agentpy

Release 0.0.7.dev0

Joël Foramitti

Jan 26, 2021

CONTENTS

CHAPTER ONE

INTRODUCTION

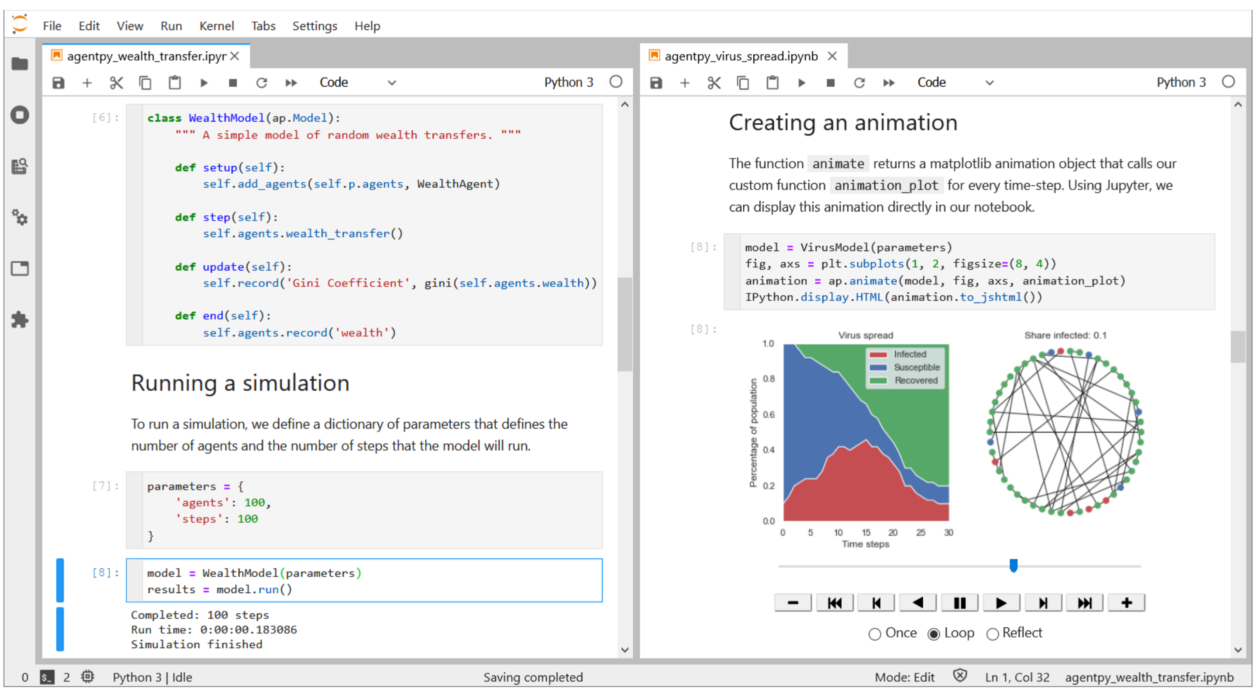
Agentpy is an open-source library for the development and analysis of agent-based models in Python. The framework integrates the tasks of model design, numerical experiments, and data analysis within a single environment, and is optimized for interactive computing with IPython and Jupyter. If you have questions or ideas for improvements, please visit the discussion forum or subscribe to the agentpy mailing list.

Quick orientation

- To get started, please take a look at *Installation* and *Overview*.
- For a simple demonstration, check out the *Wealth transfer* tutorial in the *Model Library*.
- For a detailled description of all classes and functions, refer to *API Reference*.
- To learn how agentpy compares with other frameworks, take a look at *Comparison*.

Example

A screenshot of Jupyter Lab with two interactive tutorials from the model library:



CHAPTER

TWO

INSTALLATION

To install the latest release of agentpy, run the following command on your console:

```
$ pip install agentpy
```

2.1 Dependencies

Agentpy supports Python 3.6, 3.7, 3.8, and 3.9. The installation includes the following packages:

- numpy, for scientific computing
- matplotlib, for visualization
- pandas, for output dataframes
- networkx, for network analysis
- IPython and ipywidgets, for interactive computing
- SALib, for sensitivity analysis

These optional packages can further be useful in combination with agentpy, and are required in some of the tutorials:

- jupyter, for interactive computing
- seaborn, for statistical data visualization

2.2 Development

The most recent version of agentpy can be cloned from Github:

```
$ git clone https://github.com/JoelForamitti/agentpy.git
```

Once you have a copy of the source, you can install it with:

```
$ pip install -e
```

To include all necessary packages for development, you can use:

```
$ pip install -e .['dev']
```

CHAPTER

THREE

OVERVIEW

This section aims to provide a rough overview over the main classes and functions of agentpy and how they are meant to be used. For a more detailed description of each element, please refer to the *API Reference*. Throughout this documentation, agentpy is imported as follows:

```
import agentpy as ap
```

3.1 Creating models

The basic framework for agent-based models consists of three levels:

1. `Model`, which contains agents, environments, parameters, & procedures
2. `Environment`, `Grid`, and `Network`, which contain agents
3. `Agent`, the basic building blocks of the model

All of these classes are designed to be customized through the creation of sub-classes with their own variables and methods. A custom agent type could be defined as follows:

```
class MyAgentType(ap.Agent):

    def setup(self):
        # Initialize an attribute with a parameter
        self.my_attribute = self.p.my_parameter

    def agent_method(self):
        # Define custom actions here
        pass
```

The method `Agent.setup()` is meant to be overwritten and will be called after an agents' creation. All variables of an agents should be initialized in this method. Other methods can represent actions that the agent will be able to take during a simulation.

We can further see that the agent comes with a built-in attribute `p` that allows it to access the models' parameters. All model objects (i.e. agents, environments, and the model itself) are equipped with such properties to access different parts of the model:

- `model` returns the model instance
- `model.t` returns the model's time-step
- `id` returns a unique identifier number for each object
- `p` returns an `AttrDict` of the models' parameters

- `envs` returns an *EnvList* of the objects' environments
- `agents` (not for agents) returns an *AgentList* of the objects' agents
- `log` returns a `dict` of the objects' recorded variables

Using the new agent type defined above, here is how a basic model could look like:

```
class MyModel(ap.Model):

    def setup(self):
        """ Called at the start of the simulation """
        self.add_agents(self.p.agents, MyAgentType)  # Add new agents

    def step(self):
        """ Called at every simulation step """
        self.agents.agent_method()  # Call a method for every agent

    def update(self):
        """ Called after setup as well as after each step """
        self.agents.record('my_attribute')  # Record a dynamic variable

    def end(self):
        """ Called at the end of the simulation """
        self.measure('my_measure', 1)  # Record an evaluation measure
```

This custom model is defined by four special methods that will be used automatically during different parts of a simulation. If you want to see a basic model like this in action, take a look at the *Wealth transfer* demonstration in the *Model Library*.

3.2 Using agents

Agentpy comes with various tools to create, manipulate, and delete agents. The method *Model.add_agents()* can be used to initialize new agents. A list of all agents in a model can be accessed through `Model.agents`. Lists of agents are returned as an *AgentList*, which provides special features to access and manipulate the whole group of agents.

For example, when the model defined above calls `self.agents.agent_method()`, it will call the method `MyAgentType.agent_method()` for every agent in the model. Similar commands can be used to set and access variables, or select subsets of agents with boolean operators. The following command, for example, would select all agents with an id above one:

```
self.agents.select(self.agents.id > 1)
```

Further examples can be found in the *AgentList* reference or the *Virus spread* model.

3.3 Using environments

Environments can contain agents just like the main model, and are useful if one wants to regard particular topologies for interaction or multiple environments that can hold seperate populations of agents. Agents can be moved between environments with the methods `Agent.enter()` and `Agent.exit()`.

New environments can be created with `Model.add_env()`. Similar to agents, the attribute `envs` returns an `EnvList` with special features to deal with groups of environments. There are three different types of environments:

- `Environment`, which simply contain agents without any topology.
- `Network`, in which agents can be connected via a networkx graph.
- `Grid`, in which agents occupy a position on a x-dimensional space.

Applied examples of networks can be found in the demonstration models *Virus spread* and *Button network*, while a spatial grid is used in *Forest fire*.

3.4 Recording data

As can be seen in the model defined above, there are two main types of data in agentpy. The first are dynamic variables, which can be stored for each object (agent, environment, or model) and time-step. They are useful to look at the dynamics of individual or aggregate objects over time and can be recorded by calling the method `record()` for the respective object.

The other type of recordable data are evaluation measures. These, in contrast, can be stored only for the model as a whole and only once per run. They are useful as summary statistics that can be compared over multiple runs, and can be recorded with the method `Model.measure()`.

3.5 Running a simulation

To perform a simulation, we have to initialize a new instance of our model type with a dictionary of parameters, after which we use the function `Model.run()`. This will return a `DataDict` with recorded data from the simulation. A simple run could be prepared and executed as follows:

```
parameters = {'my_parameter':42,
              'agents':10,
              'steps':10, }

model = MyModel(parameters)
results = model.run()
```

The procedure of a simulation is as follows:

0. The model initializes with the time-step `Model.t = 0`.
1. `Model.setup()` and `Model.update()` are called.
2. The model's time-step is increased by 1.
3. `Model.step()` and `Model.update()` are called.
4. Step 2 and 3 are repeated until the simulation is stopped.
5. `Model.end()` is called.

The simulation of a model can be stopped by one of the following three ways:

1. Calling the *Model.stop()* during the simulation.
2. Reaching the time-limit, which be defined as follows:
   - Defining `steps` in the paramater dictionary.
   - Passing `steps` as an argument to *Model.run()*.

3.6 Multi-run experiments

The class *Experiment* can be used to run a model multiple times with repeated iterations, varied parameters, and distinct scenarios. To prepare a sample of parameters for an experiment, one can use one of the sampling functions *sample()*, *sample_saltelli()*, or *sample_discrete()*. Here is an example of an experiment with the model defined above:

```
parameter_ranges = {'my_parameter': 42,
                    'agents': (10, 20, int),
                    'steps': (10, 20, int)}

sample = ap.sample(parameter_ranges, n=5)

exp = ap.Experiment(MyModel, sample, iterations=2,
                    scenarios=('sc1','sc2'))

results = exp.run()
```

In this experiment, we use a sample where one parameter is kept fixed while the other two are varied 5 times from 10 to 20 and set to integer. Every possible combination is repeated 2 times, which results in 50 runs. Each run further has one result for each of the two scenarios *sc1* and *sc2*. For more applied examples of experiments, check out the demonstration models *Virus spread*, *Button network*, and *Forest fire*.

3.7 Output and analysis

Both *Model* and *Experiment* can be used to run a simulation, which will return a *DataDict* with output data. The output from the experiment defined above looks as follows:

```
>>> results
DataDict {
'log': Dictionary with 5 keys
'parameters':
    'fixed': Dictionary with 1 key
    'varied': DataFrame with 2 variables and 25 rows
'measures': DataFrame with 1 variable and 50 rows
'variables':
    'MyAgentType': DataFrame with 1 variable and 10500 rows
}
```

The output can contain the following categories of data:

- `log` holds meta-data about the model and simulation performance.
- `parameters` holds the parameter values that have been used for the experiment.
- `variables` holds dynamic variables, which can be recorded at multiple time-steps.
- `measures` holds evaluation measures that are recoreded only once per simulation.

This data can be stored with `DataDict.save()` and `load()`. `DataDict.arrange()` can further be used to generate a specific dataframe for analysis or visualization. All data is given in a `pandas.DataFrame` and formatted as long-form data, which makes it compatible to use with statistical packages like seaborn. Agentpy further provides the following functions for analysis:

- `sensitivity_sobol()` performs a Sobol sensitivity analysis.
- `Experiment.interactive()` generates an interactive widget for parameter variation.
- `animate()` generates an animation that can display output over time.
- `gridplot()` visualizes agent positions on a spatial `Grid`.

To see applied examples of these functions, please check out the *Model Library*.

CHAPTER FOUR

USER GUIDE

Welcome to the agentpy user guide. This section contains various articles to help with specific problems and applications. Some of these articles are provided as interactive Jupyter Notebooks that can be downloaded and experimented with.

If you are interested to add a new article to this guide, please visit *Contribute*. If you are looking for examples of complete models, take a look at *Model Library*.

Note: You can download this demonstration as a Jupyter Notebook `here`

4.1 Stochastic processes and reproducibility

Random numbers and stochastic processes are essential to many agent-based models. In Python, we can use the pseudo-random number generator from the built-in library `random`.

Pseudo-random means that this module generates numbers in a sequence that appears random but is actually deterministic, based on an initial seed value. In other words, the generator will produce the same pseudo-random sequence over multiple runs if it is given the same seed at the beginning. We can define this seed to receive reproducible results from a model with stochastic processes.

