Maze

If a maze has two entryways, it is called a *perfect maze* if there is only one possible rout to get from one entryway to the other. An easy way to find the unique route through the perfect maze, consists of filling up the blind halls until there are no more blind halls. The remaining halls form the route you are looking for.

Assignment

Your assignment consists of finding the unique route that is used to get from one entryway to another in a perfect maze. To do so, you make a class PerfectMaze, that supports the following methods:

1. The initialize method __init__(s) gets a string that represents the maze as a parameter. This string contains *only* the following characters: hashes ('#'), spaces (' ') and newline characters ('\n'). These last characters indicate the end of a line. You may assume that only the summed up characters occur in the given string, and that all lines contain the same amount of characters. The method saves the maze as a list of lists (a two-dimensional list) with the name data. The initializing method must make this list as an attribute of the object.
2. The method __str__() gets, as usual, no arguments. When it is called, it prints a graphical display of the maze. Use the template below as a template. The maze itself comes from the attribute data.
3. Two methods getCell(t) and setCell(t,c). The first method gets a *tuple* (x,y) as a parameter and print the value of the maze on the box with position (x,y). These co-ordinates correspond with the row with number x, counted from the top, and the column with number y, counted from the left. The first row and column have number 0. The function setCell works analogous but it asks two parameters: the first is a tuple $t = (x,y)$ that corresponds with a set of co-ordinates of the maze. The second parameter is the value that must be appointed to the box. In practice, this is always a string consisting of a single character: a hash or a space.
4. A method numberWalls(t), to which one parameter must be given: a tuple $t = (x,y)$ that represents the co-ordinates of a box. Here, you may assume that this box is not situated on the border of the maze. The method prints the number of adjoining walls of this box.
5. A method firstThreeWalls() to which no parameters must be given and that prints a tuple t. This tuple represents the co-ordinates of the first box in the maze that is not a wall itself, and has exactly three walls as neighbour. To determine the *first* box with that property, you *first* go through the rows of the maze from top to bottom and then the columns of the maze, from left to right. If no such box is found, the method must print the value None.
6. A method findRoute() that finds the way in the perfect maze by repeatedly filling up all boxes with three neighbouring walls with a wall, so that there are no more boxes left.

As a start for your code you may depart from the template below:

```
class PerfectMaze:

    "Representation and manipulation of a perfect maze."

    def __init__(self, s=''):
    def __str__(self):
```

```
    def getCell(self, t):
    def setCell(self, t, i):
    def numberWalls(self, t):
    def firstThreeWalls(self):
    def findRoute(self):
```

Example

```
>>> s='''########### #
... #     #   # #
... # ### # # # #
... #   #   # # #
... ### ##### # #
... #   # ###   #
... # ### # #####
... # #   # #   #
... # # # # ### #
... # # # #     #
... # # # # # ###
... #   #   #   #
... ########### #'''
>>> d = PerfectMaze(s)
>>> print(d)
13 x 13 maze:
########### #
#     #   # #
# ### # # # #
#   #   # # #
### ##### # #
#   # ###   #
# ### # #####
# #   # #   #
# # # # ### #
# # # #     #
# # # # # ###
#   #   #   #
########### #
>>> d.firstThreeWalls()
(5, 5)
>>> d.getCell((5, 5))
' '
>>> d.setCell((5, 5), '#')
>>> d.getCell((5, 5))
'#'
>>> d.firstThreeWalls()
(6, 5)
>>> d.findRoute()
>>> print(d)
13 x 13 maze:
########### #
#     #   # #
# ### # # # #
#   #   # # #
### ##### # #
#   #####   #
# ###########
# #   #######
# # # #######
# # # #   ###
```

```
# # # # # ###
#   #   #   #
########## #
```

Een doolhof met twee ingangen wordt een *perfect doolhof* genoemd indien er slechts één mogelijk route bestaat om vanuit de ene ingang de ander ingang te bereiken. Een makkelijke manier om de unieke route doorheen een perfect doolhof te vinden, bestaat erin om alle doodlopende gangen op te vullen todat er geen doodlopende gangen meer zijn. De overblijvende gangen vormen dan de gezochte route.

