Agent-based models of segregation using Python

Patrick Vincent N. Lubenia

6 May 2020

Contents

Agent-Based Models

Agent-based modeling is a framework for creating and simulating models of complex systems. It is a mindset wherein a system is described from the point of view of the units constituting it [1]. Agent-based models are computational simulation models that involve numerous discrete agents. They show a system's emergent collective behavior resulting from the interactions of the agents. In contrast to equation-based models, each agent's behaviors in an agent-based model are described in an algorithmic fashion by rules rather than equations. Agents in the model do not typically perform actions together at constant time-steps [2]. Their decisions follow discrete-event cues or a series of interactions.

Depending on one's objectives, agents: are discrete entities, may have internal states, may be spatially localized, may perceive and interact with the environment, may behave based on predefined rules, may be able to learn and adapt, and may interact with other agents. Generally, agent-based models: often lack central supervisors/controllers and may produce nontrivial "collective behavior" as a whole.

The following scientific method-based approach must be kept in mind when designing an agent-based model:

1. Specific problem to be solved by the model
2. Availability of data
3. Method of model validation.

In order to be scientifically meaningful, an agent-based model must be:

1. Built using empirically-derived assumptions, then simulate to produce emergent behavior: for predictions; or
2. Built using hypothetical assumptions, then simulate to reproduce observed behavior: for explanations.

Once a code has been programmed, its basic implementation structure has 3 parts:

1. Initialization
2. Updating
3. Visualization.

Agents are initially placed in the model's environment. The system is then updated according to rules that govern the behavior of the environment and/or agents. Finally, states are visualized in order to appreciate the changes in the system.

The agent-based modeling framework is open-ended and flexible. It may be tempting to be detailed to make a model more realistic. But it must be remembered that increased complexity leads to increased difficulty in analysis. Moreover, the open-endedness of the framework makes it code-intense as lots of details of the simulation must be manually taken care of. Thus, codes must be kept simple and organized.

Python Basics: class

`class` with Methods

The main tool used in agent-based modeling in Python is a `class`. A `class` is created when objects with their own built-in functions (called *methods*) are desired. The following example creates a `rectangle` class with built-in functions for computing the rectangle's perimeter and area:

```python
# Initialize the class
class rectangle:

  # Initialize object when rectangle is called
  # length, width: needed inputs
  def __init__(self, length, width):

    # Assign variable length to attribute called length
    self.length = length

    # Assign variable width to attribute called width
    self.width = width

  # Define method called perimiter
  # No input needed: Uses variables in initialized object
```

```
  def perimeter(self):

    # Return perimeter of the rectangle
    return 2*self.length + 2*self.width

  # Define method called area
  # No input needed: Uses variables in initialized object
  def area(self):

    # Return area of the rectangle
    return self.length*self.width
```

self is a placeholder for the variable name given to the rectangle. To create a rectangle r with length 3 and width 4, type

```
r = rectangle(3, 4)
```

To retrieve the length and width of rectangle r, type

```
r.length
r.width
```

To get the perimeter and area of rectangle r, type

```
r.perimeter()
r.area()
```

Note that additional variables, which are not taken from the inputs to the class object, can be defined. For example, a variable called diagonal, which does not initially have a value:

```
class rectangle:
  def __init__(self, length, width):
    self.length = length
    self.width = width

    # Not in the list of needed inputs
    # Initialized for methods that use the variable diagonal
    self.diagonal = []
```

After creating rectangle r as above, typing

```
r.diagonal
```

returns an empty list. The following adds a function that computes the diagonal of the rectangle and places that value inside the variable called diagonal. The full code is as follows:

```
# Needed for computing square root
import numpy as np

class rectangle:
  def __init__(self, length, width):
    self.length = length
    self.width = width
    self.diagonal = []
  def perimeter(self):
    return 2*self.length + 2*self.width
  def area(self):
    return self.length*self.width
  def diag(self):
    self.diagonal = np.sqrt(self.length**2 + self.width**2)
```

The following computes the value of the diagonal places it in `r.diagonal`:

```
r.diag()
```

There is no output, however, because the defined function does not specify a `return` value. This use of `class` is utilized in the first model.

Empty `class`

Sometimes, a `class` is created when objects are needed to be flexible enough to be given desired attributes. In

```
1 class perfomer:
2   pass
```

the `pass` on line 2 allows one to create an empty class. This class can now be used to create a performer:

```
performer_ = performer()
```

It is simple to add attributes to the object called performer, say its location in terms of coordinates, name, and age:

```
# Assign attribute x to performer_
performer_.x = 3

# Assign attribute y to performer_
performer_.y = 4

# Assign attribute name to performer_
performer_.name = 'Luca'

# Assign attribute age to performer_
performer_.age = 16
```

This use of `class` is utilized in the second model.

Model 1

Overview

The following model is based on a tutorial by Moujahid [3]. It follows an agent-based modeling framework with the following structure:

1. Environment: Where the individual components of the system move around
2. Agents: Units interacting with each other and making decisions
3. Metrics: Rules that the agents follow
4. Behavior: How agents interact.

The case study to be used is the Schelling segregation model which was named after Thomas Schelling, an American economist and a Nobel laureate in economics. His segregation model studie the emergent segregation that occurs in a community with 2 groups of people [5]. The question that the model wishes to answer is: *How high should the residents' threshold be in order for segregation to occur?*

To start the modeling process, the elements of the framework are defined as follows:

1. Environment: A community with houses
2. Agents: Residents of the community
3. Metrics: A resident is happy if a certain number of his neighbors have the same group as his
4. Behavior: A resident stays put if he is happy; otherwise, he moves to a different location

The Code

Needed Libraries

To start the code, the following libraries are imported:

```
# For pairing x- and y-values to create Cartesian products; an ordered pair represents the address a
    resident
import itertools

# For shuffling all houses in the community to randomly assign them to residents; and for choosing a
    random empty house to move into (for unhappy residents)
import random as rd

```

```
# For visualizing the distribution of the residents using scatterplot
import matplotlib.pyplot as plt

# For checking if a filename already exists, to avoid overwriting files
import os
```

Environment: creating the class

The community is represented by a grid, and each grid cell represents a house which can be occupied by at most one resident. The following need to be defined:

- Dimensions of the community
- Percentage of houses that are empty (so residents have places to move into if they are unhappy)
- The happiness threshold that determines if a resident moves or not
- Number of groups in the community.

