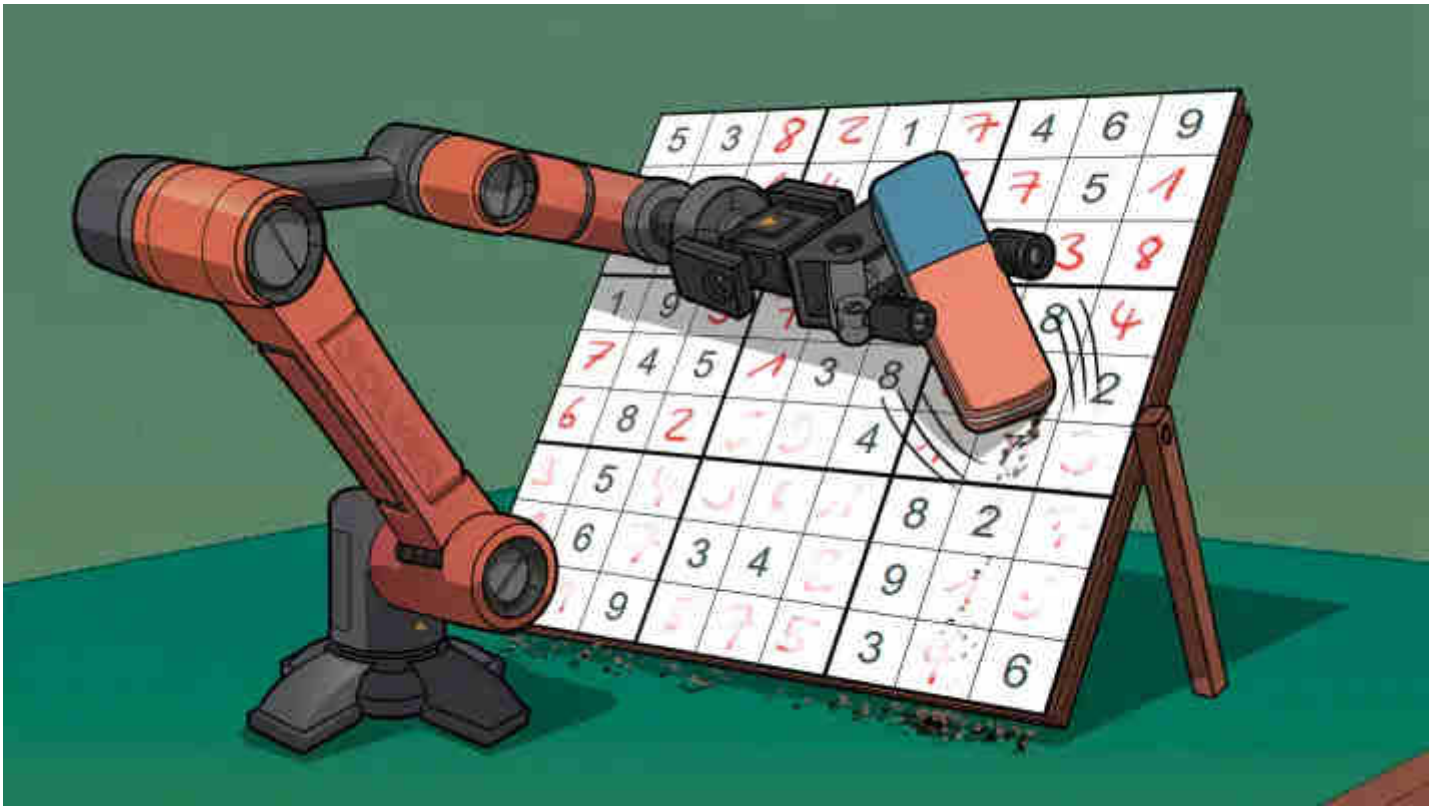
c't magazin für computer technik

Einen Sudoku-Generator in Python programmieren

| 06.04.2023 13:00 Uhr Wilhelm Drehling



Mit unserem Sudoku-Generator erstellen Sie im Handumdrehen eigene Zahlenrätsel. Ganz nebenbei lernen Sie, wie man ein solches Programm in Python realisiert.

