agentpy

Release 0.1.5

Joël Foramitti

Dec 21, 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, interactive simulations, numerical experiments, and data analysis within a single environment. The package is optimized for interactive computing with IPython, IPySimulate, and Jupyter. If you have questions or ideas for improvements, please visit the discussion forum.

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*.
- If you are interested to contribute to the library, see *Contribute*.

Citation

Please cite this software as follows:

```
Foramitti, J., (2021). AgentPy: A package for agent-based modeling in Python.
Journal of Open Source Software, 6(62), 3065, https://doi.org/10.21105/joss.03065
```

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 and higher. The installation includes the following packages:

- numpy and scipy, for scientific computing
- matplotlib, for visualization
- pandas, for data manipulation
- networkx, for networks/graphs
- SALib, for sensitivity analysis
- joblib, for parallel processing

These optional packages can further be useful in combination with agentpy:

- jupyter, for interactive computing
- ipysimulate >= 0.2.0, for interactive simulations
- ema_workbench, for exploratory modeling
- 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 & testing, you can use:

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

CHAPTER THREE

OVERVIEW

This section provides an 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 *User Guides* and *API Reference*. Throughout this documentation, AgentPy is imported as follows:

```
import agentpy as ap
```

3.1 Structure

The basic structure of the AgentPy framework has four levels:

1. The `Agent` is the basic building block of a model
2. The environment types `Grid`, `Space`, and `Network` contain agents
3. A `Model` contains agents, environments, parameters, and simulation procedures
4. An `Experiment` can run a model multiple times with different parameter combinations

All of these classes are templates that can be customized through the creation of sub-classes with their own variables and methods.

3.2 Creating models

A custom agent type can be defined as follows:

```
class MyAgent(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 automatically after an agent's creation. All variables of an agents should be initialized within this method. Other methods can represent actions that the agent will be able to take during a simulation.

All model objects (including agents, environments, and the model itself) are equipped with the following default attributes:

- `model` the model instance
- `id` a unique identifier number for each object
- `p` the model's parameters
- `log` the object's 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):
        """ Initiate a list of new agents. """
        self.agents = ap.AgentList(self, self.p.agents, MyAgent)

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

    def update(self):
        """ Record a dynamic variable. """
        self.agents.record('my_attribute')

    def end(self):
        """ Repord an evaluation measure. """
        self.report('my_measure', 1)
```

The simulation procedures of a model are defined by four special methods that will be used automatically during different parts of a simulation.

- `Model.setup` is called at the start of the simulation (*t==0*).
- `Model.step` is called during every time-step (excluding *t==0*).
- `Model.update` is called after every time-step (including *t==0*).
- `Model.end` is called at the end of the 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*.

3.3 Agent sequences

The *Sequences* module provides containers for groups of agents. The main classes are `AgentList`, `AgentDList`, and `AgentSet`, which come with special methods to access and manipulate whole groups 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, selects all agents with an id above one:

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

Further examples can be found in *Sequences* and the *Virus spread* demonstration model.

3.4 Environments

Environments are objects in which agents can inhabit a specific position. A model can contain zero, one or multiple environments which agents can enter and leave. The connection between positions is defined by the environment's topology. There are currently three types:

- `Grid` n-dimensional spatial topology with discrete positions.
- `Space` n-dimensional spatial topology with continuous positions.
- `Network` graph topology consisting of `AgentNode` and edges.

Applications of networks can be found in the demonstration models *Virus spread* and *Button network*; spatial grids in *Forest fire* and *Segregation*; and continuous spaces in *Flocking behavior*. Note that there can also be models without environments like in *Wealth transfer*.

3.5 Recording data

There are two ways to document data from the simulation for later *analysis*.

The first way is to record dynamic variables, which can be recorded 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 documented by calling the method `record()` for the respective object. Recorded variables can at run-time with the object's *log* attribute.

The second way is to document reporters, which represent summary statistics or evaluation measures of a simulation. In contrast to variables, reporters can be stored only for the model as a whole and only once per run. They will be stored in a separate dataframe for easy comparison over multiple runs, and can be documented with the method `Model.report()`. Reporters can be accessed at run-time via `Model.reporters`.

3.6 Running a simulation

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

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

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

A simulation proceeds as follows (see also Figure 1 below):

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 two 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()*.

3.7 Interactive simulations

Within a Jupyter Notebook, AgentPy models can be explored as an interactive simulation (similar to the traditional NetLogo interface) using ipysimulate and d3.js. For more information on this, please refer to *Interactive simulations*.

3.8 Multi-run experiments

The *Parameter samples* module provides tools to create a *Sample* with multiple parameter combinations from a dictionary of ranges. Here is an example using *IntRange* integer ranges:

```
parameters = {
    'my_parameter': 42,
    'agents': ap.IntRange(10, 20),
    'steps': ap.IntRange(10, 20)
}
sample = ap.Sample(parameters, n=5)
```

The class *Experiment* can be used to run a model multiple times. As shown in Figure 1, it will start with the first parameter combination in the sample and repeat the simulation for the amount of defined iterations. After, that the same cycle is repeated for the next parameter combination.

Here is an example of an experiment with the model defined above. 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 rounded to integer. Every possible combination is repeated 2 times, which results in 50 runs:

```
exp = ap.Experiment(MyModel, sample, iterations=2, record=True)
results = exp.run()
```

For more applied examples of experiments, check out the demonstration models *Virus spread*, *Button network*, and *Forest fire*. An alternative to the built-in experiment class is to use AgentPy models with the EMA workbench (see *Exploratory modelling and analysis (EMA)*).

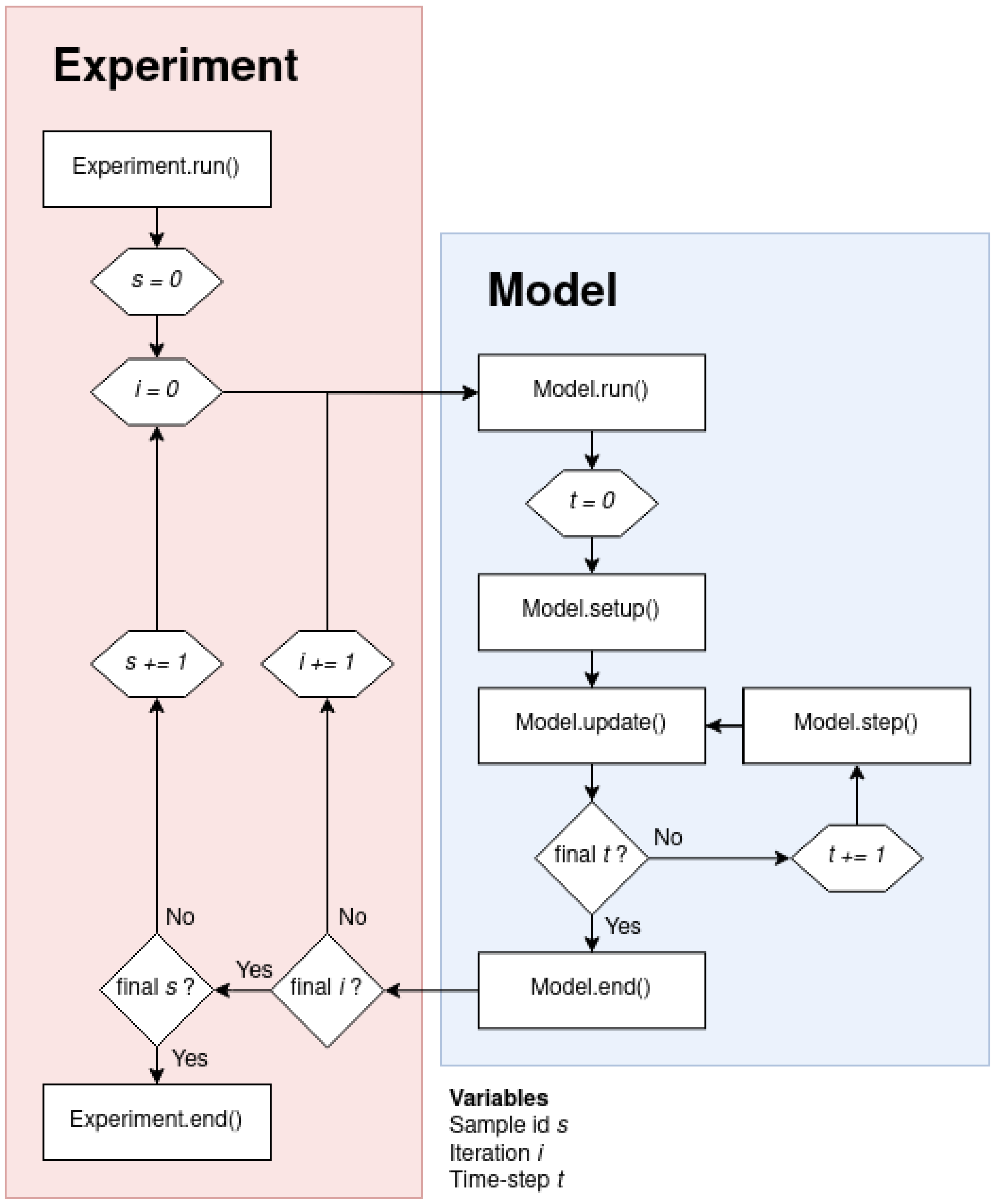


Fig. 1: Figure 1: Chain of events in `Model` and `Experiment`.

3.9 Random numbers

Model contains two random number generators:

- `Model.random` is an instance of `random.Random`
- `Model.nprandom` is an instance of `numpy.random.Generator`

The random seed for these generators can be set by defining a parameter *seed*. The *Sample* class has an argument *randomize* to control whether vary seeds over different parameter combinations. Similarly, *Experiment* also has an argument *randomize* to control whether to vary seeds over different iterations. More on this can be found in *Randomness and reproducibility*.

3.10 Data 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 {
'info': Dictionary with 5 keys
'parameters':
    'constants': Dictionary with 1 key
    'sample': DataFrame with 2 variables and 25 rows
'variables':
    'MyAgent': DataFrame with 1 variable and 10500 rows
'reporters': DataFrame with 1 variable and 50 rows
}
```

All data is given in a `pandas.DataFrame` and formatted as long-form data that can easily be used with statistical packages like seaborn. The output can contain the following categories of data:

- `info` 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.
- `reporters` holds evaluation measures that are documented only once per simulation.
- `sensitivity` holds calculated sensitivity measures.

The *DataDict* provides the following main methods to handle data:

- *DataDict.save()* and *DataDict.load()* can be used to store results.
- *DataDict.arrange()* generates custom combined dataframes.
- *DataDict.calc_sobol()* performs a Sobol sensitivity analysis.

3.11 Visualization

In addition to the *Interactive simulations*, AgentPy provides the following functions for visualization:

- `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 GUIDES

This section contains interactive notebooks with common applications of the agentpy framework. 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*. To learn how agentpy compares with other frameworks, take a look at *Comparison*.

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

4.1 Interactive simulations

The exploration of agent-based models can often be guided through an interactive simulation interface that allows users to visualize the models dynamics and adjust parameter values while a simulation is running. Examples are the traditional interface of NetLogo, or the browser-based visualization module of Mesa.

This guide shows how to create such interactive interfaces for agentpy models within a Jupyter Notebook by using the libraries IPySimulate, ipywidgets and d3.js. This approach is still in an early stage of development, and more features will follow in the future. Contributions are very welcome :)

[1]:
```
import agentpy as ap
import ipysimulate as ips

from ipywidgets import AppLayout
from agentpy.examples import WealthModel, SegregationModel
```

4.1.1 Lineplot

To begin we create an instance of the wealth transfer model (without parameters).

[2]:
```
model = WealthModel()
```

Parameters that are given as ranges will appear as interactive slider widgets. The parameter `fps` (frames per second) will be used automatically to indicate the speed of the simulation. The third value in the range defines the default position of the slider.

[3]:
```
parameters = {
    'agents': 1000,
    'steps': 100,
    'fps': ap.IntRange(1, 20, 5),
}
```

We then create an ipysimulate control panel with the model and our set of parameters. We further pass two variables `t` (time-steps) and `gini` to be displayed live during the simulation.

```
[4]: control = ips.Control(model, parameters, variables=('t', 'gini'))
```

Next, we create a lineplot of the variable `gini`:

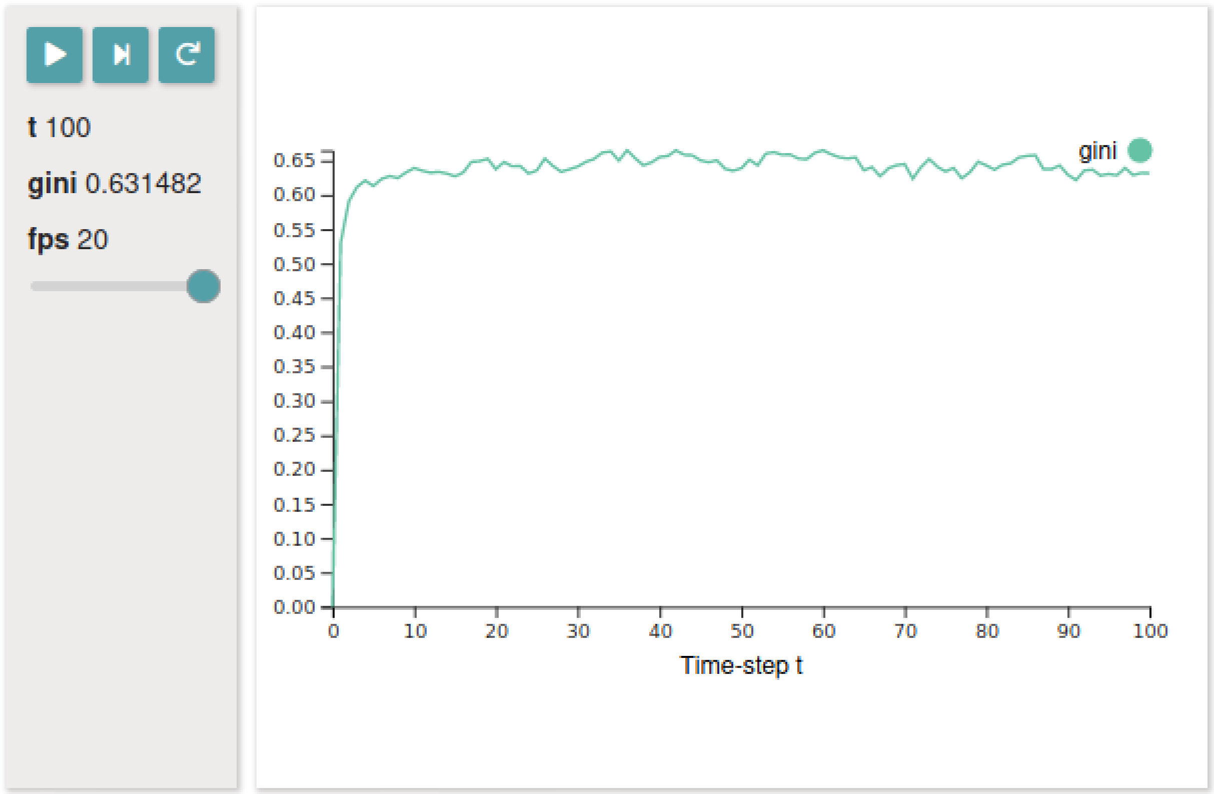
```
[5]: lineplot = ips.Lineplot(control, 'gini')
```

Finally, we want to display our two widgets `control` and `lineplot` next to each other. For this, we can use the layout templates from ipywidgets.

