THE SUDOKU

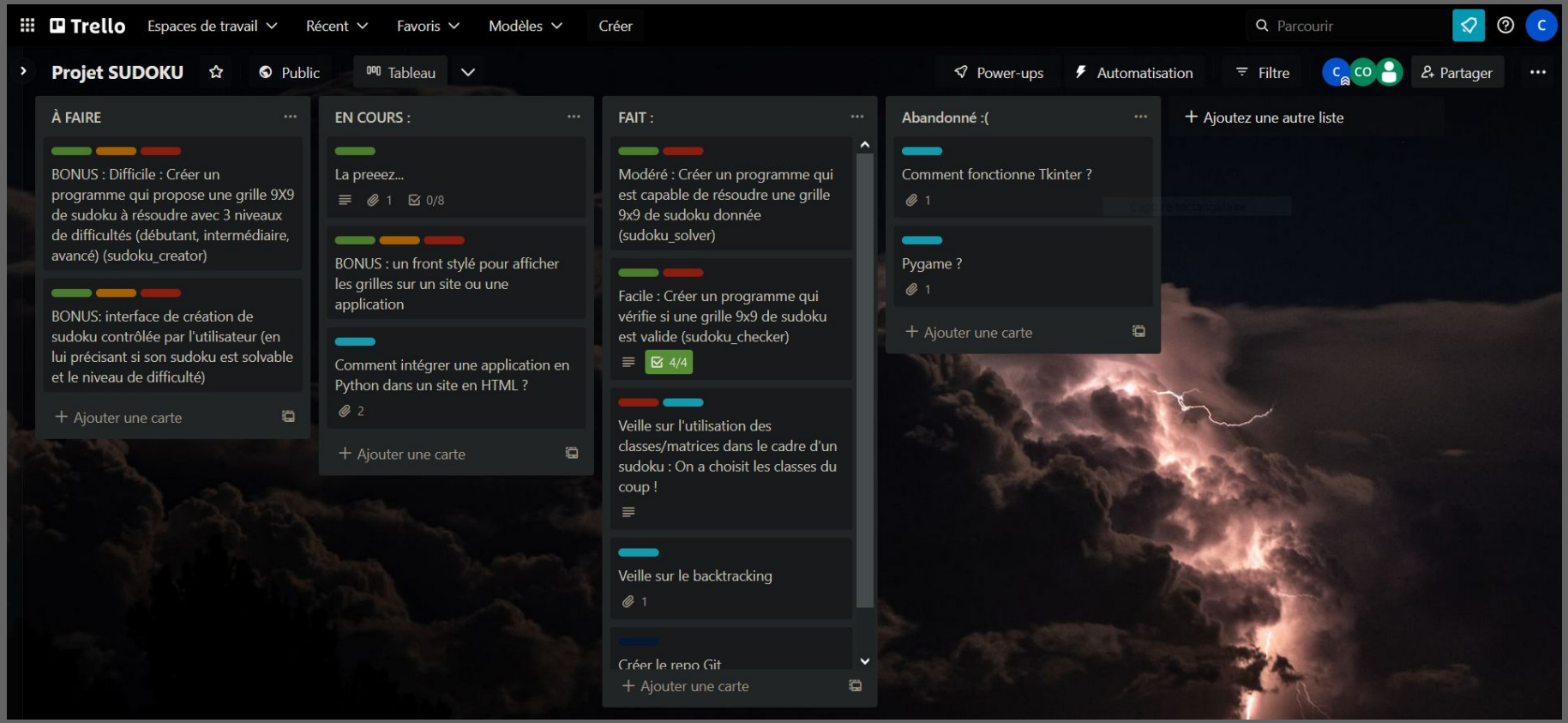
The Boss

We would like a sudoku for our web site

On it boss!

local newspaper

Organisation - Trello



Organisation - Veilles

Backtracking

! Algorithme : Série d'instruction suivies étapes par étapes par une machine, et servant à résoudre un problème.

Le backtracking est une famille d'algorithmes utilisés pour résoudre des problèmes algorithmiques, notamment de **satisfaction de contraintes** (optimisation ou décision).

CSP (Constraint Satisfaction Problem) : type de problèmes mathématiques où l'on cherche des états ou des objets satisfaisant un certain nombre de contraintes ou de critères. Ils font l'objet de nombreuses recherches, notamment en intelligence artificielle.