4.1.1 Generating random numbers

[1]:
```
import agentpy as ap
import random
```

To illustrate, let us define a model that generates a list of ten pseudo-random numbers:

[2]:
```
class RandomModel(ap.Model):

    def setup(self):
        self.random_numbers = [random.randint(0, 9) for _ in range(10)]
        print(f"Model {self.p.n} generated the numbers {self.random_numbers}")
```

Now if we run this model multiple times, we will get a different series of numbers:

[3]:
```
for i in range(2):
    parameters = {'steps':0, 'n':i}
    model = RandomModel(parameters)
    results = model.run(display=False)
```

```
Model 0 generated the numbers [9, 3, 3, 8, 8, 0, 1, 9, 4, 7]
Model 1 generated the numbers [0, 5, 9, 4, 6, 5, 3, 2, 2, 0]
```

If we want the results to be reproducible, we can define a parameter `seed` that will be used automatically at the beginning of `Model.run()`. Now, we get the same series of numbers:

[4]:
```
for i in range(2):
    parameters = {'seed':1, 'steps':0, 'n':i}
    model = RandomModel(parameters)
    model.run(display=False)
```

```
Model 0 generated the numbers [2, 9, 1, 4, 1, 7, 7, 7, 6, 3]
Model 1 generated the numbers [2, 9, 1, 4, 1, 7, 7, 7, 6, 3]
```

4.1.2 Using multiple generators

The automatic use of the `seed` parameter calls the method `random.seed()`, which affects the default number generator that is created as a hidden instance by the `random` module. For more advanced applications, we can create seperate generators for each object, using `random.Random`. We can ensure that the seeds of each object follow a controlled pseudo-random sequence by using also using seperate generator in the main model. Note that we use a different parameter name *model_seed* to avoid the automatic use of the parameter `seed` in this case.

[5]:
```
class RandomAgent2(ap.Agent):

    def setup(self):
        seed = model.seed_generator.getrandbits(128)  # Get seed from model
        self.random = random.Random(seed)  # Create generator for this agent
        self.random_numbers = [self.random.randint(0, 9) for _ in range(10)]
        print(f"{self} generated the numbers {self.random_numbers}")

class RandomModel2(ap.Model):

    def setup(self):
        self.seed_generator = random.Random(self.p.model_seed)
        self.add_agents(2, RandomAgent2)

for i in range(2):
    print(f"Model {i}:")
    parameters = {'model_seed': 1, 'steps': 0}
    model = RandomModel2(parameters)
    results = model.run(display=False)
    print()
```

```
Model 0:
RandomAgent2 (Obj 1) generated the numbers [8, 7, 0, 1, 2, 3, 9, 4, 5, 0]
RandomAgent2 (Obj 2) generated the numbers [8, 1, 4, 6, 6, 3, 4, 3, 5, 1]

Model 1:
RandomAgent2 (Obj 1) generated the numbers [8, 7, 0, 1, 2, 3, 9, 4, 5, 0]
RandomAgent2 (Obj 2) generated the numbers [8, 1, 4, 6, 6, 3, 4, 3, 5, 1]
```

Alternatively, we could also have each agent start from the same seed:

[6]:
```
class RandomAgent3(ap.Agent):
```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        self.random = random.Random(self.p.agent_seed)
        self.random_numbers = [self.random.randint(0, 9) for _ in range(10)]
        print(f"{self} generated the numbers {self.random_numbers}")

class RandomModel3(ap.Model):

    def setup(self):
        self.add_agents(2, RandomAgent3)

for i in range(2):
    print(f"\nModel {i}:")
    parameters = {'agent_seed': 1, 'steps':0, 'n':i}
    model = RandomModel3(parameters)
    results = model.run(display=False)
```

```
Model 0:
RandomAgent3 (Obj 1) generated the numbers [2, 9, 1, 4, 1, 7, 7, 7, 6, 3]
RandomAgent3 (Obj 2) generated the numbers [2, 9, 1, 4, 1, 7, 7, 7, 6, 3]

Model 1:
RandomAgent3 (Obj 1) generated the numbers [2, 9, 1, 4, 1, 7, 7, 7, 6, 3]
RandomAgent3 (Obj 2) generated the numbers [2, 9, 1, 4, 1, 7, 7, 7, 6, 3]
```

4.1.3 Modeling stochastic processes

This section presents some stochastic operations that are often used in agent-based models. To start, we prepare a generic model with ten agents:

[7]:
```
model = ap.Model()
agents = model.add_agents(10)
agents
```

[7]:
```
AgentList [10 agents]
```

If we look at the agent's ids, we see that they have been created in order:

[8]:
```
agents.id
```

[8]:
```
AttrList of attribute 'id': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We can shuffle this list with *AgentList.shuffle()*:

[9]:
```
agents.shuffle().id
```

[9]:
```
AttrList of attribute 'id': [5, 10, 3, 9, 6, 4, 7, 1, 8, 2]
```

Or create a random subset with *AgentList.random()*:

[10]:
```
agents.random(5).id
```

[10]:
```
AttrList of attribute 'id': [6, 9, 10, 3, 5]
```

Both *AgentList.shuffle()* and *AgentList.random()* can take a custom generator as an argument:

```
[11]: for _ in range(2):
          custom_generator = random.Random(1)
          print(agents.random(5, custom_generator).id)
```

```
AttrList of attribute 'id': [3, 10, 6, 5, 9]
AttrList of attribute 'id': [3, 10, 6, 5, 9]
```

Note that the above selection is without repetition, i.e. every agent can only be selected once. Outside these built-in functions of agentpy, there are many other tools that can be used for stochastic processes. For example, we can use the methods `random.choices()` to make a selection with repetition and probability weights. In the following example, agents with a higher id are more likely to be chosen:

```
[12]: choices = random.choices(agents, k=5, weights=agents.id)
```

If needed, the resulting list from such selections can be converted back into an *AgentList*:

```
[13]: ap.AgentList(choices).id
```

```
[13]: AttrList of attribute 'id': [5, 4, 5, 8, 7]
```

4.1.4 Further reading

- Random number generation in Python: https://realpython.com/python-random/
- Stochasticity in agent-based models: http://www2.econ.iastate.edu/tesfatsi/ace.htm#Stochasticity
- Pseudo-random number generators: https://en.wikipedia.org/wiki/Pseudorandom_number_generator
- What is random: https://www.youtube.com/watch?v=9rIy0xY99a0

CHAPTER

FIVE

MODEL LIBRARY

Welcome to the agentpy model library. Below you can find a set of demonstrations on how the package can be used. All of the models are provided as interactive Jupyter Notebooks that can be downloaded and experimented with.

Note: You can download this demonstration as a Jupyter Notebook `here`

5.1 Wealth transfer

This is a tutorial for beginners on how to create a simple agent-based model with the agentpy package. It shows the how to create a basic model with a custom agent type, run a simulation, record data, and visualize results.

5.1.1 About the model

The model explores the distribution of wealth under a trading population of agents. We will see that their random interaction will create an inequality of wealth that follows a Boltzmann distribution. The original version of this model been written in MESA and can be found here.

5.1.2 Getting started

To install the latest version of agentpy, run the following command:

```
[1]: # !pip install agentpy
```

Once installed, the recommended way to import the package is as follows:

```
[2]: import agentpy as ap
```

We also import two other libraries that will be used in this demonstration.

```
[3]: import numpy as np  # Scientific computing tools
     import matplotlib.pyplot as plt  # Visualization
```

5.1.3 Model definition

We start by defining a new type of agent as a subcluss of *Agent*. Each agent starts with one unit of `wealth`. When `wealth_transfer()` is called, the agent selects another agent at random and gives them one unit of their own wealth if they have one to spare.

[4]:
```
class WealthAgent(ap.Agent):

    """ An agent with wealth """

    def setup(self):

        self.wealth = 1

    def wealth_transfer(self):

        if self.wealth > 0:

            partner = self.model.agents.random()
            partner.wealth += 1
            self.wealth -= 1
```

Next, we define a method to calculate the Gini Coefficient, which will measure the inequality among our agents.

[5]:
```
def gini(x):

    """ Calculate Gini Coefficient """
    # By Warren Weckesser https://stackoverflow.com/a/39513799

    mad = np.abs(np.subtract.outer(x, x)).mean()  # Mean absolute difference
    rmad = mad / np.mean(x)  # Relative mean absolute difference
    return 0.5 * rmad
```

Finally, we define our model as a subclass of *Model*. In *Model.setup()*, we define how many agents should be created at the beginning of the simulation. In *Model.step()*, we define that at every time-step all agents will perform the action *wealth_transfer*. In *Model.update()*, we calculate and record the current Gini coefficient. And in *Model.end()*, we further record the wealth of each agent.

[6]:
```
class WealthModel(ap.Model):

    """ A simple model of random wealth transfers """

    def setup(self):

        self.add_agents(self.p.agents, WealthAgent)

    def step(self):

        self.agents.wealth_transfer()

    def update(self):

        self.record('Gini Coefficient', gini(self.agents.wealth))

    def end(self):

        self.agents.record('wealth')
```

5.1.4 Running a simulation

To run a simulation, we define a dictionary of parameters that defines the number of agents and the number of steps that the model will run.

[7]:
```
parameters = {
    'agents': 100,
    'steps': 100
}
```

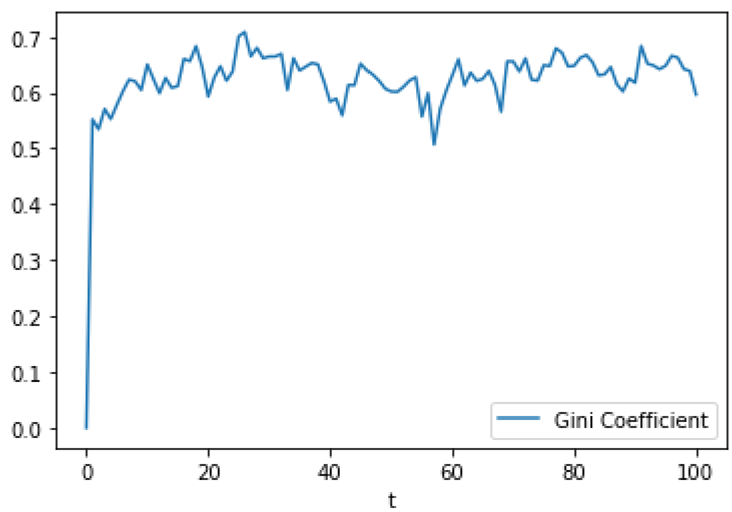
To perform a simulation, we initialize our model with these parameters and call the method *Model.run*, which returns a *DataDict* of our recorded variables and measures.

[8]:
```
model = WealthModel(parameters)
results = model.run()
```

```
Completed: 100 steps
Run time: 0:00:00.183086
Simulation finished
```

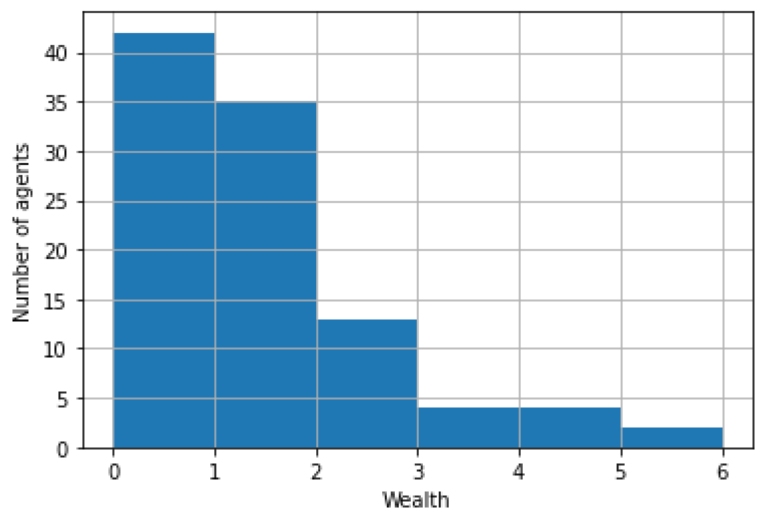
To visualize the evolution of our Gini Coefficient, we can use `pandas.DataFrame.plot()`.

[9]:
```
data = results.variables.WealthModel
ax = data.plot()
```



And to visualize the final distribution of wealth, we can use `pandas.DataFrame.hist()`.

[10]:
```
data = results.variables.WealthAgent
data.hist(bins=range(data.wealth.max()+1))

plt.title('')
plt.xlabel('Wealth')
plt.ylabel('Number of agents')
plt.show()
```



What we get is a Boltzmann distribution. For those interested to understand this result, you can read more about it here.

