```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from random import shuffle
from PIL import Image, ImageDraw
import random
import unittest
from turtle import *

"""A series of functions generate mazes and draw them"""

def coordinates_cell(cell, nx):
    """
    Takes as param the number of a cell and
    the size of a grid (width=nx and height=ny) and
    returns the coordinates of the cell
    """
    coord_cell = [cell % nx, cell // nx]
    return coord_cell


def sequence(n):
    """
    Takes a non-negative integernotas a param,returns array of len
    not containing in order the integer values from 0 to n-1 inclusively
    """
    return list(range(n))


def contains(tab, x):
    """
    Takes parameter array of numbers (tab) and a number x and
    returns a boolean
    indicating whether x is in the table
    """
    return x in tab


def add_element(tab, x):
    """
    Takes  param an array of nums(tab) and a num x and
    returns new array with
    the same content as tab except that x is added at
    end if it is not already contained in tab
    """
    if x not in tab:
        tab.append(x)
    return tab


def remove_element(tab, x):
    """
    Takes as param an array of num(tab) and a num x
    and returns new array with same content as tab except that
    x is removed from the table
    """
    if x in tab:
        tab.remove(x)
    return tab


def pledge_left(adjustment, counter):
    """
    Takes as param the adjustment value and the counter value and
    returns the new counter value
    """
    forward(adjustment)
    left(90)
    counter += 1
    return counter


def pledge_right(next_step, counter):
    """
    Takes as param the next_step value and the counter value and
    returns the new counter value
    """
    right(90)
    forward(next_step)
    counter -= 1
    return counter


def neighbors(x, y, nx, ny):
    """
    takes the coordinate (x,y) of a cell and
    the size of a grid (width=nx and height=ny) and
    returns an array containing the numbers of neighboring cells
    """
    num = nx * y + x
    tab = []

    if y > 0:
        tab.append(num - nx)
    if x > 0:
        tab.append(num - 1)
    if y < ny - 1:
        tab.append(num + nx)
    if x < nx - 1:
        tab.append(num + 1)

    return tab


def wall_number(x, y, nx, side):
    """
    takes the coordinate (x,y) of a cell and
    the size of a grid (width=nx and height=ny) and
    the side of the cell and
    returns the number of the wall
    """
    num = 0

    if side == 'N':
        num = x + y * nx
    elif side == 'E':
        num = 1 + x + y * (nx + 1)
    elif side == 'S':
        num = x + (y + 1) * nx
    elif side == 'O':
        num = x + y * (nx + 1)

    return num


def generate_labyrinth(nx, ny):
    """
    takes the size of a grid (width=nx and height=ny) and
    returns the walls of the labyrinth
    """
    nb_cell = nx * ny
    walls_h = sequence(nx * (ny + 1) - 1)
    walls_h = remove_element(walls_h, 0)
    walls_v = sequence((nx + 1) * ny)

    cell_v = []
    v_in_cave = []
    neighbor = 0

    coord_c = coordinates_cell(random_cell, nx)
    coord_cx, coord_cy = coord_c[0], coord_c[1]

    cave = [random_cell]
    front = neighbors(coord_cx, coord_cy, nx, ny)

    for nb_elements_cave in range(1, nb_cell):
        random_cell = front[random.randint(0, len(front) - 1)]

        front = remove_element(front, random_cell)
        cave = add_element(cave, random_cell)

        coord_c = coordinates_cell(random_cell, nx)
        coord_cx, coord_cy = coord_c[0], coord_c[1]

        cell_v = neighbors(coord_cx, coord_cy, nx, ny)
        v_in_cave = [neighbor for neighbor in cell_v if neighbor in cave]

        for neighbor in cell_v:
            if neighbor in cave:
                v_in_cave.append(neighbor)
            else:
                front = add_element(front, neighbor)

        coord_v = coordinates_cell(v_in_cave[random.randint(0, len(v_in_cave) -
1)], nx)
        coord_vx, coord_vy = coord_v[0], coord_v[1]

        if coord_cx == coord_vx:
            if coord_cy == coord_vy + 1:
                walls_h = remove_element(walls_h, random_cell)
            else:
                walls_h = remove_element(walls_h, random_cell + nx)
