Programming for Science
Workshop 7

The second CA is due this week, so this week please make sure that you have caught up with all the other workshops. When you've done that, work on this more extended question, which was part of a CA last year. Since it's a long question (needs more code than any of the questions this year) and bits of it may be helpful for this CA, a possible answer is given on the following pages. Please make sure you attempt the question before reading the answer!

1. A 30×30 grid of squares contains 900 fleas, initially one flea per square. When a bell is rung, each flea jumps to an adjacent square at random (usually there are 4 possibilities except at the edges or at the corners).

What is the average number of unoccupied squares after 50 rings of the bell?

To answer this question you should write a function (perhaps more than one) that simulates the jumping fleas to find the number of unoccupied squares after a given number of steps. Your function should take as arguments the size of the square grid and the number of steps to simulate, and it should return the number of unoccupied squares at the end. Then write a program or another function to call the single-run simulation many times to work out the average number of unoccupied squares. You should also write a function that displays the grid of squares with each square giving labelled with the number of fleas; for example:

```
2 3 0 1 1 0 3
1 0 1 2 0 1 1
0 1 1 1 2 2 0
1 0 0 1 2 0 2
2 1 1 0 0 1 1
0 0 1 1 3 0 1
1 0 3 1 2 0 1
```

for a 7×7 grid. Show the results of a 50 bell run. You may find this function helpful in debugging (de-fleaing?) your program.

Hints:

- Think carefully about how to represent the fleas and the grid. One possibility is to represent the grid by a list of lists (one list for each row of the grid), but another possibility is to just to keep a list of the flea locations.
- You can write more than just the single-run simulation function. Think about how the program should be decomposed into smaller functional units. Where possible write test code and use assertions to ensure these are correct.
- Although the question is phrased in terms of a 30×30 grid, it's best to write it for a grid of arbitrary size. Likewise, although it's phrased in terms of 50 rings of the bell, write the functions for any number of rings.

Make sure you've thought seriously about the question before reading these pages!

Here are a couple of ways of solving the fleas problem.

In the following code I've chosen to represent each flea as a two-element list containing the flea's location; the population of fleas is thus a 900-long list of lists, called `fleas`. The code is organised as lots of little functions, each one doing one job.

In order to make a jump from a location `(i,j)` the function `jump` adds or subtracts 1 to either `i` or `j` at random. Sometimes (when the flea is on the edge or at a corner) this jump will take the flea off the edge of the grid. In that case we (`legal_jump`) just try again until the flea jumps to another square on the board. If you think about it you can see that this means that the flea jumps to one of the neighbouring legal squares with equal probability.

The function `update_locations` uses `legal_jump` to update the positions of all the fleas in turn, corresponding to one ring of the bell.

The `single_run` function uses `update_locations` to ring the bell as many times as required.

The fleas are represented by a list of locations, so in order to print out the grid or to count the number of unoccupied squares on the grid, a grid must be populated with the fleas in their correct places. This is done by `grid_from_locations`, which just makes an empty grid and then adds one to the count of fleas in a particular square for each flea in the list of fleas. This allows the grid to easily be printed (`print_grid`) and the empty squares counted by `empty_squares`.

Finally, `test_single_run` prints the results of a single run and checks that there are still $(\texttt{size})^2$ fleas at the end of the run. The average number of unoccupied squares is calculated by `average_occupancy`, which just runs `single_run` for the required number of trials.