Logikrätsel wie Sudokus trainieren nicht nur das Gehirn, sondern geben auch eine prima Programmierübung ab. Die Regeln sind einfach: Jede Reihe, jede Spalte und jeder Kasten eines Sudokus muss die Zahlen 1 bis 9 enthalten, aber keine davon darf mehrfach vorkommen. Ein Soduko zu erstellen, ist dagegen schwieriger: Der naive Programmieransatz, ein Array der Größe 9 × 9 aufzuspannen und mit ein paar Zufallszahlen zu füllen, die nicht gegen die Spielregeln verstoßen, führt selten zum Erfolg. Denn man kann nicht ausschließen, dass mehrere Lösungen existieren.

Sudoku-Generatoren gibt es in Hülle und Fülle, eine Variante beschreiben wir in diesem Artikel: Zuerst erzeugt unser Programm ein vollständig gelöstes Sudoku. Danach entfernt es eine Zahl und prüft, ob das Sudoku nur eine Lösung besitzt. Diesen Schritt wiederholt es so lange, bis genügend Felder leer sind. Damit Sie das fertige Sudoku auch per Stift und Papier lösen können, exportiert unser Programm das Rätsel als Vektorgrafik.

MEHR ZUM PROGRAMMIEREN MIT PYTHON

Wenn es Ihnen schon unter den Fingern brennt und Sie erste Sudokus lösen wollen, dann finden Sie den **gesamten Code im GitHub-Repository [9]**. Außerdem noch ein reinen Solver, der für ein gegebenes Sudoku alle möglichen Lösungen findet.

C'T KOMPAKT

- Mit unserem Programm generieren Sie eigene Sudokus in unterschiedlichen Schwierigkeitsstufen.
- Um ein Sudoku zu erstellen, muss man es vorher lösen.
- Sudoku-Generatoren sind eine schöne Programmierübung, weil sie viele Techniken wie den Backtracking-Algorithmus enthalten.

Rätsel haben Klasse

Das fertige Programm umfasst am Ende mit Kommentaren etwa 170 Zeilen. Um nicht den Überblick zu verlieren und direkt ordentlich zu programmieren, stecken alle Funktionen in der `Sudoku`-Klasse und die Aufrufe dazu in der `main()`-Methode. Die grobe Struktur sieht wie folgt aus:

```
class Sudoku:
    # ...

def main():
    # ...

if __name__ == "__main__":
    main()
```

Die erste Funktion, die in der Sudoku-Klasse landet, ist `reset()`. Diese erstellt ein zweidimensionales Array der Größe 9 × 9 und füllt es mit Nullen auf:

```
def reset(self):
    rows = 9
    columns = 9
    self.board = [[0 for j in range(columns)] for i in range(rows)]
```

Mithilfe dieser Funktion kann man das `board` beliebig oft zurücksetzen und von vorne beginnen. Da ein leeres Sudoku auch der gewünschte Grundzustand des Sudoku-Objekts sein soll, muss der Konstruktor beim Erstellen davon die Funktion aufrufen:

```
def __init__(self):
    self.reset()
```

Jetzt gibt es zwar ein Sudoku voller Nullen, aber noch keine Möglichkeit, das Zwischenprodukt in der Konsole anzusehen. Dieses Problem behebt die `print()`-Funktion:

```
def print(self):
    for i in range(9):
        print(" ".join([str(x)if x != 0 else "." for x in self.board[i]]))
```

Die Schleife durchläuft das gesamte Array, ersetzt die Nullen der Übersichtlichkeit halber durch Punkte und fügt noch Leerzeichen zwischen den Zahlen hinzu. Nun können Sie einen ersten Testlauf starten und das leere Sudoku in der Konsole ausgeben, indem Sie das Objekt in `main()` erstellen und die `print()`-Funktion aufrufen:

```
sudoku = Sudoku()
sudoku.print()
```

Spielregeln und Backtraining-Algorithmus

Bevor es ans Eingemachte geht, benötigt das Programm noch eine Hilfsfunktion, die zunächst prüft, ob eine Zahl `num` bereits in einer Reihe `row` oder Spalte `column` steht:

```
def number_is_valid(self,
                    row, column, num):
  for i in range(9):
    if self.board[row][i] == num or self.board[i][column] == num:
        return False
```

Später greift die Funktion `solve()` sehr oft auf `number_is_valid()` zurück, um mögliche Kandidaten beim Lösungsprozess zu testen.

