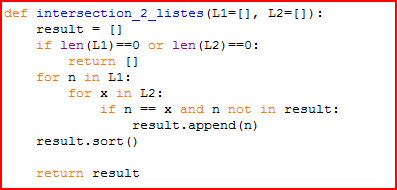
Explications sur les fonctions utilisées

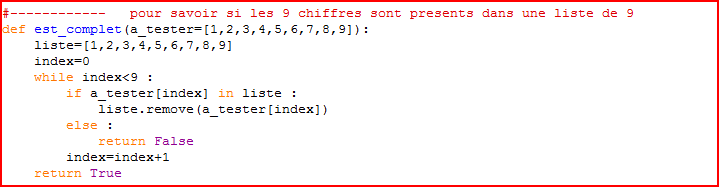
# Intersection



On passe à cette fonction deux paramètres qui sont des listes (L1 et L2)

* Si une des deux liste est vide (on teste si la longueur d’une des liste est nulle) alors l’intersection est vide et l’on renvoie une liste vide
* Sinon on compare tous les éléments de la première avec tous ceux de la deuxième et si l’on trouve un élément commun on le rajoute à la liste résultat à condition que l’on ne l’ai pas déjà rajouté. Une fois les comparaisons terminées, on renvoie la liste des éléments communs.

# Tests sur les listes

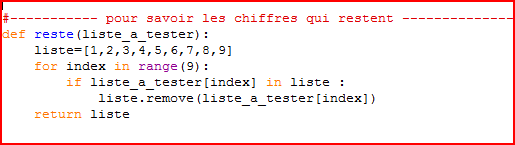


Fonction utilisée pour savoir si dans une liste on a bien tous les nombres de 1 à 9 une fois. Cette fonction renvoie True ou False

On passe à cette fonction un paramètre : la liste : a\_tester

Pour tous les éléments de la liste, obtenus en faisant varier index de 0 à 8 on teste si les éléments de la liste a\_tester sont dans la liste de départ : liste. S’il le sont on enlève l’élément de la liste de départ liste, sinon on renvoie False.

A la fin, si aucun élément ne manque on renvoie True



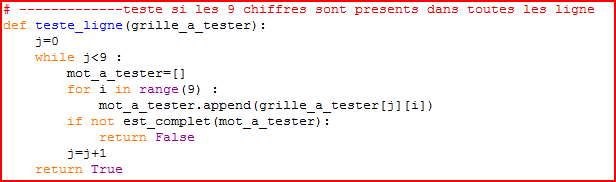
Fonction utilisée pour savoir quels sont les éléments qui restent par rapport à la liste des nombres de 1 à 9 : pour la grille de sudoku, cette fonction est utile pour connaître les possibilités qui restent dans une case en éliminant peu à peu tous les éléments qui sont impossibles

On passe à cette fonction un paramètre : la liste : liste\_a\_tester et la fonction renvoie les éléments restants.

On passe en revue tous les éléments de la liste\_a\_tester et s’ils sont dans la liste on les retire de celle-ci

A la fin on renvoi ce qui reste de la liste.

# Les tests de remplissage

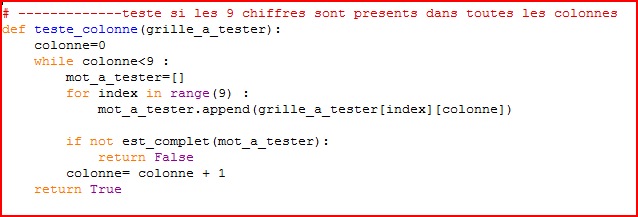


Fonction utilisée pour savoir si les 9 chiffres sont présents dans les 9 lignes de la grille.

On passe en paramètre la grille à tester et la fonction retourne True si les 9 chiffres sont présents dans toutes les lignes, elle retourne False sinon.

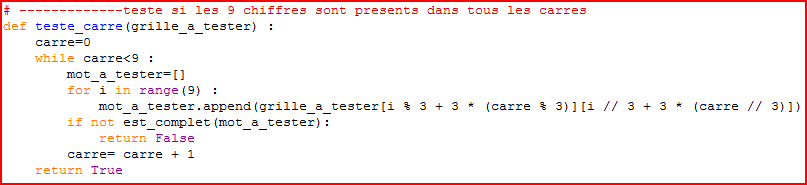
Cette fonction fait appel à la fonction « est\_complet » décrite précédemment.

Pour chaque ligne de la grille à tester, on crée une liste (mot\_a\_tester) est on teste si cette liste est complète ou non.



Fonction utilisée pour savoir si les 9 chiffres sont présents dans les 9 colonnes de la grille.

Même fonction que la précédente pour les colonnes



Fonction utilisée pour savoir si les 9 chiffres sont présents dans les 9 carrés de la grille.

Même fonction que la précédente pour les carrés.

[i % 3 + 3 \* (carre % 3)][i // 3 + 3 \* (carre // 3)]), permet d’obtenir la position [ligne][colonne] d’un élément de la grille en fonction du numéro du carré et de la position i de l’élément.

0

3

6

1

4

7

2

5

8

Position i de l’élément

0

3

6

1

4

7

2

5

8

0

3

6

1

4

7

2

5

8

0

3

6

1

4

7

2

5

8

0

3

6

1

4

7

2

5

8

0

3

6

1

4

7

2

5

8

0

3

6

1

4

7

2

5

8

0

3

6

1

4

7

2

5

8

0

3

6

1

4

7

2

5

8

0

3

6

1

4

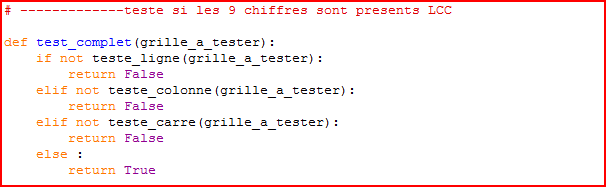
7

2

5

8

Position i du carré



Cette fonction reprend les 3 fonctions précédentes.

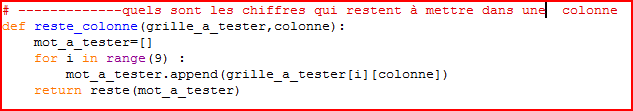
On lui passe en paramètre la grille à tester. Elle teste successivement si les 9 chiffres sont présents :

* Dans toutes les lignes
* Dans toutes les colonnes
* Dans tous les carrés

Elle retourne True si tous les chiffres sont présents dans toutes les lignes, toutes les colonnes et tous les carrés. Sinon elle retourne False.

Cette fonction permet de savoir si la grille de sudoku est bien remplie.

