Partie n°5 : Modules et classes

Les fractions permettent d'écrire les nombres rationnels à partir de deux nombres entiers, le numérateur et le dénominateur, le second devant être non nul. Parmi toutes les fractions égales, il existe une unique fraction plus simple que toutes les autres : la *fraction irréductible*. Les nombres rationnels ont une autre propriété : tous ont une écriture décimale qui est périodique à partir d'un certain rang. Pour $\frac{2}{3}=0{,}66666\ldots$, c'est le chiffre 6 qui se répète à partir du rang des dixièmes (−1), précédé par le nombre 0 ; pour $\frac{13}{11}=1{,}181818\ldots$ c'est la suite de chiffres 18 qui se répète à partir du rang des dixièmes (−1), précédée par le nombre 1 ; pour $\frac{15}{14}=1{,}071428571428571\ldots$, c'est la suite de chiffres 714285 qui se répète à partir du rang des centièmes (−2), précédée par le nombre 1,0 (le 0 est important ici).

a) Écrire les fonctions nécessaires à l'obtention de l'écriture irréductible d'un nombre rationnel donné par deux nombres *a* et *b* (le numérateur et le dénominateur de la fraction) ainsi que celles donnant la suite de chiffres se répétant, le rang à partir duquel cette suite apparaît dans la partie décimale et la partie précédent cette suite (appelé partie non périodique du quotient). Toutes ces informations doivent pouvoir être obtenues à partir d'une classe que vous appellerez « Frac » (on ne se sert pas encore de la classe « Fraction » qui existe déjà dans Python).

Pour déterminer la fraction irréductible, il faut diviser le numérateur et le dénominateur par le PGCD de ces deux nombres. Notre fonction *pgcd*() ayant déjà été écrite dans la partie 4a (en mode récursif et itératif), nous l'utiliserons en la copiant dans notre fichier *fractions.py* qui contiendra la classe *Frac*, ou bien en l'important grâce à l'instruction *from utilitaire import* * (en supposant que le fichier *utilitaire.py* soit situé dans le même répertoire que *fractions.py* et qu'il contienne la fonction *pgcd*()).

Pour déterminer la suite qui se répète, on peut essayer de transcrire le processus de la division euclidienne en repérant à quelle moment le reste obtenu fait partie des restes qui ont déjà été obtenus. Pour cela, on écrit une petite boucle de recherche sur cette liste, ou bien on utilise la méthode *count*() de la classe *list* qui fait cela (seule la classe *Fonction* de Python nous est interdite ici). La méthode *index*() de la classe *list* est utile aussi car elle donne l'indice de la 1ère occurrence d'un élément donné dans une liste. Remarque : elle pourrait se substituer ici à la méthode *count*() puisqu'au cas où l'élément est dans la liste (*count*() renvoie un nombre supérieur à 0, égal à 1 dans notre situation) elle renvoie l'indice de cet élément mais au cas où elle n'y est pas elle lève un erreur. Si l'on sait traiter les erreurs, on peut se suffire de cette méthode (mais l'interception d'une erreur est un peu trop compliquée pour qu'on développe ce point ici). Nous utilisons donc ces deux méthodes : *count*() pour repérer la pésence d'un reste dans liste des restes, et *index*() pour nous donner l'indice de reste. Pendant tout le processus, nous écrivons les chiffres du développement décimal de *a/b* dans *quotient* qui est une chaîne de caractères. Quand on connait l'*indice* de reste, on peut séparer la chaîne quotient en deux : la longueur de la suite périodique est égale à *len*(restes)–*indice* et donc le début de la chaîne quotient jusqu'à l'indice –(*len*(restes)–*indice*) est la partie non périodique (on donne l'indice en partant de la fin en utilisant un indice négatif), alors que l'autre partie de la chaîne (celle qui commence à cet indice) est la suite périodique.