Ces algorithmes permettent de tester l'ensemble des affectations potentielles du problème. Ils consistent à sélectionner une variable du problème, et pour chaque affectation possible de cette variable, à testes récursivement si une solution valide peut être construite à partir de cette affectation partielle. Si aucune solution n'est trouvée, la méthode revient sur les affectations qui auraient été faite précédemment.

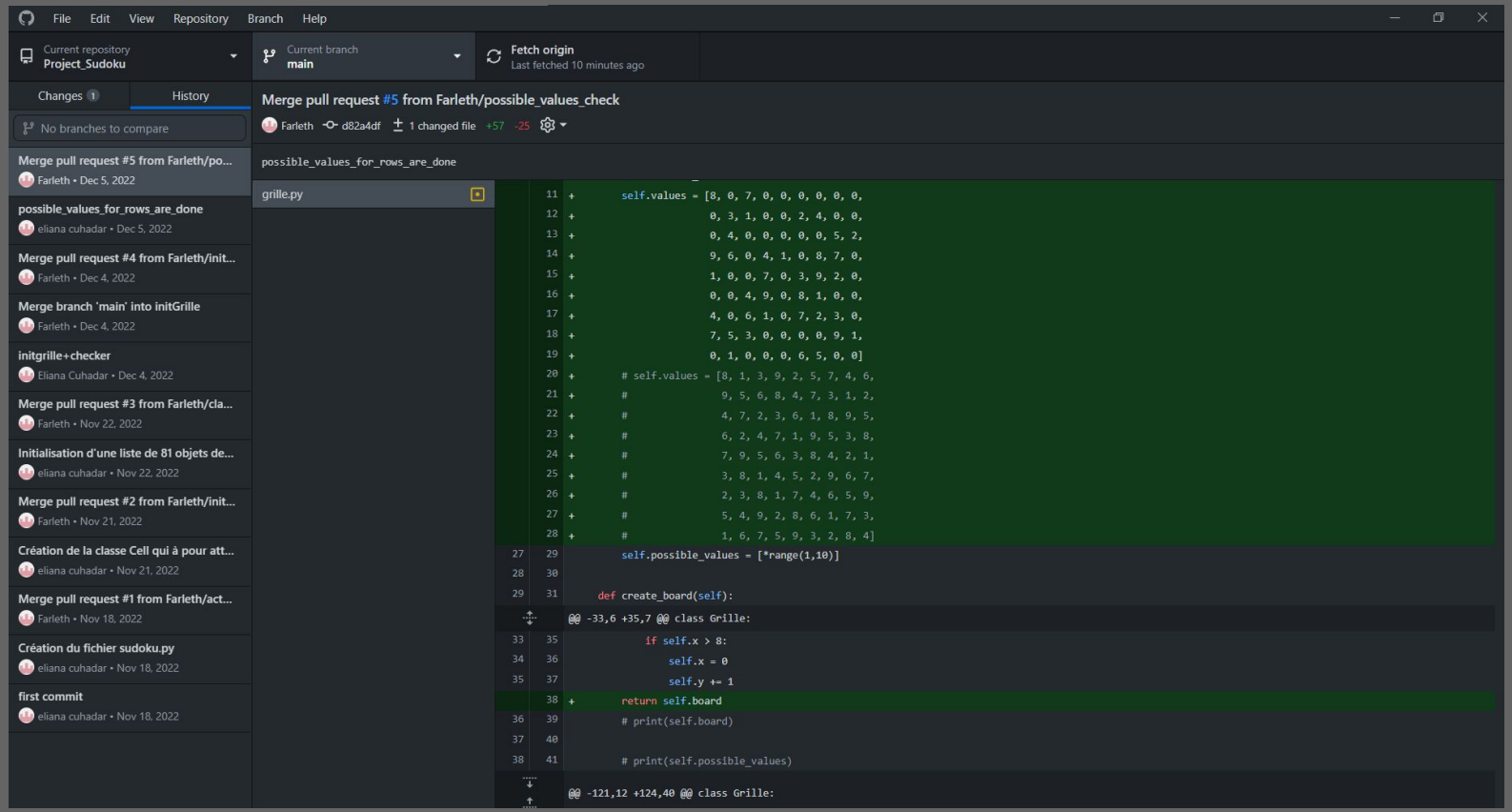
Initially, we start the backtracking from one possible option and if the problem is solved with that selected option then we return the solution else we backtrack and select another option from the remaining available options.