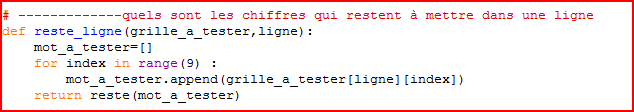
# Tests pour la résolution



Cette fonction va fournir la liste des éléments qui rentent à placer dans une colonne.

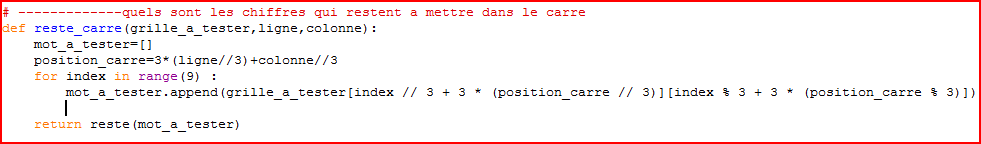
On lui passe en paramètre la grille à tester et un numéro de colonne.

Cette fonction crée une liste constituée des éléments de la colonne spécifiée : mot\_a\_tester, elle retourne la liste des éléments non encore placés en utilisant la fonction reste vue précédemment.



Cette fonction va fournir la liste des éléments qui rentent à placer dans une ligne.

Même fonction que la précédente, mais pour les lignes.



Cette fonction va fournir la liste des éléments qui rentent à placer dans un carré.

Même fonction que la précédente, mais pour les carrés.

On passe en paramètre :

* La grille à tester
* La ligne d’un élément
* La colonne de l’élément.

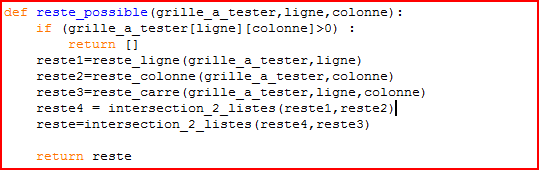
On crée une liste vide : mot\_a\_tester=[]

On détermine la position du carré en fonction de la ligne et de la colonne de l’élément :

position\_carre=3\*(ligne//3)+colonne//3

Pour tous les éléments du carré trouvé, on ajoute à la liste mot\_a\_tester tous les éléments du carré.

On renvoie le reste en utilisant la fonction reste (précédente)



Cette fonction va déterminer pour une case du sudoku quels sont les candidats possibles.

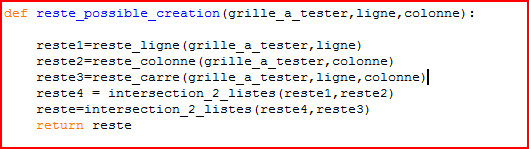
On passe en paramètre :

* La grille à tester
* La ligne d’un élément
* La colonne de l’élément.

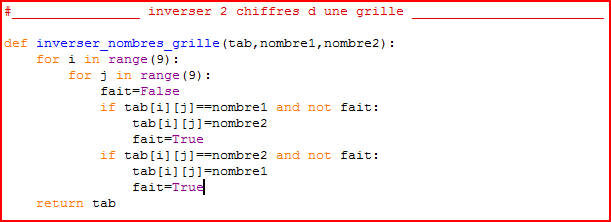
On détermine quels sont les candidats possibles pour une case de la grille (dans la ligne, dans la colonne et dans le carré). On fait ensuite l’intersection ce ces 3 ensembles pour pouvoir retourner le reste des candidats possibles.

Si la case est déjà remplie (valeur non nulle), on retourne une liste vide.

Fonctions pour la création des grilles (grilles simples par soustraction)



Même fonction que la précédente, mais qui ne renvoie pas une liste vide lorsque l’élément est présent.



Fonction créée pour inverser 2 chiffres d’une grille. Ceci permet de créer une grille différente de celle d’origine.

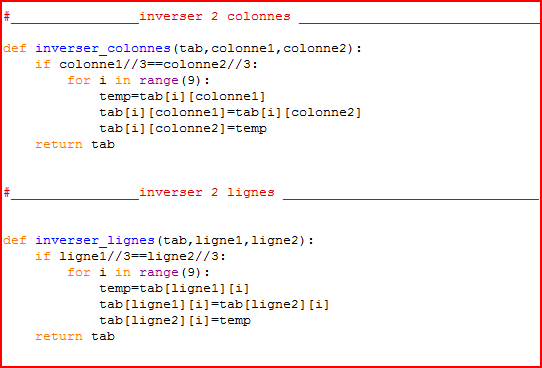
On passe en paramètre :

* Le nom de la grille (tab)
* Les deux chiffres à échanger

La fonction renvoie la grille avec les deux chiffres échangés.

Dans la grille, on passe en revue toutes les lignes (indice i) et toutes les colonnes (indice j). Si dans une case on trouve un des deux chiffres on l’échange avec l’autre. La variable « fait » est mise à False au départ et elle passe à True dès que l’échange a été fait, ceci pour ne pas échanger deux fois le contenu de la case lorsque l’on tombe sur nombre1.

Cette fonction n’est plus utilisée car la fonction «melange\_nombre\_grille » (voir plus loin) permet de mélanger tous les nombres d’un seul coup.



Fonction créée pour inverser 2 colonnes ou 2 lignes d’une grille. Ceci permet de créer une grille différente de celle d’origine.

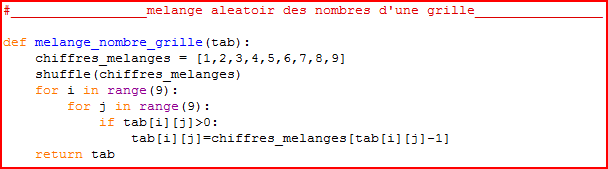
On passe en paramètre :

* Le nom de la grille (tab)
* Les deux lignes ou colonnes à échanger

La fonction renvoie la grille avec les deux lignes (ou colonnes) échangées.

La ligne : if colonne1//3==colonne2//3: (if ligne1//3==ligne2//3: )permet de vérifier que l’échange se fait dans les mêmes carrés, sinon on créerait une grille fausse.

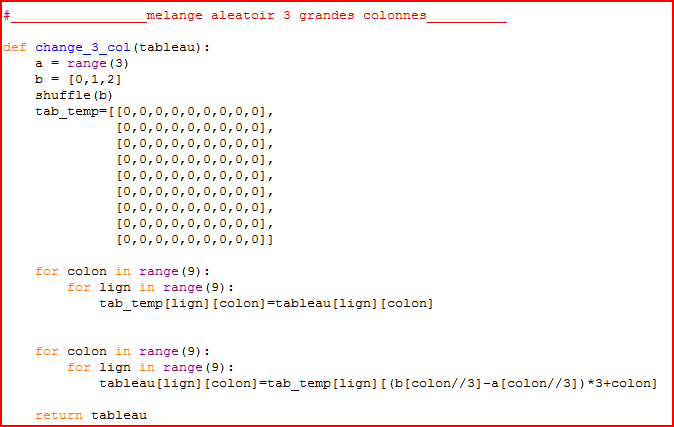
Pour le reste on fait simplement un échange en passant par une variable temporaire temp.



