Dokumentation Projektarbeit Python - Verlassene Raumstation

Inhaltsverzeichnis

Funktion & Aufbau

Dieser Code implementiert ein Minesweeper-Spiel in Python. Es handelt sich um eine Konsolen-Version, bei dem die Spieler*innen Felder auf einem quadratischen Spielfeld aufdecken können. Das Ziel ist es, alle sicheren Felder aufzudecken, ohne dabei auf eine Bombe zu stoßen. Falls die spielende Person eine Bombe trifft, endet das Spiel mit einer Niederlage. Andernfalls kann er oder sie weiterspielen, bis alle sicheren Felder aufgedeckt sind.

Das Spiel besteht aus fünf Hauptteilen:

1. Das erstellen des Spielfelds (`Board`-Klasse)

   > Die Funktion erstellt grundlegend das Spielfeld, dazu plaziert es die bomben und berechnet die Zahlen für nicht-bombige Felder.

   - `__init__(self, dim_size: int, num_bombs: int)`: Initialisiert das Spielfeld mit Bomben und Werten
   - `make_new_board(self)`: Erstellt ein neues Spielfeld und setzt Bomben zufällig
   - `assign_values_to_board(self)`: Berechnet die Anzahl der angegrenzenden Bomben für jedes Feld
   - `get_num_neighboring_bombs(self, row: int, col: int)`: Zählt die Bomben in den Nachbarfeldern eines bestimmten Feldes

2. Die Spiel-Logik (Aufdecken von Feldern & Spielmechanik)

   > Diese Funktion verarbeitet die Aktion des Spielers, insbesondere das Aufdecken von Feldern.

   - `dig(self, row: int, col: int)`: Deckt ein Feld auf und bestimmt, ob ein Spiel weitergeht oder endet
   - Falls das Feld eine `0` enthält, werden alle angrenzenden Felder rekrusiv aufgedeckt

3. Das Spielfeld anzeigen (`__str__()`)

   > Die Funktion erstellt die Benutzeroberfläche und zeigt der spielenden Person das Spielfeld in der Konsole an.

   - `__str__(self)`: Erstellt eine Zeichenkette, die das aktuelle Spielfeld grafisch darstellt

- zeigt aufgedeckte Felder und verborgene Felder mit Leerzeichen (`''`)
- Stellt sicher, dass das Spielfeld gut formatiert wird

4. Spielablauf (`play()` -Steuerung des Spiels)

> Diese Funktion kontrolliert den Ablauf des Spiels, verarbeitet Eingaben und bestimmt Sieg oder Niederlage.

- Erstellt das Spielfeld (`Board`-Objekt)
- Fragt die spielende Person wo sie graben möchte
- Überprüft die Eingabe und führt `dig()`aus
- Beendet das Spiel, wenn eine Bombe aufgedeckt wird oder alle sicheren Felder gefunden wurden. Wenn ein Feld keine Bombe ist, grabt das Programm rekursiv weiter, bis jedes Feld mindestens an eine Bombe angrenzt.
- Gibt entweder eine Sieg- oder Niederlage-Nachricht aus

5. Steuerung des Programms (`if __name__== '__main__':`)

> Die Funktion stellt sicher, dass das Spiel nur dann gestartet wird, wenn das Skript direkt ausgeführt wird.

- `if__name__== '__main__':`ruft `play()`auf, falls das Skript direkt ausgeführt wird.

Architekturbeschreibung

Für meinen Main-Code habe ich die Anweisung `import random` gewählt. Dieses Modul enthält Funktionen zur Erzeugung von Zufallszahlen und zur zufälligen Auswahl von Elementen aus Listen oder anderen Sequenzen. Welches hilfreich für die Auswahl der Zahlen auf meinem Spielbrett hilfreich sein wird.

Attribut	Typ	Beschreibung
`dim_size`	int	Die Größe des Spielfelds (`dim_size × dim_size`).
`num_bombs`	int	Anzahl der Bomben im Spiel.
`board`	list[list]	Zweidimensionales Array, das das Spielfeld speichert (`'*'` für Bomben, Zahlen für angrenzende Bombenanzahl, `None` für uninitialisierte Felder).
`dug`	list	Liste der bereits ausgegrabenen Felder (Tuple mit `(row, col)`).

