in der Zeile mit Index `<rid>` herunterfallen, wobei `<rid>` ein Index von `board` ist.
  - `('column', <cid>)`, wenn alle Steine in der Spalte mit Index `<cid>` herunterfallen, wobei `<cid>` ein Index von `board[0]` (und somit von jeder Zeile in `board`) ist.

  Die Funktion kann auf zwei Arten aufgerufen werden:
  - Entweder, um einen tatsächlich zufälligen Wert zu erhalten. Dazu muss für `desasters` der Wert `None` übergeben werden.
  - Die andere Möglichkeit ist, eine zuvor festgelegte Reihenfolge an Events abzuarbeiten. Dazu muss für `desasters` eine Liste übergeben werden. Ist diese Liste nicht leer, wird das erste Element der Liste darauf entfernt und von der Funktion zurückgegeben (jedes Tupel der Liste sollten daher einer der drei oben beschriebenen Formen entsprechen). Ist die Liste leer, liefert die Funktion `(False, )` zurück. Wir verwenden diese zweite Möglichkeit, um vorhersagbare Testfälle zu generieren.

Aufgabenstellung: Ihre Aufgabe ist es nun, das Programm fertig zu stellen:

- Ergänzen Sie die Funktion `player_input(player: str, board: list[list[str]]) -> tuple[int, int]` um eine Fehlerbehandlung.

  Die Funktion übernimmt das Abfragen einer Position von den Spieler:innen. Es wird die Zeile und Spalte des gewünschten Feldes abgefragt. Beschreibt die Eingabe ein noch nicht besetztes Feld am Spielfeld, dann werden die beiden Zahlen als Paar `(row, col)` zurückgegeben. Andernfalls wird eine `ValueError`-Exception ausgelöst.

  Erweitern Sie die Funktion um folgende Punkte:
  - Fangen Sie die Exception ab, welche erzeugt wird, wenn eine der beiden Eingaben keinen gültigen `int`-Wert darstellt. Lösen Sie eine neue `ValueError`-Exception aus. Übergeben Sie der Exception eine Nachricht die besagt, dass es sich bei der Eingabe um keine gültige Zahl gehandelt hat.
  - Nachdem zwei gültige `int`-Werte eingegeben wurden, wird überprüft, ob diese ein freies Feld am Spielfeld darstellen. Ist dies nicht der Fall, wird wiederum eine `ValueError`-Exception ausgelöst. Übergeben Sie auch dieser Exception eine Nachricht an die Spieler:innen, welche das Problem beschreibt.

    Unterscheiden Sie die folgenden drei Fehler:
    1. Keine gültige Zeile eingegeben: `row < 0` oder `row >= len(board)`
    2. Keine gültige Spalte eingegeben: `col < 0` oder `col >= len(board[row])`
    3. Feld ist schon belegt: Das Feld enthält nicht den String `' '`

Die konkrete Ausgestaltung der Nachrichten in den Exceptions ist Ihnen überlassen. Es sollte für die Spieler:innen aber klar werden, was das Problem ist. Wichtig: geben Sie diese Nachricht nicht direkt mit `print` aus, sondern übergeben Sie sie der Exception.

Beispiele für Eingaben und das erwartete Verhalten finden Sie am Ende der Aufgabenstellung.

- Vervollständigen Sie die Funktion `clear_row(board: list[list[str]], rid: int) -> tuple[int, int]`. Die Funktion setzt in `board` alle Einträge in der Zeile mit Index `rid` auf leer (`' '`). Die Rückgabe ist ein zweistelliges Tupel mit den aus der Zeile gelöschten Steinen von Spieler:in 1 an erster Stelle und jenen von Spieler:in 2 an zweiter Stelle.

  Annahmen: `board` ist eine Liste von Listen von Strings und `0 <= rid < len(board)`.

- Vervollständigen Sie die Funktion `clear_column(board: list[list[str]], cid: int) -> tuple[int, int]`. Die Funktion funktioniert analog zu `clear_row`, jedoch für die Spalte mit Index `cid`.