Mélanger aléatoirement les nombres d’une grille permet de créer des grilles différentes à partir d’une grille de départ.

On part d’une liste « chiffres\_melanges » : [1,2,3,4,5,6,7,8,9] que l’on mélange aléatoirement avec la fonction shuffle, son ordre est donc quelconque.

On passe en revue à l’aide de 2 boucles for imbriquées (indices i et j) tous les éléments de la grille tab et on remplace le chiffre de départ si celui-ci est positif (le zéro correspond à une case vide) par le contenu du tableau « chiffres\_melanges » dont le rang est le chiffre de départ -1 (pour commencer les indices à 0).



Mélanger aléatoirement 3 grandes colonnes permet de créer des grilles différentes à partir d’une grille de départ.

On part d’un tableau (grille de sudoku), on crée 2 listes contenant 0, 1, 2 :

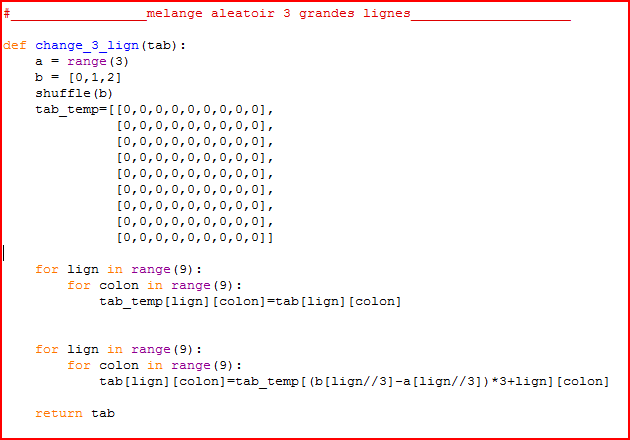
* a dans l’ordre
* b dans un ordre aléatoire après l’utilisation de la fonction shuffle.

On recopie le tableau dans un tableau temporaire : tab\_temp

Puis on recopie ce tableau temporaire dans le tableau de départ en mélangeant les 3 grandes colonnes. J’utilise pour cela la fonction [(b[colon//3]-a[colon//3])\*3+colon] qui va permettre de mélanger les colonnes par groupes de 3 : on pourra par exemple obtenir

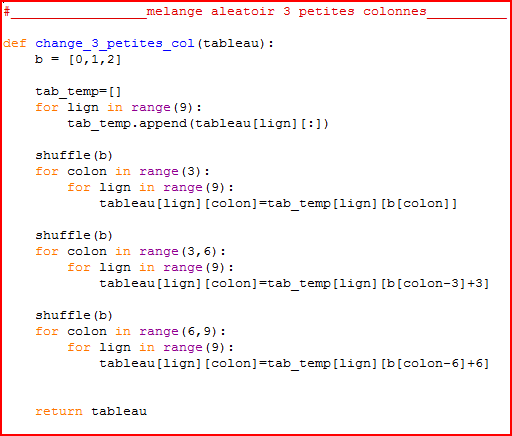
345012678 ce qui va permettre d’échanger les colonnes 012 avec les 345 la colonne 678 restant en place.

Il y a 3 ! =6 combinaisons possibles.



Mélanger aléatoirement 3 grandes lignes permet de créer des grilles différentes à partir d’une grille de départ.

C’est la même fonction que la précédente, mais appliquée aux lignes à la place des colonnes.

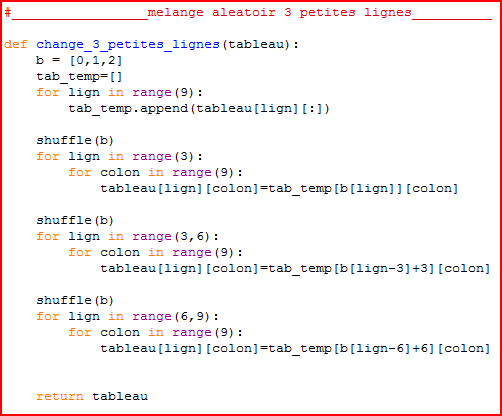


Cette fonction permet d’échanger aléatoirement les 3 petites colonnes des 3 grandes colonnes d’une grille. Ceci permet de créer des grilles différentes à partir d’une grille de départ.

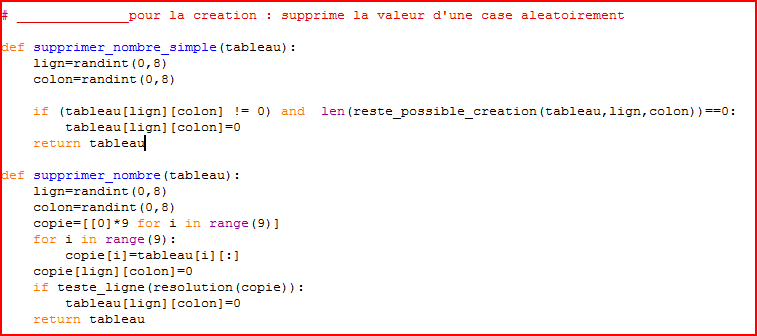
On commence par copier le tableau (grille) dans un tableau temporaire.

On mélange b avec la fonction shuffle.

On recopie les 3 premières colonnes en changeant l’ordre de celles-ci (on remplace le numéro de colonne par la valeur de b d’indice le numéro en question)

Idem avec les 3 colonnes suivantes et les 3 dernières.

Idem mélange des 3 petites colonnes, mais pour les lignes.



Ces deux fonctions peuvent être utilisées pour la création de grilles nouvelles.

On leur passe en paramètre une grille complète ou partiellement complète elles renvoient cette même grille avec eventuellement une case mise à 0.

Supprimer\_nombre simple :

* tire 2 coordonnées (ligne,colonne) aléatoirement entre 0 et 8
* Si à ces coordonnées il y a une valeur que l’on peut trouver directement, on la supprime de la grille. Cette fonction permet de générer des grilles pour débutants.