Define a class called `Schelling`:

```
class Schelling:
  def __init__(self, width, height, empty_ratio, happiness_threshold, groups, n_iterations):

    # Width of community grid
    self.width = width

    # Height of community grid
    self.height = height

    # Percentage of empty houses
    self.empty_ratio = empty_ratio

    # Minimum similarity ratio, to be considered happy
    self.happiness_threshold = happiness_threshold

    # Number of groups of residents
    self.groups = groups

    # Maximum iterations
    self.n_iterations = n_iterations

    # List of empty houses
    self.empty_houses = []

    # Dictionary of residents
    self.agents = {}
```

Calling `Schelling` needs 6 inputs. There are 2 extra variables that are used later: empty_houses and agents. In the dictionary of residents, each resident is represented by a tuple whose first element is an ordered pair (its address) and second element is a number (its group number).

Agents: creating a method

A method called `populate` is added. It randomly scatters the residents throughout the community:

```
  def populate(self):

    # Create address (coordinates) of residents
    all_houses = list(itertools.product(range(self.width), range(self.height)))

    # Shuffle the order of houses
    rd.shuffle(all_houses)

    # Determine number of empty houses
    n_empty = int(self.empty_ratio*len(all_houses))

    # Assign first few houses as empty
    self.empty_houses = all_houses[ : n_empty]

    # Assign the rest to be occupied
    remaining_houses = all_houses[n_empty : ]
```

```

    # Assign houses by group to residents
    houses_by_group = [remaining_houses[i::self.groups] for i in range(self.groups)]

    # Create dictionary of residents
    # Each resident is defined by his address and group number
    for i in range(self.groups):
      agent = dict(zip(houses_by_group[i], [i+1]*len(houses_by_group[i])))
      self.agents.update(agent)
```

Metrics: creating a method

In the `is_unhappy` method, for each resident, it checks each neighbor to see if it is of the same group as the resident, computes the happiness of the resident, then sees if he is happy. This method is of a negative nature ("is UNhappy" instead of "is happy") so that if it is true, i.e., the resident is unhappy, then the the resident transfers to another house. The method is used inside the `move` method.

```
  def is_unhappy(self, x, y):

    # Get group number of resident
    group = self.agents[(x, y)]

    # Initialize variables
    count_similar = 0
    count_different = 0

    # Check similarity with bottom left neighbor
    if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
      if self.agents[(x-1, y-1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with bottom neighbor
    if y > 0 and (x, y-1) not in self.empty_houses:
      if self.agents[(x, y-1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with bottom right neighbor
    if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
      if self.agents[(x+1, y-1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with left neighbor
    if x > 0 and (x-1, y) not in self.empty_houses:
      if self.agents[(x-1,y)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with right neighbor
    if x < (self.width-1) and (x+1, y) not in self.empty_houses:
      if self.agents[(x+1,y)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with upper left neighbor
    if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
      if self.agents[(x-1,y+1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with upper neighbor
    if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
      if self.agents[(x,y+1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with upper right neighbor
```

```
    if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
      if self.agents[(x+1,y+1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Resident is NOT unhappy, i.e., happy if he has no neighbors
    if (count_similar + count_different) == 0:
      return False

    # Check if similarity ratio is below happiness threshold
    else:
      return float(count_similar/(count_similar + count_different)) < self.happiness_threshold
```

In each similarity check, if a neighbor is of the same group as the resident, then a count is added to count_similar. Otherwise, a count is added to count_different. If the similarity ratio is less than the happiness threshold, then `is_unhappy` returns True, i.e., the resident is indeed unhappy. If the similarity ratio is NOT less than (i.e., more than) the happiness threshold, then `is_unhappy` returns False, i.e., the resident is actually happy. Note that The method is only triggered when a resident is unhappy. Once triggered, the resident is moved to action.

Behavior: creating a method

`move` allows residents to transfer to another house if they are unhappy. it uses the `is_unhappy` method.

```
  def move(self):

    # Maximum iterations allowed
    for i in range(self.n_iterations):

      # Initialize count
      n_changes = 0

      # Check each resident
      for agent in self.agents:

        # Activated if resident is unhappy
        if self.is_unhappy(agent[0], agent[1]):

          # Get a random empty house
          empty_house = rd.choice(self.empty_houses)

          # Get group number of the resident
          agent_group = self.agents[agent]

          # Assign the empty house to the resident
          self.agents[empty_house] = agent_group

          # Remove the original residence from the dictionary
          del self.agents[agent]

          # Remove the now-occupied house from the list of empty houses
          self.empty_houses.remove(empty_house)

          # Add the house the resident just left to the list of empty houses
          self.empty_houses.append(agent)

          # Count as a move
          n_changes += 1

      # Iteration stops if everyone is already happy
      if n_changes == 0:
        break
```

Quantifying Similarity: Similarity Ratio

The model is already complete at this point. But to go the extra mile, the overall similarity ratio of the entire community can be computed using the `similarity` method. The similarity ratio in each house is computed (as in the `is_unhappy` method) and the average over all households is taken.

```
  def similarity(self):

    # Initialize list
    similarity = []
```

```
    # Do for each resident
    for agent in self.agents:

      # Initialize variables
      count_similar = 0
      count_different = 0

      # Get address and group number of the resident
      x = agent[0]
      y = agent[1]
      group = self.agents[(x,y)]

      # Check similarity with bottom left neighbor
      if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
        if self.agents[(x-1, y-1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with bottom neighbor
      if y > 0 and (x, y-1) not in self.empty_houses:
        if self.agents[(x, y-1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with bottom right neighbor
      if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
        if self.agents[(x+1, y-1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with left neighbor
      if x > 0 and (x-1, y) not in self.empty_houses:
        if self.agents[(x-1, y)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with right neighbor
      if x < (self.width-1) and (x+1, y) not in self.empty_houses:
        if self.agents[(x+1, y)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with upper left neighbor
      if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
        if self.agents[(x-1, y+1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with upper neighbor
      if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
        if self.agents[(x, y+1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with upper right neighbor
      if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
        if self.agents[(x+1,y+1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Place similarity ratio in the list
      try:
        similarity.append(float(count_similar/(count_similar + count_different)))

      # If there are no neighbors, similarity ratio is 1
      except:
        similarity.append(1)
```

```
    # Compute average similarity ratio over all residents
    return sum(similarity)/len(similarity)
```

Error in computation of similarity ratio only occurs when we 0/0 happens. In this case, a similarity ratio of 1 is used.