Methode	Beschreibung
`__init__(self, dim_size: int, num_bombs: int)`	Initialisiert das Spielfeld und füllt es mit Bomben und Zahlenwerten.
`make_new_board()`	Erstellt ein leeres Spielfeld und platziert Bomben zufällig darauf.
`assign_values_to_board()`	Berechnet die Anzahl der benachbarten Bomben für jedes Feld und speichert diese als Zahlenwerte im `board`.
`get_num_neighboring_bombs(self, row: int, col: int)`	Berechnet die Anzahl der Bomben in der Nachbarschaft eines bestimmten Feldes.

Methode	Beschreibung
`dig(self, row: int, col: int)`	Simuliert das Graben an einer Position. Deckt benachbarte Felder rekursiv auf, wenn keine Bombe in der Nähe ist.
`__str__()`	Erstellt eine textuelle Darstellung des Spielfelds für die Ausgabe in der Konsole.

Die Klasse `Board`

Die `Board`-Klasse repräsentiert das Spielfeld des Spiels. Sie enthält Methoden zur Erstellung des Spielfelds, zur Platzierung der Bomben und zur Berechnung der benachbarten Bomben. Außerdem gibt sie das Spielfeld als Zeichenkette aus, sodass es in der Konsole dargestellt werden kann.

Der Konstruktor: `__init__(self, dim_size: int, num_bombs: int)`

Die Methode `__init__` wird aufgerufen, wenn ein neues `Board`-Objekt erstellt wird. Hier werden die Dimensionen des Spielfelds (`dim_size × dim_size`) und die Anzahl der Bomben (`num_bombs`) festgelegt. Anschließend wird das Spielfeld generiert, indem die Methode `make_new_board()` aufgerufen wird. Diese Methode erstellt eine zweidimensionale Liste, die das Spielfeld repräsentiert, und platziert die Bomben zufällig darauf. Danach wird die Methode `assign_values_to_board()` ausgeführt, um für jedes Feld die Anzahl der angrenzenden Bomben zu berechnen. Zusätzlich wird eine Menge (`self.dug`) initialisiert, die speichert, welche Felder bereits aufgedeckt wurden.

Die Methode `make_new_board(self)`

Diese Methode erstellt eine neue Spielfeldmatrix, indem eine Liste von Listen erzeugt wird, die zunächst mit `0` gefüllt ist.

```
board: list[list[Union[int, str]]] = [
    [0 for _ in range(self.dim_size)] for _ in range(self.dim_size)]
```

Danach werden Bomben zufällig im Spielfeld platziert. Dafür wird eine `while`-Schleife verwendet, die so lange läuft, bis die festgelegte Anzahl von Bomben (`num_bombs`) erreicht ist. Um eine zufällige Position zu bestimmen, wird eine Zahl zwischen `0` und `dim_size² − 1` gewählt. Die Zeilen- und Spaltennummer werden anschließend berechnet, indem die Zahl in Zeilen- und Spaltenindizes umgerechnet wird. Falls an dieser Stelle bereits eine Bombe vorhanden ist, wird eine neue Position gesucht. Sobald eine gültige Position gefunden wurde, wird die Bombe als `'*'` im Spielfeld gespeichert.

```
bombs_planted: int = 0
while bombs_planted < self.num_bombs:
    loc: int = random.randint(0, self.dim_size**2 - 1)
    row: int = loc // self.dim_size
    col: int = loc % self.dim_size
    if board[row][col] == '*':
        continue
    board[row][col] = '*'
    bombs_planted += 1
return board
```

Die Methode `assign_values_to_board(self)`

Nachdem die Bomben gesetzt wurden, müssen die Zahlen für die restlichen Felder berechnet werden. Jedes Feld ohne Bombe erhält eine Zahl zwischen `0` und `8`, die angibt, wie viele Bomben sich in den benachbarten Feldern befinden. Dazu durchläuft eine `for`-Schleife jedes Feld im Spielfeld. Falls sich an der aktuellen Position eine Bombe befindet, wird die Berechnung übersprungen. Andernfalls wird die Anzahl der benachbarten Bomben durch die Methode `get_num_neighboring_bombs()` ermittelt und in der jeweiligen Zelle gespeichert.

```
def assign_values_to_board(self):
    for r in range(self.dim_size):
        for c in range(self.dim_size):
            if self.board[r][c] == '*':
                continue
            self.board[r][c] = self.get_num_neighboring_bombs(r, c)
```

Die beiden `for`-Schleifen iterieren durch das gesamte Spielfeld, das als zweidimensionale Liste `self.board` gespeichert ist.