Note: You can download this demonstration as a Jupyter Notebook `here`

5.2 Virus spread

This notebook presents an agent-based model that simulates the propagation of a disease through a network. It demonstrates how to use the agentpy package to create and visualize networks, use the interactive module, and perform different types of sensitivity analysis.

[1]:
```
# Model design
import agentpy as ap
import networkx as nx
import random

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
import IPython
```

5.2.1 About the model

The agents of this model are people, which can be in one of the following three conditions: susceptible to the disease (S), infected (I), or recovered (R). The agents are connected to each other through a small-world network of peers. At every time-step, infected agents can infect their peers or recover from the disease based on random chance.

5.2.2 Defining the model

We define a new agent type `Person` by creating a subclass of *Agent*. This agent has two methods: `setup()` will be called automatically at the agent's creation, and `being_sick()` will be called by the *Model.step()* function. Three tools are used within this class:

- `Agent.p` returns the parameters of the model
- *Agent.neighbors()* returns a list of the agents' peers in the network
- `random.random()` returns a uniform random draw between 0 and 1

[2]:
```
class Person(ap.Agent):

    def setup(self):
        """ Initialize a new variable at agent creation. """
        self.condition = 0  # Susceptible = 0, Infected = 1, Recovered = 2

    def being_sick(self):
        """ Spread disease to peers in the network. """
        for n in self.neighbors():
            if n.condition == 0 and self.p.infection_chance > random.random():
                n.condition = 1  # Infect susceptible peer
        if self.p.recovery_chance > random.random():
            self.condition = 2  # Recover from infection
```

Next, we define our model `VirusModel` by creating a subclass of *Model*. The four methods will be called automatically, as described in *Running a simulation*.

[3]:
```
class VirusModel(ap.Model):

    def setup(self):
        """ Initializes the agents and network of the model. """

        self.p.population = p = int(self.p.population)
        # Prepare a small-world network graph
        graph = nx.watts_strogatz_graph(p,
                                        self.p.number_of_neighbors,
                                        self.p.network_randomness)

        # Create agents and network
        self.add_agents(p, Person)
        self.add_network(graph=graph, agents=self.agents)

        # Infect a random share of the population
        I0 = int(self.p.initial_infections * self.p.population)
        self.agents.random(I0).condition = 1

    def update(self):
        """ Records variables after setup and each step. """
        # Record share of agents with each condition
        for i, c in enumerate(('S', 'I', 'R')):
            self[c] = (len(self.agents.select(self.agents.condition == i))
                       / self.p.population)
            self.record(c)

        # Stop simulation if disease is gone
        if self.I == 0:
            self.stop()
```

(continues on next page)

(continued from previous page)

```
    def step(self):
        """ Defines the models' events per simulation step. """
        # Call 'being_sick' for infected agents
        self.agents(self.agents.condition==1).being_sick()

    def end(self):
        """ Records evaluation measures at the end of the simulation. """
        # Record final evaluation measures
        self.measure('Total share infected', self.I + self.R)
        self.measure('Peak share infected', max(self.log['I']))
```

5.2.3 Running a simulation

To run our model, we define a dictionary with our parameters. We then create a new instance of our model, passing the parameters as an argument, and use the method *Model.run()* to perform the simulation and return it's output.

[4]:
```
parameters = {
    'population': 1000,
    'infection_chance': 0.3,
    'recovery_chance': 0.1,
    'initial_infections': 0.1,
    'number_of_neighbors': 2,
    'network_randomness': 0.5
}

model = VirusModel(parameters)
results = model.run()
```

```
Completed: 75 steps
Run time: 0:00:00.420014
Simulation finished
```

5.2.4 Analyzing results

The simulation returns a *DataDict* of recorded data with dataframes:

[5]:
```
results
```

[5]:
```
DataDict {
'log': Dictionary with 4 keys
'parameters': Dictionary with 6 keys
'measures': DataFrame with 2 variables and 1 row
'variables': DataFrame with 3 variables and 76 rows
}
```

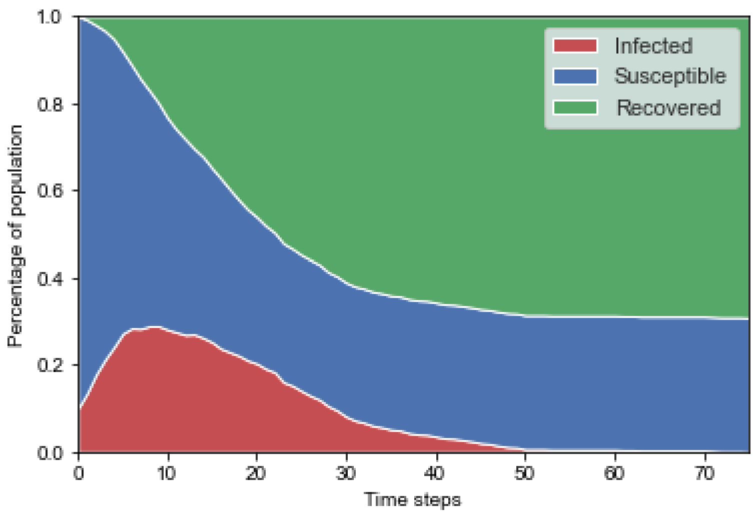
To visualize the evolution of our variables over time, we create a plot function.

[6]:
```
def virus_stackplot(data, ax):
    """ Stackplot of people's condition over time. """
    x = data.index.get_level_values('t')
    y = [data[var] for var in ['I', 'S', 'R']]

    sns.set()
```

(continues on next page)

(continued from previous page)

```
    ax.stackplot(x, y, labels=['Infected', 'Susceptible', 'Recovered'],
                 colors = ['r', 'b', 'g'])

    ax.legend()
    ax.grid(False)
    ax.set_xlim(0, max(1, len(x)-1))
    ax.set_ylim(0, 1)
    ax.set_xlabel("Time steps")
    ax.set_ylabel("Percentage of population")

fig, ax = plt.subplots()
virus_stackplot(results.variables, ax)
```



5.2.5 Creating an animation

We can also animate the model's dynamics as follows. The function `animation_plot()` takes a model instance and displays the previous stackplot together with a network graph. The function *animate()* will call this plot function for every time-step and return an `matplotlib.animation.Animation`.

[7]:
```
def animation_plot(m, axs):
    ax1, ax2 = axs
    ax1.set_title("Virus spread")
    ax2.set_title(f"Share infected: {m.I}")

    # Plot stackplot on first axis
    virus_stackplot(m.output.variables, ax1)

    # Plot network on second axis
    color_dict = {0:'b', 1:'r', 2:'g'}
    colors = [color_dict[c] for c in m.agents.condition]
    nx.draw_circular(m.env.graph, node_color=colors,
                     node_size=50, ax=ax2)

fig, axs = plt.subplots(1, 2, figsize=(8, 4)) # Prepare figure
parameters['population'] = 50 # Lower population for better visibility
animation = ap.animate(VirusModel(parameters), fig, axs, animation_plot)
```

Using Jupyter, we can display this animation directly in our notebook.

[8]:
```
IPython.display.HTML(animation.to_jshtml())
```

[8]:
```
<IPython.core.display.HTML object>
```

5.2.6 Interactive parameter variation

To explore the effect of different parameter values, we use `sample_saltelli()` to create a sample of different parameter combinations. All parameters that are given as tuples will automatically be varied. Parameter ranges that are given as integers will result in parameters that rounded to integers.

[9]:
```
param_ranges = {
    'population':(100, 1000),
    'infection_chance':(0.1, 1.),
    'recovery_chance':(0.1, 1.),
    'initial_infections':0.1,
    'number_of_neighbors':2,
    'network_randomness':(0., 1.)
}

sample = ap.sample_saltelli(param_ranges, n=100, digits=2)
```

We then create an `Experiment` that takes a model and sample as input. To explore the different parameter values in our sample, we can display a our virus stackplot interactively. The method `Experiment.interactive()` will create widgets to to change the values of our varied parameters and call this plot after each change in parameters.

[10]:
```
def interactive_plot(m):
    fig,ax = plt.subplots()
    virus_stackplot(m.output.variables, ax)

exp = ap.Experiment(VirusModel, sample)
exp.interactive(interactive_plot)
```

```
HBox(children=(VBox(children=(SelectionSlider(continuous_update=False, description=
↪'population', layout=Layout...
```

5.2.7 Multi-run experiment

We can use `Experiment.run()` to run our model repeatedly over the whole sample.

[11]:
```
exp = ap.Experiment(VirusModel, sample)
results = exp.run()
```

```
Scheduled runs: 1000
Completed: 1000, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:01:02.737876
```

[12]:
```
# To save and load data

# results.save()
# results = ap.load('VirusModel')
```

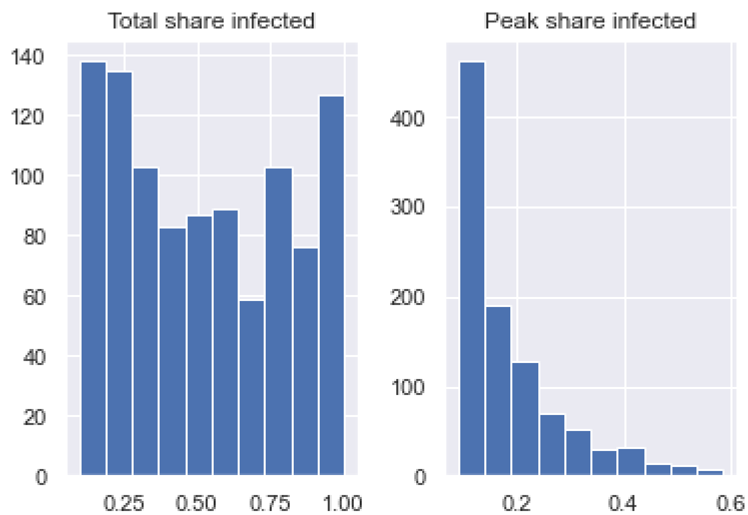
The measures in our `DataDict` now hold one row for each simulation run.

```
[13]: print(results)
```

```
DataDict {
'parameters':
    'fixed': Dictionary with 2 keys
    'varied': DataFrame with 4 variables and 1000 rows
'log': Dictionary with 5 keys
'measures': DataFrame with 2 variables and 1000 rows
}
```

We can use standard functions of the pandas library like `pandas.DataFrame.hist()` to look at summary statistics.

```
[14]: results.measures.hist()
      plt.show()
```



5.2.8 Sensitivity analysis

The function *`sensitivity_sobol()`* calculates Sobol sensitivity indices for the passed results and parameter ranges, using the SAlib package.

```
[15]: ap.sensitivity_sobol(results, param_ranges)
```

```
[15]: DataDict {
'parameters':
    'fixed': Dictionary with 2 keys
    'varied': DataFrame with 4 variables and 1000 rows
'log': Dictionary with 5 keys
'measures': DataFrame with 2 variables and 1000 rows
'sensitivity': DataFrame with 2 variables and 8 rows
'sensitivity_conf': DataFrame with 2 variables and 8 rows
}
```

This adds two new categories to our results:

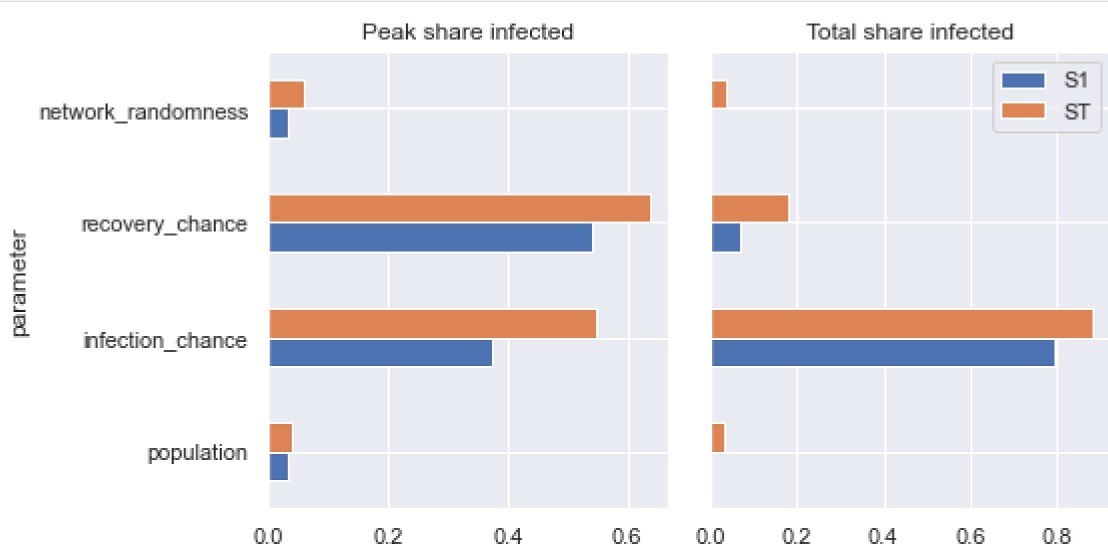
- `sensitivity` returns first-order sobol sensitivity indices
- `sensitivity_conf` returns confidence ranges for the above indices

