# Mesa Documentation

***Release 4.0.0a0***

**Mesa Team**

**Jun 04, 2026**

# CONTENTS


**1 Features** **3**

**2 Using Mesa** **5**

2.1 Installation Options . . . 5
2.2 Resources . . . 5
2.3 Development and Support . . . 6
2.4 Citing Mesa . . . 6
2.4.1 Getting started . . . 6
2.4.1.1 Overview . . . 6
2.4.1.2 Tutorials . . . 6
2.4.1.3 Examples . . . 7
2.4.1.4 Further resources . . . 7
2.4.1.4.1 Best practices . . . 7
2.4.1.4.2 API documentation . . . 7
2.4.1.4.3 Repository of models built using MESA . . . 7
2.4.1.4.4 Migration guide . . . 7
2.4.1.4.5 Source Code and development . . . 7
2.4.1.4.6 Community and support . . . 7
2.4.2 Overview of the MESA library . . . 7
2.4.2.1 Modeling modules . . . 8
2.4.2.2 Spaces in Mesa . . . 8
2.4.2.2.1 Discrete Spaces . . . 9
2.4.2.2.2 Property Layers . . . 9
2.4.2.2.3 Continuous Space . . . 9
2.4.2.3 Time Advancement and Agent Activation . . . 9
2.4.2.3.1 Running the model . . . 10
2.4.2.3.2 Agent Activation Patterns . . . 10
2.4.2.3.3 Event Scheduling . . . 10
2.4.2.4 AgentSet and model.agents . . . 11
2.4.2.4.1 model.agents . . . 11
2.4.2.4.2 AgentSet Functionality . . . 11
2.4.2.5 Analysis modules . . . 12
2.4.2.6 Visualization . . . 12
2.4.2.6.1 Creating Your First Model . . . 13
2.4.2.6.1.1 The Boltzmann Wealth Model . . . 13
2.4.2.6.1.2 Tutorial Description . . . 13
2.4.2.6.1.3 Model Description . . . 14
2.4.2.6.1.4 Tutorial Setup . . . 14
2.4.2.6.1.5 IN COLAB? - Run the next cell . . . 14
2.4.2.6.1.6 Building the Sample Model . . . 14

2.4.2.6.1.7 Import Dependencies . . . 14
2.4.2.6.1.8 Create Agent . . . 15
2.4.2.6.1.9 Create Model . . . 15
2.4.2.6.1.10 Making the Agents `do` . . . 16
2.4.2.6.1.11 Running the Model . . . 17
2.4.2.6.1.12 Exercise . . . 17
2.4.2.6.1.13 Agents Exchange . . . 18
2.4.2.6.1.14 Running your first model . . . 19
2.4.2.6.1.15 Exercise . . . 21
2.4.2.6.1.16 Next Steps . . . 21
2.4.2.6.1.17 More Mesa . . . 21
2.4.2.6.1.18 Happy Modeling! . . . 22
2.4.2.6.2 Working with AgentSets . . . 22
2.4.2.6.2.1 The Boltzmann Wealth Model . . . 22
2.4.2.6.2.2 Tutorial Description . . . 22
2.4.2.6.2.3 IN COLAB? - Run the next cell . . . 22
2.4.2.6.2.4 Import Dependencies . . . 22
2.4.2.6.2.5 Setup: The Wealth Model with Ethnicities . . . 23
2.4.2.6.2.6 What is an AgentSet? . . . 23
2.4.2.6.2.7 Retrieving Attribute Values with `get` . . . 24
2.4.2.6.2.8 Handling missing attributes . . . 24
2.4.2.6.2.9 Filtering Agents with `select` . . . 25
2.4.2.6.2.10 Basic filtering with a function . . . 25
2.4.2.6.2.11 Filtering by agent type . . . 25
2.4.2.6.2.12 Limiting results with `at_most` . . . 25
2.4.2.6.2.13 Combining criteria . . . 26
2.4.2.6.2.14 Chaining selects . . . 26
2.4.2.6.2.15 Computing Aggregates with `agg` . . . 26
2.4.2.6.2.16 Multiple aggregations at once . . . 27
2.4.2.6.2.17 Aggregating subsets . . . 27
2.4.2.6.2.18 Grouping Agents with `groupby` . . . 27
2.4.2.6.2.19 Iterating over groups . . . 27
2.4.2.6.2.20 Aggregating across groups . . . 28
2.4.2.6.2.21 Grouping by a function . . . 28
2.4.2.6.2.22 Setting Attributes with `set` . . . 28
2.4.2.6.2.23 Sorting Agents with `sort` . . . 29
2.4.2.6.2.24 Converting to a List . . . 29
2.4.2.6.2.25 Putting It Together: Analyzing the Model . . . 29
2.4.2.6.2.26 Visualizing the Results . . . 30
2.4.2.6.2.27 Summary . . . 31
2.4.2.6.2.28 Next Steps . . . 32
2.4.2.6.3 Agent Activation . . . 32
2.4.2.6.3.1 The Boltzmann Wealth Model . . . 32
2.4.2.6.3.2 Tutorial Description . . . 32
2.4.2.6.3.3 IN COLAB? - Run the next cell . . . 32
2.4.2.6.3.4 Import Dependencies . . . 32
2.4.2.6.3.5 `do` and `shuffle_do`: The Core Activation Methods . . . 32
2.4.2.6.3.6 Why Activation Order Matters . . . 33
2.4.2.6.3.7 Using Callables with `do` . . . 34
2.4.2.6.3.8 Collecting Results with `map` . . . 35
2.4.2.6.3.9 Conditional Activation with `select` + `do` . . . 36
2.4.2.6.3.10 Common Activation Patterns . . . 37
2.4.2.6.3.11 Sequential Activation . . . 37
2.4.2.6.3.12 Random Activation . . . 37

2.4.2.6.3.13 Simultaneous Activation . . . 37
2.4.2.6.3.14 Staged Activation . . . 37
2.4.2.6.3.15 Type-Based Activation . . . 38
2.4.2.6.3.16 Combining Patterns . . . 38
2.4.2.6.3.17 Summary . . . 40
2.4.2.6.3.18 Next Steps . . . 40
2.4.2.6.4 Event Scheduling & Time . . . 40
2.4.2.6.4.1 Tutorial Description . . . 40
2.4.2.6.4.2 IN COLAB? - Run the next cell . . . 41
2.4.2.6.4.3 Import Dependencies . . . 41
2.4.2.6.4.4 How Time Works in Mesa . . . 41
2.4.2.6.4.5 The default step mechanism . . . 41
2.4.2.6.4.6 A quick example . . . 41
2.4.2.6.4.7 `run_for` vs `run_until` . . . 42
2.4.2.6.4.8 Scheduling One-Off Events . . . 43
2.4.2.6.4.9 Canceling events . . . 44
2.4.2.6.4.10 Scheduling Recurring Events . . . 45
2.4.2.6.4.11 Controlling when recurring events start . . . 46
2.4.2.6.4.12 Stopping recurring events . . . 47
2.4.2.6.4.13 Using `end` and `count` for automatic limits . . . 48
2.4.2.6.4.14 Dynamic intervals . . . 48
2.4.2.6.4.15 Event Priority . . . 49
2.4.2.6.4.16 Putting It All Together: A Complete Example . . . 50
2.4.2.6.4.17 When to Use Events vs Steps . . . 53
2.4.2.6.4.18 Summary . . . 53
2.4.2.6.4.19 Next Steps . . . 54
2.4.2.6.5 Adding Space . . . 54
2.4.2.6.5.1 The Boltzmann Wealth Model . . . 54
2.4.2.6.5.2 Tutorial Description . . . 54
2.4.2.6.5.3 IN COLAB? - Run the next cell . . . 54
2.4.2.6.5.4 Import Dependencies . . . 54
2.4.2.6.5.5 Base Model . . . 55
2.4.2.6.5.6 Adding space . . . 56
2.4.2.6.5.7 Code Implementation . . . 57
2.4.2.6.5.8 Exercises . . . 60
2.4.2.6.5.9 Next Steps . . . 60
2.4.2.6.6 Collecting Data . . . 60
2.4.2.6.6.1 The Boltzmann Wealth Model . . . 60
2.4.2.6.6.2 Tutorial Description . . . 61
2.4.2.6.6.3 IN COLAB? - Run the next cell . . . 61
2.4.2.6.6.4 Import Dependencies . . . 61
2.4.2.6.6.5 Base Model . . . 61
2.4.2.6.6.6 Collecting Data . . . 62
2.4.2.6.6.7 Analyzing MoneyModel Data . . . 64
2.4.2.6.6.8 Exercises . . . 65
2.4.2.6.6.9 Analyzing an MoneyAgent Data . . . 65
2.4.2.6.6.10 Next Steps . . . 70
2.4.2.6.7 Visualization - Basic Dashboard . . . 71
2.4.2.6.7.1 The Boltzmann Wealth Model . . . 71
2.4.2.6.7.2 Tutorial Description . . . 71
2.4.2.6.7.3 Import Dependencies . . . 71
2.4.2.6.7.4 Basic Model . . . 71
2.4.2.6.7.5 Important note for SolaraViz users . . . 73
2.4.2.6.7.6 Adding visualization . . . 74

2.4.2.6.7.7 Grid Visualization . . . 75
2.4.2.6.7.8 SpaceRenderer . . . 75
2.4.2.6.7.9 Page Tab View . . . 76
2.4.2.6.7.10 **Plot Components** . . . 76
2.4.2.6.7.11 **Custom Components** . . . 76
2.4.2.6.7.12 Next Steps . . . 77
2.4.2.6.8 Visualization - Dynamic Agents . . . 77
2.4.2.6.8.1 The Boltzmann Wealth Model . . . 77
2.4.2.6.8.2 Tutorial Description . . . 77
2.4.2.6.8.3 Import Dependencies . . . 77
2.4.2.6.8.4 Basic Model . . . 78
2.4.2.6.8.5 Adding visualization . . . 80
2.4.2.6.8.6 Dynamic Agent Representation . . . 80
2.4.2.6.8.7 Page Tab View . . . 81
2.4.2.6.8.8 **Plot Components** . . . 81
2.4.2.6.8.9 **Custom Components** . . . 82
2.4.2.6.8.10 Exercise . . . 83
2.4.2.6.8.11 Next Steps . . . 83
2.4.2.6.9 Visualization - Advanced Space Rendering . . . 83
2.4.2.6.9.1 The Boltzmann Wealth Model . . . 83
2.4.2.6.9.2 Tutorial Description . . . 83
2.4.2.6.9.3 Import Dependencies . . . 83
2.4.2.6.9.4 Basic Model . . . 84
2.4.2.6.9.5 Adding visualization . . . 86
2.4.2.6.9.6 Page Tab View . . . 87
2.4.2.6.9.7 **Plot Components** . . . 87
2.4.2.6.9.8 **Custom Components** . . . 88
2.4.2.6.9.9 Advanced Rendering with `SpaceRenderer` . . . 88
2.4.2.6.9.10 Drawing the Grid and Agents . . . 89
2.4.2.6.9.11 Using Altair Backend . . . 89
2.4.2.6.9.12 Customizing the Final Output with `post_process` . . . 89
2.4.2.6.9.13 Post-Processing Line Plots . . . 90
2.4.2.6.9.14 Launching the Full Visualization . . . 90
2.4.2.6.9.15 Exercise . . . 91
2.4.2.6.9.16 Next Steps . . . 91
2.4.2.6.10 Visualization - Property Layer Visualization . . . 91
2.4.2.6.10.1 The Boltzmann Wealth Model . . . 91
2.4.2.6.10.2 Tutorial Description . . . 91
2.4.2.6.10.3 Import Dependencies . . . 91
2.4.2.6.10.4 Basic Model . . . 97
2.4.2.6.10.5 Adding visualization . . . 99
2.4.2.6.10.6 Page Tab View . . . 100
2.4.2.6.10.7 **Plot Components** . . . 100
2.4.2.6.10.8 **Custom Components** . . . 101
2.4.2.6.10.9 Visualizing property_layer . . . 101
2.4.2.6.10.10 Drawing the property_layer . . . 102
2.4.2.6.10.11 Launching the Visualization . . . 103
2.4.2.6.10.12 Exercise . . . 104
2.4.2.6.10.13 Next Steps . . . 104
2.4.2.6.11 Visualization - Custom Components . . . 104
2.4.2.6.11.1 The Boltzmann Wealth Model . . . 104
2.4.2.6.11.2 Tutorial Description . . . 104
2.4.2.6.11.3 Import Dependencies . . . 104
2.4.2.6.11.4 Basic Model . . . 105

2.4.2.6.11.5 Adding visualization . . . 107
2.4.2.6.11.6 Building Custom Components . . . 107
2.4.2.6.11.7 Page Tab View . . . 108
2.4.2.6.11.8 **Plot Components** . . . 108
2.4.2.6.11.9 **Custom Components** . . . 109
2.4.2.6.11.10 Exercise . . . 110
2.4.2.6.12 Best Practices . . . 110
2.4.2.6.12.1 Model Layout . . . 110
2.4.2.6.12.2 Randomization . . . 111
2.4.3 Mesa Core Examples . . . 111
2.4.3.1 Overview . . . 112
2.4.3.2 Basic Examples . . . 112
2.4.3.2.1 *Boltzmann Wealth Model* . . . 112
2.4.3.2.2 *Boids Flockers Model* . . . 112
2.4.3.2.3 *Conway's Game of Life* . . . 112
2.4.3.2.4 *Schelling Segregation Model* . . . 112
2.4.3.2.5 *Virus on a Network Model* . . . 112
2.4.3.3 Advanced Examples . . . 112
2.4.3.3.1 *Epstein Civil Violence Model* . . . 112
2.4.3.3.2 *Demographic Prisoner's Dilemma on a Grid* . . . 112
2.4.3.3.3 *Sugarscape Model with Traders* . . . 113
2.4.3.3.4 *Wolf-Sheep Predation Model* . . . 113
2.4.3.3.4.1 Conway's Game Of "Life" . . . 113
2.4.3.3.4.2 Summary . . . 113
2.4.3.3.4.3 How to Run . . . 113
2.4.3.3.4.4 Files . . . 113
2.4.3.3.4.5 Optional . . . 113
2.4.3.3.4.6 Further Reading . . . 113
2.4.3.3.4.7 Agents . . . 113
2.4.3.3.4.8 Model . . . 115
2.4.3.3.4.9 App . . . 115
2.4.3.3.4.10 Virus on a Network . . . 117
2.4.3.3.4.11 Summary . . . 117
2.4.3.3.4.12 How to Run . . . 117
2.4.3.3.4.13 Files . . . 118
2.4.3.3.4.14 Further Reading . . . 118
2.4.3.3.4.15 Agents . . . 118
2.4.3.3.4.16 Model . . . 119
2.4.3.3.4.17 App . . . 121
2.4.3.3.4.18 Schelling Segregation Model . . . 124
2.4.3.3.4.19 Summary . . . 124
2.4.3.3.4.20 How to Run . . . 124
2.4.3.3.4.21 Files . . . 124
2.4.3.3.4.22 Further Reading . . . 124
2.4.3.3.4.23 Agents . . . 125
2.4.3.3.4.24 Model . . . 126
2.4.3.3.4.25 App . . . 127
2.4.3.3.4.26 Boltzmann Wealth Model (Tutorial) . . . 129
2.4.3.3.4.27 Summary . . . 129
2.4.3.3.4.28 How to Run . . . 130
2.4.3.3.4.29 Files . . . 130
2.4.3.3.4.30 How the Model Is Structured . . . 130
2.4.3.3.4.31 Understanding the Output . . . 130
2.4.3.3.4.32 Optional . . . 130

2.4.3.3.4.33 Further Reading . . . 130
2.4.3.3.4.34 Agents . . . 131
2.4.3.3.4.35 Model . . . 132
2.4.3.3.4.36 App . . . 133
2.4.3.3.4.37 Boids Flockers . . . 136
2.4.3.3.4.38 Summary . . . 136
2.4.3.3.4.39 How to Run . . . 136
2.4.3.3.4.40 Files . . . 136
2.4.3.3.4.41 Further Reading . . . 136
2.4.3.3.4.42 Agents . . . 136
2.4.3.3.4.43 Model . . . 138
2.4.3.3.4.44 App . . . 140
2.4.3.3.4.45 Sugarscape Constant Growback Model with Traders . . . 142
2.4.3.3.4.46 Summary . . . 142
2.4.3.3.4.47 Agents: . . . 142
2.4.3.3.4.48 How to Run . . . 143
2.4.3.3.4.49 Files . . . 143
2.4.3.3.4.50 Further Reading . . . 144
2.4.3.3.4.51 Agents . . . 144
2.4.3.3.4.52 Model . . . 150
2.4.3.3.4.53 App . . . 152
2.4.3.3.4.54 Demographic Prisoner's Dilemma on a Grid . . . 154
2.4.3.3.4.55 Summary . . . 154
2.4.3.3.4.56 How to Run . . . 154
2.4.3.3.4.57 Files . . . 155
2.4.3.3.4.58 Further Reading . . . 155
2.4.3.3.4.59 Agents . . . 155
2.4.3.3.4.60 Model . . . 156
2.4.3.3.4.61 App . . . 158
2.4.3.3.4.62 Epstein Civil Violence Model . . . 159
2.4.3.3.4.63 Summary . . . 159
2.4.3.3.4.64 How to Run . . . 159
2.4.3.3.4.65 Files . . . 160
2.4.3.3.4.66 Further Reading . . . 160
2.4.3.3.4.67 Agents . . . 160
2.4.3.3.4.68 Model . . . 163
2.4.3.3.4.69 App . . . 165
2.4.3.3.4.70 Wolf-Sheep Predation Model . . . 167
2.4.3.3.4.71 Summary . . . 167
2.4.3.3.4.72 How to Run . . . 168
2.4.3.3.4.73 Files . . . 168
2.4.3.3.4.74 Further Reading . . . 168
2.4.3.3.4.75 Agents . . . 168
2.4.3.3.4.76 Model . . . 171
2.4.3.3.4.77 App . . . 174
2.4.3.3.4.78 Alliance Formation Model (Meta-Agent Example) . . . 176
2.4.3.3.4.79 Summary . . . 176
2.4.3.3.4.80 Files . . . 177
2.4.3.3.4.81 Further Reading . . . 177
2.4.3.3.4.82 How to Run . . . 177
2.4.3.3.4.83 Agents . . . 177
2.4.3.3.4.84 Model . . . 178
2.4.3.3.4.85 App . . . 181
2.4.4 Mesa Migration guide . . . 183

2.4.4.1 Mesa 3.5.0 . . . 183
2.4.4.1.1 Event scheduling and time advancement . . . 183
2.4.4.1.1.1 Time-based advancement replaces step loops . . . 183
2.4.4.1.1.2 One-off event scheduling . . . 183
2.4.4.1.1.3 Recurring event scheduling . . . 184
2.4.4.1.1.4 Replacing Simulator classes . . . 184
2.4.4.1.2 AgentSet sequence behavior . . . 184
2.4.4.2 Mesa 3.4.0 . . . 185
2.4.4.2.1 batch run . . . 185
2.4.4.3 Mesa 3.3.0 . . . 185
2.4.4.3.1 Defining Portrayal Components . . . 186
2.4.4.3.2 Passing portrayal arguments to draw methods . . . 186
2.4.4.3.3 Default Space Visualization . . . 187
2.4.4.3.4 Page Tab View . . . 187
2.4.4.4 Mesa 3.0 . . . 188
2.4.4.4.1 Upgrade strategy . . . 188
2.4.4.4.2 Reserved and private variables . . . 188
2.4.4.4.2.1 Reserved variables . . . 188
2.4.4.4.2.2 Private variables . . . 188
2.4.4.4.3 Removal of `mesa.flat` namespace . . . 188
2.4.4.4.4 Mandatory Model initialization with `super().__init__()` . . . 188
2.4.4.4.5 Automatic assignment of `unique_id` to Agents . . . 189
2.4.4.4.6 AgentSet and `Model.agents` . . . 189
2.4.4.4.6.1 `Model.agents` . . . 190
2.4.4.4.7 Time and schedulers . . . 190
2.4.4.4.7.1 Automatic increase of the `steps` counter . . . 190
2.4.4.4.7.2 Removal of `Model._time` and rename `._steps` . . . 190
2.4.4.4.7.3 Removal of `Model._advance_time()` . . . 190
2.4.4.4.7.4 Replacing Schedulers with AgentSet functionality . . . 190
2.4.4.4.7.5 BaseScheduler . . . 190
2.4.4.4.7.6 RandomActivation . . . 191
2.4.4.4.7.7 SimultaneousActivation . . . 191
2.4.4.4.7.8 StagedActivation . . . 191
2.4.4.4.7.9 RandomActivationByType . . . 191
2.4.4.4.7.10 Replacing `step_type` . . . 192
2.4.4.4.7.11 General Notes . . . 192
2.4.4.4.8 Visualisation . . . 192
2.4.4.4.8.1 Model Initialization . . . 193
2.4.4.4.9 Model Initialization with Keyword Arguments . . . 193
2.4.4.4.9.1 Default space visualization . . . 193
2.4.4.4.9.2 Plotting “measures” . . . 193
2.4.4.4.9.3 Plotting text . . . 194
2.4.4.4.10 Other changes . . . 194
2.4.4.4.10.1 Removal of Model.initialize_data_collector . . . 194
2.4.5 APIs . . . 194
2.4.5.1 Model . . . 194
2.4.5.2 Agent . . . 200
2.4.5.3 AgentSet . . . 202
2.4.5.4 Time . . . 213
2.4.5.5 Discrete Space . . . 218
2.4.5.6 Data collection . . . 243
2.4.5.7 Visualization . . . 245
2.4.5.7.1 Jupyter Visualization . . . 245
2.4.5.7.2 User Parameters . . . 250

2.4.5.7.3 Matplotlib-based visualizations . . . 251
2.4.5.7.4 Altair-based visualizations . . . 255
2.4.5.7.5 Command Console . . . 256
2.4.5.7.6 Portrayal Components . . . 259
2.4.5.7.7 Backends . . . 261
2.4.5.7.8 Space Renderer . . . 268
2.4.5.7.9 Space Drawers . . . 270
2.4.5.8 Experimental . . . 274
2.4.5.8.1 Continuous Space . . . 274
2.4.5.8.2 Continuous Space . . . 276
2.4.5.8.3 Scenarios . . . 278

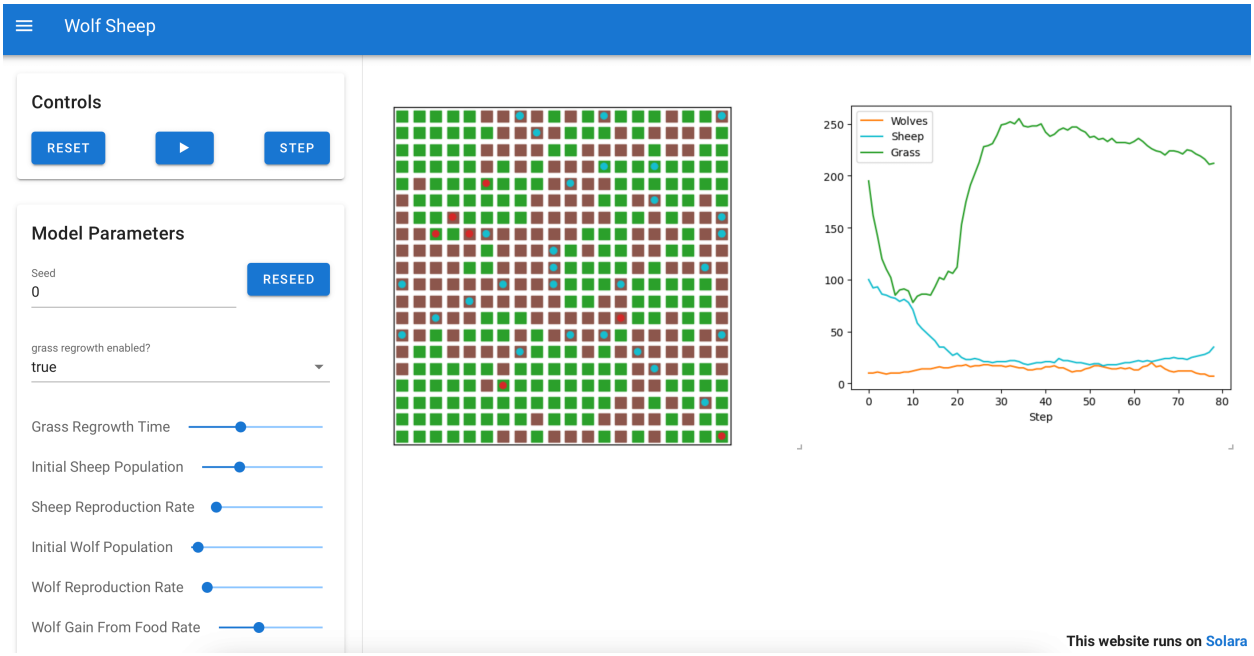
**3 Indices and tables** **281**

**Python Module Index** **283**

**Index** **285**

Mesa is an Apache2 licensed agent-based modeling (or ABM) framework in Python.

Mesa allows users to quickly create agent-based models using built-in core components (such as spatial grids and agent schedulers) or customized implementations; visualize them using a browser-based interface; and analyze their results using Python's data analysis tools. Mesa's goal is to make simulations accessible to everyone, so humanity can more effectively understand and solve complex problems.




*A visualisation of the Wolf Sheep model build with Mesa. An online demo is available here.*

CHAPTER

ONE

# FEATURES

- Built-in core modeling components
- Flexible agent and model management through AgentSet
- Browser-based Solara visualization
- Built-in tools for data collection and analysis
- Example model library

CHAPTER

TWO

# USING MESA

## 2.1 Installation Options

To install our latest stable Mesa 3 release, run:

```
pip install -U mesa
```

Development of Mesa 4 has started. To install our latest Mesa 4 pre-release, use:

```
pip install -U --pre mesa
```

To also install our recommended dependencies:

```
pip install -U mesa[rec]
```

The `[rec]` option installs additional recommended dependencies needed for visualization, plotting, and network modeling capabilities.

On a Mac, this command might cause an error stating `zsh: no matches found: mesa[all]`. In that case, change the command to `pip install -U "mesa[rec]"`.

Furthermore, if you are using `nix`, Mesa comes with a flake with devShells and a runnable app:

```
nix run github:project-mesa/mesa # for default Python shell
```

For development shell, clone the repository and run the following command from repository root:

```
nix develop .#uv2nix # pure shell
```

## 2.2 Resources

For help getting started with Mesa, check out these resources:

- *Getting started* - Learn about Mesa's core concepts and components
- *Migration Guide* - Upgrade to Mesa 3.0
- Mesa Examples - Browse user-contributed models and implementations
- Mesa releases - Check what's new in the latest Mesa version
- Mesa Extensions - Overview of mesa's Extensions
- GitHub Discussions - Ask questions and discuss Mesa
- Matrix Chat Room - Real-time chat with the Mesa community

## 2.3 Development and Support

Mesa is an open source project and welcomes contributions:

- GitHub Repository - Access the source code
- Issue Tracker - Report bugs or suggest features
- Contributors Guide - Learn how to contribute
- GSoC at Mesa — Candidates Guide - For candidates interested in participating in the Google Summer of Code at Mesa

## 2.4 Citing Mesa

To cite Mesa in your publication, you can refer to our peer-reviewed article in the Journal of Open Source Software (JOSS):

- ter Hoeven, E., Kwakkel, J., Hess, V., Pike, T., Wang, B., rht, & Kazil, J. (2025). Mesa 3: Agent-based modeling with Python in 2025. Journal of Open Source Software, 10(107), 7668. https://doi.org/10.21105/joss.07668

Our CITATION.cff can be used to generate APA, BibTeX and other citation formats.

The original Mesa conference paper from 2015 is available here.

### 2.4.1 Getting started

Mesa is a modular framework for building, analyzing and visualizing agent-based models.

**Agent-based models** are computer simulations involving multiple entities (the agents) acting and interacting with one another based on their programmed behavior. Agents can be used to represent living cells, animals, individual humans, even entire organizations or abstract entities. Sometimes, we may have an understanding of how the individual components of a system behave, and want to see what system-level behaviors and effects emerge from their interaction. Other times, we may have a good idea of how the system overall behaves, and want to figure out what individual behaviors explain it. Or we may want to see how to get agents to cooperate or compete most effectively. Or we may just want to build a cool toy with colorful little dots moving around.

#### 2.4.1.1 Overview

If you want to get a general idea of Mesa's features and structure, start here:

- *Overview of the MESA library*: Learn about the core concepts and components of Mesa.

#### 2.4.1.2 Tutorials

If you want to learn how to build agent-based models step by step using Mesa, follow these tutorials:

- *Creating Your First Model*: Learn how to create your first Mesa model.
- *AgentSet*: Learn how to more effectively manage agents with Mesa's AgentSet.
- *Agent Activation*: Learn different ways to activate agents using `do`, `shuffle_do`, and `map` — covering random, sequential, simultaneous, staged, type-based, and conditional activation patterns.
- *Event Scheduling*: Learn how to schedule events and manage time in your Mesa model.
- *Adding Space*: Learn how to add space to your Mesa model and understand Mesa's space architecture.
- *Collecting Data*: Learn how to collect model level and agent level data with Mesa's DataCollector.
- *Basic Visualization*: Learn how to build an interactive dashboard with Mesa's visualization module.

- *Dynamic Agent Visualization*: Learn how to dynamically represent your agents in your interactive dashboard.
- *Visualization using SpaceRenderer*: Learn how to use SpaceRenderer to its full extent to enhance your visualizations.
- *Property Layer Visualization*: Learn how to visualize property layers in Mesa.
- *Custom Visualization Components*: Learn how to add custom visual components to your interactive dashboard.

#### 2.4.1.3 Examples

Mesa ships with a collection of example models. These are classic ABMs, so if you are familiar with ABMs and want to get a quick sense of how MESA works, these examples are great place to start. You can find them *here*.

#### 2.4.1.4 Further resources

To further explore Mesa and its features, we have the following resources available:

##### 2.4.1.4.1 Best practices

- *Mesa best practices*: an overview of tips and guidelines for using MESA.

##### 2.4.1.4.2 API documentation

- *Mesa API reference*: Detailed documentation of Mesa's classes and functions.

##### 2.4.1.4.3 Repository of models built using MESA

- Mesa Examples repository: A collection of example models demonstrating various Mesa features and modeling techniques.

##### 2.4.1.4.4 Migration guide

- *Mesa 3.0 Migration guide*: If you're upgrading from an earlier version of Mesa, this guide will help you navigate the changes in Mesa 3.0.

##### 2.4.1.4.5 Source Code and development

- Mesa GitHub repository: Access the full source code of Mesa, contribute to its development, or report issues.
- Mesa release notes: View the detailed changelog of Mesa, including all past releases and their features.

##### 2.4.1.4.6 Community and support

- Mesa GitHub Discussions: Join discussions, ask questions, and connect with other Mesa users.
- Matrix Chat: Real-time chat for quick questions and community interaction.

Enjoy modelling with Mesa, and feel free to reach out!

### 2.4.2 Overview of the MESA library

Mesa is modular, meaning that its modeling, analysis and visualization components are kept separate but intended to work together. The modules are grouped into three categories:

1. **Modeling:** Classes used to build the models themselves: a model and agent classes, space for them to move around in, and built-in functionality for managing agents.

2. **Analysis:** Tools to collect data generated from your model, or to run it multiple times with different parameter values.
3. **Visualization:** Classes to create and launch an interactive model visualization, using a browser-based interface.

#### 2.4.2.1 Modeling modules

Most models consist of one class to represent the model itself and one or more classes for agents. Mesa provides built-in functionality for managing agents and their interactions. These are implemented in Mesa's modeling modules:

- *mesa.model*
- *mesa.agent*
- *mesa.agentset*
- *mesa.discrete_space*

The skeleton of a model might look like this:

```
import mesa

class MyAgent(mesa.Agent):
    def __init__(self, model, age):
        super().__init__(model)
        self.age = age

    def step(self):
        self.age += 1
        print(f"Agent {self.unique_id} now is {self.age} years old")
        # Whatever else the agent does when activated

class MyModel(mesa.Model):
    def __init__(self, n_agents):
        super().__init__()
        self.grid = mesa.discrete_space.OrthogonalMooreGrid((10, 10), torus=True)
        initial_ages = self.rng.integers(0, 80, size=n_agents)
        agents = MyAgent.create_agents(self, n_agents, initial_ages)
        for agent in agents:
            agent.cell = self.grid.all_cells.select_random_cell()

    def step(self):
        self.agents.shuffle_do("step")
```

You can instantiate a model and run it for one step with:

```
model = MyModel(n_agents=5)
model.run_for(1)
```

#### 2.4.2.2 Spaces in Mesa

Mesa provides several types of spaces where agents can exist and interact:

#### 2.4.2.2.1 Discrete Spaces

Mesa implements discrete spaces through the `mesa.discrete_space` module, using a doubly-linked structure where each cell maintains connections to its neighbors. Available variants include:

1. **Grid-based Spaces:**

```
# Create a Von Neumann grid (4 neighbors per cell)
grid = mesa.discrete_space.OrthogonalVonNeumannGrid((width, height), torus=False)

# Create a Moore grid (8 neighbors per cell)
grid = mesa.discrete_space.OrthogonalMooreGrid((width, height), torus=True)

# Create a hexagonal grid
grid = mesa.discrete_space.HexGrid((width, height), torus=False)
```

2. **Network Space:**

```
# Create a network-based space
network = mesa.discrete_space.Network(graph)
```

3. **Voronoi Space:**

```
# Create an irregular tessellation
mesh = mesa.discrete_space.VoronoiMesh(points)
```

#### 2.4.2.2.2 Property Layers

Discrete spaces support PropertyLayers - efficient numpy-based arrays for storing cell-level property_layers:

```
# Create and use a property layer
grid.create_property_layer("elevation", default_value=10)
high_ground = grid.property_layers["elevation"] > 50
```

#### 2.4.2.2.3 Continuous Space

For models requiring continuous movement:

```
# Create a continuous space
space = mesa.space.ContinuousSpace(x_max, y_max, torus=True)

# Move an agent to specific coordinates
space.move_agent(agent, (new_x, new_y))
```

> **Note:** The legacy `mesa.space` module (including `MultiGrid`, `SingleGrid`, etc.) is in maintenance-only mode. For new projects, use `mesa.discrete_space` and `mesa.experimental.continuous_space` instead.

### 2.4.2.3 Time Advancement and Agent Activation

Mesa supports multiple approaches to advancing time and activating agents.

#### 2.4.2.3.1 Running the model

Use the time advancement methods on `Model` to run your simulation:

```
model = MyModel()
model.run_for(100)      # Run for 100 time units
model.run_until(50.0)   # Run until absolute time 50
```

By default, the model's `step()` method is scheduled to run every 1.0 time units, so `model.run_for(10)` executes 10 steps.

#### 2.4.2.3.2 Agent Activation Patterns

Mesa provides flexible agent activation through the AgentSet API:

```
# Sequential activation
model.agents.do("step")

# Random activation
model.agents.shuffle_do("step")

# Multi-stage activation
for stage in ["move", "eat", "reproduce"]:
    model.agents.do(stage)

# Activation by agent type
for klass in model.agent_types:
    model.agents_by_type[klass].do("step")
```

#### 2.4.2.3.3 Event Scheduling

Mesa supports event-based time progression directly on the `Model`:

```
# Schedule one-off events
model.schedule_event(some_function, at=25.0)     # At absolute time
model.schedule_event(some_function, after=5.0)   # Relative to now

# Cancel a scheduled event
event = model.schedule_event(callback, at=100.0)
event.cancel()

# Schedule recurring events
from mesa.time import Schedule

model.schedule_recurring(func, Schedule(interval=10))            # Every 10 time units
model.schedule_recurring(func, Schedule(interval=10, start=0))   # Starting immediately
model.schedule_recurring(func, Schedule(interval=1.0, count=10)) # Limited to 10␣
↪executions

# Stop a recurring event
gen = model.schedule_recurring(func, Schedule(interval=5.0))
gen.stop()

# Advance time, processing all scheduled events along the way
```

(continues on next page)

(continued from previous page)

```
model.run_for(50)
model.run_until(100.0)
```

This enables pure event-driven models, hybrid approaches combining agent-based steps with scheduled events, and everything in between.

#### 2.4.2.4 AgentSet and model.agents

`model.agents` and the AgentSet class are central in managing and activating agents.

##### 2.4.2.4.1 model.agents

`model.agents` is an AgentSet containing all agents in the model. It's automatically updated when agents are added or removed:

```
# Get total number of agents
num_agents = len(model.agents)

# Iterate over all agents
for agent in model.agents:
    print(agent.unique_id)
```

##### 2.4.2.4.2 AgentSet Functionality

AgentSet offers several methods for efficient agent management:

1. **Selecting**: Filter agents based on criteria.

   ```
   high_energy_agents = model.agents.select(lambda a: a.energy > 50)
   ```

2. **Shuffling and Sorting**: Randomize or order agents.

   ```
   shuffled_agents = model.agents.shuffle()
   sorted_agents = model.agents.sort(key="energy", ascending=False)
   ```

3. **Applying methods**: Execute methods on all agents.

   ```
   model.agents.do("step")
   model.agents.shuffle_do("move")  # Shuffle then apply method
   ```

4. **Aggregating**: Compute aggregate values across agents.

   ```
   avg_energy = model.agents.agg("energy", func=np.mean)
   ```

5. **Grouping**: Group agents by attributes.

   ```
   grouped_agents = model.agents.groupby("species")

   for _, agent_group in grouped_agents:
      agent_group.shuffle_do()
   species_counts = grouped_agents.count()
   mean_age_by_group = grouped_agents.agg("age", np.mean)
   ```

`model.agents` can also be accessed within a model instance using `self.agents`.

These are just some examples of using the AgentSet, there are many more possibilities, see the *AgentSet API docs*.

#### 2.4.2.5 Analysis modules

If you're using modeling for research, you'll want a way to collect the data each model run generates. You'll probably also want to run the model multiple times, to see how some output changes with different parameters. Data collection is implemented in the appropriately-named analysis modules:

- *mesa.datacollection*

You'd add a data collector to the model like this:

```
import mesa
import numpy as np

# ...

class MyModel(mesa.Model):
    def __init__(self, n_agents):
        super().__init__()
        # ... (model initialization code)
        self.datacollector = mesa.DataCollector(
            model_reporters={"mean_age": lambda m: m.agents.agg("age", np.mean)},
            agent_reporters={"age": "age"}
        )

    def step(self):
        self.agents.shuffle_do("step")
        self.datacollector.collect(self)
```

The data collector will collect the specified model- and agent-level data at each step of the model. After you're done running it, you can extract the data as a pandas DataFrame:

```
model = MyModel(5)
model.run_for(10)
model_df = model.datacollector.get_model_vars_dataframe()
agent_df = model.datacollector.get_agent_vars_dataframe()
```

#### 2.4.2.6 Visualization

Mesa uses a browser-based visualization system called SolaraViz. This allows for interactive, customizable visualizations of your models.

Note: SolaraViz is in active development in Mesa 3.x. While we attempt to minimize them, there might be API breaking changes in minor releases.

> **Note:** SolaraViz instantiates new models using `**model_parameters.value`, so all model inputs must be keyword arguments.

Ensure your model's `__init__` method accepts keyword arguments matching the `model_params` keys.

```
class MyModel(Model):
    def __init__(self, n_agents=10):
        super().__init__()
        # Initialize the model with N agents
```

The core functionality for building your own visualizations resides in the *mesa.visualization* namespace.

Here's a basic example of how to set up a visualization:

```
from mesa.visualization import SolaraViz, make_space_component, make_plot_component


def agent_portrayal(agent):
    return AgentPortrayalStyle(color="blue", size=50)


model_params = {
    "n_agents": Slider(
        label="Number of agents:",
        value=50,
        min=1,
        max=100,
        step=1
    )
}

page = SolaraViz(
    MyModel(n_agents=42),
    [
        make_space_component(agent_portrayal),
        make_plot_component("mean_age")
    ],
    model_params=model_params
)
page
```

This will create an interactive visualization of your model, including:

- A grid visualization of agents
- A plot of a model metric over time
- A slider to adjust the number of agents

#### 2.4.2.6.1 Creating Your First Model

##### 2.4.2.6.1.1 The Boltzmann Wealth Model

**Important:**

- If you are just exploring Mesa and want the fastest way to execute the code we recommend executing this tutorial online in a Colab notebook.  or if you do not have a Google account you can use  (This can take 30 seconds to 5 minutes to load)
- If you are running locally, please ensure you have the latest Mesa version installed.

##### 2.4.2.6.1.2 Tutorial Description

Mesa is a Python framework for agent-based modeling. This tutorial is the first in a series of introductory tutorials that will assist you in getting started and discover some of the core features of Mesa. The tutorial starts with the key pieces of a model and then progressively adds functionality.

Should anyone find any errors, bugs, have a suggestion, or just are looking for clarification, let us know in our chat!

The premise of this tutorial is to create a starter-level model representing agents exchanging money.

#### 2.4.2.6.1.3 Model Description

This is a simulated agent-based economy. In an agent-based economy, the behavior of an individual economic agent, such as a consumer or producer, is studied in a market environment. This model is drawn from the field econophysics, specifically a paper prepared by Drăgulescu et al. for additional information on the modeling assumptions used in this model. [Drăgulescu, 2002].

The assumptions that govern this model are:

1. There are some number of agents.
2. All agents begin with 1 unit of money.
3. At every step of the model, an agent gives 1 unit of money (if they have it) to some other agent.

Even as a starter-level model it yields results that are both interesting and unexpected.

Due to its simplicity and intriguing results, we found it to be a good starter model.

#### 2.4.2.6.1.4 Tutorial Setup

Create and activate a virtual environment. *Python version 3.12 or higher is required.*

Install Mesa:

```
pip install mesa[rec]
```

Install Jupyter notebook (optional):

```
pip install jupyter
```

Install Seaborn (which is used for data visualization):

```
pip install seaborn
```

#### 2.4.2.6.1.5 IN COLAB? - Run the next cell

#### 2.4.2.6.1.6 Building the Sample Model

After Mesa is installed a model can be built.

This tutorial is written in Jupyter to facilitate the explanation portions.

Start Jupyter from the command line:

```
jupyter lab
```

Create a new notebook named `money_model.ipynb` (or whatever you want to call it).

#### 2.4.2.6.1.7 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np
```

(continues on next page)

(continued from previous page)

```
# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns

import mesa
```

#### 2.4.2.6.1.8 Create Agent

First create the agent. As the tutorial progresses, more functionality will be added to the agent.

**Background:** Agents are the individual entities that act in the model. Mesa automatically assigns each agent that is created an integer as a `unique_id`.

**Model-specific information:** Agents are the individuals that exchange money, in this case the amount of money an individual agent has is represented as wealth.

**Code implementation:** This is done by creating a new class (or object) that extends `mesa.Agent` creating a subclass of the `Agent` class from mesa. The new class is named `MoneyAgent`. The inherited code of the Mesa agent object can be found in the mesa repo.

The `MoneyAgent` class is created with the following code:

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
        # Pass the parameters to the parent class.
        super().__init__(model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1
```

#### 2.4.2.6.1.9 Create Model

Next, create the model. This gives us the two basic classes of any Mesa ABM - the agent class (population of agent objects that doing something) and the manager class (a model object that manages the creation, activation, data collection etc of the agents)

**Background:** The model can be visualized as a list containing all the agents. The model creates, holds and manages all the agent objects, specifically in a dictionary. The model activates agents in discrete time steps.