Maintenant que l'on a réussi à écrire ces deux fonctions, nous pouvons créer notre classe *Frac* qui prend deux entiers comme argument et qui détermine les caractéristiques de la fraction définie par ces deux entiers (fraction irréductible, partie non périodique, suite périodique et rang de cette suite dans l'écriture décimale). Notre classe pourrait faire bien d'autres choses que cela avec les fractions mais nous ne voulons pas nous substituer à la classe *Fraction* de Python. On peut néanmoins compléter notre classe *Frac* en ajoutant une méthode *compare*(Fract) qui renvoie 0 (respectivement 1 et −1) si la fraction passée en argument est égale (respectivement supérieure et inférieure) à la fraction à laquelle on applique cette méthode. Cette méthode additionnelle est juste proposée pour montrer que l'on peut faire autre chose que de déterminer les caractéristiques de l'objet auquel on l'applique.

On a gardé un morceau de programme exécutable à la suite de la classe *Frac* pour continuer à obtenir une sortie semblable à ce qui précède, mais cette partie doit être supprimée si l'on veut conserver le programme *fractions.py* comme module à importer par un autre programme. Notre classe est rudimentaire : elle possède une méthode *__init__*() qui est le « constructeur » de la classe (une fonction obligatoire qui est appelée à chaque fois qu'un objet de type *Frac* est créé). Dans ce constructeur, nous avons lancé le calcul des caractéristiques qui nous intéressent et avons renseignés les cinq attributs que l'on peut interroger/modifier pour un objet de ce type. Dans notre exécution (à droite), on voit que la modification de l'attribut *num* ne modifie pas la valeur de l'attribut *decimales*, ce qui est tout de même assez gênant...

Notre deuxième version de cette classe va apporter quelques améliorations pour pallier à ce défaut en la dotant des fonctions *setNum*() et *setDenom*() qui peuvent modifier le numérateur et le dénominateur de la fraction irréductible, entrainant une réactualisation des attributs *decimales*, *suite* et *rang*. Pour la réactualisation, nous avons choisi la solution d'instancier un nouvel objet *Frac* avec les nouvelles valeurs et d'affecter à notre ancien objet, les attributs de ce nouvel objet. Pour cette nouvelle version de notre classe, nous voulons pouvoir entrer un entier sans préciser que le dénominateur est 1 (cela se fait en donnant au dénominateur une valeur par défaut égale à 1). Nous voulons aussi définir une méthode qui permette d'ajouter deux objets *Frac* : cela passe par la fonction spéciale *__add__*(). Ainsi, on peut obtenir un résultat à la somme de objets *Frac*. Par exemple, si *a=Frac*(1,3) et que *b=Frac*(1,6), alors *c=a+b* est un objet *Frac* qui contient les attributs de ½. Sans cette méthode *__add__*(), l'addition *a+b* aurait déclenché une erreur. On peut faire cela pour toutes les opérations en redéfinissant les méthodes spéciales *__sub__*(), *__mul__*(), *__truediv__*(), *__pow__*(), etc. pour la soustraction, la multiplication, la division décimale, l'élévation à la puissance, etc. On peut aussi redéfinir (on dit aussi sucharger) les méthodes de comparaison, par exemple pour que *a<b* retourne *True* ou *False* selon les cas, au lieu du message d'erreur : TypeError: unorderable types: Frac() < Frac(). Une autre méthode intéressante à surcharger est la méthode *str*() qui transforme un objet en chaine de caractère. On écrit pour cela la fonction *__str__*() qui sera notamment appelée si on utilise la fonction *print*() avec un objet *Frac*.