Visualization

A scatterplot is produced to visualize what happens to the residents. The method `visualize` does exactly this:

```
  def visualize(self, state):

    # Initialize subplots
    fig, ax = plt.subplots()

    # Assign color to each group (define more as needed)
    agent_colors = {1:'b', 2:'r', 3:'g', 4:'c', 5:'m', 6:'y', 7:'k'}

    # Create a scatterplot for each resident, colored based on his group
    for agent in self.agents:
      ax.scatter(agent[0]+0.5, agent[1]+0.5, color = agent_colors[self.agents[agent]], edgecolors = 'white')

    # Title
    ax.set_title('Schelling Model: ' + state + ' State' + '\n' + 'Similarity: ' + str("{:.1f}".format(schelling.similarity()*100)) + '%')

    # Horizontal axis label
    ax.set_xlabel(str(self.empty_ratio*100) + '% Empty Houses' + '\n' + str(self.happiness_threshold*100) + '% Happiness Threshold' + '\n' + str(self.groups) + ' groups')

    # Ensure all residents are visible
    ax.set_xlim([0, self.width])
    ax.set_ylim([0, self.height])

    # Remove extra tick marks on the axes
    ax.set_xticks([])
    ax.set_yticks([])

    # Prepare format of file name
    filename = 'Model1_' + state

    # Starting filename count
    i = 1

    # Check if filename already exists; add 1 if it does
    while os.path.exists('{}{:d}.png'.format(filename, i)):
      i += 1

    # Save figure
    plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)
```

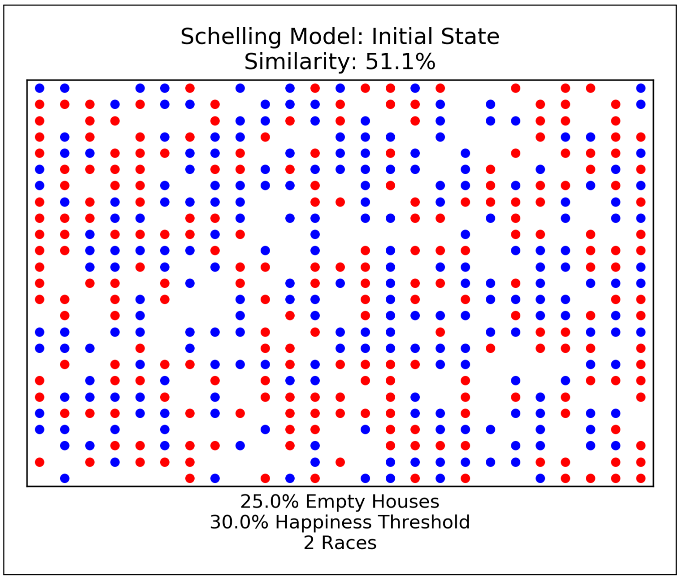
The method takes in a string which indicates the state of the system (ideally either “Initial” or “Final”). Subplots are used instead of the simpler `plt.scatter` so that when the file is run, successive runs of plots do not overlap into one figure. Up to 7 groups are assigned colors. New colors must be assigned if more than 7 groups are created. The additional 0.5 in the coordinates ensures that the plots on the edges are not squished to the sides, and the white edgecolor ensures that when plots are too close to each other, they can still be identified from each other. The tight specification in `savefig` removes extra white spaces around the figure.

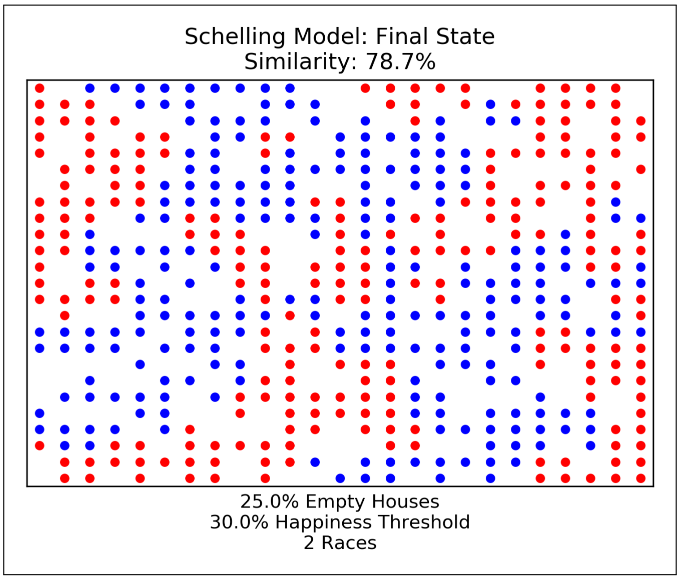
Implementation

The simulation follows the basic code implementation structure mentioned in the introduction: agents are created (Initialization), then they are allowed to move according to their defined behavior (Updating). Plots are used to visualize (Visualization) what happens to the system before and after the agents move. The model simulates a community 25×25 big with 25% empty houses populated by two groups. Residents are already happy if 30% of their neighbors are from the same group as they do. Residents are allowed to move up to 500 times if they are unhappy.

```
# Initialize parameters
schelling = Schelling(width = 25, height = 25, empty_ratio = 0.25, happiness_threshold = 0.30, groups = 2, n_iterations = 500)

# Place residents on the community
schelling.populate()
```