  Annahmen: `board` ist eine Liste von Listen von Strings, `len(board[0]) == len(board[i])` für alle Indizes `i` von `board`, `0 <= cid < len(board[0])`.

- Ergänzen Sie die Funktion `run_game(rows: int, columns: int, start_pieces: int, rounds: int, desasters: list)` so, dass sie das beschriebene Spiel implementiert. Dazu muss die gegebene Funktion um folgende Eigenschaften erweitert werden:

  – Bei einer falschen Eingabe erzeugt die Funktion `player_input` Ausnahmen, welche aktuell nicht abgefangen werden, sondern zum Absturz des Programms führen. Adaptieren Sie die Funktion `run_game` so, dass diese Ausnahmen abgefangen werden, eine entsprechende Fehlermeldung ausgegeben wird (verwenden Sie dabei die in den Exceptions enthaltenen Beschreibungen), und anschließend erneut zur Eingabe aufgefordert wird. Achten Sie darauf, dass bei einer falschen Eingabe weder der Rundenzähler erhöht wird, noch die Person wechselt, die gerade am Zug ist, und auch nicht überprüft wird, ob das Brett wackelt.

  Hinweis: all diese zusätzlichen Bedingungen lassen sich durch eine geschickt Wahl der `try`- und `except`- Abschnitte ohne weitere Änderungen sicherstellen.

  – Nachdem eine gültige Position in `player_input` eingegeben und zurückgegeben wurde, muss diese Position in `board` mit dem passenden Symbol markiert werden, und die Anzahl der Steine welche die Spielerin/der Spieler auf der Hand hält um eins verringert werden.

Beispielaufrufe und erwartete Ergebnisse: In der Datei `blatt5.py` zur Angabe finden Sie Testfälle (Eingaben und erwartete Ergebnisse) für die Funktionen `clear_row` und `clear_column`. Beachten Sie, dass die Ergebnisse jeweils auf dem Ergebnis der vorangegangenen Testfälle aufbauen, da die Funktionsaufrufe jeweils auf das selbe Spielfeld zugreifen – und dieses dabei verändern.

`player_input`:
Zum Testen dieser Funktion steht Ihnen die Funktion `test_player_input()` zur Verfügung. Dort wird ein Spielfeld `board_input` mit folgender Liste definiert:

```
[[' ', ' ', '1'],
 [' ', '2', '1'],
 ['2', '2', '2']]
```

Mögliche Abläufe bei unterschiedlichen Eingaben für den Aufruf `player_input('X', board_input)`:

```
Spieler X mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> a
```

Die Funktion erzeugt einen `ValueError`, und die Ausgabe von `test_player_input` wäre
`VALUE ERROR: Einer von uns beiden hat einen Fehler gemacht (und ich wars nicht) : kein gültiger Integer erkannt`

```
Spieler X mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 0
Welche Spalte? (0-2)> s
```

Die Funktion erzeugt einen `ValueError`, und die Ausgabe von `test_player_input` wäre
`VALUE ERROR: Einer von uns beiden hat einen Fehler gemacht (und ich wars nicht) : kein gültiger Integer erkannt`

```
Spieler X mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 2
Welche Spalte? (0-2)> 0
```

Die Funktion erzeugt einen `ValueError`, und die Ausgabe von `test_player_input` wäre
`VALUE ERROR: Uhhh, ja, ganz so einfach ist das Spiel dann auch nicht: auf dem Feld sitzt schon wer.`

```
Spieler X mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> -1
Welche Spalte? (0-2)> 2
```

Die Funktion erzeugt einen `ValueError`, und die Ausgabe von `test_player_input` wäre
`VALUE ERROR: Hihi, das ist aber keine gültige Zeile am Spielfeld!`

```
Spieler X mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 1
Welche Spalte? (0-2)> 3
```

Die Funktion erzeugt einen `ValueError`, und die Ausgabe von `test_player_input` wäre
`VALUE ERROR: Dududu, glaubst ich merke nicht, dass es die Spalte am Spielfeld nicht gibt?!`

```
Spieler X mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 0
Welche Spalte? (0-2)> 1
```

gibt das Tupel `(0, 1)` zurück.

