```
import math
import random
import numpy as np

'''
Zero is the place holder for the empty square.
the matrix is divided equally so,
[1,2,3,4,5,6,7,8, 0] =>

    __________
    |1 | 2| 3|
    |4 | 5| 6|
    |7 | 8| 0|
    ~~~~~~~~~~
'''


class Puzzle:

    def __init__(self, size=3, shuffle=True, manhat=False, ecd=False):
        self.size = size
        self.puzzle = []  # [1, 2, 3, 4, 5, 6, 7, 8, 0]
        self.createPuz(size)
        self._index = 8
        self._dist = 0
        self._solved = False
        self._globalCost = 0
        self.parent_node = None
        self._manhat=manhat
        self._ecd = ecd

        if(shuffle):
            self.scramble()
            self.distCheck()

    def createPuz(self, size):
        for x in range(1, size*size):
            self.puzzle.append(x)
        self.puzzle.append(0)

    def __str__(self):
        return "_____________\n| {0} | {1} | {2} |\n" \
            "| {3} | {4} | {5} |\n| {6} | {7} | {8} |\n~~~~~~~~~~~~~".format(
                *self.puzzle)

    def findIndex(self):
        i = 0
        for x in range(9):
            if self.puzzle[x] == 0:
                i = x
                self._index = i
            #print(self.puzzle[x], "{\}", end="")

        return i

    def scramble(self):
        random.shuffle(self.puzzle)
        self.findIndex()

    def distCheck(self):
        dist = 0
        if self._manhat:
            g1 = np.asarray(self.puzzle).reshape(3, 3)
            g2 = np.asarray([1, 2, 3, 4, 5, 6, 7, 8, 0]).reshape(3, 3)

            for i in range(8):
                a, b = np.where(g1 == i+1)
                x, y = np.where(g2 == i+1)
```

```
                dist += abs((a-x)[0])+abs((b-y)[0])

        if self._ecd:
            g1 = np.asarray(self.puzzle).reshape(3, 3)
            g2 = np.asarray([1, 2, 3, 4, 5, 6, 7, 8, 0]).reshape(3, 3)

            for i in range(8):
                a, b = np.where(g1 == i+1)
                x, y = np.where(g2 == i+1)
                dist +=  math.sqrt((abs((a-x)[0]) ** 2) + (abs((b-y)[0]) ** 2))
        else:
            for i, j in zip(self.puzzle, range(9)):
                if i != (j + 1) and (i != 0 ):
                    dist += 1

        self._dist = dist
        return dist

    def up(self):
        if(0 in self.puzzle[((self.size ** 2)-self.size):]):
            #print("in bottom: invalid")
            return False
        else:
            self.puzzle[self._index], self.puzzle[self._index +
                                                  3] = self.puzzle[self._index + 3],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            #print(self._index,"......")
            return True

    def down(self):
        if(0 in self.puzzle[0:self.size]):
            #print("in top: invalid")
            return False
        else:
            self.puzzle[self._index], self.puzzle[self._index -
                                                  3] = self.puzzle[self._index - 3],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            return True

    def right(self):
        if (self._index != 0 and self._index != 3 and self._index != 6):
            #swap the index to the left
            self.puzzle[self._index], self.puzzle[self._index -
                                                  1] = self.puzzle[self._index - 1],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            return True
        else:
            #print("Invalid Move")
            return False

    def left(self):
        if (self._index != 2 and self._index != 5 and self._index != 8):
            #swap the index to the right
            self.puzzle[self._index], self.puzzle[self._index +
                                                  1] = self.puzzle[self._index + 1],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            return True
        else:
            #print("Invalid Move")
```

```
            return False

    def __iter__(self):
        for v in self.puzzle:
            yield v

    def __lt__(self, obj):
        return (self._dist + self._globalCost) < (obj._dist + obj._globalCost)


'''
Testing
x = Puzzle()


print(x.findIndex())

print(x)
x.down()
print(x)

'''
```