Supprimer\_nombre :

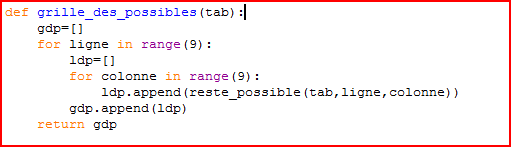
* tire 2 coordonnées (ligne,colonne) aléatoirement entre 0 et 8
* on crée une copie de la grille
* on supprime la valeur de la case tirée aléatoirement sur la copie
* On essaie de résoudre (par les fonctions de résolution) cette copie de grille.
* Si on a pu résoudre, on enlève la valeur de la grille et on la retourne.



La fonction qui permet de créer une grille aléatoirement :

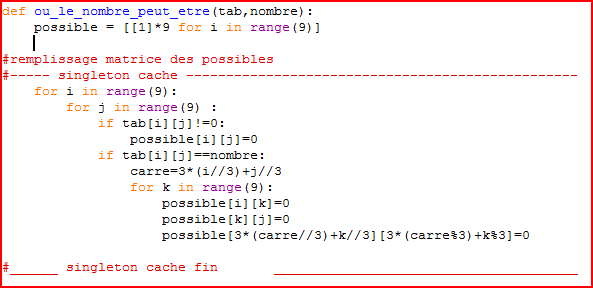
* On part d’une grille quelconque (pleine ou partiellement remplie)
* On utilise les fonctions de mélange (chiffres, colonnes, lignes, petites colonnes et petites lignes) pour créer une grille différente.
* On supprime ensuite (avec fonction supprimer nombre) des valeurs de la grille
* On recopie la grille créée pour avoir la grille de départ.
* Et l’on renvoie ensuite ces 2 mêmes grilles. La grille de départ aura une couleur d’affichage différente et ne pourra pas être effacée.

# LES FONCTIONS DE RESOLUTION



Cette fonction en elle-même ne résout rien, mais elle crée un tableau gdp image de la grille de départ qui au lieu de contenir les valeurs contient la liste des prétendants.

Pour chaque ligne de la grille, on crée une ligne des possibles (ldp) contenant tous les restes possibles de la ligne et on l’ajoute à la grilles des possibles (gdp). Lorsque celle-ci est pleine, la fonction la retourne.



Première fonction de résolution : ou\_le\_nombre\_peut\_etre

Cette fonction va rechercher dans quelles cases un nombre peut être placé.

On crée pour cela un tableau « possible » de même dimension que la grille qui contiendra 0 si le nombre ne peut pas être à la position et 1 si le nombre peut être à la position.

A départ toutes les cases sont possibles et l’on place la valeur 1 dans toutes les cases.

Ensuite au fur et à mesure on va éliminer les possibilités en placant des 0.

Première partie : recherche d’un singleton caché

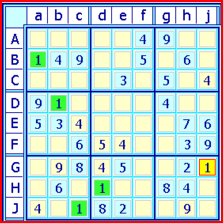
On étudie une à une les cases de la grille (une boucle for pour les lignes : i , une boucle for pour les colonnes : j)

* Si la grille contient déjà une valeur (if tab[i][j]!=0), la valeur ne peut être dans la case et donc elle est notée comme impossible : possible[i][j]=0
* Si la grille contient le nombre que l’on teste :
  + for k in range(9):
  + possible[i][k]=0 on met toute la ligne à 0
  + possible[k][j]=0 on met toute la colonne à 0
  + possible[3\*(carre//3)+k//3][3\*(carre%3)+k%3]=0 et tout le carré

Ainsi s’il ne reste plus qu’un seul 1 dans une région (ligne, colonne ou carré) ce sera la valeur



Deuxième partie : élimination indirecte

Le 1 de la ligne D impose un 1 dans la colonne g dans le rectangle milieu droit.

Le 1 du rectangle inférieur droit ne peut donc pas être :

* Colonne g
* Lignes H et J

La seule possibilié : G,j

C’est cela que nous allons rechercher dans cette deuxième partie.

J’ai numéroté les grands carrés de 0 à 8 le 0 correspondant à celui en haut à gauche, le 8 en bas à droite.

On passe en revue tous les carrés. Dans un carré si on rencontre une case possible pour notre valeur :

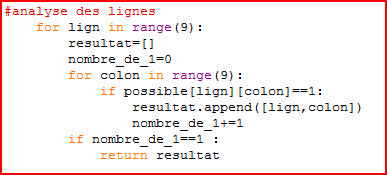
* on enregistre la position de sa colonne dans début\_colonne si c’est la première valeur rencontrée (on a alors debut\_colonne ==10)
* Si c’est une des suivantes, on enregistre sa position dans fin\_colonne.

On connaît alors la colonne de la première possibilité et celle de la dernière.

Si elles ont sur la même colonne, cela veut dire que la valeur cherchée ne se retrouvera pas dans cette colonne dans un autre carré.

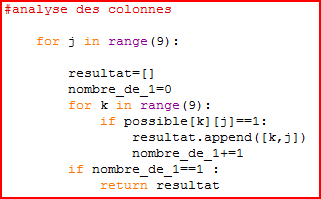
On met donc toute cette colonne à 0 sauf dans le carré concerné.

On recommence l’opération pour les lignes. Le raisonnement est identique. On aurait pu transposer la grille et refaire d=strictement le même programme.

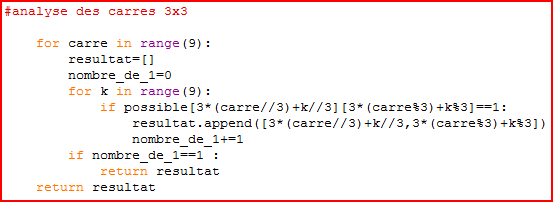


On analyse ensuite les différentes lignes de la grille à la recherche d’un 1 isolé sur la ligne :

* Si on rencontre un 1, on enregistre ses coordonnées (ligne,colonne) dans un tableau de résultats.
* Si sur la ligne on a trouvé un seul 1, on renvoie ses coordonnées.



Même chose avec les colonnes

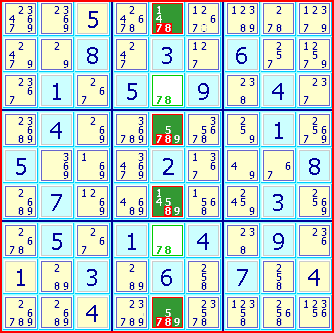


Même chose avec les carrés.

# Groupes nus et cachés

A partir de ce niveau de difficulté, j’ai abandonné le tableau des 1 indiquant où il était possible de trouver une valeur au profit d’un tableau des possibles, indiquant pour chaque case une liste des prétendants et une liste vide si la case a une valeur trouvée.