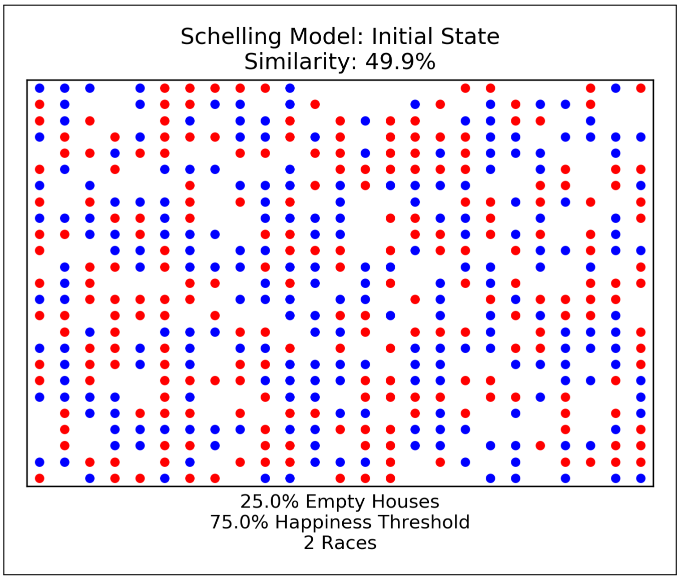
```
# Visualize initial state
schelling.visualize('Initial')

# Allow unhappy residents to move
schelling.move()

# Visualize final state
schelling.visualize('Final')
```

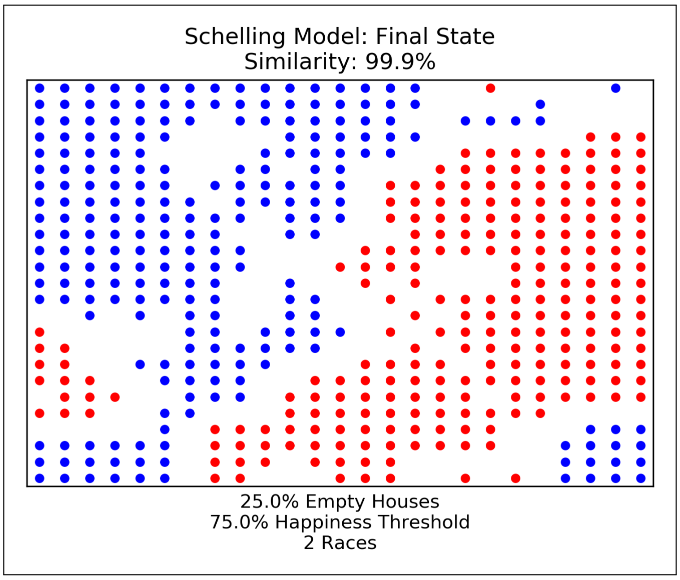




Initial overall similarity ratio among neighbors was 51.1%. After allowing unhappy residents to move, the overall similarity ratio increased to 78.7%, much higher than the happiness threshold of 30%. It can be seen visually that some form of segregation resulted in the final stage.

Using a higher happiness threshold of 75%:





Overall similarity ratio was initially 49.9%. The high happiness threshold resulted in an almost completely segregated community with overall similarity ratio of 99.9%. The diagram shows this clearly.

Randomly assigning residents to 2 groups distributes the population evenly, leading to an initial overall similarity ratio around 50%. However, even with a low happiness threshold of 30%, segregation naturally occurs, with overall similarity ratio much higher than what residents are willing to tolerate. Thus, on a macro level, observing segregation may not be indicative of what people feel at the micro level. Hence, the answer to the question *How high should the agents' threshold be in order for segregation to occur?* is: **not so high**.

Model 2

Overview

The following model is shorter than the previous one, and is based on the model by Sayama [4]. It follows an agent-based modeling framework with the following tasks that must be undertaken:

1. Design the data structure to store the:
   (a) Attributes of the agents
   (b) States of the environment
2. Describe the rules for how:
   (a) The environment behaves on its own
   (b) Agents interact with the environment
   (c) Agents behave on their own
   (d) Agents interact with each other.

Not all these tasks are needed in every agent-based model.

The succeeding model is slightly different from the previous one. In this model, there are still 2 groups of residents, and each resident observes his neighborhood. If he is not satisfied with the residents surrounding him, he moves to a different location. The differences:

- The environment is not a defined grid: the number of residents is defined, and they are located randomly
- The number of neighbors is not limited to 8: a neighborhood is defined using a radius.

The question still remains: *How high should the residents' threshold be in order for segregation to occur?*

The Code

Needed Libraries

To start the code, the following libraries are imported:

```
# For assigning a random location to agents
import random as rd

# For visualizing the distribution of agents using plot
import matplotlib.pyplot as plt

# For checking if a filename already exists, to avoid overwriting files
import os
```

Modeling Task 1a: Attributes of the agents

The first task is to "design the data structure to store the attributes of the agents". Each resident has the following attributes:

1. Spatial location: random coordinates
2. Group number: 0 or 1.

Note that this modeling task incorporates both the Environment and Agents parameters in the previous model.

The class called `agent` is initialized:

```
class agent:
  pass
```

A function called `create_agents` is defined to create the residents of the community:

```python
def create_agents():

  # Allow list of residents to be accessed outside the function
  global agents_list

  # Initialize list
  agents_list = []

  # Create specified number of residents
  for each_agent in range(n_agents):

    # Create an agent
    agent_ = agent()

    # Assign to a random address
    agent_.x = rd.random()
    agent_.y = rd.random()

    # Assign to a random group
    agent_.group = rd.randint(0, groups-1)

    # Place resident on the list
    agents_list.append(agent_)
```

Using `global` automatically creates the variable outside the function. Note that this function needs 2 variables to be predefined: n_agents (total number of residents) and groups (total number of groups).

The following function groups the residents by their number:

```python
def group_by_number():

  # Allow list of groups to be accessed outside the function
  global group

  # Initialize list
  group = []

  # Group according to group number
  for group_number in range(groups):

    # A resident is grouped with other residents with the same group number
    group.append([agent_ for agent_ in agents_list if agent_.group == group_number])
```

Note that once grouped, the residents remain in their respective groups. If they decide to move, they are still part of the same group, but their coordinates change.

Additional Notes:

1. There are no separate environments that interact with the residents so Modeling Tasks 1b ("design the data structure to store the states of the environment"), 2a ("describe the rules for how the environment behaves on its own"), and 2b ("describe the rules for how agents interact with the environment") are skipped.

2. Residents do not do anything by themselves so Modeling Task 2c ("describe the rules for how agents behave on their own") is also skipped.