**Model-specific information:** When a model is created the number of agents within the model is specified. The model then creates the agents and places them in a set of agents.

**Code implementation:** This is done by creating a new class (or object) that extends `mesa.Model` and calls `super().__init__()`, creating a subclass of the `Model` class from mesa. The new class is named `MoneyModel`. The Mesa code you are using can be found in model module and the AgentSet in the agent module. A critical point is that you can use the `rng` kwarg (keyword argument) to set a seed which controls the random number generator of the model class allowing for the reproducibility of results.

The `MoneyModel` class is created with the following code:

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10, rng=None):
        super().__init__(rng=rng)
        self.num_agents = n
        # Create agents
        MoneyAgent.create_agents(model=self, n=n)
```

#### 2.4.2.6.1.10 Making the Agents do

With the basics of the Agent class and Model class created we can now activate the agents to do things

**Background:** Mesa's do function calls agent functions to grow your ABM. A step is the smallest unit of time in the model, and is often referred to as a tick. The do function and Python functionality can be configured to activate agents in different orders. This can be important as the order in which agents are activated can impact the results of the model [Comer2014]. At each step of the model, one or more of the agents – usually all of them – are activated and take their own step, changing internally and/or interacting with one another or the environment.

**Model-specific information:** For this section we will randomly reorder the Agent activation order using `mesa.Agent.shuffle_do` and have the agents `step` function print the agent's unique id that they were assigned during the agent creation process.

**Code implementation:** Using standard ABM convention we add a `step` function to the `MoneyModel` class which calls the `mesa.Agent.shuffle_do` function. We then pass into `shuffle_do` the parameter "step". This tells mesa to look for and execute the `step` function in our MoneyAgent class.

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
        # Pass the parameters to the parent class.
        super().__init__(model)

        # Create the agent's attribute and set the initial values.
        self.wealth = 1

    def say_hi(self):
        # The agent's step will go here.
        # For demonstration purposes we will print the agent's unique_id
        print(f"Hi, I am an agent, you can call me {self.unique_id!s}.")


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10, rng=None):
        super().__init__(rng=rng)
        self.num_agents = n

        # Create n agents
        MoneyAgent.create_agents(model=self, n=n)

    def step(self):
```

(continues on next page)

(continued from previous page)

```
        """Advance the model by one step."""
        # This function pseudo-randomly reorders the list of agent objects
        # then iterates through calling the function passed in as the parameter
        self.agents.shuffle_do("say_hi")
```

#### 2.4.2.6.1.11 Running the Model

We now have the pieces of a basic model. The model can be run by creating a model object and calling the step method. The model will run for one step and print the unique_id of each agent. You may run the model for multiple steps by calling the step method multiple times.

Create the model object, and run it for one step:

```
starter_model = MoneyModel(10)
starter_model.step()
```

```
Hi, I am an agent, you can call me 6.
Hi, I am an agent, you can call me 2.
Hi, I am an agent, you can call me 1.
Hi, I am an agent, you can call me 4.
Hi, I am an agent, you can call me 3.
Hi, I am an agent, you can call me 9.
Hi, I am an agent, you can call me 10.
Hi, I am an agent, you can call me 7.
Hi, I am an agent, you can call me 5.
Hi, I am an agent, you can call me 8.
```

```
# Run this step a few times and see what happens!
starter_model.step()
# Notice the order of the agents changes each time.
```

```
Hi, I am an agent, you can call me 4.
Hi, I am an agent, you can call me 8.
Hi, I am an agent, you can call me 2.
Hi, I am an agent, you can call me 6.
Hi, I am an agent, you can call me 1.
Hi, I am an agent, you can call me 3.
Hi, I am an agent, you can call me 7.
Hi, I am an agent, you can call me 9.
Hi, I am an agent, you can call me 5.
Hi, I am an agent, you can call me 10.
```

```
# Challenge: Change the seed from None to a number like 42 and see the impact
```

```
# Challenge: Change `shuffle_do` to just `do` and see the impact
```

#### 2.4.2.6.1.12 Exercise

Modifying the code below to have every agent print out its `wealth` when it is activated.

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
        # Pass the parameters to the parent class.
        super().__init__(model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1

    def say_wealth(self):
        # The agent's step will go here.
        # FIXME: need to print the agent's wealth
        print("Hi, I am an agent and I am broke!")
```

Create a model for 12 Agents, and run it for a few steps to see the output.

```
# Fixme: Create the model object, and run it
```

#### 2.4.2.6.1.13 Agents Exchange

Returning back to the MoneyAgent the actual exchange process is now going to be created.

**Background:** This is where the agent's behavior as it relates to each step or tick of the model is defined.

**Model-specific information:** In this case, the agent will check its wealth, and if it has money, give one unit of it away to another random agent.

**Code implementation:** The agent's step method is called by `mesa.Agent.shuffle_do("exchange")`during each step of the model. To allow the agent to choose another agent at random, we use the `model.random` random-number generator. This works just like Python's `random` module, but if a fixed seed is set when the model is instantiated (see earlier challenge), this allows users to replicate a specific model run later. Once we identify this other agent object we increase their wealth by 1 and decrease this agents wealth by one.

This updates the step function as shown below:

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
        # Pass the parameters to the parent class.
        super().__init__(model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1

    def exchange(self):
        # Verify agent has some wealth
        if self.wealth > 0:
            other_agent = self.random.choice(self.model.agents)
            if other_agent is not None:
                other_agent.wealth += 1
                self.wealth -= 1
```

(continues on next page)

(continued from previous page)

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10, rng=None):
        super().__init__(rng=rng)
        self.num_agents = n

        # Create agents
        MoneyAgent.create_agents(model=self, n=n)

    def step(self):
        """Advance the model by one step."""
        # This function psuedo-randomly reorders the list of agent objects and
        # then iterates through calling the function passed in as the parameter
        self.agents.shuffle_do("exchange")
```

#### 2.4.2.6.1.14 Running your first model

With exchange behavior added, it's time for the first rudimentary run of the model.

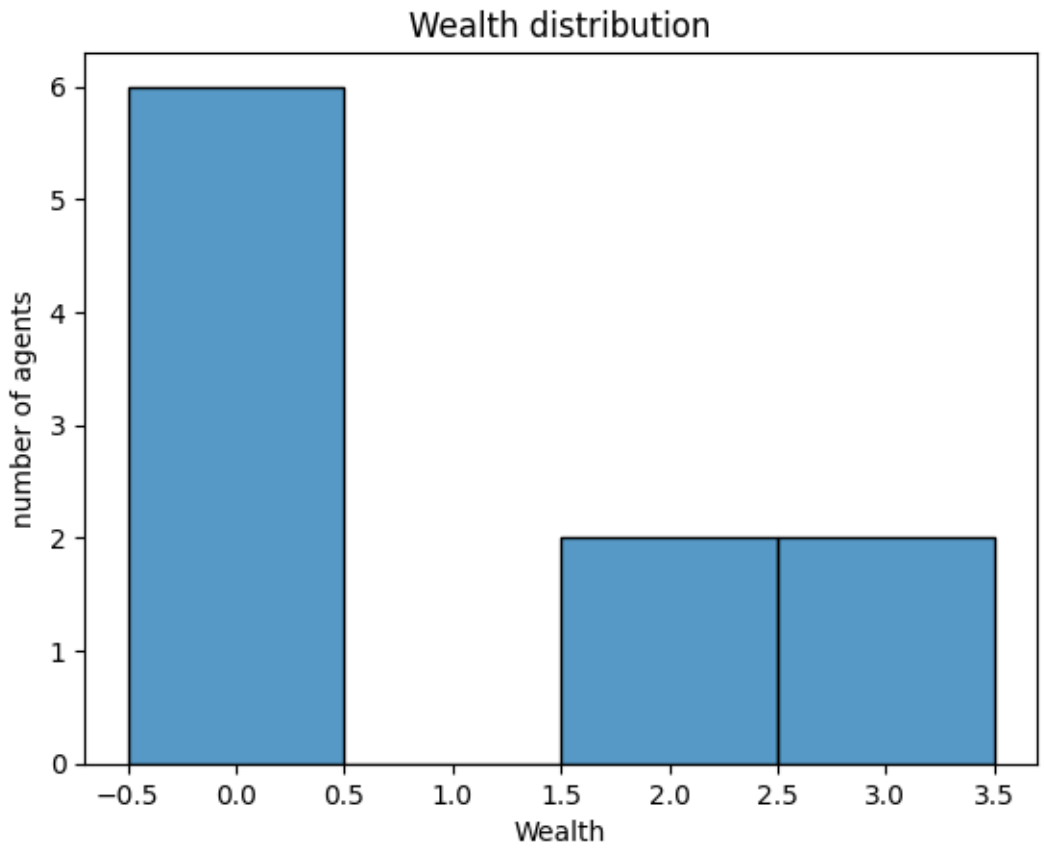
Now let's create a model with 10 agents, and run it for 30 steps.

```
model = MoneyModel(10)  # Tells the model to create 10 agents
for _ in range(30):  # Runs the model for 30 steps;
    model.step()

# Note: An underscore is common convention for a variable that is not used.
```

Next, we need to get some data out of the model. Specifically, we want to see the distribution of the agent's wealth.

We can get the wealth values with list comprehension, and then use seaborn (or another graphics library) to visualize the data in a histogram.

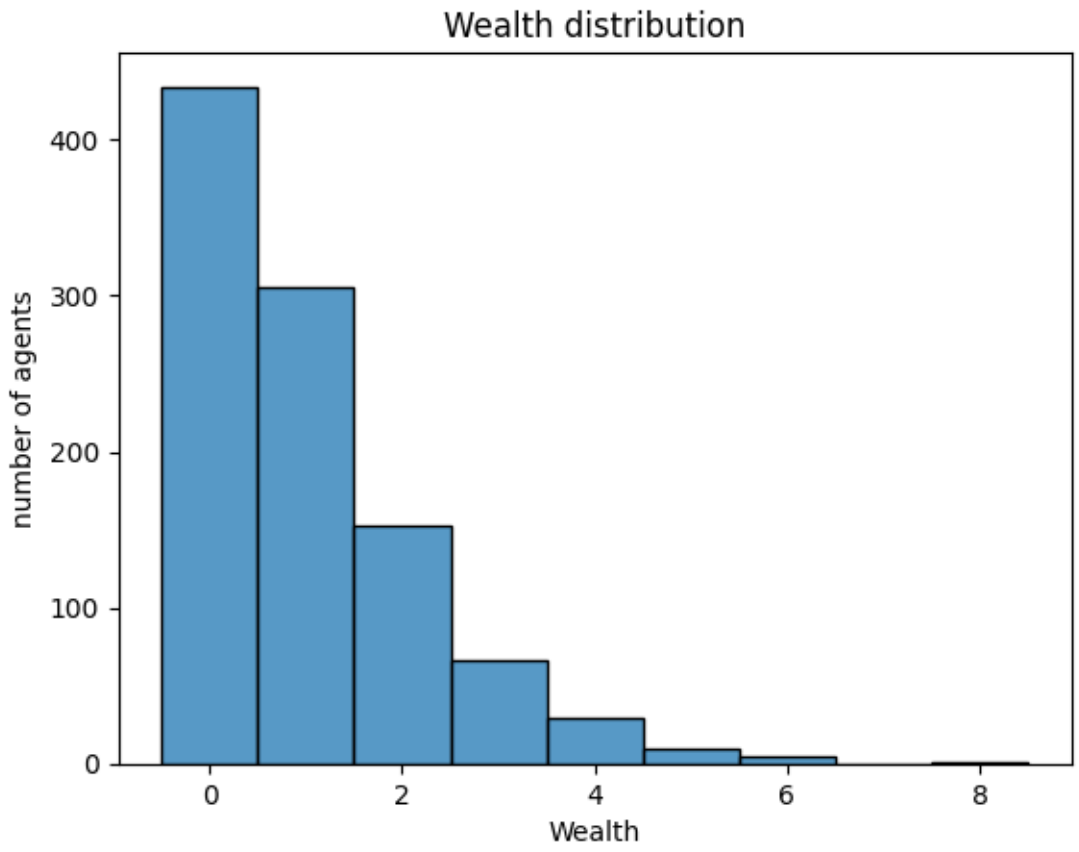
```
agent_wealth = [a.wealth for a in model.agents]
# Create a histogram with seaborn
g = sns.histplot(agent_wealth, discrete=True)
g.set(
    title="Wealth distribution", xlabel="Wealth", ylabel="number of agents"
);  # The semicolon is just to avoid printing the object representation
```




To get a better idea of how a model behaves, we can create multiple model objects and see the distribution that emerges from all of them.

We can do this with a nested for loop:

```
all_wealth = []
# This runs the model 100 times, each model executing 30 steps.
for _ in range(100):
    # Run the model
    model = MoneyModel(10)
    model.run_for(30)

    # Store the results
    for agent in model.agents:
        all_wealth.append(agent.wealth)

# Use seaborn
g = sns.histplot(all_wealth, discrete=True)
g.set(title="Wealth distribution", xlabel="Wealth", ylabel="number of agents");
```




This runs 100 instantiations of the model, and runs each for 30 steps.

Notice that we set the histogram bins to be integers (`discrete=True`), since agents can only have whole numbers of wealth.

This distribution looks a lot smoother. By running the model 100 times, we smooth out some of the 'noise' of randomness, and get to the model's overall expected behavior.

This outcome might be surprising. Despite the fact that all agents, on average, give and receive one unit of money every step, the model converges to a state where most agents have a small amount of money and a small number have a lot of money.

#### 2.4.2.6.1.15 Exercise

Change the above code to see the impact of different model runs, agent populations, and number of steps.

#### 2.4.2.6.1.16 Next Steps

Check out the *AgentSet tutorial* on how to query, filter, group, and inspect collections of agents.

#### 2.4.2.6.1.17 More Mesa

If you are looking for other Mesa models or tools here are some additional resources.

- Example ABMs: Find canonical examples and examples of ABMs demonstrating highlighted features in the Examples Tab
- Expanded Examples: Want to integrate Reinforcement Learning or work on the Traveling Salesman Problem? Checkout Mesa Examples

- Mesa-Geo: If you need an ABM with Geographic Information Systems (GIS) checkout Mesa-Geo
- Mesa Frames: Have a large complex model that you need to speed up, check out Mesa Frames

#### 2.4.2.6.1.18 Happy Modeling!

This document is a work in progress. If you see any errors, exclusions or have any problems please contact us.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

### 2.4.2.6.2 Working with AgentSets

#### 2.4.2.6.2.1 The Boltzmann Wealth Model

**Important:**

- If you are just exploring Mesa and want the fastest way to execute the code we recommend executing this tutorial online in a Colab notebook. or if you do not have a Google account you can use (This can take 30 seconds to 5 minutes to load)
- If you are running locally, please ensure you have the latest Mesa version installed.

#### 2.4.2.6.2.2 Tutorial Description

This tutorial builds on the Boltzmann Wealth Model from the *First Model tutorial*. In the first tutorial you created agents, put them in a model, and had them exchange money. Now we'll explore **AgentSet** — Mesa's core tool for querying, filtering, grouping, and inspecting collections of agents. By the end of this tutorial, you will know how to:

- Retrieve attribute values from agents
- Filter agents with `select`
- Compute aggregate statistics with `agg`
- Group agents by attributes with `groupby`
- Combine these tools to answer questions about your model's state The *next* tutorial covers how to use AgentSet for **activating** agents (calling their methods). We separate these concerns because querying agents and activating agents are conceptually different — you'll often query first and activate a subset.

#### 2.4.2.6.2.3 IN COLAB? - Run the next cell

```
# %pip install --quiet mesa[rec]
```

#### 2.4.2.6.2.4 Import Dependencies

```
import numpy as np
import pandas as pd
import seaborn as sns

import mesa
```

#### 2.4.2.6.2.5 Setup: The Wealth Model with Ethnicities

We'll use a slightly enriched version of the Boltzmann Wealth Model. Each agent has a `wealth` (starting at 1) and an `ethnicity` (randomly assigned from "Green", "Blue", or "Mixed"). This gives us meaningful attributes to query, filter, and group by.

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth and an ethnicity."""

    def __init__(self, model, ethnicity):
        super().__init__(model)
        self.wealth = 1
        self.ethnicity = ethnicity

    def exchange(self):
        if self.wealth > 0:
            other_agent = self.random.choice(self.model.agents)
            other_agent.wealth += 1
            self.wealth -= 1


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=100):
        super().__init__()
        ethnicities = ["Green", "Blue", "Mixed"]
        MoneyAgent.create_agents(
            model=self,
            n=n,
            ethnicity=self.random.choices(ethnicities, k=n),
        )

    def step(self):
        self.agents.shuffle_do("exchange")
```

Let's create a model and run it for a while so agents have different wealth levels.

```
model = MoneyModel(100)
model.run_for(50)
```

#### 2.4.2.6.2.6 What is an AgentSet?

Every Mesa model automatically tracks all its agents in `model.agents`. This is an **AgentSet** — an ordered collection of agents that provides powerful methods for querying, filtering, and manipulating groups of agents. You never need to create an AgentSet yourself for basic usage. Mesa creates and maintains `model.agents` automatically whenever agents are added to or removed from the model. Let's look at some basics:

```
# How many agents are in the model?
print(f"Total agents: {len(model.agents)}")

# Iterate over agents (just the first 5 for brevity)
for agent in model.agents.select(at_most=5):
    print(
```

(continues on next page)

(continued from previous page)

```
        f"  Agent {agent.unique_id}: wealth={agent.wealth}, ethnicity={agent.ethnicity}"
    )
```

```
Total agents: 100
  Agent 1: wealth=0, ethnicity=Green
  Agent 2: wealth=0, ethnicity=Green
  Agent 3: wealth=1, ethnicity=Mixed
  Agent 4: wealth=3, ethnicity=Green
  Agent 5: wealth=0, ethnicity=Mixed
```

#### 2.4.2.6.2.7 Retrieving Attribute Values with `get`

The `get` method retrieves attribute values from every agent in the set, returning them as a list. This is useful whenever you want to inspect or analyze a particular attribute across all agents.

```
# Get all wealth values
all_wealth = model.agents.get("wealth")
print(f"First 10 wealth values: {all_wealth[:10]}")
print(f"Total wealth in economy: {sum(all_wealth)}")
```

```
First 10 wealth values: [0, 0, 1, 3, 0, 1, 0, 0, 0, 1]
Total wealth in economy: 100
```

You can also retrieve multiple attributes at once by passing a list of attribute names. This returns a list of lists — one inner list per agent.

```
# Get both wealth and ethnicity for each agent
wealth_and_ethnicity = model.agents.get(["wealth", "ethnicity"])
print("First 5 agents (wealth, ethnicity):")
for values in wealth_and_ethnicity[:5]:
    print(f"  {values}")
```

```
First 5 agents (wealth, ethnicity):
  [0, 'Green']
  [0, 'Green']
  [1, 'Mixed']
  [3, 'Green']
  [0, 'Mixed']
```

#### 2.4.2.6.2.8 Handling missing attributes

If some agents might not have a particular attribute, you can use the `handle_missing` parameter. By default, `get` raises an `AttributeError` for missing attributes. Setting `handle_missing="default"` returns a default value instead.

```
# This would raise AttributeError if any agent lacks 'wealth':
# model.agents.get("nonexistent_attr")

# Safe alternative - returns None for missing attributes:
values = model.agents.get("wealth", handle_missing="default", default_value=0)
print(f"Retrieved {len(values)} values safely")
```

```
Retrieved 100 values safely
```

#### 2.4.2.6.2.9 Filtering Agents with `select`

The `select` method filters agents based on criteria, returning a new AgentSet containing only the agents that match. This is one of the most frequently used AgentSet operations.

#### 2.4.2.6.2.10 Basic filtering with a function

Pass a function (often a lambda) that takes an agent and returns `True` or `False`:

```
# Select only wealthy agents (wealth >= 3)
rich_agents = model.agents.select(lambda a: a.wealth >= 3)
print(f"Rich agents (wealth >= 3): {len(rich_agents)}")

# Select agents with no money
broke_agents = model.agents.select(lambda a: a.wealth == 0)
print(f"Broke agents (wealth == 0): {len(broke_agents)}")
```

```
Rich agents (wealth >= 3): 15
Broke agents (wealth == 0): 47
```

#### 2.4.2.6.2.11 Filtering by agent type

If your model has multiple agent classes, you can filter by type using the `agent_type` parameter. This is faster than using a lambda with `isinstance`.

```
# In this model we only have one type, but the syntax would be:
money_agents = model.agents.select(agent_type=MoneyAgent)
print(f"MoneyAgents: {len(money_agents)}")
```

```
MoneyAgents: 100
```

#### 2.4.2.6.2.12 Limiting results with `at_most`

The `at_most` parameter limits how many agents are returned. This is useful when you only need a few agents and want to avoid processing the entire set.

- Pass an **integer** to get at most that many agents
- Pass a **float between 0 and 1** to get at most that fraction of agents **Important:** `at_most` returns the *first* matching agents, not a random sample. If you want a random subset, call `shuffle()` first (covered in the activation tutorial).

```
# Get at most 5 rich agents
some_rich = model.agents.select(lambda a: a.wealth >= 2, at_most=5)
print(f"Up to 5 rich agents: {len(some_rich)}")

# Get roughly 10% of agents
ten_percent = model.agents.select(at_most=0.1)
print(f"~10% of agents: {len(ten_percent)}")
```

```
Up to 5 rich agents: 5
~10% of agents: 10
```

#### 2.4.2.6.2.13 Combining criteria

You can combine `filter_func`, `agent_type`, and `at_most` in a single call. All criteria are applied together (logical AND):

```
# At most 10 MoneyAgents with wealth > 0
subset = model.agents.select(
    filter_func=lambda a: a.wealth > 0,
    agent_type=MoneyAgent,
    at_most=10,
)
print(f"Subset size: {len(subset)}")
```

```
Subset size: 10
```

#### 2.4.2.6.2.14 Chaining selects

Since `select` returns an AgentSet, you can chain multiple calls. Each successive `select` narrows the set further:

```
# First get Green agents, then filter for wealthy ones
wealthy_green = model.agents.select(lambda a: a.ethnicity == "Green").select(
    lambda a: a.wealth >= 3
)
print(f"Wealthy Green agents: {len(wealthy_green)}")
```

```
Wealthy Green agents: 2
```

#### 2.4.2.6.2.15 Computing Aggregates with `agg`

The `agg` method computes aggregate statistics over an attribute for all agents in the set. Pass the attribute name and a function (like `min`, `max`, `sum`, or `np.mean`).

```
# Average wealth across all agents
avg_wealth = model.agents.agg("wealth", np.mean)
print(f"Average wealth: {avg_wealth:.2f}")

# Min and max wealth
min_wealth = model.agents.agg("wealth", min)
max_wealth = model.agents.agg("wealth", max)
print(f"Wealth range: {min_wealth} to {max_wealth}")

# Total wealth (should equal the number of agents, since money is conserved)
total = model.agents.agg("wealth", sum)
print(f"Total wealth: {total}")
```

```
Average wealth: 1.00
Wealth range: 0 to 7
Total wealth: 100
```

#### 2.4.2.6.2.16 Multiple aggregations at once

You can pass a list of functions to compute multiple statistics in a single call:

```
min_w, max_w, avg_w = model.agents.agg("wealth", [min, max, np.mean])
print(f"Min: {min_w}, Max: {max_w}, Mean: {avg_w:.2f}")
```

```
Min: 0, Max: 7, Mean: 1.00
```

#### 2.4.2.6.2.17 Aggregating subsets

Since `select` returns an AgentSet, you can chain `select` and `agg` to compute statistics for specific subgroups:

```
# Average wealth of Green agents only
green_avg = model.agents.select(lambda a: a.ethnicity == "Green").agg("wealth", np.mean)
blue_avg = model.agents.select(lambda a: a.ethnicity == "Blue").agg("wealth", np.mean)
mixed_avg = model.agents.select(lambda a: a.ethnicity == "Mixed").agg("wealth", np.mean)

print(
    f"Average wealth - Green: {green_avg:.2f}, Blue: {blue_avg:.2f}, Mixed: {mixed_avg:.
↪2f}"
)
```

```
Average wealth - Green: 0.77, Blue: 1.17, Mixed: 0.98
```

This pattern of select-then-aggregate is common, but when you want to do it for *all* groups at once, `groupby` is more elegant.

#### 2.4.2.6.2.18 Grouping Agents with `groupby`

The `groupby` method splits agents into groups based on an attribute (or a callable), returning a `GroupBy` object. This is conceptually similar to pandas' `groupby` and is ideal when you want to analyze or act on agents by category.

```
# Group agents by ethnicity
grouped = model.agents.groupby("ethnicity")

# See how many agents are in each group
print("Agents per ethnicity:", grouped.count())
```

```
Agents per ethnicity: {'Green': 22, 'Mixed': 42, 'Blue': 36}
```

#### 2.4.2.6.2.19 Iterating over groups

A `GroupBy` object is iterable. Each iteration yields a `(group_name, agent_set)` tuple:

```
for ethnicity, group in grouped:
    avg = group.agg("wealth", np.mean)
    print(f"  {ethnicity}: {len(group)} agents, avg wealth = {avg:.2f}")
```

```
  Green: 22 agents, avg wealth = 0.77
  Mixed: 42 agents, avg wealth = 0.98
  Blue: 36 agents, avg wealth = 1.17
```

#### 2.4.2.6.2.20 Aggregating across groups

The `agg` method on `GroupBy` computes an aggregate for each group in one call:

```
# Mean wealth by ethnicity
mean_by_group = grouped.agg("wealth", np.mean)
print("Mean wealth by ethnicity:", mean_by_group)
```

```
Mean wealth by ethnicity: {'Green': np.float64(0.7727272727272727), 'Mixed': np.
↪float64(0.9761904761904762), 'Blue': np.float64(1.1666666666666667)}
```

#### 2.4.2.6.2.21 Grouping by a function

Instead of an attribute name, you can pass a callable that computes the group key for each agent. This is useful for creating custom groupings:

```
# Group agents into wealth brackets
def wealth_bracket(agent):
    if agent.wealth == 0:
        return "broke"
    elif agent.wealth <= 2:
        return "modest"
    else:
        return "wealthy"


brackets = model.agents.groupby(wealth_bracket)
print("Agents per wealth bracket:", brackets.count())
```

```
Agents per wealth bracket: {'broke': 47, 'modest': 38, 'wealthy': 15}
```

#### 2.4.2.6.2.22 Setting Attributes with `set`

The `set` method assigns a value to an attribute for all agents in the set. This is useful for bulk updates — for example, applying a policy change to a group of agents.

```
# Give all broke agents a subsidy of 1
broke = model.agents.select(lambda a: a.wealth == 0)
print(f"Broke agents before subsidy: {len(broke)}")

broke.set("wealth", 1)

# Verify
still_broke = model.agents.select(lambda a: a.wealth == 0)
print(f"Broke agents after subsidy: {len(still_broke)}")
```

```
Broke agents before subsidy: 47
Broke agents after subsidy: 0
```

**Note:** `set` modifies agents in place and returns the AgentSet, so you can chain it:

```
model.agents.select(lambda a: a.wealth > 10).set("taxed", True)
```

#### 2.4.2.6.2.23 Sorting Agents with `sort`

The `sort` method orders agents by an attribute or a custom key function. By default, it returns a new sorted AgentSet (use `inplace=True` to sort in place).

```
# Sort by wealth (descending by default)
richest_first = model.agents.sort("wealth")
top_5 = richest_first.select(at_most=5)
print("Top 5 wealthiest agents:")
for agent in top_5:
    print(f"  Agent {agent.unique_id}: wealth={agent.wealth}")

# Sort ascending
poorest_first = model.agents.sort("wealth", ascending=True)
bottom_5 = poorest_first.select(at_most=5)
print("\nBottom 5:")
for agent in bottom_5:
    print(f"  Agent {agent.unique_id}: wealth={agent.wealth}")
```

```
Top 5 wealthiest agents:
  Agent 33: wealth=7
  Agent 32: wealth=5
  Agent 42: wealth=4
  Agent 95: wealth=4
  Agent 97: wealth=4

Bottom 5:
  Agent 1: wealth=1
  Agent 2: wealth=1
  Agent 3: wealth=1
  Agent 5: wealth=1
  Agent 6: wealth=1
```

#### 2.4.2.6.2.24 Converting to a List

If you need standard list operations like indexing or slicing, use the `to_list()` method to convert the AgentSet to a plain Python list:

```
agent_list = model.agents.to_list()
print(f"First agent: {agent_list[0].unique_id}")
print(f"Last agent: {agent_list[-1].unique_id}")
```

```
First agent: 1
Last agent: 100
```

#### 2.4.2.6.2.25 Putting It Together: Analyzing the Model

Let's combine what we've learned to produce a summary analysis of the model state.

```
print("=== Model Summary After 50 Steps ===\n")

# Overall statistics
```

(continues on next page)

(continued from previous page)

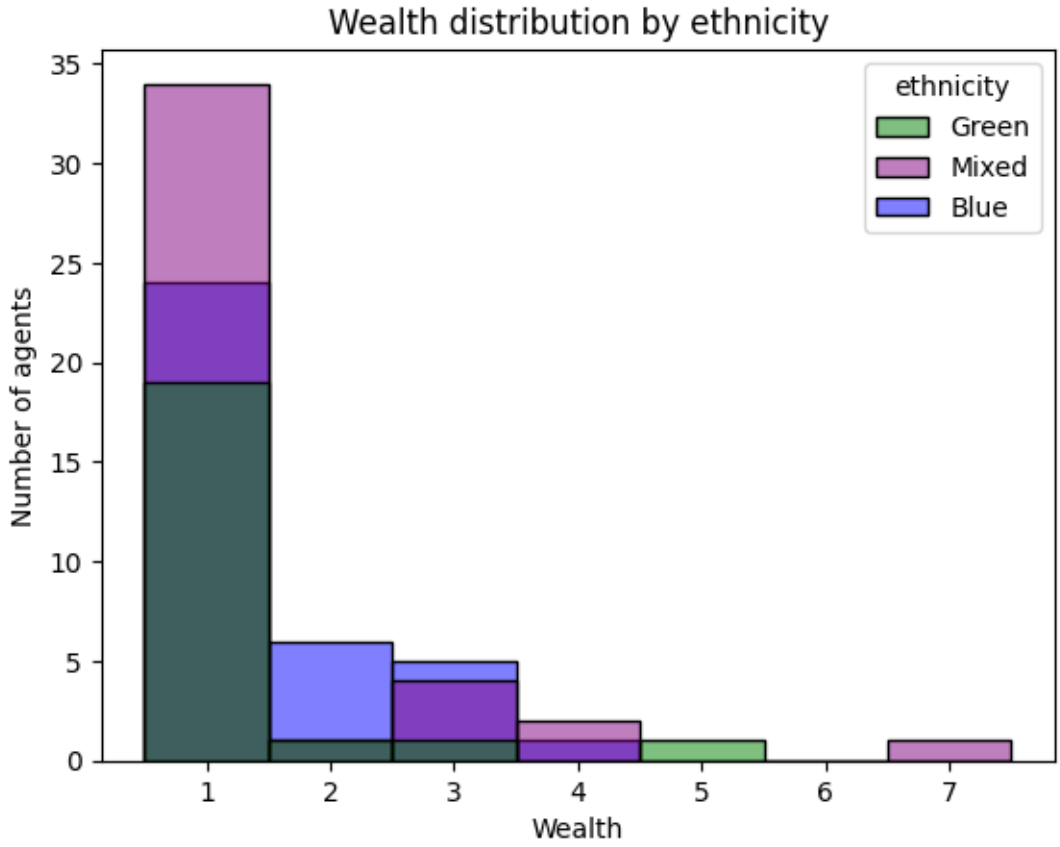
```
min_w, max_w, avg_w, total_w = model.agents.agg("wealth", [min, max, np.mean, sum])
print(f"Agents: {len(model.agents)}")
print(
    f"Total wealth: {total_w} (conserved: {'yes' if total_w == len(model.agents) else
↪'no, subsidy applied'})"
)
print(f"Wealth range: {min_w} to {max_w}, mean: {avg_w:.2f}\n")

# By ethnicity
print("By ethnicity:")
for ethnicity, group in model.agents.groupby("ethnicity"):
    count = len(group)
    avg = group.agg("wealth", np.mean)
    broke = len(group.select(lambda a: a.wealth == 0))
    print(
        f"  {ethnicity:6s}: {count:3d} agents, avg wealth = {avg:.2f}, broke = {broke}"
    )

# Wealth distribution
print("\nWealth brackets:")
for bracket, group in model.agents.groupby(wealth_bracket):
    print(f"  {bracket:8s}: {len(group)} agents")
```

```
=== Model Summary After 50 Steps ===

Agents: 100
Total wealth: 147 (conserved: no, subsidy applied)
Wealth range: 1 to 7, mean: 1.47

By ethnicity:
  Green :  22 agents, avg wealth = 1.32, broke = 0
  Mixed :  42 agents, avg wealth = 1.50, broke = 0
  Blue  :  36 agents, avg wealth = 1.53, broke = 0

Wealth brackets:
  modest  : 85 agents
  wealthy : 15 agents
```

#### 2.4.2.6.2.26 Visualizing the Results

```
# Collect data for plotting
data = []
for agent in model.agents:
    data.append({"wealth": agent.wealth, "ethnicity": agent.ethnicity})
df = pd.DataFrame(data)

palette = {"Green": "green", "Blue": "blue", "Mixed": "purple"}
g = sns.histplot(data=df, x="wealth", hue="ethnicity", discrete=True, palette=palette)
g.set(
    title="Wealth distribution by ethnicity", xlabel="Wealth", ylabel="Number of agents"
)
```

```
[Text(0.5, 1.0, 'Wealth distribution by ethnicity'),
 Text(0.5, 0, 'Wealth'),
 Text(0, 0.5, 'Number of agents')]
```




#### 2.4.2.6.2.27 Summary

In this tutorial you learned the core AgentSet **query** methods:

| Method | Purpose |
|---|---|
| `get(attr)` | Retrieve attribute values from all agents |
| `select(func)` | Filter agents by criteria |
| `agg(attr, func)` | Compute aggregate statistics |
| `groupby(attr)` | Group agents by attribute or function |
| `set(attr, value)` | Bulk-assign attribute values |
| `sort(key)` | Order agents by attribute |
| `to_list()` | Convert to a plain Python list |
| These methods are about *inspecting* and *organizing* agents. In the next tutorial, | |
| we'll cover how to **activate** agents — making them actually *do* things — using | |
| `do`, `shuffle_do`, and `map`. | |

#### 2.4.2.6.2.28 Next Steps

Check out the *Agent Activation tutorial* to learn how to make your agents act, in different orders and patterns.

### 2.4.2.6.3 Agent Activation

#### 2.4.2.6.3.1 The Boltzmann Wealth Model

**Important:**

- If you are just exploring Mesa and want the fastest way to execute the code we recommend executing this tutorial online in a Colab notebook.  or if you do not have a Google account you can use  (This can take 30 seconds to 5 minutes to load)
- If you are running locally, please ensure you have the latest Mesa version installed.

#### 2.4.2.6.3.2 Tutorial Description

In the *previous tutorial* you learned how to query, filter, and group agents using AgentSet. Now we'll cover how to make agents actually **do** things — and why the *order* and *pattern* of activation matters. By the end of this tutorial you will know how to:

- Activate agents sequentially (`do`) and in random order (`shuffle_do`)
- Collect return values with `map`
- Combine `select` with activation for conditional execution
- Implement common activation patterns: simultaneous, staged, and type-based
- Understand why activation order affects model outcomes

#### 2.4.2.6.3.3 IN COLAB? - Run the next cell

```
# %pip install --quiet mesa[rec]
```

#### 2.4.2.6.3.4 Import Dependencies

```
import numpy as np
import seaborn as sns

import mesa
```

#### 2.4.2.6.3.5 `do` and `shuffle_do`: The Core Activation Methods

AgentSet provides two primary methods for activating agents:

- **`do(method)`** — Calls the named method on each agent, in the current order of the set.
- **`shuffle_do(method)`** — Randomly reorders agents, then calls the method on each. Both accept a method name (string) or a callable. Additional arguments are passed through to each agent's method. Let's see these in action with a minimal model.

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(model)
        self.wealth = 1

    def exchange(self):
        if self.wealth > 0:
            other = self.random.choice(self.model.agents)
            other.wealth += 1
            self.wealth -= 1


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10):
        super().__init__()
        MoneyAgent.create_agents(model=self, n=n)

    def step(self):
        # Random activation - each agent acts in a random order
        self.agents.shuffle_do("exchange")
```

```
model = MoneyModel(10)
model.run_for(30)

wealth = model.agents.get("wealth")
print(f"Wealth distribution: {sorted(wealth, reverse=True)}")
print(f"Total wealth: {sum(wealth)} (should be {len(model.agents)})")
```

```
Wealth distribution: [3, 3, 2, 1, 1, 0, 0, 0, 0, 0]
Total wealth: 10 (should be 10)
```

#### 2.4.2.6.3.6 Why Activation Order Matters

The order in which agents act can significantly affect model outcomes. Consider a simple scenario: if Agent A gives money to Agent B, and then Agent B gives money away, Agent B now has more to give. If the order were reversed — B acts first, then A gives to B — the outcome differs. This is a well-studied phenomenon in agent-based modeling. Comer (2014) showed that activation order can materially impact emergent behavior. Let's demonstrate this by comparing `do` (fixed order) with `shuffle_do` (random order).

```
# Fixed order - same agent always goes first
class FixedOrderModel(mesa.Model):
    def __init__(self, n=10):
        super().__init__()
        MoneyAgent.create_agents(model=self, n=n)

    def step(self):
        self.agents.do("exchange")  # Same order every step


# Random order - different every step
class RandomOrderModel(mesa.Model):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, n=10):
        super().__init__()
        MoneyAgent.create_agents(model=self, n=n)

    def step(self):
        self.agents.shuffle_do("exchange")


# Run both multiple times and compare Gini coefficients
def gini(model):
    x = sorted(model.agents.get("wealth"))
    n = len(x)
    B = sum(xi * (n - i) for i, xi in enumerate(x)) / (n * sum(x))
    return 1 + (1 / n) - 2 * B


fixed_ginis = []
random_ginis = []
for _ in range(50):
    m = FixedOrderModel(50)
    m.run_for(100)
    fixed_ginis.append(gini(m))

    m = RandomOrderModel(50)
    m.run_for(100)
    random_ginis.append(gini(m))

print(
    f"Fixed order  - mean Gini: {np.mean(fixed_ginis):.3f} (std: {np.std(fixed_ginis):.
↪3f})"
)
print(
    f"Random order - mean Gini: {np.mean(random_ginis):.3f} (std: {np.std(random_ginis):.
↪3f})"
)
```

```
Fixed order  - mean Gini: 0.625 (std: 0.059)
Random order - mean Gini: 0.621 (std: 0.043)
```

**General guidance:** Use `shuffle_do` unless your model specifically requires a fixed activation order. Random activation avoids systematic biases where early-acting agents have an inherent advantage.

#### 2.4.2.6.3.7 Using Callables with `do`

Instead of passing a method name as a string, you can pass a callable function directly. The function receives each agent as its first argument:

```
# Using a callable instead of a method name
def tax_agent(agent):
    """Take 10% tax from agents with wealth > 5."""
    if agent.wealth > 5:
        tax = agent.wealth // 10
```

(continues on next page)

(continued from previous page)

```
        agent.wealth -= tax


model = MoneyModel(50)
model.run_for(100)

print(f"Max wealth before tax: {model.agents.agg('wealth', max)}")
model.agents.do(tax_agent)
print(f"Max wealth after tax: {model.agents.agg('wealth', max)}")
```

```
Max wealth before tax: 5
Max wealth after tax: 5
```

#### 2.4.2.6.3.8 Collecting Results with `map`

While `do` calls a method and discards the return values, `map` calls a method and **returns the results** as a list. Use `map` when each agent computes something you need to collect.

```
class ReportingAgent(mesa.Agent):
    def __init__(self, model):
        super().__init__(model)
        self.wealth = 1
        self.age = self.random.randint(18, 80)

    def report_status(self):
        return {"id": self.unique_id, "wealth": self.wealth, "age": self.age}

    def exchange(self):
        if self.wealth > 0:
            other = self.random.choice(self.model.agents)
            other.wealth += 1
            self.wealth -= 1


class ReportingModel(mesa.Model):
    def __init__(self, n=10):
        super().__init__()
        ReportingAgent.create_agents(model=self, n=n)

    def step(self):
        self.agents.shuffle_do("exchange")


model = ReportingModel(10)
model.run_for(20)

# Collect status reports from all agents
reports = model.agents.map("report_status")
print("Agent reports:")
for r in reports[:5]:
    print(f"  {r}")
```

```
Agent reports:
  {'id': 1, 'wealth': 2, 'age': 68}
  {'id': 2, 'wealth': 1, 'age': 34}
  {'id': 3, 'wealth': 0, 'age': 34}
  {'id': 4, 'wealth': 1, 'age': 50}
  {'id': 5, 'wealth': 1, 'age': 75}
```

You can also use `map` with a callable:

```
# Calculate each agent's wealth-to-age ratio
ratios = model.agents.map(lambda a: a.wealth / a.age)
print(f"Wealth/age ratios (first 5): {[f'{r:.3f}' for r in ratios[:5]]}")
```

```
Wealth/age ratios (first 5): ['0.029', '0.029', '0.000', '0.020', '0.013']
```

#### 2.4.2.6.3.9 Conditional Activation with `select` + `do`

One of the most powerful patterns in Mesa is combining `select` with activation. By filtering agents first, you can activate only those that meet specific criteria. In many real-world models, not all agents act every step. Maybe only agents with sufficient energy can move, only living agents can reproduce, or only wealthy agents pay taxes.

```
class MoneyAgent(mesa.Agent):
    def __init__(self, model):
        super().__init__(model)
        self.wealth = 1

    def exchange(self):
        if self.wealth > 0:
            other = self.random.choice(self.model.agents)
            other.wealth += 1
            self.wealth -= 1

    def donate(self, recipients):
        """Give 1 unit to a random recipient."""
        if self.wealth > 0 and len(recipients) > 0:
            recipient = self.random.choice(recipients)
            recipient.wealth += 1
            self.wealth -= 1


class PolicyModel(mesa.Model):
    """A model where only rich agents donate to poor agents."""

    def __init__(self, n=100):
        super().__init__()
        MoneyAgent.create_agents(model=self, n=n)

    def step(self):
        # First: normal exchanges
        self.agents.shuffle_do("exchange")

        # Then: redistribution policy - rich donate to poor
```

(continues on next page)

(continued from previous page)

```
        rich = self.agents.select(lambda a: a.wealth >= 5)
        poor = self.agents.select(lambda a: a.wealth == 0)
        if len(rich) > 0 and len(poor) > 0:
            rich.shuffle_do("donate", poor)


model = PolicyModel(100)
model.run_for(100)

broke = len(model.agents.select(lambda a: a.wealth == 0))
rich = len(model.agents.select(lambda a: a.wealth >= 5))
print(f"After redistribution policy: {broke} broke agents, {rich} rich agents")
```

```
After redistribution policy: 40 broke agents, 0 rich agents
```

#### 2.4.2.6.3.10 Common Activation Patterns

Before Mesa 3.0, activation patterns were hard-coded into "scheduler" classes (RandomActivation, SimultaneousActivation, etc.). Now, you compose them directly from AgentSet methods. This is more flexible — you can mix and match any combination. Here are the most common patterns:

#### 2.4.2.6.3.11 Sequential Activation

Agents act in a fixed order. The simplest pattern, but can introduce systematic bias.