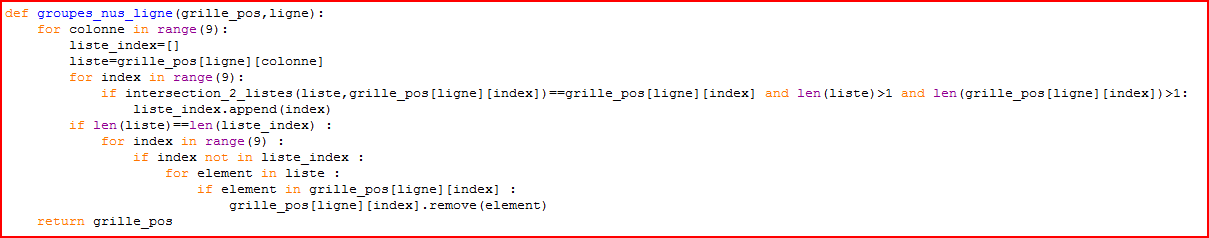
## Groupes nus

Lorsque l’on rencontre dans une même région une paire de (comme ici 7-8) en 2 endroits sur la même colonne, on est sûr que les 2 valeurs seront dans l’une des 2 cases et que par conséquent elles ne seront pas ailleurs dans la région. On pourra donc éliminer cette paire de tous les prétendants de la région excepté dans les 2 endroits où on les a trouvés.

J’ai dans un premier temps créé une fonction retrouvant les paires puis une autre les triplets, car le raisonnement est le même avec 3 triplets identiques dans une même région. (Voir FonctionsEnPlus)

Je me suis ensuite rendu compte qu’un groupe abc pouvait être incomplet :

Si on a abc puis abc puis ab, sans qu’il y ait égalité des groupes, on pouvait quand même éliminer des candidats de la région. De plus on pouvait faire 4 groupes de 4 ou 5 groupes de 5… et il aurait fallu faire autant de fonctions différentes. Je me suis donc consacré à la recherche des groupes nus quel que soit leur taille.



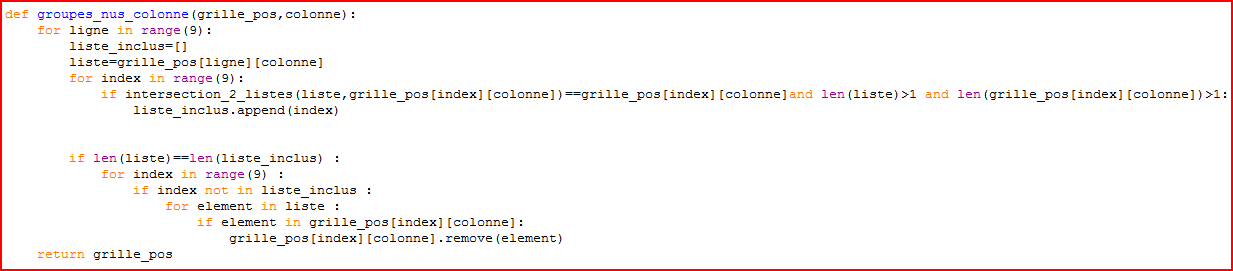
Recherche de groupes nus dans une ligne :

Les paramètres passés sont la grille des possibles et la ligne. La fonction modifie la grille des possibles.

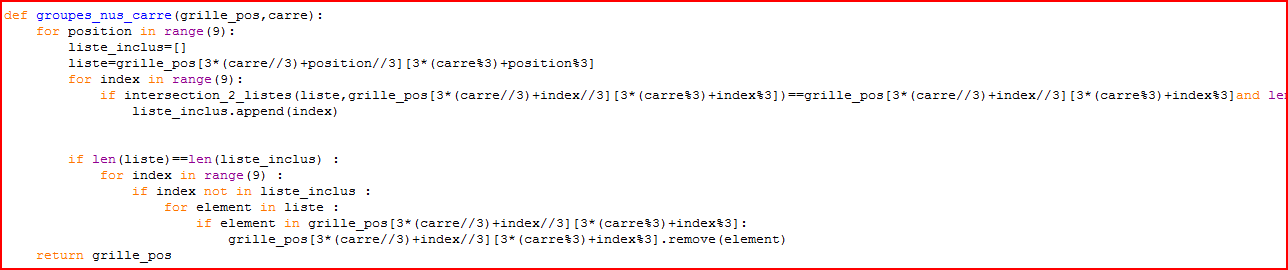
On teste toutes les colonnes, pour chaque colonne :

* La liste est définie comme les candidats de la case de la ligne et de la colonne
* On crée une liste d’index vide
* On teste tous les éléments de la ligne (y compris ceux de la colonne considérée) :
  + Si les candidats de la case sont inclus dans ceux de la liste et la case non vide et la liste non vide, on rajoute l’index (n° de la colonne) à la liste des index.
* Si la liste contient autant d’éléments que l’on a trouvé de colonne, on a mis à jour un groupe nu, les éléments de cette liste sont donc dans la liste des index : liste\_index. Ils ne sont donc pas dans les autres cases de la ligne.

On élimine donc tous les éléments de la liste des autres cases de la ligne de la grille des possibles.

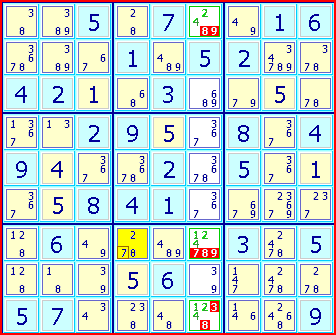


Idem pour les colonnes



Idem pour les carrés

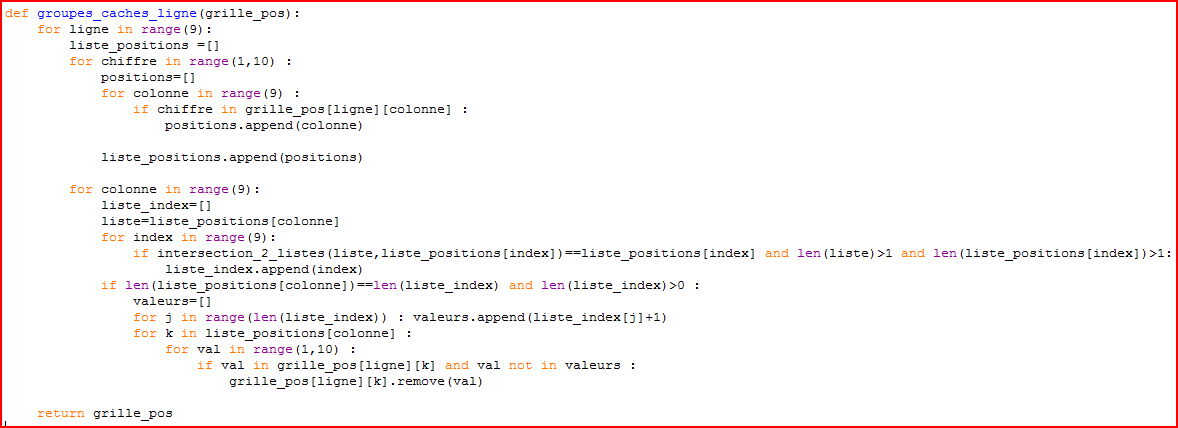
## Groupes cachés

  
On cherche cette fois des groupes de prétendants qui sont cachés parmi d’autres prétendants.