Modeling Task 2d: How agents interact with each other

The other task that needs to be done is to "describe the rules for how agents interact with each other". A resident checks everyone within its neighborhood (defined by a radius). If he is satisfied with the number of people belonging to the same group as he does, he stays put. Otherwise, he moves to a different location. Note that this modeling task incorporates both the Metrics and Behavior parameters in the previous model. A function named `move` implements this:

```python
def move():

  # Allow number of iterations done to be accessed outside the function
  global iteration

  # Maximum iterations allowed
  for iteration in range(n_iterations):

    # Initialize count
```

```
    n_changes = 0

    # Check each resident
    for agent_ in agents_list:

      # Create list of neighbors within the radius specified
      neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y - neighbor.y)**2 < radius**2]

      # Remove resident himself from list of neighbors
      neighbors.remove(agent_)

      # Check if there are neighbors within the radius
      if len(neighbors) > 0:

        # Compute similarity ratio
        satisfaction = len([neighbor for neighbor in neighbors if neighbor.group == agent_.group])/len(neighbors)

        # Move resident to a random location if similarity ratio is below threshold
        if satisfaction < threshold:
          agent_.x, agent_.y = rd.random(), rd.random()

          # Count as a move
          n_changes += 1

    # Iteration stops if everyone is already happy
    if n_changes == 0:
      break
```

A neighbor is someone who falls within the defined radius, with the resident as the center. This function needs 3 additional variables to be predefined: n_interations (maximum number of iterations allowed), radius (how small the neighborhood of the resident that needs to be checked), and threshold (minimum similarity ratio in order to be considered satisfied). Note also that the change in coordinates does not affect the grouping done by the function `group_by_number` so there is no need to rerun the grouping function.

Quantifying Similarity: Similarity Ratio

The model is already complete at this point. But to go the extra mile, the overall similarity ratio of the entire community can be computed using the `similarity_ratio` function. The similarity ratio of each person is computed (as in the `move` function) and the average over all residents is taken.

```
def similarity_ratio():

  # Allow overall similarity ratio to be accessed outside the function
  global overall_similarity_ratio

  # Initialize list
  similarity_ratios = []

  # Check each resident
  for agent_ in agents_list:

    # Create list of neighbors within the radius specified
    neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y - neighbor.y)**2 < radius**2]

    # Remove resident himself from list of neighbors
    neighbors.remove(agent_)

    # Check if there are neighbors within the radius
    if len(neighbors) > 0:

      # Place similarity ratio in the list
      try:
        similarity_ratios.append(len([neighbor for neighbor in neighbors if neighbor.type == agent_.type])/len(neighbors))

      # If there are no neighbors, similarity ratio is 1
      except:
        similarity_ratios.append(1)

  # Compute average similarity ratio over all residents
  overall_similarity_ratio = sum(similarity_ratios)/len(similarity_ratios)
```

```

  # Outputs overall similarity ratio
  return overall_similarity_ratio
```

Error in computation of similarity ratio occurs only when we 0/0 happens. In this case, a similarity ratio of 1 is used.

Visualization

To visualize the states of the system, the function `visualize` is created:

```
def visualize(state):

  # Initialize subplots
  fig, ax = plt.subplots()

  # Create a scatterplot by group, plotting each resident
  for Group in range(groups):
    ax.plot([agent_.x for agent_ in group[Group]], [agent_.y for agent_ in group[Group]], 'o')

  # Title
  ax.set_title(state + ' State' + ' || ' + 'Segregation: ' + str("{:.1f}".format(similarity_ratio()*10
    0)) + '%')

  # Horizontal axis label
  ax.set_xlabel(str(n_agents) + ' Residents' + ' || ' + str(groups) + ' Groups' + ' || ' + str(radius*
    100) + '% Neighborhood || ' + str(threshold*100) + '% Threshold' + '\n' + 'moves: ' + str(iteration
    ))

  # Remove extra tick marks on the axes
  ax.set_xticks([])
  ax.set_yticks([])

  # Prepare format of file name
  filename = 'Model2_' + state

  # Starting filename count
  i = 1

  # Check if filename already exists; add 1 if it does
  while os.path.exists('{}{:d}.png'.format(filename, i)):
    i += 1

  # Save figure
  plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)
```

The function takes in a string which indicates the state of the system (ideally either "Initial" or "Final"). Subplots are used instead of the simpler `plt.scatter` so that when the file is run, successive runs of plots do not overlap into one figure. The tight specification in `savefig` removes extra white spaces around the figure.

Implementation

The simulation follows the basic code implementation structure mentioned in the introduction: agents are created (Initialization), then they are allowed to move according to their defined behavior (Updating). Plots are used to visualize (Visualization) what happens to the system before and after the agents move. To compare with the first model, the model simulates a community with 1,000 residents belonging to 2 groups. Each resident cares only about a neighborhood radius of 10%, and are already happy if 31% of their neighbors are from the same group as they do. Residents are allowed to move up to 500 times if they are unhappy.

```
# Number of residents
n_agents = 1000

# Number of groups
groups = 2

# Maximum iterations
n_iterations = 100

# Neighborhood radius
radius = 0.1

# Satisfaction threshold
threshold = 0.31
```

```
# Create residents
create_agents()

# Group residents according to group number
group_by_number()

# Needed for label of initial state
iteration = 0

# Visualize initial state
visualize('Initial')

# Allow unsatisfied residents to move
move()

# Visualize initial state
visualize('Final')
```

