Aufgabenblatt 5

Das sind die Lösungen zu den Aufgaben der Woche 5 (eigentlich Schulwoche 6 da eine Blockwoche zwischen Woche 4 und 5 liegt)

# Aufgabe 21

Implementieren Sie Python-Code, der zwei Listen **liste1** und **liste2** bestehend aus Integer-Zahlen aneinanderfügt und die Ergebnisliste in eine Variable **liste3** speichert. Geben Sie liste3 aus.

Lösung:

Ich habe 3 Lösungen gefunden, es gibt aber noch mehr Lösungen. Alle Lösungen sind auch im Skript „Aufgabe21.py“ im .zip Ordner zu finden.

1. Zusammenfügen mit dem „+“ Operator:

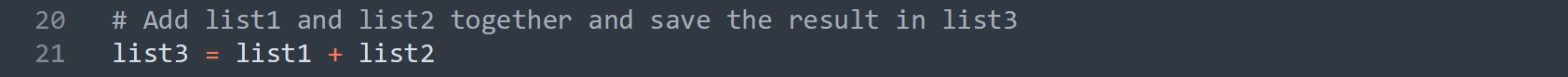


Abbildung - Aufgabe 21 a)

1. Die Werte der zweiten Liste mit einer Schleife an die erste Liste hängen

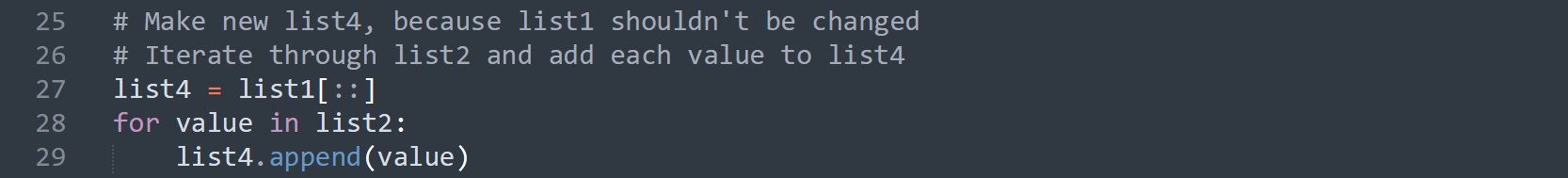


Abbildung - Aufgabe 21 b)

1. Die Methode „extend()“ verwenden

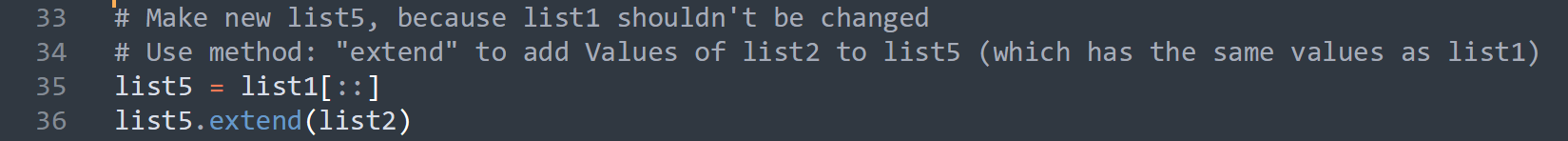


Abbildung - Aufgabe 21 c)

Info: Ich habe die beiden Listen jeweils zufällig generieren lassen, da ich mich gewundert habe ob dies möglich ist.  
Zudem ist mir aufgefallen, dass die Zuweisungen der Listen 4 & 5 nur mit list1[::] funktionieren. Ansonsten wird die Liste 1 verändert.

# Aufgabe 22

Liste aus Integer Werten in umgekehrter Reihenfolge in eine neue Variable „listeReversed“ schreiben. Es sollte keine vordefinierte Funktion verwendet werden, sondern eine Schleife.

Lösung:

Meine erste Lösung benutzt keine Schleife, aber auch keine Methode.

1. Die Zuweisung der alten Liste wird mit den eckigen Klammern in den von mir definierten Schritte, also in diesem Fall mit dem Schritt -1, also von hinten nach vorne, was auch dem reversen entspricht (Das ist auch im Skript „Aufgabe22\_01.py“ zu sehen):

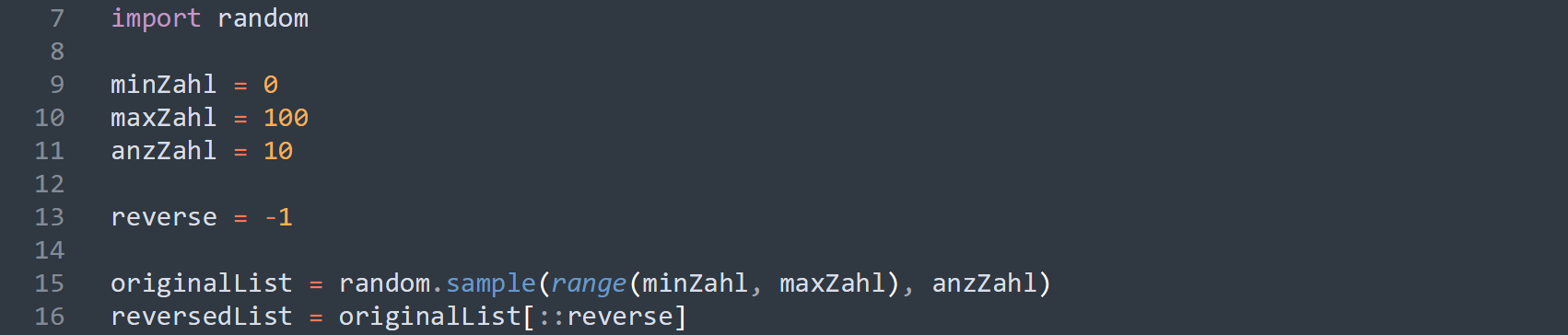


Abbildung - Aufgabe 22 (Semi korrekt)

