Program Design

Invasion Percolation: Assembly

Copyright © Software Carpentry 2010
This work is licensed under the Creative Commons Attribution License
See http://software-carpentry.org/license.html for more information.
We now know how to:
We now know how to:

- create a grid
We now know how to:

- create a grid
- fill it with random numbers
We now know how to:
- create a grid
- fill it with random numbers
- mark cells that have been filled
We now know how to:
- create a grid
- fill it with random numbers
- mark cells that have been filled
- find cells that might be filled next
We now know how to:
- create a grid
- fill it with random numbers
- mark cells that have been filled
- find cells that might be filled next
- choose one of them at random
We now know how to:
- create a grid
- fill it with random numbers
- mark cells that have been filled
- find cells that might be filled next
- choose one of them at random

It’s time to put everything together
We will show things in exactly the order that we would write them
We will show things in exactly the order that we would write them

Start at the top and work down...
We will show things in exactly the order that we would write them
Start at the top and work down...
...introducing functions and variables as we need them...
We will show things in exactly the order that we would write them
Start at the top and work down...
...introducing functions and variables as we need them...
...and tidying up a bit along the way
'''Invasion percolation simulation.
usage: invperc.py grid_size value_range random_seed
'''

import sys, random

# Main driver.
if __name__ == '__main__':
    # Get parameters from command line.
    # Run simulation.
    # Report results.
# Program Design

**Invasion Percolation**

Assembly

```python
'''Invasion percolation simulation.
usage: invperc.py grid_size value_range random_seed

import sys, random

# Main driver.
if __name__ == '__main__':
    # Get parameters from command line.
    # Run simulation.
    # Report result.
```

Import the whole module instead of just the functions we are going to use.
# Get parameters from the command line.
arguments = sys.argv[1:]
try:
    grid_size = int(arguments[0])
    value_range = int(arguments[1])
    random_seed = int(arguments[2])
except IndexError:
    fail('Expected 3 arguments, got %d' % len(arguments))
except ValueError:
    fail('Expected int arguments, got %s' % str(arguments))
# Get parameters from the command line.
arguments = sys.argv[1:]
try:
    grid_size = int(arguments[0])
    value_range = int(arguments[1])
    random_seed = int(arguments[2])
except IndexError:
    fail('Expected 3 arguments, got %d' % len(arguments))
except ValueError:
    fail('Expected int arguments, got %s' % str(arguments))

Now we write this function...
def fail(msg):
    '''Print error message and halt program.'''
    print >> sys.stderr, msg
    sys.exit(1)
def fail(msg):
    '''Print error message and halt program.'''
    print >> sys.stderr, msg
    sys.exit(1)
# Run simulation.
random.seed(random_seed)
grid = create_random_grid(grid_size, value_range)
mark_filled(grid, grid_size/2, grid_size/2)
fill_grid(grid)
# Run simulation.
random.seed(random_seed)
grid = create_random_grid(grid_size, value_range)
mark_filled(grid, grid_size/2, grid_size/2)
fill_grid(grid)

Three more functions to write...
# Report results.
# Report results.

We haven't actually decided what to do.
# Report results.

We haven't actually decided what to do.

For now, let's just count the number of filled cells.
# Run simulation.
random.seed(random_seed)
grid = create_random_grid(grid_size, value_range)
mark_filled(grid, grid_size/2, grid_size/2)

# Report results.
num_filled_cells = fill_grid(grid) + 1
print '%d cells filled' % num_filled_cells
# Run simulation.
random.seed(random_seed)
grid = create_random_grid(grid_size, value_range)
mark_filled(grid, grid_size/2, grid_size/2)