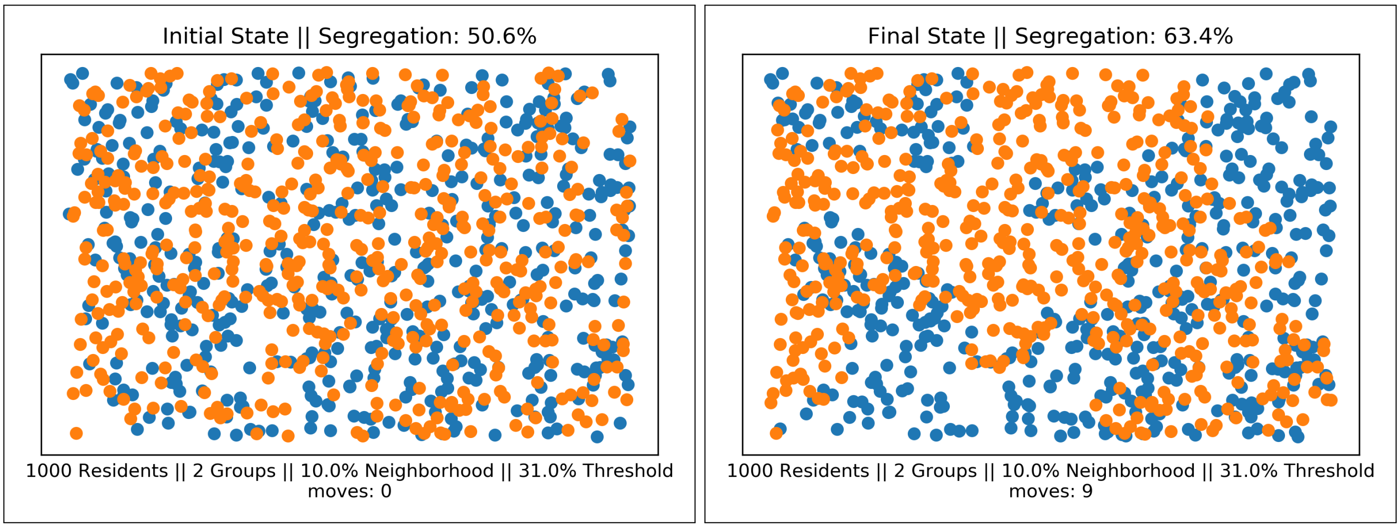


Figure 1: Simulation of Group Segregation. A high segregation level much higher than the happiness threshold naturally occurs.

Figure 1 shows that initial overall similarity ratio among neighbors was 50.6%. After allowing unhappy residents to move, the overall similarity ratio increased to 63.4%, much higher than the happiness threshold of 31%. It can be seen visually that some form of segregation resulted in the final stage.

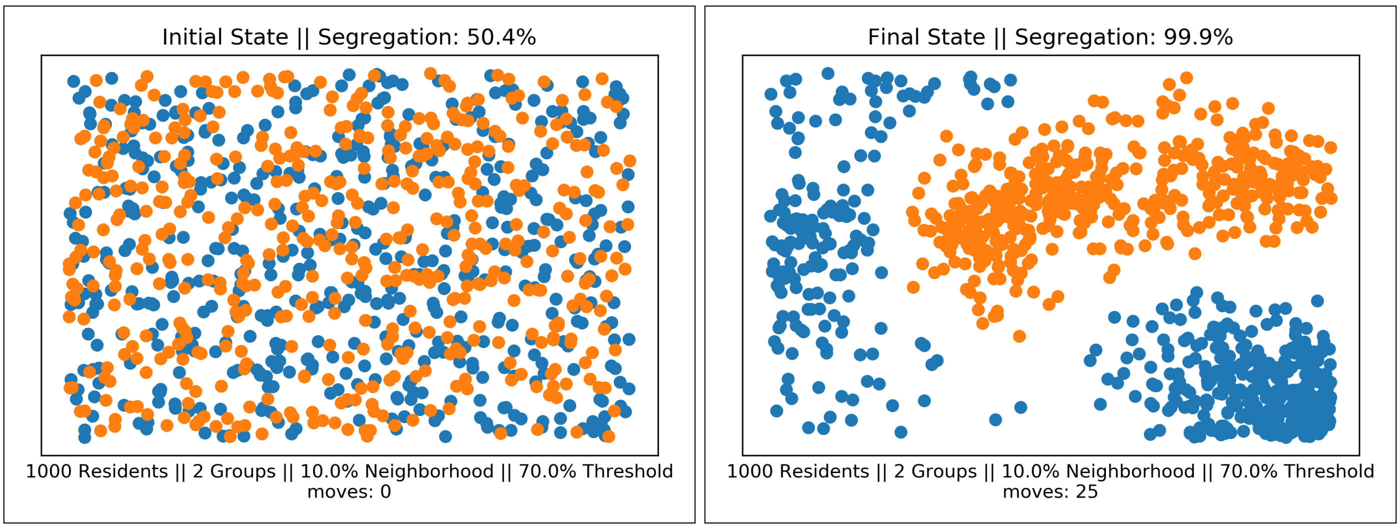


Figure 2: Simulation of Group Segregation. Higher happiness threshold leads to almost complete segregation.

Figure 2 shows that overall similarity ratio was initially 50.4%. The high happiness threshold resulted in an almost completely segregated community with overall similarity ratio of 99.9%. The diagram shows this clearly.

Just like in the first model, Randomly assigning residents to 2 groups distributes the population evenly, leading to an initial overall similarity ratio around 50%. However, even with a low happiness threshold of 31%, segregation naturally occurs, with overall similarity ratio much higher than what residents are willing to tolerate. Thus, on a macro level, observing segregation may not be indicative of what people feel at the micro level. Hence, the answer to the question *How high should the agents' threshold be in order for segregation to occur?* is also: **not so high**.

The code for the second model is much shorter than the first. One of the major drawbacks, however, is the implementation time (depending on the chosen parameters). The more agents involved, and the bigger the neighborhood radius of each agent (thus, the more neighbors to be checked for similarity), the longer it takes to run the model. This is an improvement, however, as the interaction between agents are more natural compared to the first model.