Außerdem prüft die Funktion, ob `num` schon einmal in dem Kasten mit der angegebenen Position vorkommt. Damit sie diesen durchlaufen kann, braucht sie nur seine linke obere Koordinate:

```
start_column = column // 3 * 3
start_row = row // 3 * 3
for i in range(3):
    for j in range(3):
        if self.board[i + start_row][j + start_column] == num:
            return False
return True
```

Raten und setzen

Nun steht dem Solver nichts mehr im Wege. Dieser muss nicht nur eine Lösung finden, sondern alle möglichen, denn sonst erstellt das Programm versehentlich nicht eindeutig lösbare Sudokus.

Ein möglicher Ansatz wäre, einfach alle Kombinationen mit Gewalt auszuprobieren. Das funktioniert zwar, ist aber ineffizient – vor allem, wenn nur wenige Felder belegt sind. Deshalb verwendet unser Programm einen klassischen Backtracking-Algorithmus. Das ist im Grunde eine rekursive Funktion, die systematisch nach leeren Feldern sucht und gültige Zahlen einsetzt. Bis hierhin klingt es zwar wie Brute Force, aber der Algorithmus geht mit Fehlern anders um: Anstatt wieder von ganz vorne anzufangen, setzt er die letzten Zahlen zurück und probiert andere Kombinationen aus, bis er schließlich eine korrekte Lösung gefunden hat.

```
def solve(self):
    # suche ein leeres Feld
    for r in range(9):
        for c in range(9):
            if self.board[r][c] == 0:
                # wenn möglich, fülle eine gültige Zahl ein
                for n in range(1, 10):
                    if self.number_is_valid(r, c, n):
                        self.board[r][c] = n
                        # gelöst?
                        yield from self.solve()
                        # gehe zurück
                        self.board[r][c] = 0
                return
    yield True
```

Der Backtracking-Algorithmus ist ein Fallbeispiel aus der Informatik: Ein auf den ersten Blick schwieriges Problem lässt sich durch Rekursion plötzlich in wenigen Zeilen lösen.

Das Grundgerüst der Funktion besteht aus einer geschachtelten `for`-Schleife, die das gesamte Sudoku nach leeren Feldern durchsucht (`if self.board[r][c] == 0`). Sollte der Algorithmus auf eine 0 stoßen, befragt er `number_is_valid()` nacheinander für die Zahlen von 1 bis 9, ob sie hineinpasst.

Wenn eine Zahl klappt, setzt er sie ein und versucht das neue `board` zu lösen. Dazu ruft sich `solve()` rekursiv auf. Diese Rekursion setzt sich so lange fort, bis alle Felder gefüllt sind. Damit ist das Sudoku gelöst und der Generator meldet das an den Aufrufer per `yield True` zurück. Der Aufrufer erhält also so viele `True`, wie `solve()` Lösungen gefunden hat.

Nach der Rückkehr aus der Rekursion leert der Algorithmus das Feld (`self.board[r][c] = 0`, Backtracking) und probiert die nächste Zahl. Dadurch, dass der Solver auch dann weitermacht, wenn sich die geprüfte Zahl als passend herausgestellt hat, liefert er nicht nur eine, sondern alle möglichen Lösungen. Gegenüber der oben angedeuteten Brute-Force-Methode erkennt er aber Sackgassen so früh wie möglich und spart sich aussichtsloses Herumprobieren.

Damit haben Sie nun ein kleines Prögrämmchen, das jedes klassische Sudoku lösen kann. Falls Sie später die generierten Sudokus auf Korrektheit überprüfen möchten, können Sie die hier vorgestellten Funktionen als Grundlage für einen Solver benutzen. Wie so einer aussieht, sehen Sie in **unserem GitHub-Repository [11]**.