# Report results.
num_filled_cells = fill_grid(grid) + 1
print '%d cells filled' % num_filled_cells

Because we filled one cell on the previous line to get things started
def create_random_grid(N, Z):
    assert N > 0, 'Grid size must be positive'
    assert N%2 == 1, 'Grid size must be odd'
    assert Z > 0, 'Random range must be positive'
    grid = []
    for x in range(N):
        grid.append([])
        for y in range(N):
            grid[-1].append(random.randint(1, Z))
    return grid
def create_random_grid(N, Z):
    assert N > 0, 'Grid size must be positive'
    assert N%2 == 1, 'Grid size must be odd'
    assert Z > 0, 'Random range must be positive'
    grid = []
    for x in range(N):
        grid.append([])
        for y in range(N):
            grid[-1].append(random.randint(1, Z))
    return grid

A little documentation would help...
def create_random_grid(N, Z):
    '''Return an N x N grid of random values in 1..Z.
    Assumes the RNG has already been seeded.'''
    assert N > 0, 'Grid size must be positive'
    assert N%2 == 1, 'Grid size must be odd'
    assert Z > 0, 'Random range must be positive'
    grid = []
    for x in range(N):
        grid.append([])
        for y in range(N):
            grid[-1].append(random.randint(1, Z))
    return grid
def create_random_grid(N, Z):
    '''Return an NxN grid of random values in 1..Z.
    Assumes the RNG has already been seeded.'''
    assert N > 0, 'Grid size must be positive'
    assert N%2 == 1, 'Grid size must be odd'
    assert Z > 0, 'Random range must be positive'
    grid = []
    for x in range(N):
        grid.append([])
        for y in range(N):
            grid[-1].append(random.randint(1, Z))
    return grid
def mark_filled(grid, x, y):
    '''Mark a grid cell as filled.'''

    assert 0 <= x < len(grid), \
    'X coordinate out of range (%d vs %d)' % \
    (x, len(grid))
    assert 0 <= y < len(grid), \
    'Y coordinate out of range (%d vs %d)' % \
    (y, len(grid))

    grid[x][y] = -1
def mark_filled(grid, x, y):
    '''Mark a grid cell as filled.'''

    assert 0 <= x < len(grid),
    'X coordinate out of range (%d vs %d)' % (x, len(grid))
    assert 0 <= y < len(grid),
    'Y coordinate out of range (%d vs %d)' % (y, len(grid))

    grid[x][y] = -1 ← Will people understand this?
FILLED = -1

...other functions...

def mark_filled(grid, x, y):
    ...body of function...
    grid[x][y] = FILLED
FILLED = -1

...other functions...

def mark_filled(grid, x, y):
    ...body of function...
    grid[x][y] = FILLED
```python
def fill_grid(grid):
    '''Fill an NxN grid until filled region hits boundary.
    Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled
```
def fill_grid(grid):
    '''Fill an N\times N grid until filled region hits boundary.
    Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled
def fill_grid(grid):
    '''Fill an N x N grid until filled region hits boundary. Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled

Almost always signals an "exit in the middle" loop
def fill_grid(grid):
    '''Fill an NxN grid until filled region hits boundary. Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled

The actual loop test and exit
def fill_grid(grid):
    '''Fill an N x N grid until filled region hits boundary. Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled

Another function for us to write
def fill_grid(grid):
    '''Fill an NxN grid until filled region hits boundary. Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled

Fail early, often, and loudly
def fill_grid(grid):
    '''Fill an N x N grid until filled region hits boundary.
    Assumes center cell filled before call.''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled

Fill and keep count
def fill_grid(grid):
    '''Fill an NxB grid until filled region hits boundary. Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled

Break out of the loop when we reach a boundary cell.
def fill_grid(grid):
    '''Fill an N x N grid until filled region hits boundary. Assumes center cell filled before call.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled

Report how many cells this function filled
```python
def fill_grid(grid):
    '''Fill an N\times N grid until filled region hits boundary.'''
    N, num_filled = len(grid), 0
    while True:
        candidates = find_candidates(grid)
        assert candidates, 'No fillable cells found!'  
        x, y = random.choice(list(candidates))
        mark_filled(grid, x, y)
        num_filled += 1
        if x in (0, N-1) or y in (0, N-1):
            break
    return num_filled
```