[16]:
```
results.sensitivity
```

[16]:
```
                                                  S1        ST
measure              parameter
Total share infected population            0.001626  0.030099
                     infection_chance     0.797266  0.880848
                     recovery_chance      0.069337  0.178046
                     network_randomness -0.018461  0.036570
Peak share infected  population            0.032734  0.038616
                     infection_chance     0.373695  0.548178
                     recovery_chance      0.540922  0.637894
                     network_randomness  0.032772  0.059964
```

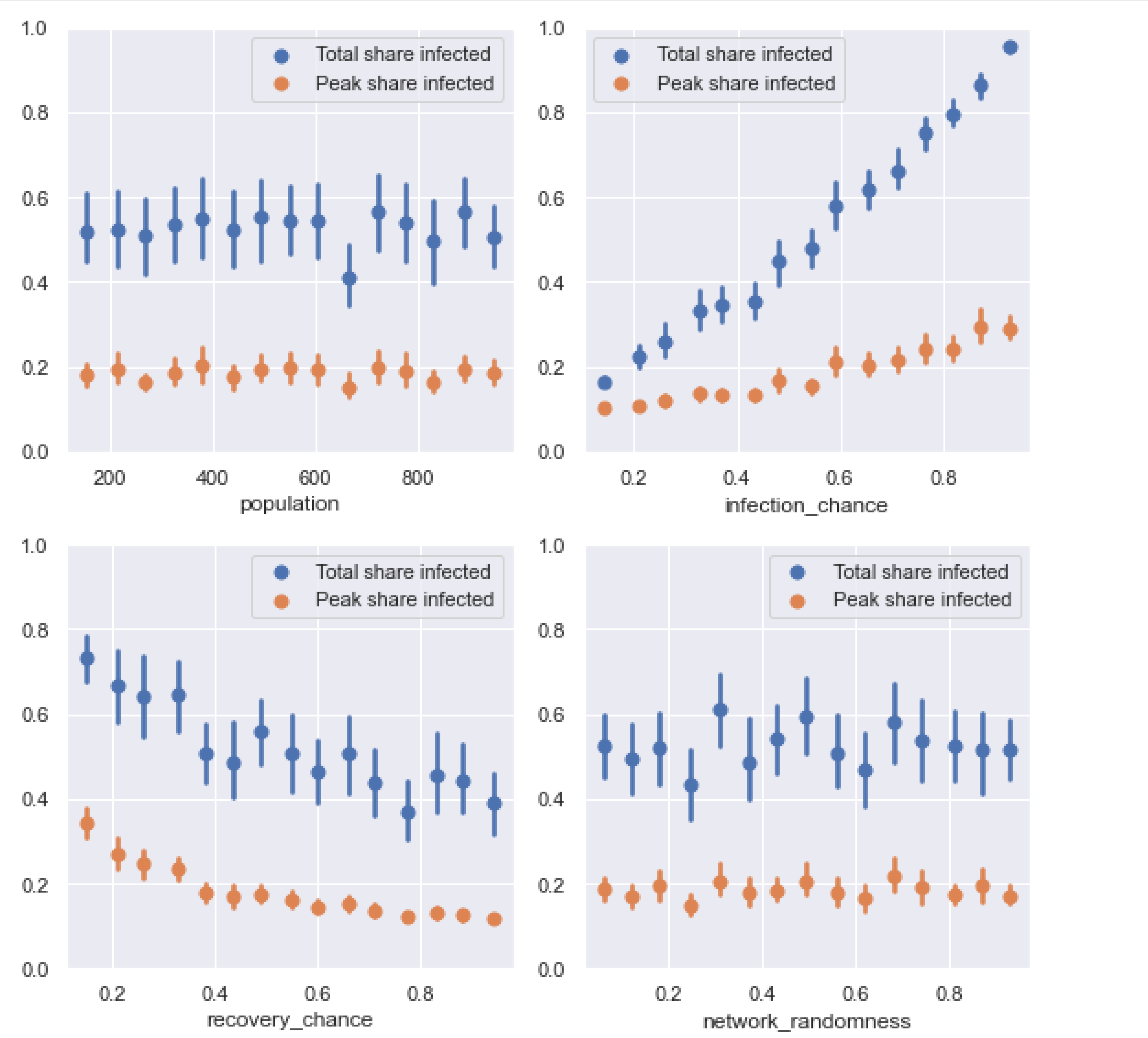
We can use pandas to create a bar plot that visualizes these sensitivity indices.

[17]:
```
def plot_sobol(results):
    """ Bar plot of Sobol sensitivity indices. """

    sns.set()
    fig, axs = plt.subplots(1, 2, figsize=(8, 4))
    SI = results.sensitivity.groupby(by='measure')
    SIT = results.sensitivity_conf.groupby(by='measure')

    for (key, si), (_, err), ax in zip(SI, SIT, axs):
        si = si.droplevel('measure')
        err = err.droplevel('measure')
        si.plot.barh(yerr=err,title=key,ax=ax)
        ax.set_xlim(0)

    axs[0].get_legend().remove()
    axs[1].set(ylabel=None, yticklabels=[])
    axs[1].tick_params(left=False)
    plt.tight_layout()

plot_sobol(results)
```



Alternatively, we can also display sensitivities by plotting average evaluation measures over our parameter variations.

```
[18]: def plot_sensitivity(results):
          """ Show average simulation results for different parameter values. """

          sns.set()
          fig, axs = plt.subplots(2, 2, figsize=(8, 8))
          axs = [i for j in axs for i in j] # Flatten list

          data = results.arrange_measures()
          params = results.parameters.varied.keys()

          for x, ax in zip(params, axs):
              for y in results.measures.columns:
                  sns.regplot(x=x, y=y, data=data, ax=ax, ci=99,
                              x_bins=15, fit_reg=False, label=y)
              ax.set_ylim(0,1)
              ax.set_ylabel('')
              ax.legend()

          plt.tight_layout()

      plot_sensitivity(results)
```



Note: You can download this demonstration as a Jupyter Notebook `here`

5.3 Segregation

This notebook presents an agent-based model of segregation dynamics. It demonstrates how to use the agentpy package to work with a spatial grid and create animations.

[1]:
```
# Model design
import agentpy as ap
import random

# Visualization
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import seaborn as sns
import IPython
```

5.3.1 About the model

The model is based on the NetLogo Segregation model from Uri Wilensky, who describes it as follows:

> This project models the behavior of two types of agents in a neighborhood. The orange agents and blue agents get along with one another. But each agent wants to make sure that it lives near some of "its own." That is, each orange agent wants to live near at least some orange agents, and each blue agent wants to live near at least some blue agents. The simulation shows how these individual preferences ripple through the neighborhood, leading to large-scale patterns.

5.3.2 Model definition

To start, we define our agents, who initiate with a random group and have two methods to check whether they are happy and to move to a new location if they are not.

[2]:
```
class Person(ap.Agent):

    def setup(self):
        self.happy = False
        self.group = random.choice(range(self.p.n_groups))

    def update_happiness(self):
        """ Be happy if rate of similar neighbors is high enough. """
        neighbors = self.neighbors()
        similar = len([n for n in neighbors if n.group == self.group])
        similar_min = self.p.want_similar * len(neighbors)
        self.happy = True if similar >= similar_min else False

    def find_new_home(self):
        """ Move to random free spot and update free spots. """
        old_spot = self.position()
        new_spot = random.choice(self.model.free_spots)
        self.move_to(new_spot)
        self.model.free_spots.remove(new_spot)
        self.model.free_spots.append(old_spot)
```

Next, we define our model, which consists of our agens and a spatial grid environment. At every step, unhappy people move to a new location. After every step (update), agents update their happiness. If all agents are happy, the simulation is stopped.

[3]:
```
class SegregationModel(ap.Model):

    def setup(self):
        # Create grid with agents
        self.add_grid(self.p.size)
        self.n_agents = int(self.p.density * (self.p.size ** 2))
        self.env.add_agents(self.n_agents, Person, random=True)

        # Create list of free spots
        self.free_spots = []
```

(continues on next page)

(continued from previous page)

```
        for pos, agents in self.env.items():
            if len(agents) == 0:
                self.free_spots.append(pos)

    def step(self):
        # Move unhappy people
        self.unhappy_people.find_new_home()

    def update(self):
        # Update list of unhappy people
        self.agents.update_happiness()
        self.unhappy_people = self.agents.select(self.agents.happy == False)

        # Stop simulation if all are happy
        if len(self.unhappy_people) == 0:
            self.stop()

    def get_segregation(self):
        # Calculate average percentage of similar neighbors
        similarity = 0
        for a in self.agents:
            neighbors = a.neighbors()
            n_neighbors = len(neighbors)
            if n_neighbors > 0:
                similarity += len([n for n in neighbors
                                   if a.group == n.group]) / n_neighbors
        return round(similarity / self.n_agents, 2)

    def end(self):
        # Measure segregation at the end of the simulation
        self.measure('segregation', self.get_segregation())
```

5.3.3 Single-run animation

Uri Wilensky explains the dynamic of the segregation model as follows:

> Agents are randomly distributed throughout the neighborhood. But many agents are "unhappy" since they don't have enough same-color neighbors. The unhappy agents move to new locations in the vicinity. But in the new locations, they might tip the balance of the local population, prompting other agents to leave. If a few agents move into an area, the local blue agents might leave. But when the blue agents move to a new area, they might prompt orange agents to leave that area.
>
> Over time, the number of unhappy agents decreases. But the neighborhood becomes more segregated, with clusters of orange agents and clusters of blue agents.
>
> In the case where each agent wants at least 30% same-color neighbors, the agents end up with (on average) 70% same-color neighbors. So relatively small individual preferences can lead to significant overall segregation.

To observe this effect in our model, we can create an animation of a single run.
To do so, we first set up an instance of our model with a chosen set of parameters.

[4]:
```
parameters = {
    'want_similar': 0.3, # For agents to be happy
    'n_groups': 2, # Number of groups
    'density': 0.95, # Density of population
    'size': 50, # Height and length of the grid
    'steps': 50  # Maximum number of steps
    }

model = SegregationModel(parameters)
```

We can now create an animation plot and display it directly in Jupyter as follows.

[5]:
```
def animation_plot(model, ax):
    group_grid = model.env.attribute('group')
    ap.gridplot(group_grid, cmap='Accent', ax=ax)
    ax.set_title(f"Segregation model \n Time-step: {model.t}, "
                 f"Segregation: {model.get_segregation()}")

fig, ax = plt.subplots()
animation = ap.animate(model, fig, ax, animation_plot)
IPython.display.HTML(animation.to_jshtml())
```

[5]:
```
<IPython.core.display.HTML object>
```

5.3.4 Multi-run experiment

To explore how different individual preferences lead to different average levels of segregation, we can conduct a multi-run experiment. To do so, we first prepare a parameter sample that includes different values for peoples' preferences and the population density.

[6]:
```
parameter_ranges = dict(parameters)
parameter_ranges.update({
    'want_similar': (0,0.125, 0.25, 0.375, 0.5, 0.625),
    'density': (0.5, 0.7, 0.95),
})
sample = ap.sample_discrete(parameter_ranges)
```

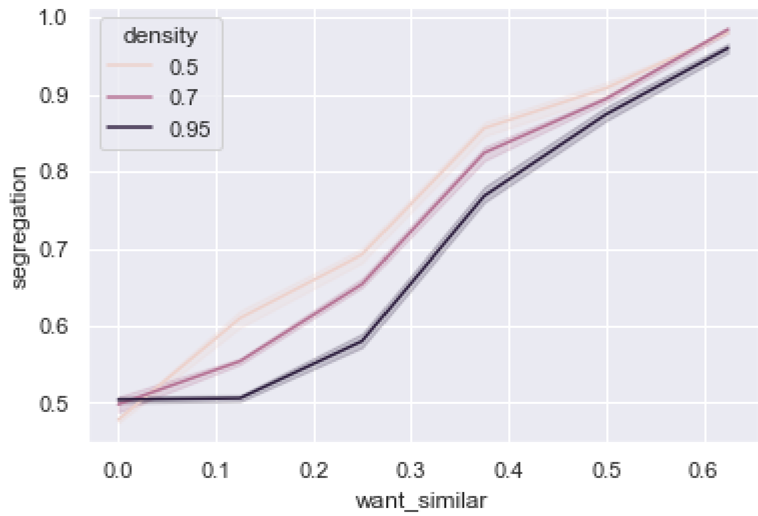
We now run an experiment where we simulate each parameter combination in our sample over 5 iterations.