- `r` repräsentiert die Zeile des Spielfelds.
- `c` repräsentiert die Spalte des Spielfelds. Da `self.dim_size` die Dimension des Spielfelds angibt (z. B. ein 5x5 Spielfeld hat `dim_size = 5`), werden diese Schleifen von `0` bis `dim_size-1` gehen.

Mit `if self.board [r] [c] == '*'` wird überprüft, ob das aktuelle Feld eine Bombe enthält. Wenn das der Fall ist, soll das Programm für dieses Feld keine benachbarten Bomben berechnen. `continue` sorgt dafür, dass der Rest des Codes innerhalb der inneren Schleife übersprungen wird und der nächste Wert für `c` (die nächste Spalte) überprüft wird. Das bedeutet, dass der Code die Berechnung nur für Felder vornimmt, die keine Bombe sind.

Die Methode `get_num_neighboring_bombs(self, row: int, col: int)`

Diese Methode berechnet die Anzahl der Bomben, die sich in den direkt angrenzenden Feldern eines bestimmten Feldes befinden.

Zuerst wird die Variable `num_neighboring_bombs` auf `0` gesetzt, um die Anzahl der gefundenen Bomben zu zählen. Anschließend durchlaufen zwei geschachtelte `for`-Schleifen alle benachbarten Felder.

Damit das Programm nicht über die Ränder des Spielfelds hinausläuft, wird bei der Iteration über die Zeilen (`r`) und Spalten (`c`) sichergestellt, dass die Werte innerhalb der gültigen Grenzen des Spielfelds bleiben. Dies geschieht mit `max(0, row-1)` und `min(self.dim_size-1, row+1)` für die Zeilen sowie den entsprechenden Werten für die Spalten. Dadurch wird verhindert, dass das Programm versucht, auf ungültige Indizes außerhalb des Spielfelds zuzugreifen.

Während der Schleifen werden alle benachbarten Felder durchlaufen, außer das aktuelle Feld selbst (also das Feld `row, col`), welches durch eine `if`-Abfrage ausgeschlossen wird. Falls eines der überprüften Felder eine Bombe (`'*'`) enthält, wird `num_neighboring_bombs` um 1 erhöht.

Nach Abschluss der Schleifen wird die Gesamtanzahl der Bomben in den Nachbarfeldern zurückgegeben. Diese Information wird später genutzt, um dem Spieler anzuzeigen, wie viele Bomben sich um ein

aufgedecktes Feld befinden.

```
def get_num_neighboring_bombs(self, row: int, col: int) -> int:
    """Gibt die Anzahl der benachbarten Bomben eines Feldes zurück."""
    num_neighboring_bombs: int = 0
    for r in range(max(0, row - 1), min(self.dim_size - 1, row + 1) + 1):
        for c in range(max(0, col - 1), min(self.dim_size - 1, col + 1) + 1):
            if r == row and c == col:
                continue
            if self.board[r][c] == '*':
                num_neighboring_bombs += 1
    return num_neighboring_bombs
```

Die Methode `dig(self, row: int, col: int)`

Die Methode ist eine zentrale Funktion des Spiels, die dafür zuständig ist, ein bestimmtes Feld auf dem Spielfeld aufzudecken. Sie entscheidet, ob das Spiel weitergeht oder ob der Spieler eine Bombe trifft und verliert.

Zunächst wird das aufgedeckte Feld in der Menge `self.dug`gespeichert. Diese Menge speichert alle bereits aufgedeckten Felder, damit nicht unnötig doppelt gegraben wird.