Opgave

Je opdracht voor deze opgave bestaat erin de unieke route weer te geven waarmee vanuit de ene ingang van een gegeven perfect doolhof de andere ingang kan bereikt worden. Hiervoor maak je een klasse `PerfectDoolhof` aan, dat ondersteuning biedt aan onderstaande methoden:

1. De initialisatiemethode `__init__(s)` krijgt als parameter een string mee die de doolhof voorstelt. Deze string bevat *enkel* de volgende drie karakters: hekjes ('#'), spaties (' ') en newline karakters ('\n'). Deze laatste geven het einde van een regel aan. Je mag ervan uitgaan dat enkel de opgesomde karakters voorkomen in de meegegeven string, en dat bovendien alle regels evenveel karakters bevatten. De methode bewaart de doolhof als een lijst van lijsten (een tweedimensionale lijst dus) met de naam `data`. De intialisatiemethode moet deze lijst dus aanmaken als attribuut van het object.
2. De methode `__str__()` krijgt zoals gebruikelijk geen argumenten mee. Wanneer ze wordt aangeroepen, geeft ze een grafische weergave van de doolhof terug. Baseer je daarvoor op de template uit het voorbeeld hieronder. De doolhof zelf wordt gehaald uit het attribuut `data`.
3. Twee methoden `getCel(t)` en `setCel(t,c)`. De eerste methode krijgt als parameter een *tupel* (x,y) mee en geeft de waarde van de doolhof weer op het vakje met positie (x,y). Deze coördinaten komen dus overeen met de rij met rangnummer x, vanaf boven te tellen, en de kolom met rangnummer y , vanaf links te tellen. De eerste rij en kolom hebben telkens rangnummer 0. De functie `setCel` werkt analoog maar deze vraagt twee parameters: de eerste is een tupel $t = (x,y)$ dat overeenkomt met een stel coördinaten van het doolhof. De tweede parameter is de waarde die aan dit vakje moet toegekend worden. Dit is in praktijk steeds een string bestaande uit één enkel karakter: een hekje of een spatie.
4. Een methode `aantalMuren(t)`, waaraan één parameter moet meegegeven worden: een tupel $t = (x,y)$ dat de coördinaten van een vakje voorstelt. Hierbij mag je veronderstellen dat dit vakje zich niet op de rand van de doolhof bevindt. De methode geeft het aantal aangrenzende muren van dit vakje terug.
5. Een methode `eersteDrieMuren()` waaraan geen parameters moeten meegegeven worden en die een tupel `$t$` teruggeeft. Dit tupel stelt de coördinaten voor van het het eerste vakje in de doolhof dat zelf geen muur is, en precies drie muren als buur heeft. Om het *eerste* vakje met die eigenschap te bepalen, doorloop je *eerst* de rijen van de doolhof van boven naar onder, en vervolgens de kolommen van de doolhof, van links naar rechts. Als er geen zo'n vakje kan gevonden worden, dan moet de methode de waarde `None` teruggeven.
6. Een methode `vindRoute()` die de weg in de perfecte doolhof vindt door herhaaldelijk alle vakjes met precies drie aangrenzende muren op te vullen met een muur, tot er geen zo'n vakjes meer over zijn.

Als aanzet voor je programmacode kan je vertrekken van onderstaand sjabloon:

```
class PerfectDoolhof:

    "Voorstelling en manipulatie van een perfect doolhof."

    def __init__(self, s=''):
    def __str__(self):
    def getCel(self, t):
    def setCel(self, t, i):
    def aantalMuren(self, t):
    def eersteDrieMuren(self):
    def vindRoute(self):
```

Voorbeeld

```
>>> s='''########### #
... #     #   # #
... # ### # # # #
... #   #   # # #
... ### ##### # #
... #   # ###   #
... # ### # #####
... # #   # #   #
... # # # # ### #
... # # # #     #
... # # # # # ###
... #   #   #   #
... ########### #'''
>>> d = PerfecteDoolhof(s)
>>> print(d)
13 x 13 doolhof:
########### #
#     #   # #
# ### # # # #
#   #   # # #
### ##### # #
#   # ###   #
# ### # #####
# #   # #   #
# # # # ### #
# # # #     #
# # # # # ###
#   #   #   #
########### #
>>> d.eersteDrieMuren()
(5, 5)
>>> d.getCel((5, 5))
' '
>>> d.setCel((5, 5), '#')
>>> d.getCel((5, 5))
'#'
>>> d.eersteDrieMuren()
(6, 5)
>>> d.vindRoute()
>>> print(d)
13 x 13 doolhof:
########### #
#     #   # #
# ### # # # #
#   #   # # #
```