```
self.agents.do("step")
```

#### 2.4.2.6.3.12 Random Activation

Agents act in a new random order each step. The most common default.

```
self.agents.shuffle_do("step")
```

#### 2.4.2.6.3.13 Simultaneous Activation

All agents first compute their next state, then all advance at once. This prevents early-acting agents from influencing later ones within the same step. Classic examples include Conway's Game of Life and Schelling's segregation model.

```
self.agents.do("compute_next_state")
self.agents.do("advance")
```

#### 2.4.2.6.3.14 Staged Activation

Agents perform multiple actions per step, in a defined sequence of stages. For example, agents might first move, then eat, then reproduce.

```
for stage in ["move", "eat", "reproduce"]:
    self.agents.shuffle_do(stage)
```

#### 2.4.2.6.3.15 Type-Based Activation

In models with multiple agent types, you often want each type to act separately. For example, predators and prey might take turns. Use `agents_by_type` to access the AgentSet for each type:

```
for agent_type in self.agent_types:
    self.agents_by_type[agent_type].shuffle_do("step")
```

Or target specific types:

```
self.agents_by_type[Prey].shuffle_do("step")
self.agents_by_type[Predator].shuffle_do("step")
```

#### 2.4.2.6.3.16 Combining Patterns

The real power is in combining these patterns freely. Here's an example that uses staged, type-based, and conditional activation all at once:

```
class Prey(mesa.Agent):
    def __init__(self, model):
        super().__init__(model)
        self.energy = 5

    def move(self):
        self.energy -= 1

    def eat(self):
        self.energy += self.random.randint(0, 2)

    def reproduce(self):
        if self.energy > 8:
            self.energy -= 4
            Prey(self.model)  # New prey is automatically registered


class Predator(mesa.Agent):
    def __init__(self, model):
        super().__init__(model)
        self.energy = 10
        self.kills = 0

    def move(self):
        self.energy -= 2  # Predators use more energy

    def hunt(self):
        prey_agents = self.model.agents_by_type.get(Prey)
        if prey_agents and len(prey_agents) > 0 and self.energy > 0:
            target = self.random.choice(prey_agents)
            target.remove()
            self.energy += 5
            self.kills += 1


class EcosystemModel(mesa.Model):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, n_prey=50, n_predators=5):
        super().__init__()
        Prey.create_agents(model=self, n=n_prey)
        Predator.create_agents(model=self, n=n_predators)

    def step(self):
        # Stage 1: All agents move (random order within each type)
        for agent_type in self.agent_types:
            self.agents_by_type[agent_type].shuffle_do("move")

        # Stage 2: Type-specific actions
        if Prey in self.agents_by_type:
            self.agents_by_type[Prey].shuffle_do("eat")
        if Predator in self.agents_by_type:
            self.agents_by_type[Predator].shuffle_do("hunt")

        # Stage 3: Only prey with enough energy reproduce
        if Prey in self.agents_by_type:
            fertile = self.agents_by_type[Prey].select(lambda a: a.energy > 8)
            fertile.do("reproduce")

        # Remove dead agents (energy depleted)
        dead = self.agents.select(lambda a: a.energy <= 0)
        for agent in dead:
            agent.remove()


eco = EcosystemModel(50, 5)
eco.run_for(20)

n_prey = len(eco.agents_by_type.get(Prey, []))
n_pred = len(eco.agents_by_type.get(Predator, []))
print(f"After 20 steps: {n_prey} prey, {n_pred} predators, {len(eco.agents)} total")
```

```
After 20 steps: 0 prey, 5 predators, 5 total
```

This example demonstrates several key points:

- **Staged activation**: All agents move first, then type-specific actions happen
- **Type-based activation**: Prey eat while predators hunt
- **Conditional activation**: Only fertile prey reproduce
- **Dynamic agent creation/removal**: Prey reproduce and dead agents are removed None of this required any special scheduler class — just `do`, `shuffle_do`, and `select`, composed in the order that makes sense for your model.

#### 2.4.2.6.3.17 Summary

| Method | Purpose | Returns |
|---|---|---|
| `do(method)` | Call method on each agent (fixed order) | The AgentSet |
| `shuffle_do(method)` | Call method on each agent (random order) | The AgentSet |
| `map(method)` | Call method on each agent and collect results | List of results |
| **Key activation patterns:** | | |

- **Random**: `agents.shuffle_do("step")` — use this as your default
- **Sequential**: `agents.do("step")` — when order is intentional
- **Simultaneous**: `agents.do("compute")` then `agents.do("advance")`
- **Staged**: loop over stages, calling `do`/`shuffle_do` for each
- **Type-based**: use `agents_by_type[Type].shuffle_do("step")`
- **Conditional**: `agents.select(condition).do("step")` Combine these freely to express exactly the activation logic your model needs.

#### 2.4.2.6.3.18 Next Steps

Check out the *Event Scheduling & Time tutorial* to learn how to schedule events at specific times, create recurring events, and control how your simulation progresses through time.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

### 2.4.2.6.4 Event Scheduling & Time

**Important:**

- If you are just exploring Mesa and want the fastest way to execute the code we recommend executing this tutorial online in a Colab notebook. or if you do not have a Google account you can use (This can take 30 seconds to 5 minutes to load)
- If you are running locally, please ensure you have the latest Mesa version installed.

#### 2.4.2.6.4.1 Tutorial Description

In the previous tutorials, you created agents, queried them with AgentSet, and learned activation patterns. In this tutorial, we cover **time** — how Mesa models progress through time, how you run simulations, and how to schedule events that happen at specific moments. By the end of this tutorial you will know how to:

- Understand how `model.time` advances during a simulation
- Run models with `run_for()` and `run_until()`
- Schedule one-off events at absolute or relative times
- Schedule recurring events with `Schedule`
- Control event priority
- Cancel events and stop recurring generators
- Combine step-based agent activation with discrete events in a single model

#### 2.4.2.6.4.2 IN COLAB? - Run the next cell

```
%pip install --quiet mesa[rec]
```

```
Note: you may need to restart the kernel to use updated packages.
```

#### 2.4.2.6.4.3 Import Dependencies

```
import numpy as np

import mesa
from mesa.time import Priority, Schedule
```

#### 2.4.2.6.4.4 How Time Works in Mesa

Mesa models track simulation time with:

- **`model.time`** — A float representing the current simulation time (starts at 0.0).

#### 2.4.2.6.4.5 The default step mechanism

When you define a `step()` method on your model and advance time, Mesa wraps your step in an internal event system. Each step:

1. Advances `model.time` by 1.0
2. Executes your `step()` method For event-driven simulations, prefer these methods:

- **`model.run_for(n)`** — Advance time by `n` units (executing `n` steps for standard models)
- **`model.run_until(t)`** — Advance time until `model.time` reaches `t` These methods are the primary way to run any Mesa model.

#### 2.4.2.6.4.6 A quick example

```
class SimpleModel(mesa.Model):
    def __init__(self):
        super().__init__()
        self.step_count = 0

    def step(self):
        self.step_count += 1
        print(f"  Step {self.step_count} at time {self.time:.1f}")


model = SimpleModel()
print("Initial state:")
print(f"  step_count={model.step_count}, time={model.time}")
print("\nRunning for 3 time units:")
model.run_for(3)
print(f"\nFinal state: step_count={model.step_count}, time={model.time}")
```

```
Initial state:
  step_count=0, time=0.0

Running for 3 time units:
  Step 1 at time 1.0
  Step 2 at time 2.0
  Step 3 at time 3.0

Final state: step_count=3, time=3.0
```

Notice that after `run_for(3)`, both the example's `step_count` and `time` are at 3. For standard step-based models, a user-defined step counter like this advances in lockstep with time. But as we'll see, events can fire at *any* time — including between steps.

#### 2.4.2.6.4.7 `run_for` vs `run_until`

Both methods advance time and process any scheduled events (including the default step events) along the way. The difference is simple:

- `run_for(duration)` advances time by the specified **duration** from the current time
- `run_until(end_time)` advances time to the specified **absolute time**

```
model = SimpleModel()

print("run_for(2):")
model.run_for(2)
print(f"  → time is now {model.time}\n")

print("run_until(5):")
model.run_until(5)
print(f"  → time is now {model.time}\n")

print("run_for(3) more:")
model.run_for(3)
print(f"  → time is now {model.time}")
```

```
run_for(2):
  Step 1 at time 1.0
  Step 2 at time 2.0
  → time is now 2.0

run_until(5):
  Step 3 at time 3.0
  Step 4 at time 4.0
  Step 5 at time 5.0
  → time is now 5

run_for(3) more:
  Step 6 at time 6.0
  Step 7 at time 7.0
  Step 8 at time 8.0
  → time is now 8
```

`run_until` is particularly useful when you have a fixed end time for your simulation, or when coordinating with external time references:

```
model.run_until(365)  # Run for one simulated year
```

#### 2.4.2.6.4.8 Scheduling One-Off Events

Beyond the regular step cycle, Mesa lets you schedule **events** — functions that execute at specific times. This is useful for modeling things that don't happen every step: policy changes, natural disasters, market shocks, seasonal effects, or any occurrence that happens at a specific point in time. Use `model.schedule_event()` to schedule a one-off event:

- **`at=`** — Schedule at an **absolute** time
- **`after=`** — Schedule at a **relative** time from now (i.e., `model.time + after`) You must specify exactly one of `at` or `after`.

```
class EconomyModel(mesa.Model):
    """A simple economy where a tax reform happens at a specific time."""

    def __init__(self, n=50):
        super().__init__()
        self.tax_rate = 0.1
        self.events_log = []

        # Create agents with wealth
        for _ in range(n):
            a = mesa.Agent(self)
            a.wealth = 10

        # Schedule a tax reform at time 5.0
        self.schedule_event(self.tax_reform, at=5.0)

        # Schedule a stimulus check 2 time units from now (so at time 2.0)
        self.schedule_event(self.stimulus, after=2.0)

    def tax_reform(self):
        self.tax_rate = 0.25
        self.events_log.append(
            f"t={self.time:.1f}: Tax reform! Rate now {self.tax_rate}"
        )

    def stimulus(self):
        for agent in self.agents:
            agent.wealth += 5
        self.events_log.append(f"t={self.time:.1f}: Stimulus! Everyone gets 5 units")

    def step(self):
        # Simple taxation each step
        for agent in self.agents:
            tax = int(agent.wealth * self.tax_rate)
            agent.wealth -= tax


model = EconomyModel(10)
```

(continues on next page)

(continued from previous page)

```
model.run_for(7)

print("Events that occurred:")
for event in model.events_log:
    print(f"  {event}")

print(f"\nFinal tax rate: {model.tax_rate}")
avg_wealth = model.agents.agg("wealth", np.mean)
print(f"Average wealth: {avg_wealth:.1f}")
```

```
Events that occurred:
  t=2.0: Stimulus! Everyone gets 5 units
  t=5.0: Tax reform! Rate now 0.25

Final tax rate: 0.25
Average wealth: 7.0
```

**Key point:** Events fire *during* time advancement. When `run_for(7)` processes time, it encounters the stimulus event at t=2.0 and the tax reform at t=5.0, executing them at the correct moments. Your `step()` method also fires at t=1.0, 2.0, 3.0, etc. as scheduled events under the hood.

#### 2.4.2.6.4.9 Canceling events

`schedule_event` returns an `Event` object. You can cancel it before it fires:

```
class CancelDemo(mesa.Model):
    def __init__(self):
        super().__init__()

        # Schedule two events
        self.event_a = self.schedule_event(self.event_a_fn, at=3.0)
        self.event_b = self.schedule_event(self.event_b_fn, at=5.0)

        # Cancel event B before it fires
        self.event_b.cancel()

    def event_a_fn(self):
        print(f"  Event A fired at t={self.time:.1f}")

    def event_b_fn(self):
        print(f"  Event B fired at t={self.time:.1f}")

    def step(self):
        pass


model = CancelDemo()
print("Running - Event B was canceled:")
model.run_for(6)
print("  (Event B never fired)")
```

```
Running - Event B was canceled:
  Event A fired at t=3.0
  (Event B never fired)
```

Cancellation is useful when model conditions change. For example, you might schedule a disaster event but cancel it if agents take preventive action.

#### 2.4.2.6.4.10 Scheduling Recurring Events

Many models have things that happen repeatedly but not every step — quarterly reports, seasonal cycles, periodic inspections, or interest payments. Use `model.schedule_recurring()` with a `Schedule` to define these patterns. The `Schedule` dataclass specifies:

- **`interval`** — Time between executions (required)
- **`start`** — When to begin (default: current time + interval)
- **`end`** — When to stop (default: never)
- **`count`** — Maximum number of executions (default: unlimited)

```python
class SeasonalModel(mesa.Model):
    """A model with regular seasonal events."""

    def __init__(self, n=20):
        super().__init__()
        self.season = "spring"
        self.season_log = []

        for _ in range(n):
            a = mesa.Agent(self)
            a.wealth = 10

        # Change season every 3 time units, starting at t=3.0
        self.schedule_recurring(
            self.change_season,
            Schedule(interval=3.0),
        )

        # Collect taxes every 5 time units, but only 4 times
        self.schedule_recurring(
            self.collect_taxes,
            Schedule(interval=5.0, count=4),
        )

    def change_season(self):
        seasons = ["spring", "summer", "autumn", "winter"]
        idx = seasons.index(self.season)
        self.season = seasons[(idx + 1) % 4]
        self.season_log.append(f"t={self.time:.1f}: Season → {self.season}")

    def collect_taxes(self):
        for agent in self.agents:
            agent.wealth -= 1
        self.season_log.append(f"t={self.time:.1f}: Taxes collected!")
```

(continues on next page)

(continued from previous page)

```
    def step(self):
        # Normal step - agents earn money
        for agent in self.agents:
            agent.wealth += self.random.randint(0, 2)


model = SeasonalModel(10)
model.run_for(20)

print("Timeline:")
for entry in model.season_log:
    print(f"  {entry}")
print(f"\nFinal season: {model.season}")
print(f"Average wealth: {model.agents.agg('wealth', np.mean):.1f}")
```

```
Timeline:
  t=3.0: Season → summer
  t=5.0: Taxes collected!
  t=6.0: Season → autumn
  t=9.0: Season → winter
  t=10.0: Taxes collected!
  t=12.0: Season → spring
  t=15.0: Taxes collected!
  t=15.0: Season → summer
  t=18.0: Season → autumn
  t=20.0: Taxes collected!

Final season: autumn
Average wealth: 25.6
```

#### 2.4.2.6.4.11 Controlling when recurring events start

By default, a recurring event first fires at `current_time + interval`. You can override this with the `start` parameter:

```
class StartDemo(mesa.Model):
    def __init__(self):
        super().__init__()

        # Fires at t=5, t=10, t=15, ... (default start)
        self.schedule_recurring(
            self.default_start_event,
            Schedule(interval=5.0),
        )

        # Fires at t=0, t=5, t=10, ... (start immediately)
        self.schedule_recurring(
            self.start_at_zero_event,
            Schedule(interval=5.0, start=0.0),
        )
```

(continues on next page)

(continued from previous page)

```
        # Fires at t=2, t=7, t=12, ... (custom start)
        self.schedule_recurring(
            self.start_at_two_event,
            Schedule(interval=5.0, start=2.0),
        )

    def default_start_event(self):
        print(f"  Default start: t={self.time:.1f}")

    def start_at_zero_event(self):
        print(f"  Start at 0:    t={self.time:.1f}")

    def start_at_two_event(self):
        print(f"  Start at 2:    t={self.time:.1f}")

    def step(self):
        pass


model = StartDemo()
print("Events firing during first 12 time units:")
model.run_for(12)
```

```
Events firing during first 12 time units:
  Start at 0:    t=0.0
  Start at 2:    t=2.0
  Default start: t=5.0
  Start at 0:    t=5.0
  Start at 2:    t=7.0
  Default start: t=10.0
  Start at 0:    t=10.0
  Start at 2:    t=12.0
```

#### 2.4.2.6.4.12 Stopping recurring events

`schedule_recurring` returns an `EventGenerator` that you can stop at any time:

```
class StopDemo(mesa.Model):
    def __init__(self):
        super().__init__()
        self.counter = 0

        # Start a recurring event
        self.ticker = self.schedule_recurring(
            self.tick,
            Schedule(interval=1.0),
        )

    def tick(self):
        self.counter += 1
```

(continues on next page)

(continued from previous page)

```
        print(f"  Tick #{self.counter} at t={self.time:.1f}")

    def step(self):
        # Stop the ticker after 5 ticks
        if self.counter >= 5 and self.ticker.is_active:
            self.ticker.stop()
            print(f"  Ticker stopped at t={self.time:.1f}")


model = StopDemo()
model.run_for(10)
print(f"\nTotal ticks: {model.counter}")
```

```
  Tick #1 at t=1.0
  Tick #2 at t=2.0
  Tick #3 at t=3.0
  Tick #4 at t=4.0
  Tick #5 at t=5.0
  Ticker stopped at t=6.0

Total ticks: 5
```

#### 2.4.2.6.4.13 Using `end` and `count` for automatic limits

Instead of manually stopping a generator, you can set limits in the `Schedule` itself:

```
# Stop after time 50.0
Schedule(interval=5.0, end=50.0)
# Execute at most 10 times
Schedule(interval=5.0, count=10)
# Both: stop after 10 executions OR after time 50.0, whichever comes first
Schedule(interval=5.0, count=10, end=50.0)
```

#### 2.4.2.6.4.14 Dynamic intervals

The `interval` parameter can be a callable that returns the next interval dynamically. The callable receives the model as its argument. This is useful for modeling processes where the frequency changes over time:

```
class AcceleratingModel(mesa.Model):
    """A model where events happen faster and faster."""

    def __init__(self):
        super().__init__()
        self.event_times = []

        # Interval starts at 4.0 and shrinks each time
        self.schedule_recurring(
            self.record_event,
            Schedule(
                interval=lambda m: max(1.0, 4.0 - m.time * 0.3),
                count=8,
```

(continues on next page)

(continued from previous page)

```
            ),
        )

    def record_event(self):
        self.event_times.append(self.time)

    def step(self):
        pass


model = AcceleratingModel()
model.run_for(25)

print("Event times (accelerating intervals):")
for i, t in enumerate(model.event_times):
    if i > 0:
        gap = t - model.event_times[i - 1]
        print(f"  t={t:.1f} (gap: {gap:.1f})")
    else:
        print(f"  t={t:.1f}")
```

```
Event times (accelerating intervals):
  t=4.0
  t=6.8 (gap: 2.8)
  t=8.8 (gap: 2.0)
  t=10.1 (gap: 1.4)
  t=11.1 (gap: 1.0)
  t=12.1 (gap: 1.0)
  t=13.1 (gap: 1.0)
  t=14.1 (gap: 1.0)
```

#### 2.4.2.6.4.15 Event Priority

When multiple events are scheduled for the same time, **priority** determines execution order. Mesa provides three priority levels:

- `Priority.HIGH` (1) — Executes first
- `Priority.DEFAULT` (5) — Normal priority
- `Priority.LOW` (10) — Executes last Lower numeric values mean higher priority. Note that the default model `step()` is scheduled at `Priority.HIGH`, so it runs before your custom events at the same time.

```
class PriorityDemo(mesa.Model):
    def __init__(self):
        super().__init__()

        # Schedule three events at the same time with different priorities
        self.schedule_event(
            self.low_priority_event,
            at=2.0,
            priority=Priority.LOW,
        )
```

(continues on next page)

(continued from previous page)

```
        self.schedule_event(
            self.high_priority_event,
            at=2.0,
            priority=Priority.HIGH,
        )
        self.schedule_event(
            self.default_priority_event,
            at=2.0,
            priority=Priority.DEFAULT,
        )

    def low_priority_event(self):
        print(f"  LOW priority event at t={self.time:.1f}")

    def high_priority_event(self):
        print(f"  HIGH priority event at t={self.time:.1f}")

    def default_priority_event(self):
        print(f"  DEFAULT priority event at t={self.time:.1f}")

    def step(self):
        if self.time == 2.0:
            print(f"  Model step (HIGH priority) at t={self.time:.1f}")


model = PriorityDemo()
print("Events at t=2.0 in execution order:")
model.run_for(3)
```

```
Events at t=2.0 in execution order:
  HIGH priority event at t=2.0
  Model step (HIGH priority) at t=2.0
  DEFAULT priority event at t=2.0
  LOW priority event at t=2.0
```

Priority is useful when the order of operations matters. For example, you might want data collection (HIGH) to happen before agent actions (DEFAULT), or environmental updates (LOW) to happen after everything else.

#### 2.4.2.6.4.16 Putting It All Together: A Complete Example

Let's build a more complete model that combines step-based agent activation with discrete events. This is a simple economy where:

- Agents exchange money every step (standard activation)
- A central bank adjusts interest rates every 10 time units (recurring event)
- A one-time economic stimulus happens at t=25 (one-off event)
- The simulation runs until t=50

```
class Citizen(mesa.Agent):
    def __init__(self, model):
        super().__init__(model)
```

(continues on next page)

(continued from previous page)

```
        self.wealth = 10
        self.savings = 0

    def exchange(self):
        """Give 1 unit to a random other agent."""
        if self.wealth > 0:
            others = [agent for agent in self.model.agents if agent is not self]
            if others:
                other = self.random.choice(others)
                other.wealth += 1
                self.wealth -= 1

    def earn_interest(self):
        """Earn interest on savings based on current rate."""
        interest = int(self.savings * self.model.interest_rate)
        self.savings += interest


class CentralBankModel(mesa.Model):
    """An economy with monetary policy events."""

    def __init__(self, n_citizens=50):
        super().__init__()
        self.interest_rate = 0.05
        self.log = []

        Citizen.create_agents(model=self, n=n_citizens)

        # Distribute initial savings randomly
        for agent in self.agents:
            agent.savings = self.random.randint(0, 20)

        # Recurring: Central bank reviews interest rate every 10 time units
        self.rate_review = self.schedule_recurring(
            self.review_interest_rate,
            Schedule(interval=10.0, start=10.0),
        )

        # Recurring: Interest is paid every 5 time units
        self.schedule_recurring(
            self.pay_interest,
            Schedule(interval=5.0),
        )

        # One-off: Economic stimulus at t=25
        self.schedule_event(self.economic_stimulus, at=25.0)

    def review_interest_rate(self):
        """Central bank adjusts rate based on average wealth."""
        avg_wealth = self.agents.agg("wealth", np.mean)
        if avg_wealth < 8:
            self.interest_rate = min(0.15, self.interest_rate + 0.02)
```

(continues on next page)

(continued from previous page)

```
            action = "raised"
        elif avg_wealth > 12:
            self.interest_rate = max(0.01, self.interest_rate - 0.02)
            action = "lowered"
        else:
            action = "held"
        self.log.append(
            f"t={self.time:5.1f} | Rate review: {action} to {self.interest_rate:.0%} "
            f"(avg wealth: {avg_wealth:.1f})"
        )

    def pay_interest(self):
        """Pay interest to all citizens."""
        total_paid = 0
        for agent in self.agents:
            interest = int(agent.savings * self.interest_rate)
            agent.savings += interest
            total_paid += interest
        self.log.append(f"t={self.time:5.1f} | Interest paid: {total_paid} total")

    def economic_stimulus(self):
        """One-time stimulus: every citizen gets 5 units."""
        for agent in self.agents:
            agent.wealth += 5
        self.log.append(f"t={self.time:5.1f} | *** STIMULUS: +5 to all citizens ***")

    def step(self):
        """Regular step: agents exchange money."""
        self.agents.shuffle_do("exchange")

        # Some agents save a portion of their wealth
        for agent in self.agents.select(lambda a: a.wealth > 3):
            save_amount = agent.wealth // 4
            agent.wealth -= save_amount
            agent.savings += save_amount


# Run the simulation
model = CentralBankModel(50)
model.run_until(50)

print("=== Central Bank Economy: Event Log ===\n")
for entry in model.log:
    print(f"  {entry}")

print(f"\n=== Final State (t={model.time:.0f}) ===")
print(f"Interest rate: {model.interest_rate:.0%}")
print(f"Avg wealth: {model.agents.agg('wealth', np.mean):.1f}")
print(f"Avg savings: {model.agents.agg('savings', np.mean):.1f}")
total = sum(a.wealth + a.savings for a in model.agents)
print(f"Total money in economy: {total}")
```

```
=== Central Bank Economy: Event Log ===

  t=  5.0 | Interest paid: 12 total
  t= 10.0 | Rate review: raised to 7% (avg wealth: 2.1)
  t= 10.0 | Interest paid: 29 total
  t= 15.0 | Interest paid: 33 total
  t= 20.0 | Rate review: raised to 9% (avg wealth: 1.2)
  t= 20.0 | Interest paid: 55 total
  t= 25.0 | *** STIMULUS: +5 to all citizens ***
  t= 25.0 | Interest paid: 60 total
  t= 30.0 | Rate review: raised to 11% (avg wealth: 2.7)
  t= 30.0 | Interest paid: 109 total
  t= 35.0 | Interest paid: 123 total
  t= 40.0 | Rate review: raised to 13% (avg wealth: 1.4)
  t= 40.0 | Interest paid: 171 total
  t= 45.0 | Interest paid: 198 total
  t= 50.0 | Rate review: raised to 15% (avg wealth: 1.0)
  t= 50.0 | Interest paid: 264 total

=== Final State (t=50) ===
Interest rate: 15%
Avg wealth: 1.0
Avg savings: 43.3
Total money in economy: 2215
```

This model demonstrates the core pattern of Mesa 3.5: the `step()` method handles regular per-step agent activation, while `schedule_event` and `schedule_recurring` handle things that happen at specific times or on different schedules. The event system manages all timing automatically — you just specify *what* should happen and *when*.

#### 2.4.2.6.4.17 When to Use Events vs Steps

| Use case | Approach |
|---|---|
| Agents act every time unit | `step()` with `agents.shuffle_do()` |
| Something happens once at a known time | `schedule_event(fn, at=...)` |
| Something happens repeatedly on a schedule | `schedule_recurring(fn, Schedule(...))` |
| Something happens after a delay from now | `schedule_event(fn, after=...)` |
| Different processes run at different frequencies | Combine step + recurring events |
| Pure discrete-event simulation (no regular steps) | Use only `schedule_event` / `schedule_recurring` |
| The step mechanism itself is implemented as a recurring event under the hood (with | |
| `Priority.HIGH`, interval 1.0, starting at t=1.0). This means steps and custom events | |
| coexist naturally in the same time-ordered event queue. | |

#### 2.4.2.6.4.18 Summary

**Running models:**

- `model.run_for(n)` — Advance time by `n` units
- `model.run_until(t)` — Advance time to absolute time `t` **One-off events:**

- `model.schedule_event(fn, at=t)` — Fire at absolute time `t`
- `model.schedule_event(fn, after=d)` — Fire `d` time units from now
- `event.cancel()` — Cancel before it fires **Recurring events:**
- `model.schedule_recurring(fn, Schedule(interval=...))` — Repeat on a schedule
- `Schedule(interval, start, end, count)` — Control timing, limits
- `generator.stop()` — Stop a recurring event
- `generator.is_active` — Check if still running **Priority:** `Priority.HIGH` → `Priority.DEFAULT` → `Priority.LOW` **Time tracking:** `model.time` (float), plus any user-defined counters you maintain in your model

#### 2.4.2.6.4.19 Next Steps

Check out the *adding space tutorial* on how to add a space to your Mesa model.

### 2.4.2.6.5 Adding Space

#### 2.4.2.6.5.1 The Boltzmann Wealth Model

If you want to get straight to the tutorial checkout these environment providers: (with Google Account) (no Google Account) (This can take 30 seconds to 5 minutes to load)

*If you are running locally, please ensure you have the latest Mesa version installed.*

#### 2.4.2.6.5.2 Tutorial Description

This tutorial extends the Boltzmann wealth model from the *Running Your First Model tutorial*, by adding Mesa's discrete space module.

In this portion, `MoneyAgents` will move in a two dimensional grid, made up of discrete cells and randomly exchange money with other agents.

*If you are starting here please see the* Running Your First Model tutorial *for dependency and start-up instructions*

#### 2.4.2.6.5.3 IN COLAB? - Run the next cell

#### 2.4.2.6.5.4 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns

import mesa
```

#### 2.4.2.6.5.5 Base Model

The below provides the base model from which we will add our space functionality.

This is from the *Running Your First Model tutorial* tutorial. If you have any questions about it functionality please review that tutorial.

```
class MoneyAgent(mesa.Agent):
    """An agent with fixed initial wealth."""

    def __init__(self, model):
        # Pass the parameters to the parent class.
        super().__init__(model)

        # Create the agent's variable and set the initial values.
        self.wealth = 1

    def exchange(self):
        # Verify agent has some wealth
        if self.wealth > 0:
            other_agent = self.random.choice(self.model.agents)
            if other_agent is not None:
                other_agent.wealth += 1
                self.wealth -= 1


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n, rng=None):
        super().__init__(rng=rng)
        self.num_agents = n

        # Create agents
        MoneyAgent.create_agents(model=self, n=n)

    def step(self):
        """Advance the model by one step."""
        # This function psuedo-randomly reorders the list of agent objects and
        # then iterates through calling the function passed in as the parameter
        self.agents.shuffle_do("exchange")
```

```
# Execute the model
model = MoneyModel(10)
model.step()
# Make sure it worked
print(f"You have {len(model.agents)} agents.")
```

```
You have 10 agents.
```

#### 2.4.2.6.5.6 Adding space

**Background:** Due to the complex dynamics of space and movement, Mesa offers a wide range space options and has built a structure to allow for the addition of even more spaces or custom user space creation. (Please contribute to Mesa if you develop a new space that can add to user options.)

The two main approaches to space are discrete space (think cells or nodes that agents occupy) and continuous space (agents can occupy any location(s) in a three-dimensional space). Continuous space is still experimental as we continue to develop it.

**Overview of Discrete Space:** For this tutorial we will be using discrete space in the classic cartesian coordinated system. As indicated in the diagram discrete space is made up of two modules. Cells and Cell Agents.

**Cells:** The cell class represents a location that can:

- Have properties (like temperature or resources)
- Track and limit the agents it contains
- Connect to neighboring cells
- Provide neighborhood information

Cells form the foundation of the cell space system, enabling rich spatial environments where both location properties and agent behaviors matter. They're useful for modeling things like varying terrain, infrastructure capacity, or environmental conditions.

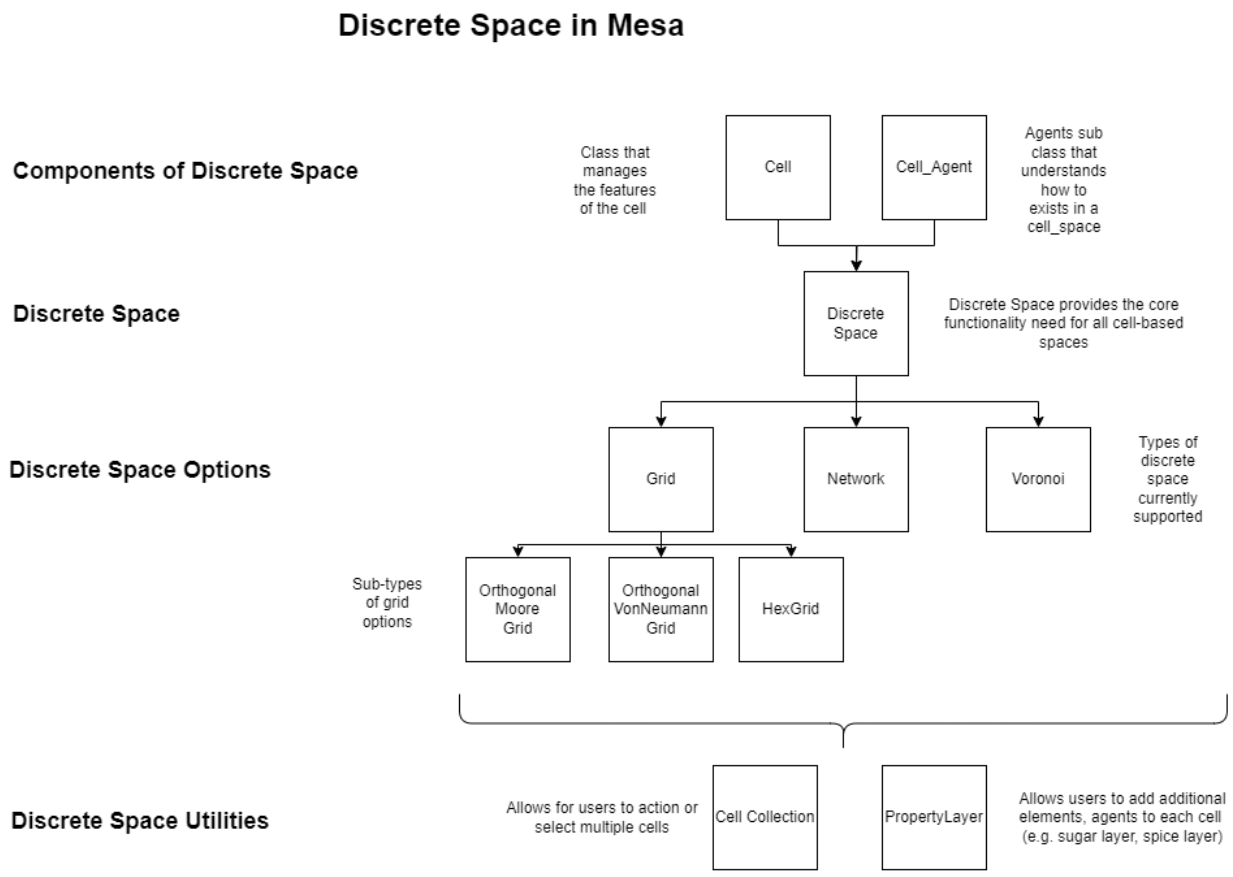
**Cell Agents:** Agents that understand how to exist in and move through cell spaces.

Cell Agents are specialized agent classes that handle cell occupation, movement, and proper registration:

- `CellAgent`: Mobile agents that can move between cells
- `FixedAgent`: Immobile agents permanently fixed to cells
- `Grid2DMovingAgent`: Agents with grid-specific movement capabilities

These classes ensure consistent agent-cell relationships and proper state management as agents move through the space. They can be used directly or as examples for creating custom cell-aware agents.

From these basic building blocks we can then add features to allow for different types of spaces and behaviors. To keep this tutorial concise we will not go through all of them, however, the current layout of discrete space is below as well as the different support modules. To find out more about the other options and what they can do, check out the Discrete Space API




*A big thanks to maintainer qualquel and his creation of this exceptional space dynamic.*

**Model-specific information:** In addition to using discrete space, the agents will access their Moore neighborhood. A Moore neighborhood means agents can interact with 8 neighbors. Instead of giving their unit of money to any random agent, they'll give it to an agent on the same cell. For the Money model multiple agents can be in the same spaces and since they are on a torus the agents on the left side can exchange money with agent on the right. Agents on the top can exchange with agents on the bottom.

#### 2.4.2.6.5.7 Code Implementation

To ensure we give our agents discrete space functionality we now instantiate our `MoneyAgents` as `CellAgents`. `Cell Agent` is a subclass to Mesa's `Agent` class that is specifically built to interact and move within the `discrete space` module.

Below highlights each of the changes to the base code to add space and movement of agents.

**Imports** # Import Cell Agent and OrthogonalMooreGrid

- *Description:* Import the cell agent class and a specific grid construct the OrthognalMooreGrid.
- *API*: CellAgent and OrthogonalMooreGrid

**MoneyAgent Class** # Instantiate MoneyAgent as CellAgent

- *Description:* `MoneyAgent` inherits `CellAgent`, a subclass of `Agent`.
- *API*: CellAgent

# Instantiate agent with location (x,y)

- *Description:* Pass the cell object as a parameter to the agent to give the agent a location

- *API*: N/A

# Move function

- *Description:* Update the agents cell through methods in Mesa's discrete_space module `neighborhood`, which defaults to radius one and `select_random_cell` which selects a random cell for the provided neighborhood
- *API*: neighborhood and select_random_cell

**MoneyModel Class** # Instantiate an instance of Moore neighborhood space

- *Description:* Instantiate a OrthgonalMooreGrid as `self.grid` with passing in the parameters width and height as a tuple, torus as True, and the models random seed to the discrete space
- *API*: OrthogonalMooreGrid

# Randomly select agents cell

- *Description:* Use Python's `random.choices` and pass in all cells with discrete space all_cells properties and the number of choices `k` to assign each agent a location.
- *API*: random.choices and all_cells

```
# Import Cell Agent and OrthogonalMooreGrid
from mesa.discrete_space import CellAgent, OrthogonalMooreGrid


# Instantiate MoneyAgent as CellAgent
class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        super().__init__(model)
        self.cell = cell  # Instantiate agent with location (x,y)
        self.wealth = 1

    # Move Function
    def move(self):
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        cellmates = [
            a for a in self.cell.agents if a is not self
        ]  # Get all agents in cell

        if self.wealth > 0 and cellmates:
            other_agent = self.random.choice(cellmates)
            other_agent.wealth += 1
            self.wealth -= 1


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n, width, height, rng=None):
        super().__init__(rng=rng)
        self.num_agents = n
        # Instantiate an instance of Moore neighborhood space
```

(continues on next page)

(continued from previous page)

```
        self.grid = OrthogonalMooreGrid((width, height), torus=True, random=self.random)

        # Create agents
        agents = MoneyAgent.create_agents(
            self,
            self.num_agents,
            # Randomly select agents cell
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

    def step(self):
        self.agents.shuffle_do("move")
        self.agents.do("give_money")
```

Let's create a model with 100 agents on a 10x10 grid, and run it for 20 steps.

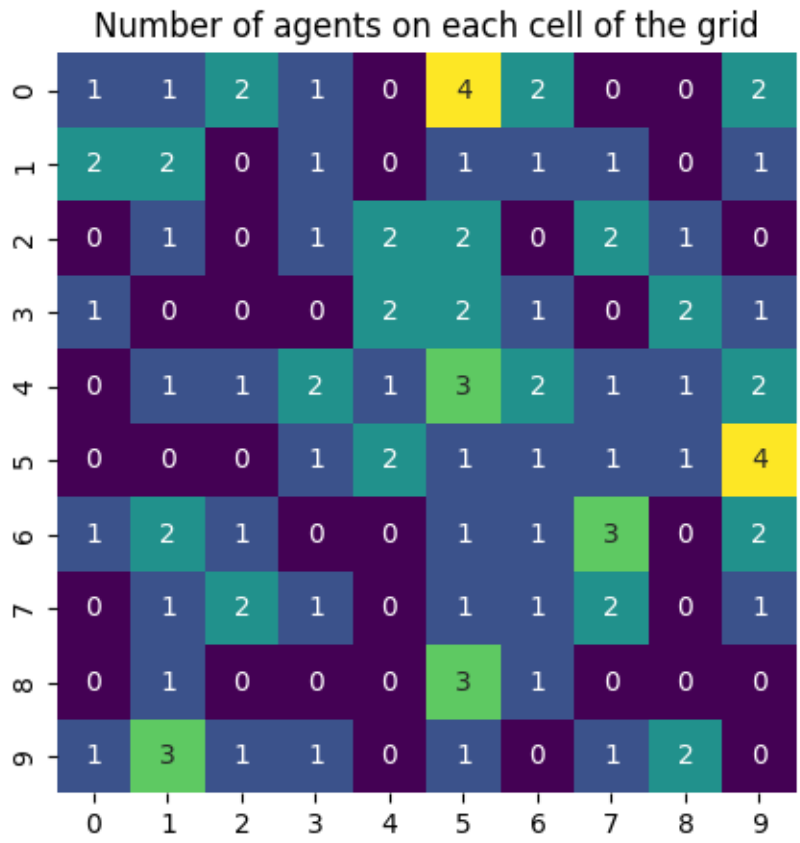
```
model = MoneyModel(100, 10, 10)
for _ in range(20):
    model.step()
```

Now let's use seaborn and numpy to visualize the number of agents residing in each cell. To do that, we create a numpy array of the same size as the grid, filled with zeros.

Then again use `all_cells` to loop over every cell in the grid, giving us each cell's position (cell coordinate attribute) and its contents (cell agent attribute).

Cell API

```
agent_counts = np.zeros((model.grid.width, model.grid.height))

for cell in model.grid.all_cells:
    agent_counts[cell.coordinate] = len(cell.agents)
# Plot using seaborn, with a visual size of 5x5
g = sns.heatmap(agent_counts, cmap="viridis", annot=True, cbar=False, square=True)
g.figure.set_size_inches(5, 5)
g.set(title="Number of agents on each cell of the grid");
```




#### 2.4.2.6.5.8 Exercises

- Change the size of the grid
- Change the capacity of the cells and account for this in Agent movement
- Try a different grid space like OrthognalVonNeumann, Network, or Voronoi

#### 2.4.2.6.5.9 Next Steps

Check out the *collecting data tutorial* on how to collect data from your model.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

### 2.4.2.6.6 Collecting Data

#### 2.4.2.6.6.1 The Boltzmann Wealth Model

If you want to get straight to the tutorial checkout these environment providers: (with Google Account) (No Google Account) (This can take 30 seconds to 5 minutes to load)

*If you are running locally, please ensure you have the latest Mesa version installed.*

#### 2.4.2.6.6.2 Tutorial Description

This tutorial extends the Boltzmann wealth model from the *Adding Space tutorial*, by adding Mesa's data collection module.

In this portion, we will collect both model level data and agent level data to better understand the dynamics of our model.

*If you are starting here please see the* Running Your First Model tutorial *for dependency and start-up instructions*

#### 2.4.2.6.6.3 IN COLAB? - Run the next cell

#### 2.4.2.6.6.4 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns

import mesa

# Import Cell Agent and OrthogonalMooreGrid
from mesa.discrete_space import CellAgent, OrthogonalMooreGrid
```

#### 2.4.2.6.6.5 Base Model

The below provides the base model from which we will add our space functionality.

This is from the *Adding Space tutorial* tutorial. If you have any questions about it functionality please review that tutorial.

```
class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        super().__init__(model)
        self.cell = cell  # Instantiate agent with location (x,y)
        self.wealth = 1

    # Move Function
    def move(self):
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        cellmates = [
            a for a in self.cell.agents if a is not self
        ]  # Get all agents in cell
```

(continues on next page)

(continued from previous page)

```
        if self.wealth > 0 and cellmates:
            other_agent = self.random.choice(cellmates)
            other_agent.wealth += 1
            self.wealth -= 1


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n, width, height, rng=None):
        super().__init__(rng=rng)
        self.num_agents = n
        # Instantiate an instance of Moore neighborhood space
        self.grid = OrthogonalMooreGrid((width, height), torus=True, random=self.random)

        # Create agents
        agents = MoneyAgent.create_agents(
            self,
            self.num_agents,
            # Randomly select agents cell
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

    def step(self):
        self.agents.shuffle_do("move")
        self.agents.do("give_money")
```

Let's create a model with 100 agents on a 10x10 grid, and run it for 20 steps to make sure our base model works.