```
[6]: AppLayout(
         left_sidebar=control,
         center=lineplot,
         pane_widths=['125px', 1, 1],
         height='400px'
     )
```

```
AppLayout(children=(Control(layout=Layout(grid_area='left-sidebar'), parameters={'agents
↪': 1000, 'steps': 100,...
```

Note that this widget is not displayed interactively if viewed in the docs. To view the widget, please download the Jupyter Notebook at the top of this page or launch this notebook as a binder. Here is a screenshot of an interactive simulation:



4.1.2 Scatterplot

In this second demonstration, we create an instance of the segregation model:

[7]:
```
model = SegregationModel()
```

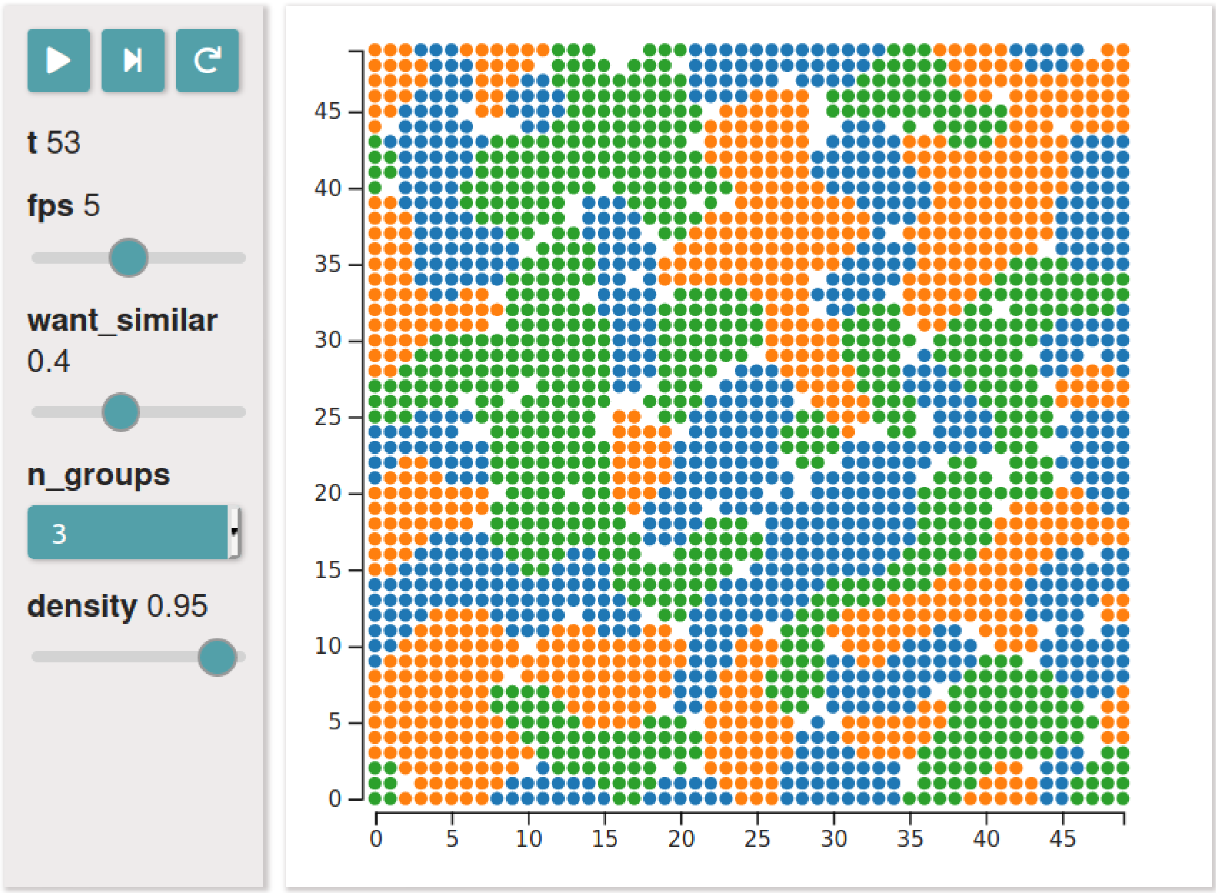
[8]:
```
parameters = {
    'fps': ap.IntRange(1, 10, 5),
    'want_similar': ap.Range(0, 1, 0.3),
    'n_groups': ap.Values(2, 3, 4),
    'density': ap.Range(0, 1, 0.95),
    'size': 50,
}
```

[9]:
```
control = ips.Control(model, parameters, ('t'))
scatterplot = ips.Scatterplot(
    control,
    xy=lambda m: m.grid.positions.values(),
    c=lambda m: m.agents.group
)
```

[10]:
```
AppLayout(left_sidebar=control,
          center=scatterplot,
          pane_widths=['125px', 1, 1],
          height='400px')
```

```
AppLayout(children=(Control(layout=Layout(grid_area='left-sidebar'), parameters={'fps':␣
↪5, 'want_similar': 0.3...
```

Note that this widget is not displayed interactively if viewed in the docs. To view the widget, please download the Jupyter Notebook at the top of this page or launch this notebook as a binder. Here is a screenshot of an interactive simulation:



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

4.2 Randomness and reproducibility

Random numbers and stochastic processes are essential to most agent-based models. Pseudo-random number generators can be used to create numbers in a sequence that appears random but is actually a deterministic sequence 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. Note that is possible that the generators will draw the same number repeatedly, as illustrated in this comic strip from Scott Adams:

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

4.2.1 Random number generators

Agentpy models contain two internal pseudo-random number generators with different features:

- `Model.random` is an instance of `random.Random` (more info here)
- `Model.nprandom` is an instance of `numpy.random.Generator` (more info here)

To illustrate, let us define a model that uses both generators to draw a random integer:

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

         def setup(self):
             self.x = self.random.randint(0, 99)
             self.y = self.nprandom.integers(99)
             self.report(['x', 'y'])
             self.stop()
```

If we run this model multiple times, we will likely get a different series of numbers in each iteration:

```
[3]: exp = ap.Experiment(RandomModel, iterations=5)
     results = exp.run()
```

```
Scheduled runs: 5
Completed: 5, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.027836
```

```
[4]: results.reporters
```

```
[4]:
                                                seed   x   y
iteration
0          163546198553218547629179155646693947592  75   1
1          248413101981860191382115517400004092470  57  61
2           71182126006424514048330534400698800795  96  37
3          319505356893330694850769146666666339584  89  95
4           64281825103124977892605409325092957646  37  84
```

4.2.2 Defining custom seeds

If we want the results to be reproducible, we can define a parameter `seed` that will be used automatically at the beginning of a simulation to initialize both generators.

```
[5]: parameters = {'seed': 42}
     exp = ap.Experiment(RandomModel, parameters, iterations=5)
     results = exp.run()
```

```
Scheduled runs: 5
Completed: 5, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.039785
```

By default, the experiment will use this seed to generate different random seeds for each iteration:

```
[6]: results.reporters
```

[6]:
```
                                           seed   x   y
iteration
0          252336560693540533935881068298825202077  26  68
1           47482295457342411543800303662309855831  70   9
2          252036172554514852379917073716435574953  58  66
3          200934189435493509245876840523779924304  48  77
4           318828394973076304960075763008606744 57  94  65
```

Repeating this experiment will yield the same results:

[7]:
```python
exp2 = ap.Experiment(RandomModel, parameters, iterations=5)
results2 = exp2.run()
```

```
Scheduled runs: 5
Completed: 5, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.047647
```

[8]:
```python
results2.reporters
```

[8]:
```
                                           seed   x   y
iteration
0          252336560693540533935881068298825202077  26  68
1           47482295457342411543800303662309855831  70   9
2          252036172554514852379917073716435574953  58  66
3          200934189435493509245876840523779924304  48  77
4           31882839497307630496007576300860674457  94  65
```

Alternatively, we can set the argument `randomize=False` so that the experiment will use the same seed for each iteration:

[9]:
```python
exp3 = ap.Experiment(RandomModel, parameters, iterations=5, randomize=False)
results3 = exp3.run()
```

```
Scheduled runs: 5
Completed: 5, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.021621
```

Now, each iteration yields the same results:

[10]:
```python
results3.reporters
```

[10]:
```
           seed   x   y
iteration
0            42  35  39
1            42  35  39
2            42  35  39
3            42  35  39
4            42  35  39
```

4.2.3 Sampling seeds

For a sample with multiple parameter combinations, we can treat the seed like any other parameter. The following example will use the same seed for each parameter combination:

```
[11]: parameters = {'p': ap.Values(0, 1), 'seed': 0}
      sample1 = ap.Sample(parameters, randomize=False)
      list(sample1)
```

```
[11]: [{'p': 0, 'seed': 0}, {'p': 1, 'seed': 0}]
```

If we run an experiment with this sample, the same iteration of each parameter combination will have the same seed (remember that the experiment will generate different seeds for each iteration by default):

```
[12]: exp = ap.Experiment(RandomModel, sample1, iterations=2)
      results = exp.run()
```

```
Scheduled runs: 4
Completed: 4, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.052923
```

```
[13]: results.reporters
```

```
[13]:
                                                          seed   x   y
sample_id iteration
0         0          302934307671667531413257853548643485645  68  31
          1          328530677494498397859470651507255972949  55  30
1         0          302934307671667531413257853548643485645  68  31
          1          328530677494498397859470651507255972949  55  30
```

Alternatively, we can use `Sample` with `randomize=True` (default) to generate random seeds for each parameter combination in the sample.

```
[14]: sample3 = ap.Sample(parameters, randomize=True)
      list(sample3)
```

```
[14]: [{'p': 0, 'seed': 302934307671667531413257853548643485645},
       {'p': 1, 'seed': 328530677494498397859470651507255972949}]
```

This will always generate the same set of random seeds:

```
[15]: sample3 = ap.Sample(parameters)
      list(sample3)
```

```
[15]: [{'p': 0, 'seed': 302934307671667531413257853548643485645},
       {'p': 1, 'seed': 328530677494498397859470651507255972949}]
```

An experiment will now have different results for every parameter combination and iteration:

```
[16]: exp = ap.Experiment(RandomModel, sample3, iterations=2)
      results = exp.run()
```

```
Scheduled runs: 4
Completed: 4, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.050806
```

[17]:
```
results.reporters
```

[17]:
```
                                                          seed   x   y
sample_id iteration
0         0          189926022767640608296581374469671322148  53  18
          1          179917731653904247792112551705722901296   3  60
1         0          255437819654147499963378822313666594855  83  62
          1           68871684356256783618296489618877951982  80  68
```

Repeating this experiment will yield the same results:

[18]:
```
exp = ap.Experiment(RandomModel, sample3, iterations=2)
results = exp.run()
```

```
Scheduled runs: 4
Completed: 4, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.037482
```

[19]:
```
results.reporters
```

[19]:
```
                                                          seed   x   y
sample_id iteration
0         0          189926022767640608296581374469671322148  53  18
          1          179917731653904247792112551705722901296   3  60
1         0          255437819654147499963378822313666594855  83  62
          1           68871684356256783618296489618877951982  80  68
```

4.2.4 Stochastic methods of AgentList

Let us now look at some stochastic operations that are often used in agent-based models. To start, we create a list of five agents:

[20]:
```
model = ap.Model()
agents = ap.AgentList(model, 5)
```

[21]:
```
agents
```

[21]:
```
AgentList (5 objects)
```

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

[22]:
```
agents.id
```

[22]:
```
[1, 2, 3, 4, 5]
```

To shuffle this list, we can use `AgentList.shuffle`:

[23]:
```
agents.shuffle().id
```

[23]:
```
[3, 2, 1, 4, 5]
```

To create a random subset, we can use `AgentList.random`:

[24]:
```
agents.random(3).id
```

[24]:
```
[2, 1, 4]
```

And if we want it to be possible to select the same agent more than once:

[25]:
```
agents.random(6, replace=True).id
```

[25]:
```
[5, 3, 2, 5, 2, 3]
```

4.2.5 Agent-specific generators

For more advanced applications, we can create separate generators for each object. We can ensure that the seeds of each object follow a controlled pseudo-random sequence by using the models' main generator to generate the seeds.