The code ends up being quite long, partly because it is broken up into many little functions (and each has a doc-string) which adds to the line count. Nonetheless, writing the code as lots of small functions is worthwhile because it makes how the code is constructed clear.

```
from random import random

def islegal(i, j, size):
    """
    Return True if the location (i,j) is on a grid of size by size.
    """
    return (0 <= i < size) and (0 <= j < size)

def randsign():
    """
    Return +1 or -1 with equal probability.
    """
    # This code uses the tenary operator, but the same could be achieved with
    # if random() > 0.5:
    #    return 1
    # else:
    #    return -1
    return 1 if random() > 0.5 else -1
```

```
def jump(i, j):
    """
    Given a location (i,j) simulate the jump of a flea by one square to
    the north, south, east or west at random. Returns the location of the
    flea after it has jumped.

    Note that this may move the flea off the grid.
    """
    if random() > 0.5:
        # Adjust i
        i += randsign()
    else:
        # Adjust j
        j += randsign()
    return i, j

def legal_jump(i, j, size):
    """
    Given a location (i, j) simulate the jump of a flea by one square to
    the north, south, east or west at random, making sure that the returned
    location is still on the grid (of size by size). This is achieved by
    trying jumps until one that is on the grid is found. Returns the new
    location.
    """
    while True:
        newi, newj = jump(i, j)
        if islegal(newi, newj, size):
            return newi, newj

def update_positions(fleas, size):
    """
    Update the positions of the fleas whose locations are given in the list
    fleas. Each element of this list should be a two-element list giving
    the coordinates of a flea.  The fleas list is modified in-place.
    """
    for flea in fleas:
        flea[0], flea[1] = legal_jump(flea[0], flea[1], size)

def single_run(Nrings=50, size=30):
    """
    Simulate the Nrings jumps of the fleas on a size by size grid.

    Returns a list of the flea's locations at the end of the run.
    """
    # Initialise a flea in each square.
    # Could be done with a list comprehension
```

```
    fleas = []
    for i in range(size):
        for j in range(size):
            fleas.append([i, j])

    for n in range(Nrings):
        update_positions(fleas, size)
        assert len(fleas) == size*size
    return fleas

def make_grid(size=30, Nfleas=0):
    """
    Make a size by size grid, initialised to have Nfleas (default 0)
    in each square.
    """
    grid = []
    for i in range(size):
        grid.append([Nfleas]*size)
    return grid

def grid_from_locations(fleas, size):
    """
    Given a list of locations of the fleas (on a grid of the given size),
    make a two-dimensional grid containing the number of fleas in each
    location.
    """
    grid = make_grid(size)

    # Populate the grid with the fleas
    for flea in fleas:
        grid[flea[0]][flea[1]] += 1
    return grid

def print_grid(grid):
    """
    Print the grid
    """
    for row in grid:
        for col in row:
            print col,
        print

def empty_squares(grid):
    """
    Return the number of empty squares in the given grid.
    """
    empty = 0
    for row in grid:
```

```
        for square in row:
            if square == 0:
                empty += 1
    return empty

def test_single_run(rings=50, size=10):
    """
    Make a single run of the given number of rings (default 50), using a
    grid of the given size (default 10).

    Print the grid and the number of empty squares at the end of the run
    and check that no fleas have been lost or gained.
    """
    fleas = single_run(Nrings=rings, size=size)
    grid = grid_from_locations(fleas, size)
    print_grid(grid)
    print 'There are %d empty squares' % empty_squares(grid)

    # Count the total number of fleas
    N = 0
    for row in grid:
        for square in row:
            N += square
    assert N == size*size

def average_occupancy(rings=50, size=30, trials=10):
    """
    Find the average number of empty squares (over trials runs) for fleas
    jumping on a size by size grid (default size 30).
    """
    empty = 0
    for n in range(trials):
        fleas = single_run(Nrings=rings, size=size)
        grid = grid_from_locations(fleas, size)
        empty += empty_squares(grid)
    return empty/float(trials)

if __name__ == "__main__":
    rings = 50
    size = 30
    trials = 100
    test_single_run(rings, size)

    empty = average_occupancy(rings=rings, size=size, trials=trials)
    print 'Average (%d trials) occupancy over %d rings on a %d by %d grid is %
        (trials, rings, size, size, empty, empty/float(size*size))
```