b) Notre classe *Frac* peut s'étoffer encore un peu : nous voulons pouvoir obtenir l'écriture de notre fraction sous sa forme de fraction continue. Par exemple $\frac{105}{8}=13+\frac{1}{8}$ ou $\frac{51}{7}=7+\frac{1}{3+\frac{1}{2}}$. On notera ces fractions avec une notation linéaire [13,8] et [7,3,2]. Ainsi, toutes les fractions peuvent s'écrire sous la forme d'une liste finie $[a_0, a_1, a_2, \ldots, a_n]$, cette notation ayant un intérêt qui dépasse largement ce niveau anecdotique. Comment obtient-on les coefficients de cette notations : avec l'algorithme d'Euclide (encore), les quotients partiels de chaque étape de la recherche du PGCD nous les fournit : $51=7\times\underline{7}+2$ (donc $a_0=7$) puis $7=2\times\underline{3}+1$ (donc $a_1=3$) et enfin $2=1\times\underline{2}+0$ (donc $a_2=2$). Une fois définie cette fonction *fracContinue*(), nous voudrions pouvoir instancier un objet de la classe *Frac* avec une liste d'entiers. Par exemple, nous voudrions pouvoir faire *a=Frac*([7,3,2]) et que dans *a.num* on trouve 51.
La première partie de ce projet est réalisé en écrivant la nouvelle fonction *fracContinue*() dans le fichier *utilitaire.py*. Ensuite, dans le constructeur de la classe *Frac*, on ajoute une ligne qui appelle cette méthode pour renseigner le nouvel attribut *fcontinue* de notre fraction, à savoir, la liste des coefficients de la décomposition en fraction continue. Il nous reste juste à modifier la fonction *__str__*() pour tenir compte de ce nouvel attribut et vérifier au passage que la fonction *fracContinue*() fait bien ce qu'elle doit faire. Nos deux exemples, sur l'illustration, montrent que c'est bien le cas puisque [7,3,2] est bien la décomposition de $\frac{51}{7}$. Pour la fraction suivante, qui est égale au rapport entre le 13ème et le 12ème nombre de la suite de Fibonacci, la vérification est moins immédiate. On doit calculer le nombre :
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/2)))))))))) !
On a envie de faire confiance à l'algorithme, non ? Sinon, on peut toujours procéder par étape :
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+2/3)))))))))).
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+3/5)))))))).
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+5/8))))))).
A=1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1+8/13)))))).
A=1+1/(1+1/(1+1/(1+1/(1+13/21))))).
A=1+1/(1+1/(1+1/(1+21/34))).
A=1+1/(1+1/(1+34/55)).
A=1+1/(1+55/89).
A=1+89/144).
A=233/144).
Youpi ! Cela fonctionne bien, on dirait. Il ne nous reste plus qu'à modifier le constructeur pour qu'il accepte de définir un objet *Frac* à partir d'autre chose que deux nombres entiers. Nous devons lui faire accepter une liste pour pouvoir définir une fraction à partir de sa décomposition en fraction continue. Comme le typage des variables n'est pas fort en Python, on peut fournir en premier argument un entier ou une liste, il suffit de mettre le test *A is int* qui renvoie *True* si *A* est entier. Nous réécrivons donc le constructeur de notre classe *Frac* pour qu'il accepte une liste comme argument. Ensuite, bien sûr, il faut écrire la fonction *composition*() dans le module *utilitaire* qui réalise la recomposition du nombre rationnel à partir de sa liste de décomposition sous la forme d'un numérateur et d'un dénominateur. Nous avons choisi d'écrire cette

fonction de façon récursive car si f=L[0]+1/f' où L est la liste des coefficients de la décomposition, f' est défini de la même manière f'=L[1]+1/f'' et le dernier terme de la liste est un entier qui permet d'arrêter la récursivité et de remonter dans les calculs. En considérant que f' est une fraction, donnée par son numérateur n et son dénominateur d, nous avons calculé le numérateur et le dénominateur de f en remarquant que f=L[0]+1/(n/d)=(L[0]×n+d)/n.