1. In der zweiten Lösung habe ich die einzelnen Werte aus der Liste gelesen und in die „reversedList“ mit einer „for“ Schleife gespeichert:

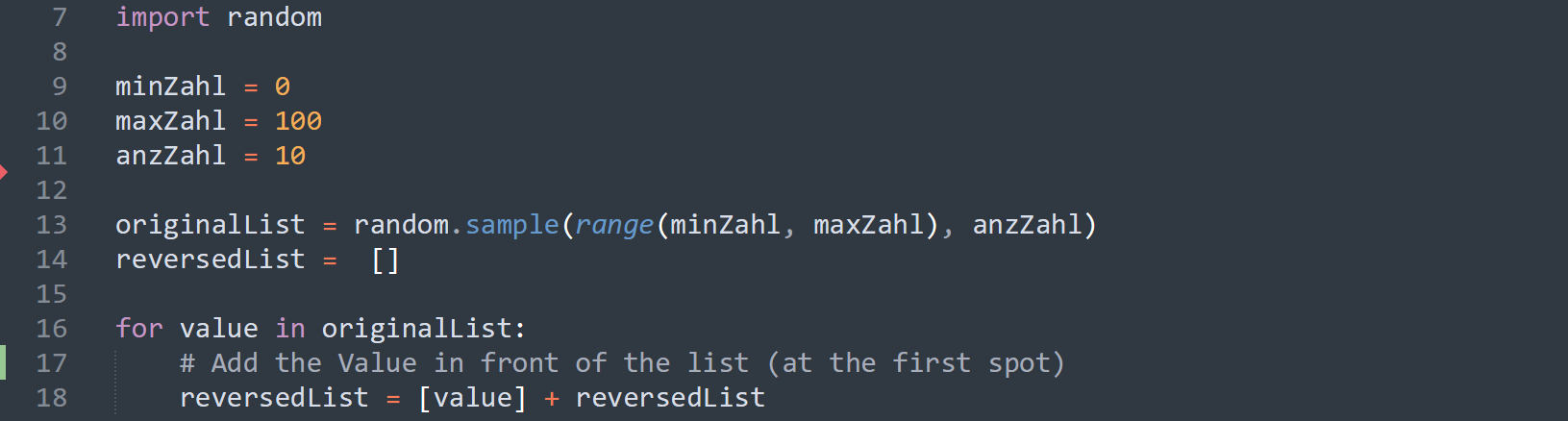


Abbildung - Aufgabe 22 mit for Schleife

Info: originale Liste wird zufällig erstellt.

# Aufgabe 23

Listen **listeA = [1,3,11,23,4,5]** und **listeB = [4,3,11,23,7,8]** vergleichen:

1. Sind die beiden Listen identisch?
2. An welchen Stellen sind sie identisch? (Liste mit boolean Werten)
3. Sind die Listen gleich lang?
4. Listenelemente in aufsteigender Reihenfolge ausgeben und in eine neue Liste speichern

Lösung:

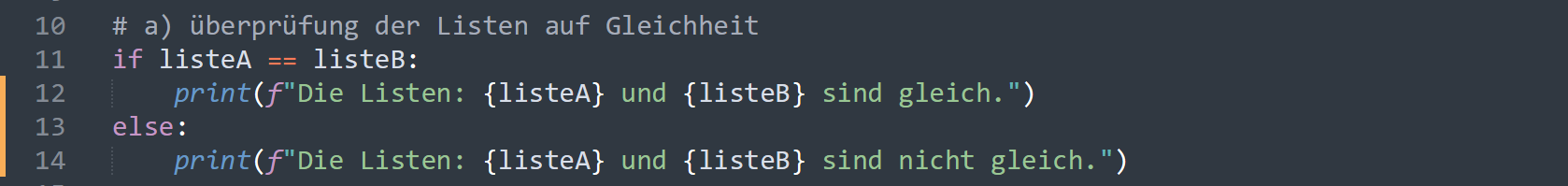
1. Um die Listen mir ihren Werten zu vergleichen, lässt sich der „=“ Operator verwenden:  
   

Abbildung 6 - Aufgabe 23 a)

Es werden die Werte und die Positionen (in der Liste) dieser in Betracht gezogen. Deshalb ist kein Loop notwendig.

1. Die Werte in den Listen werden mit einer „for“ Schleife index für index miteinander verglichen:

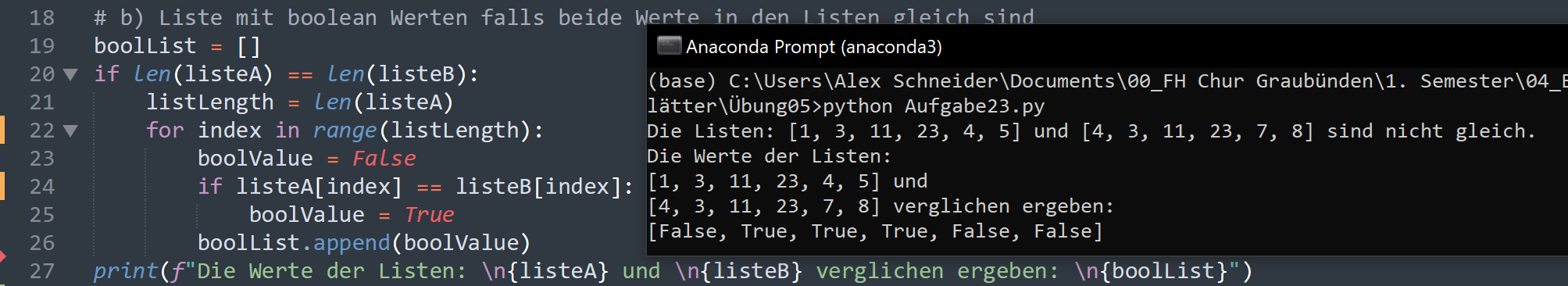


Abbildung - Aufgabe23 b)

Um die Ausgabe zu machen wird ein f String verwendet.