```
model = MoneyModel(100, 10, 10)
model.run_for(20)
# Let's make sure it worked
print(len(model.agents))
```

```
100
```

#### 2.4.2.6.6.6 Collecting Data

**Background:** So far, at the end of every model run, we've had to go and write our own code to get the data out of the model. This has two problems: it isn't very efficient, and it only gives us end results. If we wanted to know the wealth of each agent at each step, we'd have to add that to the loop of executing steps, and figure out some way to store the data.

Since one of the main goals of agent-based modeling is generating data for analysis, Mesa provides a class which can handle data collection and storage for us and make it easier to analyze.

The data collector stores three categories of data:

- Model-level variables : Model-level collection functions take a model object as an input. Such as a function that computes a dynamic of the whole model (in this case we will compute a measure of wealth inequality based on all agent's wealth)
- Agent-level variables: Agent-level collection functions take an agent object as an input and is typically the state of an agent attributes, in this case wealth.

- Tables (which are a catch-all for everything else).

**Model-specific information:** We will collect two variables to show Mesa capabilities.

- At the model level, let's measure the model's Gini Coefficient, a measure of wealth inequality.
- At the agent level, we want to collect every agent's wealth at every step.

**Code implementation:**

Let's add a DataCollector to the model with `mesa.DataCollector`, and collect the agent's wealth and the gini coefficient at each time step. In the below code each new line of code is described with a comment. These additions are described below.

**Helper Function** # Add function for model level collection -*Description:* Helper function used by the model class to compute the gini coefficient as described previously. -*API:* N/A

**MoneyModel Class** # Instantiate DataCollector

- *Description:* Create a mesa data collector instance and use keyword arguments (kwargs) `model_reporters` and `agent_reporters` to pass in a dictionary, where the key is the name of the data collected and the value is either function (i.e. computer gini) or an attribute (i.e. "wealth"). If it is an attribute it is passed in as a string.
- *API:* Data Collection

# Collect data each step

- *Description:* Call the `collect` method from `DataCollector`. This causes the reporters to collect the data at each step. If this is not put in the step function then the data collector will collect the described information at the end of the model run. If you want to collect the data only on lets say the 5th step, then you can just add an `if` statement to only collect on the fifth step.
- *API:* DataCollector.collect

```
# Add function for model level collection
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.agents]
    x = sorted(agent_wealths)
    n = model.num_agents
    B = sum(xi * (n - i) for i, xi in enumerate(x)) / (n * sum(x))
    return 1 + (1 / n) - 2 * B


class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        super().__init__(model)
        self.cell = cell
        self.wealth = 1

    def move(self):
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        cellmates = [a for a in self.cell.agents if a is not self]

        if self.wealth > 0 and cellmates:
            other_agent = self.random.choice(cellmates)
```

(continues on next page)

(continued from previous page)

```
            other_agent.wealth += 1
            self.wealth -= 1


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n, width, height, rng=None):
        super().__init__(rng=rng)
        self.num_agents = n
        self.grid = OrthogonalMooreGrid((width, height), torus=True, random=self.random)
        # Instantiate DataCollector
        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini}, agent_reporters={"Wealth": "wealth"}
        )

        # Create agents
        agents = MoneyAgent.create_agents(
            self,
            self.num_agents,
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

    def step(self):
        # Collect data each step
        self.datacollector.collect(self)
        self.agents.shuffle_do("move")
        self.agents.do("give_money")
```

At every step of the model, the datacollector will collect and store the model-level current Gini coefficient, as well as each agent's wealth, associating each with the current step.

We run the model just as we did above. Now is when an interactive session, especially via a notebook, comes in handy: the DataCollector can export the data it has collected as a pandas* DataFrame, for easy and interactive analysis.

*If you are new to Python, please be aware that pandas is already installed as a dependency of Mesa and that pandas is a "fast, powerful, flexible and easy to use open source data analysis and manipulation tool". Pandas is a great resource to help analyze the data collected in your models.

```
model = MoneyModel(100, 10, 10)
model.run_for(100)
```

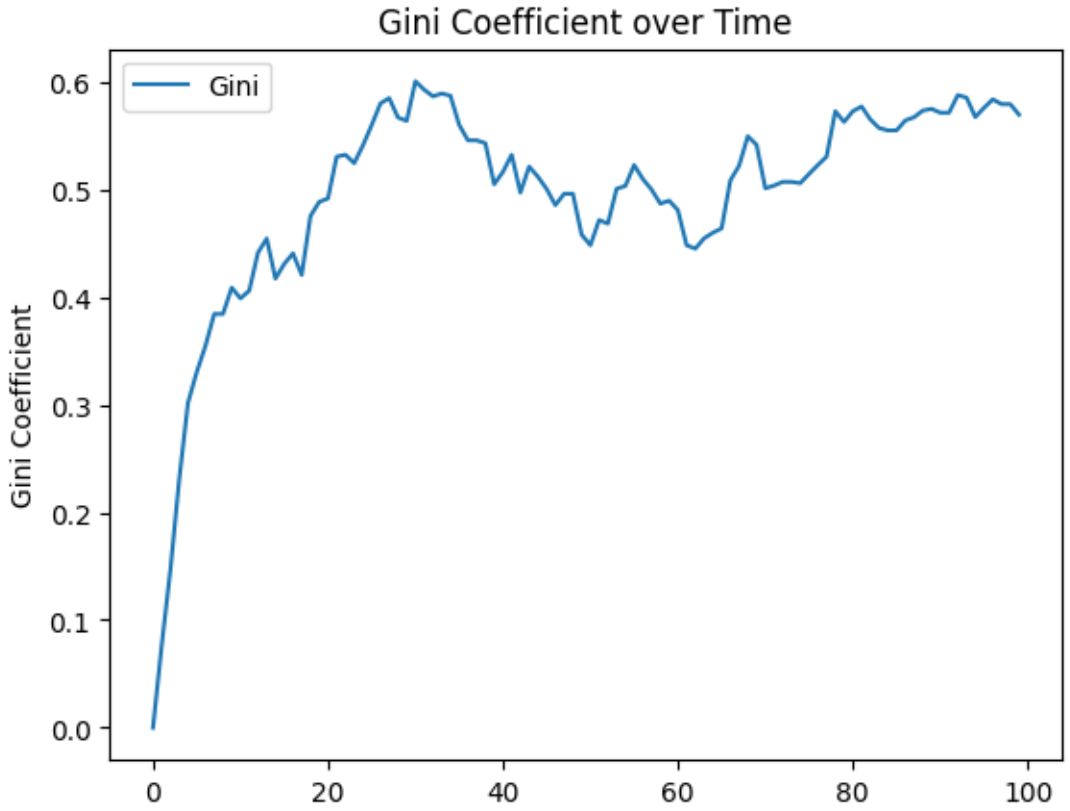
#### 2.4.2.6.6.7 Analyzing MoneyModel Data

**Code implementation:**

# Extract MoneyModel data in a Pandas dataframe

- *Description:* Call `DataCollector.get_model_vars_dataframe()` method to get the model reporters (in this case gini coefficient) from the model object. We the use seaborn (sns) to do a line plot of the data of the model run.
- *API:* get_model_vars_dataframe

```
# Extract MoneyModel data in a Pandas dataframe
gini = model.datacollector.get_model_vars_dataframe()
g = sns.lineplot(data=gini)
g.set(title="Gini Coefficient over Time", ylabel="Gini Coefficient");
```




#### 2.4.2.6.6.8 Exercises

- Display just the data to see the format
- Comment on the collect method on the step function and see the impact
- Increase agents and time to see how the plot changes

#### 2.4.2.6.6.9 Analyzing an MoneyAgent Data

**Code implementation:**

# Extract MoneyAgent data in a Pandas dataframe

- *Description:* Call `DataCollector.get_model_agent_dataframe()` method to get the agent reporters (in this case agent wealth attribute) from the model object.
- *API:* get_model_agent_dataframe

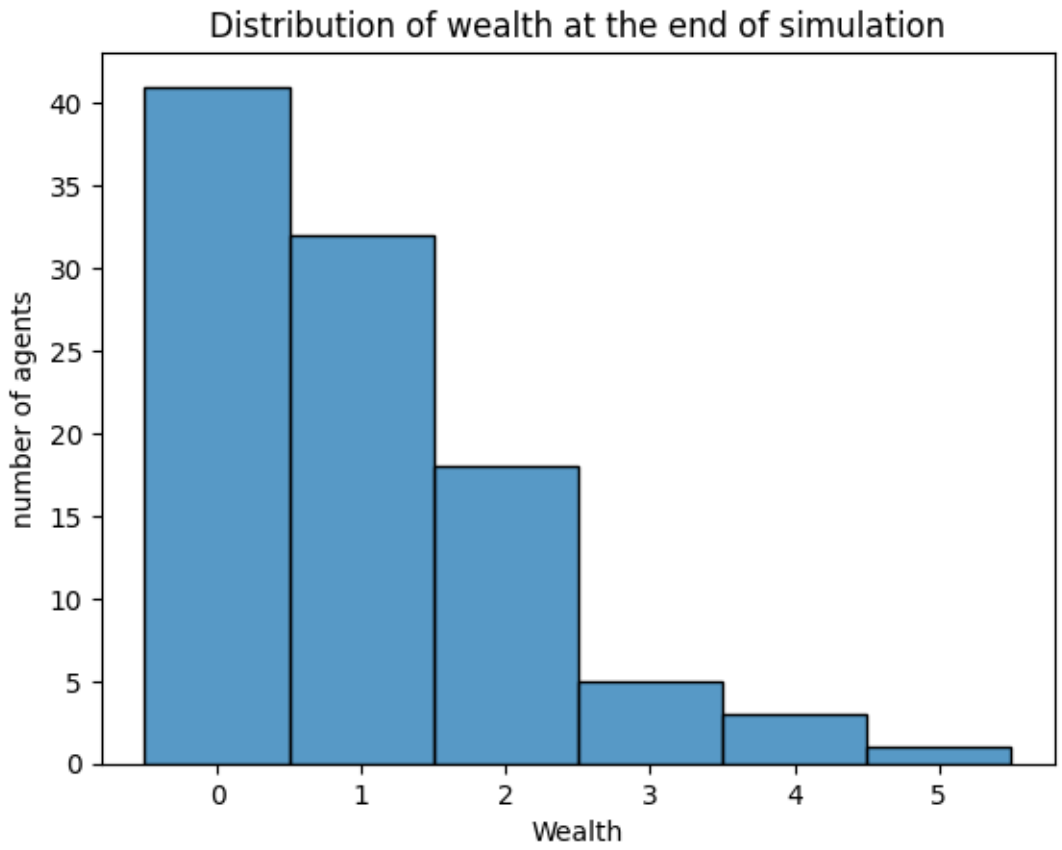
```
# Extract MoneyAgent data in a Pandas dataframe
agent_wealth = model.datacollector.get_agent_vars_dataframe()
agent_wealth.head()
```

```
              Wealth
Step AgentID
1.0  1              1
     2              1
     3              1
     4              1
     5              1
```

You'll see that the DataFrame's index is pairings of model step and agent ID. This is because the data collector stores the data in a dictionary, with the step number as the key, and a dictionary of agent ID and variable value pairs as the value. The data collector then converts this dictionary into a DataFrame, which is why the index is a pair of (model step, agent ID). You can analyze it the way you would any other DataFrame. For example, to get a histogram of agent wealth at the model's end.

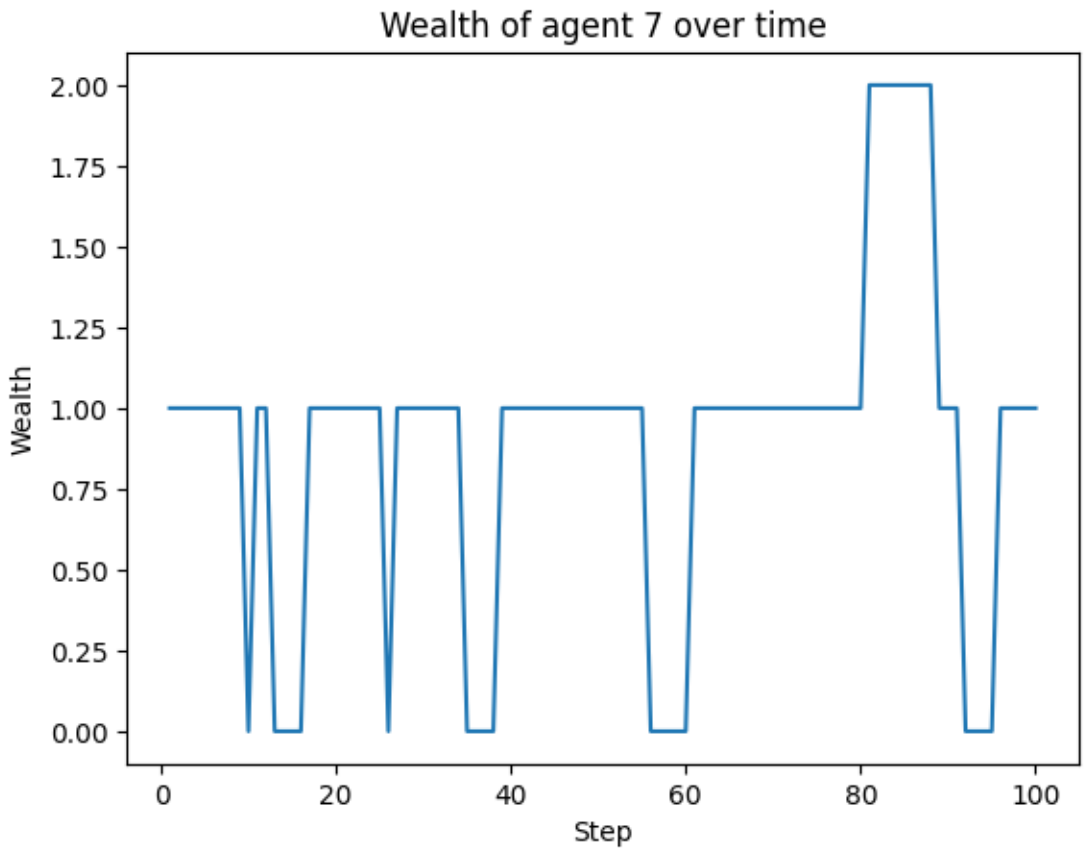
*Note: As the following code is pandas and seaborn we do not provide explanatory text*

```
last_step = agent_wealth.index.get_level_values("Step").max()  # Get the last step
end_wealth = agent_wealth.xs(last_step, level="Step")[
    "Wealth"
]  # Get the wealth of each agent at the last step
# Create a histogram of wealth at the last step
g = sns.histplot(end_wealth, discrete=True)
g.set(
    title="Distribution of wealth at the end of simulation",
    xlabel="Wealth",
    ylabel="number of agents",
);
```



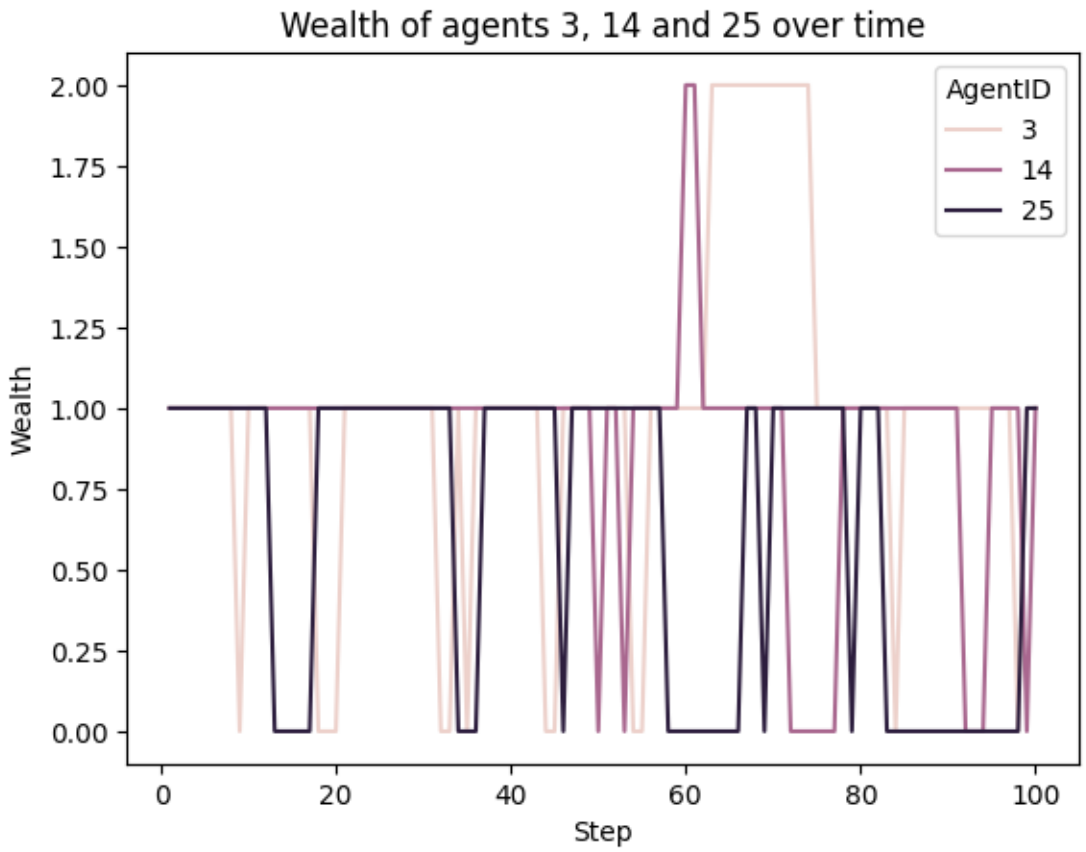

Or to plot the wealth of a given agent (in this example, agent 7):

```
# Get the wealth of agent 7 over time
one_agent_wealth = agent_wealth.xs(7, level="AgentID")

# Plot the wealth of agent 7 over time
g = sns.lineplot(data=one_agent_wealth, x="Step", y="Wealth")
g.set(title="Wealth of agent 7 over time");
```




You can also plot a reporter of multiple agents over time.

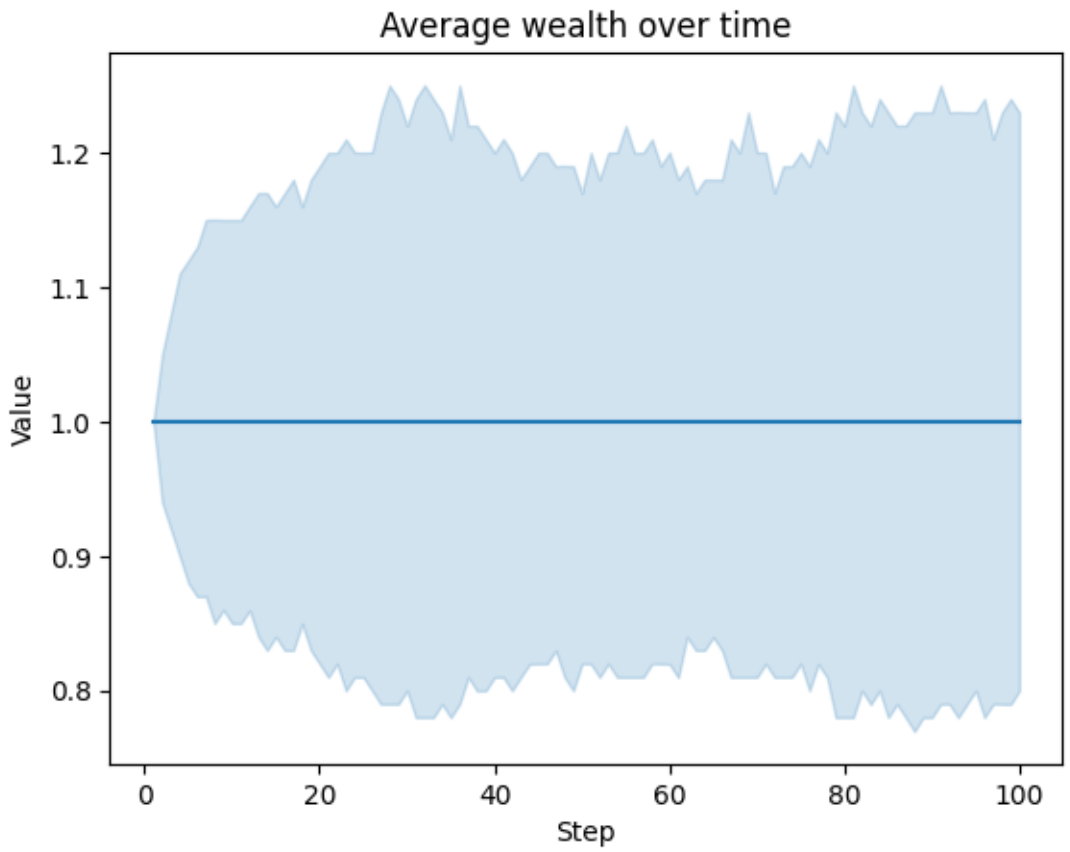
```
agent_list = [3, 14, 25]

# Get the wealth of multiple agents over time
multiple_agents_wealth = agent_wealth[
    agent_wealth.index.get_level_values("AgentID").isin(agent_list)
]
# Plot the wealth of multiple agents over time
g = sns.lineplot(data=multiple_agents_wealth, x="Step", y="Wealth", hue="AgentID")
g.set(title="Wealth of agents 3, 14 and 25 over time");
```




We can also plot the average of all agents, with a 95% confidence interval for that average.

```
# Transform the data to a long format
agent_wealth_long = agent_wealth.T.unstack().reset_index()
agent_wealth_long.columns = ["Step", "AgentID", "Variable", "Value"]
agent_wealth_long.head(3)

# Plot the average wealth over time
g = sns.lineplot(data=agent_wealth_long, x="Step", y="Value", errorbar=("ci", 95))
g.set(title="Average wealth over time")
```

```
[Text(0.5, 1.0, 'Average wealth over time')]
```




Which is exactly 1, as expected in this model, since each agent starts with one wealth unit, and each agent gives one wealth unit to another agent at each step.

You can also use pandas to export the data to a CSV (comma separated value) file, which can be opened by any common spreadsheet application or opened by pandas.

If you do not specify a file path, the file will be saved in the local directory. After you run the code below you will see two files appear (*model_data.csv* and *agent_data.csv*)

```
# save the model data (stored in the pandas gini object) to CSV
gini.to_csv("model_data.csv")

# save the agent data (stored in the pandas agent_wealth object) to CSV
agent_wealth.to_csv("agent_data.csv")
```

```
# Challenge update the model, conduct a parameter sweep,
# and visualize your results
```

#### 2.4.2.6.6.10 Next Steps

Check out the *Basic Visualization tutorial* on how to build interactive dashboards for displaying your models.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and

Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

#### 2.4.2.6.7 Visualization - Basic Dashboard

##### 2.4.2.6.7.1 The Boltzmann Wealth Model

If you want to get straight to the tutorial checkout these environment providers: (This can take 30 seconds to 5 minutes to load)

Due to conflict with Colab and Solara there are no colab links for this tutorial

*If you are running locally, please ensure you have the latest Mesa version installed.*

##### 2.4.2.6.7.2 Tutorial Description

This tutorial extends the Boltzmann wealth model from the *AgentSet tutorial*, by adding an interactive dashboard.

In this portion, we will demonstrate how users can employ build a basic dashboard. This is part one of three part series on building interactive dashboards in Mesa.

*If you are starting here please see the* Running Your First Model tutorial *for dependency and start-up instructions*

##### 2.4.2.6.7.3 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns

import mesa
from mesa.discrete_space import CellAgent, OrthogonalMooreGrid

# Check Mesa version for visualization compatibility
if mesa.__version__.startswith(("3.0", "3.1", "3.2")):
    print(
        f"  Mesa {mesa.__version__} detected. Visualization features require Mesa 3.3+"
    )
    print("To upgrade: pip install --upgrade mesa")

from mesa.visualization import SolaraViz, SpaceRenderer, make_plot_component
from mesa.visualization.components import AgentPortrayalStyle
```

##### 2.4.2.6.7.4 Basic Model

The following is the basic model we will be using to build the dashboard. This is the same model seen in tutorials 0-3.

```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.agents]
```

(continues on next page)

(continued from previous page)

```
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B


class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        """initialize a MoneyAgent instance.

        Args:
            model: A model instance
        """
        super().__init__(model)
        self.cell = cell
        self.wealth = 1

    def move(self):
        """Move the agent to a random neighboring cell."""
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        """Give 1 unit of wealth to a random agent in the same cell."""
        cellmates = [a for a in self.cell.agents if a is not self]

        if cellmates:  # Only give money if there are other agents present
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

    def step(self):
        """do one step of the agent."""
        self.move()
        if self.wealth > 0:
            self.give_money()


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10, width=10, height=10, rng=None):
        """Initialize a MoneyModel instance.

        Args:
            N: The number of agents.
            width: width of the grid.
            height: Height of the grid.
        """
        super().__init__(rng=rng)
        self.num_agents = n
```

(continues on next page)

(continued from previous page)

```
        self.grid = OrthogonalMooreGrid((width, height), random=self.random)

        # Create agents
        MoneyAgent.create_agents(
            self,
            self.num_agents,
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini}, agent_reporters={"Wealth": "wealth"}
        )
        self.datacollector.collect(self)

    def step(self):
        """do one step of the model"""
        self.agents.shuffle_do("step")
        self.datacollector.collect(self)
```

#### 2.4.2.6.7.5 Important note for SolaraViz users

When using **SolaraViz**, Mesa models must be instantiated **using keyword arguments only**. SolaraViz creates model instances internally via keyword-based parameters, and positional arguments are **not supported**.

**Not supported:**

```
MyModel(10, 10)
```

**Supported:**

```
MyModel(width=10, height=10)
```

**Common Pitfall:**

When converting from positional to keyword arguments, make sure to include ALL parameters. For example:

```
model=MoneyModel(100,10,10) #n=100, width=10, height=10

# Correct conversion (keyword)
model = MoneyModel(n=100, width=10, height=10)  # All parameters included

# Incorrect conversion (missing n)
model = MoneyModel(width=10, height=10)  # Defaults to n=10
```

To avoid errors, it is recommended to define your model constructor with keyword-only arguments, for example:

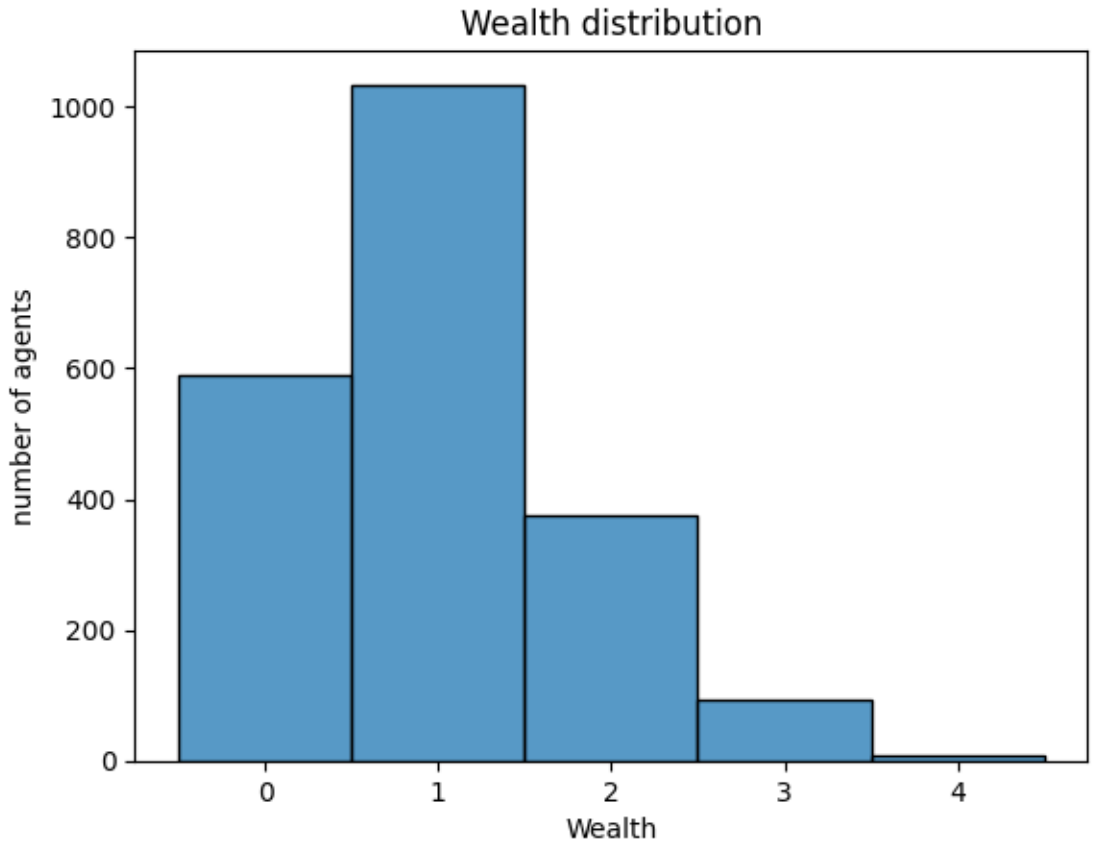
```
class MyModel(Model):
    def __init__(self, *, width, height, rng=None):
        ...
```

```
# Lets make sure the model works
model = MoneyModel(n=100, width=100, height=10, rng=10)
model.run_for(20)
```

(continues on next page)

(continued from previous page)

```
data = model.datacollector.get_agent_vars_dataframe()
# Use seaborn
g = sns.histplot(data["Wealth"], discrete=True)
g.set(title="Wealth distribution", xlabel="Wealth", ylabel="number of agents");
```




#### 2.4.2.6.7.6 Adding visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components, and (for advanced users) how to create a new visualization element.

First, a quick explanation of how Mesa's interactive visualization works. The visualization is done in a browser window or Jupyter instance, using the Solara framework, a pure Python, React-style web framework. Running `solara run app.py` will launch a web server, which runs the model, and displays model detail at each step via a plotting library. Alternatively, you can execute everything inside a Jupyter instance and display it inline.

#### 2.4.2.6.7.7 Grid Visualization

Mesa's grid visualizer works by iterating over each cell in the grid and generating a portrayal for every agent it finds. The portrayal function is called for each agent and returns an `AgentPortrayalStyle`—an object that defines how the agent is visually represented.

All you need to provide is a function that takes an agent as input and returns an `AgentPortrayalStyle`.

Here's the simplest example: it draws each agent as a filled orange circle with a radius of 50.

```
def agent_portrayal(agent):
    return AgentPortrayalStyle(color="tab:orange", size=50)
```

In addition to the portrayal method, we instantiate the model parameters, some of which are modifiable by user inputs. In this case, the number of agents, N, is specified as a slider of integers.

```
model_params = {
    "n": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
    "width": 10,
    "height": 10,
}
```

Next, we instantiate the visualization object which (by default) displays the grid containing the agents, and timeseries of values computed by the model's data collector. In this example, we specify the Gini coefficient.

There are 3 main buttons (we will discuss the play interval, render interval and use threads in lesson 6):

- the step button, which advances the model by 1 step
- the play button, which advances the model indefinitely until it is paused
- the pause button, which pauses the model

To reset the model, the order of operations are important

1. Stop the model
2. Update the parameters (e.g. move the sliders)
3. Press reset

#### 2.4.2.6.7.8 SpaceRenderer

This is the Python object used to visualize the grid, agents, and property layers associated with the space and the model and is passed to the `SolaraViz`

We initialize the SpaceRenderer with a model instance (e.g., `money_model` in this case) and specify the rendering backend. The available backends are `matplotlib`(default) and `altair`.

Both backends can be extended using the post_process function, which can either be passed directly to the render() method or set as a property of the renderer. (We'll cover this in more detail later.)

The method shown here is the quickest way to set up the visualization using the `render()` function.

It renders the space, agents, and property layers—provided that the corresponding portrayal functions are supplied for both agents and property layers.

**Note:**

You can make the small window full screen by clicking the button in the top-right corner of the title bar.

#### 2.4.2.6.7.9 Page Tab View

#### 2.4.2.6.7.10 Plot Components

You can place different components (except the renderer) on separate pages according to your preference. There are no restrictions on page numbering — pages do not need to be sequential or positive. Each page acts as an independent window where components may or may not exist.

The default page is `page=0`. If pages are not sequential (e.g., `page=1` and `page=10`), the system will automatically create the 8 empty pages in between to maintain consistent indexing. To avoid empty pages in your dashboard, use sequential page numbers.

To assign a plot component to a specific page, pass the `page` keyword argument to `make_plot_component`. For example, the following will display the plot component on page 1:

```
plot_comp = make_plot_component("encoding", page=1)
```

#### 2.4.2.6.7.11 Custom Components

In tutorial 8, you will learn how to create custom components for the Solara dashboard. If you want a custom component to appear on a specific page, you must pass it as a tuple containing the component and the page number.

```
@solara.component
def CustomComponent():
    ...

page = SolaraViz(
    model,
    renderer,
    components=[(CustomComponent, 1)]  # Custom component will appear on page 1
)
```

**Warning** Running the model can be performance-intensive. It is strongly recommended to pause the model in the dashboard before switching pages.

```
# Create initial model instance
money_model = MoneyModel(n=50, width=10, height=10)  # keyword arguments

renderer = (
    SpaceRenderer(model=money_model, backend="matplotlib")
    .setup_agents(agent_portrayal)
    .render()
)

GiniPlot = make_plot_component("Gini", page=1)

page = SolaraViz(
    money_model,
```

(continues on next page)

(continued from previous page)

```
    renderer,
    components=[GiniPlot],
    model_params=model_params,
    name="Boltzmann Wealth Model",
)
# This is required to render the visualization in the Jupyter notebook
page
```

```
Cannot show ipywidgets in text
```

#### 2.4.2.6.7.12 Next Steps

Check out the next *visualization tutorial dynamic agents* on how to further enhance your interactive dashboard.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

### 2.4.2.6.8 Visualization - Dynamic Agents

#### 2.4.2.6.8.1 The Boltzmann Wealth Model

If you want to get straight to the tutorial checkout these environment providers: (This can take 30 seconds to 5 minutes to load)

Due to conflict with Colab and Solara there are no colab links for this tutorial

*If you are running locally, please ensure you have the latest Mesa version installed.*

#### 2.4.2.6.8.2 Tutorial Description

This tutorial extends the Boltzmann wealth model from the *Visualization Basic Dashboard tutorial*, by adding an interactive dashboard.

In this portion, we will demonstrate how users can employ create dynamic agent representation with their Mesa dashboards. This is part two of three visualization tutorials.

*If you are starting here please see the* Running Your First Model tutorial *for dependency and start-up instructions*

#### 2.4.2.6.8.3 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns
```

(continues on next page)

(continued from previous page)

```
import mesa
from mesa.discrete_space import CellAgent, OrthogonalMooreGrid

# Check Mesa version for visualization compatibility
if mesa.__version__.startswith(("3.0", "3.1", "3.2")):
    print(
        f"  Mesa {mesa.__version__} detected. Visualization features require Mesa 3.3+"
    )
    print("To upgrade: pip install --upgrade mesa")

from mesa.visualization import SolaraViz, SpaceRenderer, make_plot_component
from mesa.visualization.components import AgentPortrayalStyle
```

#### 2.4.2.6.8.4 Basic Model

The following is the basic model we will be using to build the dashboard. This is the same model seen in tutorials 0-3.

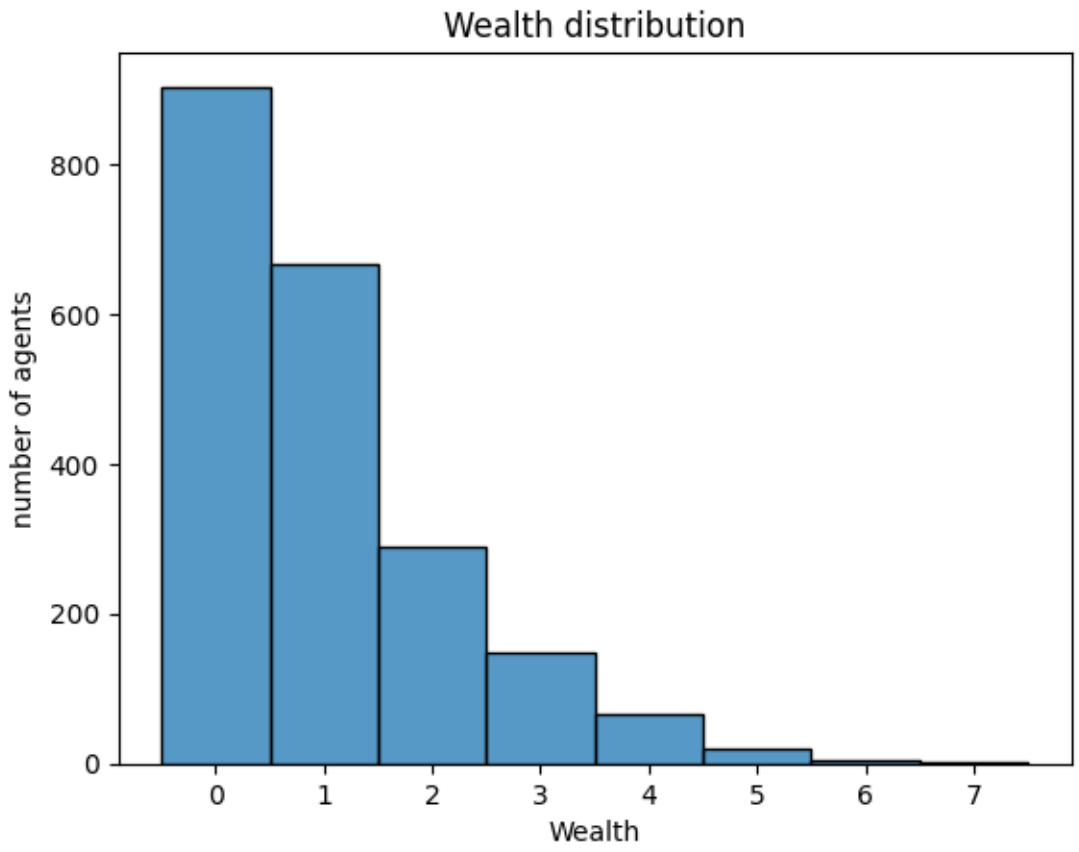
```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B


class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        """initialize a MoneyAgent instance.

        Args:
            model: A model instance
        """
        super().__init__(model)
        self.cell = cell
        self.wealth = 1

    def move(self):
        """Move the agent to a random neighboring cell."""
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        """Give 1 unit of wealth to a random agent in the same cell."""
        cellmates = [a for a in self.cell.agents if a is not self]

        if cellmates:  # Only give money if there are other agents present
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1
```

(continues on next page)

(continued from previous page)

```
    def step(self):
        """do one step of the agent."""
        self.move()
        if self.wealth > 0:
            self.give_money()


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10, width=10, height=10, rng=None):
        """Initialize a MoneyModel instance.

        Args:
            N: The number of agents.
            width: width of the grid.
            height: Height of the grid.
        """
        super().__init__(rng=rng)
        self.num_agents = n
        self.grid = OrthogonalMooreGrid((width, height), random=self.random)

        # Create agents
        MoneyAgent.create_agents(
            self,
            self.num_agents,
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini}, agent_reporters={"Wealth": "wealth"}
        )
        self.datacollector.collect(self)

    def step(self):
        """do one step of the model"""
        self.agents.shuffle_do("step")
        self.datacollector.collect(self)
```

```
# Lets make sure the model works
model = MoneyModel(100, 10, 10)
model.run_for(20)


data = model.datacollector.get_agent_vars_dataframe()
# Use seaborn
g = sns.histplot(data["Wealth"], discrete=True)
g.set(title="Wealth distribution", xlabel="Wealth", ylabel="number of agents");
```




#### 2.4.2.6.8.5 Adding visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components, and (for advanced users) how to create a new visualization element.

First, a quick explanation of how Mesa's interactive visualization works. The visualization is done in a browser window or Jupyter instance, using the Solara framework, a pure Python, React-style web framework. Running `solara run app.py` will launch a web server, which runs the model, and displays model detail at each step via a plotting library. Alternatively, you can execute everything inside a Jupyter instance and display it inline.

#### 2.4.2.6.8.6 Dynamic Agent Representation

In the first visualization, all we could see is the agents moving around – but not how much money they had, or anything else of interest. In this tutorial let's change it so that agents are represented by the units of wealth they have. So those who are broke (wealth 0) are drawn in red, smaller.

Since Mesa is open source, if you have ideas to improve the visualization stack, feel free to contribute.

When using the default drawer, an agent's shape can be customized in addition to its size and color.

- For `matplotlib`, allowed shape values can be found here.
- For `altair`, supported shapes include: `"circle"`, `"square"`, `"cross"`, `"diamond"`, `"triangle-up"`, `"triangle-down"`, `"triangle-right"`, and `"triangle-left"`.

**Note:** Always check which backend is being used before assigning shapes. If a shape name doesn't match what's supported by the selected backend, it may cause errors.

In some cases, Mesa implicitly converts common shape names between Altair and Matplotlib (e.g., from Altair to Matplotlib), but this mapping is limited—and there's no support for conversion in the reverse direction (Matplotlib to Altair).

To do this, we go back to our `agent_portrayal` code and add some code to change the portrayal based on the agent properties and launch the server again.

```
def agent_portrayal(agent):
    portrayal = AgentPortrayalStyle(size=50, color="tab:orange")
    if agent.wealth > 0:
        portrayal.update(("color", "tab:blue"), ("size", 100))
    return portrayal
```

As like last time we then instantiate the model parameters, some of which are modifiable by user inputs. In this case, the number of agents, N, is specified as a slider of integers.

```
model_params = {
    "n": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
    "width": 10,
    "height": 10,
}
```

Then just like last time we instantiate the visualization object which (by default) displays the grid containing the agents, and timeseries of values computed by the model's data collector. In this example, we specify the Gini coefficient.

There are 3 main buttons (we will discuss the play interval, render interval and use threads in lesson 6):

- the step button, which advances the model by 1 step
- the play button, which advances the model indefinitely until it is paused
- the pause button, which pauses the model

To reset the model, the order of operations are important

1. Stop the model
2. Update the parameters (e.g. move the sliders)
3. Press reset

#### 2.4.2.6.8.7 Page Tab View

#### 2.4.2.6.8.8 Plot Components

You can place different components (except the renderer) on separate pages according to your preference. There are no restrictions on page numbering — pages do not need to be sequential or positive. Each page acts as an independent window where components may or may not exist.

The default page is `page=0`. If pages are not sequential (e.g., `page=1` and `page=10`), the system will automatically create the 8 empty pages in between to maintain consistent indexing. To avoid empty pages in your dashboard, use sequential page numbers.

To assign a plot component to a specific page, pass the `page` keyword argument to `make_plot_component`. For example, the following will display the plot component on page 1:

```
plot_comp = make_plot_component("encoding", page=1)
```

#### 2.4.2.6.8.9 Custom Components

In tutorial 8, you will learn how to create custom components for the Solara dashboard. If you want a custom component to appear on a specific page, you must pass it as a tuple containing the component and the page number.