[26]:
```
class RandomAgent(ap.Agent):

    def setup(self):
        seed = self.model.random.getrandbits(128) # Seed from model
        self.random = random.Random(seed)  # Create agent generator
        self.x = self.random.random()  # Create a random number

class MultiRandomModel(ap.Model):

    def setup(self):
        self.agents = ap.AgentList(self, 2, RandomAgent)
        self.agents.record('x')
        self.stop()
```

[27]:
```
parameters = {'seed': 42}
exp = ap.Experiment(
    MultiRandomModel, parameters, iterations=2,
    record=True, randomize=False)
results = exp.run()
```

```
Scheduled runs: 2
Completed: 2, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.033219
```

[28]:
```
results.variables.RandomAgent
```

[28]:
```
                           x
iteration obj_id t
0         1      0  0.414688
          2      0  0.591608
1         1      0  0.414688
          2      0  0.591608
```

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

[29]:
```
class RandomAgent2(ap.Agent):
```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        self.random = random.Random(self.p.agent_seed)  # Create agent generator
        self.x = self.random.random()  # Create a random number

class MultiRandomModel2(ap.Model):

    def setup(self):
        self.agents = ap.AgentList(self, 2, RandomAgent2)
        self.agents.record('x')
        self.stop()
```

[30]:
```
parameters = {'agent_seed': 42}
exp = ap.Experiment(
    MultiRandomModel2, parameters, iterations=2,
    record=True, randomize=False)
results = exp.run()
```

```
Scheduled runs: 2
Completed: 2, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:00.033855
```

[31]:
```
results.variables.RandomAgent2
```

[31]:
```
                           x
iteration obj_id t
0         1      0  0.639427
          2      0  0.639427
1         1      0  0.639427
          2      0  0.639427
```

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

4.3 Exploratory modelling and analysis (EMA)

This guide shows how to use agentpy models together with the EMA Workbench. Similar to the agentpy `Experiment` class, this library can be used to perform experiments over different parameter combinations and multiple runs, but offers more advanced tools for parameter sampling and analysis with the aim to support decision making under deep uncertainty.

4.3.1 Converting an agentpy model to a function

Let us start by defining an agent-based model. Here, we use the wealth transfer model from the model library.

[1]:
```
import agentpy as ap
from agentpy.examples import WealthModel
```

To use the EMA Workbench, we need to convert our model to a function that takes each parameter as a keyword argument and returns a dictionary of the recorded evaluation measures.

[2]:
```
wealth_model = WealthModel.as_function()
```

[3]:
```
help(wealth_model)
```

```
Help on function agentpy_model_as_function in module agentpy.model:

agentpy_model_as_function(**kwargs)
    Performs a simulation of the model 'WealthModel'.

    Arguments:
        **kwargs: Keyword arguments with parameter values.

    Returns:
        dict: Reporters of the model.
```

Let us test out this function:

[4]:
```
wealth_model(agents=5, steps=5)
```

[4]:
```
{'gini': 0.32}
```

4.3.2 Using the EMA Workbench

Here is an example on how to set up an experiment with the EMA Workbench. For more information, please visit the documentation of EMA Workbench.

[9]:
```
from ema_workbench import (IntegerParameter, Constant, ScalarOutcome,
                           Model, perform_experiments, ema_logging)
```

[6]:
```
if __name__ == '__main__':

    ema_logging.LOG_FORMAT = '%(message)s'
    ema_logging.log_to_stderr(ema_logging.INFO)

    model = Model('WealthModel', function=wealth_model)
    model.uncertainties = [IntegerParameter('agents', 10, 100)]
    model.constants = [Constant('steps', 100)]
    model.outcomes = [ScalarOutcome('gini')]

    results = perform_experiments(model, 100)
```

```
performing 100 scenarios * 1 policies * 1 model(s) = 100 experiments
performing experiments sequentially
10 cases completed
20 cases completed
30 cases completed
40 cases completed
50 cases completed
60 cases completed
70 cases completed
80 cases completed
90 cases completed
100 cases completed
experiments finished
```

[7]:
```
results[0]
```

[7]:
```
    agents scenario policy        model
0     70.0        0   None  WealthModel
1     44.0        1   None  WealthModel
2     77.0        2   None  WealthModel
3     87.0        3   None  WealthModel
4     51.0        4   None  WealthModel
..     ...      ...    ...          ...
95    38.0       95   None  WealthModel
96    26.0       96   None  WealthModel
97    59.0       97   None  WealthModel
98    94.0       98   None  WealthModel
99    75.0       99   None  WealthModel

[100 rows x 4 columns]
```

[10]:
```
results[1]
```

[10]:
```
{'gini': array([0.67877551, 0.61880165, 0.6392309 , 0.62491743, 0.65820838,
        0.62191358, 0.61176471, 0.66986492, 0.6134068 , 0.63538062,
        0.69958848, 0.63777778, 0.61862004, 0.6786    , 0.6184424 ,
        0.61928474, 0.6446281 , 0.6358    , 0.7283737 , 0.60225922,
        0.6404321 , 0.59729448, 0.63516068, 0.515     , 0.58301785,
        0.66780045, 0.6321607 , 0.58131488, 0.6201873 , 0.70083247,
        0.7       , 0.58666667, 0.58131382, 0.5964497 , 0.56014692,
        0.6446281 , 0.59146814, 0.70919067, 0.61592693, 0.59736561,
        0.52623457, 0.64604402, 0.56790123, 0.65675193, 0.49905482,
        0.55250979, 0.62606626, 0.49864792, 0.63802469, 0.62722222,
        0.65500945, 0.69010417, 0.64160156, 0.67950052, 0.60207612,
        0.63115111, 0.64246914, 0.65162722, 0.65759637, 0.66392948,
        0.63971072, 0.57375   , 0.55310287, 0.58692476, 0.59410431,
        0.61950413, 0.6228125 , 0.52444444, 0.59119898, 0.63180975,
        0.6592    , 0.6540149 , 0.60133914, 0.67884977, 0.57852447,
        0.58739596, 0.52040816, 0.52077562, 0.66304709, 0.59750567,
        0.57692308, 0.65189289, 0.64697266, 0.68507561, 0.66874582,
        0.67857143, 0.59410431, 0.55953251, 0.63651717, 0.62809917,
        0.61111111, 0.6328    , 0.64003673, 0.65140479, 0.65972222,
        0.62465374, 0.65384615, 0.64464234, 0.61588954, 0.63111111])}
```

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

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

# Visualization
import seaborn as sns
```

5.1.1 About the model

The model explores the distribution of wealth under a trading population of agents. Each agent starts with one unit of wealth. During each time-step, each agents with positive wealth randomly selects a trading partner and gives them one unit of their wealth. We will see that this 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 Model definition

We start by defining a new type of `Agent` with the following methods:

- `setup()` is called automatically when a new agent is created and initializes a variable `wealth`.
- `wealth_transfer()` describes the agent's behavior at every time-step and will be called by the model.

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

    """ An agent with wealth """
```

(continues on next page)

(continued from previous page)

```
    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.

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

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

        x = np.array(x)
        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 <https://agentpy.readthedocs.io/en/stable/reference_models.html>`__ `` with the following methods:

- `setup` defines how many agents should be created at the beginning of the simulation.
- `step` calls all agents during each time-step to perform their `wealth_transfer` method.
- `update` calculates and record the current Gini coefficient after each time-step.
- `end`, which is called at the end of the simulation, we record the wealth of each agent.

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

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

        def setup(self):

            self.agents = ap.AgentList(self, 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')
```

5.1.3 Simulation run

To prepare, we define parameter dictionary with a random seed, the number of agents, and the number of time-steps.

[5]:
```
parameters = {
    'agents': 100,
    'steps': 100,
    'seed': 42,
}
```

To perform a simulation, we initialize our model with a given set of parameters and call `Model.run() <https://agentpy.readthedocs.io/en/stable/reference_models.html>`__.

[6]:
```
model = WealthModel(parameters)
results = model.run()
```

```
Completed: 100 steps
Run time: 0:00:00.124199
Simulation finished
```

5.1.4 Output analysis

The simulation returns a `DataDict <https://agentpy.readthedocs.io/en/stable/reference_output.html>`__ with our recorded variables.

[7]:
```
results
```

[7]:
```
DataDict {
'info': Dictionary with 9 keys
'parameters':
    'constants': Dictionary with 3 keys
'variables':
    'WealthModel': DataFrame with 1 variable and 101 rows
    'WealthAgent': DataFrame with 1 variable and 100 rows
}
```

The output's `info` provides general information about the simulation.

[8]:
```
results.info
```

[8]:
```
{'model_type': 'WealthModel',
 'time_stamp': '2021-05-28 09:33:50',
 'agentpy_version': '0.0.8.dev0',
 'python_version': '3.8.5',
 'experiment': False,
 'completed': True,
 'created_objects': 100,
 'completed_steps': 100,
 'run_time': '0:00:00.124199'}
```

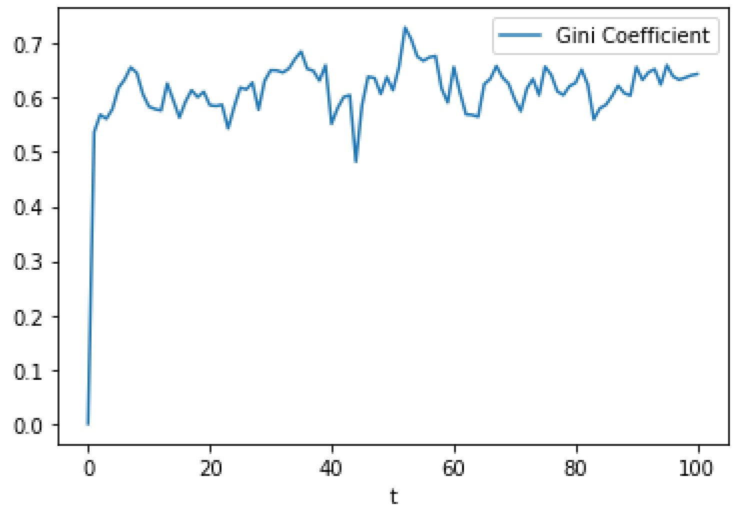
To explore the evolution of inequality, we look at the recorded `DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>`__ of the model's variables.

[9]:
```
results.variables.WealthModel.head()
```

[9]:
```
   Gini Coefficient
t
0            0.0000
1            0.5370
2            0.5690
3            0.5614
4            0.5794
```

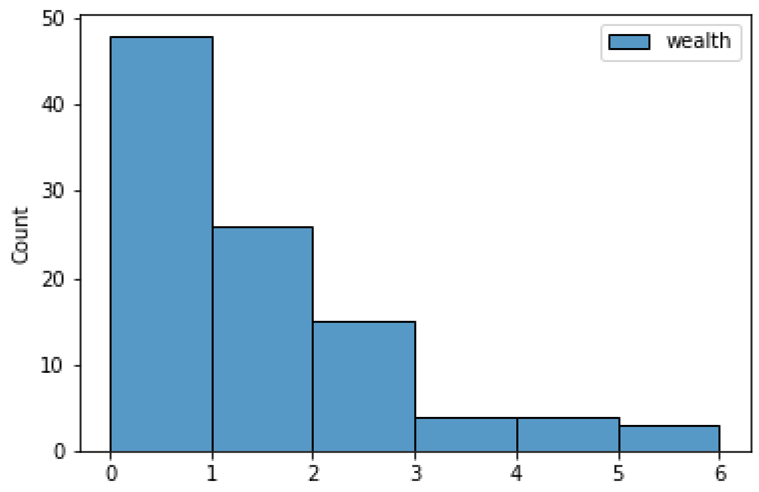
To visualize this data, we can use `` `DataFrame.plot <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html>`__ ``.

[10]:
```
data = results.variables.WealthModel
ax = data.plot()
```



To look at the distribution at the end of the simulation, we visualize the recorded agent variables with seaborn.

[11]:
```
sns.histplot(data=results.variables.WealthAgent, binwidth=1);
```



The result resembles a Boltzmann distribution.

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. """
        rng = self.model.random
        for n in self.network.neighbors(self):
            if n.condition == 0 and self.p.infection_chance > rng.random():
                n.condition = 1  # Infect susceptible peer
        if self.p.recovery_chance > rng.random():
            self.condition = 2  # Recover from infection
```

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

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

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

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

        # Create agents and network
        self.agents = ap.AgentList(self, self.p.population, Person)
        self.network = self.agents.network = ap.Network(self, graph)
        self.network.add_agents(self.agents, self.network.nodes)

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

    def update(self):
        """ Record variables after setup and each step. """

        # Record share of agents with each condition
        for i, c in enumerate(('S', 'I', 'R')):
            n_agents = len(self.agents.select(self.agents.condition == i))
            self[c] = n_agents / self.p.population
            self.record(c)

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

    def step(self):
        """ Define the models' events per simulation step. """

        # Call 'being_sick' for infected agents
        self.agents.select(self.agents.condition == 1).being_sick()

    def end(self):
        """ Record evaluation measures at the end of the simulation. """

        # Record final evaluation measures
        self.report('Total share infected', self.I + self.R)
        self.report('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_infection_share': 0.1,
    'number_of_neighbors': 2,
    'network_randomness': 0.5
}

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

```
Completed: 77 steps
Run time: 0:00:00.152576
Simulation finished
```

5.2.4 Analyzing results

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

[5]:
```
results
```

[5]:
```
DataDict {
'info': Dictionary with 9 keys
'parameters':
    'constants': Dictionary with 6 keys
'variables':
    'VirusModel': DataFrame with 3 variables and 78 rows
'reporters': DataFrame with 2 variables and 1 row
}
```

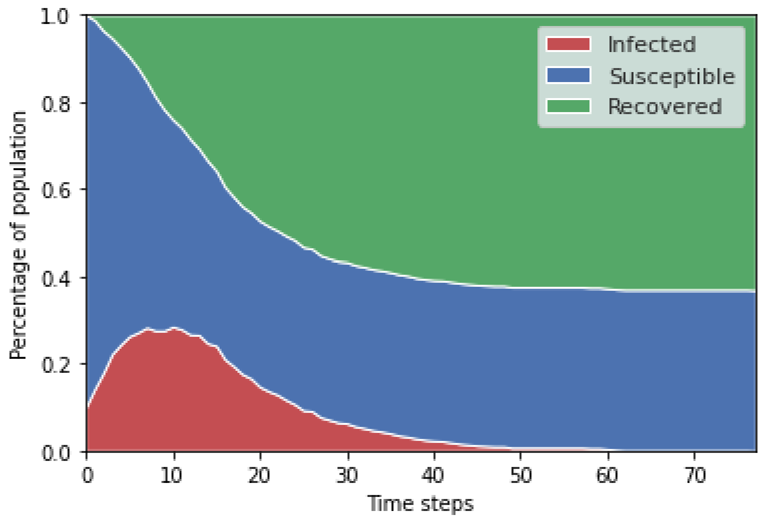
To visualize the evolution of our variables over time, we create a plot function.