Appendix A: Model 1 Full Code

```
### Needed libraries ###

# For pairing x- and y-values to create Cartesian products; an ordered pair represents the address a
    resident
import itertools

# For shuffling all houses in the community to randomly assign them to residents; and for choosing a
    random empty house to move into (for unhappy residents)
import random as rd

# For visualizing the distribution of the residents using scatterplot
import matplotlib.pyplot as plt

# For checking if a filename already exists, to avoid overwriting files
import os



### Agent-based model ###

class Schelling:
  def __init__(self, width, height, empty_ratio, happiness_threshold, groups, n_iterations):

    # Width of community grid
    self.width = width

    # Height of community grid
    self.height = height

    # Percentage of empty houses
    self.empty_ratio = empty_ratio

    # Minimum similarity ratio, to be considered happy
    self.happiness_threshold = happiness_threshold

    # Number of groups of residents
    self.groups = groups

    # Maximum iterations
    self.n_iterations = n_iterations

    # List of empty houses
    self.empty_houses = []

    # Dictionary of residents
    self.agents = {}


  ## Place residents on the community

  def populate(self):

    # Create address (coordinates) of residents
    all_houses = list(itertools.product(range(self.width), range(self.height)))

    # Shuffle the order of houses
    rd.shuffle(all_houses)

    # Determine number of empty houses
    n_empty = int(self.empty_ratio*len(all_houses))

    # Assign first few houses as empty
    self.empty_houses = all_houses[ : n_empty]

    # Assign the rest to be occupied
    remaining_houses = all_houses[n_empty : ]

    # Assign houses by group to residents
    houses_by_group = [remaining_houses[i::self.groups] for i in range(self.groups)]

    # Create dictionary of residents
    # Each resident is defined by his address and group number
    for i in range(self.groups):
      agent = dict(zip(houses_by_group[i], [i+1]*len(houses_by_group[i])))
      self.agents.update(agent)
```

```python


## Checker if resident is unhappy

def is_unhappy(self, x, y):

  # Get group number of resident
  group = self.agents[(x, y)]

  # Initialize variables
  count_similar = 0
  count_different = 0

  # Check similarity with bottom left neighbor
  if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
    if self.agents[(x-1, y-1)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Check similarity with bottom neighbor
  if y > 0 and (x, y-1) not in self.empty_houses:
    if self.agents[(x, y-1)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Check similarity with bottom right neighbor
  if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
    if self.agents[(x+1, y-1)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Check similarity with left neighbor
  if x > 0 and (x-1, y) not in self.empty_houses:
    if self.agents[(x-1,y)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Check similarity with right neighbor
  if x < (self.width-1) and (x+1, y) not in self.empty_houses:
    if self.agents[(x+1,y)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Check similarity with upper left neighbor
  if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
    if self.agents[(x-1,y+1)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Check similarity with upper neighbor
  if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
    if self.agents[(x,y+1)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Check similarity with upper right neighbor
  if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
    if self.agents[(x+1,y+1)] == group:
      count_similar += 1
    else:
      count_different += 1

  # Resident is NOT unhappy, i.e., happy if he has no neighbors
  if (count_similar + count_different) == 0:
    return False

  # Check if similarity ratio is below happiness threshold
  else:
    return float(count_similar/(count_similar + count_different)) < self.happiness_threshold

```

```

## Resident moves if he is unhappy

def move(self):

  # Maximum iterations allowed
  for i in range(self.n_iterations):

    # Initialize count
    n_changes = 0

    # Check each resident
    for agent in self.agents:

      # Activated if resident is unhappy
      if self.is_unhappy(agent[0], agent[1]):

        # Get a random empty house
        empty_house = rd.choice(self.empty_houses)

        # Get group number of the resident
        agent_group = self.agents[agent]

        # Assign the empty house to the resident
        self.agents[empty_house] = agent_group

        # Remove the original residence from the dictionary
        del self.agents[agent]

        # Remove the now-occupied house from the list of empty houses
        self.empty_houses.remove(empty_house)

        # Add the house the resident just left to the list of empty houses
        self.empty_houses.append(agent)

        # Count as a move
        n_changes += 1

    # Iteration stops if everyone is already happy
    if n_changes == 0:
      break


## Compute similarity ratio

def similarity(self):

  # Initialize list
  similarity = []

  # Do for each resident
  for agent in self.agents:

    # Initialize variables
    count_similar = 0
    count_different = 0

    # Get address and group number of the resident
    x = agent[0]
    y = agent[1]
    group = self.agents[(x,y)]

    # Check similarity with bottom left neighbor
    if x > 0 and y > 0 and (x-1, y-1) not in self.empty_houses:
      if self.agents[(x-1, y-1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with bottom neighbor
    if y > 0 and (x, y-1) not in self.empty_houses:
      if self.agents[(x, y-1)] == group:
        count_similar += 1
      else:
        count_different += 1

    # Check similarity with bottom right neighbor
```

```
      if x < (self.width-1) and y > 0 and (x+1, y-1) not in self.empty_houses:
        if self.agents[(x+1, y-1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with left neighbor
      if x > 0 and (x-1, y) not in self.empty_houses:
        if self.agents[(x-1, y)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with right neighbor
      if x < (self.width-1) and (x+1, y) not in self.empty_houses:
        if self.agents[(x+1, y)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with upper left neighbor
      if x > 0 and y < (self.height-1) and (x-1, y+1) not in self.empty_houses:
        if self.agents[(x-1, y+1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with upper neighbor
      if x > 0 and y < (self.height-1) and (x, y+1) not in self.empty_houses:
        if self.agents[(x, y+1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Check similarity with upper right neighbor
      if x < (self.width-1) and y < (self.height-1) and (x+1, y+1) not in self.empty_houses:
        if self.agents[(x+1,y+1)] == group:
          count_similar += 1
        else:
          count_different += 1

      # Place similarity ratio in the list
      try:
        similarity.append(float(count_similar/(count_similar + count_different)))

      # If there are no neighbors, similarity ratio is 1
      except:
        similarity.append(1)

    # Compute average similarity ratio over all residents
    return sum(similarity)/len(similarity)


  ## Visualize the state

  def visualize(self, state):

    # Initialize subplots
    fig, ax = plt.subplots()

    # Assign color to each group (define more as needed)
    agent_colors = {1:'b', 2:'r', 3:'g', 4:'c', 5:'m', 6:'y', 7:'k'}

    # Create a scatterplot for each resident, colored based on his group
    for agent in self.agents:
      ax.scatter(agent[0]+0.5, agent[1]+0.5, color = agent_colors[self.agents[agent]], edgecolors = 'white')

    # Title
    ax.set_title('Schelling Model: ' + state + ' State' + '\n' + 'Similarity: ' + str("{:.1f}".format(schelling.similarity()*100)) + '%')

    # Horizontal axis label
    ax.set_xlabel(str(self.empty_ratio*100) + '% Empty Houses' + '\n' + str(self.happiness_threshold*100) + '% Happiness Threshold' + '\n' + str(self.groups) + ' groups')

    # Ensure all residents are visible
```

```
    ax.set_xlim([0, self.width])
    ax.set_ylim([0, self.height])

    # Remove extra tick marks on the axes
    ax.set_xticks([])
    ax.set_yticks([])

    # Prepare format of file name
    filename = 'Model1_' + state

    # Starting filename count
    i = 1

    # Check if filename already exists; add 1 if it does
    while os.path.exists('{}{:d}.png'.format(filename, i)):
      i += 1

    # Save figure
    plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)



### Simulation ###

# Initialize parameters
schelling = Schelling(width = 25, height = 25, empty_ratio = 0.25, happiness_threshold = 0.30, groups
    = 2, n_iterations = 500)

# Place residents on the community
schelling.populate()

# Visualize initial state
schelling.visualize('Initial')

# Allow unhappy residents to move
schelling.move()

# Visualize final state
schelling.visualize('Final')

####### End of Code #######
```