1. Die Länge der beiden Listen wird mit der Methode „len(Liste)“ ausgegeben:

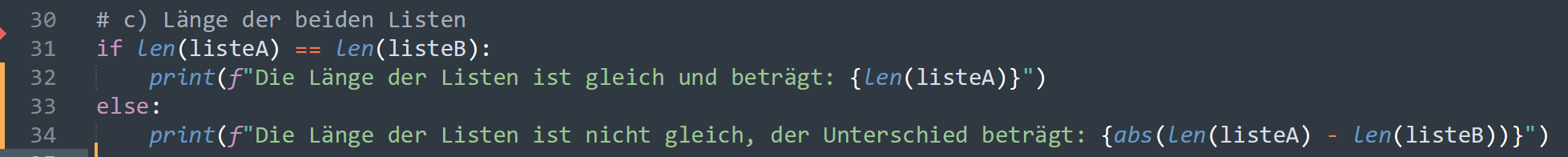


Abbildung - Aufgabe 23 c)

Wenn die beiden Listen nicht gleichlang sind, wird der Unterschied berechnet und mir der Methode «abs(value)» positiv gemacht, falls der Wert negativ ist.

1. Das Sortieren der Listen und zusammenfügen wurde in zwei Schritten gemacht: Im ersten Schritt werden die Listen geklont, sortiert und ausgegeben. Im zweiten Schritt werden die Listen zusammengefügt und ausgegeben. Ich habe das so gelöst, da ich mir nicht sicher war, wie der Auftrag gemeint war.

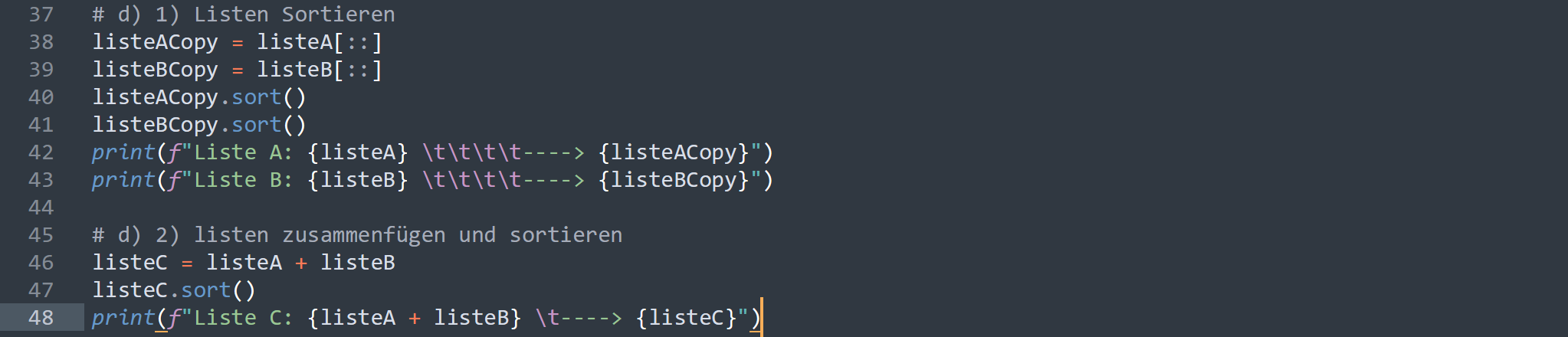


Abbildung - Aufgabe 23 d)

# Aufgabe 24

Schreiben Sie Python-Code, der die Eingabe einer Integer-Zahl erlaubt, eine Liste der entsprechenden Länge vom Wert der Integer-Zahl generiert und die Liste mit Zufallszahlen füllt, die von Index 0 bis zum grössten Index immer grösser werden. Geben Sie die generierte Liste aus.

Lösung:

Um diese Aufgabe zu lösen habe ich mich entschieden zwei Inputs einzugeben:

1. Länge der Liste als Integer
2. Zufallszahl, damit die sortierte Liste einen definierten Wertebereich erhält (an zufälligen Zahlen)

Gelöst habe ich das mit der „random“ Bibliothek:

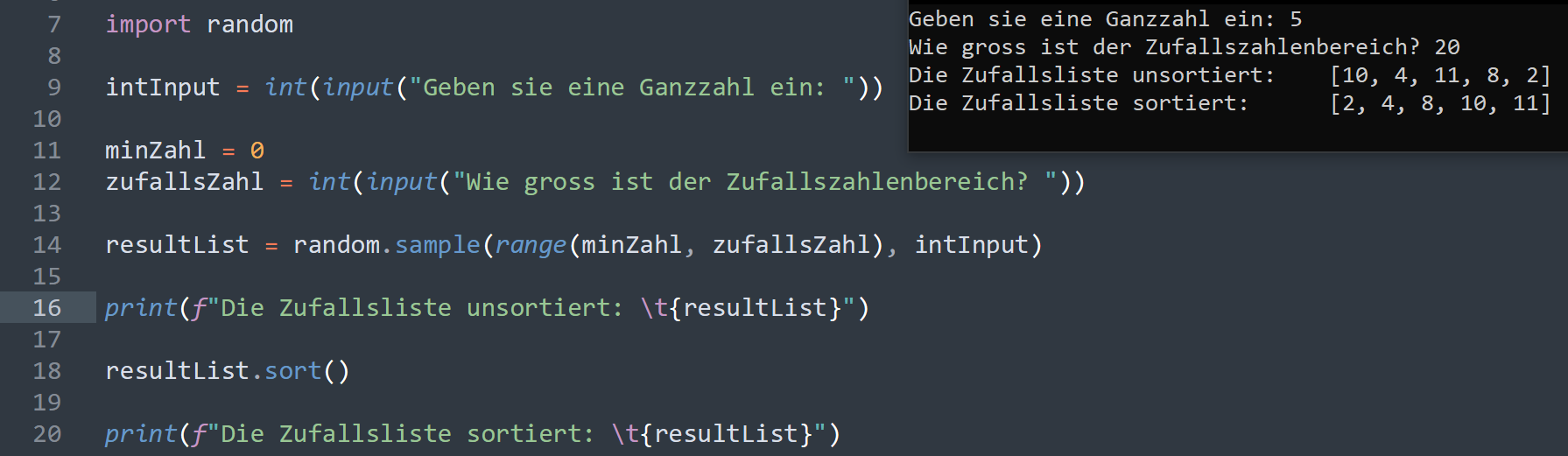


Abbildung - Aufgabe 24

Um die Zufallsliste zu erstellen habe ich die gleiche Methode verwendet wie bei den vorhergehenden Aufgaben.

# Aufgabe 25 – Sudoku

Sudoku ist ein bekanntes Logikrätsel. In der üblichen Version ist es das Ziel, ein 9×9-Gitter mit den Ziffern 1 bis 9 so zu füllen, dass jede Ziffer in jeder Einheit (Spalte, Zeile, Block = 3×3-Unterquadrat) genau einmal vorkommt – und in jedem der 81 Felder exakt eine Ziffer vorkommt. Schreiben Sie Python-Code, der die 81 Felder einer 2D-Liste mit Werten von 1 bis 9 so belegt, dass die oben beschriebene Sudoku-Eigenschaft erreicht wird. (Hinweis: Sie dürfen die Felder zufällig belegen und testen, ob die Eigenschaft erfüllt ist oder Sie belegen Feld für Feld zufällig und nutzen einen Backtracking-Algorithmus.) **(3 P.)** Können Sie das Sudoku im Bildbeispiel unten mit Ihrem Algorithmus lösen?



Abbildung - Bild aus dem Auftrag

Lösung:

Zuerst habe ich die Aufgabenstellung analysiert. Dabei stellt sich heraus, dass es zwei Aufgaben sind:

1. Sudoku erstellen (gegeben ist nichts)
2. Sudoku lösen (Sudoku ist gegeben)

Bei beiden Aufgaben gelten die gleichen Regeln für das Lösen / Erstellen des Sudokus:

1. In der horizontalen dürfen alle Werte von 1 bis 9 jeweils nur einmal erscheinen
2. In der vertikalen dürfen alle Werte von 1 bis 9 jeweils nur einmal erscheinen
3. In den Kacheln von 3x3 dürfen die Werte von 1 bis 9 jeweils nur einmal erscheinen
4. Für Lösen: Es müssen mindestens 17 Werte gegeben sein um eine eindeutige Lösung zu erhalten

Nun wie es im Code gelöst wurde:

1. Alle Funktionen definiert (also die Regeln mit einzelnen Funktionen überprüft)
2. Funktion „Sudoku(grid, startRow, startCol, make)” erstellt
   1. Grid ist das Sudoku Feld
   2. startRow (oder row im Code) ist die Stelle, an welcher gestartet wird
   3. startCol (oder col im Code) ist die Stelle, an welcher gestartet wird
   4. make steht dafür, falls im Sudoku weniger als 17 Werte sind, dann wird nämlich irgendeine Lösung generiert und nicht alle bzw. ein Fehler ausgeworfen, dass nicht 17 Werte gegeben wurden.

## CheckNumberOfValues

Diese Methode ist für das Einhalten der 4. Regel zuständig.

Es wird einfach das ganze Board (grid) überprüft und geschaut ob es mehr oder 17 Werte gibt, welche nicht 0 sind:



Abbildung - Aufgabe 25 Anzahl Werte Check

Um das zu überprüfen wird ein counter verwendet und alle Zahlen, welche über 0 sind, gezählt. Falls dieser Counter am Ende grösser oder gleich 17 ist, dann wird True zurückgegeben, ansonsten False.

## CheckValueInRow

Diese Methode überprüft alle Werte in derselben Zeile:

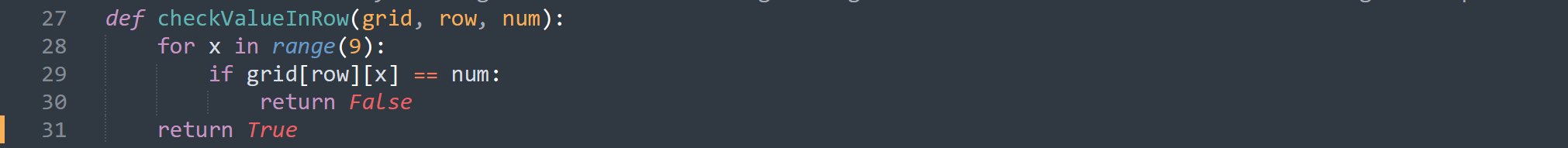


Abbildung - Aufgabe 25 Werte in der Zeile

Dabei wird eine for Schleife verwendet. Ich habe mir überlegt das abhängig vom Sudoku zu erstellen. So könnten auch Sudokus mit mehr als 9 Zeilen und Spalten gelöst und erstellt werden. Schlussendlich habe ich mich dagegen entschieden, da das Testen einer solchen Funktionalität nach Testklasse und Klassen ruft und ich mir nicht zu viel vornehmen wollte.