Backtracking is a form of recursion. This is because the process of finding the solution from the various option available is repeated recursively until we don't find the solution or we reach the final state.

There are three types of problems in backtracking :

- **Decision Problem** – In this, we search for a feasible solution.
- **Optimization Problem** – In this, we search for the best solution.

GitHub


File Edit View Repository Branch Help
Current repository
Project_Sudoku
Current branch
main
Fetch origin
Last fetched 10 minutes ago
Changes 1
History
No branches to compare
Merge pull request #5 from Farleth/po...
Farleth • Dec 5, 2022
possible_values_for_rows_are_done
eliana cuhadar • Dec 5, 2022
Merge pull request #4 from Farleth/init...
Farleth • Dec 4, 2022
Merge branch 'main' into initGrille
Farleth • Dec 4, 2022
initgrille+checker
Eliana Cuhadar • Dec 4, 2022
Merge pull request #3 from Farleth/cla...
Farleth • Nov 22, 2022
Initialisation d'une liste de 81 objets de...
eliana cuhadar • Nov 22, 2022
Merge pull request #2 from Farleth/init...
Farleth • Nov 21, 2022
Création de la classe Cell qui à pour att...
eliana cuhadar • Nov 21, 2022
Merge pull request #1 from Farleth/act...
Farleth • Nov 18, 2022
Création du fichier sudoku.py
eliana cuhadar • Nov 18, 2022
first commit
eliana cuhadar • Nov 18, 2022
Merge pull request #5 from Farleth/possible_values_check
Farleth d82a4df 1 changed file +57 -25
possible_values_for_rows_are_done
grille.py
+ self.values = [8, 0, 7, 0, 0, 0, 0, 0, 0,
+ 0, 3, 1, 0, 0, 2, 4, 0, 0,
+ 0, 4, 0, 0, 0, 0, 0, 5, 2,
+ 9, 6, 0, 4, 1, 0, 8, 7, 0,
+ 1, 0, 0, 7, 0, 3, 9, 2, 0,
+ 0, 0, 4, 9, 0, 8, 1, 0, 0,
+ 4, 0, 6, 1, 0, 7, 2, 3, 0,
+ 7, 5, 3, 0, 0, 0, 0, 9, 1,
+ 0, 1, 0, 0, 0, 6, 5, 0, 0]
+ # self.values = [8, 1, 3, 9, 2, 5, 7, 4, 6,
+ # 9, 5, 6, 8, 4, 7, 3, 1, 2,
+ # 4, 7, 2, 3, 6, 1, 8, 9, 5,
+ # 6, 2, 4, 7, 1, 9, 5, 3, 8,
+ # 7, 9, 5, 6, 3, 8, 4, 2, 1,
+ # 3, 8, 1, 4, 5, 2, 9, 6, 7,
+ # 2, 3, 8, 1, 7, 4, 6, 5, 9,
+ # 5, 4, 9, 2, 8, 6, 1, 7, 3,
+ # 1, 6, 7, 5, 9, 3, 2, 8, 4]
self.possible_values = [*range(1,10)]
def create_board(self):
@@ -33,6 +35,7 @@ class Grille:
if self.x > 8:
self.x = 0
self.y += 1
+ return self.board
# print(self.board)
# print(self.possible_values)
@@ -121,12 +124,40 @@ class Grille:

Sudoku Checker

Comment s'est-on Organisé ??

Sudoku Checker Step 1

```python
# Notre Sudoku.
sudoku = np.array([[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],
                   [9, 1, 2, 3, 4, 5, 6, 7, 8],
                   [3, 4, 5, 6, 7, 8, 9, 1, 2],
                   [6, 7, 8, 9, 1, 2, 3, 4, 5],
                   [8, 9, 1, 2, 3, 4, 5, 6, 7],
                   [2, 3, 4, 5, 6, 7, 8, 9, 1],
                   [5, 6, 7, 8, 9, 1, 2, 3, 4]])


# On détermine la position des groups.
A =  [list(sudoku[i][0:3]) for i in range(3)]
B =  [list(sudoku[i][3:6]) for i in range(3)]
C =  [list(sudoku[i][6:9]) for i in range(3)]

D =  [list(sudoku[i+3][0:3]) for i in range(3)]
E =  [list(sudoku[i+3][3:6]) for i in range(3)]
F =  [list(sudoku[i+3][6:9]) for i in range(3)]

G =  [list(sudoku[i+6][0:3]) for i in range(3)]
H =  [list(sudoku[i+6][3:6]) for i in range(3)]
I =  [list(sudoku[i+6][6:9]) for i in range(3)]
```