c) Notre classe *Frac* peut sans doute accepter encore un autre type de déclaration : nous voulons pouvoir définir une fraction par la partie non-périodique de son développement décimal et la suite de chiffres qui se répète (deux chaînes de caractères). Par exemple *a=Frac*("2.1","6") doit être interprété comme le nombre 2,16666... qui est égal à la fraction $\frac{13}{6}$. Pour réaliser cela, on calcule $10^1 a - a = 9a$ (1 car il n'y a qu'un seul chiffre qui se répète) qui vaut autant que 21,6−2,1 (les chiffres d'après sont identiques dans les écritures décimales) soit 19,5. Il ne reste plus qu'à simplifier $\frac{19,5}{9}=\frac{195}{90}=\frac{13\times15}{6\times15}=\frac{13}{6}$. Une fois que la classe *Frac* est au point, la transposer comme une classe héritée de la classe *Fraction* du module *fractions* de Python (*from fractions import Fraction*).

L'algorithme concernant cette transformation a été étudié en seconde (voir TD algorithmique de seconde III-algo3). Les arguments *A* et *B* sont identifiés par leur type (*str*). La fonction *composition2*() de *utilitaire* va se charger du calcul des entiers qui vont définir la fraction (l'autre fonction *composition*() a été renommée *composition1*()). À partir de *N=len*(*B*), le nombre de chiffres qui se répètent, elle enlève *float*(*A*+*B*) à $10^N \times float(A+B+B)$ pour trouver le numérateur *num*. Le dénominateur *denom* est alors 10^N-1. Il ne reste plus qu'à simplifier mais le constructeur de la classe *Frac* sait faire cela.

Pour la fraction qui s'écrit 0,142857142857... on la construit en écrivant *f=Frac*('0.','142857').

On a alors *N*=6 et $10^N \times float(A+B) = 10^6 \times 0.142857142857 = 142857.142857$,

d'où *num*= $142857.142857-0.142857=142857.0$ et *denom*= $10^N-1=999999$. Il faut convertir en entiers les deux nombres (le numérateur n'est pas toujours un entier) en les multipliant chacun par la puissance de dix qui convient (cela dépend de la partie décimale de *num*). On trouve alors $\frac{142857}{999999}$ qui est converti en $\frac{1}{7}$ lors de l'instanciation de l'objet *Frac* correspondant.

Voilà notre classe *Frac* complétée. Elle offre des fonctionnalités intéressantes qui ne sont pas prévues dans la classe *Fraction* de Python et d'autres moins intéressantes comme l'addition qui est déjà implémentée (ainsi que toutes les autres opérations habituelles sur les fractions, sans oublier les opérateurs de comparaison ainsi que d'autres fonctionnalités importantes).

Si nous voulons pouvoir utiliser les fonctionnalités de notre classe *Frac* combinées avec celles de la classe *Fraction* de Python, il faut faire dériver (on dit hériter) les propriétés de *Frac* de la classe mère *Fraction*. Pour commencer, on déclare la classe Frac comme une enfant de Fraction avec, au début, *class Frac*(*Fraction*) : mais ensuite, on ne peut pas continuer à initialiser un objet *Frac* de trois façons différentes, car la superclasse *Fraction* refuse de s'instancier avec une liste par exemple. Nous avons trouvé une sorte de parade qui fait un peu penser à du bricolage, en conservant dans le constructeur la partie concernant un premier argument de type *int* (celui-là ne pose pas de problème). Nous avons ensuite fabriqué une méthode de la classe *Frac* qui prend en argument une liste (*list*), deux chaines (*str*) ou même une fraction (*Fraction*) et qui renvoie un objet de type *Frac* selon les arguments envoyés. L'utilisation de cette méthode est un peu lourde, par exemple en faisant *a=Frac*().*setFrac*([1,2,5]) *a* est un objet *Frac*. On peut s'en convaincre en tapant *print*(*a*) qui renvoie, non pas seulement 16/11 (comme l'instruction *print*() de la classe *Fraction* le prévoit), mais les trois lignes donnant les caractéristiques calculées pour un objet *Frac* : 16/11=1.4545.... La suite de 2 chiffres (45) se répète à partir du rang 1. etc.