Comme ici le groupe 124 que l’on retrouve 2 fois entier et une fois partiellement.

Ces 3 éléments sont donc forcément dans les 3 cases et par conséquent les autres prétendants n’y sont pas.

L’idée pour trouver ces groupes : créer une liste des positions de chaque élément et comparer les éléments de la liste.



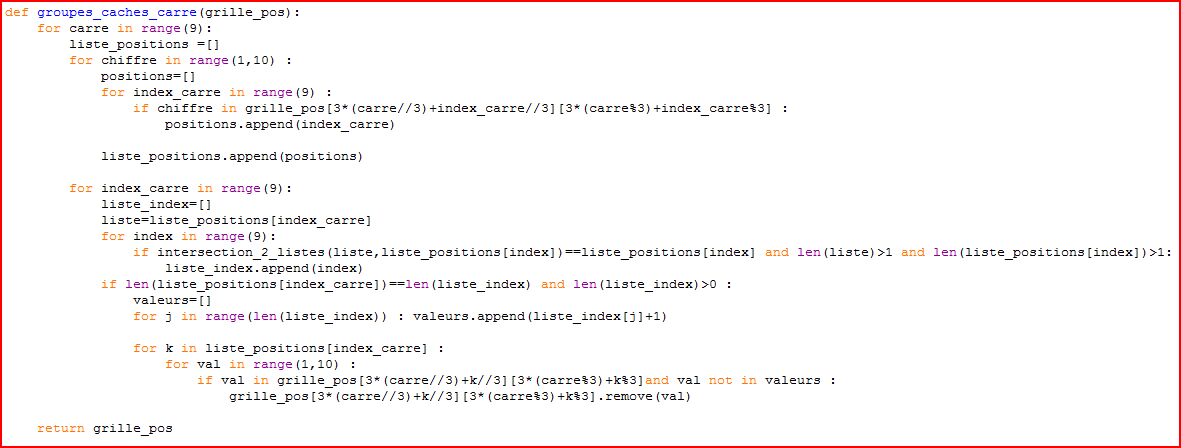
On teste une à une toutes les lignes de la grille. Pour chaque ligne :

On crée la liste de toutes les positions possibles pour chacune des valeurs de 1 à 9 : liste\_position

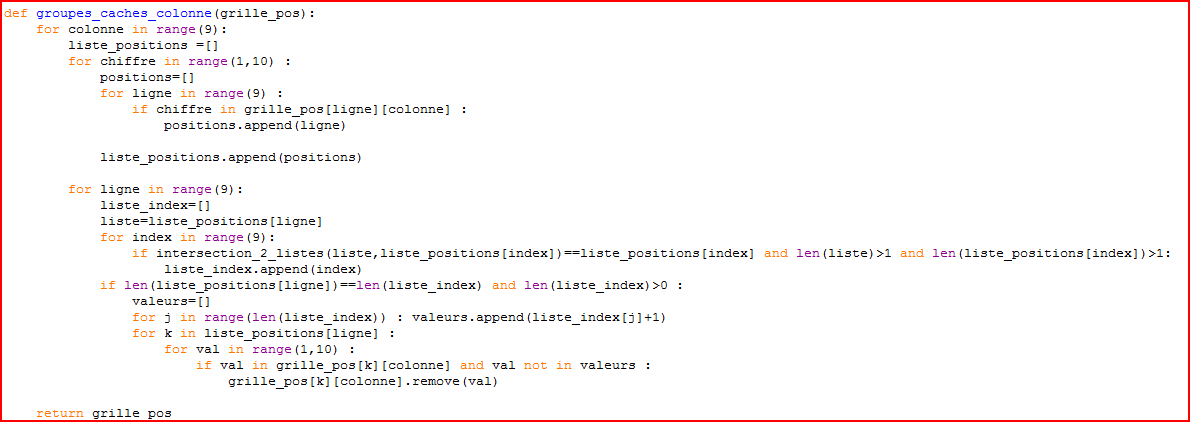
On va ensuite comparer les listes de position aux autres et si l’on en trouve une incluse dans la référence, on va enregistrer sa valeur (index) dans la liste des index. L’index correspond à 1 près à la valeur du candidat.

S’il y a autant de groupes que de candidats dans la liste :

* On va créer la liste des valeurs en décalant de 1 les index de la liste des index
* Pour tout k de la liste des positions :
  + Pour toutes les valeurs val qui ne sont pas dans la liste des valeurs, on va les oter de la liste des possibles



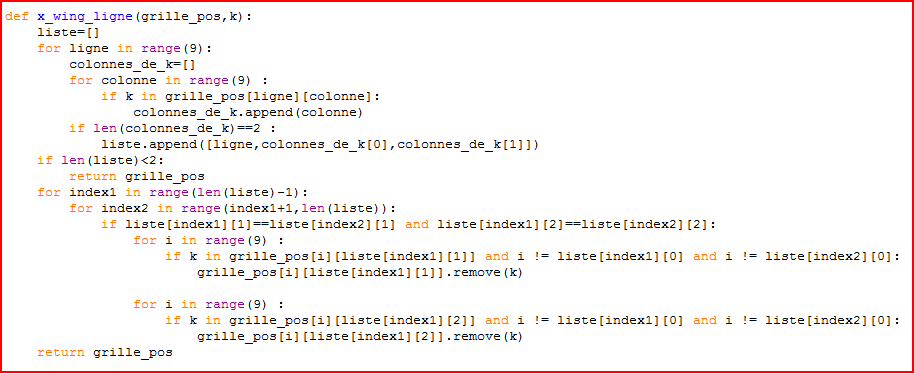
Idem pour les carrés



Idem pour les colonnes.

## X-WING

Si sur 2 lignes, on retrouve un même candidat sur 2 mêmes colonnes uniquement, dans les colonnes des sommets, cette valeur peut être éliminée de tous les prétendants des colonnes des sommets sauf celles des lignes des sommets .