[7]:
```
exp = ap.Experiment(SegregationModel, sample, iterations=5)
results = exp.run()
```

```
Scheduled runs: 90
Completed: 90, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:01:38.763931
```

Finally, we can arrange the results from our experiment into a dataframe with measures and variable parameters, and use the seaborn library to visualize the different segregation levels over our parameter ranges.

[8]:
```
data = results.arrange_measures()

sns.set()
ax = sns.lineplot(data=data, x='want_similar', y='segregation', hue='density')
```



Note: You can download this demonstration as a Jupyter Notebook `here`

5.4 Forest fire

This notebook presents an agent-based model that simulates a forest fire. It demonstrates how to use the agentpy package to work with a spatial grid and create animations, and perform a parameter sweep.

[1]:
```
# Model design
import agentpy as ap
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
import IPython
```

5.4.1 About the model

The model ist based on the NetLogo FireSimple model by Uri Wilensky and William Rand, who describe it as follows:

> "This model simulates the spread of a fire through a forest. It shows that the fire's chance of reaching the right edge of the forest depends critically on the density of trees. This is an example of a common feature of complex systems, the presence of a non-linear threshold or critical parameter. [. . .]
>
> The fire starts on the left edge of the forest, and spreads to neighboring trees. The fire spreads in four directions: north, east, south, and west.
>
> The model assumes there is no wind. So, the fire must have trees along its path in order to advance. That is, the fire cannot skip over an unwooded area (patch), so such a patch blocks the fire's motion in that direction."

5.4.2 Model definition

[2]:
```
class ForestModel(ap.Model):

    def setup(self):

        # Create grid (forest)
        forest = self.add_grid(self.p.size)

        # Create agents (trees)
        n_trees = int(self.p.density * (self.p.size**2))
        forest.add_agents(n_trees, random=True)

        # Initiate a dynamic variable for all trees
        # Condition 0: Alive, 1: Burning, 2: Burned
        self.agents.condition = 0

        # Start a fire from the left side of the grid
        unfortunate_trees = forest.get_agents([(0, self.p.size), (0, 0)])
        unfortunate_trees.condition = 1

    def step(self):

        # Select burning trees
        burning_trees = self.agents.select(self.agents.condition == 1)

        # Spread fire
        for agent in burning_trees:
            for neighbor in agent.neighbors():
                if neighbor.condition == 0:
                    neighbor.condition = 1 # Neighbor starts burning
            agent.condition = 2 # Tree burns out

        # Stop simulation if no fire is left
        if len(burning_trees) == 0: self.stop()

    def end(self):

        # Document a measure at the end of the simulation
        burned_trees = len(self.agents.select(self.agents.condition == 2))
        self.measure('Percentage of burned trees',
                     burned_trees / len(self.agents))
```

5.4.3 Single-run animation

[3]:
```
# Define parameters

parameters = {
    'density': 0.6, # Percentage of grid covered by trees
    'size': 50 # Height and length of the grid
}
```

[4]:
```
# Create single-run animation with custom colors

def animation_plot(model, ax):
```

(continues on next page)

(continued from previous page)

```
    attr_grid = model.env.attribute('condition', empty=3)
    color_dict = {0:'#7FC97F', 1:'#d62c2c', 2:'#e5e5e5', 3:'#d5e5d5'}
    ap.gridplot(attr_grid, ax=ax, color_dict=color_dict, convert=True)
    ax.set_title(f"Simulation of a forest fire\n"
                 f"Time-step: {model.t}, Trees left: "
                 f"{len(model.agents.select(model.agents.condition == 0))}")

fig, ax = plt.subplots()
model = ForestModel(parameters)
animation = ap.animate(model, fig, ax, animation_plot)
IPython.display.HTML(animation.to_jshtml())
```

[4]:
```
<IPython.core.display.HTML object>
```

5.4.4 Parameter sweep

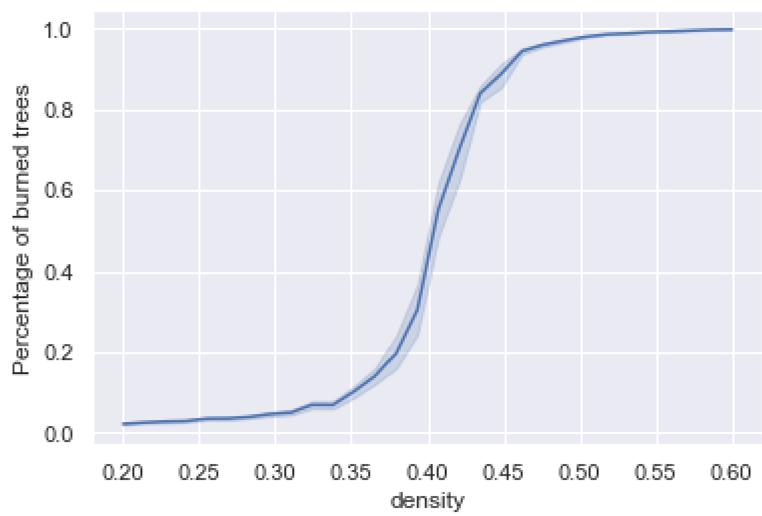
[5]:
```
# Prepare parameter sample
# Arranges 30 values for density from 0.1 to 1

parameter_ranges = {
    'density': (0.2,0.6),
    'size': 100
    }

sample = ap.sample(parameter_ranges, n=30)
```

[6]:
```
# Perform experiment
# Repeats simulation 30 times for each value of density

exp = ap.Experiment(ForestModel, sample, iterations=30)
results = exp.run()
```

```
Scheduled runs: 900
Completed: 900, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:04:52.087680
```

[7]:
```
# To save and load data

# results.save()
# results = ap.load('ForestModel')
```

[8]:
```
# Plot sensitivity
# Every point shows average over 50 runs

data = results.arrange_measures() # Create plotting data

sns.set()
ax = sns.lineplot(data=data, x='density', y='Percentage of burned trees')
```



Note: You can download this demonstration as a Jupyter Notebook `here`

5.5 Button network

This notebook presents an agent-based model of randomly connecting buttons. It demonstrates how to use the agentpy package to work with networks and visualize averaged time-series for discrete parameter samples.

[1]:
```
# Model design
import agentpy as ap
import networkx as nx
import random

# Visualization
import seaborn as sns
```

5.5.1 About the model

This model is based on the Agentbase Button model by Wybo Wiersma and the following analogy from Stuart Kauffman:

> "Suppose you take 10,000 buttons and spread them out on a hardwood floor. You have a large spool of red thread. Now, what you do is you pick up a random pair of buttons and you tie them together with a piece of red thread. Put them down and pick up another random pair of buttons and tie them together with a red thread, and you just keep doing this. Every now and then lift up a button and see how many buttons you've lifted with your first button. A connective cluster of buttons is called a cluster or a component. When you have 10,000 buttons and only a few threads that tie them together, most of the times you'd pick up a button you'll pick up a single button.
>
> As the ratio of threads to buttons increases, you're going to start to get larger clusters, three or four buttons tied together; then larger and larger clusters. At some point, you will have a number of intermediate clusters, and when you add a few more threads, you'll have linked up the intermediate-sized clusters into one giant cluster.

So that if you plot on an axis, the ratio of threads to buttons: 10,000 buttons and no threads; 10,000 buttons and 5,000 threads; and so on, you'll get a curve that is flat, and then all of a sudden it shoots up when you get this giant cluster. This steep curve is in fact evidence of a phase transition.

If there were an infinite number of threads and an infinite number of buttons and one just tuned the ratios, this would be a step function; it would come up in a sudden jump. So it's a phase transition like ice freezing.

Now, the image you should take away from this is if you connect enough buttons all of a sudden they all go connected. To think about the origin of life, we have to think about the same thing."

5.5.2 Model definition

[2]:
```
# Define the model

class ButtonModel(ap.Model):

    def setup(self):

        # Create a graph with n agents
        self.buttons = self.add_network()
        self.buttons.add_agents(self.p.n)
        self.threads = 0

    def update(self):

        # Record size of the biggest cluster
        clusters = nx.connected_components(self.buttons.graph)
        max_cluster_size = max([len(g) for g in clusters]) / self.p.n
        self.record('max_cluster_size', max_cluster_size)

        # Record threads to button ratio
        self.record('threads_to_button', self.threads / self.p.n)

    def step(self):

        # Create random edges based on parameters
        for _ in range(int(self.p.n * self.p.speed)):
            self.buttons.graph.add_edge(*self.agents.random(2))
            self.threads += 1
```

5.5.3 Multi-run experiment

[3]:
```
# Define parameter ranges
parameter_ranges = {
    'steps': 30,  # Number of simulation steps
    'speed': 0.05,  # Speed of connections per step
    'n': (100, 1000, 10000)  # Number of agents
}

# Create sample for different values of n
sample = ap.sample_discrete(parameter_ranges)

# Keep dynamic variables
exp = ap.Experiment(ButtonModel, sample, iterations=25, record=True)
```

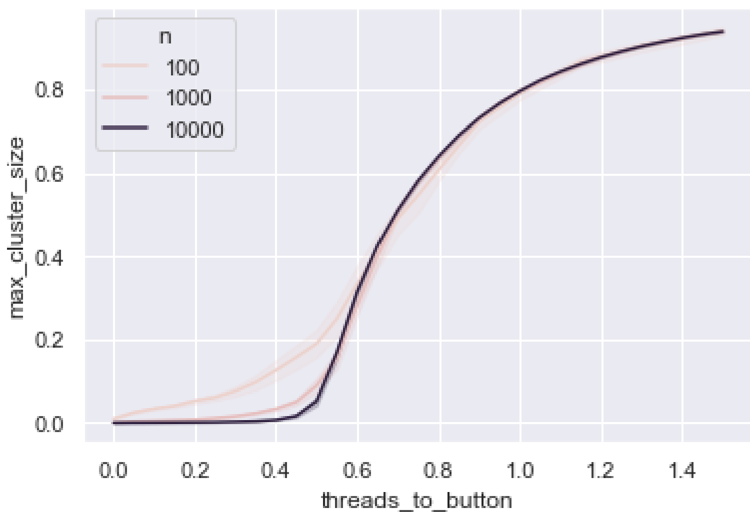
(continues on next page)

(continued from previous page)

```
# Perform 75 seperate simulations (3 parameter combinations * 25 repetitions)
results = exp.run()
```

```
Scheduled runs: 75
Completed: 75, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:59.906876
```

[4]:
```
# Plot averaged time-series for discrete parameter samples

sns.set()
data = results.arrange_variables()
ax = sns.lineplot(data=data, x='threads_to_button', y='max_cluster_size', hue='n')
```



CHAPTER SIX

API REFERENCE

6.1 Agents

class Agent(*model*, ***kwargs*)

Individual agent of an agent-based model.

This class can be used as a parent class for custom agent types. All agentpy model objects call the method `setup()` after creation, and can access class attributes like dictionary items. To add new agents to a model, use `Model.add_agents()` or `Environment.add_agents()`.

Parameters

- **model** (`Model`) – Instance of the current model.
- ****kwargs** – Will be forwarded to `Agent.setup()`.

Variables

- **model** (`Model`) – Model instance.
- **p** (`AttrDict`) – Model parameters.
- **envs** (`EnvList`) – Environments of the agent.
- **log** (`dict`) – Recorded variables of the agent.
- **id** (`int`) – Unique identifier of the agent.

delete()

Remove agent from all environments and the model.

enter(*env*)

Adds agent to passed environment.

Parameters env (`int or Environment, optional`) – Instance or id of environment that should be used. If none is given, the first environment in `Agent.envs` is used.

property env

The objects first environment.

exit(*env=None*)

Removes agent from chosen environment.

Parameters env (`int or Environment, optional`) – Instance or id of environment that should be used. If none is given, the first environment in `Agent.envs` is used.

move_by(*path*, *env=None*)

Changes the agents' location in the selected environment, relative to the current position.

Parameters

- **path** (*list of int*) – Relative change of position.
- **env** (*int or Environment, optional*) – Instance or id of environment that should be used. Must have topology 'grid'. If none is given, the first environment of that topology in `Agent.envs` is used.

move_to (*position*, *env=None*)
Changes the agents' location in the selected environment.

Parameters

- **position** (*list of int*) – Position to move to.
- **env** (*int or Environment, optional*) – Instance or id of environment that should be used. Must have topology 'grid'. If none is given, the first environment of that topology in `Agent.envs` is used.

neighbors (*env=None*, *distance=1*, *diagonal=True*)
Returns the agents' neighbor's from an environment, by calling the environments `neighbors()` function.