## CheckValueInCol

Diese Methode überprüft die Werte auf Duplikate in derselben Spalte (gleiches Prinzip wie beim Überprüfen der Werte in einer Zeile).

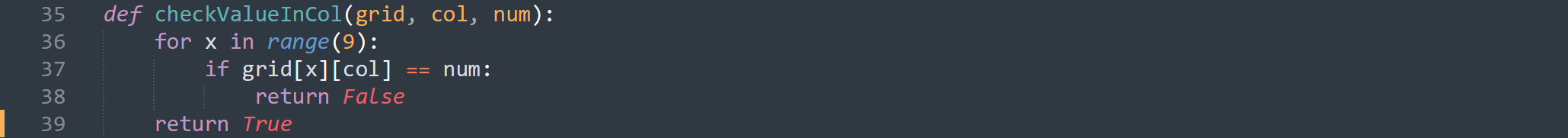


Abbildung - Aufgabe 25 Werte in der Spalte

Hier gilt das gleiche wie bei der Zeile. Der Unterschied zur Zeilenüberprüfung liegt in der Zeile 37. Da werden die Zeilen iteriert, anstatt die Spalten.

## CheckValueInBox

Diese Methode überprüft die Werte in einer 3x3 Box.

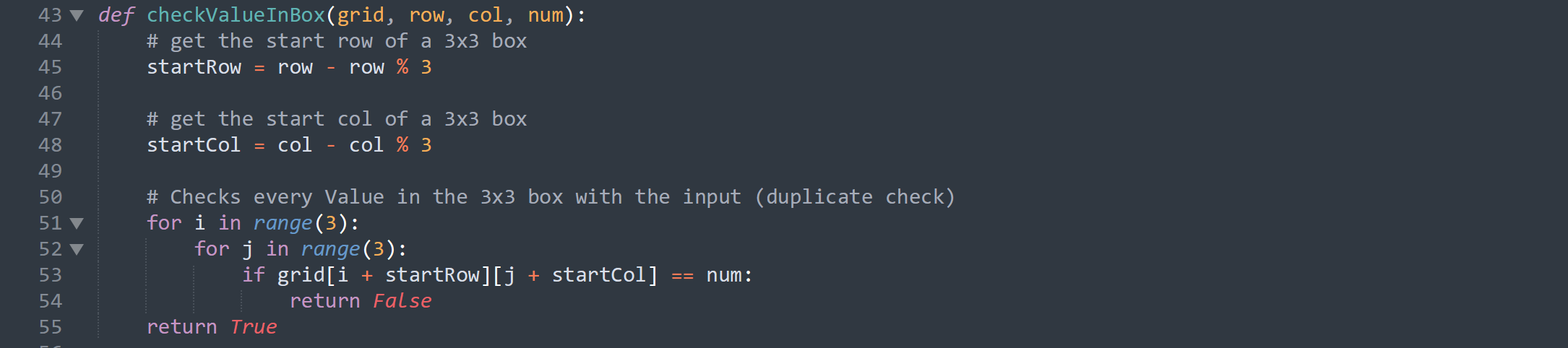


Abbildung - Aufgabe 25 Werte in der Box 3x3

Das Ganze funktioniert indem zuerst ermittelt werden muss, welche Box überprüft werden muss, da es 9 Boxen gibt.

**Beispiel**:

Wir befinden uns an der Stelle x:

1 2 3 | 4 5 6 | 7 8 9

4 5 6 | 7 8 9 | 1 2 3

7 8 9 | 1 2 3 | 4 5 6

------+-------+------

2 1 4 | 3 6 5 | 8 9 7

3 6 5 | 2 1 x | 0 0 0

0 0 0 | 0 0 0 | 0 0 0

------+-------+------

0 0 0 | 0 0 0 | 0 0 0

0 0 0 | 0 0 0 | 0 0 0

0 0 0 | 0 0 0 | 0 0 0

Da bei 0 angefangen wird wäre row dann 5 und column wäre 4,

also würde die

* Startzeile: 5 – 5%3 = 5 – 2 = 3
* Startspalte: 4 – 4%3 = 4 – 1 = 3

So wird dann von oben links begonnen und im 3x3 Feld nach unten rechts durchiteriert.

Da wir zweidimensional überprüfen müssen, sind 2 Schleifen notwendig.

## CheckRules

Diese Methode führt die 3 Werte-Tester zusammen und gibt True oder False zurück:

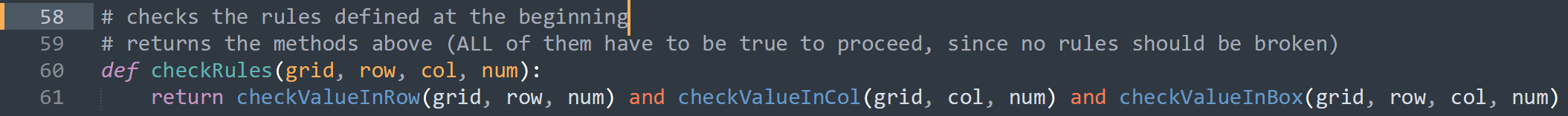


Abbildung - Aufgabe 25 Regel Check

In dieser Methode werden nur die ersten 3 Regeln überprüft. Warum?

Das geht auf die Performance zurück, denn es kann ganz am Anfang überprüft werden ob das Sudoku wirklich mehr oder 17 Werte besitzt.