`run_game`:
Im folgenden finden Sie ein Beispiel für einen Ablauf des Spiels für den Aufruf `run_game(3, 4, 3, 6, desasters_0)`, wobei `desaster_0` in der Datei `blatt5.py` gegeben ist, um einen reproduzierbaren Ablauf zu ermöglichen. In der Datei `blatt5.py` finden Sie auch die Ausgabe eines weiteren Durchlauf des Spiels. Bitte beachten Sie, dass es natürlich viele unterschiedliche Abläufe gibt, und Sie auch die Situation mit einem Unentschieden austesten sollten:

```
  0|1|2|3
---------
0: | | |
---------
1: | | |
---------
2: | | |
Spieler 1 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 1
Welche Spalte? (0-3)> 1
  0|1|2|3
---------
0: | | |
---------
1: |1| |
---------
2: | | |
## Steine übrig: Player 1: [2]  Player 2: [3]

Spieler 2 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 0
Welche Spalte? (0-3)> 2
  0|1|2|3
---------
0: | |2|
---------
1: |1| |
---------
2: | | |

======
 -- !! UHOH - Desaster struck: ('row', 0) !!

  0|1|2|3
---------
0: | | |
---------
1: |1| |
---------
2: | | |
```

```
## Steine übrig: Player 1: [2]  Player 2: [3]

Spieler 1 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 2
Welche Spalte? (0-3)> 3
  0|1|2|3
---------
0: | | |
---------
1: |1| |
---------
2: | | |1

======
 -- !! UHOH - Desaster struck: ('column', 1) !!

  0|1|2|3
---------
0: | | |
---------
1: | | |
---------
2: | | |1
## Steine übrig: Player 1: [2]  Player 2: [3]

Spieler 2 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 1
Welche Spalte? (0-3)> 0
  0|1|2|3
---------
0: | | |
---------
1:2| | |
---------
2: | | |1
## Steine übrig: Player 1: [2]  Player 2: [2]

Spieler 1 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 1
Welche Spalte? (0-3)> 3
  0|1|2|3
---------
0: | | |
---------
1:2| | |1
---------
2: | | |1
```

```
======
 -- !! UHOH - Desaster struck: ('row', 2) !!

  0|1|2|3
---------
0: | | |
---------
1:2| | |1
---------
2: | | |
## Steine übrig: Player 1: [2]  Player 2: [2]

Spieler 2 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 0
Welche Spalte? (0-3)> 2
  0|1|2|3
---------
0: | |2|
---------
1:2| | |1
---------
2: | | |
## Steine übrig: Player 1: [2]  Player 2: [1]

Spieler 1 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 2
Welche Spalte? (0-3)> 1
  0|1|2|3
---------
0: | |2|
---------
1:2| | |1
---------
2: |1| |
## Steine übrig: Player 1: [1]  Player 2: [1]

Spieler 2 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 2
Welche Spalte? (0-3)> 3
  0|1|2|3
---------
0: | |2|
---------
1:2| | |1
---------
2: |1| |2
```

```
======
 -- !! UHOH - Desaster struck: ('row', 2) !!

  0|1|2|3
---------
0: | |2|
---------
1:2| | |1
---------
2: | | |
## Steine übrig: Player 1: [2]  Player 2: [1]

Spieler 1 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 2
Welche Spalte? (0-3)> 0
  0|1|2|3
---------
0: | |2|
---------
1:2| | |1
---------
2:1| | |
## Steine übrig: Player 1: [1]  Player 2: [1]

Spieler 2 mit der ruhigen Hand, wohin mit der Figur?
Welche Zeile? (0-2)> 2
Welche Spalte? (0-3)> 2
  0|1|2|3
---------
0: | |2|
---------
1:2| | |1
---------
2:1| |2|
## Steine übrig: Player 1: [1]  Player 2: [0]


 #### SPIEL IST AUS ######
Player 2 hat gewonnen - die Letzten werden die Ersten sein.
```