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

    ax.legend()
    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")
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
virus_stackplot(results.variables.VirusModel, 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.VirusModel, 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.network.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 Multi-run experiment

To explore the effect of different parameter values, we use the classes `Sample`, `Range`, and `IntRange` to create a sample of different parameter combinations.

[9]:
```
parameters = {
    'population': ap.IntRange(100, 1000),
    'infection_chance': ap.Range(0.1, 1.),
    'recovery_chance': ap.Range(0.1, 1.),
    'initial_infection_share': 0.1,
    'number_of_neighbors': 2,
    'network_randomness': ap.Range(0., 1.)
}

sample = ap.Sample(
    parameters,
    n=128,
    method='saltelli',
    calc_second_order=False
)
```

We then create an `Experiment` that takes a model and sample as input. `Experiment.run()` runs our model repeatedly over the whole sample with ten random iterations per parameter combination.

[10]:
```
exp = ap.Experiment(VirusModel, sample, iterations=10)
results = exp.run()
```

```
Scheduled runs: 7680
Completed: 7680, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:04:55.800449
```

Optionally, we can save and load our results as follows:

[11]:
```
results.save()
```

```
Data saved to ap_output/VirusModel_1
```

[12]:
```
results = ap.DataDict.load('VirusModel')
```

```
Loading from directory ap_output/VirusModel_1/
Loading parameters_constants.json - Successful
Loading parameters_sample.csv - Successful
Loading parameters_log.json - Successful
Loading reporters.csv - Successful
Loading info.json - Successful
```

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

[13]:
```
results
```

[13]:
```
DataDict {
'parameters':
    'constants': Dictionary with 2 keys
    'sample': DataFrame with 4 variables and 768 rows
    'log': Dictionary with 5 keys
```

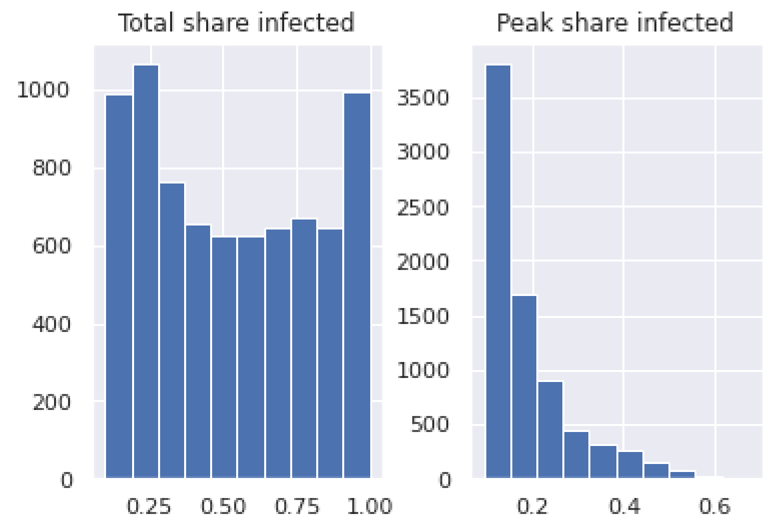
(continues on next page)

(continued from previous page)

```
'reporters': DataFrame with 2 variables and 7680 rows
'info': Dictionary with 12 keys
}
```

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

[14]:
```
results.reporters.hist();
```



5.2.7 Sensitivity analysis

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

[15]:
```
results.calc_sobol()
```

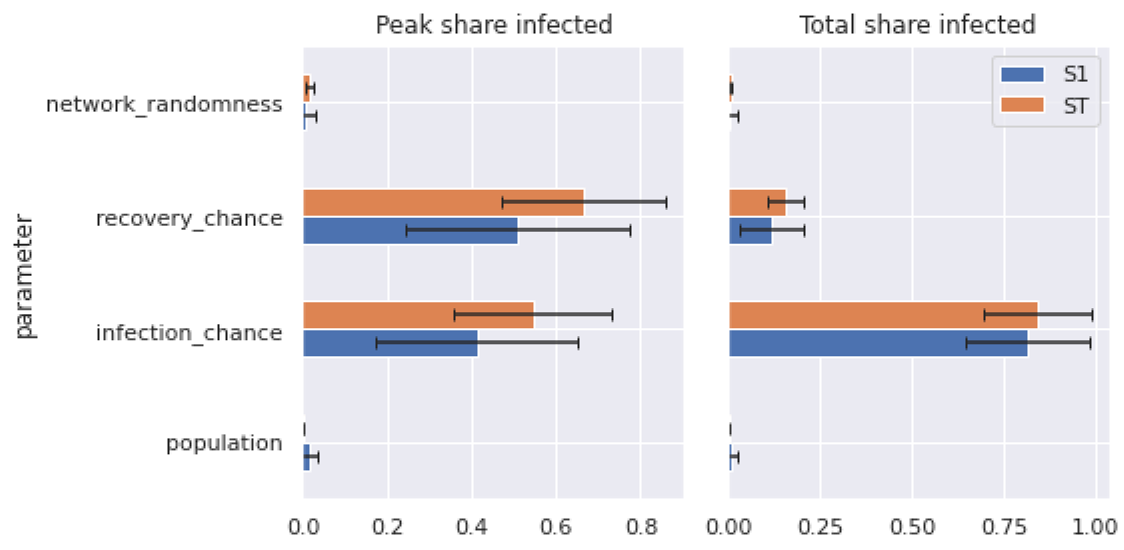
[15]:
```
DataDict {
'parameters':
    'constants': Dictionary with 2 keys
    'sample': DataFrame with 4 variables and 768 rows
    'log': Dictionary with 5 keys
'reporters': DataFrame with 2 variables and 7680 rows
'info': Dictionary with 12 keys
'sensitivity':
    'sobol': DataFrame with 2 variables and 8 rows
    'sobol_conf': DataFrame with 2 variables and 8 rows
}
```

This adds a new category *sensitivity* to our results, which includes:

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

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

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

          sns.set()
          fig, axs = plt.subplots(1, 2, figsize=(8, 4))
          si_list = results.sensitivity.sobol.groupby(by='reporter')
          si_conf_list = results.sensitivity.sobol_conf.groupby(by='reporter')

          for (key, si), (_, err), ax in zip(si_list, si_conf_list, axs):
              si = si.droplevel('reporter')
              err = err.droplevel('reporter')
              si.plot.barh(xerr=err, title=key, ax=ax, capsize = 3)
              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)
```



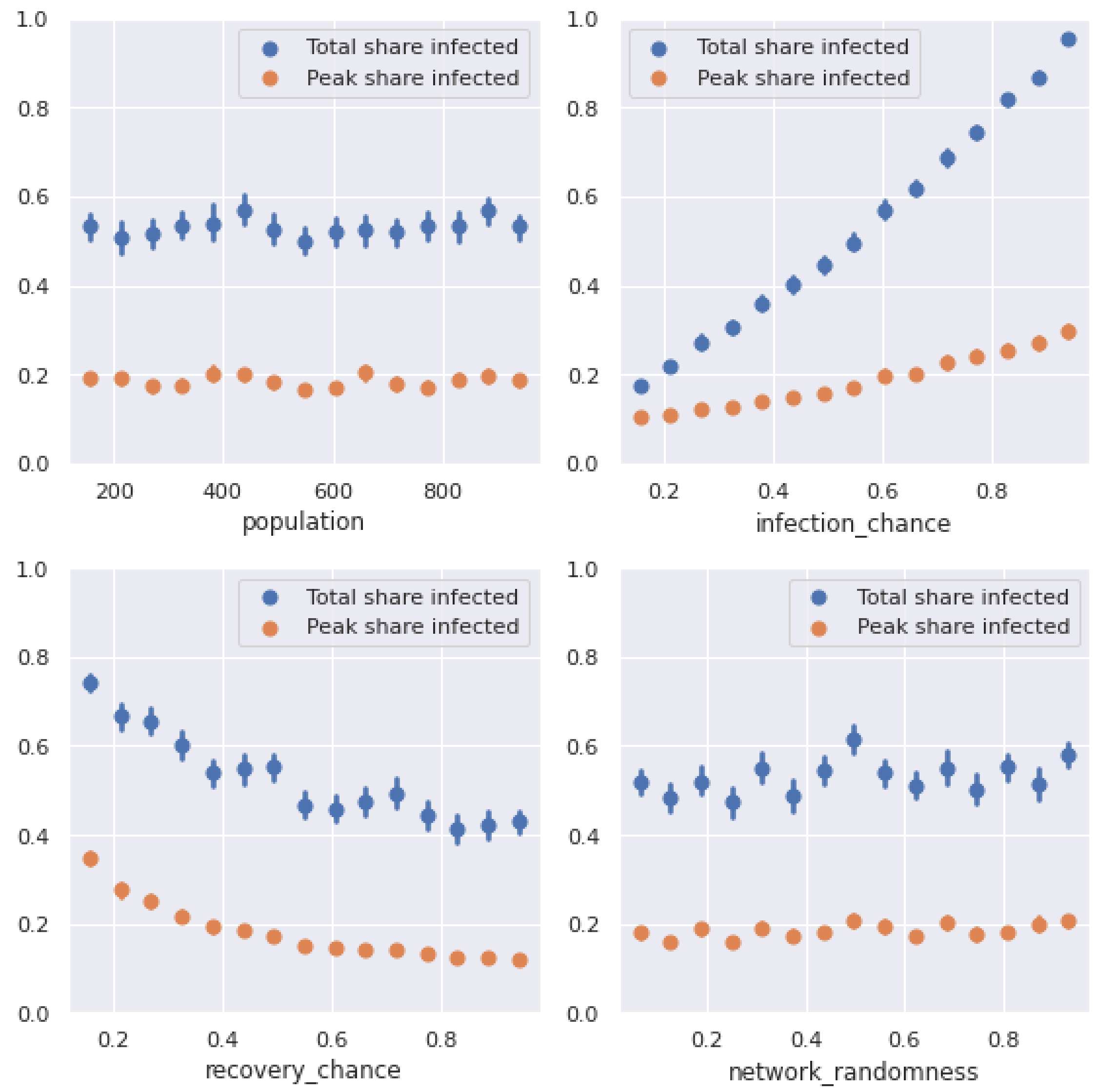
Alternatively, we can also display sensitivities by plotting average evaluation measures over our parameter variations.

```
[17]: 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_reporters()
          params = results.parameters.sample.keys()

          for x, ax in zip(params, axs):
              for y in results.reporters.columns:
```

(continues on next page)

(continued from previous page)

```
            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 Flocking behavior

This notebook presents an agent-based model that simulates the flocking behavior of animals. It demonstrates how to use the agentpy package for models with a continuous space with two or three dimensions.

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

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

5.3.1 About the model

The boids model was invented by Craig Reynolds, who describes it as follows:

> In 1986 I made a computer model of coordinated animal motion such as bird flocks and fish schools. It was based on three dimensional computational geometry of the sort normally used in computer animation or computer aided design. I called the generic simulated flocking creatures boids. The basic flocking model consists of three simple steering behaviors which describe how an individual boid maneuvers based on the positions and velocities its nearby flockmates: - Separation: steer to avoid crowding local flockmates - Alignment: steer towards the average heading of local flockmates - Cohesion: steer to move toward the average position of local flockmates

The model presented here is a simplified implementation of this algorithm, following the Boids Pseudocode written by Conrad Parker.

If you want to see a real-world example of flocking behavior, check out this fascinating video of Starling murmurations from National Geographic:

```
[2]: IPython.display.YouTubeVideo('V4f_1_r80RY', width=600, height=350)
```

[2]:

5.3.2 Model definition

The Boids model is based on two classes, one for the agents, and one for the overall model. For more information about this structure, take a look at the creating models.

Each agent starts with a random position and velocity, which are implemented as numpy arrays. The position is defined through the space environment, which the agent can access via `Agent.position()` and `Agent.neighbors()`.

The methods `update_velocity()` and `update_position()` are separated so that all agents can update their velocity before the actual movement takes place. For more information about the algorithm in `update_velocity()`, check out the Boids Pseudocode.

[3]:
```python
def normalize(v):
    """ Normalize a vector to length 1. """
    norm = np.linalg.norm(v)
    if norm == 0:
        return v
    return v / norm
```

[4]:
```python
class Boid(ap.Agent):
    """ An agent with a position and velocity in a continuous space,
    who follows Craig Reynolds three rules of flocking behavior;
    plus a fourth rule to avoid the edges of the simulation space. """

    def setup(self):

        self.velocity = normalize(
            self.model.nprandom.random(self.p.ndim) - 0.5)

```

(continues on next page)

(continued from previous page)

```
    def setup_pos(self, space):

        self.space = space
        self.neighbors = space.neighbors
        self.pos = space.positions[self]

    def update_velocity(self):

        pos = self.pos
        ndim = self.p.ndim

        # Rule 1 - Cohesion
        nbs = self.neighbors(self, distance=self.p.outer_radius)
        nbs_len = len(nbs)
        nbs_pos_array = np.array(nbs.pos)
        nbs_vec_array = np.array(nbs.velocity)
        if nbs_len > 0:
            center = np.sum(nbs_pos_array, 0) / nbs_len
            v1 = (center - pos) * self.p.cohesion_strength
        else:
            v1 = np.zeros(ndim)

        # Rule 2 - Seperation
        v2 = np.zeros(ndim)
        for nb in self.neighbors(self, distance=self.p.inner_radius):
            v2 -= nb.pos - pos
        v2 *= self.p.seperation_strength

        # Rule 3 - Alignment
        if nbs_len > 0:
            average_v = np.sum(nbs_vec_array, 0) / nbs_len
            v3 = (average_v - self.velocity) * self.p.alignment_strength
        else:
            v3 = np.zeros(ndim)

        # Rule 4 - Borders
        v4 = np.zeros(ndim)
        d = self.p.border_distance
        s = self.p.border_strength
        for i in range(ndim):
            if pos[i] < d:
                v4[i] += s
            elif pos[i] > self.space.shape[i] - d:
                v4[i] -= s

        # Update velocity
        self.velocity += v1 + v2 + v3 + v4
        self.velocity = normalize(self.velocity)

    def update_position(self):

        self.space.move_by(self, self.velocity)
```