## Sudoku

In dieser Methode wird das Sudoku gelöst. Wie?

Es werden alle Zahlen von 1 bis 9 probiert bis der check sagt, dass diese Zahl möglich ist und wird dann zur nächsten weiter gehen. Es wird von links nach rechts und von oben nach unten jeder Wert überprüft.

Folgende Schritte (if Bedingungen) werden immer durchgelaufen:

1. Sind es 17 oder mehr Werte bzw. existieren mehrere Lösungen oder nicht?
2. Ist der letzte Wert erreicht? Also row = 8 UND col = 9 (Es ist 8 weil bei 0 begonnen wird und 9 damit die Zahl unten rechts auch überprüft wird)
3. Falls letzte Spalte erreicht wurde, nimm die nächste Zeile (Hier muss nicht auf die letzte Zeile geachtet werden, da dies schon im 2. überprüft wird)
4. Und als letztes wird überprüft ob an jener Stelle schon ein Wert steht (Also kein 0)

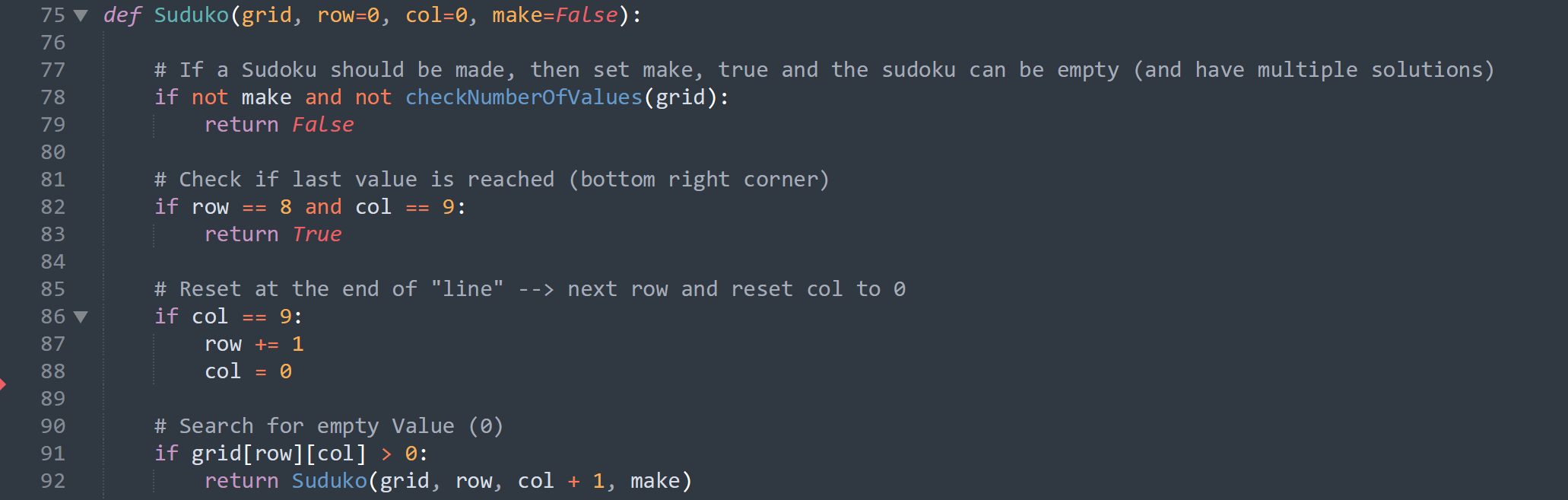


Abbildung - Aufgabe 25 Teil 1

Falls nun 17 Werte oder mehr gegeben wurden, keine 0 dasteht und es nicht der letzte Wert oder die letzte Spalte ist, dann beginnt das Spiel wieder von vorne. Warum?

Naja, nicht ganz von vorne, es wird die Spalte um eins erhöht, um den nächsten Wert zu bestimmen.

Fall nun an der Wert 0 ist, werden alle Zahlen von 1 aufsteigend bis 9 ausprobiert.   
Das funktioniert etwa so:

1. For Schleife, um von 1 bis 9 zu zählen
2. Überprüfen, ob dieser Wert den Regeln zufolge korrekt ist
3. Falls ja dann den Wert setzen und von vorne beginnen, falls nein nächsten Wert nehmen