Das Programm durchläuft drei verschiedene Szenarien.

1. Falls das aufgedeckte Feld eine Bombe (`'0'`) ist, dann endet das Spiel sofort. Die Methode gibt `False` zurück, um anzuzeigen, dass die spielende Person verloren hat.

   ```
   if self.board[row][col] == '*':
       return False
   ```

2. Das Feld enthält eine Zahl größer als 0. Falls das Feld eine Zahl enthält (z. B. 1, 2 oder 3), bedeutet das, dass sich in der Nähe Bomben befinden. Das Feld wird einfach als aufgedeckt markiert, aber es passiert nichts weiter. Die Methode gibt `True` zurück, da das Spiel weitergeht.

   ```
   val = self.board[row][col]
   if isinstance(val, int) and val > 0:
       return True
   ```

3. Das Feld enthält ein `0`(keine benachbarten Bomben). In diesem Fall müssen alle benachbarten Felder ebenfalls aufgedeckt werden. Dies geschieht durch eine rekursive Tiefensuche: Die Methode ruft sich selbst für jedes angrenzende Feld auf, das noch nicht aufgedeckt wurde. Dadurch wird eine ganze zusammenhängende Fläche von sicheren Feldern automatisch aufgedeckt.

   ```
   for r in range(max(0, row-1), min(self.dim_size-1, row+1)+1):
       for c in range(max(0, col-1), min(self.dim_size-1, col+1)+1):
           if (r, c) in self.dug:
               continue
           self.dig(r, c)
   return True
   ```

Damit das Programm nicht aus den Spielfeldgrenzen herausläuft, werden die Werte für die Zeilen `(r)` und Spalten `(c)` durch die Funktionen `max(0, row-1)` und `min(self.dim_size-1, row+1)` begrenzt. Dies stellt sicher, dass nur gültige Positionen überprüft werden. Nachdem alle notwendigen Felder

aufgedeckt wurden, gibt die Methode abschließend True zurück, sofern keine Bombe getroffen wurde. Das Spiel läuft dann weiter, und der Spieler kann weitere Züge machen.

Die Methode `__str__(self)`

Die Methode `__str__` ist eine spezielle Funktion in Python, die bestimmt, wie ein Objekt als Zeichenkette dargestellt wird – etwa beim Ausgeben mit `print()`. In diesem Fall sorgt sie dafür, dass das Spielfeld übersichtlich und spielerfreundlich formatiert wird.

Zunächst wird ein sichtbares Spielfeld (`visible_board`) erstellt – ein zweidimensionales Array mit denselben Abmessungen wie das tatsächliche Spielfeld (`self.board`). Zu Beginn ist jedes Feld mit einem Leerzeichen (`' '`) belegt, da anfangs noch keine Informationen über aufgedeckte Felder vorliegen. Erst wenn der Spieler ein Feld aufdeckt, wird der entsprechende Inhalt – eine Zahl oder ein `*` für eine Bombe – sichtbar gemacht.

```
visible_board: List[List[str]] = [[' ' for _ in range(self.dim_size)] for _ in range(self.dim_size)]
```

Nun wird das sichtbare Spielfeld befüllt: Eine verschachtelte `for`-Schleife durchläuft jede Zeile `(row)` und jede Spalte `(col)` des Spielfelds. Für jedes Feld wird geprüft, ob es sich in der Menge `self.dug` befindet – das bedeutet, ob der Spieler dieses Feld bereits aufgedeckt hat. Ist das der Fall, wird der entsprechende Wert aus `self.board[row][col]` – also entweder eine Zahl, die die Anzahl benachbarter Bomben angibt, oder ein `'*'` für eine Bombe – in die `visible_board` übernommen. Andernfalls bleibt das Feld leer und somit für den Spieler verborgen.

```
for row in range(self.dim_size):
    for col in range(self.dim_size):
        if (row, col) in self.dug:
            visible_board[row][col] = str(self.board[row][col])
string_rep: str = ''
```