[5]:
```python
class BoidsModel(ap.Model):
    """
    An agent-based model of animals' flocking behavior,
    based on Craig Reynolds' Boids Model [1]
    and Conrad Parkers' Boids Pseudocode [2].

    [1] http://www.red3d.com/cwr/boids/
    [2] http://www.vergenet.net/~conrad/boids/pseudocode.html
    """

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

        self.space = ap.Space(self, shape=[self.p.size]*self.p.ndim)
        self.agents = ap.AgentList(self, self.p.population, Boid)
        self.space.add_agents(self.agents, random=True)
        self.agents.setup_pos(self.space)

    def step(self):
        """ Defines the models' events per simulation step. """

        self.agents.update_velocity()  # Adjust direction
        self.agents.update_position()  # Move into new direction
```

5.3.3 Visualization functions

Next, we define a plot function that can take our model and parameters as an input and creates an animated plot with `animate()`:

[6]:
```python
def animation_plot_single(m, ax):
    ndim = m.p.ndim
    ax.set_title(f"Boids Flocking Model {ndim}D t={m.t}")
    pos = m.space.positions.values()
    pos = np.array(list(pos)).T  # Transform
    ax.scatter(*pos, s=1, c='black')
    ax.set_xlim(0, m.p.size)
    ax.set_ylim(0, m.p.size)
    if ndim == 3:
        ax.set_zlim(0, m.p.size)
    ax.set_axis_off()

def animation_plot(m, p):
    projection = '3d' if p['ndim'] == 3 else None
    fig = plt.figure(figsize=(7,7))
    ax = fig.add_subplot(111, projection=projection)
    animation = ap.animate(m(p), fig, ax, animation_plot_single)
    return IPython.display.HTML(animation.to_jshtml(fps=20))
```

5.3.4 Simulation (2D)

To run a simulation, we define a dictionary with our parameters:

[7]:
```
parameters2D = {
    'size': 50,
    'seed': 123,
    'steps': 200,
    'ndim': 2,
    'population': 200,
    'inner_radius': 3,
    'outer_radius': 10,
    'border_distance': 10,
    'cohesion_strength': 0.005,
    'seperation_strength': 0.1,
    'alignment_strength': 0.3,
    'border_strength': 0.5
}
```

We can now display our first animation with two dimensions:

[8]:
```
animation_plot(BoidsModel, parameters2D)
```

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

5.3.5 Simulation (3D)

Finally, we can do the same with three dimensions, a larger number of agents, and a bit more space:

[9]:
```
new_parameters = {
    'ndim': 3,
    'population': 1000,
    'wall_avoidance_distance': 5,
    'wall_avoidance_strength': 0.3
}

parameters3D = dict(parameters2D)
parameters3D.update(new_parameters)

animation_plot(BoidsModel, parameters3D)
```

[9]:
```
<IPython.core.display.HTML object>
```

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

5.4 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

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

5.4.1 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.

5.4.2 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):
        """ Initiate agent attributes. """
        self.grid = self.model.grid
        self.random = self.model.random
        self.group = self.random.choice(range(self.p.n_groups))
        self.share_similar = 0
        self.happy = False

    def update_happiness(self):
        """ Be happy if rate of similar neighbors is high enough. """
        neighbors = self.grid.neighbors(self)
        similar = len([n for n in neighbors if n.group == self.group])
        ln = len(neighbors)
        self.share_similar = similar / ln if ln > 0 else 0
        self.happy = self.share_similar >= self.p.want_similar

    def find_new_home(self):
        """ Move to random free spot and update free spots. """
        new_spot = self.random.choice(self.model.grid.empty)
        self.grid.move_to(self, new_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]:
```python
class SegregationModel(ap.Model):

    def setup(self):

        # Parameters
        s = self.p.size
        n = self.n = int(self.p.density * (s ** 2))

        # Create grid and agents
        self.grid = ap.Grid(self, (s, s), track_empty=True)
        self.agents = ap.AgentList(self, n, Person)
        self.grid.add_agents(self.agents, random=True, empty=True)

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

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

    def step(self):
        # Move unhappy people to new location
        self.unhappy.find_new_home()

    def get_segregation(self):
        # Calculate average percentage of similar neighbors
        return round(sum(self.agents.share_similar) / self.n, 2)

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

5.4.3 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 define a 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
    }
```

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

[5]:
```
def animation_plot(model, ax):
    group_grid = model.grid.attr_grid('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()
model = SegregationModel(parameters)
animation = ap.animate(model, fig, ax, animation_plot)
IPython.display.HTML(animation.to_jshtml())
```

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

5.4.4 Interactive simulation

An interactive simulation of this model can be found in this guide.

5.4.5 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]:
```
parameters_multi = dict(parameters)
parameters_multi.update({
    'want_similar': ap.Values(0,0.125, 0.25, 0.375, 0.5, 0.625),
    'density': ap.Values(0.5, 0.7, 0.95),
})
sample = ap.Sample(parameters_multi)
```

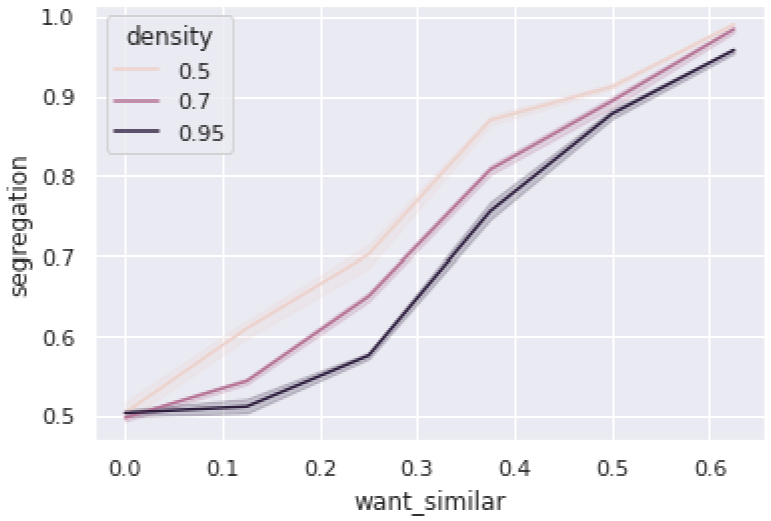
We now run an experiment where we simulate each parameter combination in our sample over 5 iterations.

[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: 0:00:56.914258
```

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]: sns.set_theme()
sns.lineplot(
    data=results.arrange_reporters(),
    x='want_similar',
    y='segregation',
    hue='density'
);
```



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

5.5 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

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

5.5.1 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.”

5.5.2 Model definition

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

    def setup(self):

        # Create agents (trees)
        n_trees = int(self.p['Tree density'] * (self.p.size**2))
        trees = self.agents = ap.AgentList(self, n_trees)

        # Create grid (forest)
        self.forest = ap.Grid(self, [self.p.size]*2, track_empty=True)
        self.forest.add_agents(trees, random=True, empty=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 = self.forest.agents[0:self.p.size, 0:2]
        unfortunate_trees.condition = 1

    def step(self):

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

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

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

(continues on next page)

(continued from previous page)

```
    def end(self):

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

5.5.3 Single-run animation

[3]:
```
# Define parameters

parameters = {
    'Tree density': 0.6, # Percentage of grid covered by trees
    'size': 50, # Height and length of the grid
    'steps': 100,
}
```

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

def animation_plot(model, ax):
    attr_grid = model.forest.attr_grid('condition')
    color_dict = {0:'#7FC97F', 1:'#d62c2c', 2:'#e5e5e5', None:'#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(fps=15))
```

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

5.5.4 Parameter sweep

[5]:
```
# Prepare parameter sample
parameters = {
    'Tree density': ap.Range(0.2, 0.6),
    'size': 100
}
sample = ap.Sample(parameters, n=30)
```

[6]:
```
# Perform experiment
exp = ap.Experiment(ForestModel, sample, iterations=40)
results = exp.run()
```

```
Scheduled runs: 1200
Completed: 1200, estimated time remaining: 0:00:00
```

(continues on next page)

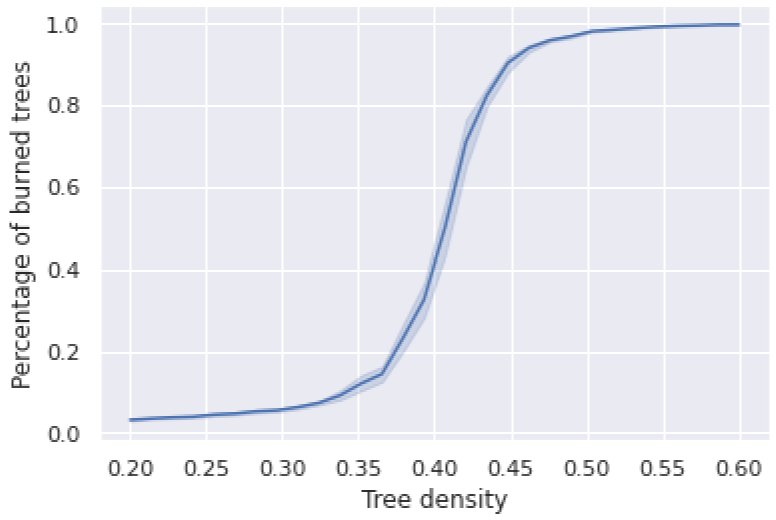
(continued from previous page)

```
Experiment finished
Run time: 0:04:23.286950
```

[7]:
```
# Save and load data
results.save()
results = ap.DataDict.load('ForestModel')
```

```
Data saved to ap_output/ForestModel_1
Loading from directory ap_output/ForestModel_1/
Loading parameters_constants.json - Successful
Loading parameters_sample.csv - Successful
Loading parameters_log.json - Successful
Loading reporters.csv - Successful
Loading info.json - Successful
```

[8]:
```
# Plot sensitivity
sns.set_theme()
sns.lineplot(
    data=results.arrange_reporters(),
    x='Tree density',
    y='Percentage of burned trees'
);
```



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

5.6 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

# Visualization
import seaborn as sns
```

5.6.1 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."

5.6.2 Model definition

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

    def setup(self):

        # Create a graph with n agents
        self.buttons = ap.Network(self)
        self.agents = ap.AgentList(self, self.p.n)
        self.buttons.add_agents(self.agents)
        self.agents.node = self.buttons.nodes
        self.threads = 0

```

(continues on next page)

(continued from previous page)

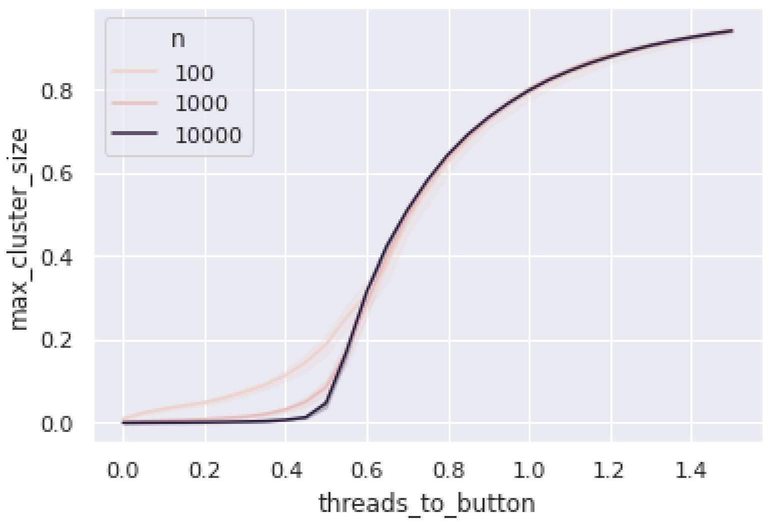
```
    def update(self):

        # 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).node)
            self.threads += 1
```

5.6.3 Multi-run experiment

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

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

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

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

```
Scheduled runs: 75
Completed: 75, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:00:36.012666
```

[4]:
```
# Plot averaged time-series for discrete parameter samples
sns.set_theme()
sns.lineplot(
    data=results.arrange_variables(),
    x='threads_to_button',
    y='max_cluster_size',
    hue='n'
);
```


n
100
1000
10000
max_cluster_size
0.0
0.2
0.4
0.6
0.8
threads_to_button
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

CHAPTER

SIX

API REFERENCE

6.1 Agent-based models

The `Model` contains all objects and defines the procedures of an agent-based simulation. It is meant as a template for custom model classes that override the *custom procedure methods*.

class Model(*parameters=None*, *_run_id=None*, ***kwargs*)

Template of an agent-based model.

Parameters

- **parameters** (`dict, optional`) – Dictionary of the model's parameters. Default values will be selected from entries of type `Range`, `IntRange`, and `Values`. The following parameters will be used automatically:
  - steps (int, optional): Defines the maximum number of time-steps. If none is passed, there will be no step limit.
  - seed (int, optional): Used to initiate the model's random number generators. If none is passed, a random seed will be generated.
  - report_seed (bool, optional): Whether to document the random seed used (default True).
- ****kwargs** – Will be forwarded to `Model.setup()`.

Variables

- **type** (`str`) – The model's class name.
- **info** (`InfoStr`) – Information about the model's current state.
- **p** (`AttrDict`) – The model's parameters.
- **t** (`int`) – Current time-step of the model.
- **id** (`int`) – The model's object id, which will always be zero.
- **random** (`random.Random`) – Random number generator.
- **nprandom** (`numpy.random.Generator`) – Numpy random number generator.
- **var_keys** (`list`) – Names of the model's custom variables.
- **running** (`bool`) – Indicates whether the model is currently running.
- **log** (`dict`) – The model's recorded variables.
- **reporters** (`dict`) – The model's documented reporters.
- **output** (`DataDict`) – Output data after a completed simulation.