Cette nouvelle classe *Frac*, héritant des caractéristiques de la classe mère *Fraction*, peut se permettre maintenant des choses qu'elle ne savait pas faire. Plus besoin de redéfinir l'addition, la soustraction, etc. les comparaisons, etc. tout cela, les objets de la classe *Frac* peuvent le faire. Il y a cependant un bémol : si *a* et *b* sont des objets *Frac*, *a-b* n'en est pas un directement (il faudrait redéfinir la fonction *__sub__*() pour cela) mais peut le devenir en tapant *c=Frac*().*setFrac*(*a-b*) car nous avons prévu dans notre méthode *setFrac*() qu'une fraction soit transformée en objet *Frac*. La comparaison de deux objets *Frac* ne pose cependant pas de problème (pas besoin de redéfinir les méthodes qui permettent normalement cela), ni le parcours d'une liste d'objets de type *Frac*.

d) Pour compléter notre étude des classes Python et enrichir le domaine de nos investigations (il n'y a pas que les nombres!), intéressons nous à un jeu. Il est assez simple de résoudre un sudoku quand on envisage toutes les possibilités, les unes après les autres. L'algorithme de *backtracking* réalise cela : on essaie méthodiquement le remplissage de la grille (sans réfléchir), chiffre après chiffre, et si cela conduit à une

impasse, on revient au point précédent où l'on augmente le chiffre, jusqu'à ce qu'on ai essayé toutes les possibilités, dans ce cas on revient encore plus en arrière, etc. En principe une bonne grille de sudoku ne doit avoir qu'une seule solution, nous la cherchons. L'objectif fixé ici : la grille initiale (une liste de 81 éléments donnée par un fichier) devient une instance de la classe *Grille* qui contient les méthodes nécessaires à la recherche de la solution et à son affichage.
Le programme principal effectue la lecture du fichier *sudoku.txt* qui contient la grille initiale sous une forme lisible (pour faciliter sa mise au point) qui doit être transformée en une liste de 81 entiers, cette liste étant l'argument qui permet d'instancier l'objet *probleme* de la classe *Grille*.
L'instruction *solution=Grille*(*probleme.resoudre*(*probleme.grille*)) lance la recherche (méthode *resoudre*() de la Grille *probleme* avec la grille initiale comme argument, et nomme *solution* la Grille renvoyée par cette méthode. Ainsi, il ne reste plus qu'à utiliser la méthode *affiche*() de cette classe pour obtenir notre grille solution.
Le fonctionnement de la méthode *resoudre*() est récursif : on place un chiffre *n* dans la 1ère case contenant un 0 (d'abord le chiffre 1, les autres ensuite) et on lance la méthode *resoudre*() avec pour argument, la liste contenant ce nouveau chiffre. Lorsque la méthode *resoudre*() active ne réussit pas à placer un nouveau chiffre, elle ne renvoie rien (None) ce qui conduit la méthode *resoudre*() appelante à essayer de placer le chiffre suivant, jusqu'à épuisement des possibilités (ce qui la conduit à renvoyer *None* et produit une rétrogradation) ou jusqu'au succès... Le cœur de cette méthode *résoudre*() est la liste *impossible* des chiffres impossibles pour la case examinée. Cette liste est constituée par l'examen des 20 cases liées à la case examinée (8 cases liées dans le même bloc, plus 6 dans la même ligne et 6 dans la même colonne). En fait, ce sont les 81 cases de la grille qui sont examinées, et la méthode *meme*() est chargée d'identifier si les deux cases examinées sont liées pour au moins une de ces contraintes :
(*i-j*)%9==0 est *True* si *i* et *j* sont dans la même colonne ; *i*//9==*j*//9 est *True* si *i* et *j* sont dans la même ligne ; 3*(*i*//27)+(*i*%9)//3==3*(*j*//27)+(*j*%9)//3 est *True* si *i* et *j* sont dans le même bloc. Ce dernier test est sans doute moins facile à comprendre : *i*%9 représente la colonne (de 0 à 8) donc (*i*%9)//3 représente la colonne de blocs (de 0 à 3), *i*//27 représente la ligne de blocs (0, 1 ou 2) donc 3*(*i*//27) représente la ligne de blocs (0, 3 ou 6), il suffit ensuite d'ajouter ligne et colonne du bloc.