        else:
            if coord_cx == coord_vx - 1:
                walls_v = remove_element(walls_v, coord_vx + coord_vy * (nx +
1))
            else:
                walls_v = remove_element(walls_v, coord_vx + 1 + coord_vy * (nx
+ 1))

    walls = [walls_h, walls_v]
    return walls


def draw_labyrinth(walls_h, walls_v, nx, ny, step):
    """
    takes the walls of the labyrinth and
    the size of a grid (width=nx and height=ny) and
    the step of the cell and
    draws the labyrinth
    """
    x0 = -nx * step / 2
    y0 = ny * step / 2

    for i in range(len(walls_v)):
        wall_indv = walls_v[i]
        penup()
        goto(x0 + wall_indv % (nx + 1) * step, y0 + (wall_indv // (nx + 1)) * -
step)
        pendown()
        backward(step)

    right(90)

    for i in range(len(walls_h)):
        wall_indv = walls_h[i]
        penup()
        goto(x0 + (wall_indv % nx) * step, y0 + (wall_indv // nx) * (-step))
        pendown()
        forward(step)

    right(90)


def labyrinth(nx, ny, step):
    """
    takes the size of a grid (width=nx and height=ny) and
    the step of the cell and
    draws the labyrinth
    """
    walls = generate_labyrinth(nx, ny)
    walls_h, walls_v = walls[0], walls[1]
    draw_labyrinth(walls_h, walls_v, nx, ny, step)


def labyrinth_solution(nx, ny, step):
    """
    takes the size of a grid (width=nx and height=ny) and
    the step of the cell and
    draws the labyrinth with the solution
    """
    walls = generate_labyrinth(nx, ny)
    walls_h, walls_v = walls[0], walls[1]
    draw_labyrinth(walls_h, walls_v, nx, ny, step)
    walls_h.append(0)
    x0 = -nx * step / 2
    y0 = ny * step / 2
    counter = 0
    adjustment = 0.6 * step
    next_step = 0.4 * step
    orientation = 0
    pencolor(1, 0, 0)
    penup()
    goto(x0 + 0.2 * step, y0 + 0.2 * step)
    pendown()
    forward(next_step)
    x, y = 0, 0

    while y < ny:
        if counter == 0:
            if wall_number(x, y, nx, 'S') in walls_h:
                counter = pledge_left(adjustment, counter)
            else:
                forward(step)
                y += 1
        else:
            orientation = counter % 4

            if orientation == 1:
                if wall_number(x, y, nx, 'S') not in walls_h:
                    counter = pledge_right(next_step, counter)
                    y += 1
                    continue
            elif orientation == 2:
                if wall_number(x, y, nx, 'E') not in walls_v:
                    counter = pledge_right(next_step, counter)
                    x += 1
                    continue
            elif orientation == 3:
                if wall_number(x, y, nx, 'N') not in walls_h:
                    counter = pledge_right(next_step, counter)
                    y -= 1
                    continue
            elif orientation == 0:
                if wall_number(x, y, nx, 'O') not in walls_v:
                    counter = pledge_right(next_step, counter)
                    x -= 1
                    continue

            counter = pledge_left(adjustment, counter)

    return


class TestMazeFunctions(unittest.TestCase):
    """tests for maze.py"""

    def test_sequence(self):
        """Test case for sequence function"""
        result = sequence(5)
        self.assertEqual(result, [0, 1, 2, 3, 4])

    def test_contains(self):
        """Test case for contains function - positive case"""
        result_positive = contains([1, 2, 3], 2)
        self.assertTrue(result_positive)

        """Test case for contains function - negative case"""
        result_negative = contains([1, 2, 3], 4)
        self.assertFalse(result_negative)

    def test_add(self):
        """Test case for add function"""
        result = add_element([9, 2, 5], 2)
        self.assertEqual(result, [9, 2, 5])

    def test_remove(self):
        """Test case for remove function"""
        result = remove_element([1, 2, 3], 2)
        self.assertEqual(result, [1, 3])

    def test_neighbors(self):
        """Test case for neighbors function"""
        result = neighbors(7, 2, 8, 4)
        self.assertEqual(result, [15, 22, 31])



if __name__ == "__main__":
    unittest.main()
```