Here is the output:

```
$ python fleas.py
1 0 0 2 2 1 1 1 1 1 3 0 0 1 1 1 1 0 2 1 1 1 2 0 2 0 1 0 0 1
1 1 0 0 1 0 1 2 2 2 2 1 4 4 2 1 2 1 3 1 2 1 1 1 0 1 1 2 1 0
0 1 2 1 1 2 0 2 0 1 0 2 0 2 0 0 0 1 3 2 0 4 0 1 0 0 4 1 1 1
1 1 0 0 0 1 2 0 2 1 0 0 0 0 0 2 1 0 0 0 3 0 1 4 4 0 1 0 1 3
0 0 0 1 1 0 2 0 1 1 2 0 0 4 0 1 0 1 0 2 2 2 0 0 1 3 1 4 0 2
2 3 1 1 2 0 1 1 0 1 1 2 1 1 2 0 1 1 2 1 0 2 2 0 1 0 1 2 0 1
0 0 1 0 0 3 0 0 1 0 4 0 2 1 0 1 0 0 1 0 0 0 2 2 0 1 2 2 0 1
2 2 0 0 2 1 1 1 1 1 1 3 1 1 0 3 1 2 0 0 1 2 0 1 1 0 2 1 0 0
1 1 0 0 2 0 1 1 1 0 1 1 2 1 0 0 1 0 1 1 0 1 0 1 0 0 0 1 1 2
4 0 0 1 1 0 1 3 1 0 3 1 0 0 1 0 0 0 1 2 3 1 0 3 0 4 2 1 0 0
0 1 2 1 2 1 0 2 4 0 2 1 0 1 1 0 0 1 0 3 0 0 1 1 0 0 2 0 2 1
1 0 1 3 1 2 2 1 2 1 1 2 0 2 0 0 1 4 1 1 2 0 2 1 0 0 0 3 1 3
0 0 2 2 2 2 2 0 1 1 2 1 0 3 1 2 2 1 0 0 1 0 2 1 0 1 0 1 3 0
0 0 1 1 0 0 0 1 2 1 0 0 0 1 0 3 1 1 0 1 1 0 0 3 0 1 4 1 0 0
0 0 2 0 0 1 0 1 0 2 0 0 1 0 3 1 0 2 1 0 0 1 2 1 0 1 0 0 1 0
1 0 3 2 0 0 3 1 0 0 0 0 1 1 0 0 3 1 1 1 1 1 1 2 2 1 2 0 2 0
2 1 0 1 0 1 2 0 2 0 0 1 0 1 0 0 2 0 0 1 4 1 0 0 2 1 1 0 0 2
0 2 0 0 1 0 0 1 1 1 2 0 1 1 0 0 0 0 2 1 2 1 2 1 1 0 0 0 1 0
2 1 2 0 0 2 1 2 0 1 1 1 1 0 2 3 0 2 1 0 1 2 0 2 2 0 3 2 0 1
0 3 0 1 1 0 1 2 3 2 1 1 0 0 1 1 0 0 1 3 1 0 0 2 1 0 0 1 0 3
1 2 3 0 2 1 1 2 3 1 1 2 3 1 1 0 2 2 1 0 0 0 0 1 3 3 2 1 0 0
1 2 4 0 5 0 1 2 1 2 0 1 0 2 2 0 0 1 1 2 0 0 1 1 1 1 0 1 2 1
1 1 0 0 1 1 0 1 0 1 4 0 0 1 2 4 0 0 2 0 0 1 1 0 1 2 0 0 0 2
0 2 1 3 1 1 0 0 3 2 3 0 1 0 0 2 0 0 1 2 1 4 0 1 3 2 0 2 0 1
2 0 1 1 1 0 0 0 0 0 2 2 2 3 1 0 0 2 1 1 0 2 2 1 1 3 3 1 3 1
0 3 0 1 1 1 1 2 0 1 1 0 0 1 1 0 2 0 0 1 1 1 2 1 0 0 1 1 0 1
0 2 0 1 0 1 0 2 2 0 2 0 0 0 2 0 0 2 3 0 0 1 1 3 1 0 1 1 1 1
3 1 1 2 3 2 1 1 1 1 4 1 2 3 0 0 1 0 0 2 0 0 2 2 0 1 0 1 1 2
1 0 1 0 0 4 1 1 0 0 2 4 3 1 0 2 1 4 1 1 0 2 2 2 0 0 0 1 1 2
0 0 0 0 2 0 1 1 0 1 2 0 0 0 0 1 1 0 1 2 0 0 1 0 1 1 1 1 0 0
There are 341 empty squares
Average occupancy over 50 rings on a 30 by 30 grid is 330.86 (fraction 0.367622)
```

An alternative way of representing the fleas is to keep a count of the number of fleas in each square. In this case we have to keep a another grid which holds the locations of the fleas *after* they have jumped. Code using this approach is in the following, which imports several functions from `fleas.py`. It turns out that the first representation is faster (about 7.6s for 50 rings compared with 11.5s on my computer) and I think the first is easier to code, but you might think different – they give the same results.

```
#
#  A second version of the fleas program that doesn't keep a list of the fleas
#

from fleas import legal_jump, print_grid, empty_squares, make_grid

def jump_grid(grid):
    """
    Make the fleas in the grid jump, returning a new grid with their
    updated positions.
    """
    size = len(grid)
    newgrid = make_grid(size, 0)

    for i in range(size):
        for j in range(size):
            # Make each flea in this square jump, recording where
            # it jumps in newgrid
            for fleas in range(grid[i][j]):
                p, q = legal_jump(i, j, size)
                newgrid[p][q] += 1
    return newgrid
```

```
def single_run(Nrings=50, size=30):
    """
    Simulate the Nrings jumps of the fleas on a size by size grid.

    Returns a list of the fleas locations at the end of the run.
    """
    # Make a grid with one flea in each square
    grid = make_grid(size, 1)

    for ring in range(Nrings):
        newgrid = jump_grid(grid)
        grid = newgrid
    return grid

def test_single_run(rings=50, size=30):
    """
    Make a single run of the given number of rings (default 50), using a
    grid of the given size (default 10).

    Print the grid and the number of empty squares at the end of the run
    and check that no fleas have been lost or gained.
    """
    grid = single_run(rings, size)
    print_grid(grid)
    print 'There are %d empty squares' % empty_squares(grid)

    # Count the total number of fleas
    N = 0
    for row in grid:
        for square in row:
            N += square
    assert N == size*size


def average_occupancy(rings=50, size=30, trials=10):
    """
    Find the average number of empty squares (over trials runs) for fleas
    jumping on a size by size grid (default size 30).
    """
    empty = 0
    for n in range(trials):
        grid = single_run(Nrings=rings, size=size)
        empty += empty_squares(grid)
    return empty/float(trials)

if __name__ == "__main__":
    rings = 50
```

```
    size = 30
    trials = 100
    test_single_run(rings, size)

    empty = average_occupancy(rings=rings, size=size, trials=trials)
    print 'Average (%d trials) occupancy over %d rings on a %d by %d grid is %
        (trials, rings, size, size, empty, empty/float(size*size))
```