e) Vous noterez que l'algorithme employé ici pour résoudre les sudokus n'est pas du tout intelligent : il essaie bêtement toutes les possibilités jusqu'à trouver une solution. S'il n'y a en a pas, il va produire une erreur (rien n'est prévu pour ce cas) et s'il y en a deux ou plus, il ne nous le dira pas (et c'est pourtant disqualificatoire pour un sudoku). Nous voulons améliorer notre programme et compter les solutions (aller, tout aussi bêtement, jusqu'au bout du processus). Pour prolonger cette étude, nous pouvons essayer de générer une grille de sudoku valide (a une solution unique) et dont le nombre de chiffres de la grille initiale n'est pas trop grand (entre 17 qui est la valeur minimum et 25, une valeur raisonnable arbitraire) .
Pour la première question posée (compter les solutions d'une grille), nous devons modifier la condition d'arrêt de la fonction récursive. Le fait d'avoir complété une grille (il n'y a plus de 0) ne doit pas arrêter la description de l'arbre des possibilités. Nous devons au moins aller jusqu'à une 2ème solution, si elle existe, pour pouvoir éliminer une grille qui n'est pas valide. À cette fin, nous allons enregistrer les solutions trouvées dans une liste (une liste contenant des listes, les solutions) et c'est tout ! La fonction appelante ne recevant aucune valeur de retour (None), le processus d'exploration de l'arbre des possibilités continue jusqu'à la fin. Le programme principal a été légèrement modifié, la fonction *cherche*() contient toutes les instructions pour effectuer le travail de recherche des solutions, alors que le reste (à droite) se charge de la lecture de la grille initiale. Dans la classe *Grille*, il y a la fonction *isValide*() qui examine si une grille est valide. Cette fonction apparaît inutile si on part d'une grille valide, mais comme l'intérêt de ce programme est de tester des grilles nouvelles, il vaut mieux pouvoir les identifier dès le départ (sinon, elle seraient reconnues comme insolubles (sans solution), ce qui n'est pas la même chose). Nous n'avons pas remis les fonctions *meme*() et *affiche*() de la classe *Grille* dans l'illustration, pour gagner de la place.
La fonction *resoudre*() est presque identique à la précédente. Ce qui est changé, c'est la réponse au fait qu'une solution soit trouvée : on l'ajoute ici à la liste. Dans la fonction *cherche*(), cette liste des *solutions* est retournée et sa longueur différencie les grilles valides (une solution) de celles qui ont trop de solutions. La grille inventée, proposée dans le fichier *sudoku7.txt* possède ainsi 1518 solutions.
Pour le prolongement suggéré, la difficulté augmente. Notre programme actuel trouve la ou les solutions d'une grille valide, mais ne permet pas de modifier la grille pour en faire une grille ayant une solution unique. Cela sera notre approche pour rechercher une grille valide : choisir aléatoirement, jusqu'à 25