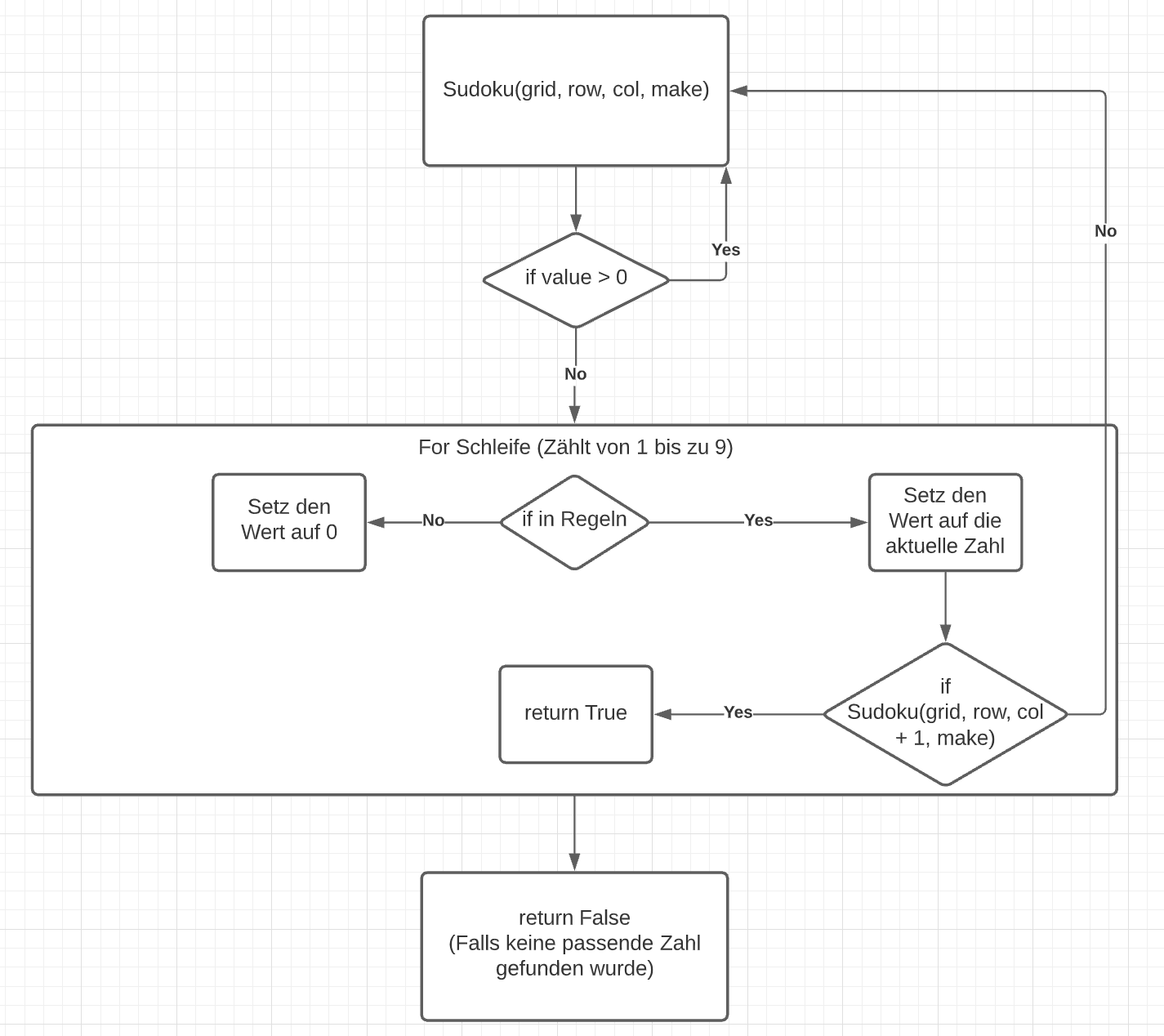


Abbildung - Aufgabe 25 Funktion Sudoku(grid, row, col, make) - Hergestellt mit Lucidchart

Im Bild oben ist der Programmfluss zu sehen. Im Code sieht das so aus:

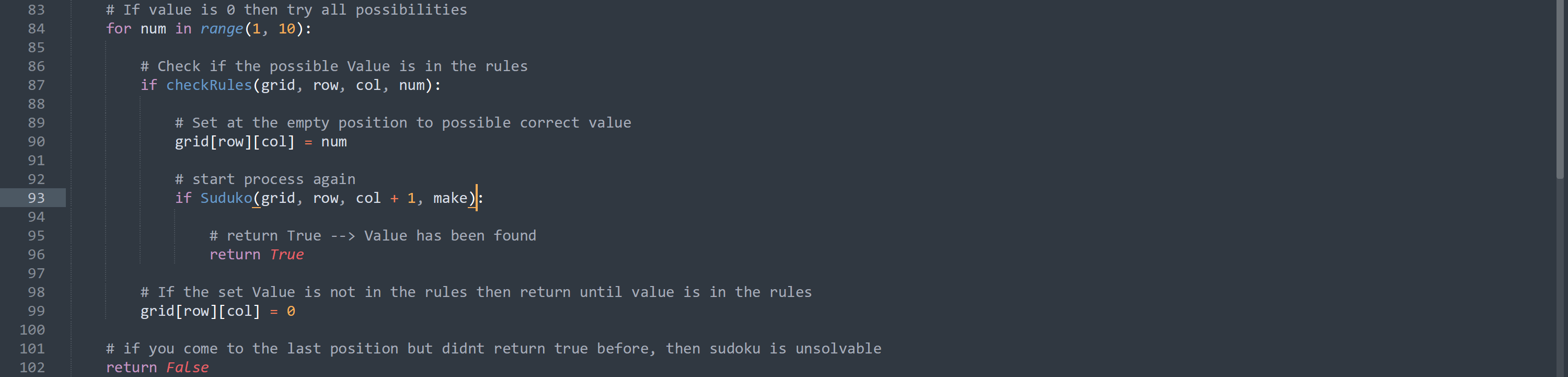


Abbildung - Aufgabe 25 Teil 2

Wenn man ganz am Ende ankommt ist das Sudoku nicht lösbar.