```
@solara.component
def CustomComponent():
    ...

page = SolaraViz(
    model,
    renderer,
    components=[(CustomComponent, 1)]  # Custom component will appear on page 1
)
```

**Warning** Running the model can be performance-intensive. It is strongly recommended to pause the model in the dashboard before switching pages.

```
# Create initial model instance
money_model = MoneyModel(n=50, width=10, height=10)
renderer = (
    SpaceRenderer(model=money_model, backend="matplotlib")
    .setup_agents(agent_portrayal)
    .render()
)

GiniPlot = make_plot_component("Gini")

page = SolaraViz(
    money_model,
    renderer,
    components=[GiniPlot],
    model_params=model_params,
    name="Boltzmann Wealth Model",
)
# This is required to render the visualization in the Jupyter notebook
page
```

```
Cannot show ipywidgets in text
```

#### 2.4.2.6.8.10 Exercise

- Change the agent representations, such as squares, triangles or even .pngs

#### 2.4.2.6.8.11 Next Steps

Check out the next tutorial *visualization rendering with* `SpaceRenderer` on how to further enhance your interactive dashboard.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

### 2.4.2.6.9 Visualization - Advanced Space Rendering

#### 2.4.2.6.9.1 The Boltzmann Wealth Model

If you want to get straight to the tutorial checkout these environment providers: (This can take 30 seconds to 5 minutes to load)

Due to conflict with Colab and Solara there are no colab links for this tutorial

*If you are running locally, please ensure you have the latest Mesa version installed.*

#### 2.4.2.6.9.2 Tutorial Description

This tutorial builds upon the *Dynamic Agent Representation* tutorial. We will explore more advanced features of the SpaceRenderer to create more informative and visually appealing spatial visualizations.

Specifically, we'll learn how to style the grid lines.

*If you are starting here please see the* Running Your First Model tutorial *for dependency and start-up instructions*

#### 2.4.2.6.9.3 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns

import mesa
from mesa.discrete_space import CellAgent, OrthogonalMooreGrid

# Check Mesa version for visualization compatibility
if mesa.__version__.startswith(("3.0", "3.1", "3.2")):
    print(
        f"  Mesa {mesa.__version__} detected. Visualization features require Mesa 3.3+"
```

(continues on next page)

(continued from previous page)

```
    )
    print("To upgrade: pip install --upgrade mesa")

from mesa.visualization import SolaraViz, SpaceRenderer, make_plot_component
from mesa.visualization.components import AgentPortrayalStyle
```

#### 2.4.2.6.9.4 Basic Model

The following is the basic model we will be using to build the dashboard. This is the same model seen in tutorials 0-3.

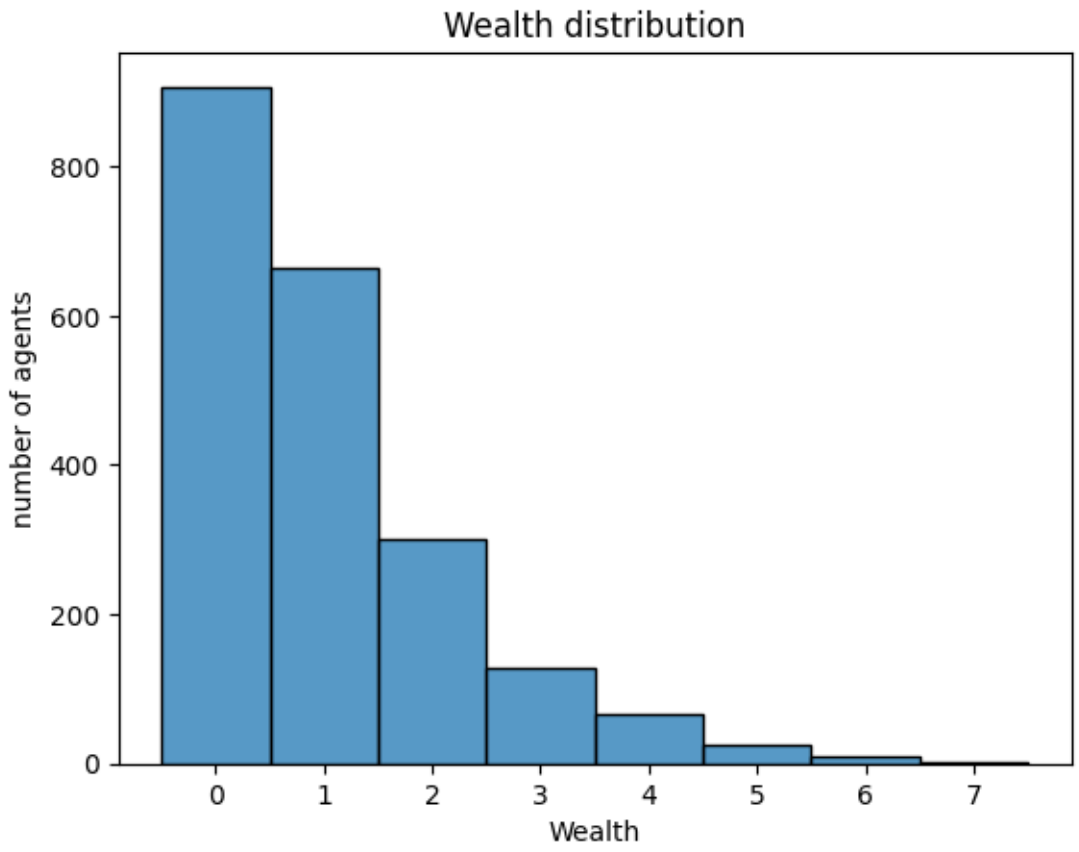
```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B


class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        """initialize a MoneyAgent instance.

        Args:
            model: A model instance
        """
        super().__init__(model)
        self.cell = cell
        self.wealth = 1

    def move(self):
        """Move the agent to a random neighboring cell."""
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        """Give 1 unit of wealth to a random agent in the same cell."""
        cellmates = [a for a in self.cell.agents if a is not self]

        if cellmates:  # Only give money if there are other agents present
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

    def step(self):
        """do one step of the agent."""
        self.move()
        if self.wealth > 0:
            self.give_money()


class MoneyModel(mesa.Model):
```

(continues on next page)

(continued from previous page)

```
    """A model with some number of agents."""

    def __init__(self, n=10, width=10, height=10, rng=None):
        """Initialize a MoneyModel instance.

        Args:
            N: The number of agents.
            width: width of the grid.
            height: Height of the grid.
        """
        super().__init__(rng=rng)
        self.num_agents = n
        self.grid = OrthogonalMooreGrid((width, height), random=self.random)

        # Create agents
        MoneyAgent.create_agents(
            self,
            self.num_agents,
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini}, agent_reporters={"Wealth": "wealth"}
        )
        self.datacollector.collect(self)

    def step(self):
        """do one step of the model"""
        self.agents.shuffle_do("step")
        self.datacollector.collect(self)
```

```
# Lets make sure the model works
model = MoneyModel(100, 10, 10)
model.run_for(20)


data = model.datacollector.get_agent_vars_dataframe()
# Use seaborn
g = sns.histplot(data["Wealth"], discrete=True)
g.set(title="Wealth distribution", xlabel="Wealth", ylabel="number of agents");
```




#### 2.4.2.6.9.5 Adding visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components, and (for advanced users) how to create a new visualization element.

First, a quick explanation of how Mesa's interactive visualization works. The visualization is done in a browser window or Jupyter instance, using the Solara framework, a pure Python, React-style web framework. Running `solara run app.py` will launch a web server, which runs the model, and displays model detail at each step via a plotting library. Alternatively, you can execute everything inside a Jupyter instance and display it inline.

As like last time we then instantiate the model parameters, some of which are modifiable by user inputs. In this case, the number of agents, N, is specified as a slider of integers.

```
model_params = {
    "n": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
```

(continues on next page)

(continued from previous page)

```
    "width": 10,
    "height": 10,
}
```

Then just like last time we instantiate the visualization object which (by default) displays the grid containing the agents, and timeseries of values computed by the model's data collector. In this example, we specify the Gini coefficient.

There are 3 buttons:

- the step button, which advances the model by 1 step
- the play button, which advances the model indefinitely until it is paused
- the pause button, which pauses the model

To reset the model, the order of operations are important

1. Stop the model
2. Update the parameters (e.g. move the sliders)
3. Press reset

**Additional Interactive Controls**

In addition to the basic controls (Play, Pause, Step), there are three extra interactive UI elements that give you more control over the simulation and visualization performance:

1. **Play Interval Slider** This slider controls the time delay (in milliseconds) between each step of the simulation when it is playing.
    - **Lower values** = faster simulation updates
    - **Higher values** = slower, more observable step-by-step updates
2. **Render Interval Slider** This slider determines how frequently the visualization updates, based on the number of steps.
    - For example, if set to `5`, the visualization will update only **after every 5 steps** of the model.
    - Note: This interval is **step-based**, not time-based.
3. **Use Threads Checkbox** This checkbox enables threaded execution of the model.
    - When enabled, the visualization runs on a separate thread, allowing the UI to remain responsive even during heavy computations.
    - It also ensures the visualization only updates at fixed intervals, improving performance and responsiveness during rapid simulations.

#### 2.4.2.6.9.6 Page Tab View

#### 2.4.2.6.9.7 Plot Components

You can place different components (except the renderer) on separate pages according to your preference. There are no restrictions on page numbering — pages do not need to be sequential or positive. Each page acts as an independent window where components may or may not exist.

The default page is `page=0`. If pages are not sequential (e.g., `page=1` and `page=10`), the system will automatically create the 8 empty pages in between to maintain consistent indexing. To avoid empty pages in your dashboard, use sequential page numbers.

To assign a plot component to a specific page, pass the `page` keyword argument to `make_plot_component`. For example, the following will display the plot component on page 1:

```
plot_comp = make_plot_component("encoding", page=1)
```

#### 2.4.2.6.9.8 Custom Components

In tutorial 8, you will learn how to create custom components for the Solara dashboard. If you want a custom component to appear on a specific page, you must pass it as a tuple containing the component and the page number.

```
@solara.component
def CustomComponent():
    ...

page = SolaraViz(
    model,
    renderer,
    components=[(CustomComponent, 1)]  # Custom component will appear on page 1
)
```

**Warning** Running the model can be performance-intensive. It is strongly recommended to pause the model in the dashboard before switching pages.

#### 2.4.2.6.9.9 Advanced Rendering with `SpaceRenderer`

**Important:** Mesa supports both `matplotlib` and `altair` backends for rendering, but **they are not interchangeable** when it comes to visualization methods and `post_process` functions.

- `matplotlib`-specific functions like `ax.set_title()` will not work with `altair`, and vice versa.
- **Be sure to run only the code blocks corresponding to the backend you are using**.
- Mixing backend-specific calls will lead to runtime errors.

`SpaceRenderer` is a powerful tool in Mesa for visualizing spatial grids. It goes beyond simply drawing agents — it allows detailed customization of the grid, dynamic styling of agents, and the ability to overlay calculated values as *property layers*. Property layers effectively act as heatmaps, coloring each grid cell based on model-defined data (more on this in the next tutorial).

In this section, we'll demonstrate:

- Styling the grid using `setup_structure()`
- Customizing agent appearance with `setup_agents()`
- Enhancing the final visualization using the `post_process` function
- Applying these techniques to both `matplotlib` and `altair` backends

Before we begin, we'll reuse the `agent_portrayal` function and `money_model` defined in the previous tutorial.

```
def agent_portrayal(agent):
    portrayal = AgentPortrayalStyle(size=50, color="orange")
    if agent.wealth > 0:
        portrayal.update(("color", "blue"), ("size", 100))
    return portrayal
```

(continues on next page)

(continued from previous page)

```
# Create initial model instance
money_model = MoneyModel(n=50, width=10, height=10)
```

#### 2.4.2.6.9.10 Drawing the Grid and Agents

We'll now create a renderer and draw both the grid structure and the agents using the `matplotlib` backend.

```
%%capture

renderer = SpaceRenderer(model=money_model, backend="matplotlib")
renderer.setup_structure(lw=2, ls="solid", color="black", alpha=0.1)
renderer.setup_agents(agent_portrayal)
renderer.render()
```

You can pass drawing keyword arguments (kwargs) directly to setup_structure() to customize the grid appearance. See the full list of accepted arguments in the SpaceDrawer documentation.

#### 2.4.2.6.9.11 Using Altair Backend

The same can be achieved with the `altair` backend:

```
%%capture

renderer = SpaceRenderer(model=money_model, backend="altair")
renderer.setup_structure(
    xlabel="x",
    ylabel="y",
    grid_width=2,
    grid_dash=[1],
    grid_color="black",
    grid_opacity=0.1,
    title="Boltzmann Wealth Model",
)
renderer.setup_agents(agent_portrayal)
renderer.render()
```

#### 2.4.2.6.9.12 Customizing the Final Output with `post_process`

You can use a post_process function to make high-level visual tweaks after the rendering is complete. This is especially useful for setting axis labels, titles, aspect ratios, and more.

The post_process can be passed into either the render() function or set differently as a property of SpaceRenderer.

**With `matplotlib`:**

```
def post_process(ax):
    """Customize the matplotlib axes after rendering."""
    ax.set_title("Boltzmann Wealth Model")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.grid(True, which="both", linestyle="--", linewidth=0.5, alpha=0.5)
    ax.set_aspect("equal", adjustable="box")
```

(continues on next page)

(continued from previous page)

```
renderer.post_process = post_process
```

**With `Altair`:**

```
def post_process(chart):
    """Customize the Altair chart after rendering."""
    chart = (
        chart.properties(
            title="Boltzmann Wealth Model",
            width=600,
            height=400,
        )
        .configure_axis(
            labelFontSize=12,
            titleFontSize=14,
        )
        .configure_title(fontSize=16)
    )
    return chart


renderer.post_process = post_process
```

You can also use `post_process` with plots or other components.

#### 2.4.2.6.9.13 Post-Processing Line Plots

> **NOTE**: The backend of plot components can be the same as, or different from, that of the SpaceRenderer, depending on your preference.

Let's apply `post_process` to a line chart of the Gini coefficient over time:

```
def post_process_lines(ax):
    """Customize the matplotlib axes for the Gini line plot."""
    ax.set_title("Gini Coefficient Over Time")
    ax.set_xlabel("Time Step")
    ax.set_ylabel("Gini Coefficient")
    ax.grid(True, which="both", linestyle="--", linewidth=0.5, alpha=0.5)
    ax.set_aspect("auto")


GiniPlot = make_plot_component("Gini", post_process=post_process_lines)
```

#### 2.4.2.6.9.14 Launching the Full Visualization

Now that we have the model, visual renderer, and plot components defined, we can bring everything together using `SolaraViz`:

```
page = SolaraViz(
    money_model,
    renderer,
    components=[GiniPlot],
```

(continues on next page)

(continued from previous page)

```
    model_params=model_params,
    name="Boltzmann Wealth Model",
)

# This is required to render the visualization in a Jupyter notebook
page
```

```
Cannot show ipywidgets in text
```

#### 2.4.2.6.9.15 Exercise

- Try different agent shapes, colors, and grid styles.
- Experiment with post_process to improve the look and feel of the final output.
- Add additional time-series plots.

#### 2.4.2.6.9.16 Next Steps

Checkout mesa examples to further explore the capabilities of the visualization stack. Check out the next *property layer visualization* on how to further enhance your interactive dashboard.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

### 2.4.2.6.10 Visualization - Property Layer Visualization

#### 2.4.2.6.10.1 The Boltzmann Wealth Model

If you want to get straight to the tutorial checkout these environment providers: (This can take 30 seconds to 5 minutes to load)

Due to conflict with Colab and Solara there are no colab links for this tutorial

*If you are running locally, please ensure you have the latest Mesa version installed.*

#### 2.4.2.6.10.2 Tutorial Description

This tutorial builds upon the *Visualization Rendering with SpaceRenderer* tutorial. We will explore more advanced features of the SpaceRenderer to create property_layer and their visualization.

*If you are starting here please see the* Running Your First Model tutorial *for dependency and start-up instructions*

#### 2.4.2.6.10.3 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
pip install mesa solara
```

```
Requirement already satisfied: mesa in /home/docs/checkouts/readthedocs.org/user_builds/
↪mesa/envs/latest/lib/python3.14/site-packages (4.0.0a0)
Requirement already satisfied: solara in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (1.57.4)
Requirement already satisfied: numpy in /home/docs/checkouts/readthedocs.org/user_builds/
↪mesa/envs/latest/lib/python3.14/site-packages (from mesa) (2.4.6)
Requirement already satisfied: pandas in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from mesa) (3.0.3)
Requirement already satisfied: scipy in /home/docs/checkouts/readthedocs.org/user_builds/
↪mesa/envs/latest/lib/python3.14/site-packages (from mesa) (1.17.1)
Requirement already satisfied: tqdm in /home/docs/checkouts/readthedocs.org/user_builds/
↪mesa/envs/latest/lib/python3.14/site-packages (from mesa) (4.67.3)
Requirement already satisfied: solara-server==1.57.4 in /home/docs/checkouts/readthedocs.
↪org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server[dev,
↪starlette]==1.57.4->solara) (1.57.4)
Requirement already satisfied: solara-ui==1.57.4 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui[all]==1.57.4-
↪>solara) (1.57.4)
Requirement already satisfied: click>=7.1.0 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server==1.57.4->
↪solara-server[dev,starlette]==1.57.4->solara) (8.4.1)
Requirement already satisfied: filelock in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server==1.57.4->
↪solara-server[dev,starlette]==1.57.4->solara) (3.29.1)
Requirement already satisfied: ipykernel!=7.0.0,!=7.0.1 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-
↪server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (7.2.0)
Requirement already satisfied: jinja2 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server==1.57.4->
↪solara-server[dev,starlette]==1.57.4->solara) (3.1.6)
Requirement already satisfied: jupyter-client in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server==1.57.4->
↪solara-server[dev,starlette]==1.57.4->solara) (8.8.0)
Requirement already satisfied: nbformat in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server==1.57.4->
↪solara-server[dev,starlette]==1.57.4->solara) (5.10.4)
Requirement already satisfied: rich-click in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server==1.57.4->
↪solara-server[dev,starlette]==1.57.4->solara) (1.9.8)
Requirement already satisfied: watchdog in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server[dev,
↪starlette]==1.57.4->solara) (6.0.0)
Requirement already satisfied: watchfiles in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server[dev,
↪starlette]==1.57.4->solara) (1.2.0)
Requirement already satisfied: starlette in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server[dev,
↪starlette]==1.57.4->solara) (0.52.1)
Requirement already satisfied: uvicorn in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server[dev,
↪starlette]==1.57.4->solara) (0.49.0)
Requirement already satisfied: websockets in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-server[dev,
```

(continues on next page)

(continued from previous page)

```
↪starlette]==1.57.4->solara) (16.0)
Requirement already satisfied: humanize in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui==1.57.4->solara-
↪ui[all]==1.57.4->solara) (4.15.0)
Requirement already satisfied: ipyvue>=1.9.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui==1.57.4->
↪solara-ui[all]==1.57.4->solara) (1.12.0)
Requirement already satisfied: ipyvuetify>=1.6.10 in /home/docs/checkouts/readthedocs.
↪org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui==1.57.4->
↪solara-ui[all]==1.57.4->solara) (1.11.3)
Requirement already satisfied: ipywidgets>=7.7 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui==1.57.4->
↪solara-ui[all]==1.57.4->solara) (8.1.8)
Requirement already satisfied: reacton>=1.9 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui==1.57.4->solara-
↪ui[all]==1.57.4->solara) (1.9.1)
Requirement already satisfied: requests in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui==1.57.4->solara-
↪ui[all]==1.57.4->solara) (2.34.2)
Requirement already satisfied: cachetools in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui[all]==1.57.4->
↪solara) (7.1.4)
Requirement already satisfied: markdown in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui[all]==1.57.4->
↪solara) (3.10.2)
Requirement already satisfied: pillow in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui[all]==1.57.4->
↪solara) (12.2.0)
Requirement already satisfied: pygments in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui[all]==1.57.4->
↪solara) (2.20.0)
Requirement already satisfied: pymdown-extensions in /home/docs/checkouts/readthedocs.
↪org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from solara-ui[all]==1.
↪57.4->solara) (10.21.3)
Requirement already satisfied: comm>=0.1.1 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.1->
↪solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (0.2.3)
Requirement already satisfied: debugpy>=1.6.5 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.
↪1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (1.8.21)
Requirement already satisfied: ipython>=7.23.1 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.
↪1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (9.14.0)
Requirement already satisfied: jupyter-core!=6.0.*,>=5.1 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->
↪solara) (5.9.1)
Requirement already satisfied: matplotlib-inline>=0.1 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->
↪solara) (0.2.2)
Requirement already satisfied: nest-asyncio>=1.4 in /home/docs/checkouts/readthedocs.org/
```

(continues on next page)

(continued from previous page)

```
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.
↪1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (1.6.0)
Requirement already satisfied: packaging>=22 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.
↪1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (26.2)
Requirement already satisfied: psutil>=5.7 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.1->
↪solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (7.2.2)
Requirement already satisfied: pyzmq>=25 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.1->
↪solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (27.1.0)
Requirement already satisfied: tornado>=6.4.1 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.
↪1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (6.5.6)
Requirement already satisfied: traitlets>=5.4.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipykernel!=7.0.0,!=7.0.
↪1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (5.15.1)
```

```
Requirement already satisfied: decorator>=5.1.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipython>=7.23.1->
↪ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->
↪solara) (5.3.1)
Requirement already satisfied: ipython-pygments-lexers>=1.0.0 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪ipython>=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (1.1.1)
Requirement already satisfied: jedi>=0.18.2 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from ipython>=7.23.1->ipykernel!
↪=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara)␣
↪(0.20.0)
Requirement already satisfied: pexpect>4.6 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from ipython>=7.23.1->ipykernel!
↪=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara)␣
↪(4.9.0)
Requirement already satisfied: prompt_toolkit<3.1.0,>=3.0.41 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪ipython>=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (3.0.52)
Requirement already satisfied: stack_data>=0.6.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from ipython>=7.23.1->
↪ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->
↪solara) (0.6.3)
Requirement already satisfied: wcwidth in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from prompt_toolkit<3.1.0,>=3.0.
↪41->ipython>=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-
↪server[dev,starlette]==1.57.4->solara) (0.7.0)
Requirement already satisfied: widgetsnbextension~=4.0.14 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪ipywidgets>=7.7->solara-ui==1.57.4->solara-ui[all]==1.57.4->solara) (4.0.15)
Requirement already satisfied: jupyterlab_widgets~=3.0.15 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪ipywidgets>=7.7->solara-ui==1.57.4->solara-ui[all]==1.57.4->solara) (3.0.16)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: parso<0.9.0,>=0.8.6 in /home/docs/checkouts/readthedocs.
↪org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from jedi>=0.18.2->
↪ipython>=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (0.8.7)
Requirement already satisfied: python-dateutil>=2.8.2 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪jupyter-client->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara)␣
↪(2.9.0.post0)
Requirement already satisfied: platformdirs>=2.5 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from jupyter-core!=6.0.*,>
↪=5.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,starlette]==1.
↪57.4->solara) (4.10.0)
Requirement already satisfied: ptyprocess>=0.5 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from pexpect>4.6->ipython>
↪=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (0.7.0)
Requirement already satisfied: six>=1.5 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from python-dateutil>=2.8.2->
↪jupyter-client->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara)␣
↪(1.17.0)
Requirement already satisfied: typing-extensions>=4.1.1 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪reacton>=1.9->solara-ui==1.57.4->solara-ui[all]==1.57.4->solara) (4.15.0)
```

```
Requirement already satisfied: executing>=1.2.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from stack_data>=0.6.0->
↪ipython>=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (2.2.1)
Requirement already satisfied: asttokens>=2.1.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from stack_data>=0.6.0->
↪ipython>=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (3.0.1)
Requirement already satisfied: pure-eval in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from stack_data>=0.6.0->ipython>
↪=7.23.1->ipykernel!=7.0.0,!=7.0.1->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (0.2.3)
Requirement already satisfied: MarkupSafe>=2.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from jinja2->solara-
↪server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (3.0.3)
Requirement already satisfied: fastjsonschema>=2.15 in /home/docs/checkouts/readthedocs.
↪org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from nbformat->solara-
↪server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (2.21.2)
Requirement already satisfied: jsonschema>=2.6 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from nbformat->solara-
↪server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (4.26.0)
Requirement already satisfied: attrs>=22.2.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from jsonschema>=2.6->
↪nbformat->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (26.1.0)
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /home/docs/
↪checkouts/readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages␣
↪(from jsonschema>=2.6->nbformat->solara-server==1.57.4->solara-server[dev,
↪starlette]==1.57.4->solara) (2025.9.1)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: referencing>=0.28.4 in /home/docs/checkouts/readthedocs.
↪org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from jsonschema>=2.6->
↪nbformat->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (0.37.0)
Requirement already satisfied: rpds-py>=0.25.0 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from jsonschema>=2.6->
↪nbformat->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (2026.5.
↪1)
```

```
Requirement already satisfied: pyyaml in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from pymdown-extensions->solara-
↪ui[all]==1.57.4->solara) (6.0.3)
Requirement already satisfied: charset_normalizer<4,>=2 in /home/docs/checkouts/
↪readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from␣
↪requests->solara-ui==1.57.4->solara-ui[all]==1.57.4->solara) (3.4.7)
Requirement already satisfied: idna<4,>=2.5 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from requests->solara-ui==1.57.4-
↪>solara-ui[all]==1.57.4->solara) (3.18)
Requirement already satisfied: urllib3<3,>=1.26 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from requests->solara-ui==1.
↪57.4->solara-ui[all]==1.57.4->solara) (2.7.0)
Requirement already satisfied: certifi>=2023.5.7 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from requests->solara-ui==1.
↪57.4->solara-ui[all]==1.57.4->solara) (2026.5.20)
Requirement already satisfied: rich>=12 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from rich-click->solara-
↪server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (15.0.0)
Requirement already satisfied: markdown-it-py>=2.2.0 in /home/docs/checkouts/readthedocs.
↪org/user_builds/mesa/envs/latest/lib/python3.14/site-packages (from rich>=12->rich-
↪click->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara) (4.2.0)
Requirement already satisfied: mdurl~=0.1 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from markdown-it-py>=2.2.0->rich>
↪=12->rich-click->solara-server==1.57.4->solara-server[dev,starlette]==1.57.4->solara)␣
↪(0.1.2)
Requirement already satisfied: anyio<5,>=3.6.2 in /home/docs/checkouts/readthedocs.org/
↪user_builds/mesa/envs/latest/lib/python3.14/site-packages (from starlette->solara-
↪server[dev,starlette]==1.57.4->solara) (4.13.0)
Requirement already satisfied: h11>=0.8 in /home/docs/checkouts/readthedocs.org/user_
↪builds/mesa/envs/latest/lib/python3.14/site-packages (from uvicorn->solara-server[dev,
↪starlette]==1.57.4->solara) (0.16.0)
```

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns

import mesa
```

(continues on next page)

(continued from previous page)

```
from mesa.discrete_space import CellAgent, OrthogonalMooreGrid

# Check Mesa version for visualization compatibility
if mesa.__version__.startswith(("3.0", "3.1", "3.2")):
    print(
        f"  Mesa {mesa.__version__} detected. Visualization features require Mesa 3.3+"
    )
    print("To upgrade: pip install --upgrade mesa")

from mesa.visualization import SolaraViz, SpaceRenderer, make_plot_component
from mesa.visualization.components import AgentPortrayalStyle, PropertyLayerStyle
```

#### 2.4.2.6.10.4 Basic Model

The following is the base model we'll use to build the dashboard. It's an extension of the model introduced in Tutorials 0–3, with an added property_layer called *Test* to demonstrate property_layer visualization functionalities.

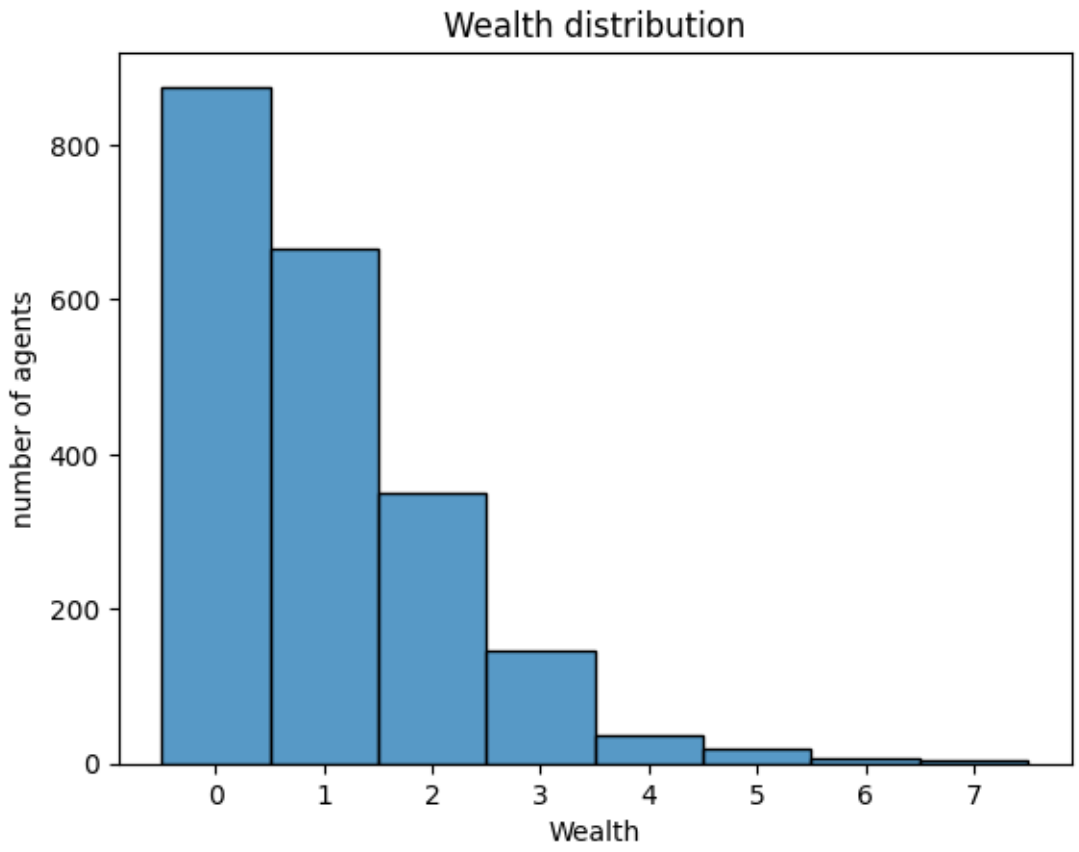
```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B


class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        """initialize a MoneyAgent instance.

        Args:
            model: A model instance
        """
        super().__init__(model)
        self.cell = cell
        self.wealth = 1

    def move(self):
        """Move the agent to a random neighboring cell."""
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        """Give 1 unit of wealth to a random agent in the same cell."""
        cellmates = [a for a in self.cell.agents if a is not self]

        if cellmates:  # Only give money if there are other agents present
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

    def step(self):
```

(continues on next page)

(continued from previous page)

```
        """do one step of the agent."""
        self.move()
        if self.wealth > 0:
            self.give_money()


class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10, width=10, height=10, rng=None):
        """Initialize a MoneyModel instance.

        Args:
            N: The number of agents.
            width: Width of the grid.
            height: Height of the grid.
        """
        super().__init__(rng=rng)
        self.num_agents = n
        self.grid = OrthogonalMooreGrid((width, height), random=self.random)

        # Add a test property with random data
        self.grid.add_property_layer(
            "test", np.random.randint(0, 10, size=(width, height))
        )

        # Create agents
        MoneyAgent.create_agents(
            self,
            self.num_agents,
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini}, agent_reporters={"Wealth": "wealth"}
        )
        self.datacollector.collect(self)

    def step(self):
        """do one step of the model"""
        self.agents.shuffle_do("step")
        self.datacollector.collect(self)
```

```
# Let's make sure the model works
model = MoneyModel(100, 10, 10)
model.run_for(20)


data = model.datacollector.get_agent_vars_dataframe()
# Use seaborn
g = sns.histplot(data["Wealth"], discrete=True)
g.set(title="Wealth distribution", xlabel="Wealth", ylabel="number of agents");
```




#### 2.4.2.6.10.5 Adding visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components, and (for advanced users) how to create a new visualization element.

First, a quick explanation of how Mesa's interactive visualization works. The visualization is done in a browser window or Jupyter instance, using the Solara framework, a pure Python, React-style web framework. Running `solara run app.py` will launch a web server, which runs the model, and displays model detail at each step via a plotting library. Alternatively, you can execute everything inside a Jupyter instance and display it inline.

As in the previous tutorial we instantiate the model parameters, some of which are modifiable by user inputs. In this case, the number of agents, N, is specified as a slider of integers.

```
model_params = {
    "n": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
```

(continues on next page)

(continued from previous page)

```
    "width": 10,
    "height": 10,
}
```

Then just like last time we instantiate the visualization object which (by default) displays the grid containing the agents, and timeseries of values computed by the model's data collector. In this example, we specify the Gini coefficient.

There are 3 buttons:

- the step button, which advances the model by 1 step
- the play button, which advances the model indefinitely until it is paused
- the pause button, which pauses the model

To reset the model, the order of operations are important

1. Stop the model
2. Update the parameters (e.g. move the sliders)
3. Press reset

**Additional Interactive Controls**

In addition to the basic controls (Play, Pause, Step), there are three extra interactive UI elements that give you more control over the simulation and visualization performance:

1. **Play Interval Slider** This slider controls the time delay (in milliseconds) between each step of the simulation when it is playing.
    - **Lower values** = faster simulation updates
    - **Higher values** = slower, more observable step-by-step updates
2. **Render Interval Slider** This slider determines how frequently the visualization updates, based on the number of steps.
    - For example, if set to `5`, the visualization will update only **after every 5 steps** of the model.
    - Note: This interval is **step-based**, not time-based.
3. **Use Threads Checkbox** This checkbox enables threaded execution of the model.
    - When enabled, the visualization runs on a separate thread, allowing the UI to remain responsive even during heavy computations.
    - It also ensures the visualization only updates at fixed intervals, improving performance and responsiveness during rapid simulations.

#### 2.4.2.6.10.6 Page Tab View

#### 2.4.2.6.10.7 Plot Components

You can place different components (except the renderer) on separate pages according to your preference. There are no restrictions on page numbering — pages do not need to be sequential or positive. Each page acts as an independent window where components may or may not exist.

The default page is `page=0`. If pages are not sequential (e.g., `page=1` and `page=10`), the system will automatically create the 8 empty pages in between to maintain consistent indexing. To avoid empty pages in your dashboard, use sequential page numbers.

To assign a plot component to a specific page, pass the `page` keyword argument to `make_plot_component`. For example, the following will display the plot component on page 1:

```
plot_comp = make_plot_component("encoding", page=1)
```

#### 2.4.2.6.10.8 Custom Components

In the next tutorial, you will learn how to create custom components for the Solara dashboard. If you want a custom component to appear on a specific page, you must pass it as a tuple containing the component and the page number.

```
@solara.component
def CustomComponent():
    ...

page = SolaraViz(
    model,
    renderer,
    components=[(CustomComponent, 1)]  # Custom component will appear on page 1
)
```

**Warning** Running the model can be performance-intensive. It is strongly recommended to pause the model in the dashboard before switching pages.

#### 2.4.2.6.10.9 Visualizing property_layer

**Important:** property_layer visualization on `HexGrid` is not supported with the `altair` backend; use `matplotlib` instead.

You can visualize **property_layer** in a way that's very similar to how agents are visualized—by defining a custom portrayal function. Let's call this function `property_layer_portrayal`.

Mesa provides a dedicated component for property_layer styling, called `PropertyLayerStyle` (similar to `AgentPortrayalStyle` for agents). You can import it from `mesa.visualization.components` as shown earlier.

In `PropertyLayerStyle`, you can define:

- `color` or `colormap`: Determines how the values in the property_layer are visualized
- `alpha`: Controls the transparency (opacity) of the property_layer
- `colorbar`: A boolean that determines whether a colorbar is shown alongside the visualization
- `vmin` and `vmax`: The minimum and maximum data values to be visualized, controlling the color scale range, these default to the minimum and maximum values in your data respectively if not defined.

The portrayal function receives a `layer` object as an argument, just like how `agent_portrayal` receives an `agent`. If your model includes multiple property_layer, you can conditionally adjust the visualization logic based on the its name. This allows you to apply different styles to each property_layer as needed.

Here's a quick example:

```
def property_layer_portrayal(layer):
    if layer == "WealthDensity":
        return PropertyLayerStyle(
            colormap="viridis",
            alpha=0.6,
            colorbar=True,
            vmin=0,
```

(continues on next page)

(continued from previous page)

```
            vmax=10,
        )
    elif layer == "Temperature":
        return PropertyLayerStyle(
            colormap="coolwarm",
            alpha=0.5,
            colorbar=False,
            vmin=-1,
            vmax=1,
        )
```

This approach allows you to customize each property_layer's appearance independently while keeping your visualization code clean and modular.

We'll reuse the previously defined `agent_portrayal` function and introduce a new `property_layer_portrayal` function specifically for visualizing the property_layer.

```
def agent_portrayal(agent):
    portrayal = AgentPortrayalStyle(size=50, color="orange")
    if agent.wealth > 0:
        portrayal.update(("color", "blue"), ("size", 100))
    return portrayal


def property_layer_portrayal(layer):
    if layer == "test":
        return PropertyLayerStyle(color="blue", alpha=0.8, colorbar=True)


# Create initial model instance
money_model = MoneyModel(n=50, width=10, height=10)
```

#### 2.4.2.6.10.10 Drawing the property_layer

We'll now create a renderer and draw the grid structure, the agents and the *test* using the `matplotlib` backend.

```
%%capture

renderer = SpaceRenderer(model=money_model, backend="matplotlib")
renderer.draw_structure(lw=2, ls="solid", color="black", alpha=0.1)
renderer.draw_agents(agent_portrayal)
renderer.draw_property_layer(property_layer_portrayal)
```

You can also use `render()` function to draw property_layer in one go.

```
%%capture

renderer = SpaceRenderer(model=money_model, backend="matplotlib")
renderer.render(
    space_kwargs={  # an alternative way to customize the grid structure
        "lw": 2,
        "ls": "solid",
        "color": "black",
```

(continues on next page)

(continued from previous page)

```
        "alpha": 0.1,
    },
    agent_portrayal=agent_portrayal,
    property_layer_portrayal=property_layer_portrayal,
)
```

We'll keep the `post_process` to make the grid look good.

```
def post_process(ax):
    """Customize the matplotlib axes after rendering."""
    ax.set_title("Boltzmann Wealth Model")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.grid(True, which="both", linestyle="--", linewidth=0.5, alpha=0.5)
    ax.set_aspect("equal", adjustable="box")


renderer.post_process = post_process


def post_process_lines(ax):
    """Customize the matplotlib axes for the Gini line plot."""
    ax.set_title("Gini Coefficient Over Time")
    ax.set_xlabel("Time Step")
    ax.set_ylabel("Gini Coefficient")
    ax.grid(True, which="both", linestyle="--", linewidth=0.5, alpha=0.5)
    ax.set_aspect("auto")


GiniPlot = make_plot_component("Gini", post_process=post_process_lines)
```

#### 2.4.2.6.10.11 Launching the Visualization

Now that we have the model, visual renderer, and plot components defined, we can bring everything together using `SolaraViz`:

```
page = SolaraViz(
    money_model,
    renderer,
    components=[GiniPlot],
    model_params=model_params,
    name="Boltzmann Wealth Model",
)

# This is required to render the visualization in a Jupyter notebook
page
```

```
Cannot show ipywidgets in text
```

```
<Figure size 640x480 with 0 Axes>
```

#### 2.4.2.6.10.12 Exercise

- Try removing the `post_process` and changing the backend.
- Try visualizing multiple property_layers.

#### 2.4.2.6.10.13 Next Steps

Checkout this mesa example to further explore the capabilities of the property_layer. Check out the next *visualization tutorial custom components* on how to further enhance your interactive dashboard.

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

### 2.4.2.6.11 Visualization - Custom Components

#### 2.4.2.6.11.1 The Boltzmann Wealth Model

If you want to get straight to the tutorial checkout these environment providers: (This can take 30 seconds to 5 minutes to load)

Due to conflict with Colab and Solara there are no colab links for this tutorial

*If you are running locally, please ensure you have the latest Mesa version installed.*

#### 2.4.2.6.11.2 Tutorial Description

This tutorial extends the Boltzmann wealth model from the *Visualization Basic Dashboard tutorial*, by adding an interactive dashboard.

In this portion, we will demonstrate how users can employ create dynamic agent representation with their Mesa dashboards. This is part two of three visualization tutorials.

*If you are starting here please see the* Running Your First Model tutorial *for dependency and start-up instructions*

#### 2.4.2.6.11.3 Import Dependencies

This includes importing of dependencies needed for the tutorial.

```
# Has multi-dimensional arrays and matrices.
# Has a large collection of mathematical functions to operate on these arrays.
import numpy as np

# Data manipulation and analysis.
import pandas as pd

# Data visualization tools.
import seaborn as sns

import mesa
from mesa.discrete_space import CellAgent, OrthogonalMooreGrid

# Check Mesa version for visualization compatibility
```

(continues on next page)

(continued from previous page)

```
if mesa.__version__.startswith(("3.0", "3.1", "3.2")):
    print(
        f"  Mesa {mesa.__version__} detected. Visualization features require Mesa 3.3+"
    )
    print("To upgrade: pip install --upgrade mesa")

from mesa.visualization import SolaraViz, make_plot_component, make_space_component
```

#### 2.4.2.6.11.4 Basic Model

The following is the basic model we will be using to build the dashboard. This is the same model seen in tutorials 0-3.