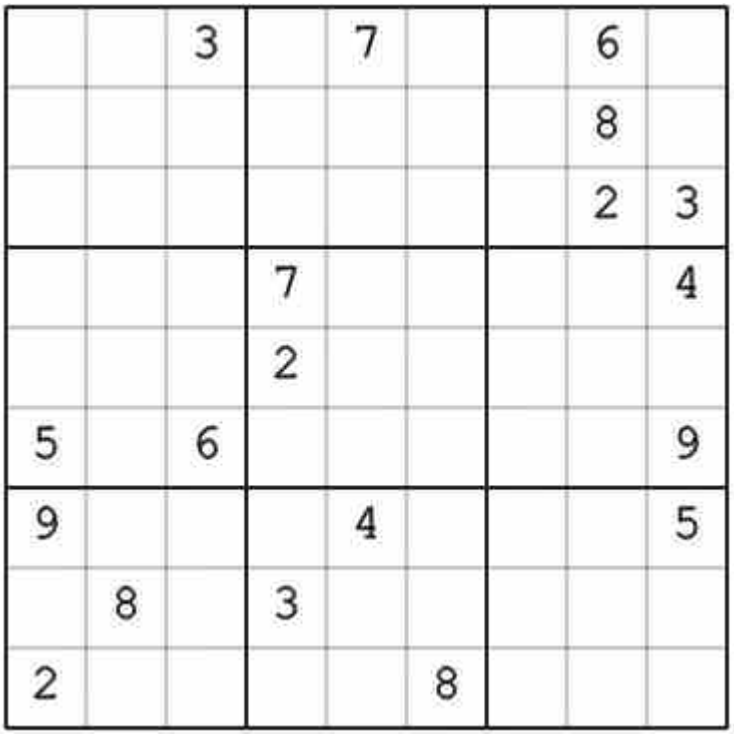
Schwierigkeitsgrade und Rätselerzeugung

Leicht, Mittel, Teuflisch

Damit Sie nicht immer Sudokus der gleichen Schwierigkeit erstellen, soll das Programm eine Funktion erhalten, die Sudokus des gewünschten Schwierigkeitsgrades erzeugt. Je mehr Felder frei bleiben, umso schwieriger wird das Sudoku. Die Funktion `evaluate()` stellt diesen Zusammenhang her:

```
def evaluate(self, difficulty):
  empty_cells = [0, 25, 35, 45, 52, 58, 64]
    if difficulty < 1 or difficulty > len(empty_cells)-1:
        print("invalid difficulty", file=sys.stderr)
    return empty_cells[difficulty]
```

Die Hilfsfunktion erwartet einen Parameter (`difficulty`) zwischen 1 und 6. Dabei entspricht Stufe 1 der Schwierigkeit sehr leicht, Stufe 3 mittel und Stufe 5 ist äußerst schwer. Die sechste Stufe stellt eine Besonderheit dar: Kluge Köpfe haben herausgefunden, dass mindestens 17 Felder besetzt bleiben müssen, damit ein Sudoku eine eindeutige Lösung besitzt. Diese Schwierigkeit können Sie auswählen, aber seien Sie darauf gefasst, Ihren Rechner unglaublich lange mit der Aufgabe zu beschäftigen, da solche Sudokus sehr schwer zu finden sind.



		3		7			6	
							8	
							2	3
			7					4
			2					
5		6						9
9				4				5
	8		3					
2					8			

Nach mehreren Stunden spuckte unser Generator folgendes Sudoku mit 62 leeren Feldern aus. Schaffen Sie es das Sudoku zu lösen?

Damit man später beim Aufruf in der Konsole bequem die Schwierigkeit steuern kann, nimmt der Generator `difficulty` als Argument in `main()` auf:

```
args = [int(x) if x.isdecimal() else x for x in sys.argv[1:]]
difficulty = args[0] if len(args) > 0 else 3
```

Gibt man keinen Parameter an, generiert das Programm standardmäßig Sudokus der Schwierigkeit Mittel (also 3).

Löschungswahn

Nun geht es ans Herzstück des Programms: die Funktion `generate()`. Diese füllt zuerst das leere `board` per Zufall auf und löscht anschließend so lange Zahlen, bis der geforderte Schwierigkeitslevel erreicht ist.