Examples

To define a custom model with a custom agent type:

```
class MyAgent(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

class MyModel(ap.Model):

    def setup(self):
        # Called at the start of the simulation
        self.agents = ap.AgentList(self, self.p.agents, MyAgent)

    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 variable

    def end(self):
        # Called at the end of the simulation
        self.report('my_reporter', 1)  # Report a simulation result
```

To run a simulation:

```
parameters = {
    'my_parameter': 42,
    'agents': 10,
    'steps': 10  # Used automatically to define simulation length
}

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

6.1.1 Simulation tools

`Model.`**`run`**(*steps=None*, *seed=None*, *display=True*)

Executes the simulation of the model. Can also be used to continue a partly-run simulation for a given number of additional steps.

It starts by calling `Model.run_setup()` and then calls `Model.run_step()` until the method *`Model.stop()`* is called or *steps* is reached. After that, *`Model.end()`* and `Model.create_output()` are called. The simulation results can be found in `Model.output`.

Parameters

- **steps** (*int, optional*) – Number of (additional) steps for the simulation to run. If passed, the parameter 'Model.p.steps' will be ignored. The simulation can still be stopped with :func:'Model.stop'.
- **seed** (*int, optional*) – Seed to initialize the model's random number generators. If none is given, the parameter 'Model.p.seed' is used. If there is no such parameter, a random seed will be used. For a partly-run simulation, this argument will be ignored.
- **display** (*bool, optional*) – Whether to display simulation progress (default True).

Returns Recorded variables and reporters.

Return type *DataDict*

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

6.1.2 Custom procedures

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

`Model.step()`
Defines the model's actions during each simulation step (excluding *t==0*). Can be overwritten to define the models' main dynamics.

`Model.update()`
Defines the model's actions after each simulation step (including *t==0*). Can be overwritten for the recording of dynamic variables.

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

6.1.3 Data collection

`Model.record(`*var_keys*, *value=None*`)`
Records an object's variables at the current time-step. Recorded variables can be accessed via the object's *log* attribute and will be saved to the model's output at the end of a simulation.

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.

Notes

Recording mutable objects like lists can lead to wrong results if the object's content will be changed during the simulation. Make a copy of the list or record each list entry seperately.

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.vars)
```

`Model.`**`report`**(*rep_keys*, *value=None*)

Reports a new simulation result. Reporters are meant to be 'summary statistics' or 'evaluation measures' of the simulation as a whole, and only one value can be stored per run. In comparison, variables that are recorded with *`Model.record()`* can be recorded multiple times for each time-step and object.

Parameters

- **rep_keys** (*str or list of str*) – Name(s) of the reporter(s) to be documented.
- **value** (*int or float, optional*) – Value to be reported. The same value will be used for all *rep_keys*. If none is given, the values of object attributes with the same name as each rep_key will be used.

Examples

Store a reporter *x* with a value *42*:

```
model.report('x', 42)
```

Define a custom model that stores a reporter *sum_id* with the sum of all agent ids at the end of the simulation:

```
class MyModel(ap.Model):
    def setup(self):
        agents = ap.AgentList(self, self.p.agents)
    def end(self):
        self.report('sum_id', sum(self.agents.id))
```

Running an experiment over different numbers of agents for this model yields the following datadict of reporters:

```
>>> sample = ap.sample({'agents': (1, 3)}, 3)
>>> exp = ap.Experiment(MyModel, sample)
>>> results = exp.run()
>>> results.reporters
        sum_id
run_id
```

(continues on next page)

(continued from previous page)

```
0             1
1             3
2             6
```

6.1.4 Conversion

classmethod `Model.as_function`(***kwargs*)

Converts the model into a function that can be used with the ema_workbench library.

Parameters `**kwargs` – Additional keyword arguments that will passed to the model in addition to the parameters.

Returns The model as a function that takes parameter values as keyword arguments and returns a dictionary of reporters.

Return type function

6.2 Agents

Agent-based models can contain multiple agents of different types. This module provides a base class *`Agent`* that is meant to be used as a template to create custom agent types. Initial variables should be defined by overriding *`Agent.setup()`*.

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

Template for an individual agent.

Parameters

- **model** (`Model`) – The model instance.
- ****kwargs** – Will be forwarded to *`Agent.setup()`*.

Variables

- **id** (*`int`*) – Unique identifier of the agent.
- **log** (*`dict`*) – Recorded variables of the agent.
- **type** (*`str`*) – Class name of the agent.
- **model** (`Model`) – The model instance.
- **p** (`AttrDict`) – The model parameters.
- **vars** (*`list of str`*) – Names of the agent's custom variables.

record(*var_keys*, *value=None*)

Records an object's variables at the current time-step. Recorded variables can be accessed via the object's *log* attribute and will be saved to the model's output at the end of a simulation.

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.

Notes

Recording mutable objects like lists can lead to wrong results if the object's content will be changed during the simulation. Make a copy of the list or record each list entry seperately.

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.vars)
```

`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
```

6.3 Sequences

This module offers various data structures to create and manage groups of both agents and environments. Which structure best to use depends on the specific requirements of each model.

- *`AgentList`* is a list of agentpy objects with methods to select and manipulate its entries.
- *`AgentDList`* is an ordered collection of agentpy objects, optimized for removing and looking up objects.
- *`AgentSet`* is an unordered collection of agents that can access agent attributes.
- *`AgentIter`* and *`AgentDListIter`* are a list-like iterators over a selection of agentpy objects.
- *`AttrIter`* is a list-like iterator over the attributes of each agent in a selection of agentpy objects.

All of these sequence classes can access and manipulate the methods and variables of their objects as an attribute of the container. For examples, see *`AgentList`*.

6.3.1 Containers

class AgentList(*model*, *objs=()*, *cls=None*, **args*, ***kwargs*)

List of agentpy objects. Attribute calls and assignments are applied to all agents and return an `AttrIter` with the 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 the list based on agents' attributes. Standard `list` methods can also be used.

Parameters

- **model** (`Model`) – The model instance.
- **objs** (*int or Sequence, optional*) – An integer number of new objects to be created, or a sequence of existing objects (default empty).
- **cls** (*type, optional*) – Class for the creation of new objects.
- ****kwargs** – Keyword arguments are forwarded to the constructor of the new objects. Keyword arguments with sequences of type `AttrIter` will be broadcasted, meaning that the first value will be assigned to the first object, the second to the second, and so forth. Otherwise, the same value will be assigned to all objects.

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 'x': [1, 1, 1]
```

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

```
>>> agents.y = ap.AttrIter([1, 2, 3])
>>> agents.y
AttrList of '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 'x': [2, 3, 4]

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

Attributes of specific agents can be changed through setting items:

```
>>> agents.x[2] = 10
>>> agents.x
AttrList of 'x': [4, 6, 10]
```

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]
```

random(*n=1*, *replace=False*)

Creates a random sample of agents.

Parameters

- **n** (*int, optional*) – Number of agents (default 1).
- **replace** (*bool, optional*) – Select with replacement (default False). If True, the same agent can be selected more than once.

Returns The selected agents.

Return type *AgentIter*

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()

Shuffles the list in-place, and returns self.

sort(*var_key*, *reverse=False*)

Sorts the list in-place, and returns self.

Parameters

- **var_key** (*str*) – Attribute of the lists' objects, based on which the list will be sorted from lowest value to highest.
- **reverse** (*bool, optional*) – Reverse sorting (default False).

class AgentDList(*model*, *objs=()*, *cls=None*, **args*, ***kwargs*)

Ordered collection of agentpy objects. This container behaves similar to `AgentList` in most aspects, but comes with additional features for object removal and lookup.

The key differences to `AgentList` are the following:

- Faster removal of objects.
- Faster lookup if object is part of group.
- No duplicates are allowed.
- The order of agents in the group cannot be changed.
- Removal of agents changes the order of the group.
- `AgentDList.buffer()` makes it possible to remove objects from the group while iterating over the group.

- `AgentDList.shuffle()` returns an iterator instead of shuffling in-place.

Parameters

- **`model`** (`Model`) – The model instance.
- **`objs`** (*`int or Sequence, optional`*) – An integer number of new objects to be created, or a sequence of existing objects (default empty).
- **`cls`** (*`type, optional`*) – Class for the creation of new objects.
- **`**kwargs`** – Keyword arguments are forwarded to the constructor of the new objects. Keyword arguments with sequences of type `AttrIter` will be broadcasted, meaning that the first value will be assigned to the first object, the second to the second, and so forth. Otherwise, the same value will be assigned to all objects.

`buffer()`

Return `AgentIter` over the content of the group that supports deletion of objects from the group during iteration.

`random`(*n=1*, *replace=False*)

Creates a random sample of agents.

Parameters

- **`n`** (*`int, optional`*) – Number of agents (default 1).
- **`replace`** (*`bool, optional`*) – Select with replacement (default False). If True, the same agent can be selected more than once.

Returns The selected agents.

Return type *AgentIter*

`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()`

Return `AgentIter` over the content of the group with the order of objects being shuffled.

`sort`(*var_key*, *reverse=False*)

Returns a new sorted `AgentList`.

Parameters

- **`var_key`** (*`str`*) – Attribute of the lists' objects, based on which the list will be sorted from lowest value to highest.
- **`reverse`** (*`bool, optional`*) – Reverse sorting (default False).

class `AgentSet`(*model*, *objs=()*, *cls=None*, **args*, ***kwargs*)

Unordered collection of agentpy objects.

Parameters

- **`model`** (`Model`) – The model instance.
- **`objs`** (*`int or Sequence, optional`*) – An integer number of new objects to be created, or a sequence of existing objects (default empty).
- **`cls`** (*`type, optional`*) – Class for the creation of new objects.

- **`**kwargs`** – Keyword arguments are forwarded to the constructor of the new objects. Keyword arguments with sequences of type `AttrIter` will be broadcasted, meaning that the first value will be assigned to the first object, the second to the second, and so forth. Otherwise, the same value will be assigned to all objects.

6.3.2 Iterators

`class AgentIter(`*model*, *source=()*`)`

Iterator over agentpy objects.

`to_dlist()`

Returns an `AgentDList` of the iterator.

`to_list()`

Returns an `AgentList` of the iterator.

`class AgentDListIter(`*model*, *source=()*, *shuffle=False*, *buffer=False*`)`

Iterator over agentpy objects in an `AgentDList`.

`class AttrIter(`*source*, *attr=None*`)`

Iterator over an attribute of objects in a sequence. Length, items access, and representation work like with a normal list. 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. If applied to another *AttrList*, the first entry of the first list will be matched with the first entry of the second list, and so on. Else, the same value will be applied to each entry of the list. See `AgentList` for examples.

6.4 Environments

Environments are objects in which agents can inhabit a specific position. The connection between positions is defined by the environment's topology. There are currently three types:

- `Grid` n-dimensional spatial topology with discrete positions.
- `Space` n-dimensional spatial topology with continuous positions.
- `Network` graph topology consisting of `AgentNode` and edges.

All three environment classes contain the following methods:

- `add_agents()` adds agents to the environment.
- `remove_agents()` removes agents from the environment.
- `move_to()` changes an agent's position.
- `move_by()` changes an agent's position, relative to their current position.
- `neighbors()` returns an agent's neighbors within a given distance.

6.4.1 Discrete spaces (Grid)

class Grid(*model*, *shape*, *torus=False*, *track_empty=False*, *check_border=True*, ***kwargs*)

Environment that contains agents with a discrete spatial topology, supporting multiple agents and attribute fields per cell. For a continuous spatial topology, see `Space`.

This class can be used as a parent class for custom grid types. All agentpy model objects call the method `setup()` after creation, and can access class attributes like dictionary items.

Parameters

- **model** (`Model`) – The model instance.
- **shape** (*tuple of int*) – Size of the grid. The length of the tuple defines the number of dimensions, and the values in the tuple define the length of each dimension.
- **torus** (*bool, optional*) – Whether to connect borders (default False). If True, the grid will be toroidal, meaning that agents who move over a border will re-appear on the opposite side. If False, they will remain at the edge of the border.
- **track_empty** (*bool, optional*) – Whether to keep track of empty cells (default False). If true, empty cells can be accessed via `Grid.empty`.
- **check_border** (*bool, optional*) – Ensure that agents stay within border (default True). Can be set to False for faster performance.
- ****kwargs** – Will be forwarded to `Grid.setup()`.

Variables

- **agents** (`GridIter`) – Iterator over all agents in the grid.
- **positions** (*dict of Agent*) – Dictionary linking each agent instance to its position.
- **grid** (*numpy.rec.array*) – Structured numpy record array with a field 'agents' that holds an `AgentSet` in each position.
- **shape** (*tuple of int*) – Length of each dimension.
- **ndim** (*int*) – Number of dimensions.
- **all** (*list*) – List of all positions in the grid.
- **empty** (*ListDict*) – List of unoccupied positions, only available if the Grid was initiated with *track_empty=True*.

add_agents(*agents*, *positions=None*, *random=False*, *empty=False*)

Adds agents to the grid environment.

Parameters

- **agents** (*Sequence of Agent*) – Iterable of agents to be added.
- **positions** (*Sequence of positions, optional*) – The positions of the agents. Must have the same length as 'agents', with each entry being a tuple of integers. If none is passed, positions will be chosen automatically based on the arguments 'random' and 'empty':
  - random and empty: Random selection without repetition from *Grid.empty*.
  - random and not empty: Random selection with repetition from *Grid.all*.
  - not random and empty: Iterative selection from *Grid.empty*.
  - not random and not empty: Iterative selection from *Grid.all*.
- **random** (*bool, optional*) – Whether to choose random positions (default False).

- **empty** (*bool, optional*) – Whether to choose only empty cells (default False). Can only be True if Grid was initiated with *track_empty=True*.

add_field(*key*, *values=None*)
Add an attribute field to the grid.

Parameters

- **key** (*str*) – Name of the field.
- **values** (*optional*) – Single value or `numpy.ndarray` of values (default None).

apply(*func*, *field='agents'*)
Applies a function to each grid position, end returns an *numpy.ndarray* of return values.

Parameters

- **func** (*function*) – Function that takes cell content as input.
- **field** (*str, optional*) – Field to use (default 'agents').

attr_grid(*attr_key*, *otypes='f'*, *field='agents'*)
Returns a grid with the value of the attribute of the agent in each position, using `numpy.vectorize`. Positions with no agent will contain *numpy.nan*. Should only be used for grids with zero or one agents per cell. Other kinds of attribute grids can be created with `Grid.apply()`.

Parameters

- **attr_key** (*str*) – Name of the attribute.
- **otypes** (*str or list of dtypes, optional*) – Data type of returned grid (default float). For more information, see `numpy.vectorize`.
- **field** (*str, optional*) – Field to use (default 'agents').

del_field(*key*)
Delete a attribute field from the grid.

Parameters **key** (*str*) – Name of the field.

move_by(*agent*, *path*)
Moves agent to new position, relative to current position.

Parameters

- **agent** (*Agent*) – Instance of the agent.
- **path** (*tuple of int*) – Relative change of position.

move_to(*agent*, *pos*)
Moves agent to new position.

Parameters

- **agent** (*Agent*) – Instance of the agent.
- **pos** (*tuple of int*) – New position of the agent.

neighbors(*agent*, *distance=1*)
Select neighbors of an agent within a given distance.

Parameters

- **agent** (*Agent*) – Instance of the agent.
- **distance** (*int, optional*) – Number of cells to cover in each direction, including diagonally connected cells (default 1).

Returns Iterator over the selected neighbors.

Return type *AgentIter*

record(*var_keys*, *value=None*)

Records an object's variables at the current time-step. Recorded variables can be accessed via the object's *log* attribute and will be saved to the model's output at the end of a simulation.

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.

Notes

Recording mutable objects like lists can lead to wrong results if the object's content will be changed during the simulation. Make a copy of the list or record each list entry seperately.

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.vars)
```

record_positions(*label='p'*)

Records the positions of each agent.