In diesem Abschnitt wird die oberste Zeilenbeschriftung des sichtbaren Spielfelds erzeugt, um den Spielerinnen und Spielern die Orientierung in den Spalten zu erleichtern. Zunächst wird mit der List Comprehension `indices = [str(i) for i in range(self.dim_size)]` eine Liste von Zeichenketten erstellt, die die Spaltenindizes repräsentieren – zum Beispiel `["0", "1", "2", ..., "n-1"]` für ein Spielfeld mit größe `n x n`. Die Umwandlung in Strings (`str(i)`) ist notwendig, weil die anschließende `join()`-Funktion eine Zeichenkette erzeugt und nur mit iterierbaren Strings funktioniert. Mit `' '.join(indices)` werden diese einzelnen Zahlenwerte dann zu einer durch doppelte Leerzeichen getrennten Zeichenkette zusammengesetzt. Zusätzlich werden zu Beginn noch drei Leerzeichen (`' '`) eingefügt, damit die Spaltenüberschrift später optisch korrekt über dem eigentlichen Spielfeld sitzt – genau oberhalb der jeweiligen Spaltenwerte. Das Ergebnis wird in der Variable `indices_row` gespeichert und bildet die Kopfzeile des Spielfelds, welche bei jeder Darstellung mitausgegeben wird.

```
indices: List[str] = [str(i) for i in range(self.dim_size)]
indices_row: str = '   ' + '  '.join(indices) + '\n'
```

Als nächstes wird das gesamte sichtbare Spielfeld zeilenweise als formatierter Text zusammengesetzt. Mit `enumerate(visible_board)` wird jede Zeile des Spielfelds durchlaufen, wobei `i` den aktuellen Zeilenindex darstellt und `row_vals` die Inhalte der jeweiligen Zeile. Für jede dieser Zeilen wird eine neue

Textzeile (`string_rep`) aufgebaut: Zunächst wird der Zeilenindex `i` links vorangestellt, gefolgt von einem senkrechten Strich als Trennzeichen. Dann werden alle Felder der aktuellen Zeile (`row_vals`) mithilfe von `' |'.join(row_vals)` durch senkrechte Striche getrennt aneinandergehängt. Am Ende jeder Zeile wird ebenfalls ein senkrechter Strich sowie ein Zeilenumbruch `\n` angefügt. Nachdem alle Zeilen so formatiert wurden, wird die Kopfzeile (`indices_row`) mit einer Trennlinie (bestehend aus Bindestrichen in der gleichen Länge wie die Kopfzeile) und dem gesamten zusammengesetzten Spielfeld (`string_rep`) kombiniert und als finaler String zurückgegeben. So entsteht eine übersichtliche Textdarstellung des aktuellen Spielfeldzustands für den Spieler.

- `f'{i} |'` → Fügt die Zeilennummer hinzu (z. B. `0 |`).
- `' |'.join(row_vals)` → Fügt die Felder der Zeile hinzu, getrennt durch `|`.
- `|\n'` → Fügt ein abschließendes `|` sowie einen Zeilenumbruch `\n` hinzu.

```
for i, row_vals in enumerate(visible_board):
    string_rep += f'{i} |' + ' |'.join(row_vals) + ' |\n'
return indices_row + '-' * len(indices_row) + '\n' + string_rep
```

Funktion `play(dim_size: int = 5, num_bombs: int = 5)`

Die Funktion ist das Hauptprogramm für das Spiel 'Verlassene Raumstation'. Sie initialisiert das Spielfeld, verarbeitet die Eingaben des Spielers und steuert den Spielablauf.

Initialisierung des Spiels

Die Funktion `play(dim_size=5, num_bombs=5)` startet das Spiel, wobei die Größe des Spielfelds (`dim_size`) und die Anzahl der Bomben (`num_bombs`) als Parameter übergeben werden können (Standardwerte sind ein 5x5-Feld mit 5 Bomben). Zu Beginn wird ein `Board`-Objekt erstellt, das das Spielfeld verwaltet.

```
def play(dim_size: int = 5, num_bombs: int = 5) -> None:
    while True:
        board: Board = Board(dim_size, num_bombs)
        safe: bool = True
```

Spielablauf – Wiederholungsschleife

Die Hauptspielschleife läuft mit der Bedingung `while len(board.dug) < board.dim_size ** 2 - num_bombs:`. Diese Bedingung stellt sicher, dass das Spiel so lange weiterläuft, bis alle sicheren Felder aufgedeckt wurden. Dabei entspricht `board.dug` der Menge der bereits aufgedeckten Felder, während `board.dim_size ** 2 - num_bombs` die Gesamtanzahl der nicht von Bomben belegten Felder darstellt. Solange noch sichere Felder übrig sind, bleibt der Spieler im Spiel.