chiffres, en ajoutant un nouveau chiffre lorsque la grille valide en cours est surabondante. La plupart du temps, la grille trouvée sera invalide (dernier chiffre incompatible), insoluble (pas de solution) ou surabondante (trop de solutions). En cas d'invalidité, nous augmentons le dernier chiffre (il y en a forcément un qui rend la grille valide (sans incompatibilité) avec l'instruction *n*=*n*%9+1. En cas d'insolubilité, nous enlevons un chiffre au hasard et en cas de surabondance nous en ajoutons un. On peut raffiner cette méthode ou la transformer en procédure de *backtracking* (exploration systématique), mais l'objectif dépasse les limites que nous nous sommes fixés pour ce document. La génération d'une grille de sudoku valide n'est d'ailleurs pas une fin en soi, car il faut encore se confronter au challenge d'évaluer la difficulté de la grille. On ne peut, en effet, remettre entre les mains d'un joueur, une grille non évaluée. Le nombre de chiffres présents sur la grille n'est pas une bonne mesure de la difficulté. Certaines grilles à *n* chiffres sont faciles et d'autres extrêmement difficiles (comme ce sudoku qui a été résolu en 20 secondes par notre programme sur la 1ère illustration). L'évaluation de la difficulté d'un sudoku passe nécessairement par les techniques intelligentes développées par les joueur.

La 1ère des techniques consiste à compléter la grille lorsqu'il n'y a aucun autre choix pour une case. Cette technique parfois suffit, à elle seule, pour compléter la grille entière. Le problème est alors classé « très facile ». La plupart du temps, il est nécessaire de faire la liste des choix possibles pour toutes les cases de la grille, et l'on peut alors discerner les cases où il n'y a qu'un seul choix. Une grille qui serait réalisable avec ces deux seules techniques peut être qualifiée de « facile ». Les niveaux de difficulté augmentent encore lorsque ces méthodes élémentaires ne suffisent plus, et l'on peut discerner, dans le vaste ensemble des sudokus valides, des problèmes de difficulté très élevée, qui résistent aux différentes techniques employées. Les sudokus présentés ci-dessus ont tous les deux 23 chiffres « révélés » et, pourtant, ils sont de difficultés très différentes : celui de gauche ne nécessitant que les deux techniques évoquées plus haut (unicité des chiffres « candidats » pour une case ; la méthode *evalue*() de la classe *Grille* ci-dessous nous montre une implémentation de ces deux techniques de base) est facile alors que celui de droite, inventé en 2006 par le finlandais Arto Inkala et surnommé « Al Escargot », a été réputé le « sudoku le plus difficile au monde » jusqu'à ce qu'on lui conteste cette suprématie. Cette discussion d'expert ne nous concerne pas, mais il va sans dire que pour mesurer la difficulté du sudoku le plus difficile, il faut connaître et avoir programmé toutes les techniques logiques applicables, ce qui paraît irréalisable.

Telle qu'elle est présentée, cette méthode *evalue*() teste d'abord les techniques *verifLigne*(), *verifColonne*(), et *verifBloc*(), qui, si elles suffisent à résoudre la grille (*if self.grille.count*(0)!=0), attribuent la difficulté 0 au sudoku. Si le sudoku résiste à ce traitement, on lui applique la méthode *verifCase*() qui examine tous les chiffres « candidats » possibles pour une case et remplit la case qui n'a qu'un seul candidat. Pour l'application de ces deux techniques, le programme a besoin d'une 2ème liste, en plus de celle des chiffres, qui contient les différentes possibilités pour chaque case. Cette liste, nommée *possible* dans ce programme, contient les listes des 9 valeurs booléennes (*True* ou *False*) correspondant à la possibilité d'un chiffre dans une case. Par exemple, pour le sudoku facile à 23 chiffres ci-dessus, cette liste commence par :

[[*True, False, False, False, True, True, False, False, False*], [*True, False, False, False, False, True, True, False, False*], etc.]

Le 1er *True* signifie que le chiffre 1 peut se mettre dans la case 0 (en haut à gauche). Dans cette case, on ne peut mettre que les chiffres 1 et 5, alors que dans la case 1 suivante, on peut mettre le 1 ou le 7... La liste *possible* est initialisée au départ à *True* pour tous les chiffres dans toutes les cases vides où ils peuvent se loger et à *False* ailleurs, puis, elle est mise à jour après chaque placement d'un chiffre par la méthode *placeElement*().