Es gibt mehrere Möglichkeiten, ein leeres Sudoku mit Zahlen zu besetzen: zum Beispiel, indem man Zufallszahlen so lange ausprobiert, bis das Sudoku vollständig ist. Einfach auf die `solve()`-Funktion ausweichen und sie das leere Sudoku lösen lassen, führt nur zu identischen Rätseln, weil sie es immer auf die gleiche Weise füllt. Ein Trick bringt Abwechslung hinein: Bevor der Generator `solve()` die Bühne überlässt, füllt er die Kästen auf einer Diagonalen, weil die keine anderen Kästen, Spalten oder Reihen beeinflussen. Außerdem geht das flott.

```
for i in range(0, 9, 3):
  square = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
    random.shuffle(square)
    for r in range(3):
      for c in range(3):
          self.board[r + i][c + i] = square.pop()
```

Dazu erstellt er ein kleines Array mit neun Zahlen (`square = [...]`), mixt diese durch (`random.shuffle(square)`) und holt sie nacheinander mit `square.pop()` heraus.

Um die restlichen Kästen aufzufüllen, darf nun `solve()` heran. Da man nur die erstbeste Lösung benötigt und die Generator-Funktion nicht endlos viele produzieren soll, bricht die Schleife direkt nach dem ersten Erfolg ab:

```
for solutions in self.solve():
    break
```

Anschließend fragt `generate()` mit dem Aufruf `empty_cells = self.evaluate(difficulty)` die Hilfsfunktion, wie viele Zahlen es aus dem Sudoku entfernen soll, und speichert die Antwort in der Variablen `empty_cells`. Als Nächstes generiert man eine Liste aller Koordinaten des Sudokus namens `unvisited` und mischt diese durch.

```
while empty_cells > 0 and len(unvisited) > 0:
    # wähle eine zufällige Koordinate
    # und speichere eine Kopie
    r, c = unvisited.pop()
    copy = self.board[r][c]
    self.board[r][c] = 0

    # prüfe, wie viele Lösungen es gibt
    solutions = [solution for solution in self.solve()]

    # wenn es mehr als eine Lösung gibt
    # setze Zahl wieder ein
    if len(solutions) > 1:
        self.board[r][c] = copy
    else:
        empty_cells -= 1
```

Das Herzstück des Generators besteht nur aus wenigen Zeilen Code: In dieser Schleife löscht er Zahlen, prüft, wie viele Lösungen es gibt, und setzt notfalls die Zahl wieder ein. Das wiederholt er so lange, bis er einmal alle Felder durchlaufen oder so viele Felder gelöscht hat, wie der gewünschte Schwierigkeitsgrad vorgibt.

Der gesamte Löschprozess spielt sich in einer großen `while`-Schleife ab. Zuerst pickt sich der Generator eine zufällige Koordinate im Sudoku (`r, c = unvisited.pop()`), bewahrt sich eine Kopie der dort stehenden Zahl auf (`copy = self.board[r][c]`) und entfernt sie danach (`self.board[r][c] = 0`). Dann checkt `solve()`, wie viele Lösungen es gibt. Lässt sich das Sudoku noch eindeutig lösen, entfernt das Programm die Zahl gefahrlos, anderenfalls setzt es wieder die Kopie ein (`self.board[r][c] = copy`) und löscht eine andere Zahl. Das wiederholt sich, bis `empty_cells` 0 oder die Liste der Koordinaten leer ist. Fertig ist der Sudoku-Generator.

Kleinigkeiten

Vielleicht ist es Ihnen schon aufgefallen, aber das Programm hat noch eine kleine Macke: Wenn `generate()` die Liste der Koordinaten durchlaufen hat, aber `empty_cells` noch nicht 0 ist, dann endet das Programm, ohne genügend Felder gelöscht zu haben. Dieser Fall tritt besonders häufig bei den höheren Schwierigkeitsstufen auf.