Program Design  Invasion Percolation  Assembly
def find_candidates(grid):
    '''Find low-valued neighbor cells.'''
    N = len(grid)
    min_val = sys.maxint
    min_set = set()
    for x in range(N):
        for y in range(N):
            if (x > 0) and (grid[x-1][y] == FILLED) or (x < N-1) and (grid[x+1][y] == FILLED)
            or (y > 0) and (grid[x][y+1] == FILLED) or (y < N-1) and (grid[x][y+1] == FILLED):
                min_set.add((x, y))
    return min_set
def find_candidates(grid):
    '''Find low-valued neighbor cells.''
    N = len(grid)
    min_val = sys.maxint
    min_set = set()
    for x in range(N):
        for y in range(N):
            if (x > 0) and (grid[x-1][y] == FILLED) or (x < N-1) and (grid[x+1][y] == FILLED) or (y > 0) and (grid[x][y+1] == FILLED) or (y < N-1) and (grid[x][y+1] == FILLED):
                Let's stop right there.
def find_candidates(grid):
    '''Find low-valued neighbor cells.'''
    N = len(grid)
    min_val = sys.maxint
    min_set = set()
    for x in range(N):
        for y in range(N):
            if (x > 0) and (grid[x-1][y] == FILLED) 
            or (x < N-1) and (grid[x+1][y] == FILLED) 
            or (y > 0) and (grid[x][y+1] == FILLED) 
            or (y < N-1) and (grid[x][y+1] == FILLED):

                That's kind of hard to read.
def find_candidates(grid):
    '''Find low-valued neighbor cells.'''
    N = len(grid)
    min_val = sys.maxint
    min_set = set()
    for x in range(N):
        for y in range(N):
            if (x > 0) and (grid[x-1][y] == FILLED) or (x < N-1) and (grid[x+1][y] == FILLED) or (y > 0) and (grid[x][y+1] == FILLED) or (y < N-1) and (grid[x][y+1] == FILLED):
                min_set.add((x, y))

    min_set = sorted(min_set, key=lambda x: grid[x[0]][x[1]])
    min_set = min_set[:min_val]

    return min_set

In fact, it contains a bug.
def find_candidates(grid):
    '''Find low-valued neighbor cells.''
    N = len(grid)
    min_val = sys.maxint
    min_set = set()
    for x in range(N):
        for y in range(N):
            if (x > 0) and (grid[x-1][y] == FILLED) or (x < N-1) and (grid[x+1][y] == FILLED) or (y > 0) and (grid[x][y+1] == FILLED) or (y < N-1) and (grid[x][y+1] == FILLED):
                Should be y-1
Listen to your code as you write it.
def find_candidates(grid):
    '''Find low-valued neighbor cells.'''
    N = len(grid)
    min_val = sys.maxint
    min_set = set()
    for x in range(N):
        for y in range(N):
            if is_candidate(grid, x, y):
def find_candidates(grid):
    '''Find low-valued neighbor cells.'''
    N = len(grid)
    min_val = sys.maxint
    min_set = set()
    for x in range(N):
        for y in range(N):
            if is_candidate(grid, x, y):
                Much clearer when read aloud.
def find_candidates(grid):
    ...loop...:
        if is_candidate(grid, x, y):
            # Has current lowest value.
            if grid[x][y] == min_val:
                min_set.add((x, y))
            # New lowest value.
            elif grid[x][y] < min_val:
                min_val = grid[x][y]
                min_set = set([(x, y)])
def find_candidates(grid):
    ...loop...:
        if is_candidate(grid, x, y):
            # Has current lowest value.
            if grid[x][y] == min_val:
                min_set.add((x, y))
            # New lowest value.
            elif grid[x][y] < min_val:
                min_val = grid[x][y]
                min_set = set([(x, y)])
    ...
def is_candidate(grid, x, y):
    '''Determine whether the cell at (x,y) is now a candidate for filling.'''
    ...see previous episode...
It's finally time to run our program.
It's finally time to run our program.
It's finally time to run our program.

Because there's a bug lurking in what we just wrote.
It's finally time to run our program.
Because there's a bug lurking in what we just wrote.
Try to find it by reading the code carefully before moving on.