## Verbesserungen:

Es könnten aus dem Lösungsprozess kann vereinfacht werden, indem einmalige Überprüfungen wie das Überprüfen, dass mehr oder 17 Werte im Sudoku enthalten sind oder ob das angegebene Sudoku überhaupt den Regeln entspricht, ausserhalb der Funktion Sudoku(grid, row, col, make) zu überprüfen:

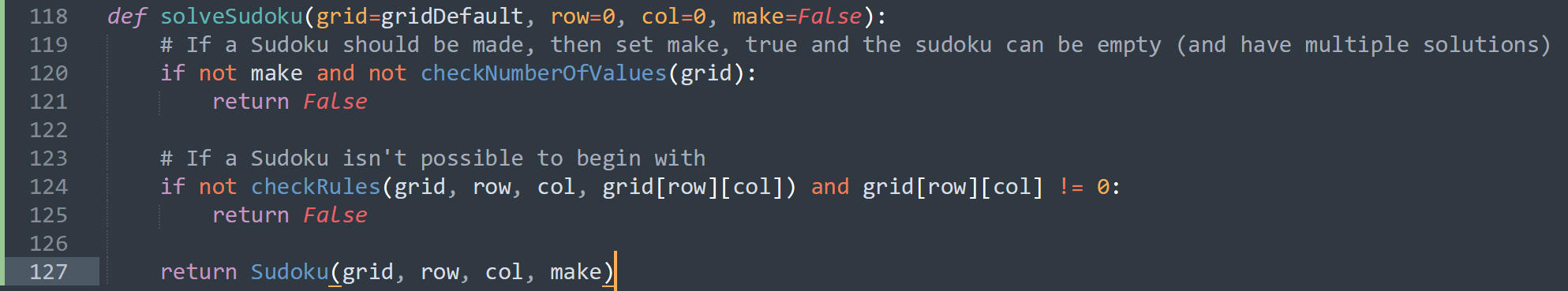


Abbildung – Verbesserungsvorschlag

Es wäre dann nicht mehr nötig die Variable make mitzunehmen bei jeder Rekursion.

Das ist auch im Skript «Aufgabe25\_Verbessert.py» zu sehen.

Ebenfalls habe ich eine Ausgabe der benötigten Zeit in Sekunden, damit der Unterschied zwischen den verschiedenen Varianten des Sudokulösers zu sehen ist:

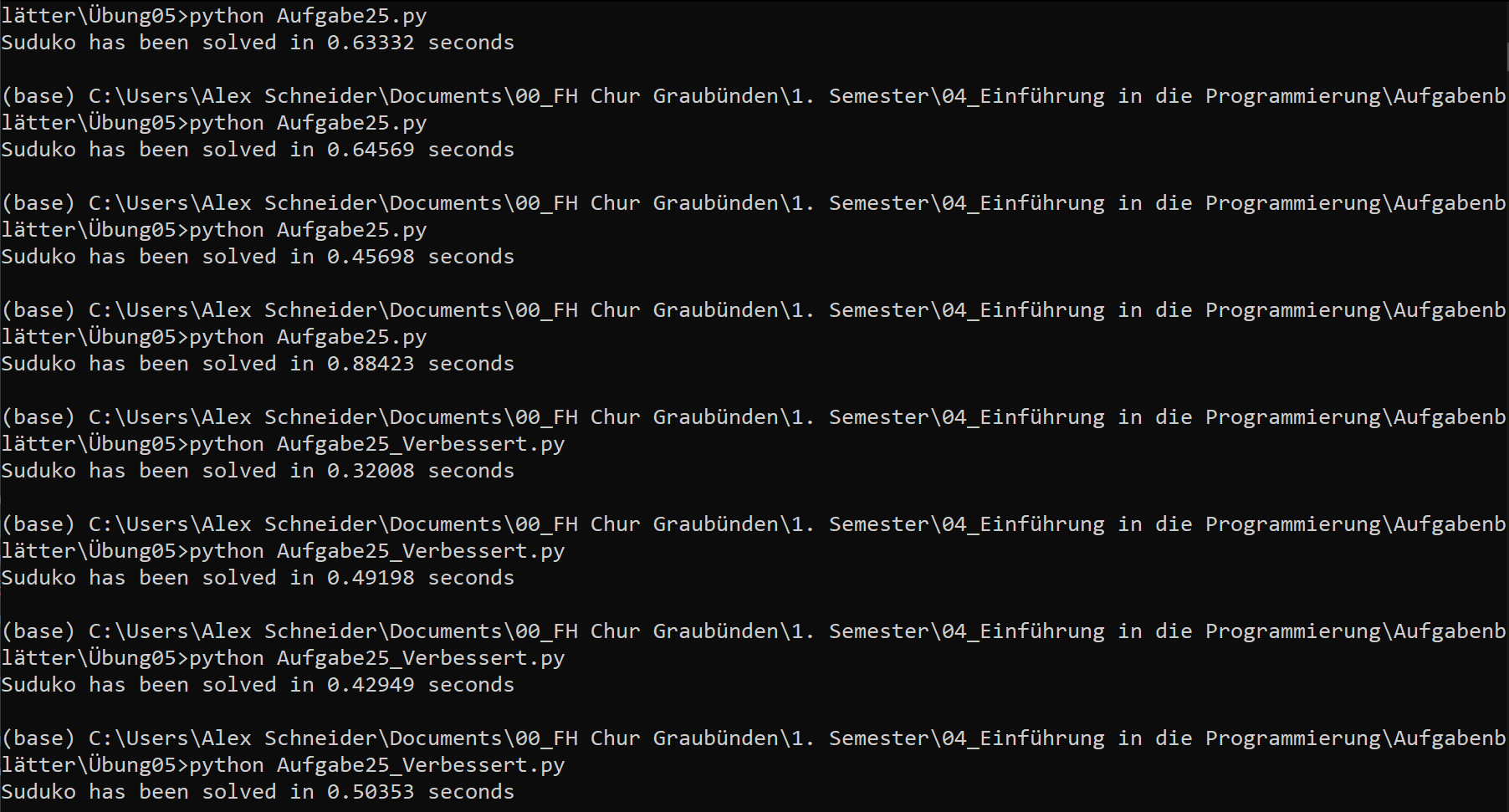


Abbildung - Geschwindigkeitsunterschied Sudokulöser

Die Verbesserte Variante ist etwa 20% schneller als die alte Lösung.

Bei Fragen oder Unklarheiten BITTE melden 😊 .