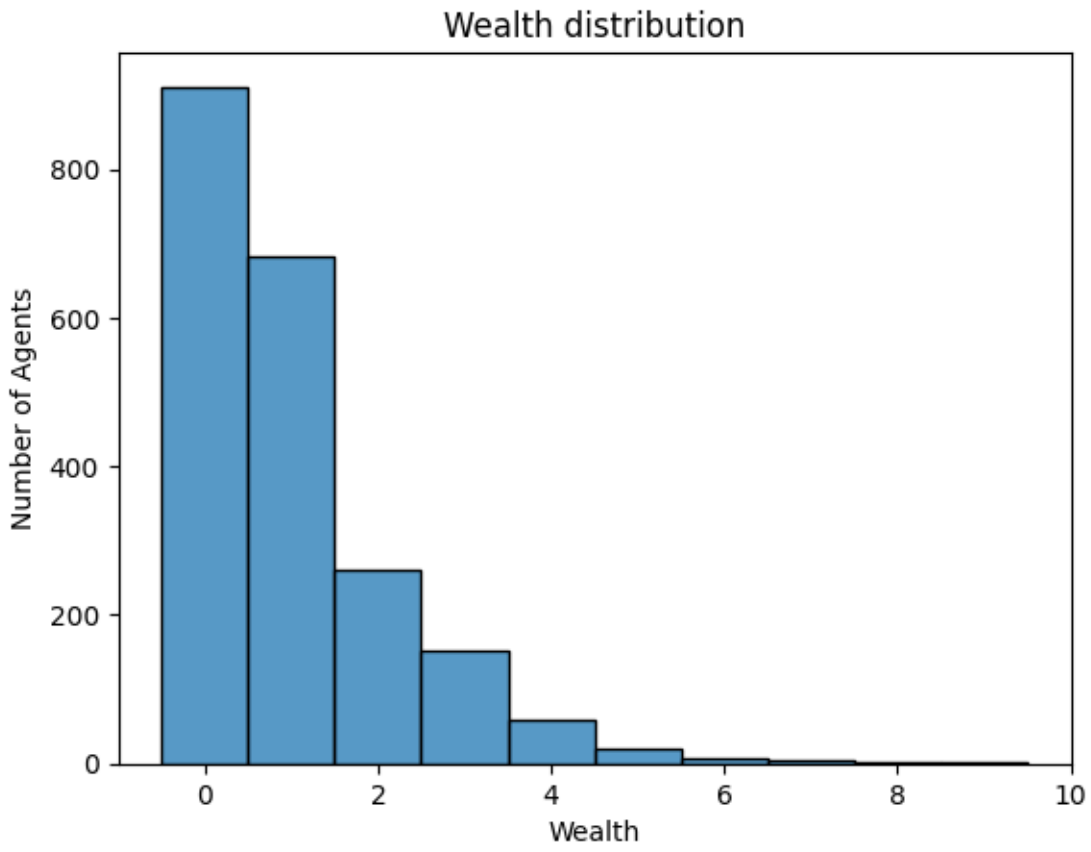
```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
    return 1 + (1 / N) - 2 * B


class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth."""

    def __init__(self, model, cell):
        """initialize a MoneyAgent instance.

        Args:
            model: A model instance
        """
        super().__init__(model)
        self.cell = cell
        self.wealth = 1

    def move(self):
        """Move the agent to a random neighboring cell."""
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        """Give 1 unit of wealth to a random agent in the same cell."""
        cellmates = [a for a in self.cell.agents if a is not self]

        if cellmates:  # Only give money if there are other agents present
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

    def step(self):
        """do one step of the agent."""
        self.move()
        if self.wealth > 0:
            self.give_money()
```

(continues on next page)

(continued from previous page)

```
class MoneyModel(mesa.Model):
    """A model with some number of agents."""

    def __init__(self, n=10, width=10, height=10, rng=None):
        """Initialize a MoneyModel instance.

        Args:
            N: The number of agents.
            width: width of the grid.
            height: Height of the grid.
        """
        super().__init__(rng=rng)
        self.num_agents = n
        self.grid = OrthogonalMooreGrid((width, height), random=self.random)

        # Create agents
        MoneyAgent.create_agents(
            self,
            self.num_agents,
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

        self.datacollector = mesa.DataCollector(
            model_reporters={"Gini": compute_gini}, agent_reporters={"Wealth": "wealth"}
        )
        self.datacollector.collect(self)

    def step(self):
        """do one step of the model"""
        self.agents.shuffle_do("step")
        self.datacollector.collect(self)
```

```
# Lets make sure the model works
model = MoneyModel(100, 10, 10)
model.run_for(20)


data = model.datacollector.get_agent_vars_dataframe()
# Use seaborn
g = sns.histplot(data["Wealth"], discrete=True)
g.set(title="Wealth distribution", xlabel="Wealth", ylabel="Number of Agents");
```




#### 2.4.2.6.11.5 Adding visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components, and (for advanced users) how to create a new visualization element.

First, a quick explanation of how Mesa's interactive visualization works. The visualization is done in a browser window or Jupyter instance, using the Solara framework, a pure Python, React-style web framework. Running `solara run app.py` will launch a web server, which runs the model, and displays model detail at each step via a plotting library. Alternatively, you can execute everything inside a Jupyter instance and display it inline.

*Thanks to @Corvince for all his work creating Mesa's visualization capability*

#### 2.4.2.6.11.6 Building Custom Components

This section is for users who have a basic familiarity with Python's Matplotlib plotting library.

If the visualization elements provided by Mesa aren't enough for you, you can build your own and plug them into the model server.

For this example, let's build a simple histogram visualization, which can count the number of agents with each value of wealth.

First we need to update our imports

We use Matplotlib in this tutorial, but Mesa also has Altair. If you would like other visualization support like Plotly or Bokeh, please feel free to contribute

In addition, due to the way Solara works we need to trigger an update whenever the underlying model changes. For this you need to register an update counter with every component.

```
import solara
from matplotlib.figure import Figure

from mesa.visualization.utils import update_counter
```

Next we provide a function for our agent portrayal and our model parameters.

```
def agent_portrayal(agent):
    size = 10
    color = "tab:red"
    if agent.wealth > 0:
        size = 50
        color = "tab:blue"
    return {"size": size, "color": color}


model_params = {
    "n": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
    "width": 10,
    "height": 10,
}
```

#### 2.4.2.6.11.7 Page Tab View

#### 2.4.2.6.11.8 Plot Components

You can place different components (except the renderer) on separate pages according to your preference. There are no restrictions on page numbering — pages do not need to be sequential or positive. Each page acts as an independent window where components may or may not exist.

The default page is `page=0`. If pages are not sequential (e.g., `page=1` and `page=10`), the system will automatically create the 8 empty pages in between to maintain consistent indexing. To avoid empty pages in your dashboard, use sequential page numbers.

To assign a plot component to a specific page, pass the `page` keyword argument to `make_plot_component`. For example, the following will display the plot component on page 1:

```
plot_comp = make_plot_component("encoding", page=1)
```

#### 2.4.2.6.11.9 Custom Components

If you want a custom component to appear on a specific page, you must pass it as a tuple containing the component and the page number.

```
@solara.component
def CustomComponent():
    ...

page = SolaraViz(
    model,
    renderer,
    components=[(CustomComponent, 1)]  # Custom component will appear on page 1
)
```

**Warning** Running the model can be performance-intensive. It is strongly recommended to pause the model in the dashboard before switching pages.

Now we add our custom component. In this case we will build a histogram of agent wealth.

Besides the standard matplotlib code to build a histogram, please notice 3 key features.

1. `@solara.component` this is needed for any compoenent you add
2. `update_counter.get()` this is needed so solara updates the dashboard with your agent based model
3. you must initialize a `figure` using this method instead of `plt.figure()`, for thread safety purpose

```
@solara.component
def Histogram(model):
    update_counter.get()  # This is required to update the counter
    # Note: you must initialize a figure using this method instead of
    # plt.figure(), for thread safety purpose
    fig = Figure()
    ax = fig.subplots()
    wealth_vals = [agent.wealth for agent in model.agents]
    # Note: you have to use Matplotlib's OOP API instead of plt.hist
    # because plt.hist is not thread-safe.
    ax.hist(wealth_vals, bins=10)
    solara.FigureMatplotlib(fig)
```

Now we create the model an initialize the visualization

```
# Create initial model instance
money_model = MoneyModel(n=50, width=10, height=10)

SpaceGraph = make_space_component(agent_portrayal)
GiniPlot = make_plot_component("Gini", page=1)

page = SolaraViz(
    money_model,
    components=[SpaceGraph, GiniPlot, (Histogram, 2)],
    model_params=model_params,
    name="Boltzmann Wealth Model",
)
# This is required to render the visualization in the Jupyter notebook
page
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
/home/docs/checkouts/readthedocs.org/user_builds/mesa/envs/latest/lib/python3.14/site-
↪packages/mesa/visualization/mpl_space_drawing.py:435: FutureWarning: Returning a dict␣
↪from agent_portrayal is deprecated and will be removed in Mesa 4.0. Please return an␣
↪AgentPortrayalStyle instance instead. For more information, refer to the migration␣
↪guide: https://mesa.readthedocs.io/latest/migration_guide.html#defining-portrayal-
↪components
  arguments = collect_agent_data(space, agent_portrayal, default_size=s_default)
```

```
Cannot show ipywidgets in text
```

You can even run the visuals independently by calling it with the model instance

```
Histogram(money_model)
```

```
Cannot show ipywidgets in text
```

#### 2.4.2.6.11.10 Exercise

- Build you own custom component

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in AgentBased Models." George Mason University, 2014. http://mars.gmu.edu/bitstream/handle/1920/9070/Comer_gmu_0883E_10539.pdf

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

### 2.4.2.6.12 Best Practices

Here are some general principles that have proven helpful for developing models.

#### 2.4.2.6.12.1 Model Layout

A model should be contained in a folder named with lower-case letters and underscores, such as `wolf_sheep`. Within that directory:

- `Readme.md` describes the model, how to use it, and any other details.
- `model.py` should contain the model class.
- `agents.py` should contain the agent class(es).
- `app.py` should contain the Solara-based visualization code (optional).

You can add more files as needed, for example:

- `run.py` could contain the code to run the model.
- `analysis.py` could contain any analysis code.

Input data can be stored in a `data` directory, output data in an `output`, processed results in a `results` directory, images in an `images` directory, etc.

All our *examples* follow this layout.

#### 2.4.2.6.12.2 Randomization

If your model involves some random choice, you can use the built-in `random` property that many Mesa objects have, including `Model`, `Agent`, and `AgentSet`. This works exactly like the built-in `random` library.

```
class AwesomeModel(Model):
  # ...

  def cool_method(self):
    interesting_number = self.random.random()
    print(interesting_number)

class AwesomeAgent(Agent):
  # ...
  def __init__(self, unique_id, model, ...):
    super().__init__(unique_id, model)
    # ...

  def my_method(self):
    random_number = self.random.randint(0, 100)
```

`Agent.random` is just a convenient shorthand in the Agent class to `self.model.random`. If you create your own `AgentSet` instances, you have to pass `random` explicitly. Typically, you can simply do, in a Model instance, `my_agentset = AgentSet([], random=self.random)`. This ensures that `my_agentset` uses the same random number generator as the rest of the model.

When a model object is created, its random property is automatically seeded with the current time. The seed determines the sequence of random numbers; if you instantiate a model with the same seed, you will get the same results. To allow you to set the seed, make sure your model has a `seed` argument in its `__init__`.

```
class AwesomeModel(Model):

  def __init__(self, seed=None):
    super().__init__(seed=seed)
    ...

  def cool_method(self):
    interesting_number = self.random.random()
    print(interesting_number)

>>> model0 = AwesomeModel(seed=0)
>>> model0._seed
0
>>> model0.cool_method()
0.8444218515250481
>>> model1 = AwesomeModel(seed=0)
>>> model1.cool_method()
0.8444218515250481
```

### 2.4.3 Mesa Core Examples

This repository contains a curated set of classic agent-based models implemented using Mesa. These core examples are maintained by the Mesa development team and serve as both demonstrations of Mesa's capabilities and starting points for your own models.

#### 2.4.3.1 Overview

The examples are categorized into two groups:

1. **Basic Examples** - Simpler models that use only stable Mesa features; ideal for beginners
2. **Advanced Examples** - More complex models that demonstrate additional concepts and may use some experimental features

   **Note:** Looking for more examples? Visit the mesa-examples repository for user-contributed models and showcases.

#### 2.4.3.2 Basic Examples

The basic examples are relatively simple and only use stable Mesa features. They are good starting points for learning how to use Mesa.

*An online version of these examples is available here.*

##### 2.4.3.2.1 *Boltzmann Wealth Model*

Completed code to go along with the tutorial on making a simple model of how a highly-skewed wealth distribution can emerge from simple rules.

##### 2.4.3.2.2 *Boids Flockers Model*

Boids-style flocking model, demonstrating the use of agents moving through a continuous space following direction vectors.

##### 2.4.3.2.3 *Conway's Game of Life*

Implementation of Conway's Game of Life, a cellular automata where simple rules can give rise to complex patterns.

##### 2.4.3.2.4 *Schelling Segregation Model*

Mesa implementation of the classic Schelling segregation model.

##### 2.4.3.2.5 *Virus on a Network Model*

This model is based on the NetLogo Virus on a Network model.

#### 2.4.3.3 Advanced Examples

The advanced examples are more complex and may use experimental Mesa features. They are good starting points for learning how to build more complex models.

##### 2.4.3.3.1 *Epstein Civil Violence Model*

Joshua Epstein's model of how a decentralized uprising can be suppressed or reach a critical mass of support.

##### 2.4.3.3.2 *Demographic Prisoner's Dilemma on a Grid*

Grid-based demographic prisoner's dilemma model, demonstrating how simple rules can lead to the emergence of widespread cooperation – and how a model activation regime can change its outcome.

#### 2.4.3.3.3 *Sugarscape Model with Traders*

This is Epstein & Axtell's Sugarscape model with Traders, a detailed description is in Chapter four of *Growing Artificial Societies: Social Science from the Bottom Up (1996)*. The model shows how emergent price equilibrium can happen via decentralized dynamics.

#### 2.4.3.3.4 *Wolf-Sheep Predation Model*

Implementation of an ecological model of predation and reproduction, based on the NetLogo Wolf Sheep Predation model.

##### 2.4.3.3.4.1 Conway's Game Of "Life"

##### 2.4.3.3.4.2 Summary

The Game of Life, also known simply as "Life", is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input by a human. One interacts with the Game of "Life" by creating an initial configuration and observing how it evolves, or, for advanced "players", by creating patterns with particular properties.

##### 2.4.3.3.4.3 How to Run

Install Mesa with recommended dependencies:

pip install "mesa[rec]"

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

##### 2.4.3.3.4.4 Files

- `agents.py`: Defines the behavior of an individual cell, which can be in two states: DEAD or ALIVE.
- `model.py`: Defines the model itself, initialized with a random configuration of alive and dead cells.
- `app.py`: Defines an interactive visualization using solara.
- `st_app.py`: Defines an interactive visualization using Streamlit.

##### 2.4.3.3.4.5 Optional

- For the streamlit version, you need to have streamlit installed (can be done via pip install streamlit)

##### 2.4.3.3.4.6 Further Reading

Conway's Game of Life

##### 2.4.3.3.4.7 Agents

```
from mesa.discrete_space import FixedAgent
```

(continues on next page)

(continued from previous page)

```
class Cell(FixedAgent):
    """Represents a single ALIVE or DEAD cell in the simulation."""

    DEAD = 0
    ALIVE = 1

    @property
    def x(self):
        return self.cell.coordinate[0]

    @property
    def y(self):
        return self.cell.coordinate[1]

    def __init__(self, model, cell, init_state=DEAD):
        """Create a cell, in the given state, at the given x, y position."""
        super().__init__(model)
        self.cell = cell
        self.state = init_state
        self._next_state = None

    @property
    def is_alive(self):
        return self.state == self.ALIVE

    @property
    def neighbors(self):
        return self.cell.neighborhood.agents

    def determine_state(self):
        """Compute if the cell will be dead or alive at the next tick.  This is
        based on the number of alive or dead neighbors.  The state is not
        changed here, but is just computed and stored in self._nextState,
        because our current state may still be necessary for our neighbors
        to calculate their next state.
        """
        # Get the neighbors and apply the rules on whether to be alive or dead
        # at the next tick.
        live_neighbors = sum(neighbor.is_alive for neighbor in self.neighbors)

        # Assume nextState is unchanged, unless changed below.
        self._next_state = self.state
        if self.is_alive:
            if live_neighbors < 2 or live_neighbors > 3:
                self._next_state = self.DEAD
        else:
            if live_neighbors == 3:
                self._next_state = self.ALIVE

    def assume_state(self):
        """Set the state to the new computed state -- computed in step()."""
        self.state = self._next_state
```

(continues on next page)

(continued from previous page)

#### 2.4.3.3.4.8 Model

```
from mesa import Model
from mesa.discrete_space import OrthogonalMooreGrid
from mesa.examples.basic.conways_game_of_life.agents import Cell


class ConwaysGameOfLife(Model):
    """Represents the 2-dimensional array of cells in Conway's Game of Life."""

    def __init__(self, width=50, height=50, initial_fraction_alive=0.2, rng=None):
        """Create a new playing area of (width, height) cells."""
        super().__init__(rng=rng)
        # Use a simple grid, where edges wrap around.
        self.grid = OrthogonalMooreGrid(
            (width, height), capacity=1, random=self.random, torus=True
        )

        # Place a cell at each location, with some initialized to
        # ALIVE and some to DEAD.
        for cell in self.grid.all_cells:
            Cell(
                self,
                cell,
                init_state=Cell.ALIVE
                if self.random.random() < initial_fraction_alive
                else Cell.DEAD,
            )

        self.running = True

    def step(self):
        """Perform the model step in two stages:
        - First, all cells assume their next state (whether they will be dead or alive)
        - Then, all cells change state to their next state.
        """
        self.agents.do("determine_state")
        self.agents.do("assume_state")
```

#### 2.4.3.3.4.9 App

```
from mesa.examples.basic.conways_game_of_life.model import ConwaysGameOfLife
from mesa.visualization import (
    SolaraViz,
    SpaceRenderer,
)
from mesa.visualization.components import AgentPortrayalStyle
```

(continues on next page)

(continued from previous page)

```
def agent_portrayal(agent):
    return AgentPortrayalStyle(
        color="white" if agent.state == 0 else "black",
        marker="s",
        size=30,
    )


def post_process(ax):
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "width": {
        "type": "SliderInt",
        "value": 50,
        "label": "Width",
        "min": 5,
        "max": 60,
        "step": 1,
    },
    "height": {
        "type": "SliderInt",
        "value": 50,
        "label": "Height",
        "min": 5,
        "max": 60,
        "step": 1,
    },
    "initial_fraction_alive": {
        "type": "SliderFloat",
        "value": 0.2,
        "label": "Cells initially alive",
        "min": 0,
        "max": 1,
        "step": 0.01,
    },
}

# Create initial model instance
model1 = ConwaysGameOfLife()

renderer = SpaceRenderer(model1, backend="matplotlib").setup_agents(agent_portrayal)
# In this case the renderer only draws the agents because we just want to observe
```

(continues on next page)

(continued from previous page)

```
# the state of the agents, not the structure of the grid.
renderer.draw_agents()
renderer.post_process = post_process

# Create the SolaraViz page. This will automatically create a server and display the
# visualization elements in a web browser.
# Display it using the following command in the example directory:
# solara run app.py
# It will automatically update and display any changes made to this file
page = SolaraViz(
    model1,
    renderer,
    model_params=model_params,
    name="Game of Life",
)
page  # noqa
```

#### 2.4.3.3.4.10 Virus on a Network

#### 2.4.3.3.4.11 Summary

This model is based on the NetLogo model "Virus on Network". It demonstrates the spread of a virus through a network and follows the SIR model, commonly seen in epidemiology.

The SIR model is one of the simplest compartmental models, and many models are derivatives of this basic form. The model consists of three compartments:

S: The number of susceptible individuals. When a susceptible and an infectious individual come into "infectious contact", the susceptible individual contracts the disease and transitions to the infectious compartment. I: The number of infectious individuals. These are individuals who have been infected and are capable of infecting susceptible individuals. R for the number of removed (and immune) or deceased individuals. These are individuals who have been infected and have either recovered from the disease and entered the removed compartment, or died. It is assumed that the number of deaths is negligible with respect to the total population. This compartment may also be called "recovered" or "resistant".

For more information about this model, read the NetLogo's web page: http://ccl.northwestern.edu/netlogo/models/VirusonaNetwork.

JavaScript library used in this example to render the network: d3.js.

#### 2.4.3.3.4.12 How to Run

Install Mesa with recommended dependencies:

pip install "mesa[rec]"

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.13 Files

- `model.py`: Contains the agent class, and the overall model class.
- `agents.py`: Contains the agent class.
- `app.py`: Contains the code for the interactive Solara visualization.

#### 2.4.3.3.4.14 Further Reading

Stonedahl, F. and Wilensky, U. (2008). NetLogo Virus on a Network model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wilensky, U. (1999). NetLogo Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

#### 2.4.3.3.4.15 Agents

```
from enum import Enum

from mesa.discrete_space import FixedAgent


class State(Enum):
    SUSCEPTIBLE = 0
    INFECTED = 1
    RESISTANT = 2


class VirusAgent(FixedAgent):
    """Individual Agent definition and its properties/interaction methods."""

    def __init__(
        self,
        model,
        initial_state,
        virus_spread_chance,
        virus_check_frequency,
        recovery_chance,
        gain_resistance_chance,
        cell,
    ):
        super().__init__(model)

        self.state = initial_state

        self.virus_spread_chance = virus_spread_chance
        self.virus_check_frequency = virus_check_frequency
        self.recovery_chance = recovery_chance
        self.gain_resistance_chance = gain_resistance_chance
        self.cell = cell

    def try_to_infect_neighbors(self):
        for agent in self.cell.neighborhood.agents:
            if (agent.state is State.SUSCEPTIBLE) and (
```

(continues on next page)

(continued from previous page)

```
                self.random.random() < self.virus_spread_chance
            ):
                agent.state = State.INFECTED

    def try_gain_resistance(self):
        if self.random.random() < self.gain_resistance_chance:
            self.state = State.RESISTANT

    def try_remove_infection(self):
        # Try to remove
        if self.random.random() < self.recovery_chance:
            # Success
            self.state = State.SUSCEPTIBLE
            self.try_gain_resistance()
        else:
            # Failed
            self.state = State.INFECTED

    def check_situation(self):
        if (self.state is State.INFECTED) and (
            self.random.random() < self.virus_check_frequency
        ):
            self.try_remove_infection()

    def step(self):
        if self.state is State.INFECTED:
            self.try_to_infect_neighbors()
        self.check_situation()
```

#### 2.4.3.3.4.16 Model

```
import math

import networkx as nx

import mesa
from mesa import Model
from mesa.discrete_space import CellCollection, Network
from mesa.examples.basic.virus_on_network.agents import State, VirusAgent


def number_state(model, state):
    return sum(1 for a in model.grid.all_cells.agents if a.state is state)


def number_infected(model):
    return number_state(model, State.INFECTED)


def number_susceptible(model):
```

(continues on next page)

(continued from previous page)

```
    return number_state(model, State.SUSCEPTIBLE)


def number_resistant(model):
    return number_state(model, State.RESISTANT)


class VirusOnNetwork(Model):
    """A virus model with some number of agents."""

    def __init__(
        self,
        num_nodes=10,
        avg_node_degree=3,
        initial_outbreak_size=1,
        virus_spread_chance=0.4,
        virus_check_frequency=0.4,
        recovery_chance=0.3,
        gain_resistance_chance=0.5,
        rng=None,
    ):
        super().__init__(rng=rng)
        prob = avg_node_degree / num_nodes
        graph = nx.erdos_renyi_graph(n=num_nodes, p=prob)
        self.grid = Network(graph, capacity=1, random=self.random)

        self.initial_outbreak_size = (
            initial_outbreak_size if initial_outbreak_size <= num_nodes else num_nodes
        )

        self.datacollector = mesa.DataCollector(
            {
                "Infected": number_infected,
                "Susceptible": number_susceptible,
                "Resistant": number_resistant,
                "R over S": self.resistant_susceptible_ratio,
            }
        )

        VirusAgent.create_agents(
            self,
            num_nodes,
            State.SUSCEPTIBLE,
            virus_spread_chance,
            virus_check_frequency,
            recovery_chance,
            gain_resistance_chance,
            list(self.grid.all_cells),
        )

        # Infect some nodes
        infected_nodes = CellCollection(
```

(continues on next page)

(continued from previous page)

```
            self.random.sample(list(self.grid.all_cells), self.initial_outbreak_size),
            random=self.random,
        )
        for a in infected_nodes.agents:
            a.state = State.INFECTED

        self.running = True
        self.datacollector.collect(self)

    def resistant_susceptible_ratio(self):
        try:
            return number_state(self, State.RESISTANT) / number_state(
                self, State.SUSCEPTIBLE
            )
        except ZeroDivisionError:
            return math.inf

    def step(self):
        self.agents.shuffle_do("step")
        # collect data
        self.datacollector.collect(self)
```

#### 2.4.3.3.4.17 App

```
import math

import solara

from mesa.examples.basic.virus_on_network.model import (
    State,
    VirusOnNetwork,
    number_infected,
)
from mesa.visualization import (
    Slider,
    SolaraViz,
    SpaceRenderer,
    make_plot_component,
)
from mesa.visualization.components import AgentPortrayalStyle


def agent_portrayal(agent):
    node_color_dict = {
        State.INFECTED: "red",
        State.SUSCEPTIBLE: "green",
        State.RESISTANT: "gray",
    }
    return AgentPortrayalStyle(color=node_color_dict[agent.state], size=20)
```

(continues on next page)

(continued from previous page)

```
def get_resistant_susceptible_ratio(model):
    ratio = model.resistant_susceptible_ratio()
    ratio_text = r"$\infty$" if ratio is math.inf else f"{ratio:.2f}"
    infected_text = str(number_infected(model))

    return solara.Markdown(
        f"Resistant/Susceptible Ratio: {ratio_text}<br>Infected Remaining: {infected_
↪text}"
    )


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "num_nodes": Slider(
        label="Number of agents",
        value=10,
        min=10,
        max=100,
        step=1,
    ),
    "avg_node_degree": Slider(
        label="Avg Node Degree",
        value=3,
        min=3,
        max=8,
        step=1,
    ),
    "initial_outbreak_size": Slider(
        label="Initial Outbreak Size",
        value=1,
        min=1,
        max=10,
        step=1,
    ),
    "virus_spread_chance": Slider(
        label="Virus Spread Chance",
        value=0.4,
        min=0.0,
        max=1.0,
        step=0.1,
    ),
    "virus_check_frequency": Slider(
        label="Virus Check Frequency",
        value=0.4,
        min=0.0,
        max=1.0,
        step=0.1,
```

(continues on next page)

(continued from previous page)

```
    ),
    "recovery_chance": Slider(
        label="Recovery Chance",
        value=0.3,
        min=0.0,
        max=1.0,
        step=0.1,
    ),
    "gain_resistance_chance": Slider(
        label="Gain Resistance Chance",
        value=0.5,
        min=0.0,
        max=1.0,
        step=0.1,
    ),
}


def post_process_lineplot(chart):
    chart = chart.properties(
        width=400,
        height=400,
    ).configure_legend(
        strokeColor="black",
        fillColor="#ECE9E9",
        orient="right",
        cornerRadius=5,
        padding=10,
        strokeWidth=1,
    )
    return chart


model1 = VirusOnNetwork()
renderer = (
    SpaceRenderer(model1, backend="altair")
    .setup_structure(  # Do this to draw the underlying network and customize it
        node_kwargs={"color": "black", "filled": False, "strokeWidth": 5},
        edge_kwargs={"strokeDash": [6, 1]},
    )
    .setup_agents(agent_portrayal)
)

renderer.render()

# Plot components can also be in altair and support post_process
StatePlot = make_plot_component(
    {"Infected": "red", "Susceptible": "green", "Resistant": "gray"},
    backend="altair",
    post_process=post_process_lineplot,
)
```

(continues on next page)

(continued from previous page)

```
page = SolaraViz(
    model1,
    renderer,
    components=[
        StatePlot,
        get_resistant_susceptible_ratio,
    ],
    model_params=model_params,
    name="Virus Model",
)
page  # noqa
```

#### 2.4.3.3.4.18 Schelling Segregation Model

#### 2.4.3.3.4.19 Summary

The Schelling segregation model is a classic agent-based model, demonstrating how even a mild preference for similar neighbors can lead to a much higher degree of segregation than we would intuitively expect. The model consists of agents on a square grid, where each grid cell can contain at most one agent. Agents come in two colors: orange and blue. They are happy if a certain number of their eight possible neighbors are of the same color, and unhappy otherwise. Unhappy agents will pick a random empty cell to move to each step, until they are happy. The model keeps running until there are no unhappy agents.

By default, the number of similar neighbors the agents need to be happy is set to 3. That means the agents would be perfectly happy with a majority of their neighbors being of a different color (e.g. a Blue agent would be happy with five Orange neighbors and three Blue ones). Despite this, the model consistently leads to a high degree of segregation, with most agents ending up with no neighbors of a different color.

#### 2.4.3.3.4.20 How to Run

Install Mesa with recommended dependencies:

pip install “mesa[rec]”

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.21 Files

- `model.py`: Contains the Schelling model class
- `agents.py`: Contains the Schelling agent class
- `app.py`: Code for the interactive visualization.
- `analysis.ipynb`: Notebook demonstrating how to run experiments and parameter sweeps on the model.

#### 2.4.3.3.4.22 Further Reading

Schelling’s original paper describing the model:

Schelling, Thomas C. Dynamic Models of Segregation. Journal of Mathematical Sociology. 1971, Vol. 1, pp 143-186.

An interactive, browser-based explanation and implementation:

Parable of the Polygons, by Vi Hart and Nicky Case.

#### 2.4.3.3.4.23 Agents

```python
from mesa.discrete_space import CellAgent


class SchellingAgent(CellAgent):
    """Schelling segregation agent."""

    def __init__(
        self, model, cell, agent_type: int, homophily: float = 0.4, radius: int = 1
    ) -> None:
        """Create a new Schelling agent.
        Args:
            model: The model instance the agent belongs to
            agent_type: Indicator for the agent's type (minority=1, majority=0)
            homophily: Minimum number of similar neighbors needed for happiness
            radius: Search radius for checking neighbor similarity
        """
        super().__init__(model)
        self.cell = cell
        self.type = agent_type
        self.homophily = homophily
        self.radius = radius
        self.happy = False

    def assign_state(self) -> None:
        """Determine if agent is happy and move if necessary."""
        neighbors = list(self.cell.get_neighborhood(radius=self.radius).agents)

        # Count similar neighbors
        similar_neighbors = len([n for n in neighbors if n.type == self.type])

        # Calculate the fraction of similar neighbors
        if (valid_neighbors := len(neighbors)) > 0:
            similarity_fraction = similar_neighbors / valid_neighbors
        else:
            # If there are no neighbors, the similarity fraction is 0
            similarity_fraction = 0.0

        if similarity_fraction < self.homophily:
            self.happy = False
        else:
            self.happy = True
            self.model.happy += 1

    def step(self) -> None:
        # Move if unhappy
        if not self.happy:
            self.cell = self.model.grid.select_random_empty_cell()
```

#### 2.4.3.3.4.24 Model

```python
from mesa import Model
from mesa.datacollection import DataCollector
from mesa.discrete_space import OrthogonalMooreGrid
from mesa.examples.basic.schelling.agents import SchellingAgent
from mesa.experimental.scenarios import Scenario


class SchellingScenario(Scenario):
    """Scenario for the Schelling model.

    Args:
        width: Width of the grid
        height: Height of the grid
        density: Initial chance for a cell to be populated (0-1)
        minority_pc: Chance for an agent to be in minority class (0-1)
        homophily: Minimum number of similar neighbors needed for happiness
        radius: Search radius for checking neighbor similarity
        rng: Seed for reproducibility
    """

    height: int = 20
    width: int = 20
    density: float = 0.8
    minority_pc: float = 0.5
    homophily: float = 0.4
    radius: int = 1


class Schelling(Model):
    """Model class for the Schelling segregation model."""

    def __init__(self, scenario: SchellingScenario = SchellingScenario):
        """Create a new Schelling model.

        Args:
            scenario: SchellingScenario containing model parameters.
        """
        super().__init__(scenario=scenario)

        # Model parameters
        self.density = scenario.density
        self.minority_pc = scenario.minority_pc

        # Initialize grid
        self.grid = OrthogonalMooreGrid(
            (scenario.width, scenario.height), random=self.random, capacity=1
        )

        # Track happiness
        self.happy = 0
```

(continues on next page)

(continued from previous page)

```
        # Set up data collection
        self.datacollector = DataCollector(
            model_reporters={
                "happy": "happy",
                "pct_happy": lambda m: (
                    (m.happy / len(m.agents)) * 100 if len(m.agents) > 0 else 0
                ),
                "population": lambda m: len(m.agents),
                "minority_pct": lambda m: (
                    sum(1 for agent in m.agents if agent.type == 1)
                    / len(m.agents)
                    * 100
                    if len(m.agents) > 0
                    else 0
                ),
            },
            agent_reporters={"agent_type": "type"},
        )

        # Create agents and place them on the grid
        for cell in self.grid.all_cells:
            if self.random.random() < self.density:
                agent_type = 1 if self.random.random() < scenario.minority_pc else 0
                SchellingAgent(
                    self,
                    cell,
                    agent_type,
                    homophily=scenario.homophily,
                    radius=scenario.radius,
                )

        # Collect initial state
        self.agents.do("assign_state")
        self.datacollector.collect(self)

    def step(self):
        """Run one step of the model."""
        self.happy = 0  # Reset counter of happy agents
        self.agents.shuffle_do("step")  # Activate all agents in random order
        self.agents.do("assign_state")
        self.datacollector.collect(self)  # Collect data
        self.running = self.happy < len(self.agents)  # Continue until everyone is happy
```

#### 2.4.3.3.4.25 App

```
import os

import solara

from mesa.examples.basic.schelling.model import Schelling, SchellingScenario
```

(continues on next page)

(continued from previous page)

```
from mesa.visualization import (
    Slider,
    SolaraViz,
    SpaceRenderer,
    make_plot_component,
)
from mesa.visualization.components import AgentPortrayalStyle


def get_happy_agents(model):
    """Display a text count of how many happy agents there are."""
    return solara.Markdown(f"**Happy agents: {model.happy}**")


path = os.path.dirname(os.path.abspath(__file__))


def agent_portrayal(agent):
    style = AgentPortrayalStyle(
        x=agent.cell.coordinate[0],
        y=agent.cell.coordinate[1],
        marker=os.path.join(path, "resources", "orange_happy.png"),
        size=75,
    )
    if agent.type == 0:
        if agent.happy:
            style.update(
                (
                    "marker",
                    os.path.join(path, "resources", "blue_happy.png"),
                ),
            )
        else:
            style.update(
                (
                    "marker",
                    os.path.join(path, "resources", "blue_unhappy.png"),
                ),
                ("size", 50),
                ("zorder", 2),
            )
    else:
        if not agent.happy:
            style.update(
                (
                    "marker",
                    os.path.join(path, "resources", "orange_unhappy.png"),
                ),
                ("size", 50),
                ("zorder", 2),
            )
```

(continues on next page)

(continued from previous page)

```
    return style


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "density": Slider("Agent density", 0.8, 0.1, 1.0, 0.1),
    "minority_pc": Slider("Fraction minority", 0.2, 0.0, 1.0, 0.05),
    "homophily": Slider("Homophily", 0.4, 0.0, 1.0, 0.125),
    "width": 20,
    "height": 20,
}

# Note: Models with images as markers are very performance intensive.
model1 = Schelling(scenario=SchellingScenario())
renderer = SpaceRenderer(model1, backend="matplotlib").setup_agents(agent_portrayal)
# Here we use renderer.render() to render the agents and grid in one go.
# This function always renders the grid and then renders the agents or
# property layers on top of it if specified.
renderer.render()

HappyPlot = make_plot_component({"happy": "tab:green"})

page = SolaraViz(
    model1,
    renderer,
    components=[
        HappyPlot,
        get_happy_agents,
    ],
    model_params=model_params,
)
page  # noqa
```

#### 2.4.3.3.4.26 Boltzmann Wealth Model (Tutorial)

#### 2.4.3.3.4.27 Summary

A simple model of agents exchanging wealth. All agents start with the same amount of money. Every step, each agent with one unit of money or more gives one unit of wealth to another random agent. Mesa's Getting Started section walks through the Boltzmann Wealth Model in a series of short introductory tutorials, starting withCreating your First Model.

As the model runs, the distribution of wealth among agents goes from being perfectly uniform (all agents have the same starting wealth), to highly skewed – a small number have high wealth, more have none at all.

#### 2.4.3.3.4.28 How to Run

Install Mesa with recommended dependencies:

pip install “mesa[rec]”

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.29 Files

- `model.py`: Final version of the model.
- `agents.py`: Final version of the agent.
- `app.py`: Code for the interactive visualization.

#### 2.4.3.3.4.30 How the Model Is Structured

This example follows Mesa’s standard separation of concerns:

- `agents.py`: Defines individual agent behavior (WealthAgent with its step method for giving money to other agents)
- `model.py`: Manages the simulation environment, instantiates agents, handles the grid/space, and collects data
- `app.py`: Sets up the visualization components to display the model in a web interface
- `st_app.py`: (Optional) Alternative Streamlit-based visualization

The visualization displays the state of the model but does not influence how agents behave or how the system evolves over time.

#### 2.4.3.3.4.31 Understanding the Output

The **Gini coefficient** shown in the visualization measures wealth inequality, ranging from 0 (perfect equality) to 1 (perfect inequality). It is computed in `model.py` using Mesa’s `DataCollector` at each simulation step.

#### 2.4.3.3.4.32 Optional

An optional visualization is also provided using Streamlit, which is another popular Python library for creating interactive web applications.

To run the Streamlit app, you will need to install the `streamlit` and `altair` libraries:

```
$ pip install streamlit altair
```

Then, you can run the Streamlit app using the following command:

```
$ streamlit run st_app.py
```

#### 2.4.3.3.4.33 Further Reading

This model is drawn from econophysics and presents a statistical mechanics approach to wealth distribution. Some examples of further reading on the topic can be found at:

Milakovic, M. A Statistical Equilibrium Model of Wealth Distribution. February, 2001.

Dragulescu, A and Yakovenko, V. Statistical Mechanics of Money, Income, and Wealth: A Short Survey. November, 2002

#### 2.4.3.3.4.34 Agents

```
from mesa.discrete_space import CellAgent


class MoneyAgent(CellAgent):
    """An agent with fixed initial wealth.

    Each agent starts with 1 unit of wealth and can give 1 unit to other agents
    if they occupy the same cell.

    Attributes:
        wealth (int): The agent's current wealth (starts at 1)
    """

    def __init__(self, model, cell):
        """Create a new agent.

        Args:
            model (Model): The model instance that contains the agent
        """
        super().__init__(model)
        self.cell = cell
        self.wealth = 1

    def move(self):
        """Move the agent to a random neighboring cell."""
        self.cell = self.cell.neighborhood.select_random_cell()

    def give_money(self):
        """Give 1 unit of wealth to a random agent in the same cell."""
        cellmates = [a for a in self.cell.agents if a is not self]

        if cellmates:  # Only give money if there are other agents present
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1

    def step(self):
        """Execute one step for the agent:
        1. Move to a neighboring cell
        2. If wealth > 0, maybe give money to another agent in the same cell
        """
        self.move()
        if self.wealth > 0:
            self.give_money()
```

#### 2.4.3.3.4.35 Model

```
"""
Boltzmann Wealth Model
======================

A simple model of wealth distribution based on the Boltzmann-Gibbs distribution.
Agents move randomly on a grid, giving one unit of wealth to a random neighbor
when they occupy the same cell.
"""

from mesa import Model
from mesa.datacollection import DataCollector
from mesa.discrete_space import OrthogonalMooreGrid
from mesa.examples.basic.boltzmann_wealth_model.agents import MoneyAgent
from mesa.experimental.data_collection import DataRecorder, DatasetConfig
from mesa.experimental.scenarios import Scenario


class BoltzmannScenario(Scenario):
    """Scenario parameters for the Boltzmann Wealth model."""

    n: int = 100
    width: int = 10
    height: int = 10


class BoltzmannWealth(Model):
    """A simple model of an economy where agents exchange currency at random.

    All agents begin with one unit of currency, and each time step agents can give
    a unit of currency to another agent in the same cell. Over time, this produces
    a highly skewed distribution of wealth.

    Attributes:
        num_agents (int): Number of agents in the model
        grid (MultiGrid): The space in which agents move
        running (bool): Whether the model should continue running
        datacollector (DataCollector): Collects and stores model data
    """

    def __init__(self, scenario: BoltzmannScenario = BoltzmannScenario):
        """Initialize the model.

        Args:
            scenario: BoltzmannScenario object containing model parameters.
        """
        super().__init__(scenario=scenario)

        self.num_agents = scenario.n
        self.grid = OrthogonalMooreGrid(
            (scenario.width, scenario.height), random=self.random
        )
```

(continues on next page)

(continued from previous page)

```
        self.recorder = DataRecorder(self)
        (
            self.data_registry.track_agents(self.agents, "agent_data", "wealth").record(
                self.recorder
            )
        )
        (
            self.data_registry.track_model(self, "model_data", "gini").record(
                self.recorder, configuration=DatasetConfig(start_time=4, interval=2)
            )
        )

        # Set up data collection
        self.datacollector = DataCollector(
            model_reporters={"Gini": "gini"},
            agent_reporters={"Wealth": "wealth"},
        )
        MoneyAgent.create_agents(
            self,
            self.num_agents,
            self.random.choices(self.grid.all_cells.cells, k=self.num_agents),
        )

        self.running = True
        self.datacollector.collect(self)

    def step(self):
        self.agents.shuffle_do("step")  # Activate all agents in random order
        self.datacollector.collect(self)  # Collect data

    @property
    def gini(self):
        """Calculate the Gini coefficient for the model's current wealth distribution.

        The Gini coefficient is a measure of inequality in distributions.
        - A Gini of 0 represents complete equality, where all agents have equal wealth.
        - A Gini of 1 represents maximal inequality, where one agent has all wealth.
        """
        agent_wealths = [agent.wealth for agent in self.agents]
        x = sorted(agent_wealths)
        n = self.num_agents
        # Calculate using the standard formula for Gini coefficient
        b = sum(xi * (n - i) for i, xi in enumerate(x)) / (n * sum(x))
        return 1 + (1 / n) - 2 * b
```

#### 2.4.3.3.4.36 App

```
import altair as alt
```

(continues on next page)

(continued from previous page)

```
from mesa.examples.basic.boltzmann_wealth_model.model import (
    BoltzmannScenario,
    BoltzmannWealth,
)
from mesa.mesa_logging import INFO, log_to_stderr
from mesa.visualization import (
    SolaraViz,
    SpaceRenderer,
    make_plot_component,
)
from mesa.visualization.components import AgentPortrayalStyle

log_to_stderr(INFO)


def agent_portrayal(agent):
    return AgentPortrayalStyle(
        color=agent.wealth,
        tooltip={"Agent ID": agent.unique_id, "Wealth": agent.wealth},
    )  # we are using a colormap to translate wealth to color


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "n": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
    "width": 10,
    "height": 10,
}


def post_process(chart):
    """Post-process the Altair chart to add a colorbar legend."""
    chart = chart.encode(
        color=alt.Color(
            "color:N",
            scale=alt.Scale(scheme="viridis", domain=[0, 10]),
            legend=alt.Legend(
                title="Wealth",
                orient="right",
                type="gradient",
                gradientLength=200,
```

(continues on next page)

(continued from previous page)

```
            ),
        ),
    )
    return chart


model = BoltzmannWealth(
    scenario=BoltzmannScenario(
        n=50,
        width=10,
        height=10,
    )
)

# The SpaceRenderer is responsible for drawing the model's space and agents.
# It builds the visualization in layers, first drawing the grid structure,
# and then drawing the agents on top. It uses a specified backend
# (like "altair" or "matplotlib") for creating the plots.

renderer = (
    SpaceRenderer(model, backend="altair")
    .setup_structure(  # To customize the grid appearance.
        grid_color="black", grid_dash=[6, 2], grid_opacity=0.3
    )
    .setup_agents(agent_portrayal, cmap="viridis", vmin=0, vmax=10)
)
renderer.render()

# The post_process function is used to modify the Altair chart after it has been created.
# It can be used to add legends, colorbars, or other visual elements.
renderer.post_process = post_process

# Creates a line plot component from the model's "Gini" datacollector.
GiniPlot = make_plot_component("Gini")

# The SolaraViz page combines the model, renderer, and components into a web interface.
# To run the visualization, save this code as app.py and run `solara run app.py`
page = SolaraViz(
    model,
    renderer,
    components=[GiniPlot],
    model_params=model_params,
    name="Boltzmann Wealth Model",
)
page  # noqa
```

#### 2.4.3.3.4.37 Boids Flockers

#### 2.4.3.3.4.38 Summary

An implementation of Craig Reynolds's Boids flocker model. Agents (simulated birds) try to fly towards the average position of their neighbors and in the same direction as them, while maintaining a minimum distance. This produces flocking behavior.