Parameters

- **env** (*int or Environment, optional*) – Instance or id of environment that should be used. Must have topology 'grid' or 'network'. If none is given, the first environment of that topology in `Agent.envs` is used.
- **distance** (*int, optional*) – Distance from agent in which to look for neighbors.
- **diagonal** (*bool, optional*) – Whether to include diagonal neighbors (only for *Grid*).

Returns Neighbors of the agent.

Return type *AgentList*

position (*env=None*)
Returns the agents' position from a grid.

Parameters env (*int or Environment, optional*) – Instance or id of environment that should be used. Must have topology 'grid'. If none is given, the first environment of that topology in `Agent.envs` is used.

record (*var_keys*, *value=None*)
Records an objects variables.

Parameters

- **var_keys** (*str or list of str*) – Names of the variables to be recorded.
- **value** (*optional*) – Value to be recorded. The same value will be used for all *var_keys*. If none is given, the values of object attributes with the same name as each var_key will be used.

Examples

Record the existing attributes x and y of an object a:

```
a.record(['x', 'y'])
```

Record a variable z with the value 1 for an object a:

```
a.record('z', 1)
```

Record all variables of an object:

```
a.record(a.var_keys)
```

setup (***kwargs*)
This empty method is called automatically at the objects' creation. Can be overwritten in custom subclasses to define initial attributes and actions.

Parameters ****kwargs** – Keyword arguments that have been passed to *Agent* or *Model.add_agents()*. If the original setup method is used, they will be set as attributes of the object.

Examples

The following setup initializes an object with three variables:

```
def setup(self, y):
    self.x = 0  # Value defined locally
    self.y = y  # Value defined in kwargs
    self.z = self.p.z   # Value defined in parameters
```

property type
Class name of the object (str).

property var_keys
The object's variables (list of str).

class AgentList (*iterable=(), /*)
List of agents.

Attribute calls and assignments are applied to all agents and return an *AttrList* with attributes of each agent. This also works for method calls, which returns a list of return values. Arithmetic operators can further be used to manipulate agent attributes, and boolean operators can be used to filter list based on agent attributes.

Examples

Prepare an *AgentList* with three agents:

```
>>> model = ap.Model()
>>> agents = model.add_agents(3)
>>> agents
AgentList [3 agents]
```

The assignment operator can be used to set a variable for each agent. When the variable is called, an *AttrList* is returned:

```
>>> agents.x = 1
>>> agents.x
AttrList of attribute 'x': [1, 1, 1]
```

One can also set different variables for each agent by passing another *AttrList*:

```
>>> agents.y = ap.AttrList([1, 2, 3])
>>> agents.y
AttrList of attribute 'y': [1, 2, 3]
```

Arithmetic operators can be used in a similar way. If an *AttrList* is passed, different values are used for each agent. Otherwise, the same value is used for all agents:

```
>>> agents.x = agents.x + agents.y
>>> agents.x
AttrList of attribute 'x': [2, 3, 4]

>>> agents.x *= 2
>>> agents.x
AttrList of attribute 'x': [4, 6, 8]
```

Boolean operators can be used to select a subset of agents:

```
>>> subset = agents(agents.x > 5)
>>> subset
AgentList [2 agents]

>>> subset.x
AttrList of attribute 'x': [6, 8]
```

append (*object*, /)
Append object to the end of the list.

clear ()
Remove all items from list.

copy ()
Return a shallow copy of the list.

count (*value*, /)
Return number of occurrences of value.

extend (*iterable*, /)
Extend list by appending elements from the iterable.

index (*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

insert (*index*, *object*, /)
Insert object before index.

pop (*index=-1*, /)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

random (*n=1*, *generator=None*)
Returns a new *AgentList* with a random subset of agents.

Parameters

- **n** (*int, optional*) – Number of agents (default 1).
- **generator** (*random.Random, optional*) – Random number generator. If none is passed, the hidden instance of *random* is used.

remove (*value, /*)

Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse ()

Reverse *IN PLACE*.

select (*selection*)

Returns a new `AgentList` based on *selection*.

Parameters **selection** (*list of bool*) – List with same length as the agent list. Positions that return True will be selected.

shuffle (*generator=None*)

Shuffles the list randomly and returns itself.

Parameters **generator** (*random.Random, optional*) – Random number generator. If none is passed, the hidden instance of *random* is used.

sort (*var_key, reverse=False*)

Sorts the list based on the *var_key* of its agents and returns itself.

6.2 Environments

6.2.1 Default

class Environment (*model, agents=None, **kwargs*)

Standard environment for agents (no topology).

This class can be used as a parent class for custom environment types. All agentpy model objects call the method `setup()` after creation, and can access class attributes like dictionary items. To add new environments to a model, use `Model.add_env()`.

Parameters

- **model** (`Model`) – The model instance.
- **agents** (`AgentList`, *optional*) – Agents to be added to the environment (default None).
- ****kwargs** – Will be forwarded to `Environment.setup()`.

Variables

- **model** (`Model`) – The model instance.
- **agents** (`AgentList`) – The environments' agents.
- **p** (`AttrDict`) – The models' parameters.
- **key** (*str*) – The environments' name.
- **topology** (*str*) – Topology of the environment.
- **log** (*dict*) – The environments' recorded variables.

add_agents(*agents=1*, *agent_class=<class 'agentpy.objects.Agent'>*, ***kwargs*)
Adds agents to the environment.

Parameters

- **agents** (*int or AgentList, optional*) – Either number of new agents to be created or list of existing agents (default 1).
- **agent_class** (*type, optional*) – Type of new agents to be created if int is passed for agents (default `Agent`).
- ****kwargs** – Forwarded to `Agent.setup()` if new agents are created (i.e. if an integer number is passed to *agents*).

Returns List of the new agents.

Return type *AgentList*

property env
The objects first environment.

record(*var_keys*, *value=None*)
Records an objects variables.

Parameters

- **var_keys** (*str or list of str*) – Names of the variables to be recorded.
- **value** (*optional*) – Value to be recorded. The same value will be used for all *var_keys*. If none is given, the values of object attributes with the same name as each var_key will be used.

Examples

Record the existing attributes `x` and `y` of an object `a`:

```
a.record(['x', 'y'])
```

Record a variable `z` with the value `1` for an object `a`:

```
a.record('z', 1)
```

Record all variables of an object:

```
a.record(a.var_keys)
```

remove_agents(*agents*)
Removes agents from the environment.

setup(***kwargs*)
This empty method is called automatically at the objects' creation. Can be overwritten in custom subclasses to define initial attributes and actions.

Parameters ****kwargs** – Keyword arguments that have been passed to `Agent` or `Model.add_agents()`. If the original setup method is used, they will be set as attributes of the object.

Examples

The following setup initializes an object with three variables:

```
def setup(self, y):
    self.x = 0  # Value defined locally
    self.y = y  # Value defined in kwargs
    self.z = self.p.z  # Value defined in parameters
```

property type
Class name of the object (str).

property var_keys
The object's variables (list of str).

class EnvList (*iterable=()*, */*)
List of environments.

Attribute calls and assignments are applied to all environments and return an `AttrList` with attributes of each environment. This also works for method calls, which returns a list of return values. Arithmetic operators can further be used to manipulate attributes, and boolean operators can be used to filter list based on attributes.

See `AgentList` for examples.

add_agents (**args*, ***kwargs*)
Add the same agents to all environments in the list. See `Environment.add_agents()` for arguments and keywords.

6.2.2 Networks

class Network (*model*, *graph=None*, *agents=None*, ***kwargs*)
Agent environment with a graph topology. Every node of the network represents an agent in the environment. To add new network environments to a model, use `Model.add_network()`.

This class can be used as a parent class for custom network types. All agentpy model objects call the method `setup()` after creation, and can access class attributes like dictionary items. See `Environment` for general properties of all environments.

Parameters

- **model** (`Model`) – The model instance.
- **graph** (*networkx.Graph, optional*) – The environments' graph. Agents of the same number as graph nodes must be passed. If none is passed, an empty graph is created.
- **agents** (`AgentList`, *optional*) – Agents of the network (default None). If a graph is passed, agents are mapped to each node of the graph. Otherwise, new nodes will be created for each agent.
- ****kwargs** – Will be forwarded to `Network.setup()`.

Variables graph (*networkx.Graph*) – The environments' graph.

add_agents (*agents*, *agent_class=<class 'agentpy.objects.Agent'>*, ***kwargs*)
Adds agents to the network environment as new nodes. See `Environment.add_agents()` for standard arguments.

property env
The objects first environment.

neighbors (*agent*, ***kwargs*)
Returns an *AgentList* of agents that are connected to the passed agent.

record (*var_keys*, *value=None*)
Records an objects variables.

Parameters

- **var_keys** (*str or list of str*) – Names of the variables to be recorded.
- **value** (*optional*) – Value to be recorded. The same value will be used for all *var_keys*. If none is given, the values of object attributes with the same name as each var_key will be used.

Examples

Record the existing attributes `x` and `y` of an object `a`:

```
a.record(['x', 'y'])
```

Record a variable `z` with the value `1` for an object `a`:

```
a.record('z', 1)
```

Record all variables of an object:

```
a.record(a.var_keys)
```

remove_agents (*agents*)
Removes agents from the environment.

setup (***kwargs*)
This empty method is called automatically at the objects' creation. Can be overwritten in custom sub-classes to define initial attributes and actions.

Parameters ****kwargs** – Keyword arguments that have been passed to *Agent* or *Model.add_agents()*. If the original setup method is used, they will be set as attributes of the object.

Examples

The following setup initializes an object with three variables:

```
def setup(self, y):
    self.x = 0  # Value defined locally
    self.y = y  # Value defined in kwargs
    self.z = self.p.z  # Value defined in parameters
```

property type
Class name of the object (str).

property var_keys
The object's variables (list of str).

6.2.3 Spatial grids

class Grid(*model*, *shape*, ***kwargs*)

Environment that contains agents with a spatial topology. Every location consists of an `AgentList` that can hold zero, one, or more agents. To add new grid environments to a model, use `Model.add_grid()`.

This class can be used as a parent class for custom network types. All agentpy model objects call the method `setup()` after creation, and can access class attributes like dictionary items. See `Environment` for general properties of all environments.

Parameters

- **model** (`Model`) – The model instance.
- **shape** (*int or tuple of int*) – Size of the grid. If an integer is given, this value is taken as both the height and width for a two-dimensional grid. If a tuple is given, the length of the tuple defines the number of dimensions, and the values in the tuple define the length of each dimension.
- ****kwargs** – Will be forwarded to `Grid.setup()`.

Variables

- **grid** (*list of lists*) – Matrix of `AgentList`.
- **shape** (*tuple of int*) – Length of each grid dimension.

add_agents(*agents=1*, *agent_class=<class 'agentpy.objects.Agent'>*, *positions=None*, *random=False*, ***kwargs*)

Adds agents to the grid environment. See `Environment.add_agents()` for standard arguments. Additional arguments are listed below.

Parameters

- **positions** (*list of tuples, optional*) – The positions of the added agents. List must have the same length as number of agents to be added, and each entry must be a tuple with coordinates. If none is passed, agents will fill up the grid systematically.
- **random** (*bool, optional*) – If no positions are passed, agents will be placed in random locations instead of systematic filling (default False).

apply(*func*, **args*, ***kwargs*)

Applies a function to all grid positions, and returns grid with return values.

attribute(*attr_key*, *sum_values=True*, *empty=nan*)

Returns a grid with the value of the attributes of the agents in each position.

Parameters

- **attr_key** (*str*) – Name of the attribute.
- **sum_values** (*str, optional*) – What to return in a position where there are multiple agents. If True (default), the sum of attributes. If False, a list of attributes.
- **empty** (*optional*) – What to return for empty positions without agents (default numpy.nan).

property env

The objects first environment.

get_agents(*area=None*)

Returns an `AgentList` with agents in the selected positions or area.

Parameters **area** (*tuple of integers or tuples*) – Area from which agents should be gathered. Can either indicate a single position *[x, y, …]* or an area *[(x_start, x_end), (y_start, y_end), …]*.

items (*area=None*)
Returns iterator with tuples of style: (position, agents).

move_agent (*agent, position*)
Moves agent to new position.

Parameters

- **agent** (*int or Agent*) – Id or instance of the agent.
- **position** (*list of int*) – New position of the agent.

neighbors (*agent, distance=1, diagonal=True*)
Returns agent neighbors.

Parameters

- **agent** (*int or Agent*) – Id or instance of the agent.
- **distance** (*int, optional*) – Number of positions to cover in each direction.
- **diagonal** (*bool, optional*) – If True (default), diagonal neighbors are included. If False, only direct neighbors are included (currently only works with distance == 1).

position (*agent*)
Returns position of a passed agent.

Parameters **agent** (*int or Agent*) – Id or instance of the agent.

positions (*area=None*)
Returns iterable of all grid positions in area.