Damit man das Programm also nicht immer neu starten muss, befindet sich in `main()` eine kleine Schleife, die zehn Minuten lang versucht, ein Sudoku zu finden:

```
timeout = 600
start_time = time.time()
end_time = start_time + timeout
while time.time() < end_time:
    if sudoku.generate(difficulty):
        break
    else:
        sudoku.reset()
```

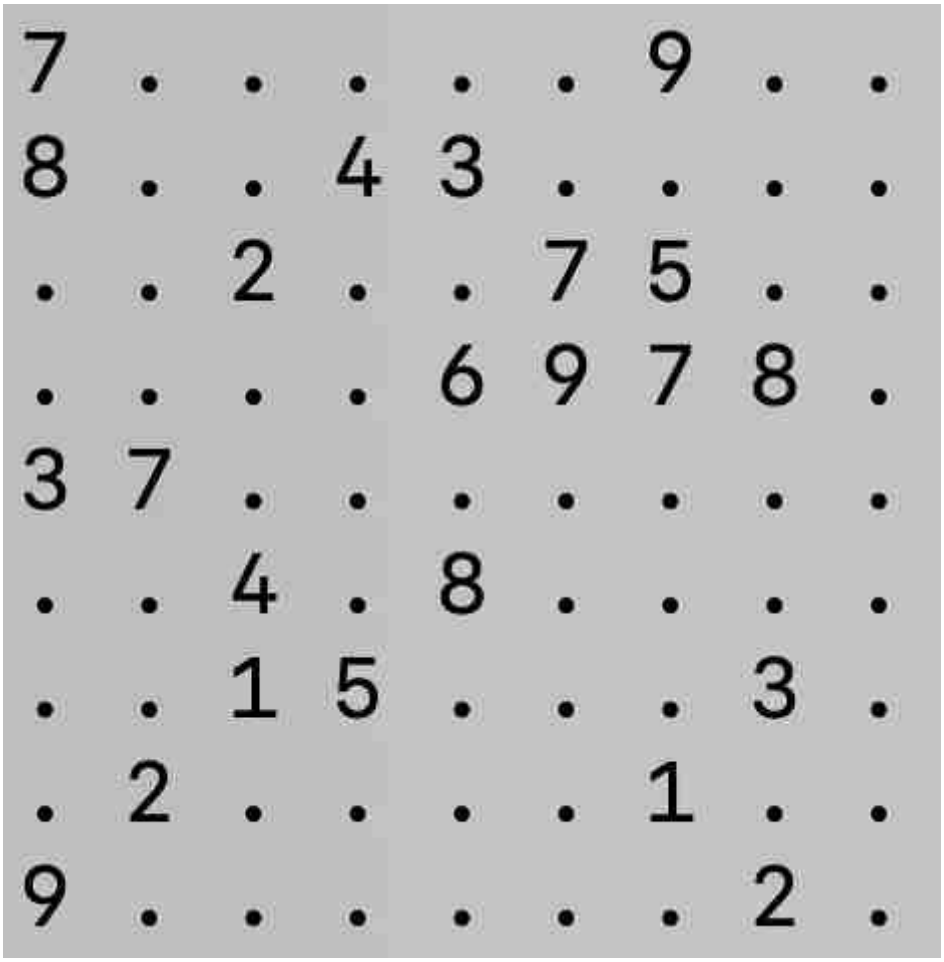
`timeout` regelt, wie lange das Programm insgesamt läuft, in diesem Fall 600 Sekunden. Wenn `generate()` innerhalb dieser Zeit ohne gültige Lösung zurückkehrt, löscht `reset()` das gesamte `board` und `generate()` darf sich nochmal daran wagen.

Um `main()` mitzuteilen, dass `generate()` kein Sudoku gefunden hat, fehlt am Ende der Funktion noch eine `if`-Abfrage:

```
if empty_cells > 0:
    print("Kein Sudoku gefunden.")
    return False
else:
    return True
```

Grafische Ausgabe

Bisher können Sie die fertigen Sudokus nur in der Konsole betrachten. Das ist weder schön noch nützlich, wenn Sie sie lösen wollen. Daher bekommt das Programm noch eine `toSVG()`-Funktion, die das komplette Sudoku schick als Vektorgrafik exportiert.



Zum Testen genügt die Konsolenausgabe, zum Rätseln taugt sie dagegen weniger.