Appendix B: Model 2 Full Code

```
### Needed libraries ###

# For assigning a random location to agents
import random as rd

# For visualizing the distribution of agents using plot
import matplotlib.pyplot as plt

# For checking if a filename already exists, to avoid overwriting files
import os



### Initialize class to create residents ###

class agent:
  pass



### Create residents ###

def create_agents():

  # Allow list of residents to be accessed outside the function
  global agents_list

  # Initialize list
  agents_list = []

  # Create specified number of residents
  for each_agent in range(n_agents):

    # Create an agent
    agent_ = agent()

    # Assign to a random address
    agent_.x = rd.random()
    agent_.y = rd.random()

    # Assign to a random group
    agent_.group = rd.randint(0, groups-1)

    # Place resident on the list
    agents_list.append(agent_)



### Group residents according to group number ###

def group_by_number():

  # Allow list of groups to be accessed outside the function
  global group

  # Initialize list
  group = []

  # Group according to group number
  for group_number in range(groups):

    # A resident is grouped with other residents with the same group number
    group.append([agent_ for agent_ in agents_list if agent_.group == group_number])



### Let resident move if he is not satisfied with his neighbors ###

def move():

  # Allow number of iterations done to be accessed outside the function
  global iteration

  # Maximum iterations allowed
  for iteration in range(n_iterations):
```

```

    # Initialize count
    n_changes = 0

    # Check each resident
    for agent_ in agents_list:

      # Create list of neighbors within the radius specified
      neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y - neighbor.y)**2 < radius**2]

      # Removes the resident himself from list of neighbors
      neighbors.remove(agent_)

      # Check if there are neighbors within the radius
      if len(neighbors) > 0:

        # Compute similarity ratio
        satisfaction = len([neighbor for neighbor in neighbors if neighbor.group == agent_.group])/len(neighbors)

        # Move resident to a random location if similarity ratio is below threshold
        if satisfaction < threshold:
          agent_.x, agent_.y = rd.random(), rd.random()

          # Count as a move
          n_changes += 1

    # Iteration stops if everyone is already happy
    if n_changes == 0:
      break



### Compute similarity ratio ###

def similarity_ratio():

  # Allow overall similarity ratio to be accessed outside the function
  global overall_similarity_ratio

  # Initialize list
  similarity_ratios = []

  # Check each resident
  for agent_ in agents_list:

    # Create list of neighbors within the radius specified
    neighbors = [neighbor for neighbor in agents_list if (agent_.x - neighbor.x)**2 + (agent_.y - neighbor.y)**2 < radius**2]

    # Remove resident himself from list of neighbors
    neighbors.remove(agent_)

    # Check if there are neighbors within the radius
    if len(neighbors) > 0:

      # Place similarity ratio in the list
      try:
        similarity_ratios.append(len([neighbor for neighbor in neighbors if neighbor.type == agent_.type])/len(neighbors))

      # If there are no neighbors, similarity ratio is 1
      except:
        similarity_ratios.append(1)

  # Compute average similarity ratio over all residents
  overall_similarity_ratio = sum(similarity_ratios)/len(similarity_ratios)

  # Outputs overall similarity ratio
  return overall_similarity_ratio



### Visualize the state ###

def visualize(state):
```

```
    # Initialize subplots
    fig, ax = plt.subplots()

    # Create a scatterplot by group, plotting each resident
    for Group in range(groups):
        ax.plot([agent_.x for agent_ in group[Group]], [agent_.y for agent_ in group[Group]], 'o')

    # Title
    ax.set_title(state + ' State' + ' || ' + 'Segregation: ' + str("{:.1f}".format(similarity_ratio()*100)) + '%')

    # Horizontal axis label
    ax.set_xlabel(str(n_agents) + ' Residents' + ' || ' + str(groups) + ' Groups' + ' || ' + str(radius*100) + '% Neighborhood || ' + str(threshold*100) + '% Threshold' + '\n' + 'moves: ' + str(iteration))

    # Remove extra tick marks on the axes
    ax.set_xticks([])
    ax.set_yticks([])

    # Prepare format of file name
    filename = 'Model2_' + state

    # Starting filename count
    i = 1

    # Check if filename already exists; add 1 if it does
    while os.path.exists('{}{:d}.png'.format(filename, i)):
        i += 1

    # Save figure
    plt.savefig('{}{:d}.png'.format(filename, i), bbox_inches = 'tight', dpi = 300)



### Simulation ###

# Number of residents
n_agents = 1000

# Number of groups
groups = 2

# Maximum iterations
n_iterations = 100

# Neighborhood radius
radius = 0.1

# Satisfaction threshold
threshold = 0.31

# Create residents
create_agents()

# Group residents according to group number
group_by_number()

# Needed for label of initial state
iteration = 0

# Visualize initial state
visualize('Initial')

# Allow unsatisfied residents to move
move()

# Visualize initial state
visualize('Final')

####### End of Code #######
```

References

[1] Bonabeau, Eric. "Agent-Based Modeling: Methods and Techniques for Simulating Human Systems." *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, 14 Mar. 2002, pp. 7280-7287.

[2] Castiglione, Filippo. "Agent Based Modeling." *Scholarpedia*, Brain Corporation, 29 Sep. 2006, www.scholarpedia.org/article/Agent_based_modeling.

[3] Moujahid, Adil. "An Introduction to Agent-Based Models: Simulating Segregation with Python." *binPress*, www.binpress.com/simulating-segregation-with-python/.

[4] Sayama, Hiroki. "Agent-Based Models." *LibreTexts*, 23 June 2019, https://math.libretexts.org/Bookshelves/Applied_Mathematics/Book%3A_Introduction_to_the_Modeling_and_Analysis_of_Complex_Systems_(Sayama)/19%3A_Agent-Based_Models/19.02%3A_Building_an_Agent-Based_Model.

[5] Schelling, Thomas C. "Dynamic Models of Segregation." *Journal of Mathematical Sociology*, vol. 1, 1971, pp. 143-186.