Sudoku Checker Step 2

```python
# Fonction qui tcheque les lignes.
def tcheck_rows():
    # on insert dans une liste toutes les lignes.
    liste = np.asarray([])
    for i in range(9):
        a = sudoku[i]
        np.append(liste, a)
    # On vérifie s'il y a des doublons dans nos colonnes.
    if not len(liste) == len(set(liste)):
        return False
    return True
```

Sudoku Checker Step 3

```python
# Fonction qui tcheque les colonnes.
def tcheck_column():
    # on insert dans une liste toutes les colonnes.
    for i in range(9):
        liste = []
        for j in range(9):
            a = sudoku[j][i]
            liste.append(a)
    # On vérifie s'il y a des doublons dans nos colonnes.
    if not len(liste) == len(set(liste)):
        return False
    return True
```

Sudoku Checker Step 4

```
# Fonction qui tchek tous les groupes.
def tcheck_all_group():
    liste_group = [A,B,C,D,E,F,G,H,I]
    for i in liste_group:
        # On convertie la liste à 2 dimensions en 1 dimension.
        flat_list = itertools.chain(*i)
        convert_liste = (list(flat_list))
        if not len(convert_liste) == len(set(convert_liste)):
            return False
            break
    return True
```

Sudoku Checker Step 5

```python
# On créer le sudoku checker.
def sudoku_checker():
    point = 0
    if tcheck_column():
        point +=1
    if tcheck_rows():
        point +=1
    if tcheck_all_group():
        point +=1
    if point == 3:
        print("Le Sudoku est valide !")
    else:
        print("Le Sudoku est invalide !")
```

And the needed set-up

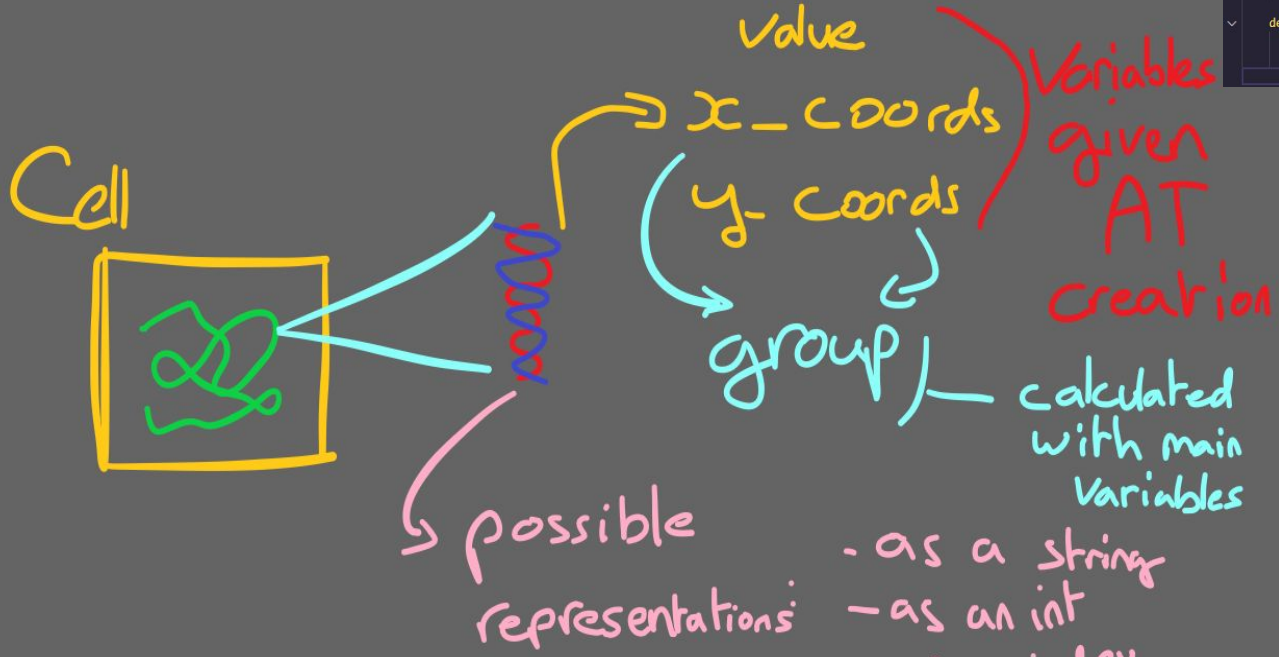
The prerequisites for a sudoku using Object Oriented Programming are as follows:

- A Cell class with all the needed info `class Cell:`
- A Grid class that instantiates all the cells of a sudoku board `class Grille:`
  - It should contain all of the needed functions to check if the sudoku is valid and to solve it

```
def __init__(self, values):
def init_sudoku_grid(self):
def all_unique(self, list_of_cells):
def check_row(self):
def check_column(self):
def check_groups(self):
def check_grid(self):
def display_grid(self):
def find_empty(self):
def solver_check_row(self, checking_cell : Cell):
def solver_check_column(self, checking_cell : Cell):
def solver_check_valid(self, cell):
def solver_check_groups(self, checking_cell : Cell):
def solve(self):
```

The Cell class

```python
class Cell:

    def __init__(self, value : int, x_coords : int, y_coords : int):
            self.value = value
            self.x_coords = x_coords
            self.y_coords = y_coords
            self.group  : int = ((y_coords//3) * 3) + x_coords//3

    def __str__(self) -> str:
        return f'|x{self.x_coords}-y{self.y_coords}-g{self.group}-v{self.value}|'

    def __repr__(self) -> str:
        return self.__str__()

    def __int__(self):
        return self.value

    def __index__(self):
        cellidx = self.x_coords + self.y_coords * 9
        return cellidx
```



The Grid class

Grid: → Cell: is given a list of values

1) to give to cells

2) I call 81 cells!

3) You are x=0 y=0 v=a number between 1 and 9

×81

Every cell is assigned a position and a value from the given list →

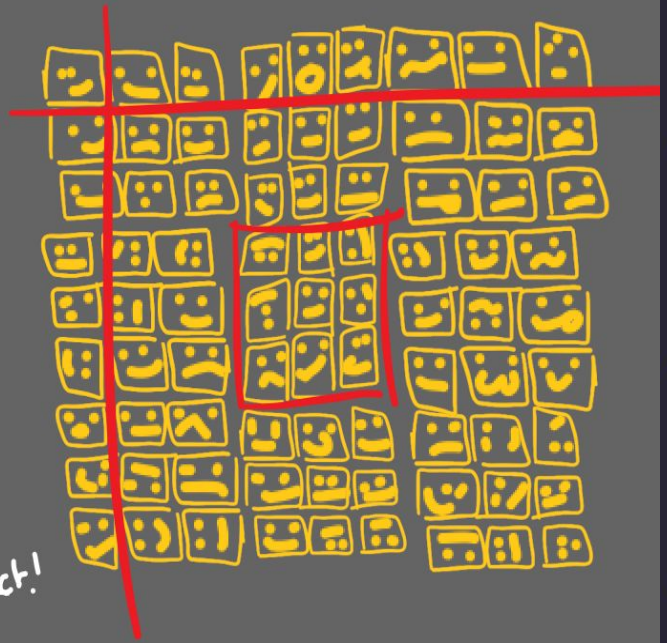
x=0 y:0 → v=? g=0

x=8 v=? y:2 g:2

```python
from sudoku import Cell

class Grille:

    def __init__(self, values):
        self.x=0
        self.y=0
        self.board = []
        self.size = 9
        self.values = values
        self.init_sudoku_grid()

    def init_sudoku_grid(self):
        #initialisation of the grid
        for i in range(0, self.size * self.size):
            #checking if the provided list of falues has the correct number of values
            if len(self.values) == self.size*self.size:
                #instantiating all cells in a list
                self.board.append(Cell(value=self.values[i], x_coords=self.x, y_coords=self.y))
                self.x += 1
                #not forgeting the line breaks for x and y
                if self.x == self.size:
                    self.x = 0
                    self.y += 1
            else: return f"The list of values doesn't match a sudoku grid. Please provide a list of 81 values"
```

The Checker using O.O.P

Grid can see if there are non-unique values in rows, columns and groups



If there are:

your sudoku is incorrect...