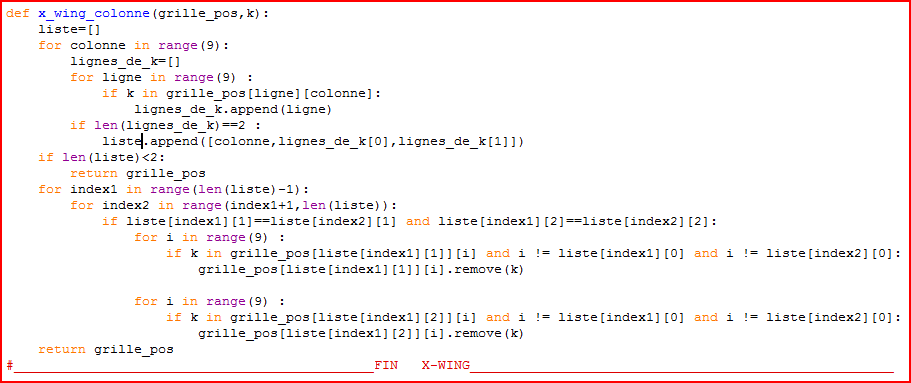


Pour une valeur k donnée, on va enregistrer pour chaque ligne chaque fois que l’on a trouvé 2 fois la valeur : la ligne, la première position, la seconde position.

A l’aide de deux index (index1 et index2) on va tester toutes les lignes 2 à 2 :

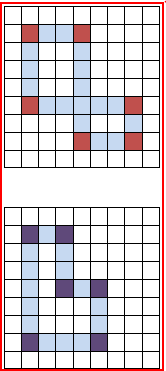
* Si on retrouve deux listes ayant les 2 mêmes colonnes, on a un X-wing, alors :
  + Pour tous les éléments de la colonne de gauche qui ne sont pas sur la première ligne, si k est présent, on le retire de la liste des possibles.
  + Pour tous les éléments de la colonne de droite qui ne sont pas sur la première ligne, si k est présent, on le retire de la liste des possibles.

On retourne la liste des possibles modifiée.



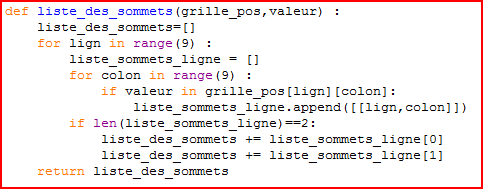
Idem Pour les colonnes.

## Sword-fish

 Cest une extension du X-wing : si sur trois lignes différentes, un candidat n’apparaît que sur trois colonnes(voir exemples ci-contre), alors on supprime ce candidat sur les trois colonnes sauf sur les trois lignes de la grille des possibles.

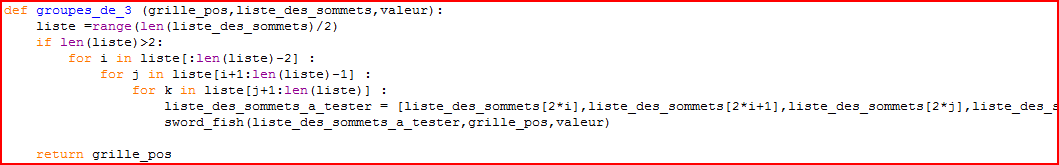
Pour cela j’ai créé 3 fonctions :

* liste\_des\_sommets qui enregistre toutes les paires de sommets
* groupes\_de\_3 qui vérifie que dans la liste des sommets on peut en trouver 3 alignés
* sword\_fish qui va supprimer sur les colonnes la valeur aux bons endroits.



Pour chacune des lignes, on ajoute à la liste\_sommets\_ligne la ligne et la colonne où se trouve la valeur.

Si dans liste\_sommets\_ligne,il y a 2 éléments, on rajoute ces 2éléments à liste\_des\_sommets

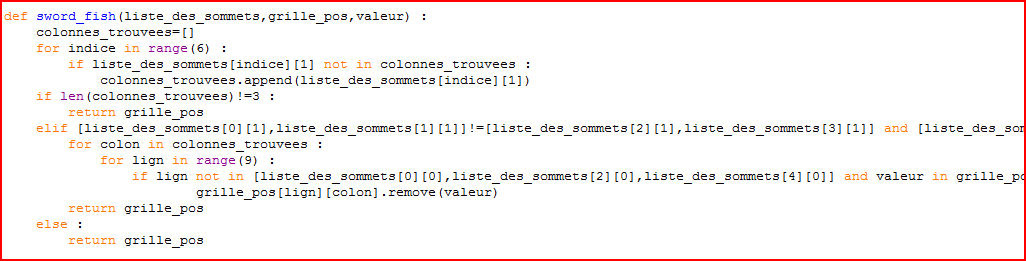


On passe en paramètre : une grille des possibles, la liste des sommets trouvés avec la fonction précédente, la valeur recherchée. La fonction retourne la grille des possibles modifiée.

La liste\_des\_sommets est rangée sous la forme [ligne1,colonne1,ligne2,colonne2….]

On commence par créer une liste[0,1,2…] comportant autant de valeurs qu’il y a de sommets.