Aufdecken eines Feldes

In jedem Schleifendurchlauf wird zuerst das aktuelle Spielfeld angezeigt. Anschließend wird der Spieler aufgefordert, ein Feld durch Eingabe von Koordinaten im Format „Reihe, Spalte“ zu wählen. Die Eingabe wird verarbeitet und geprüft:

- Die Koordinaten werden mithilfe von `.split(',')` getrennt und in Ganzzahlen umgewandelt.
- Es folgt eine Validierung, ob die Koordinaten im erlaubten Bereich liegen.
- Falls die Eingabe ungültig ist, wird eine Fehlermeldung angezeigt und der Spieler erneut zur Eingabe aufgefordert.

Ist die Eingabe korrekt, wird das entsprechende Feld mit `board.dig(row, col)` aufgedeckt. Diese Methode gibt `True` zurück, wenn der Spielzug sicher war, und `False`, wenn eine Bombe getroffen wurde. Im Falle eines Bombentreffers wird die Schleife sofort mit `break` verlassen.

```
while len(board.dug) < board.dim_size ** 2 - num_bombs:
    print(board)
    try:
        user_input: str = input("Gib die Koordinaten zum Scannen des Spielfelds ein (Reihe, Spalte): ")
        row_str, col_str = user_input.split(',')
        row, col = int(row_str.strip()), int(col_str.strip())
        if row < 0 or row >= board.dim_size or col < 0 or col >= dim_size:
            print("Ungültige Position. Bitte versuche es erneut.")
            continue
    except ValueError:
        print("Ungültiges Eingabeformat. Verwende: Reihe, Spalte.")
        continue

    safe = board.dig(row, col)
    if not safe:
        break
```

Spielende: Gewinn oder Niederlage

Nachdem die Schleife beendet wurde – entweder durch das Aufdecken aller sicheren Felder oder durch eine Bombe – folgt die Auswertung:

- Falls `safe == True`, bedeutet das, dass die Spieler*in alle sicheren Felder aufgedeckt hat – er oder sie gewinnt das Spiel. Eine Siegesnachricht wird ausgegeben.
- Falls `safe == False`, bedeutet dies, dass der Spieler eine Bombe aufgedeckt hat – das Spiel ist verloren.

Zur Verdeutlichung werden alle Felder aufgedeckt (`board.dug` wird auf alle Positionen gesetzt). Das Spielfeld wird erneut ausgegeben, diesmal mit allen Bomben sichtbar.

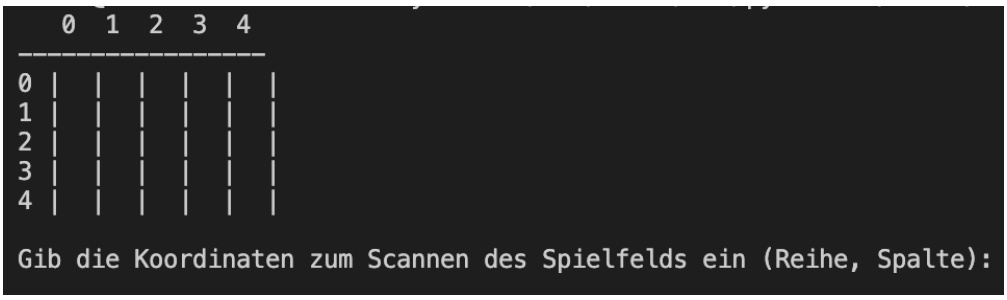
```
    safe = board.dig(row, col)
    if not safe:
        break
if safe:
    print("🎉 GLÜCKWUNSCH!!!! DU HAST GEWONNEN!")
else:
    print("💀 GAME OVER!")
    board.dug = {(r, c) for r in range(board.dim_size) for c in range(board.dim_size)}
    print(board)
```

Eine letzte `if`-Schleife ist dafür zuständig, die spielende Person zu Fragen, ob sie nachdem sie gewonnen oder verloren hat das Spiel nochmal spielen möchte. Falls nicht mit `"ja", "j", "yes", "y"`geantwortet wird, wird das Spiel beendet.