This model tests Mesa's continuous space feature, and uses numpy arrays to represent vectors.

#### 2.4.3.3.4.39 How to Run

Install Mesa with recommended dependencies:

pip install "mesa[rec]"

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.40 Files

- *model.py*: Contains the Boid Model
- *agents.py*: Contains the Boid agent
- *app.py*: Solara based Visualization code.

#### 2.4.3.3.4.41 Further Reading

The following link can be visited for more information on the boid flockers model: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/modeling-natural-systems/boids.html

#### 2.4.3.3.4.42 Agents

```
"""A Boid (bird-oid) agent for implementing Craig Reynolds's Boids flocking model.

This implementation uses numpy arrays to represent vectors for efficient computation
of flocking behavior.
"""

import numpy as np

from mesa.experimental.continuous_space import ContinuousSpaceAgent


class Boid(ContinuousSpaceAgent):
    """A Boid-style flocker agent.

    The agent follows three behaviors to flock:
        - Cohesion: steering towards neighboring agents
        - Separation: avoiding getting too close to any other agent
        - Alignment: trying to fly in the same direction as neighbors

    Boids have a vision that defines the radius in which they look for their
```

(continues on next page)

(continued from previous page)

```
    neighbors to flock with. Their speed (a scalar) and direction (a vector)
    define their movement. Separation is their desired minimum distance from
    any other Boid.
    """

    def __init__(
        self,
        model,
        space,
        position=(0, 0),
        speed=1,
        direction=(1, 1),
        vision=1,
        separation=1,
        cohere=0.03,
        separate=0.015,
        match=0.05,
    ):
        """Create a new Boid flocker agent.

        Args:
            model: Model instance the agent belongs to
            speed: Distance to move per step
            direction: numpy vector for the Boid's direction of movement
            vision: Radius to look around for nearby Boids
            separation: Minimum distance to maintain from other Boids
            cohere: Relative importance of matching neighbors' positions (default: 0.03)
            separate: Relative importance of avoiding close neighbors (default: 0.015)
            match: Relative importance of matching neighbors' directions (default: 0.05)
        """
        super().__init__(space, model)
        self.position = position
        self.speed = speed
        self.direction = direction
        self.vision = vision
        self.separation = separation
        self.cohere_factor = cohere
        self.separate_factor = separate
        self.match_factor = match
        self.neighbors = []
        self.angle = 0.0  # represents the angle at which the boid is moving

    def step(self):
        """Get the Boid's neighbors, compute the new vector, and move accordingly."""
        neighbors, distances = self.get_neighbors_in_radius(radius=self.vision)
        self.neighbors = [n for n in neighbors if n is not self]

        # If no neighbors, maintain current direction
        if not neighbors:
            self.position += self.direction * self.speed
            return
```

(continues on next page)

(continued from previous page)

```
        delta = self.space.calculate_difference_vector(self.position, agents=neighbors)

        cohere_vector = delta.sum(axis=0) * self.cohere_factor
        separation_vector = (
            -1 * delta[distances < self.separation].sum(axis=0) * self.separate_factor
        )
        match_vector = (
            np.asarray([n.direction for n in neighbors]).sum(axis=0) * self.match_factor
        )

        # Update direction based on the three behaviors
        self.direction += (cohere_vector + separation_vector + match_vector) / len(
            neighbors
        )

        # Normalize direction vector
        self.direction /= np.linalg.norm(self.direction)

        # Move boid
        self.position += self.direction * self.speed
```

#### 2.4.3.3.4.43 Model

```
"""
Boids Flocking Model
====================
A Mesa implementation of Craig Reynolds's Boids flocker model.
Uses numpy arrays to represent vectors.
"""

import os
import sys

sys.path.insert(0, os.path.abspath("../../../.."))


import numpy as np

from mesa import Model
from mesa.examples.basic.boid_flockers.agents import Boid
from mesa.experimental.continuous_space import ContinuousSpace
from mesa.experimental.scenarios import Scenario


class BoidsScenario(Scenario):
    """Scenario parameters for the Boids Flocking model.

    Args:
        population_size: Number of Boids in the simulation (default: 100)
        width: Width of the space (default: 100)
```

(continues on next page)

(continued from previous page)

```
        height: Height of the space (default: 100)
        speed: How fast the Boids move (default: 1)
        vision: How far each Boid can see (default: 10)
        separation: Minimum distance between Boids (default: 2)
        cohere: Weight of cohesion behavior (default: 0.03)
        separate: Weight of separation behavior (default: 0.015)
        match: Weight of alignment behavior (default: 0.05)
        rng: Random rng for reproducibility (default: None)
    """

    population_size: int = 100
    width: int = 100
    height: int = 100
    speed: float = 1.0
    vision: float = 10.0
    separation: float = 2.0
    cohere: float = 0.03
    separate: float = 0.015
    match: float = 0.05


class BoidFlockers(Model):
    """Flocker model class. Handles agent creation, placement and scheduling."""

    def __init__(self, scenario: BoidsScenario = BoidsScenario):
        """Create a new Boids Flocking model.

        Args:
            scenario: BoidsScenario object containing model parameters.
        """
        super().__init__(scenario=scenario)

        self.agent_angles = np.zeros(
            scenario.population_size
        )  # holds the angle representing the direction of all agents at a given step

        # Set up the space
        self.space = ContinuousSpace(
            [[0, scenario.width], [0, scenario.height]],
            torus=True,
            random=self.random,
            n_agents=scenario.population_size,
        )

        # Create and place the Boid agents
        positions = (
            self.rng.random(size=(scenario.population_size, 2)) * self.space.size
        )
        directions = self.rng.uniform(-1, 1, size=(scenario.population_size, 2))
        Boid.create_agents(
            self,
            scenario.population_size,
```

(continues on next page)

(continued from previous page)

```
            self.space,
            position=positions,
            direction=directions,
            cohere=scenario.cohere,
            separate=scenario.separate,
            match=scenario.match,
            speed=scenario.speed,
            vision=scenario.vision,
            separation=scenario.separation,
        )

        # For tracking statistics
        self.average_heading = None
        self.update_average_heading()

    # vectorizing the calculation of angles for all agents
    def calculate_angles(self):
        d1 = np.array([agent.direction[0] for agent in self.agents])
        d2 = np.array([agent.direction[1] for agent in self.agents])
        self.agent_angles = np.degrees(np.arctan2(d1, d2))
        for agent, angle in zip(self.agents, self.agent_angles):
            agent.angle = angle

    def update_average_heading(self):
        """Calculate the average heading (direction) of all Boids."""
        if not self.agents:
            self.average_heading = 0
            return

        headings = np.array([agent.direction for agent in self.agents])
        mean_heading = np.mean(headings, axis=0)
        self.average_heading = np.arctan2(mean_heading[1], mean_heading[0])

    def step(self):
        """Run one step of the model.

        All agents are activated in random order using the AgentSet shuffle_do method.
        """
        self.agents.shuffle_do("step")
        self.update_average_heading()
        self.calculate_angles()
```

#### 2.4.3.3.4.44 App

```
from matplotlib.markers import MarkerStyle

from mesa.examples.basic.boid_flockers.model import BoidFlockers, BoidsScenario
from mesa.visualization import Slider, SolaraViz, SpaceRenderer
from mesa.visualization.components import AgentPortrayalStyle
```

(continues on next page)

(continued from previous page)

```
# Pre-compute markers for different angles (e.g., every 10 degrees)
MARKER_CACHE = {}
for angle in range(0, 360, 10):
    marker = MarkerStyle(10)
    marker._transform = marker.get_transform().rotate_deg(angle)
    MARKER_CACHE[angle] = marker


def boid_draw(agent):
    neighbors = len(agent.neighbors)

    # Calculate the angle
    deg = agent.angle
    # Round to nearest 10 degrees
    rounded_deg = round(deg / 10) * 10 % 360

    # using cached markers to speed things up
    boid_style = AgentPortrayalStyle(
        color="red", size=20, marker=MARKER_CACHE[rounded_deg]
    )
    if neighbors >= 2:
        boid_style.update(("color", "green"), ("marker", MARKER_CACHE[rounded_deg]))
    return boid_style


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "population_size": Slider(
        label="Number of boids",
        value=100,
        min=10,
        max=200,
        step=10,
    ),
    "width": 100,
    "height": 100,
    "speed": Slider(
        label="Speed of Boids",
        value=5,
        min=1,
        max=20,
        step=1,
    ),
    "vision": Slider(
        label="Vision of Bird (radius)",
        value=10,
        min=1,
        max=50,
```

(continues on next page)

(continued from previous page)

```
        step=1,
    ),
    "separation": Slider(
        label="Minimum Separation",
        value=2,
        min=1,
        max=20,
        step=1,
    ),
}

model = BoidFlockers(scenario=BoidsScenario())

# Quickest way to visualize grid along with agents or property layers.
renderer = (
    SpaceRenderer(
        model,
        backend="matplotlib",
    )
    .setup_agents(boid_draw)
    .render()
)

page = SolaraViz(
    model,
    renderer,
    model_params=model_params,
    name="Boid Flocking Model",
)
page  # noqa
```

#### 2.4.3.3.4.45 Sugarscape Constant Growback Model with Traders

#### 2.4.3.3.4.46 Summary

This is Epstein & Axtell's Sugarscape model with Traders, a detailed description is in Chapter four of *Growing Artificial Societies: Social Science from the Bottom Up (1996)*. The model shows an emergent price equilibrium can happen via a decentralized dynamics.

This code generally matches the code in the Complexity Explorer Tutorial, but in `.py` instead of `.ipynb` format.

#### 2.4.3.3.4.47 Agents:

- **Resource**: Resource agents grow back at one unit of sugar and spice per time step up to a specified max amount and can be harvested and traded by the trader agents. (if you do the interactive run, the color will be green if the resource agent has a bigger amount of sugar, or yellow if it has a bigger amount of spice)
- **Traders**: Trader agents have the following attributes: (1) metabolism for sugar, (2) metabolism for spice, (3) vision, (4) initial sugar endowment and (5) initial spice endowment. The traverse the landscape harvesting sugar and spice and trading with other agents. If they run out of sugar or spice then they are removed from the model. (red circle if you do the interactive run)

The trader agents traverse the landscape according to rule **M**:

- Look out as far as vision permits in the four principal lattice directions and identify the unoccupied site(s).
- Considering only unoccupied sites find the nearest position that produces the most welfare using the Cobb-Douglas function.
- Move to the new position
- Collect all the resources (sugar and spice) at that location (Epstein and Axtell, 1996, p. 99)

The traders trade according to rule **T**:

- Agents and potential trade partner compute their marginal rates of substitution (MRS), if they are equal *end*.
- Exchange resources, with spice flowing from the agent with the higher MRS to the agent with the lower MRS and sugar flowing the opposite direction.
- The price (p) is calculated by taking the geometric mean of the agents' MRS.
- If p > 1 then p units of spice are traded for 1 unit of sugar; if p < 1 then 1/p units of sugar for 1 unit of spice
- The trade occurs if it will (a) make both agent better off (increases MRS) and (b) does not cause the agents' MRS to cross over one another otherwise *end*.
- This process then repeats until an *end* condition is met. (Epstein and Axtell, 1996, p. 105)

The model demonstrates several Mesa concepts and features:

- OrthogonalMooreGrid
- Multiple agent types (traders, sugar, spice)
- Dynamically removing agents from the grid and schedule when they die
- Data Collection at the model and agent level
- custom solara matplotlib space visualization

#### 2.4.3.3.4.48 How to Run

Install Mesa with recommended dependencies:

pip install "mesa[rec]"

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.49 Files

- `model.py`: The Sugarscape Constant Growback with Traders model.
- `agents.py`: Defines the Trader agent class and the Resource agent class which contains an amount of sugar and spice.
- `app.py`: Runs a visualization server via Solara (`solara run app.py`).
- `sugar_map.txt`: Provides sugar and spice landscape in raster type format.
- `tests.py`: Has tests to ensure that the model reproduces the results in shown in Growing Artificial Societies.

#### 2.4.3.3.4.50 Further Reading

- Growing Artificial Societies
- Complexity Explorer Sugarscape with Traders Tutorial

#### 2.4.3.3.4.51 Agents

```
import math

from mesa.discrete_space import CellAgent


# Helper function
def get_distance(cell_1, cell_2):
    """
    Calculate the Euclidean distance between two positions

    used in trade.move()
    """

    x1, y1 = cell_1.coordinate
    x2, y2 = cell_2.coordinate
    dx = x1 - x2
    dy = y1 - y2
    return math.sqrt(dx**2 + dy**2)


class Trader(CellAgent):
    """
    Trader:
    - has a metabolism of sugar and spice
    - harvest and trade sugar and spice to survive
    """

    def __init__(
        self,
        model,
        cell,
        sugar=0,
        spice=0,
        metabolism_sugar=0,
        metabolism_spice=0,
        vision=0,
    ):
        super().__init__(model)
        self.cell = cell
        self.sugar = sugar
        self.spice = spice
        self.metabolism_sugar = metabolism_sugar
        self.metabolism_spice = metabolism_spice
        self.vision = vision
        self.prices = []
        self.trade_partners = []
```

(continues on next page)

(continued from previous page)

```
    def get_trader(self, cell):
        """
        helper function used in self.trade_with_neighbors()
        """

        for agent in cell.agents:
            if isinstance(agent, Trader):
                return agent

    def calculate_welfare(self, sugar, spice):
        """
        helper function

        part 2 self.move()
        self.trade()
        """

        # calculate total resources
        m_total = self.metabolism_sugar + self.metabolism_spice
        # Cobb-Douglas functional form; starting on p. 97
        # on Growing Artificial Societies
        return sugar ** (self.metabolism_sugar / m_total) * spice ** (
            self.metabolism_spice / m_total
        )

    def is_starved(self):
        """
        Helper function for self.maybe_die()
        """

        return (self.sugar <= 0) or (self.spice <= 0)

    def calculate_MRS(self, sugar, spice):
        """
        Helper function for
          - self.trade()
          - self.maybe_self_spice()

        Determines what trader agent needs and can give up
        """

        return (spice / self.metabolism_spice) / (sugar / self.metabolism_sugar)

    def calculate_sell_spice_amount(self, price):
        """
        helper function for self.maybe_sell_spice() which is called from
        self.trade()
        """

        if price >= 1:
            sugar = 1
```

(continues on next page)

(continued from previous page)

```
            spice = int(price)
        else:
            sugar = int(1 / price)
            spice = 1
        return sugar, spice

    def sell_spice(self, other, sugar, spice):
        """
        used in self.maybe_sell_spice()

        exchanges sugar and spice between traders
        """

        self.sugar += sugar
        other.sugar -= sugar
        self.spice -= spice
        other.spice += spice

    def maybe_sell_spice(self, other, price, welfare_self, welfare_other):
        """
        helper function for self.trade()
        """

        sugar_exchanged, spice_exchanged = self.calculate_sell_spice_amount(price)

        # Assess new sugar and spice amount - what if change did occur
        self_sugar = self.sugar + sugar_exchanged
        other_sugar = other.sugar - sugar_exchanged
        self_spice = self.spice - spice_exchanged
        other_spice = other.spice + spice_exchanged

        # double check to ensure agents have resources

        if (
            (self_sugar <= 0)
            or (other_sugar <= 0)
            or (self_spice <= 0)
            or (other_spice <= 0)
        ):
            return False

        # trade criteria #1 - are both agents better off?
        both_agents_better_off = (
            welfare_self < self.calculate_welfare(self_sugar, self_spice)
        ) and (welfare_other < other.calculate_welfare(other_sugar, other_spice))

        # trade criteria #2 is their mrs crossing with potential trade
        mrs_not_crossing = self.calculate_MRS(
            self_sugar, self_spice
        ) > other.calculate_MRS(other_sugar, other_spice)

        if not (both_agents_better_off and mrs_not_crossing):
```

(continues on next page)

(continued from previous page)

```
            return False

        # criteria met, execute trade
        self.sell_spice(other, sugar_exchanged, spice_exchanged)

        return True

    def trade(self, other):
        """
        helper function used in trade_with_neighbors()

        other is a trader agent object
        """

        # sanity check to verify code is working as expected
        assert self.sugar > 0
        assert self.spice > 0
        assert other.sugar > 0
        assert other.spice > 0

        # calculate marginal rate of substitution in Growing Artificial Societies p. 101
        mrs_self = self.calculate_MRS(self.sugar, self.spice)
        mrs_other = other.calculate_MRS(other.sugar, other.spice)

        # calculate each agents welfare
        welfare_self = self.calculate_welfare(self.sugar, self.spice)
        welfare_other = other.calculate_welfare(other.sugar, other.spice)

        if math.isclose(mrs_self, mrs_other):
            return

        # calculate price
        price = math.sqrt(mrs_self * mrs_other)

        if mrs_self > mrs_other:
            # self is a sugar buyer, spice seller
            sold = self.maybe_sell_spice(other, price, welfare_self, welfare_other)
            # no trade - criteria not met
            if not sold:
                return
        else:
            # self is a spice buyer, sugar seller
            sold = other.maybe_sell_spice(self, price, welfare_other, welfare_self)
            # no trade - criteria not met
            if not sold:
                return

        # Capture data
        self.prices.append(price)
        self.trade_partners.append(other.unique_id)

        # continue trading
```

(continues on next page)

(continued from previous page)

```
        self.trade(other)

    ##############################################################################
    #                                                                            #
    #                        MAIN TRADE FUNCTIONS                                #
    #                                                                            #
    ##############################################################################

    def move(self):
        """
        Function for trader agent to identify optimal move for each step in 4 parts
        1 - identify all possible moves
        2 - determine which move maximizes welfare
        3 - find closest best option
        4 - move
        """

        # 1. identify all possible moves

        neighboring_cells = [
            cell
            for cell in self.cell.get_neighborhood(self.vision, include_center=True)
            if cell.is_empty
        ]

        if not neighboring_cells:
            # all neighboring cells are occupied
            return

        # 2. determine which move maximizes welfare

        welfares = [
            self.calculate_welfare(
                self.sugar + cell.sugar,
                self.spice + cell.spice,
            )
            for cell in neighboring_cells
        ]

        # 3. Find closest best option

        # find the highest welfare in welfares
        max_welfare = max(welfares)
        # Get cells with the highest welfare
        candidates = [
            cell
            for cell, welfare in zip(neighboring_cells, welfares)
            if math.isclose(welfare, max_welfare)
        ]

        min_dist = min(get_distance(self.cell, cell) for cell in candidates)
```

(continues on next page)

(continued from previous page)

```
        final_candidates = [
            cell
            for cell in candidates
            if math.isclose(get_distance(self.cell, cell), min_dist, rel_tol=1e-02)
        ]

        # 4. Move Agent
        self.cell = self.random.choice(final_candidates)

    def eat(self):
        self.sugar += self.cell.sugar
        self.cell.sugar = 0
        self.sugar -= self.metabolism_sugar

        self.spice += self.cell.spice
        self.cell.spice = 0
        self.spice -= self.metabolism_spice

    def maybe_die(self):
        """
        Function to remove Traders who have consumed all their sugar or spice
        """

        if self.is_starved():
            self.remove()

    def step(self):
        """Agent step method."""
        self.prices = []
        self.trade_partners = []
        self.move()
        self.eat()
        self.maybe_die()

    def trade_with_neighbors(self):
        """
        Function for trader agents to decide who to trade with in three parts

        1- identify neighbors who can trade
        2- trade (2 sessions)
        3- collect data
        """
        # iterate through traders in neighboring cells and trade
        for a in self.cell.get_neighborhood(radius=self.vision).agents:
            self.trade(a)

        return
```

#### 2.4.3.3.4.52 Model

```
from pathlib import Path

import numpy as np

import mesa
from mesa.discrete_space import OrthogonalVonNeumannGrid
from mesa.examples.advanced.sugarscape_g1mt.agents import Trader
from mesa.experimental.scenarios import Scenario


# Helper Functions
def flatten(list_of_lists):
    """
    helper function for model datacollector for trade price
    collapses agent price list into one list
    """
    return [item for sublist in list_of_lists for item in sublist]


def geometric_mean(list_of_prices):
    """
    find the geometric mean of a list of prices
    """
    # protects against an invalid value if no prices
    if len(list_of_prices) == 0:
        return -1
    return np.exp(np.log(list_of_prices).mean())


class SugarScapeScenario(Scenario):
    """Sugarscape scenario class."""

    initial_population: int = 200
    endowment_min: int = 25
    endowment_max: int = 50
    metabolism_min: int = 1
    metabolism_max: int = 5
    vision_min: int = 1
    vision_max: int = 5
    enable_trade: bool = True


class SugarscapeG1mt(mesa.Model):
    """
    Manager class to run Sugarscape with Traders
    """

    def __init__(self, scenario: SugarScapeScenario = SugarScapeScenario):
        super().__init__(scenario=scenario)
        # Initiate width and height of sugarscape
        self.width = 50
```

(continues on next page)

(continued from previous page)

```
        self.height = 50

        # Initiate population attributes
        self.enable_trade = self.scenario.enable_trade
        self.running = True

        # initiate mesa grid class
        self.grid = OrthogonalVonNeumannGrid(
            (self.width, self.height), torus=False, random=self.random
        )
        # initiate datacollector
        self.datacollector = mesa.DataCollector(
            model_reporters={
                "#Traders": lambda m: len(m.agents),
                "Trade Volume": lambda m: sum(len(a.trade_partners) for a in m.agents),
                "Price": lambda m: geometric_mean(
                    flatten([a.prices for a in m.agents])
                ),
            },
            agent_reporters={"Trade Network": "trade_partners"},
        )

        # read in landscape file from supplementary material
        self.sugar_distribution = np.genfromtxt(Path(__file__).parent / "sugar-map.txt")
        self.spice_distribution = np.flip(self.sugar_distribution, 1)

        self.grid.add_property_layer("sugar", self.sugar_distribution.copy())
        self.grid.add_property_layer("spice", self.spice_distribution.copy())

        n = self.scenario.initial_population
        Trader.create_agents(
            self,
            self.scenario.initial_population,
            self.random.choices(self.grid.all_cells.cells, k=n),
            sugar=self.rng.integers(
                self.scenario.endowment_min,
                self.scenario.endowment_max,
                (n,),
                endpoint=True,
            ),
            spice=self.rng.integers(
                self.scenario.endowment_min,
                self.scenario.endowment_max,
                (n,),
                endpoint=True,
            ),
            metabolism_sugar=self.rng.integers(
                self.scenario.metabolism_min,
                self.scenario.metabolism_max,
                (n,),
                endpoint=True,
            ),
```

(continues on next page)

(continued from previous page)

```
            metabolism_spice=self.rng.integers(
                self.scenario.metabolism_min,
                self.scenario.metabolism_max,
                (n,),
                endpoint=True,
            ),
            vision=self.rng.integers(
                self.scenario.vision_min,
                self.scenario.vision_max,
                (n,),
                endpoint=True,
            ),
        )

    def step(self):
        """
        Unique step function that does staged activation of sugar and spice
        and then randomly activates traders
        """
        self.grid.sugar[:] = np.minimum(self.grid.sugar + 1, self.sugar_distribution)
        self.grid.spice[:] = np.minimum(self.grid.spice + 1, self.spice_distribution)

        # step trader agents
        self.agents.shuffle_do("step")
        if self.enable_trade:
            self.agents.shuffle_do("trade_with_neighbors")
        self.datacollector.collect(self)

    def run_model(self, step_count=1000):
        for _ in range(step_count):
            self.step()
```

#### 2.4.3.3.4.53 App

```
from mesa.examples.advanced.sugarscape_g1mt.model import SugarscapeG1mt
from mesa.visualization import Slider, SolaraViz, SpaceRenderer, make_plot_component
from mesa.visualization.components import AgentPortrayalStyle, PropertyLayerStyle


def agent_portrayal(agent):
    return AgentPortrayalStyle(
        x=agent.cell.coordinate[0],
        y=agent.cell.coordinate[1],
        color="red",
        marker="o",
        size=10,
        zorder=1,
    )
```

(continues on next page)

(continued from previous page)

```
def property_layer_portrayal(layer):
    if layer == "sugar":
        return PropertyLayerStyle(
            color="blue", alpha=0.8, colorbar=True, vmin=0, vmax=10
        )
    return PropertyLayerStyle(color="red", alpha=0.8, colorbar=True, vmin=0, vmax=10)


def post_process(chart):
    chart = chart.properties(width=400, height=400)
    return chart


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "width": 50,
    "height": 50,
    # Population parameters
    "initial_population": Slider(
        "Initial Population", value=200, min=50, max=500, step=10
    ),
    # Agent endowment parameters
    "endowment_min": Slider("Min Initial Endowment", value=25, min=5, max=30, step=1),
    "endowment_max": Slider("Max Initial Endowment", value=50, min=30, max=100, step=1),
    # Metabolism parameters
    "metabolism_min": Slider("Min Metabolism", value=1, min=1, max=3, step=1),
    "metabolism_max": Slider("Max Metabolism", value=5, min=3, max=8, step=1),
    # Vision parameters
    "vision_min": Slider("Min Vision", value=1, min=1, max=3, step=1),
    "vision_max": Slider("Max Vision", value=5, min=3, max=8, step=1),
    # Trade parameter
    "enable_trade": {"type": "Checkbox", "value": True, "label": "Enable Trading"},
}

model = SugarscapeG1mt()

# Here, the renderer uses the Altair backend, while the plot components
# use the Matplotlib backend.
# Both can be mixed and matched to enhance the visuals of your model.
renderer = (
    SpaceRenderer(model, backend="altair")
    .setup_agents(agent_portrayal)
    .setup_property_layer(property_layer_portrayal)
)
# Specifically, avoid drawing the grid to hide the grid lines.
renderer.draw_agents()
renderer.draw_property_layer()
```

(continues on next page)

(continued from previous page)

```
renderer.post_process = post_process

# Note: It is advised to switch the pages after pausing the model
# on the Solara dashboard.
page = SolaraViz(
    model,
    renderer,
    components=[
        make_plot_component("#Traders", page=1),
        make_plot_component("Price", page=1),
    ],
    model_params=model_params,
    name="Sugarscape {G1, M, T}",
    play_interval=150,
)
page  # noqa
```

#### 2.4.3.3.4.54 Demographic Prisoner's Dilemma on a Grid

#### 2.4.3.3.4.55 Summary

The Demographic Prisoner's Dilemma is a family of variants on the classic two-player [Prisoner's Dilemma]. The model consists of agents, each with a strategy of either Cooperate or Defect. Each agent's payoff is based on its strategy and the strategies of its spatial neighbors. After each step of the model, the agents adopt the strategy of their neighbor with the highest total score.

The model payoff table is:

| | Cooperate | Defect |
|---|---|---|
| **Cooperate** | 1, 1 | 0, D |
| **Defect** | D, 0 | 0, 0 |

Where *D* is the defection bonus, generally set higher than 1. In these runs, the defection bonus is set to $D=1.6$.

The Demographic Prisoner's Dilemma demonstrates how simple rules can lead to the emergence of widespread cooperation, despite the Defection strategy dominating each individual interaction game. However, it is also interesting for another reason: it is known to be sensitive to the activation regime employed in it.

#### 2.4.3.3.4.56 How to Run

Install Mesa with recommended dependencies:

pip install "mesa[rec]"

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.57 Files

- `agents.py`: contains the agent class.
- `model.py`: contains the model class; the model takes a `activation_order` string as an argument, which determines in which order agents are activated: Sequential, Random or Simultaneous.
- `app.py`: contains the interactive visualization server.
- `Demographic Prisoner's Dilemma Activation Schedule.ipynb`: Jupyter Notebook for running the scheduling experiment. This runs the model three times, one for each activation type, and demonstrates how the activation regime drives the model to different outcomes.

#### 2.4.3.3.4.58 Further Reading

This model is adapted from:

Wilensky, U. (2002). NetLogo PD Basic Evolutionary model. http://ccl.northwestern.edu/netlogo/models/PDBasicEvolutionary. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

The Demographic Prisoner's Dilemma originates from:

Epstein, J. Zones of Cooperation in Demographic Prisoner's Dilemma. 1998.

#### 2.4.3.3.4.59 Agents

```
from mesa.discrete_space import CellAgent


class PDAgent(CellAgent):
    """Agent member of the iterated, spatial prisoner's dilemma model."""

    def __init__(self, model, starting_move=None, cell=None):
        """
        Create a new Prisoner's Dilemma agent.

        Args:
            model: model instance
            starting_move: If provided, determines the agent's initial state:
                           C(ooperating) or D(efecting). Otherwise, random.
        """
        super().__init__(model)
        self.score = 0
        self.cell = cell
        if starting_move:
            self.move = starting_move
        else:
            self.move = self.random.choice(["C", "D"])
        self.next_move = None

    @property
    def is_cooperating(self):
        return self.move == "C"

    def step(self):
        """Get the best neighbor's move, and change own move accordingly
```

(continues on next page)

(continued from previous page)

```
        if better than own score."""

        # neighbors = self.model.grid.get_neighbors(self.pos, True, include_center=True)
        neighbors = [*list(self.cell.neighborhood.agents), self]
        best_neighbor = max(neighbors, key=lambda a: a.score)
        self.next_move = best_neighbor.move

    def advance(self):
        self.move = self.next_move
        self.score += self.increment_score()

    def increment_score(self):
        neighbors = self.cell.neighborhood.agents
        if self.model.activation_order == "Simultaneous":
            moves = [neighbor.next_move for neighbor in neighbors]
        else:
            moves = [neighbor.move for neighbor in neighbors]
        return sum(self.model.payoff[(self.move, move)] for move in moves)
```

#### 2.4.3.3.4.60 Model

```
import typing
from typing import Literal

import mesa
from mesa.discrete_space import OrthogonalMooreGrid
from mesa.examples.advanced.pd_grid.agents import PDAgent
from mesa.experimental.scenarios import Scenario


class PrisonersDilemmaScenario(Scenario):
    """Scenario for Prisoner's Dilemma model."""

    width: int = 50
    height: int = 50
    activation_order: Literal["Sequential", "Random", "Simultaneous"] = "Random"
    payoff: None | dict[tuple[str, str], float] = None
    torus: bool = True


class PdGrid(mesa.Model):
    """Model class for iterated, spatial prisoner's dilemma model."""

    activation_regimes: typing.ClassVar[list[str]] = [
        "Sequential",
        "Random",
        "Simultaneous",
    ]

    # This dictionary holds the payoff for this agent,
```

(continues on next page)

(continued from previous page)

```
    # keyed on: (my_move, other_move)

    payoff: typing.ClassVar[dict[tuple[str, str], float]] = {
        ("C", "C"): 1,
        ("C", "D"): 0,
        ("D", "C"): 1.6,
        ("D", "D"): 0,
    }

    def __init__(
        self,
        scenario: PrisonersDilemmaScenario = PrisonersDilemmaScenario,
    ):
        """
        Create a new Spatial Prisoners' Dilemma Model.

        Args:
            width, height: Grid size. There will be one agent per grid cell.
            activation_order: Can be "Sequential", "Random", or "Simultaneous".
                              Determines the agent activation regime.
            payoffs: (optional) Dictionary of (move, neighbor_move) payoffs.
        """
        super().__init__(scenario=scenario)
        self.activation_order = scenario.activation_order
        self.grid = OrthogonalMooreGrid(
            (scenario.width, scenario.height), torus=scenario.torus, random=self.random
        )

        if scenario.payoff is not None:
            self.payoff = scenario.payoff

        PDAgent.create_agents(
            self, len(self.grid.all_cells.cells), cell=self.grid.all_cells.cells
        )

        self.datacollector = mesa.DataCollector(
            {
                "Cooperating_Agents": lambda m: len(
                    [a for a in m.agents if a.move == "C"]
                )
            }
        )

        self.running = True
        self.datacollector.collect(self)

    def step(self):
        # Activate all agents, based on the activation regime
        match self.activation_order:
            case "Sequential":
                self.agents.do(lambda a: (a.step(), a.advance()))
            case "Random":
```

(continues on next page)

(continued from previous page)

```
                self.agents.shuffle_do(lambda a: (a.step(), a.advance()))
            case "Simultaneous":
                self.agents.do("step").do("advance")
            case _:
                raise ValueError(f"Unknown activation order: {self.activation_order}")

        # Collect data
        self.datacollector.collect(self)
```

#### 2.4.3.3.4.61 App

```
"""
Solara-based visualization for the Spatial Prisoner's Dilemma Model.
"""

from mesa.examples.advanced.pd_grid.model import PdGrid, PrisonersDilemmaScenario
from mesa.visualization import (
    Slider,
    SolaraViz,
    SpaceRenderer,
    make_plot_component,
)
from mesa.visualization.components import AgentPortrayalStyle


def pd_agent_portrayal(agent):
    """
    Portrayal function for rendering PD agents in the visualization.
    """
    return AgentPortrayalStyle(
        color="blue" if agent.move == "C" else "red", marker="s", size=25
    )


# Model parameters
model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "width": Slider("Grid Width", value=50, min=10, max=100, step=1),
    "height": Slider("Grid Height", value=50, min=10, max=100, step=1),
    "activation_order": {
        "type": "Select",
        "value": "Random",
        "values": PdGrid.activation_regimes,
        "label": "Activation Regime",
    },
}
```

(continues on next page)

(continued from previous page)

```
# Create plot for tracking cooperating agents over time
plot_component = make_plot_component("Cooperating_Agents", backend="altair", grid=True)

# Initialize model
initial_model = PdGrid(scenario=PrisonersDilemmaScenario())
# Create grid and agent visualization component using Altair
renderer = (
    SpaceRenderer(initial_model, backend="altair")
    .setup_agents(pd_agent_portrayal)
    .render()
)

# Create visualization with all components
page = SolaraViz(
    model=initial_model,
    renderer=renderer,
    components=[plot_component],
    model_params=model_params,
    name="Spatial Prisoner's Dilemma",
)
page  # noqa B018
```

#### 2.4.3.3.4.62 Epstein Civil Violence Model

#### 2.4.3.3.4.63 Summary

This model is based on Joshua Epstein's simulation of how civil unrest grows and is suppressed. Citizen agents wander the grid randomly, and are endowed with individual risk aversion and hardship levels; there is also a universal regime legitimacy value. There are also Cop agents, who work on behalf of the regime. Cops arrest Citizens who are actively rebelling; Citizens decide whether to rebel based on their hardship and the regime legitimacy, and their perceived probability of arrest.

The model generates mass uprising as self-reinforcing processes: if enough agents are rebelling, the probability of any individual agent being arrested is reduced, making more agents more likely to join the uprising. However, the more rebelling Citizens the Cops arrest, the less likely additional agents become to join.

#### 2.4.3.3.4.64 How to Run

Install Mesa with recommended dependencies:

pip install "mesa[rec]"

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.65 Files

- `model.py`: Core model code.
- `agent.py`: Agent classes.
- `app.py`: Sets up the interactive visualization.
- `Epstein Civil Violence.ipynb`: Jupyter notebook conducting some preliminary analysis of the model.

#### 2.4.3.3.4.66 Further Reading

This model is based adapted from:

Epstein, J. "Modeling civil violence: An agent-based computational approach", Proceedings of the National Academy of Sciences, Vol. 99, Suppl. 3, May 14, 2002

A similar model is also included with NetLogo:

Wilensky, U. (2004). NetLogo Rebellion model. http://ccl.northwestern.edu/netlogo/models/Rebellion. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

#### 2.4.3.3.4.67 Agents

```
import math
from enum import Enum

import mesa


class CitizenState(Enum):
    ACTIVE = 1
    QUIET = 2
    ARRESTED = 3


class EpsteinAgent(mesa.discrete_space.CellAgent):
    def update_neighbors(self):
        """
        Look around and see who my neighbors are
        """
        self.neighborhood = self.cell.get_neighborhood(radius=self.scenario.cop_vision)
        self.neighbors = self.neighborhood.agents
        self.empty_neighbors = [c for c in self.neighborhood if c.is_empty]

    def move(self):
        """Move to a random empty neighboring cell if movement is enabled."""
        if self.scenario.movement and self.empty_neighbors:
            new_pos = self.random.choice(self.empty_neighbors)
            self.move_to(new_pos)


class Citizen(EpsteinAgent):
    """
    A member of the general population, may or may not be in active rebellion.
    Summary of rule: If grievance - risk > threshold, rebel.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        hardship: Agent's 'perceived hardship (i.e., physical or economic
            privation).' Exogenous, drawn from U(0,1).
        risk_aversion: Exogenous, drawn from U(0,1).
        state: Can be CitizenState.QUIET, ACTIVE, or ARRESTED
        jail_sentence: remaining jail time (0 if not in jail)
        grievance: deterministic function of hardship and regime_legitimacy;
            how aggrieved is agent at the regime?
        arrest_probability: agent's assessment of arrest probability, given rebellion

    Notes:
        Parameters accessed via model.scenario: legitimacy, active_threshold, citizen_
↪vision, arrest_prob_constant
    """

    def __init__(self, model):
        """
        Create a new Citizen.

        Args:
            model: the model to which the agent belongs
        """
        super().__init__(model)
        self.hardship = self.random.random()
        self.risk_aversion = self.random.random()
        self.state = CitizenState.QUIET
        self.jail_sentence = 0
        self.grievance = self.hardship * (1 - self.scenario.legitimacy)
        self.arrest_probability = None
        self.neighborhood = []
        self.neighbors = []
        self.empty_neighbors = []

    def step(self):
        """
        Decide whether to activate, then move if applicable.
        """
        if self.jail_sentence:
            self.jail_sentence -= 1
            return  # no other changes or movements if agent is in jail.
        self.update_neighbors()
        self.update_estimated_arrest_probability()

        net_risk = self.risk_aversion * self.arrest_probability
        if (self.grievance - net_risk) > self.scenario.active_threshold:
            self.state = CitizenState.ACTIVE
        else:
            self.state = CitizenState.QUIET

        self.move()
```

(continues on next page)

(continued from previous page)

```
    def update_estimated_arrest_probability(self):
        """
        Based on the ratio of cops to actives in my neighborhood, estimate the
        p(Arrest | I go active).
        """
        cops_in_vision = 0
        actives_in_vision = 1  # citizen counts herself
        for neighbor in self.neighbors:
            if isinstance(neighbor, Cop):
                cops_in_vision += 1
            elif neighbor.state == CitizenState.ACTIVE:
                actives_in_vision += 1

        # there is a body of literature on this equation
        # the round is not in the pnas paper but without it, its impossible to replicate
        # the dynamics shown there.
        self.arrest_probability = 1 - math.exp(
            -1
            * self.scenario.arrest_prob_constant
            * round(cops_in_vision / actives_in_vision)
        )


class Cop(EpsteinAgent):
    """
    A cop for life. No defection.
    Summary of rule: Inspect local vision and arrest a random active agent.

    Notes:
        Parameters accessed via model.scenario: cop_vision, max_jail_term
    """

    def __init__(self, model):
        """
        Create a new Cop.

        Args:
            model: model instance
        """
        super().__init__(model)

    def step(self):
        """
        Inspect local vision and arrest a random active agent. Move if
        applicable.
        """
        self.update_neighbors()
        active_neighbors = []
        for agent in self.neighbors:
            if isinstance(agent, Citizen) and agent.state == CitizenState.ACTIVE:
                active_neighbors.append(agent)
        if active_neighbors:
```

(continues on next page)

(continued from previous page)

```
            arrestee = self.random.choice(active_neighbors)
            arrestee.jail_sentence = self.random.randint(0, self.scenario.max_jail_term)
            arrestee.state = CitizenState.ARRESTED

        self.move()
```

#### 2.4.3.3.4.68 Model

```
from typing import Literal

import mesa
from mesa.discrete_space import OrthogonalMooreGrid, OrthogonalVonNeumannGrid
from mesa.examples.advanced.epstein_civil_violence.agents import (
    Citizen,
    CitizenState,
    Cop,
)
from mesa.experimental.scenarios import Scenario


# Define a typed scenario subclass with defaults
class EpsteinScenario(Scenario):
    """Scenario parameters for Epstein Civil Violence model."""

    citizen_density: float = 0.7
    cop_density: float = 0.074
    citizen_vision: int = 7
    cop_vision: int = 7
    legitimacy: float = 0.8
    max_jail_term: int = 1000
    active_threshold: float = 0.1
    arrest_prob_constant: float = 2.3
    movement: bool = True
    max_iters: int = 1000
    activation_order: Literal["Random", "Sequential"] = "Random"
    grid_type: Literal["Von Neumann", "Moore"] = "Von Neumann"
    rng: int = 42


class EpsteinCivilViolence(mesa.Model):
    """
    Model 1 from "Modeling civil violence: An agent-based computational
    approach," by Joshua Epstein.
    http://www.pnas.org/content/99/suppl_3/7243.full

    Args:
        height: grid height
        width: grid width
        seed: random seed for reproducibility
        scenario: EpsteinScenario object containing model parameters.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(
        self,
        width=40,
        height=40,
        scenario: EpsteinScenario = EpsteinScenario,
    ):
        super().__init__(scenario=scenario)

        match self.scenario.grid_type:
            case "Moore":
                self.grid = OrthogonalMooreGrid(
                    (width, height), capacity=1, torus=True, random=self.random
                )
            case "Von Neumann":
                self.grid = OrthogonalVonNeumannGrid(
                    (width, height), capacity=1, torus=True, random=self.random
                )
            case _:
                raise ValueError(
                    f"Unknown value of grid_type: {self.scenario.grid_type}"
                )

        model_reporters = {
            "active": CitizenState.ACTIVE.name,
            "quiet": CitizenState.QUIET.name,
            "arrested": CitizenState.ARRESTED.name,
        }
        agent_reporters = {
            "jail_sentence": lambda a: getattr(a, "jail_sentence", None),
            "arrest_probability": lambda a: getattr(a, "arrest_probability", None),
        }
        self.datacollector = mesa.DataCollector(
            model_reporters=model_reporters, agent_reporters=agent_reporters
        )
        if self.scenario.cop_density + self.scenario.citizen_density > 1:
            raise ValueError("Cop density + citizen density must be less than 1")

        for cell in self.grid.all_cells:
            klass = self.random.choices(
                [Citizen, Cop, None],
                cum_weights=[
                    self.scenario.citizen_density,
                    self.scenario.citizen_density + self.scenario.cop_density,
                    1,
                ],
            )[0]

            if klass is not None:
                agent = klass(self)  # Either Citizen or Cop
                agent.move_to(cell)
```

(continues on next page)

(continued from previous page)

```
        self.running = True
        self._update_counts()
        self.datacollector.collect(self)

    def step(self):
        """
        Advance the model by one step and collect data.
        """
        match self.scenario.activation_order:
            case "Random":
                self.agents.shuffle_do("step")
            case "Sequential":
                self.agents.do("step")
            case _:
                raise ValueError(
                    f"unknown value of activation_order: {self.scenario.activation_order}
↪"
                )

        self._update_counts()
        self.datacollector.collect(self)

        if self.time > self.scenario.max_iters:
            self.running = False

    def _update_counts(self):
        """Helper function for counting nr. of citizens in given state."""
        counts = self.agents_by_type[Citizen].groupby("state").count()

        for state in CitizenState:
            setattr(self, state.name, counts.get(state, 0))
```

#### 2.4.3.3.4.69 App

```
from mesa.examples.advanced.epstein_civil_violence.agents import (
    Citizen,
    CitizenState,
    Cop,
)
from mesa.examples.advanced.epstein_civil_violence.model import EpsteinCivilViolence
from mesa.visualization import (
    Slider,
    SolaraViz,
    SpaceRenderer,
    make_plot_component,
)
from mesa.visualization.components import AgentPortrayalStyle

COP_COLOR = "#000000"
```

(continues on next page)

(continued from previous page)

```
agent_colors = {
    CitizenState.ACTIVE: "#FE6100",
    CitizenState.QUIET: "#648FFF",
    CitizenState.ARRESTED: "#808080",
}


def citizen_cop_portrayal(agent):
    if agent is None:
        return

    portrayal = AgentPortrayalStyle(size=200)

    if isinstance(agent, Citizen):
        portrayal.update(("color", agent_colors[agent.state]))
    elif isinstance(agent, Cop):
        portrayal.update(("color", COP_COLOR))

    return portrayal


def post_process(ax):
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])
    ax.get_figure().set_size_inches(10, 10)


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "height": 40,
    "width": 40,
    "citizen_density": Slider("Initial Agent Density", 0.7, 0.1, 0.9, 0.1),
    "cop_density": Slider("Initial Cop Density", 0.04, 0.0, 0.1, 0.01),
    "citizen_vision": Slider("Citizen Vision", 7, 1, 10, 1),
    "cop_vision": Slider("Cop Vision", 7, 1, 10, 1),
    "legitimacy": Slider("Government Legitimacy", 0.82, 0.0, 1, 0.01),
    "max_jail_term": Slider("Max Jail Term", 30, 0, 50, 1),
    "activation_order": {
        "type": "Select",
        "value": "Random",
        "values": ["Random", "Sequential"],
        "label": "Activation Order",
    },
    "grid_type": {
        "type": "Select",
        "value": "Von Neumann",
```

(continues on next page)

(continued from previous page)

```
        "values": ["Von Neumann", "Moore"],
        "label": "Grid Type",
    },
}


chart_component = make_plot_component(
    {state.name.lower(): agent_colors[state] for state in CitizenState}
)

epstein_model = EpsteinCivilViolence()
renderer = SpaceRenderer(epstein_model, backend="matplotlib").setup_agents(
    citizen_cop_portrayal
)
# Specifically, avoid drawing the grid to hide the grid lines.
renderer.draw_agents()
renderer.post_process = post_process

page = SolaraViz(
    epstein_model,
    renderer,
    components=[chart_component],
    model_params=model_params,
    name="Epstein Civil Violence",
)
page  # noqa
```

#### 2.4.3.3.4.70 Wolf-Sheep Predation Model

#### 2.4.3.3.4.71 Summary

A simple ecological model, consisting of three agent types: wolves, sheep, and grass. The wolves and the sheep wander around the grid at random. Wolves and sheep both expend energy moving around, and replenish it by eating. Sheep eat grass, and wolves eat sheep if they end up on the same grid cell.

