Comment résoudre une grille de Sudoku :

Pour cela, on va devoir coder deux méthodes : une méthode *solve* résolvant la grille de sudoku et *n_valide* qui va permettre de tester, à chaque fois qu'on positionne un nombre, si celui-ci est bien positionné ou non.

SOLVE

On définit *solve()*, qui va prendre une grille appelée *grid* (ici on a une variable globale). Il faut au début définir *grid* ici en une liste.

INDENTATION *Global* (placé devant grid) nous permet de travailler tout de même avec cette grille dans notre méthode.

INDENTATION Ensuite on crée une boucle *for y in range (9):* , avec y qui correspond a l'axe de ordonnées/les lignes qu'on va parcourir.

INDENTATION On crée une autre boucle *for x in range (9):*, avec x qui correspond a l'axe de abscisses/quand on va se déplacer de colonne en colonne.

INDENTATION Par la suite, nous créons une structure conditionnelle *if grid [y][x]* ==0, si la grille de y, x est égale à zéro (par rapport à la grille donnée en entrée). Si cette grille, à la place d'avoir un chiffre compris entre 1 et 9 inclus, elle contient 0, cela signifie que cette case n'a pas encore été résolue et donc qu'il va falloir la résoudre.

IDENTATION : On crée donc ici une nouvelle boucle *for n in range (1,10) :* correspondant donc à toutes les valeurs à tester sur cette case (de 1 à 9 car le 10 est exclu).

INDENTATION : On crée enfin notre structure conditionnelle *if n_valide (y,x,n) :* (paramètres y,x,n, pour voir si ce chiffre est déjà présent sur la ligne, la colonne et la sous grille) , INDENTATION alors on pourra affecter à grid de y,x la valeur de n *grille [y][x] =n* (le chiffre si il peut se positionner ici alors on va le mettre, et si ce n'est pas le cas, on ne rentrera pas dans cette condition et on passera au n supérieur), et ensuite il faut repartir dans *solve()* après avoir positionner un nombre pour refaire le même processus. Si jamais on ressort de *solve* cela signifie que le nombre n'était pas bon, une situation sans nombre possible à placer. Il faut donc faire un retour sur trace, retour en arrière, jusqu'à avoir des chiffres valides. Pour cela on va faire *grid [y][x]=0* , on remet la case égale à 0 pour pouvoir la retravailler plus tard. Finalement, on met un *return*, au niveau du *for n...*

➔ Méthode par récurrence

N_VALIDE

Quand on essaie de placer un chiffre n sur une case, il faut voir si ce chiffre n'est pas déjà présent sur la ligne, la colonne et la sous grille.

On code la méthode *def n_valide (y,x,n)* (Paramètres : x la ligne, y la colonne, et n le nombre qu'on essaie de placer). LIGNE INDENTATION on refait appel à *global grid,* puis on détermine si le nombre est valide sur sa ligne : *for x0 in range (len(grid))* : INDENTATION *if grid [y][x0] ==n* : alors on renvoie False INDENTATION *return False.* Pour la ligne x sur laquelle on est on va parcourir y et cela de 0 à 8 et si on trouve le n, on ne peut pas le placer d'où le *False.*

On détermine si le nombre est valide sur sa colonne : *for y0 in range (len(grid))* : INDENTATION *if grid [y0][x] ==n* : alors on renvoie False INDENTATION *return False.* Pour la colonne y sur laquelle on est on va parcourir x et cela de 0 à 8 et si on trouve le n, on ne peut pas le placer d'où le *False.*

On determine si le nombre est valide dans sa sous-grille (sous-grilles 3 par 3) ; Pour cela on prépare deux variables : *x0 = (x//3) * 3* et *y0 = (y//3) * 3.* (x0 et y0 nous donne ici les points de départ d'une sous

grille, là on nous somme dans le sudoku). Ensuite, on veut donc parcourir nos sous-grilles et pour cela on a besoin de deux boucles : i et j qui vont de 0 à 3 exclus donc de 0 et 2.

Quand notre grille sera complète, on va faire l'affichage dans la méthode solve, tout en bas. On sort de la double boucle et on écrit (i lignes, j colonnes et on fait print, le *end*= « » evite qu'à chaque élément, on fasse un saut de ligne)