Parameters **label** (*string, optional*) – Name under which to record each position (default p). A number will be added for each coordinate (e.g. p1, p2, . . .).

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
```

class GridIter(*model*, *iter_*, *items*)

Iterator over objects in *Grid* that supports slicing.

Examples

Create a model with a 10 by 10 grid with one agent in each position:

```
model = ap.Model()
agents = ap.AgentList(model, 100)
grid = ap.Grid(model, (10, 10))
grid.add_agents(agents)
```

The following returns an iterator over the agents in all position:

```
>>> grid.agents
GridIter (100 objects)
```

The following returns an iterator over the agents in the top-left quarter of the grid:

```
>>> grid.agents[0:5, 0:5]
GridIter (25 objects)
```

to_dlist()

Returns an *AgentDList* of the iterator.

to_list()

Returns an *AgentList* of the iterator.

6.4.2 Continuous spaces (Space)

class Space(*model*, *shape*, *torus=False*, ***kwargs*)

Environment that contains agents with a continuous spatial topology. To add new space environments to a model, use `Model.add_space()`. For a discrete spatial topology, see *Grid*.

This class can be used as a parent class for custom space types. All agentpy model objects call the method *setup()* after creation, and can access class attributes like dictionary items.

Parameters

- **model** (`Model`) – The model instance.
- **shape** (*tuple of float*) – Size of the space. The length of the tuple defines the number of dimensions, and the values in the tuple define the length of each dimension.
- **torus** (*bool, optional*) – Whether to connect borders (default False). If True, the space will be toroidal, meaning that agents who move over a border will re-appear on the opposite side. If False, they will remain at the edge of the border.

- `**kwargs` – Will be forwarded to `Space.setup()`.

Variables

- **agents** (`AgentIter`) – Iterator over all agents in the space.
- **positions** (*dict of Agent*) – Dictionary linking each agent instance to its position.
- **shape** (*tuple of float*) – Length of each spatial dimension.
- **ndim** (*int*) – Number of dimensions.
- **kdtree** (*scipy.spatial.cKDTree or None*) – KDTree of agent positions for neighbor lookup. Will be recalculated if agents have moved. If there are no agents, tree is None.

add_agents(*agents*, *positions=None*, *random=False*)

Adds agents to the space environment.

Parameters

- **agents** (*Sequence of Agent*) – Instance or iterable of agents to be added.
- **positions** (*Sequence of positions, optional*) – The positions of the agents. Must have the same length as 'agents', with each entry being a position (array of float). If none is passed, all positions will be either be zero or random based on the argument 'random'.
- **random** (*bool, optional*) – Whether to choose random positions (default False).

move_by(*agent*, *path*)

Moves agent to new position, relative to current position.

Parameters

- **agent** (`Agent`) – Instance of the agent.
- **path** (*array_like*) – Relative change of position.

move_to(*agent*, *pos*)

Moves agent to new position.

Parameters

- **agent** (`Agent`) – Instance of the agent.
- **pos** (*array_like*) – New position of the agent.

neighbors(*agent*, *distance*)

Select agent neighbors within a given distance. Takes into account wether space is toroidal.

Parameters

- **agent** (`Agent`) – Instance of the agent.
- **distance** (*float*) – Radius around the agent in which to search for neighbors.

Returns Iterator over the selected neighbors.

Return type *AgentIter*

record(*var_keys*, *value=None*)

Records an object's variables at the current time-step. Recorded variables can be accessed via the object's *log* attribute and will be saved to the model's output at the end of a simulation.

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.

Notes

Recording mutable objects like lists can lead to wrong results if the object's content will be changed during the simulation. Make a copy of the list or record each list entry seperately.

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.vars)
```

record_positions(*label='p'*)
Records the positions of each agent.

Parameters **label** (*string, optional*) – Name under which to record each position (default p). A number will be added for each coordinate (e.g. p1, p2, …).

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

select(*center*, *radius*)
Select agents within a given area.

Parameters

- **center** (*array_like*) – Coordinates of the center of the search area.
- **radius** (*float*) – Radius around the center in which to search.

Returns Iterator over the selected agents.

Return type *AgentIter*

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
```

6.4.3 Graph topologies (Network)

class Network(*model*, *graph=None*, ***kwargs*)

Agent environment with a graph topology. Every node of the network is a *AgentNode* that can hold multiple agents as well as node attributes.

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.

Parameters

- **model** (Model) – The model instance.
- **graph** (*networkx.Graph, optional*) – The environments' graph. Can also be a DiGraph, MultiGraph, or MultiDiGraph. Nodes will be converted to *AgentNode*, with their original label being kept as *AgentNode.label*. If none is passed, an empty networkx.Graph is created.
- ****kwargs** – Will be forwarded to *Network.setup()*.

Variables

- **graph** (*networkx.Graph*) – The network's graph instance.
- **agents** (AgentIter) – Iterator over the network's agents.
- **nodes** (AttrIter) – Iterator over the network's nodes.

add_agents(*agents*, *positions=None*)

Adds agents to the network environment.

Parameters

- **agents** (*Sequence of Agent*) – Instance or iterable of agents to be added.
- **positions** (*Sequence of AgentNode, optional*) – The positions of the agents. Must have the same length as 'agents', with each entry being an *AgentNode* of the network. If none is passed, new nodes will be created for each agent.

add_node(*label=None*)

Adds a new node to the network.

Parameters label (*int or string, optional*) – Unique name of the node, which must be different from all other nodes. If none is passed, an integer number will be chosen.

Returns The newly created node.

Return type *AgentNode*

move_to(*agent*, *node*)

Moves agent to new position.

Parameters

- **agent** (Agent) – Instance of the agent.
- **node** (AgentNode) – New position of the agent.

`neighbors`(*agent*)
Select agents from neighboring nodes. Does not include other agents from the agents' own node.

Parameters **agent** (Agent) – Instance of the agent.

Returns Iterator over the selected neighbors.

Return type *AgentIter*

`record`(*var_keys*, *value=None*)
Records an object's variables at the current time-step. Recorded variables can be accessed via the object's *log* attribute and will be saved to the model's output at the end of a simulation.

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.

Notes

Recording mutable objects like lists can lead to wrong results if the object's content will be changed during the simulation. Make a copy of the list or record each list entry seperately.

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.vars)
```

`remove_agents`(*agents*)
Removes agents from the network.

`remove_node`(*node*)
Removes a node from the network.

Parameters **node** (AgentNode) – Node to be removed.

`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
```

class AgentNode(*label*)
Node of `Network`. Functions like a set of agents.

add()
Add an element to a set.

This has no effect if the element is already present.

clear()
Remove all elements from this set.

copy()
Return a shallow copy of a set.

difference()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

difference_update()
Remove all elements of another set from this set.

discard()
Remove an element from a set if it is a member.

If the element is not a member, do nothing.

intersection()
Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

intersection_update()
Update a set with the intersection of itself and another.

isdisjoint()
Return True if two sets have a null intersection.

issubset()
Report whether another set contains this set.

issuperset()
Report whether this set contains another set.

pop()
Remove and return an arbitrary set element. Raises KeyError if the set is empty.

remove()
Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

symmetric_difference()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

symmetric_difference_update()
Update a set with the symmetric difference of itself and another.

union()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

update()
Update a set with the union of itself and others.

6.5 Parameter samples

6.5.1 Value sets and ranges

class **Range**(*vmin=0*, *vmax=1*, *vdef=None*)
A range of parameter values that can be used to create a `Sample`.

Parameters

- **vmin** (*float, optional*) – Minimum value for this parameter (default 0).
- **vmax** (*float, optional*) – Maximum value for this parameter (default 1).
- **vdef** (*float, optional*) – Default value. Default value. If none is passed, *vmin* is used.

class **IntRange**(*vmin=0*, *vmax=1*, *vdef=None*)
A range of integer parameter values that can be used to create a `Sample`. Similar to `Range`, but sampled values will be rounded and converted to integer.

Parameters

- **vmin** (*int, optional*) – Minimum value for this parameter (default 0).
- **vmax** (*int, optional*) – Maximum value for this parameter (default 1).
- **vdef** (*int, optional*) – Default value. If none is passed, *vmin* is used.

class **Values**(**args*, *vdef=None*)
A pre-defined set of discrete parameter values that can be used to create a `Sample`.

Parameters

- ***args** – Possible values for this parameter.
- **vdef** – Default value. If none is passed, the first passed value is used.

6.5.2 Sample generation

class **Sample**(*parameters*, *n=None*, *method='linspace'*, *randomize=True*, ***kwargs*)
A sequence of parameter combinations that can be used for `Experiment`.

Parameters

- **parameters** (*dict*) – Dictionary of parameter keys and values. Entries of type `Range` and `Values` will be sampled based on chosen *method* and *n*. Other types wil be interpreted as constants.
- **n** (*int, optional*) – Sampling factor used by chosen *method* (default None).

- **method** (*str, optional*) – Method to use to create parameter combinations from entries of type *Range*. Options are:
  - `linspace` (default): Arange *n* evenly spaced values for each *Range* and combine them with given *Values* and constants. Additional keyword arguments:
    - `product` (bool, optional): Return all possible combinations (default True). If False, value sets are 'zipped' so that the i-th parameter combination contains the i-th entry of each value set. Requires all value sets to have the same length.
  - `saltelli`: Apply Saltelli's sampling scheme, using `SALib.sample.saltelli.sample()` with *N=n*. This enables the analysis of Sobol Sensitivity Indices with *DataDict.calc_sobol()* after the experiment. Additional keyword arguments:
    - `calc_second_order` (bool, optional): Whether to calculate second-order indices (default True).
- **randomize** (*bool, optional*) – Whether to use the constant parameter 'seed' to generate different random seeds for every parameter combination (default True). If False, every parameter combination will have the same seed. If there is no constant parameter 'seed', this option has no effect.
- ****kwargs** – Additional keyword arguments for chosen *method*.

6.6 Experiments

class **Experiment**(*model_class*, *sample=None*, *iterations=1*, *record=False*, *randomize=True*, ***kwargs*)

Experiment that can run an agent-based model over for multiple iterations and parameter combinations and generate combined output data.

Parameters

- **model** (*type*) – The model class for the experiment to use.
- **sample** (*dict or list of dict or* `Sample`*, optional*) – Parameter combination(s) to test in the experiment (default None).
- **iterations** (*int, optional*) – How often to repeat every parameter combination (default 1).
- **record** (*bool, optional*) – Keep the record of dynamic variables (default False).
- **randomize** (*bool, optional*) – Generate different random seeds for every iteration (default True). If True, the parameter 'seed' will be used to initialize a random seed generator for every parameter combination in the sample. If False, the same seed will be used for every iteration. If no parameter 'seed' is defined, this option has no effect. For more information, see *Randomness and reproducibility* .
- ****kwargs** – Will be forwarded to all model instances created by the experiment.

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

end()

Defines the experiment's actions after the last simulation. Can be overwritten for final calculations and reporting.

run(*n_jobs=1*, *pool=None*, *display=True*, ***kwargs*)

Perform the experiment. The simulation will run the model once for each set of parameters and will repeat this process for the set number of iterations. Simulation results will be stored in *Experiment.output*. Parallel processing is supported based on `joblib.Parallel()`.

Parameters

- **n_jobs** (*int, optional*) – Number of processes to run in parallel (default 1). If 1, no parallel processing is used. If -1, all CPUs are used. Will be forwarded to `joblib.Parallel()`.
- **pool** (*multiprocessing.Pool, optional*) – [This argument is depreciated. Please use 'n_jobs' instead.] Pool of active processes for parallel processing. If none is passed, normal processing is used.
- **display** (*bool, optional*) – Display simulation progress (default True).
- ****kwargs** – Additional keyword arguments for `joblib.Parallel()`.

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 on all CPUs with status updates:

```
exp = ap.Experiment(MyModel, parameters)
results = exp.run(n_jobs=-1, verbose=10)
```

6.7 Data analysis

This module offers tools to access, arrange, analyse, and store output data from simulations. A `DataDict` can be generated by the methods `Model.run()`, `Experiment.run()`, and `DataDict.load()`.

class DataDict(**args*, ***kwargs*)

Nested dictionary for output data of simulations. Items can be accessed like attributes. Attributes can differ from the standard ones listed below.

Variables

- **info** (*dict*) – Metadata of the simulation.
- **parameters** (DataDict) – Simulation parameters.
- **variables** (DataDict) – Recorded variables, separatedper object type.
- **reporters** (*pandas.DataFrame*) – Reported outcomes of the simulation.
- **sensitivity** (DataDict) – Sensitivity data, if calculated.

6.7.1 Data arrangement

`DataDict.`**`arrange`**(*variables=False*, *reporters=False*, *parameters=False*, *constants=False*, *obj_types=True*, *index=False*)

Combines and/or filters data based on passed arguments.

Parameters

- **`variables`** (*`bool or str or list of str, optional`*) – Key or list of keys of variables to include in the dataframe. If True, all available variables are selected. If False (default), no variables are selected.
- **`reporters`** (*`bool or str or list of str, optional`*) – Key or list of keys of reporters to include in the dataframe. If True, all available reporters are selected. If False (default), no reporters are selected.
- **`parameters`** (*`bool or str or list of str, optional`*) – Key or list of keys of parameters to include in the dataframe. If True, all non-constant parameters are selected. If False (default), no parameters are selected.
- **`constants`** (*`bool, optional`*) – Include constants if 'parameters' is True (default False).
- **`obj_types`** (*`str or list of str, optional`*) – Agent and/or environment types to include in the dataframe. If True (default), all objects are selected. If False, no objects are selected.
- **`index`** (*`bool, optional`*) – Whether to keep original multi-index structure (default False).

Returns The newly arranged dataframe.

Return type pandas.DataFrame

`DataDict.`**`arrange_reporters`**()

Common use case of *`DataDict.arrange`* with *reporters=True* and *parameters=True*.

`DataDict.`**`arrange_variables`**()

Common use case of *`DataDict.arrange`* with *variables=True* and *parameters=True*.

6.7.2 Analysis methods

`DataDict.`**`calc_sobol`**(*reporters=None*, ***kwargs*)

Calculates Sobol Sensitivity Indices using `SALib.analyze.sobol.analyze()`. Data must be from an *`Experiment`* with a *`Sample`* that was generated with the method 'saltelli'. If the experiment had more than one iteration, the mean value between iterations will be taken.

Parameters

- **`reporters`** (*`str or list of str, optional`*) – The reporters that should be used for the analysis. If none are passed, all existing reporters except 'seed' are used.
- **`**kwargs`** – Will be forwarded to `SALib.analyze.sobol.analyze()`.

Returns The DataDict itself with an added category 'sensitivity'.

Return type *DataDict*

6.7.3 Save and load

`DataDict.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.info['model_type']* is used.
- **exp_id** (*int, optional*) – Number of the experiment. Note that passing an existing id can overwrite existing data. 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).

`classmethod DataDict.load`(*exp_name=None*, *exp_id=None*, *path='ap_output'*, *display=True*)

Reads 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*

6.8 Visualization

`animate`(*model*, *fig*, *axs*, *plot*, *steps=None*, *seed=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 the arguments *model, axs, *fargs* and creates the desired plots on each axis at each time-step.
- **steps** (*int, optional*) – Number of (additional) steps for the simulation to run. If passed, the parameter 'Model.p.steps' will be ignored. The simulation can still be stopped with :func:'Model.stop'. If there is no step-limit through either this argument or the parameter 'Model.p.steps', the animation will stop at t=10000.

- **seed** (*int, optional*) – Seed for the models random number generators. If none is given, the parameter 'Model.p.seed' will be used. If there is no such parameter, a random seed will be used.
- **skip** (*int, optional*) – Steps 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())
```

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

Parameters

- **grid** (*numpy.array*) – Two-dimensional array 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. If there is an entry *None*, it will be used for all NaN values.
- **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()`.

Returns `matplotlib.image.AxesImage`

6.9 Examples

The following example models are presented in the *Model Library*.

To use these classes, they have to be imported as follows:

```
from agentpy.examples import WealthModel
```

class WealthModel(*parameters=None, _run_id=None, **kwargs*)
Demonstration model of random wealth transfers.

See also:

Notebook in the model library: *Wealth transfer*

Parameters **parameters** (*dict*) –

- agents (int): Number of agents.
- steps (int, optional): Number of time-steps.

class SegregationModel(*parameters=None*, *_run_id=None*, ***kwargs*)

Demonstration model of segregation dynamics.

See also:

Notebook in the model library: *Segregation*

Parameters **parameters** (*dict*) –

- want_similar (float): Percentage of similar neighbors for agents to be happy
- n_groups (int): Number of groups
- density (float): Density of population
- size (int): Height and length of the grid
- steps (int, optional): Maximum number of steps

6.10 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
```

CHAPTER SEVEN

COMPARISON

There are numerous modeling and simulation tools for ABMs, each with their own particular focus and style (find an overview here). The three main distinguishing features of agentpy are the following:

- Agentpy integrates the multiple tasks of agent-based modeling - model design, interactive simulations, numerical experiments, and data analysis - within a single environment and is optimized for interactive computing with IPython and Jupyter.
- Agentpy is designed for scientific use with experiments over multiple runs. It provides tools for parameter sampling (similar to NetLogo's BehaviorSpace), Monte Carlo experiments, stochastic processes, parallel computing, and sensitivity analysis.
- Agentpy is written in Python, one of the world's most popular programming languages that offers a vast number of tools and libraries for scientific use. It is further designed for compatibility with established packages like numpy, scipy, networkx, pandas, ema_workbench, seaborn, and SALib.

The main alternative to agentpy in Python is Mesa. To allow for an comparison of the syntax, 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 (*Agentpy* & Mesa).

Agentpy

```python
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.agents = ap.AgentList(
            self, self.p.n, MoneyAgent)

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










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

# Perform multiple runs
variable_params = {
    'n': ap.IntRange(10, 500),
    'steps': 10
}
sample = ap.Sample(variable_params, n=49)
exp = ap.Experiment(
    MoneyModel,
    sample,
    iterations=5,
    record=True
)
results = exp.run()
```

Mesa

```python
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()
```

The following table further provides a comparison of the main features of each framework.

Feature	**Agentpy**	**Mesa**
Containers	Sequence classes like AgentList and AgentDList	Scheduler classes for different activation orders
Topologies	Spatial grid, continuous space, network	Spatial grid, continuous space, network
Data recording	Recording methods for variables of agents, environments, and model; as well as reporters	DataCollector class that can collect variables of agents and model
Parameter sampling	Classes for sample generation and different types of parameter ranges	
Multi-run experiments	Experiment class that supports multiple iterations, parameter samples, randomization, and parallel processing	BatchRunner class that supports multiple iterations and parameter ranges
Output data	DataDict class to store, save, load, and re-arrange output data	Methods to generate dataframes
Visualization	Gridplots, animations, and interactive visualization within Jupyter Notebooks	Plots and interactive visualization in a separate web-server
Analysis	Tools for data arrangement and sensitivity analysis	

CHAPTER

EIGHT

CHANGELOG

8.1 0.1.5 (December 2021)

- *Experiment.run()* has a new argument 'n_jobs' that allows for parallel processing with `joblib.Parallel()`.
- Two new methods - *Grid.record_positions()* and *Space.record_positions()* - can be used to record agent positions.
- *Model.run()* can now continue simulations that have already been run. The steps defined in the argument 'steps' now reflect additional steps, which will be added to the models current time-step. Random number generators will not be re-initialized in this case.
- *animate()* has been improved. It used to stop the animation one step too early, which has been fixed. Two faulty import statements have been corrected. And, as above, the argument 'steps' now also reflects additional steps.
- *Grid.add_field()* has been fixed. Single values can now be passed.

8.2 0.1.4 (September 2021)

- *AttrIter* now returns a new *AttrIter* when called as a function.
- *gridplot()* now returns an `matplotlib.image.AxesImage`
- *DataDict.save()* now supports values of type `numpy.bool_` and can re-write to existing directories if an existing *exp_id* is passed.
- *DataDict.load()* now supports the argument *exp_id = 0*.
- *animate()* now supports more than 100 steps.
- *AttrIter* now returns a new *AttrIter* when called as a function.
- *Model* can take a new parameter *report_seed* (default True) that indicates whether the seed of the current run should be reported.

8.3 0.1.3 (August 2021)

- The `Grid` functionality *track_empty* has been fixed to work with multiple agents per cell.
- Getting and setting items in `AttrIter` has been fixed.
- Sequences like `AgentList` and `AgentDList` no longer accept *args*, only *kwargs*. These keyword arguments are forwarded to the constructor of the new objects. Keyword arguments with sequences of type `AttrIter` will be broadcasted, meaning that the first value will be assigned to the first object, the second to the second, and so forth. Otherwise, the same value will be assigned to all objects.

8.4 0.1.2 (June 2021)

- The property `Network.nodes` now returns an `AttrIter`, so that network nodes can be assigned to agents as follows:

```
self.nw = ap.Network(self)
self.agents = ap.AgentList(self, 10)
self.nw.add_agents(self.agents)
self.agents.node = self.nw.nodes
```

- `AgentIter` now requires the model to be passed upon creation and has two new methods `AgentIter.to_list()` and `AgentIter.to_dlist()` for conversion between sequence types.
- Syntax highlighting in the documentation has been fixed.

8.5 0.1.1 (June 2021)

- Marked release for the upcoming JOSS publication of AgentPy.
- Fixed `Grid.move_to()`: Agents can now move to their current position.

8.6 0.1.0 (May 2021)

This update contains major revisions of most classes and methods in the library, including new features, better performance, and a more coherent syntax. The most important API changes are described below.

8.6.1 Object creation

The methods `add_agents()`, `add_env()`, etc. have been removed. Instead, new objects are now created directly or through *Sequences*. This allows for more control over data structures (see next point) and attribute names. For example:

```
class Model(ap.Model):
    def setup(self):
        self.single_agent = ap.Agent()  # Create a single agent
        self.agents = ap.AgentList(self, 10)  # Create a sequence of 10 agents
        self.grid = ap.Grid(self, (5, 5))  # Create a grid environment
```

8.6.2 Data structures

The new way of object creation makes it possible to choose specific data structures for different groups of agents. In addition to `AgentList`, there is a new sequence type `AgentDList` that provides increased performance for the lookup and deletion of agents. It also comes with a method `AgentDList.buffer()` that allows for safe deletion of agents from the list while it is iterated over

`AttrList` has been replaced by `AttrIter`. This improves performance and makes it possible to change agent attributes by setting new values to items in the attribute list (see `AgentList` for an example). In most other ways, the class still behaves like a normal list. There are also two new classes `AgentIter` and `AgentDListIter` that are returned by some of the library's methods.

8.6.3 Environments

The three environment classes have undergone a major revision. The `add_agents()` functions have been extended with new features and are now more consistent between the three environment classes. The method `move_agents()` has been replaced by `move_to()` and `move_by()`. `Grid` is now defined as a structured numpy array that can hold field attributes per position in addition to agents, and can be customized with the arguments *torus*, *track_empty*, and *check_border*. `gridplot()` has been adapted to support this new numpy structure. `Network` now consists of `AgentNode` nodes that can hold multiple agents per node, as well as node attributes.

8.6.4 Environment-agent interaction

The agents' *env* attribute has been removed. Instead, environments are manually added as agent attributes, giving more control over the attribute name in the case of multiple environments. For example, agents in an environment can be set up as follows:

```
class Model(ap.Model):
    def setup(self):
        self.agents = ap.AgentList(self, 10)
        self.grid = self.agents.mygrid = ap.Grid(self, (10, 10))
        self.grid.add_agents(self.agents)
```

The agent methods *move_to*, *move_by*, and *neighbors* have also been removed. Instead, agents can access these methods through their environment. In the above example, a given agent *a* could for example access their position through *a.mygrid.positions[a]* or their neighbors through calling *a.mygrid.neighbors(a)*.

8.6.5 Parameter samples

Variable parameters can now be defined with the three new classes `Range` (for continuous parameter ranges), `IntRange` (for integer parameter ranges), and `Values` (for pre-defined of discrete parameter values). Parameter dictionaries with these classes can be used to create samples, but can also be passed to a normal model, which will then use default values. The sampling methods `sample()`, `sample_discrete()`, and `sample_saltelli()` have been removed and integrated into the new class `Sample`, which comes with additional features to create new kinds of samples.

8.6.6 Random number generators

`Model` now contains two random number generators *Model.random* and *Model.nprandom* so that both standard and numpy random operations can be used. The parameter *seed* can be used to initialize both generators. `Sample` has an argument *randomize* to vary seeds over parameter samples. And `Experiment` has a new argument *randomize* to control whether to vary seeds over different iterations. More on this can be found in *Randomness and reproducibility*.

8.6.7 Data analysis

The structure of output data in `DataDict` has been changed. The name of *measures* has been changed to *reporters*. Parameters are now stored in the two categories *constants* and *sample*. Variables are stored in separate dataframes based on the object type. The dataframe's index is now separated into *sample_id* and *iteration*. The function `sensitivity_sobol()` has been removed and is replaced by the method `DataDict.calc_sobol()`.

8.6.8 Interactive simulations

The method `Experiment.interactive()` has been removed and is replaced by an interactive simulation interface that is being developed in the separate package ipysimulate. This new package provides interactive javascript widgets with parameter sliders and live plots similar to the traditional NetLogo interface. Examples can be found in *Interactive simulations*.

8.7 0.0.7 (March 2021)

8.7.1 Continuous space environments

A new environment type `Space` and method `Model.add_space()` for agent-based models with continuous space topologies has been added. There is a new demonstration model *Flocking behavior* in the model library, which shows how to simulate the flocking behavior of animals and demonstrates the use of the continuous space environment.

8.7.2 Random number generators

`Model` has a new property `Model.random`, which returns the models' random number generator of type `numpy.random.Generator()`. A custom seed can be set for `Model.run()` and `animate()` by either passing an argument or defining a parameter `seed`. All methods with stochastic elements like `AgentList.shuffle()` or `AgentList.random()` now take an optional argument *generator*, with the model's main generator being used if none is passed. The function `AgentList.random()` now uses `numpy.random.Generator.choice()` and has three new arguments 'replace', 'weights', and 'shuffle'. More information with examples can be found in the API reference and the new user guide *Randomness and reproducibility*.

8.7.3 Other changes

- The function `sensitivity_sobol()` now has an argument `calc_second_order` (default False). If True, the function will add second-order indices to the output.
- The default value of `calc_second_order` in `sample_saltelli()` has also been changed to False for consistency.
- For consistency with *`Space`*, *`Grid`* no longer takes an integer as argument for 'shape'. A tuple with the lengths of each spatial dimension has to be passed.
- The argument 'agents' has been removed from `Environment`. Agents have to be added through `Environment.add_agents()`.

8.7.4 Fixes

- The step limit in *`animate()`* is now the same as in *`Model.run()`*.
- A false error message in *`DataDict.save()`* has been removed.

8.8 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()`.

8.9 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`.

8.10 0.0.4 (November 2020)

First documented 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. Contributions of clear and simple demonstration models for the *Model Library* that illustrate a particular application are also very welcome.

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.

Thanks to everyone who has contributed or supported the developement of this package:

- Jeroen C.J.M. van den Bergh
- Ivan Savin
- James Millington
- Martí Bosch
- Sebastian Benthall
- Bhakti Stephan Onggo

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).

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

INDEX