Die Funktion erstellt eine Vektorgrafik im SVG-Format, die am Ende in der Datei Sudoku.svg gespeichert wird. Ganz oben steht der SVG-Header, der das XML-Namespace und die Version für das SVG-Dokument definiert:

```
svg = '<svg xmlns="http://www.w3.org/2000/svg" version="1.1">'
```

Danach zeichnet die Grafik ein Viereck und füllt dieses mit weißer Farbe aus:

```
 svg += f'<rect x="0" y="0" width="{9 * cell_size}" height="{9 * cell_size}"
fill="white" />'
```

Eine `for`-Schleife zieht abwechselnd die Linien für die Spalten und Reihen, wobei jede dritte davon fettgedruckt wird (`line_width = 2 if i % 3 == 0 else 0.5`), um das typische Aussehen eines Sudokus zu erreichen. Beide Befehle sehen ähnlich aus, daher folgt exemplarisch der für die Reihe:

```
svg += f'<line x1="{i * cell_size}" y1="0"  x2="{i * cell_size}" y2="{9 *
cell_size}" style="stroke:{line_color}; stroke-width:{line_width}" />'
```

Diese zieht eine Linie von den Koordinaten (x1, y1) zu (x2, y2). Mit dem `style`-Parameter legt man das Erscheinungsbild der Linie fest, zum Beispiel welche Breite (`stroke-width`) oder welche Farbe (`stroke:{...}`) sie haben soll.

Anschließend durchläuft eine weitere `for`-Schleife das gesamte Sudoku und platziert alle Zahlen:

```
svg += f'<text x="{(column + 0.5) * cell_size}" y="{(row + 0.5) * cell_size}"
style="font-size:20; text-anchor:middle; dominant-baseline:middle">
{str(self.board[row][column])}
  </text>'
```

Mithilfe von `text-anchor` und `dominant-baseline` richtet die Funktion die Zahlen in der Mitte ihrer Felder aus.

Wenn Sie das Programm nun ausführen, exportiert das Programm ein schönes klares Sudoku. Sie finden es in einer Datei wie zum Beispiel sudoku-20230317T163934-4.svg wieder. Es enthält als Namen das aktuelle Datum, die Uhrzeit und die Schwierigkeitsstufe.

Verbesserungspotenzial

Das ist aber nicht das Ende der Geschichte: Fühlen Sie sich hiermit ruhig dazu ermutigt, am Code herumzuspielen, die Schwierigkeiten nach persönlicher Präferenz anzupassen und eigene Funktionen zu implementieren. Zum Beispiel könnten Sie das Programm ein wenig aufbauschen, damit es Sudokus der Größen 4 × 4, 6 × 6 oder 12 × 12 erzeugt. Oder **das Programm ein wenig beschleunigen und optimieren [12]**, um Sudokus mit nur 17 oder 18 Zahlen zu finden. Ein Fleißsternchen gibt es für Sudokus mit abstrakten Formen.

(**wid [13]**)

URL dieses Artikels:

https://www.heise.de/-8645547

Links in diesem Artikel:

[1] https://www.heise.de/ratgeber/Einen-Sudoku-Generator-in-Python-programmieren-8645547.html

[2] https://www.heise.de/ratgeber/DB-Management-Datendubletten-mit-Python-entfernen-7166150.html

[3] https://www.heise.de/hintergrund/Python-im-Browser-Wortraetsel-mit-PyScript-programmieren-7193701.html

[4] https://www.heise.de/ratgeber/Python-Mit-vaex-grosse-Datenmengen-verwalten-7066766.html

[5] https://www.heise.de/ratgeber/Wie-Sie-Netzwerkmitschnitte-mit-Python-und-Scapy-auswerten-

7168114.html

[6] https://www.heise.de/hintergrund/Google-Colab-Wie-Sie-Python-Skripte-mit-Eingabefeldern-anpassen-7142452.html

[7] https://www.heise.de/ratgeber/Algorithmen-mit-Tabellenkalkulation-Python-und-CAS-erklaert-7121997.html

[8] https://www.heise.de/hintergrund/Python-und-Django-Einstieg-in-serverseitige-Webprogrammierung-6300113.html

[9] https://github.com/Periculum/SudokuGenerator

[10] https://www.heise.de/ct/

[11] https://github.com/Periculum/SudokuGenerator

[12] https://github.com/607011/sudokuplusplus

[13] mailto:wid@heise.de