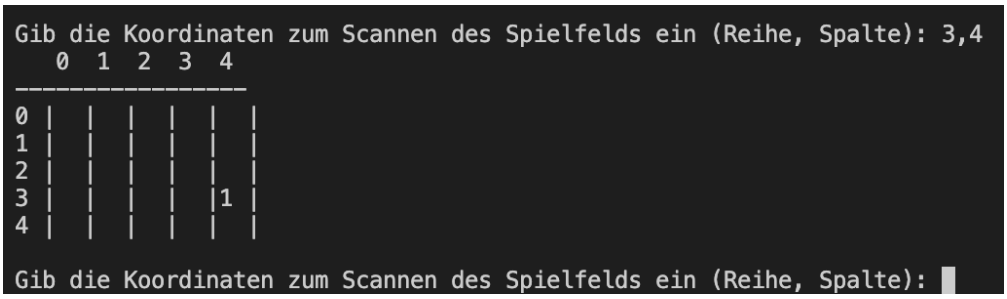
Zuletzt, wenn das Skript direkt ausgeführt wird (`if __name__ == '__main__':`), startet automatisch die `play()`-Funktion, wodurch das Spiel beginnt.

Nutzerinterface

1. Die Benutzeroberfläche zeigt das Spielfeld an und fordert die spielende Person auf Koordinaten (Reihe, Spalte) zum Scannen des Spielfeld einzugeben:

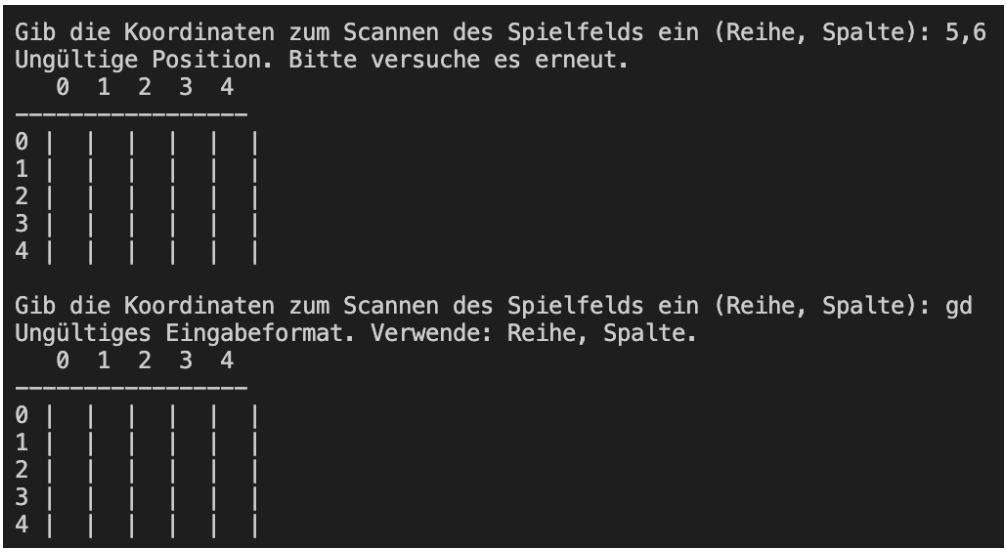


2. Wenn korrekten Koordinaten eingeben wurden, dann wird das erwünschte Feld aufgedeckt und eine Zahl ist zu sehen:



3. Die spielende Person kann so nach und nach Zahlen eingeben. Wenn sie eine Bombe trifft gibt das Programm "Game Over" aus, somit ist das Spiel beendet.

4. Fehlermeldung bei ungültiger Eingabe. Entweder, wenn eine ungültige Zahl eingegeben wurde, die sich nicht im Koordinatenbereich befindet oder andere ungültige Buchstaben oder Zeichen.