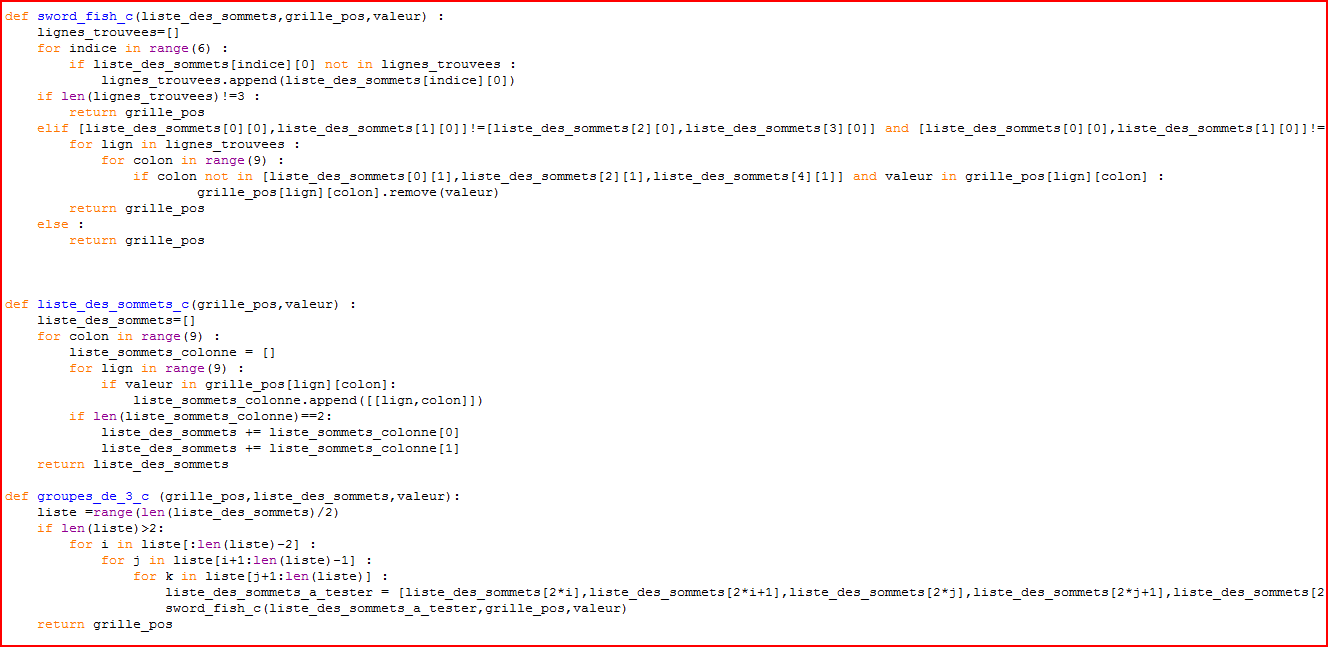
On va créer ensuite toutes les combinaisons possibles de 3 éléments des valeurs de la liste qui serviront d’indice pour tester les combinaisons des différents sommets avec la fonction sword\_fish suivante.



On fait la liste des colonnes\_trouvées en rajoutant pour tous les sommets la colonne où il se trouve à condition que cette colonne n’ai pas déjà été enregistrée.

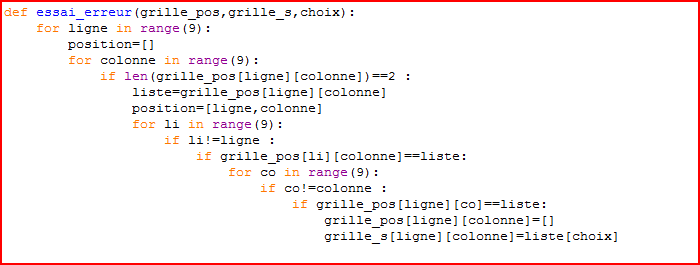
Si le nombre de colonnes\_trouvées n’est pas égal à 3 on ne peut pas faire de sword-fish et la grille\_des\_possibles n’est pas modifiée.

Sinon on vérifie que l’on a pas 2 listes de sommets identiques et là, on est sûr d’être dans le cas du sword-fish et l’on élimine de la grille des possibles ce candidat sur les trois colonnes sauf sur les trois lignes.



Les mêmes fonctions pour un sword-fish sur les colonnes.

# Cas où un choix s’impose



Lorsque qu’aucune des méthodes précédentes n’arrive à se sortir d’une impasse, il arrive que l’on soit obligé de faire une hypothèse pour une case. C’est le cas si sur une ligne et une colonne on a les 2 mêmes prétendants en 2 cases différentes.

On passe pour paramètres : la grille des possibles, la grille de sudoku traitée et le choix (0 ou 1) que l’on va faire quand à l’hypothèse à prendre.

On recherche dans tout le tableau de la grille\_des\_possibles un couple de candidats.

On cherche si ce couple est présent sur la même ligne et sur la même colonne. Si c’est le cas on modifie la grille\_de\_sudoku et la grille\_des\_possibles avec le choix 0 ou 1 et l’on renvoie ces nouvelles grilles comme hypothèse de départ. Bien entendu si le premier choix ne mène pas à la solution finale, il faudra tester le deuxième cas.



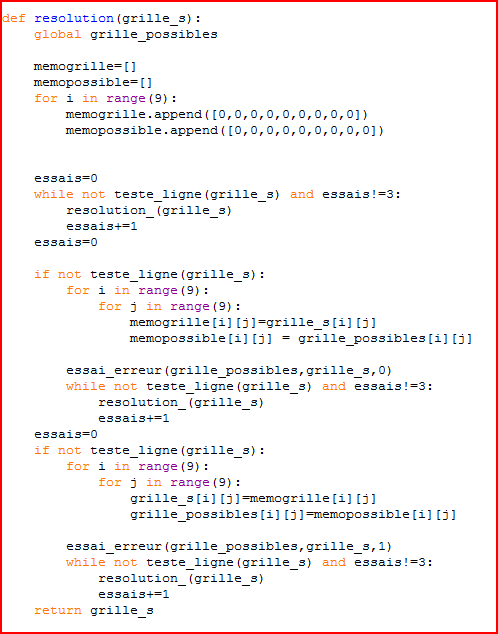
Une deuxième variante de recherche en cas de blocage avec recherche de 2 cases contenant les mêmes prétendants sur une même ligne, une même grande colonne ou un carré de 9.

# La résolution



On fait subir aux grilles toutes les fonctions sauf la fonction essai\_erreur, les unes après les autres.

Les deux premières bouclant jusqu’à ce qu’elles ne trouvent plus de solution (ce sont des fonctions simples qui remplissent beaucoup de cases sans consommer trop de puissance de calcul).



On crée 2 tableaux memogrille et memopossible au cas où on ait besoin de faire la fonction essai\_erreur.

On boucle sur la fonction de résolution précédente jusqu’à ce que le tableau soit rempli ou un nombre d’essais donné, pour éviter de rester coincé dans la boucle .

Si la grille n’est pas résolue, on mémorise les 2 grilles (sudoku et possibles) on teste avec la fonction essai\_erreur et le choix 0 si l’on peut résoudre.

Si l’on a pas résolu on teste avec la fonction essai\_erreur et le choix 1 si l’on peut résoudre.

# 

Deuxième version de la fonction résolution :

Après avoir mémorisé l’état de la grille, on teste une des valeurs puis l’autre avec la première version de la fonction essai\_erreur. Si on n’a pas réussi, on revient à la grille mémorisée et l’on teste la deuxième version essai\_erreur2.

# Epilogue

On doit pouvoir trouver des grilles qui ne peuvent pas être résolues.

En particulier, je pense que la méthode sword\_fish peut être étendue à des carrés imbriqués plus complexes.

Je n’ai pas non plus implémenté la théorie des chaines.

Et la fonction essai\_erreur pourrait être étendue à des choix plus multiples.

Mais ces fonctions arrivent à résoudre les sudokus les plus difficiles que j’ai pu trouver (diaboliques) et même ceux à 17 cases remplies.

# Complexité