Parameters **area** (*list of tuples, optional*) – Area of positions that should be returned. If none is passed, the whole grid is selected. Style: *[(x_start, x_end), (y_start, y_end), …]*

record (*var_keys, value=None*)
Records an objects variables.

Parameters

- **var_keys** (*str or list of str*) – Names of the variables to be recorded.
- **value** (*optional*) – Value to be recorded. The same value will be used for all *var_keys*. If none is given, the values of object attributes with the same name as each var_key will be used.

Examples

Record the existing attributes `x` and `y` of an object `a`:

```
a.record(['x', 'y'])
```

Record a variable `z` with the value `1` for an object `a`:

```
a.record('z', 1)
```

Record all variables of an object:

```
a.record(a.var_keys)
```

remove_agents(*agents*)
Removes agents from the environment.

setup(***kwargs*)
This empty method is called automatically at the objects' creation. Can be overwritten in custom subclasses to define initial attributes and actions.

Parameters ****kwargs** – Keyword arguments that have been passed to `Agent` or `Model.add_agents()`. If the original setup method is used, they will be set as attributes of the object.

Examples

The following setup initializes an object with three variables:

```
def setup(self, y):
    self.x = 0  # Value defined locally
    self.y = y  # Value defined in kwargs
    self.z = self.p.z  # Value defined in parameters
```

property type
Class name of the object (str).

property var_keys
The object's variables (list of str).

6.3 Agent-based models

class Model(*parameters=None*, *run_id=None*, *scenario=None*, ***kwargs*)
An agent-based model that can hold environments and agents.

This class can be used as a parent class for custom models. Class attributes can be accessed like dictionary items. To define the procedures of a simulation, override the methods `Model.setup()`, `Model.step()`, `Model.update()`, and `Model.end()`. See `Model.run()` for more information on the simulation procedure.

Variables

- **name** (`str`) – The models' name.
- **envs** (`EnvList`) – The models' environments.
- **agents** (`AgentList`) – The models' agents.
- **p** (`AttrDict`) – The models' parameters.
- **t** (`int`) – Current time-step of the model.
- **log** (`dict`) – The models' recorded variables.
- **output** (`DataDict`) – Output data after simulation.

Parameters

- **parameters** (`dict, optional`) – Dictionary of model parameters. Recommended types for parameters are int, float, str, list, numpy.integer, numpy.floating, and numpy.ndarray. Other types might cause errors.

- **run_id** (*int, optional*) – Number of current run (default None).
- **scenario** (*str, optional*) – Current scenario (default None).
- ****kwargs** – Will be forwarded to `Model.setup()`

add_agents (*agents=1, agent_class=<class 'agentpy.objects.Agent'>, **kwargs*)
Adds agents to the environment.

Parameters

- **agents** (*int or AgentList, optional*) – Either number of new agents to be created or list of existing agents (default 1).
- **agent_class** (*type, optional*) – Type of new agents to be created if int is passed for agents (default `Agent`).
- ****kwargs** – Forwarded to `Agent.setup()` if new agents are created (i.e. if an integer number is passed to *agents*).

Returns List of the new agents.

Return type *AgentList*

add_env (*env_class=<class 'agentpy.objects.Environment'>, **kwargs*)
Creates a new environment.

add_grid (*shape, **kwargs*)
Creates a new environment with a spatial grid. Arguments are forwarded to `Grid`.

add_network (*graph=None, agents=None, **kwargs*)
Creates a new environment with a network. Arguments are forwarded to `Network`.

end ()
Defines the model's actions after the last simulation step. Can be overwritten and used for final calculations and measures.

property env
The objects first environment.

get_obj (*obj_id*)
Return model object with obj_id (int).

measure (*measure, value*)
Records an evaluation measure.

property objects
The models agents and environments (list of objects).

record (*var_keys, value=None*)
Records an objects variables.

Parameters

- **var_keys** (*str or list of str*) – Names of the variables to be recorded.
- **value** (*optional*) – Value to be recorded. The same value will be used for all *var_keys*. If none is given, the values of object attributes with the same name as each var_key will be used.

Examples

Record the existing attributes x and y of an object a:

```
a.record(['x', 'y'])
```

Record a variable z with the value 1 for an object a:

```
a.record('z', 1)
```

Record all variables of an object:

```
a.record(a.var_keys)
```

remove_agents(*agents*)
Removes agents from the environment.

run(*steps=None*, *seed=None*, *display=True*)
Executes the simulation of the model.

The simulation proceeds as follows. It starts by calling *Model.setup()* and *Model.update()*. After that, Model.t is increased by 1 and *Model.step()* and *Model.update()* are called. This step is repeated until the method *Model.stop()* is called or steps is reached. After the last step, *Model.end()* is called.

Parameters

- **steps** (*int, optional*) – Maximum number of steps for the simulation to run. If none is given, the parameter 'Model.p.steps' will be used. If there is no such parameter, 'steps' will be set to 1000.
- **seed** (*int, optional*) – Seed to set for random at the beginning of the simulation. If none is given, the parameter 'Model.p.seed' will be used. If there is no such parameter, no custom seed will be set.
- **display** (*bool, optional*) – Whether to display simulation progress (default True).

Returns Recorded model data, which can also be found in Model.output.

Return type *DataDict*

setup(***kwargs*)
Defines the model's actions before the first simulation step. Can be overwritten and used to initiate agents and environments.

step()
Defines the model's actions during each simulation step. Can be overwritten and used to set the models' main dynamics.

stop()
Stops *Model.run()* during an active simulation.

property type
Class name of the object (str).

update()
Defines the model's actions after setup and each simulation step. Can be overwritten and used for the recording of dynamic variables.

property var_keys
The object's variables (list of str).

6.4 Parameter sampling

sample (*parameter_ranges*, *n*, *digits=None*)

Creates a sample of different parameter combinations by seperating each range into 'n' values, using `numpy.linspace()`.

Parameters

- **parameter_ranges** (*dict*) – Dictionary of parameters. Only values that are given as a tuple will be varied. Tuple must be of the following style: (min_value, max_value). If both values are of type int, the output will be rounded and converted to int.
- **n** (*int*) – Number of values to sample per varied parameter.
- **digits** (*int, optional*) – Number of digits to round the output values to (default None).

Returns List of parameter dictionaries

Return type list of dict

sample_discrete (*parameter_ranges*)

Creates a sample of different parameter combinations from all possible combinations within the passed parameter ranges.

Parameters **parameter_ranges** (*dict*) – Dictionary of parameters. Only values that are given as a tuple will be varied. Tuples must be of the following style: (value1, value2, value3, . . .).

Returns List of parameter dictionaries

Return type list of dict

sample_saltelli (*parameter_ranges*, *n*, *calc_second_order=True*, *digits=None*)

Creates a sample of different parameter combinations, using `SALib.sample.saltelli.sample()`.

Parameters

- **parameter_ranges** (*dict*) – Dictionary of parameters. Only values that are given as a tuple will be varied. Tuple must be of the following style: (min_value, max_value). If both values are of type int, the output will be rounded and converted to int.
- **n** (*int*) – The number of samples to generate, see `SALib.sample.saltelli.sample()`.
- **calc_second_order** (*bool, optional*) – Calculate second-order sensitivities (default True).
- **digits** (*int, optional*) – Number of digits to round the output values to (default None).

Returns List of parameter dictionaries

Return type list of dict

6.5 Experiments

class Experiment(*model_class*, *parameters=None*, *name=None*, *scenarios=None*, *iterations=1*, *record=False*, ***kwargs*)

Experiment for an agent-based model. Allows for multiple iterations, parameter samples, scenario comparison, and parallel processing. See `Experiment.run()` for standard simulations and `Experiment.interactive()` for interactive output.

Parameters

- **model_class** (*type*) – The model class type that the experiment should use.
- **parameters** (*dict or list of dict, optional*) – Parameter dictionary or sample (default None).
- **name** (*str, optional*) – Name of the experiment (default model.name).
- **scenarios** (*str or list, optional*) – Experiment scenarios (default None).
- **iterations** (*int, optional*) – Experiment repetitions (default 1).
- **record** (*bool, optional*) – Whether to keep the record of dynamic variables (default False). Note that this does not affect evaluation measures.
- ****kwargs** – Will be forwarded to the creation of every model instance during the experiment.

Variables **output** (`DataDict`) – Recorded experiment data

interactive(*plot*, **args*, ***kwargs*)

Displays interactive output for Jupyter notebooks, using `IPython` and `ipywidgets`. A slider will be shown for varied parameters. Every time a parameter value is changed on the slider, the experiment will re-run the model and pass it to the 'plot' function.

Parameters

- **plot** – Function that takes a model instance as input and prints or plots the desired output..
- ***args** – Will be forwarded to 'plot'.
- ****kwargs** – Will be forwarded to 'plot'.

Returns Interactive output widget

Return type ipywidgets.HBox

Examples

The following example uses a custom model `MyModel` and creates a slider for the parameters 'x' and 'y', both of which can be varied interactively over 10 different values. Every time a value is changed, the experiment will simulate the model with the new parameters and pass it to the plot function:

```
def plot(model):
    # Display interactive output here
    print(model.output)

param_ranges = {'x': (0, 10), 'y': (0., 1.)}
sample = ap.sample(param_ranges, n=10)
exp = ap.Experiment(MyModel, sample)
exp.interactive(plot)
```

run (*pool=None*, *display=True*)
Executes a multi-run experiment.

The simulation will run the model once for each set of parameters and will repeat this process for the set number of iterations. Parallel processing is possible if a *pool* is passed. Simulation results will be stored in *Experiment.output*.

Parameters

- **pool** (*multiprocessing.Pool, optional*) – Pool of active processes for parallel processing. If none is passed, normal processing is used.
- **display** (*bool, optional*) – Display simulation progress (default True).

Returns Recorded experiment data.

Return type *DataDict*

Examples

To run a normal experiment:

```
exp = ap.Experiment(MyModel, parameters)
results = exp.run()
```

To use parallel processing:

```
import multiprocessing as mp
if __name__ == '__main__':
    exp = ap.Experiment(MyModel, parameters)
    pool = mp.Pool(mp.cpu_count())
    results = exp.run(pool)
```

6.6 Output data

class DataDict (**args*, ***kwargs*)
Dictionary for recorded simulation data.

Generated by *Model*, *Experiment*, or *load()*. Dictionary items can be defined and accessed like attributes. Attributes can differ from the standard ones listed below.

Variables

- **log** (*dict*) – Meta-data of the simulation (e.g. name, time-stamps, settings, etc.).
- **parameters** (*dict, pandas.DataFrame, or DataDict*) – Parameters that have been used for the simulation.
- **variables** (*pandas.DataFrame or DataDict)*) – Dynamic variables, seperated per object type, which can be recorded once per time-step with `record()`.
- **measures** (*pandas.DataFrame*) – Evaluation measures, which can be recorded once per run with `measure()`.

arrange (*variables=None*, *measures=None*, *parameters=None*, *obj_types='all'*, *scenarios='all'*, *index=False*)
Combines and/or filters data based on passed arguments.

Parameters

- **variables** (*str or list of str, optional*) – Variables to include in the new dataframe (default None). If 'all', all are selected.
- **measures** (*str or list of str, optional*) – Measures to include in the new dataframe (default None). If 'all', all are selected.
- **parameters** (*str or list of str, optional*) – Parameters to include in the new dataframe (default None). If 'fixed', all fixed parameters are selected. If 'varied', all varied parameters are selected. If 'all', all are selected.
- **obj_types** (*str or list of str, optional*) – Agent and/or environment types to include in the new dataframe. Note that the selected object types will only be included if at least one of their variables is declared in 'variables'. If 'all', all are selected (default).
- **scenarios** (*str or list of str, optional*) – Scenarios to include in the new dataframe. If 'all', all are selected (default).
- **index** (*bool, optional*) – Whether to keep original multi-index structure (default False).

Returns The arranged dataframe

Return type pandas.DataFrame

arrange_measures (*variables=None, measures='all', parameters='varied', obj_types='all', scenarios='all', index=False*)

Returns a dataframe with measures and varied parameters. See `DataDict.arrange()` for further information.

arrange_variables (*variables='all', measures=None, parameters='varied', obj_types='all', scenarios='all', index=False*)

Returns a dataframe with variables and varied parameters. See `DataDict.arrange()` for further information.

save (*exp_name=None, exp_id=None, path='ap_output', display=True*)

Writes data to directory *{path}/{exp_name}_{exp_id}/*. Works only for entries that are of type `DataDict`, `pandas.DataFrame`, or serializable with JSON (int, float, str, dict, list). Numpy objects will be converted to standard objects, if possible.