> 5. Das letzte Senario kommt zustande, wenn die Person das Spiel gewinnt.

Programmablauf

1. Initialisierung des Spiels:

- Das Programm startet mit der Funktion `play(dim_size=5, num_bombs=5)`.
- Ein neues Spiel wird jedes Mal gestartet, wenn der Benutzer eine neue Runde spielen möchte.
- Die Dimension des Spielfelds (`dim_size`) und die Anzahl der Bomben (`num_bombs`) werden als Parameter für das Spiel übergeben. Standardmäßig sind diese Werte auf 5x5 Felder und 5 Bomben gesetzt.

2. Spielfeld-Erstellung:

Der `Board`-Konstruktor wird aufgerufen:

- Der Konstruktor initialisiert das Spielfeld mit der gewünschten Dimension und der Anzahl der Bomben.
- Die Methode `make_new_board` wird verwendet, um das Spielfeld zufällig zu erstellen und Bomben zu platzieren.
- Anschließend wird `assign_values_to_board` aufgerufen, um jedem Feld die Anzahl der benachbarten Bomben zuzuweisen.

3. Haupt-Spielschleife:

Die Spiellogik läuft innerhalb der `while True`-Schleife:

- Solange der Benutzer keine Bombe trifft und noch nicht alle sicheren Felder aufgedeckt sind, wird die Schleife fortgesetzt.
- Das aktuelle Spielfeld wird nach jedem Zug des Spielers angezeigt, wobei nur die aufgedeckten Felder sichtbar sind.

4. Benutzereingabe:

- Der Benutzer wird aufgefordert, Koordinaten im Format "Reihe, Spalte" einzugeben.
- Die Eingabe wird überprüft:
    - Wenn die Eingabe ungültig oder im falschen Format ist, wird der Benutzer gebeten, es erneut zu versuchen.
    - Wenn die Eingabe gültig ist, wird das entsprechende Feld aufgedeckt.

5. Feld-Aufdeckung:

Die Methode `dig(row, col)` wird aufgerufen, um das Feld bei den eingegebenen Koordinaten aufzudecken:

- Wenn das aufgedeckte Feld eine Bombe enthält (`'*'`), endet das Spiel sofort mit der Ausgabe "GAME OVER".
- Wenn das aufgedeckte Feld eine Zahl enthält, zeigt es die Anzahl der benachbarten Bomben an und das Spiel geht weiter.
- Wenn das aufgedeckte Feld leer ist (keine benachbarten Bomben), wird die Methode rekursiv auf benachbarte Felder angewendet, bis alle leeren Felder aufgedeckt sind.

6. Spielende:

Das Spiel endet, wenn entweder:

- Eine Bombe getroffen wird: Das Spielfeld wird vollständig aufgedeckt und das Spiel zeigt "GAME OVER".
- Alle sicheren Felder aufgedeckt sind: Das Spiel zeigt "GLÜCKWUNSCH, DU HAST GEWONNEN!".

7. Neustart der Runde:

- Nach jedem Spiel (egal ob gewonnen oder verloren) wird der Benutzer gefragt, ob er eine neue Runde spielen möchte.
- Wenn der Benutzer mit "ja" oder "yes" antwortet, startet eine neue Runde.
- Bei einer Antwort wie "nein" oder "no" wird das Spiel beendet und der Benutzer verabschiedet.

Ergebnisse statische & dynamische Tests

Statische Tests mit `pylint`

Nachdem alle Fehler behoben wurden, sehen die Warnungen von `pylint` nun so aus:

```
atsmh@Atussas-MacBook-Pro source % pylint main.py

------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 9.55/10, +0.45)
```

Statiche Tests mit `mypy`

```
Atussas-MacBook-Pro:VRS-AtussaMehrawari atsmh$ mypy .
Success: no issues found in 4 source files
```

Dynamische Unittests

```
atsmh@Atussas-MacBook-Pro Python % /usr/local/bin/python3 "/Users/atsmh/Documents/CS Study/Programmieren/Python/6.Projektarbeit/VRS/tests/test_main.py"
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```