If not: It's correct!

```
def all_unique(self, list_of_cells):
        # check that all the values in a list of cells are unique
    return len(set(list_of_cells)) == len(list_of_cells)

def check_row(self):
    cell : Cell
    for y in range(self.size):
        row = []
        for x in range(self.size):
            cell = self.board[x + y*self.size]
            row.append(int(cell))
        if not self.all_unique(row):
            print(f"pb row at y={y}")
            return False


def check_column(self):
    cell : Cell
    for x in range(self.size):
        column = []
        for y in range(self.size):
            cell = self.board[x + y*self.size]
            column.append(int(cell))
        if not self.all_unique(column):
            print(f"pb col at x={x}")
            return False

def check_groups(self):
    cell : Cell
    for group in range(self.size):
        group_row = []
        for cell in self.board:
            if cell.group == group:
                group_row.append(int(cell))
        if not self.all_unique(group_row):
            print(f"pb group at g={group}")
            return False

def check_grid(self):
    if self.check_row() == False or self.check_column() == False or self.che
        return print("your sudoku is incorrect")
    else:
        return print("your sudoku is correct")
```

The Checker in action

```
  5, 4, 1, 8, 2, 9, 3, 7, 6,
  7, 8, 2, 6, 1, 3, 9, 5, 4,
  1, 9, 8, 4, 6, 7, 5, 2, 3,
  3, 6, 5, 9, 8, 2, 4, 1, 7,
  4, 2, 7, 1, 3, 5, 8, 6, 9,
  9, 5, 6, 7, 4, 8, 2, 3, 1,
  8, 1, 3, 2, 9, 6, 7, 4, 5,
  2, 7, 4, 3, 5, 1, 6, 9, 8
]
values = [
  6, 3, 9, 5, 7, 4, 1, 8, 2,
  5, 4, 1, 8, 2, 9, 3, 7, 6,
  7, 8, 2, 6, 1, 3, 9, 5, 4,
  1, 9, 8, 4, 6, 7, 5, 2, 3,
  3, 6, 5, 9, 8, 2, 4, 1, 7,
  4, 2, 7, 8, 3, 5, 8, 6, 9,
  9, 5, 6, 7, 4, 8, 2, 3, 1,
  8, 1, 3, 2, 9, 6, 7, 4, 5,
  2, 7, 4, 3, 5, 1, 6, 9, 8
]
```

```
your sudoku is correct
6 3 9 5 7 4 1 8 2
5 4 1 8 2 9 3 7 6
7 8 2 6 1 3 9 5 4
1 9 8 4 6 7 5 2 3
3 6 5 9 8 2 4 1 7
4 2 7 1 3 5 8 6 9
9 5 6 7 4 8 2 3 1
8 1 3 2 9 6 7 4 5
2 7 4 3 5 1 6 9 8
```

```
pb row at y=5
your sudoku is incorrect
6 3 9 5 7 4 1 8 2
5 4 1 8 2 9 3 7 6
7 8 2 6 1 3 9 5 4
1 9 8 4 6 7 5 2 3
3 6 5 9 8 2 4 1 7
4 2 7 8 3 5 8 6 9
9 5 6 7 4 8 2 3 1
8 1 3 2 9 6 7 4 5
2 7 4 3 5 1 6 9 8
```

The Solver

The prerequisites for a sudoku solver are as follows:

- A way to **find** empty cells `def find_empty(self):`
- A way to check if a cell has a **valid number** `def solver_check_valid(self, cell):`
- A **backtracking** algorithm to tie it all up `def solve(self):`

The Solver

```python
def solve(self):

    empty_cell = self.find_empty()
    if not self.find_empty():
        self.display_grid()
        return True

    for possible_num in range(1, self.size + 1):
        empty_cell.value = possible_num
        if self.solver_check_valid(empty_cell):

            if self.solve():
                self.display_grid()
                return True
        empty_cell.value = 0
    return False
```

1) is anybody empty?

yes :c

2) here, take this number

(n) Thank you!