Parameters

- **exp_name** (*str, optional*) – Name of the experiment to be saved. If none is passed, *self.log['name']* is used.
- **exp_id** (*int, optional*) – Number of the experiment. If none is passed, a new id is generated.
- **path** (*str, optional*) – Target directory (default 'ap_output').
- **display** (*bool, optional*) – Display saving progress (default True).

load (*exp_name=None, exp_id=None, path='ap_output', display=True*)

Reads output data from directory *{path}/{exp_name}_{exp_id}/*.

Parameters

- **exp_name** (*str, optional*) – Experiment name. If none is passed, the most recent experiment is chosen.
- **exp_id** (*int, optional*) – Id number of the experiment. If none is passed, the highest available id used.
- **path** (*str, optional*) – Target directory (default 'ap_output').

- **display** (*bool, optional*) – Display loading progress (default True).

Returns The loaded data from the chosen experiment.

Return type *DataDict*

6.7 Analysis

6.7.1 Sensitivity

sensitivity_sobol (*output*, *param_ranges*, *measures=None*, ***kwargs*)

Calculates Sobol Sensitivity Indices and adds them to the output, using `SALib.analyze.sobol.analyze()`.

Parameters

- **output** (DataDict) – The output of an experiment that was set to only one iteration (default) and used a parameter sample that was generated with *sample_saltelli()*.
- **param_ranges** (*dict*) – The same dictionary that was used for the generation of the parameter sample with *sample_saltelli()*.
- **measures** (*str or list of str, optional*) – The measures that should be used for the analysis. If none are passed, all are used.
- ****kwargs** – Will be forwarded to `SALib.analyze.sobol.analyze()`. The kwarg `calc_second_order` must be the same as for *sample_saltelli()*.

6.7.2 Animations

animate (*model*, *fig*, *axs*, *plot*, *steps=None*, *skip=0*, *fargs=()*, ***kwargs*)

Returns an animation of the model simulation, using `matplotlib.animation.FuncAnimation()`.

Parameters

- **model** (Model) – The model instance.
- **fig** (*matplotlib.figure.Figure*) – Figure for the animation.
- **axs** (*matplotlib.axes.Axes or list*) – Axis or list of axis of the figure.
- **plot** (*function*) – Function that takes *(model, ax, *fargs)* and creates the desired plots on each axis at each time-step.
- **steps** (*int, optional*) – Maximum number of steps for the simulation to run. If none is given, the parameter 'Model.p.steps' will be used. If there is no such parameter, 'steps' will be set to 1000.
- **skip** (*int, optional*) – Number of rounds to skip before the animation starts (default 0).
- **fargs** (*tuple, optional*) – Forwarded fo the *plot* function.
- ****kwargs** – Forwarded to `matplotlib.animation.FuncAnimation()`.

Examples

An animation can be generated as follows:

```
def my_plot(model, ax):
    pass  # Call pyplot functions here

fig, ax = plt.subplots()
my_model = MyModel(parameters)
animation = ap.animate(my_model, fig, ax, my_plot)
```

One way to display the resulting animation object in Jupyter:

```
from IPython.display import HTML
HTML(animation.to_jshtml())
```

6.7.3 Plots

gridplot (*grid*, *color_dict=None*, *convert=False*, *ax=None*, ***kwargs*)
Visualizes values on a two-dimensional grid with `matplotlib.pyplot.imshow()`.

Parameters

- **grid** (*list of list*) – Two-dimensional grid with values. numpy.nan values will be plotted as empty patches.
- **color_dict** (*dict, optional*) – Dictionary that translates each value in *grid* to a color specification.
- **convert** (*bool, optional*) – Convert values to rgba vectors, using `matplotlib.colors.to_rgba()` (default False).
- **ax** (*matplotlib.pyplot.axis, optional*) – Axis to be used for plot.
- ****kwargs** – Forwarded to `matplotlib.pyplot.imshow()`.

6.8 Other

class AttrDict (**args*, ***kwargs*)
Dictionary where attribute calls are handled like item calls.

Examples

```
>>> ad = ap.AttrDict()
>>> ad['a'] = 1
>>> ad.a
1
```

```
>>> ad.b = 2
>>> ad['b']
2
```

class AttrList(**args*, *attr=None*)

List of attributes from an `AgentList`.

Calls are forwarded to each entry and return a list of return values. Boolean operators are applied to each entry and return a list of bools. Arithmetic operators are applied to each entry and return a new list. See `AgentList` for examples.

CHAPTER

SEVEN

COMPARISON

7.1 Agentpy vs. Mesa

An alternative framework for agent-based modeling in Python is Mesa. The stated goal of Mesa is *"to be the Python 3-based counterpart to NetLogo, Repast, or MASON"*. The focus of these frameworks is traditionally on spatial environments, with an interface where one can observe live dynamics and adjust parameters while the model is running.

Agentpy, in contrast, is more focused on networks and *multi-run experiments*, with tools to generate and analyze *output data* from these experiments. Agentpy further has a different model structure that is built around *agent lists*, which allow for simple selection and manipulation of agent groups; and *environments*, which can contain agents but also act as agents themselves.

To allow for an comparison of the syntax of each framework, here are two examples for a simple model of wealth transfer, both of which realize exactly the same operations. More information on the two models can be found in the documentation of each framework (link for *Agentpy* & Mesa).

<table>
<tr><th>Agentpy</th><th>Mesa</th></tr>
<tr><td>

```
import agentpy as ap



class MoneyAgent(ap.Agent):

    def setup(self):
        self.wealth = 1

    def wealth_transfer(self):
        if self.wealth == 0:
            return
        a = self.model.agents.random()
        a.wealth += 1
        self.wealth -= 1



class MoneyModel(ap.Model):

    def setup(self):
        self.add_agents(
            self.p.agents, MoneyAgent)

    def step(self):
        self.agents.record('wealth')
        self.agents.wealth_transfer()










# Perform single run
parameters = {'agents': 10, 'steps': 10}
model = MoneyModel(parameters)
results = model.run()

# Perform multiple runs
parameters['agents'] = (10, 500, int)
sample = ap.sample(parameters, n=49)

exp = ap.Experiment(
    MoneyModel,
    sample,
    iterations=5,
    record=True
)


results = exp.run()
```

</td><td>

```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.batchrunner import BatchRunner
from mesa.datacollection \
    import DataCollector

class MoneyAgent(Agent):

    def __init__(self, unique_id, model):
        super().__init__(unique_id,␣
↪model)
        self.wealth = 1

    def step(self):
        if self.wealth == 0:
            return
        other_agent = self.random.choice(
            self.model.schedule.agents)
        other_agent.wealth += 1
        self.wealth -= 1

class MoneyModel(Model):

    def __init__(self, N):
        self.running = True
        self.num_agents = N
        self.schedule = \
            RandomActivation(self)
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)

        self.collector = DataCollector(
            agent_reporters={
                "Wealth": "wealth"})

    def step(self):
        self.collector.collect(self)
        self.schedule.step()

# Perform single run
model = MoneyModel(10)
for i in range(10):
    model.step()

# Perform multiple runs
variable_params = {
    "N": range(10, 500, 10)}

batch_run = BatchRunner(
    MoneyModel,
    variable_params,
    iterations=5,
    max_steps=10,
    agent_reporters={"Wealth": "wealth"}
)

batch_run.run_all()
```

</td></tr>
</table>

Finally, the following table provides a comparison of the main features of each framework.

Feature	Agentpy	Mesa
Customizable objects	Agent, Environment, Model	Agent, Model
Container classes	AgentList and EnvDict for selection and manipulation of agent and environment groups	Scheduler (see below)
Time management	Custom activation order has to be defined in the Model.step method	Multiple scheduler classes for different activation orders
Supported topologies	Spatial grid, networkx graph	Spatial grid, network grid, continuous space
Data recording	Recording methods for variables (of agents, environments, and model) and evaluation measures	DataCollector class that can collect variables of agents and model
Parameter sampling	Multiple sampling functions	Custom sample has to be defined
Multi-run experiments	Experiment class that supports multiple iterations, parameter samples, scenario comparison, and parallel processing	BatchRunner class that supports multiple iterations and parameter samples
Output data	DataDict class that can save, load, and re-arrange output data	Multiple methods to generate dataframes
Visualization	Tools for plots and animations, and interactive visualization in Python	Extensive browser-based visualization module
Analysis	Tools for data arrangement and sensitivity analysis	

CHAPTER

EIGHT

CHANGELOG

8.1 0.0.7.dev0

- A custom seed can now be set for `Model.run()` by either passing an argument or defining a parameter `seed`.
- `Environment` has a new optional argument `agents` to add existing agents at the creation of the environment.
- `AgentList.random()` and `AgentList.shuffle()` have a new optional argument `generator` for custom instances of `random.Random`.

8.2 0.0.6 (January 2021)

- A new demonstration model *Segregation* has been added.
- All model objects now have a unique id number of type `int`. Methods that take an agent or environment as an argument can now take either the instance or id of the object. The `key` attribute of environments has been removed.
- Extra keyword arguments to `Model` and `Experiment` are now forwarded to `Model.setup()`.
- `Model.run()` now takes an optional argument *steps*.
- `EnvDict` has been replaced by `EnvList`, which has the same functionalities as `AgentList`.
- Model objects now have a property `env` that returns the first environment of the object.
- Revision of `Network`. The argument *map_to_nodes* has been removed from `Network.add_agents()`. Instead, agents can be mapped to nodes by passing an AgentList to the agents argument of `Model.add_network()`. Direct forwarding of attribute calls to `Network.graph` has been removed to avoid confusion.
- New and revised methods for `Grid`:
  - `Agent.move_to()` and `Agent.move_by()` can be used to move agents.
  - `Grid.items()` returns an iterator of position and agent tuples.
  - `Grid.get_agents()` returns agents in selected position or area.
  - `Grid.position()` returns the position coordinates for an agent.
  - `Grid.positions()` returns an iterator of position coordinates.
  - `Grid.attribute()` returns a nested list with values of agent attributes.
  - `Grid.apply()` returns nested list with return values of a custom function.
  - `Grid.neighbors()` has new arguments *diagonal* and *distance*.

- `gridplot()` now takes a grid of values as an input and can convert them to rgba.
- `animate()` now takes a model instance as an input instead of a class and parameters.
- `sample()` and `sample_saltelli()` will now return integer values for parameters if parameter ranges are given as integers. For float values, a new argument *digits* can be passed to round parameter values.
- The function `interactive()` has been removed, and is replaced by the new method `Experiment.interactive()`.
- `sobol_sensitivity()` has been changed to `sensitivity_sobol()`.

8.3 0.0.5 (December 2020)

- `Experiment.run()` now supports parallel processing.
- New methods `DataDict.arrange_variables()` and `DataDict.arrange_measures()`, which generate a dataframe of recorded variables or measures and varied parameters.
- Major revision of `DataDict.arrange()`, see new description in the documentation.
- New features for `AgentList`: Arithmethic operators can now be used with `AttrList`.

8.4 0.0.4 (November 2020)

- First major release.

CHAPTER

NINE

CONTRIBUTE

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

9.1 Types of contributions

9.1.1 Report bugs

Report bugs at https://github.com/JoelForamitti/agentpy/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

9.1.2 Fix bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

9.1.3 Implement features

Look through the GitHub issues and discussion forum for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

9.1.4 Write documentation

Agentpy could always use more documentation, whether as part of the official agentpy docs, in docstrings, or even on the web in blog posts, articles, and such.

9.1.5 Submit feedback

The best way to send feedback is to write in the agentpy discussion forum at https://github.com/JoelForamitti/agentpy/discussions.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

9.2 How to contribute

Ready to contribute? Here's how to set up *agentpy* for local development.

1. Fork the *agentpy* repository on GitHub: https://github.com/JoelForamitti/agentpy
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/agentpy.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv agentpy
$ cd agentpy/
$ pip install -e .['dev']
```

4. Create a branch for local development:

```
 $ git checkout -b name-of-your-bugfix-or-feature

Now you can make your changes locally.
```

5. When you're done making changes, check that your changes pass the tests and that the new features are covered by the tests:

```
$ coverage run -m pytest
$ coverage report
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

9.3 Pull request guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests. For more information, check out the tests directory and https://docs.pytest.org/.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to docs/changelog.rst.
3. The pull request should pass the automatic tests on travis-ci. Check https://travis-ci.com/JoelForamitti/agentpy/pull_requests and make sure that the tests pass for all supported Python versions.

CHAPTER TEN

ABOUT

Agentpy has been created by Joël Foramitti and is available under the open-source BSD 3-Clause license. Source files can be found on the GitHub repository.

This project has benefited from an ERC Advanced Grant from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement n° 741087).

Special thanks for their feedback and support go to Ivan Savin and Jeroen C.J.M van den Bergh.

Parts of this package where created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

INDEX

S

T

U

V