3) or

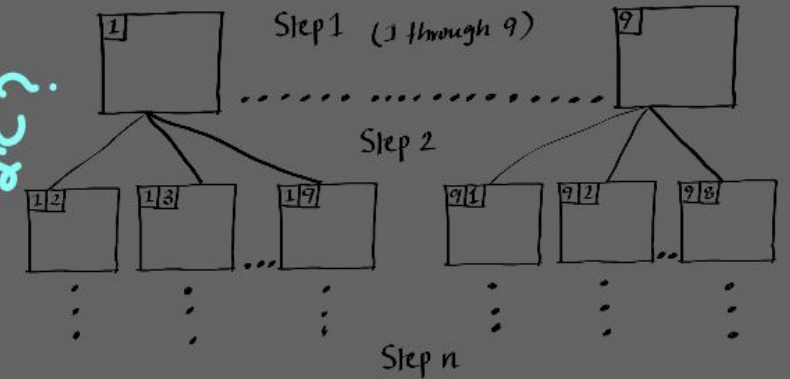
The number you gave was incorrect

back to 2) with n+1

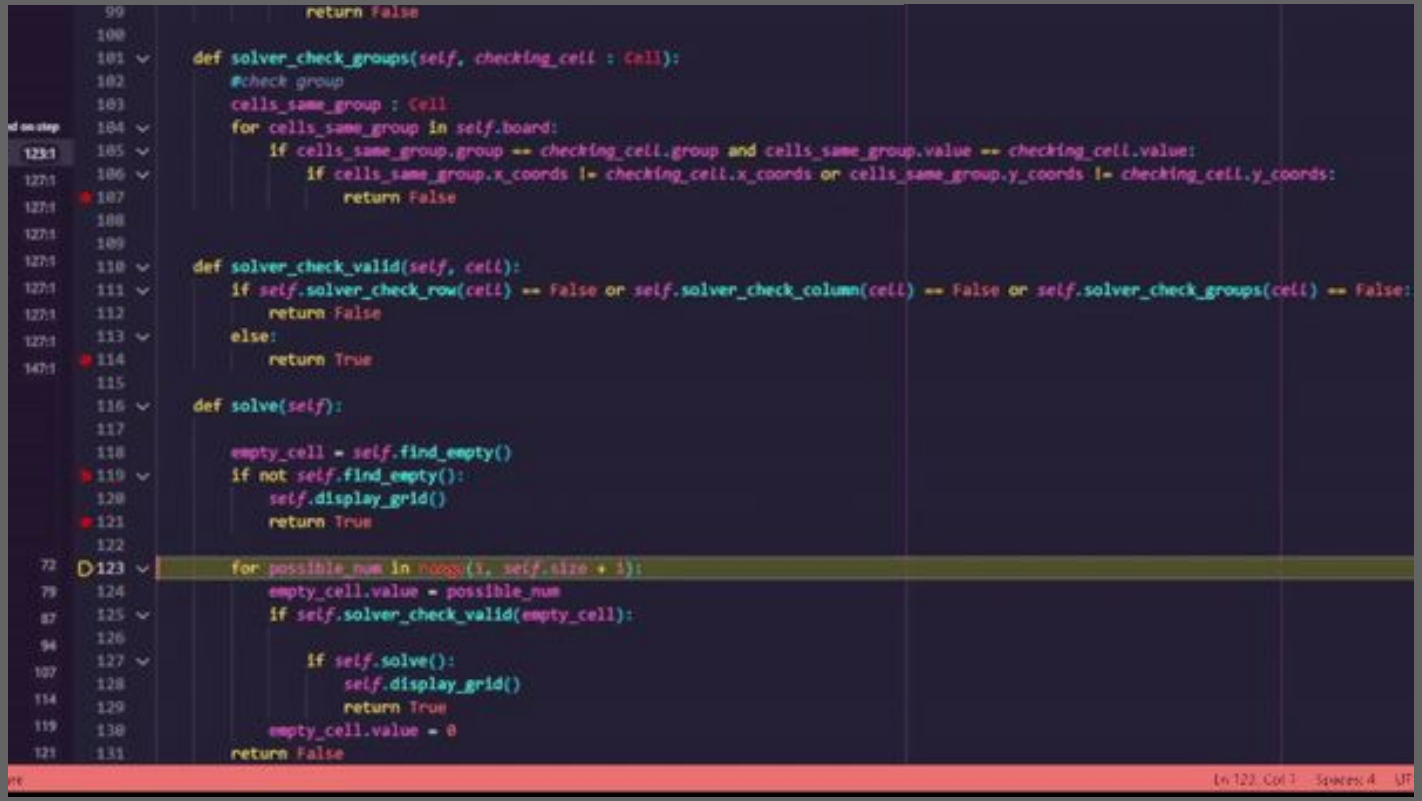
you gave a correct number

back to 1)

4) magic?.



The Solver in Action



Merci !