If wolves and sheep have enough energy, they reproduce, creating a new wolf or sheep (in this simplified model, only one parent is needed for reproduction). The grass on each cell regrows at a constant rate. If any wolves and sheep run out of energy, they die.

The model is tests and demonstrates several Mesa concepts and features:

- MultiGrid
- Multiple agent types (wolves, sheep, grass)
- Overlay arbitrary text (wolf's energy) on agent's shapes while drawing on CanvasGrid
- Agents inheriting a behavior (random movement) from an abstract parent
- Writing a model composed of multiple files.
- Dynamically adding and removing agents from the schedule

#### 2.4.3.3.4.72 How to Run

Install Mesa with recommended dependencies:

pip install “mesa[rec]”

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.73 Files

- `wolf_sheep/random_walk.py`: This defines the `RandomWalker` agent, which implements the behavior of moving randomly across a grid, one cell at a time. Both the Wolf and Sheep agents will inherit from it.
- `wolf_sheep/test_random_walk.py`: Defines a simple model and a text-only visualization intended to make sure the RandomWalk class was working as expected. This doesn't actually model anything, but serves as an ad-hoc unit test. To run it, `cd` into the `wolf_sheep` directory and run `python test_random_walk.py`. You'll see a series of ASCII grids, one per model step, with each cell showing a count of the number of agents in it.
- `wolf_sheep/agents.py`: Defines the Wolf, Sheep, and GrassPatch agent classes.
- `wolf_sheep/scheduler.py`: Defines a custom variant on the RandomActivationByType scheduler, where we can define filters for the `get_type_count` function.
- `wolf_sheep/model.py`: Defines the Wolf-Sheep Predation model itself
- `wolf_sheep/server.py`: Sets up the interactive visualization server
- `run.py`: Launches a model visualization server.

#### 2.4.3.3.4.74 Further Reading

This model is closely based on the NetLogo Wolf-Sheep Predation Model:

Wilensky, U. (1997). NetLogo Wolf Sheep Predation model. http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

See also the Lotka–Volterra equations for an example of a classic differential-equation model with similar dynamics.

#### 2.4.3.3.4.75 Agents

```
from mesa.discrete_space import CellAgent, FixedAgent


class Animal(CellAgent):
    """The base animal class."""

    def __init__(
        self, model, energy=8, p_reproduce=0.04, energy_from_food=4, cell=None
    ):
        """Initialize an animal.

        Args:
            model: Model instance
            energy: Starting amount of energy
            p_reproduce: Probability of reproduction (asexual)
```

(continues on next page)

(continued from previous page)

```
            energy_from_food: Energy obtained from 1 unit of food
            cell: Cell in which the animal starts
        """
        super().__init__(model)
        self.energy = energy
        self.p_reproduce = p_reproduce
        self.energy_from_food = energy_from_food
        self.cell = cell

    def spawn_offspring(self):
        """Create offspring by splitting energy and creating new instance."""
        self.energy /= 2
        self.__class__(
            self.model,
            self.energy,
            self.p_reproduce,
            self.energy_from_food,
            self.cell,
        )

    def feed(self):
        """Abstract method to be implemented by subclasses."""

    def step(self):
        """Execute one step of the animal's behavior."""
        # Move to random neighboring cell
        self.move()

        self.energy -= 1

        # Try to feed
        self.feed()

        # Handle death and reproduction
        if self.energy < 0:
            self.remove()
        elif self.random.random() < self.p_reproduce:
            self.spawn_offspring()


class Sheep(Animal):
    """A sheep that walks around, reproduces (asexually) and gets eaten."""

    def feed(self):
        """If possible, eat grass at current location."""
        grass_patch = next(
            (obj for obj in self.cell.agents if isinstance(obj, GrassPatch)), None
        )
        if grass_patch is not None and grass_patch.fully_grown:
            self.energy += self.energy_from_food
            grass_patch.get_eaten()
```

(continues on next page)

(continued from previous page)

```
    def move(self):
        """Move towards a cell where there isn't a wolf, and preferably with grown grass.
↪"""
        cells_without_wolves = []
        cells_with_grass = []

        for cell in self.cell.neighborhood:
            has_wolf = False
            has_grass = False

            for obj in cell.agents:
                # If there's a wolf, we can early exit
                if isinstance(obj, Wolf):
                    has_wolf = True
                    break
                elif isinstance(obj, GrassPatch) and obj.fully_grown:
                    has_grass = True

            # Prefer cells without wolves
            if not has_wolf:
                cells_without_wolves.append(cell)

                # Among safe cells, pick those with grown grass
                if has_grass:
                    cells_with_grass.append(cell)

        # If all surrounding cells have wolves, stay put
        if len(cells_without_wolves) == 0:
            return

        # Move to a cell with grass if available, otherwise move to any safe cell
        target_cells = (
            cells_with_grass if len(cells_with_grass) > 0 else cells_without_wolves
        )
        self.cell = self.random.choice(target_cells)


class Wolf(Animal):
    """A wolf that walks around, reproduces (asexually) and eats sheep."""

    def feed(self):
        """If possible, eat a sheep at current location."""
        sheep = [obj for obj in self.cell.agents if isinstance(obj, Sheep)]
        if sheep:  # If there are any sheep present
            sheep_to_eat = self.random.choice(sheep)
            self.energy += self.energy_from_food
            sheep_to_eat.remove()

    def move(self):
        """Move to a neighboring cell, preferably one with sheep."""
        cells_with_sheep = self.cell.neighborhood.select(
            lambda cell: any(isinstance(obj, Sheep) for obj in cell.agents)
```

(continues on next page)

(continued from previous page)

```
        )
        target_cells = (
            cells_with_sheep if len(cells_with_sheep) > 0 else self.cell.neighborhood
        )
        self.cell = target_cells.select_random_cell()


class GrassPatch(FixedAgent):
    """A patch of grass that grows at a fixed rate and can be eaten by sheep."""

    def __init__(self, model, countdown, grass_regrowth_time, cell):
        """Create a new patch of grass.

        Args:
            model: Model instance
            countdown: Time until grass is fully grown again
            grass_regrowth_time: Time needed to regrow after being eaten
            cell: Cell to which this grass patch belongs
        """
        super().__init__(model)
        self.fully_grown = countdown == 0
        self.grass_regrowth_time = grass_regrowth_time
        self.cell = cell

        # Schedule initial growth if not fully grown
        if not self.fully_grown:
            self.model.schedule_event(self.regrow, after=countdown)

    def regrow(self):
        """Regrow the grass."""
        self.fully_grown = True

    def get_eaten(self):
        """Mark grass as eaten and schedule regrowth."""
        self.fully_grown = False
        self.model.schedule_event(self.regrow, after=self.grass_regrowth_time)
```

#### 2.4.3.3.4.76 Model

```
"""
Wolf-Sheep Predation Model
================================

Replication of the model found in NetLogo:
    Wilensky, U. (1997). NetLogo Wolf Sheep Predation model.
    http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation.
    Center for Connected Learning and Computer-Based Modeling,
    Northwestern University, Evanston, IL.
"""
```

(continues on next page)

(continued from previous page)

```
import math

from mesa import Model
from mesa.datacollection import DataCollector
from mesa.discrete_space import OrthogonalVonNeumannGrid
from mesa.examples.advanced.wolf_sheep.agents import GrassPatch, Sheep, Wolf
from mesa.experimental.scenarios import Scenario


class WolfSheepScenario(Scenario):
    """Scenario parameters for the Wolf-Sheep model.

    Args:
        height: Height of the grid
        width: Width of the grid
        initial_sheep: Number of sheep to start with
        initial_wolves: Number of wolves to start with
        sheep_reproduce: Probability of each sheep reproducing each step
        wolf_reproduce: Probability of each wolf reproducing each step
        wolf_gain_from_food: Energy a wolf gains from eating a sheep
        grass: Whether to have the sheep eat grass for energy
        grass_regrowth_time: How long it takes for a grass patch to regrow
                             once it is eaten
        sheep_gain_from_food: Energy sheep gain from grass, if enabled
        rng: Random rng
    """

    width: int = 20
    height: int = 20
    initial_sheep: int = 100
    initial_wolves: int = 50
    sheep_reproduce: float = 0.04
    wolf_reproduce: float = 0.05
    wolf_gain_from_food: float = 20.0
    grass: bool = True
    grass_regrowth_time: int = 30
    sheep_gain_from_food: float = 4.0


class WolfSheep(Model):
    """Wolf-Sheep Predation Model.

    A model for simulating wolf and sheep (predator-prey) ecosystem modelling.
    """

    description = (
        "A model for simulating wolf and sheep (predator-prey) ecosystem modelling."
    )

    def __init__(self, scenario: WolfSheepScenario = WolfSheepScenario):
        """Create a new Wolf-Sheep model with the given parameters.
```

(continues on next page)

(continued from previous page)

```
        Args:
            scenario: WolfSheepScenario containing model parameters.
        """
        super().__init__(scenario=scenario)

        # Initialize model parameters
        self.height = scenario.height
        self.width = scenario.width
        self.grass = scenario.grass

        # Create grid using experimental cell space
        self.grid = OrthogonalVonNeumannGrid(
            [self.height, self.width],
            torus=True,
            capacity=math.inf,
            random=self.random,
        )

        # Set up data collection
        model_reporters = {
            "Wolves": lambda m: len(m.agents_by_type[Wolf]),
            "Sheep": lambda m: len(m.agents_by_type[Sheep]),
        }
        if self.grass:
            model_reporters["Grass"] = lambda m: len(
                m.agents_by_type[GrassPatch].select(lambda a: a.fully_grown)
            )

        self.datacollector = DataCollector(model_reporters)

        # Create sheep:
        Sheep.create_agents(
            self,
            scenario.initial_sheep,
            energy=self.rng.random((scenario.initial_sheep,))
            * 2
            * scenario.sheep_gain_from_food,
            p_reproduce=scenario.sheep_reproduce,
            energy_from_food=scenario.sheep_gain_from_food,
            cell=self.random.choices(
                self.grid.all_cells.cells, k=scenario.initial_sheep
            ),
        )
        # Create Wolves:
        Wolf.create_agents(
            self,
            scenario.initial_wolves,
            energy=self.rng.random((scenario.initial_wolves,))
            * 2
            * scenario.wolf_gain_from_food,
            p_reproduce=scenario.wolf_reproduce,
            energy_from_food=scenario.wolf_gain_from_food,
```

(continues on next page)

(continued from previous page)

```
            cell=self.random.choices(
                self.grid.all_cells.cells, k=scenario.initial_wolves
            ),
        )

        # Create grass patches if enabled
        if self.grass:
            possibly_fully_grown = [True, False]
            for cell in self.grid:
                fully_grown = self.random.choice(possibly_fully_grown)
                countdown = (
                    0
                    if fully_grown
                    else self.random.randrange(0, scenario.grass_regrowth_time)
                )
                GrassPatch(self, countdown, scenario.grass_regrowth_time, cell)

        # Collect initial data
        self.running = True
        self.datacollector.collect(self)

    def step(self):
        """Execute one step of the model."""
        # First activate all sheep, then all wolves, both in random order
        self.agents_by_type[Sheep].shuffle_do("step")
        self.agents_by_type[Wolf].shuffle_do("step")

        # Collect data
        self.datacollector.collect(self)
```

#### 2.4.3.3.4.77 App

```
from mesa.examples.advanced.wolf_sheep.agents import GrassPatch, Sheep, Wolf
from mesa.examples.advanced.wolf_sheep.model import WolfSheep, WolfSheepScenario
from mesa.visualization import (
    CommandConsole,
    Slider,
    SolaraViz,
    SpaceRenderer,
    make_plot_component,
)
from mesa.visualization.components import AgentPortrayalStyle


def wolf_sheep_portrayal(agent):
    if agent is None:
        return

    portrayal = AgentPortrayalStyle(
        size=50,
```

(continues on next page)

(continued from previous page)

```
        marker="o",
        zorder=2,
    )

    if isinstance(agent, Wolf):
        portrayal.update(("color", "red"))
    elif isinstance(agent, Sheep):
        portrayal.update(("color", "cyan"))
    elif isinstance(agent, GrassPatch):
        if agent.fully_grown:
            portrayal.update(("color", "tab:green"))
        else:
            portrayal.update(("color", "tab:brown"))
        portrayal.update(("marker", "s"), ("size", 125), ("zorder", 1))

    return portrayal


model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "grass": {
        "type": "Select",
        "value": True,
        "values": [True, False],
        "label": "grass regrowth enabled?",
    },
    "grass_regrowth_time": Slider("Grass Regrowth Time", 20, 1, 50),
    "initial_sheep": Slider("Initial Sheep Population", 100, 10, 300),
    "sheep_reproduce": Slider("Sheep Reproduction Rate", 0.04, 0.01, 1.0, 0.01),
    "initial_wolves": Slider("Initial Wolf Population", 10, 5, 100),
    "wolf_reproduce": Slider(
        "Wolf Reproduction Rate",
        0.05,
        0.01,
        1.0,
        0.01,
    ),
    "wolf_gain_from_food": Slider("Wolf Gain From Food Rate", 20, 1, 50),
    "sheep_gain_from_food": Slider("Sheep Gain From Food", 4, 1, 10),
}


def post_process_space(ax):
    ax.set_aspect("equal")
    ax.set_xticks([])
    ax.set_yticks([])
```

(continues on next page)

(continued from previous page)

```
def post_process_lines(ax):
    ax.legend(loc="center left", bbox_to_anchor=(1, 0.9))


lineplot_component = make_plot_component(
    {"Wolves": "tab:orange", "Sheep": "tab:cyan", "Grass": "tab:green"},
    post_process=post_process_lines,
)

model = WolfSheep(scenario=WolfSheepScenario(grass=True))

renderer = SpaceRenderer(
    model,
    backend="matplotlib",
).setup_agents(wolf_sheep_portrayal)
renderer.post_process = post_process_space
renderer.draw_agents()

page = SolaraViz(
    model,
    renderer,
    components=[lineplot_component, CommandConsole],
    model_params=model_params,
    name="Wolf Sheep",
)
page  # noqa
```

#### 2.4.3.3.4.78 Alliance Formation Model (Meta-Agent Example)

#### 2.4.3.3.4.79 Summary

This model demonstrates Mesa's meta agent capability.

**Overview of meta agent:** Complex systems often have multiple levels of components. A city is not a single entity, but it is made of districts,neighborhoods, buildings, and people. A forest comprises an ecosystem of trees, plants, animals, and microorganisms. An organization is not one entity, but is made of departments, sub-departments, and people. A person is not a single entity, but it is made of micro biomes, organs and cells.

This reality is the motivation for meta-agents. It allows users to represent these multiple levels, where each level can have agents with sub-agents.

This model demonstrates Mesa's ability to dynamically create new classes of agents that are composed of existing agents. These meta-agents inherits functions and attributes from their sub-agents and users can specify new functionality or attributes they want the meta agent to have. For example, if a user is doing a factory simulation with autonomous systems, each major component of that system can be a sub-agent of the overall robot agent. Or, if someone is doing a simulation of an organization, individuals can be part of different organizational units that are working for some purpose.

To provide a simple demonstration of this capability is an alliance formation model.

In this simulation n agents are created, who have two attributes (1) power and (2) preference. Each attribute is a number between 0 and 1 over a gaussian distribution. Agents then randomly select other agents and use the bilateral shapley value to determine if they should form an alliance. If the expected utility support an alliances, the agent creates a meta-agent. Subsequent steps may add agents to the meta-agent, create new instances of similar hierarchy, or create

a new hierarchy level where meta-agents form an alliance of meta-agents. In this visualization of this model a new meta-agent hierarchy will be a larger node and a new color.

In MetaAgents current configuration, agents being part of multiple meta-agents is not supported.

If you would like to see an example of explicit meta-agent formation see the warehouse model in the Mesa example's repository

#### 2.4.3.3.4.80 Files

- `model.py`: Contains creation of agents, the network and management of agent execution.
- `agents.py`: Contains logic for forming alliances and creation of new agents
- `app.py`: Contains the code for the interactive Solara visualization.

#### 2.4.3.3.4.81 Further Reading

The full tutorial describing how the model is built can be found at: https://mesa.readthedocs.io/en/latest/tutorials/intro_tutorial.html

An example of the bilateral shapley value in another model: Techno-Social Energy Infrastructure Siting: Sustainable Energy Modeling Programming (SEMPro)

#### 2.4.3.3.4.82 How to Run

Install Mesa with recommended dependencies:

pip install "mesa[rec]"

Then run the example:

solara run app.py

Open the displayed local URL in your browser.

#### 2.4.3.3.4.83 Agents

```
import mesa


class AllianceAgent(mesa.Agent):
    """
    Agent has three attributes power (float), position (float) and level (int)

    """

    def __init__(self, model, power, position, level=0):
        super().__init__(model)
        self.power = power
        self.position = position
        self.level = level

    """
    For this demo model agent only need attributes.

    More complex models could have functions that define agent behavior.
    """
```

#### 2.4.3.3.4.84 Model

```
import networkx as nx
import numpy as np

import mesa
from mesa import Agent
from mesa.examples.advanced.alliance_formation.agents import AllianceAgent
from mesa.experimental.meta_agents.meta_agent import (
    create_meta_agent,
    find_combinations,
)
from mesa.experimental.scenarios import Scenario


class AllianceScenario(Scenario):
    """Scenario for the Alliance model."""

    n: int = 50
    mean: float = 0.5
    std_dev: float = 0.1


class MultiLevelAllianceModel(mesa.Model):
    """
    Model for simulating multi-level alliances among agents.
    """

    def __init__(self, scenario: AllianceScenario = AllianceScenario):
        """
        Initialize the model.

        Args:
            n (int): Number of agents.
            mean (float): Mean value for normal distribution.
            std_dev (float): Standard deviation for normal distribution.
            rng (int): Random rng.
        """
        super().__init__(scenario=scenario)
        self.network = nx.Graph()  # Initialize the network
        self.datacollector = mesa.DataCollector(model_reporters={"Network": "network"})

        # Create Agents
        power = self.rng.normal(scenario.mean, scenario.std_dev, scenario.n)
        power = np.clip(power, 0, 1)
        position = self.rng.normal(scenario.mean, scenario.std_dev, scenario.n)
        position = np.clip(position, 0, 1)
        AllianceAgent.create_agents(self, scenario.n, power, position)
        agent_ids = [
            (agent.unique_id, {"size": 300, "level": 0}) for agent in self.agents
        ]
        self.network.add_nodes_from(agent_ids)
```

(continues on next page)

(continued from previous page)

```
    def add_link(self, meta_agent, agents):
        """
        Add links between a meta agent and its constituent agents in the network.

        Args:
            meta_agent (MetaAgent): The meta agent.
            agents (list): List of agents.
        """
        for agent in agents:
            self.network.add_edge(meta_agent.unique_id, agent.unique_id)

    def calculate_shapley_value(self, agents):
        """
        Calculate the Shapley value of the two agents.

        Args:
            agents: Pair of agents.

        Returns:
            tuple: Potential utility, new position, and level.
        """
        agent_0, agent_1 = agents

        new_position = 1 - (
            max(agent_0.position, agent_1.position)
            - min(agent_0.position, agent_1.position)
        )
        potential_utility = (agent_0.power + agent_1.power) * 1.2 * new_position

        value_0 = 0.5 * agent_0.power + 0.5 * (potential_utility - agent_1.power)
        value_1 = 0.5 * agent_1.power + 0.5 * (potential_utility - agent_0.power)

        if value_0 > agent_0.power and value_1 > agent_1.power:
            if agent_0.level > agent_1.level:
                level = agent_0.level
            elif agent_0.level == agent_1.level:
                level = agent_0.level + 1
            else:
                level = agent_1.level

            return potential_utility, new_position, level

    def only_best_combination(self, combinations):
        """
        Filter to keep only the best combination for each agent.

        Args:
            combinations (list): List of combinations.

        Returns:
            dict: Unique combinations.
        """
```

(continues on next page)

(continued from previous page)

```
        best = {}
        # Determine best option for EACH agent
        for group, value in combinations:
            agent_ids = sorted(a.unique_id for a in group)
            # Deal with all possibilities
            if (
                agent_ids[0] not in best and agent_ids[1] not in best
            ):  # if neither in add both
                best[agent_ids[0]] = [group, value, agent_ids]
                best[agent_ids[1]] = [group, value, agent_ids]
            elif (
                agent_ids[0] in best and agent_ids[1] in best
            ):  # if both in, see if both would be trading up
                if (
                    value[0] > best[agent_ids[0]][1][0]
                    and value[0] > best[agent_ids[1]][1][0]
                ):
                    # Remove the old alliances
                    del best[best[agent_ids[0]][2][1]]
                    del best[best[agent_ids[1]][2][0]]
                    # Add the new alliance
                    best[agent_ids[0]] = [group, value, agent_ids]
                    best[agent_ids[1]] = [group, value, agent_ids]
            elif (
                agent_ids[0] in best
            ):  # if only agent_ids[0] in, see if it would be trading up
                if value[0] > best[agent_ids[0]][1][0]:
                    # Remove the old alliance for agent_ids[0]
                    del best[best[agent_ids[0]][2][1]]
                    # Add the new alliance
                    best[agent_ids[0]] = [group, value, agent_ids]
                    best[agent_ids[1]] = [group, value, agent_ids]
            elif (
                agent_ids[1] in best
            ):  # if only agent_ids[1] in, see if it would be trading up
                if value[0] > best[agent_ids[1]][1][0]:
                    # Remove the old alliance for agent_ids[1]
                    del best[best[agent_ids[1]][2][0]]
                    # Add the new alliance
                    best[agent_ids[0]] = [group, value, agent_ids]
                    best[agent_ids[1]] = [group, value, agent_ids]

        # Create a unique dictionary of the best combinations
        unique_combinations = {}
        for group, value, agents_nums in best.values():
            unique_combinations[tuple(agents_nums)] = [group, value]

        return unique_combinations.values()

    def step(self):
        """
        Execute one step of the model.
```

(continues on next page)

(continued from previous page)

```
        """
        # Get all other agents of the same type
        agent_types = list(self.agents_by_type.keys())

        for agent_type in agent_types:
            similar_agents = self.agents_by_type[agent_type]

            # Find the best combinations using find_combinations
            if (
                len(similar_agents) > 1
            ):  # only form alliances if there are more than 1 agent
                combinations = find_combinations(
                    self,
                    similar_agents,
                    size=2,
                    evaluation_func=self.calculate_shapley_value,
                    filter_func=self.only_best_combination,
                )

                for alliance, attributes in combinations:
                    class_name = f"MetaAgentLevel{attributes[2]}"
                    meta = create_meta_agent(
                        self,
                        class_name,
                        alliance,
                        Agent,
                        meta_attributes={
                            "level": attributes[2],
                            "power": attributes[0],
                            "position": attributes[1],
                        },
                    )

                    # Update the network if a new meta agent instance created
                    if meta:
                        self.network.add_node(
                            meta.unique_id,
                            size=(meta.level + 1) * 300,
                            level=meta.level,
                        )
                        self.add_link(meta, meta.agents)
```

#### 2.4.3.3.4.85 App

```
import matplotlib.pyplot as plt
import networkx as nx
import solara
from matplotlib.figure import Figure

from mesa.examples.advanced.alliance_formation.model import (
```

(continues on next page)

(continued from previous page)

```
    AllianceScenario,
    MultiLevelAllianceModel,
)
from mesa.visualization import SolaraViz
from mesa.visualization.utils import update_counter

model_params = {
    "rng": {
        "type": "InputText",
        "value": 42,
        "label": "Random Seed",
    },
    "n": {
        "type": "SliderInt",
        "value": 50,
        "label": "Number of agents:",
        "min": 10,
        "max": 100,
        "step": 1,
    },
}

# Create visualization elements. The visualization elements are solara components
# that receive the model instance as a "prop" and display it in a certain way.
# Under the hood these are just classes that receive the model instance.
# You can also author your own visualization elements, which can also be functions
# that receive the model instance and return a valid solara component.


@solara.component
def plot_network(model):
    update_counter.get()
    g = model.network
    pos = nx.fruchterman_reingold_layout(g)
    fig = Figure()
    ax = fig.subplots()
    labels = {agent.unique_id: agent.unique_id for agent in model.agents}
    node_sizes = [g.nodes[node]["size"] for node in g.nodes]
    node_colors = [g.nodes[node]["size"] for node in g.nodes()]

    nx.draw(
        g,
        pos,
        node_size=node_sizes,
        node_color=node_colors,
        cmap=plt.cm.coolwarm,
        labels=labels,
        ax=ax,
    )

    solara.FigureMatplotlib(fig)
```

(continues on next page)

(continued from previous page)

```
# Create initial model instance
model = MultiLevelAllianceModel(scenario=AllianceScenario(n=50, rng=42))

# Create the SolaraViz page. This will automatically create a server and display the
# visualization elements in a web browser.
# Display it using the following command in the example directory:
# solara run app.py
# It will automatically update and display any changes made to this file
page = SolaraViz(
    model,
    components=[plot_network],
    model_params=model_params,
    name="Alliance Formation Model",
)
page  # noqa
```

### 2.4.4 Mesa Migration guide

This guide contains breaking changes between major Mesa versions and how to resolve them.

Non-breaking changes aren't included, for those see our Release history.

#### 2.4.4.1 Mesa 3.5.0

##### 2.4.4.1.1 Event scheduling and time advancement

Mesa 3.5 introduces public methods for event scheduling and time advancement directly on `Model`, replacing the need for `Simulator` classes.

###### 2.4.4.1.1.1 Time-based advancement replaces step loops

```
# Old
for _ in range(10):
    model.step()

# New
model.run_for(10)    # Functionally equivalent for standard ABMs
model.run_until(10)  # You can now also run until a specific time
```

`run_for(1)` produces identical results to `step()` for traditional models.

###### 2.4.4.1.1.2 One-off event scheduling

```
# Schedule a single event at or after a specific time
model.schedule_event(callback, at=50.0)    # Absolute time
model.schedule_event(callback, after=5.0)  # Relative time

# Cancel if needed
event = model.schedule_event(callback, at=100.0)
event.cancel()
```

#### 2.4.4.1.1.3 Recurring event scheduling

```
from mesa.time import Schedule

# Schedule an event every 10 time units
model.schedule_recurring(func, Schedule(interval=10))  # By default, starting after 1␣
↪interval (t=10 in this case)
model.schedule_recurring(func, Schedule(interval=10, start=0))  # Start immediately or␣
↪at any other time

# Save the event and stop it when needed
gen = model.schedule_recurring(func, Schedule(interval=5.0))
gen.stop()

# Limit executions
model.schedule_recurring(func, Schedule(interval=1.0, count=10))
```

#### 2.4.4.1.1.4 Replacing Simulator classes

The experimental Simulator classes are now also deprecated.

```
# Old - ABMSimulator
from mesa.experimental.devs.simulator import ABMSimulator
simulator = ABMSimulator()
simulator.setup(model)
simulator.run_for(100)

# New
model.run_for(100)
```

```
# Old - DEVSimulator
from mesa.experimental.devs.simulator import DEVSimulator
simulator = DEVSimulator()
simulator.setup(model)
simulator.schedule_event_absolute(callback, time=10.5)
simulator.run_for(50)

# New
model.schedule_event(callback, at=10.5)
model.run_for(50)
```

Mesa 3.5 doesn't introduce any breaking changes. Mesa 4 will clean up many deprecated components and thus will break unmodified models.

- Ref: Discussion #2921, PR #3266

#### 2.4.4.1.2 AgentSet sequence behavior

The Sequence behavior (indexing and slicing) on `AgentSet` is deprecated and will be removed in Mesa 4.0. Use the new `to_list()` method instead.

```
# Old (deprecated)
first_agent = model.agents[0]
```

(continues on next page)

(continued from previous page)

```
some_agents = model.agents[1:5]
last_agent = model.agents[-1]

# New
first_agent = model.agents.to_list()[0]
some_agents = model.agents.to_list()[1:5]
last_agent = model.agents.to_list()[-1]
```

For multiple list operations, convert once and reuse:

```
agent_list = model.agents.to_list()
first = agent_list[0]
last = agent_list[-1]
subset = agent_list[2:8]
```

- Ref: PR #3208

#### 2.4.4.2 Mesa 3.4.0

##### 2.4.4.2.1 batch run

`batch_run` has been updated to offer explicit control over the random seeds that are used to run multiple replications of a given experiment. For this a new keyword argument, `rng` has been added and `iterations` will issue a `DeprecationWarning`. The new `rng` keyword argument takes a valid value for seeding or a list of valid values. If you want to run multiple iterations/replications of a given experiment, you need to pass the required seeds explicitly.

Below is a simple example of the new recommended usage of `batch_run`. Note how we first create 5 random integers which we then use as seed values for the new `rng` keyword argument.

```
import numpy as np
import sys

# let's create 5 random integers
rng = np.random.default_rng(42)
rng_values = rng.integers(0, sys.maxsize, size=(5,))

results = mesa.batch_run(
    MoneyModel,
    parameters=params,
    rng=rng_values.tolist(), # we pass the 5 seed values to rng
    max_steps=100,
    number_processes=1,
    data_collection_period=1,
    display_progress=True,
)
```

#### 2.4.4.3 Mesa 3.3.0

Mesa 3.3.0 is a visualization upgrade introducing a new and improved API, full support for both `altair` and `matplotlib` backends, and resolving several recurring issues from previous versions. For full details on how to visualize your model, refer to the Mesa Documentation.

*This guide is a work in progress. The development of it is tracked in Issue #2233.*

#### 2.4.4.3.1 Defining Portrayal Components

Previously, `agent_portrayal` returned a dictionary. Now, it returns an instance of a dedicated portrayal component called `AgentPortrayalStyle`.

```
# Old
def agent_portrayal(agent):
    return {
        "color": "white" if agent.state == 0 else "black",
        "marker": "s",
        "size": "30"
    }

# New
def agent_portrayal(agent):
    return AgentPortrayalStyle(
        color="white" if agent.state == 0 else "black",
        marker="s",
        size=30,
    )
```

Similarly, `propertylayer_portrayal` has moved from a dictionary-based interface to a function-based one, following the same pattern as `agent_portrayal`. It now returns a `PropertyLayerStyle` instance instead of a dictionary.

```
# Old
propertylayer_portrayal = {
    "sugar": {
        "colormap": "pastel1",
        "alpha": 0.75,
        "colorbar": True,
        "vmin": 0,
        "vmax": 10,
    }
}

# New
def propertylayer_portrayal(layer):
    if layer.name == "sugar":
        return PropertyLayerStyle(
            color="pastel1", alpha=0.75, colorbar=True, vmin=0, vmax=10
        )
```

- Ref: PR #2786

#### 2.4.4.3.2 Passing portrayal arguments to draw methods

Passing portrayal arguments directly to `draw_agents()` and `draw_propertylayer()` is deprecated. Use the `setup_agents()` and `setup_propertylayer()` methods before calling the draw methods.

```
# Old
renderer.draw_agents(agent_portrayal=agent_portrayal)
renderer.draw_propertylayer(propertylayer_portrayal)

# New
```

(continues on next page)

(continued from previous page)

```
renderer.setup_agents(agent_portrayal).draw_agents()
renderer.setup_propertylayer(propertylayer_portrayal).draw_propertylayer()
```

This change allows for better method chaining and separates the configuration phase from the rendering phase.

- Ref: PR #2893

#### 2.4.4.3.3 Default Space Visualization

While the visualization methods from Mesa versions before 3.3.0 still work, version 3.3.0 introduces `SpaceRenderer`, which changes how space visualizations are rendered. Check out the updated Mesa documentation for guidance on upgrading your model's visualization using `SpaceRenderer`.

A basic example of how `SpaceRenderer` works:

```
# Old
from mesa.visualization import SolaraViz, make_space_component

SolaraViz(model, components=[make_space_component(agent_portrayal)])

# New
from mesa.visualization import SolaraViz, SpaceRenderer

renderer = SpaceRenderer(model, backend="matplotlib").render(
    agent_portrayal=agent_portrayal,
    ...
)

SolaraViz(
    model,
    renderer,
    components=[],
    ...
)
```

- Ref: PR #2803, PR #2810

#### 2.4.4.3.4 Page Tab View

Version 3.3.0 adds support for defining pages for different plot components. Learn more in the Mesa documentation.

In short, you can define multiple pages using the following syntax:

```
from mesa.visualization import SolaraViz, make_plot_component

SolaraViz(
    model,
    components=[
        make_plot_component("foo", page=1),
        make_plot_component("bar", "baz", page=2),
    ],
)
```

- Ref: PR #2827

#### 2.4.4.4 Mesa 3.0

Mesa 3.0 introduces significant changes to core functionalities, including agent and model initialization, scheduling, and visualization. The guide below outlines these changes and provides instructions for migrating your existing Mesa projects to version 3.0.

##### 2.4.4.4.1 Upgrade strategy

We recommend the following upgrade strategy:

- Update to the latest Mesa 2.x release (`mesa<3`).
- Update to the latest Mesa 3.0.x release (`mesa<3.1`).
- Update to the latest Mesa 3.x release (`mesa<4`).

With each update, resolve all errors and warnings, before updating to the next one.

##### 2.4.4.4.2 Reserved and private variables

###### 2.4.4.4.2.1 Reserved variables

Currently, we have reserved the following variables:

- Model: `agents`, `current_id`, `random`, `running`, `steps`, `time`.
- Agent: `unique_id`, `model`.

You can use (read) any reserved variable, but Mesa may update them automatically and rely on them, so modify/update at your own risk.

###### 2.4.4.4.2.2 Private variables

Any variables starting with an underscore (_) are considered private and for Mesa's internal use. We might use any of those. Modifying or overwriting any private variable is at your own risk.

- Ref: Discussion #2230, PR #2225

##### 2.4.4.4.3 Removal of `mesa.flat` namespace

The `mesa.flat` namespace is removed. Use the full namespace for your imports.

- Ref: PR #2091

##### 2.4.4.4.4 Mandatory Model initialization with `super().__init__()`

In Mesa 3.0, it is now mandatory to call `super().__init__()` when initializing your model class. This ensures that all necessary Mesa model variables are correctly set up and agents are properly added to the model. If you want to control the seed of the random number generator, you have to pass this as a keyword argument to super as shown below.

Make sure all your model classes explicitly call `super().__init__()` in their `__init__` method:

```
class MyModel(mesa.Model):
    def __init__(self, some_arg_I_need, seed=None, some_kwarg_I_need=True):
        super().__init__(seed=seed)  # Calling super is now required, passing seed is␣
↪highly recommended
        # Your model initialization code here
        # this code uses some_arg_I_need and my_init_kwarg
```

This change ensures that all Mesa models are properly initialized, which is crucial for:

- Correctly adding agents to the model
- Setting up other essential Mesa model variables
- Maintaining consistency across all models

If you forget to call `super().__init__()`, you'll now see this error:

```
RuntimeError: The Mesa Model class was not initialized. You must explicitly initialize␣
↪the Model by calling super().__init__() on initialization.
```

- Ref: PR #2218, PR #1928, Mesa-examples PR #83

#### 2.4.4.4.5 Automatic assignment of `unique_id` to Agents

In Mesa 3.0, `unique_id` for agents is now automatically assigned, simplifying agent creation and ensuring unique IDs across all agents in a model.

1. Remove `unique_id` from agent initialization:

```
# Old
agent = MyAgent(unique_id=unique_id, model=self, ...)
agent = MyAgent(unique_id, self, ...)
agent = MyAgent(self.next_id(), self, ...)

# New
agent = MyAgent(model=self, ...)
agent = MyAgent(self, ...)
```

2. Remove `unique_id` from Agent super() call:

```
# Old
class MyAgent(Agent):
    def __init__(self, unique_id, model, ...):
        super().__init__(unique_id, model)

# New
class MyAgent(Agent):
    def __init__(self, model, ...):
        super().__init__(model)
```

3. Important notes:
    - `unique_id` is now automatically assigned relative to a Model instance and starts from 1
    - `Model.next_id()` is removed
    - If you previously used custom `unique_id` values, store that information in a separate attribute

- Ref: PR #2226, PR #2260, Mesa-examples PR #194, Issue #2213

#### 2.4.4.4.6 AgentSet and `Model.agents`

In Mesa 3.0, the Model class internally manages agents using several data structures:

- `self._agents`: A dictionary containing hard references to all agents, indexed by their `unique_id`.
- `self._agents_by_type`: A dictionary of AgentSets, organizing agents by their type.
- `self._all_agents`: An AgentSet containing all agents in the model.

These internal structures are used to efficiently manage and access agents. Users should interact with agents through the public `model.agents` property, which returns the `self._all_agents` AgentSet.

##### 2.4.4.4.6.1 `Model.agents`

- Attempting to set `model.agents` now raises an `AttributeError` instead of a warning. This attribute is reserved for internal use by Mesa.
- If you were previously setting `model.agents` in your code, you must update it to use a different attribute name for custom agent storage.

For example, replace:

```
model.agents = my_custom_agents
```

With:

```
model.custom_agents = my_custom_agents
```

#### 2.4.4.4.7 Time and schedulers

##### 2.4.4.4.7.1 Automatic increase of the `steps` counter

The `steps` counter is now automatically increased. With each call to `Model.steps()` it's increased by 1, at the beginning of the step.

You can access it by `Model.steps`, and it's internally in the datacollector, batchrunner and the visualisation.

- Ref: PR #2223, Mesa-examples PR #161

##### 2.4.4.4.7.2 Removal of `Model._time` and rename `._steps`

- `Model._time` is removed. You can define your own time variable if needed.
- `Model._steps` steps is renamed to `Model.steps`.

##### 2.4.4.4.7.3 Removal of `Model._advance_time()`

- The `Model._advance_time()` method is removed. This now happens automatically.

##### 2.4.4.4.7.4 Replacing Schedulers with AgentSet functionality

The whole Time module in Mesa is deprecated and will be removed in Mesa 3.1. All schedulers should be replaced with AgentSet functionality and the internal `Model.steps` counter. This allows much more flexibility in how to activate Agents and makes it explicit what's done exactly.

Here's how to replace each scheduler:

##### 2.4.4.4.7.5 BaseScheduler

Replace:

```
self.schedule = BaseScheduler(self)
self.schedule.step()
```

With:

```
self.agents.do("step")
```

#### 2.4.4.4.7.6 RandomActivation

Replace:

```
self.schedule = RandomActivation(self)
self.schedule.step()
```

With:

```
self.agents.shuffle_do("step")
```

#### 2.4.4.4.7.7 SimultaneousActivation

Replace:

```
self.schedule = SimultaneousActivation(self)
self.schedule.step()
```

With:

```
self.agents.do("step")
self.agents.do("advance")
```

#### 2.4.4.4.7.8 StagedActivation

Replace:

```
self.schedule = StagedActivation(self, ["stage1", "stage2", "stage3"])
self.schedule.step()
```

With:

```
for stage in ["stage1", "stage2", "stage3"]:
    self.agents.do(stage)
```

If you were using the `shuffle` and/or `shuffle_between_stages` options:

```
stages = ["stage1", "stage2", "stage3"]
if shuffle:
    self.random.shuffle(stages)
for stage in stages:
    if shuffle_between_stages:
        self.agents.shuffle_do(stage)
    else:
        self.agents.do(stage)
```

#### 2.4.4.4.7.9 RandomActivationByType

Replace:

```
self.schedule = RandomActivationByType(self)
self.schedule.step()
```

With:

```
for agent_class in self.agent_types:
    self.agents_by_type[agent_class].shuffle_do("step")
```

#### 2.4.4.4.7.10 Replacing `step_type`

The `RandomActivationByType` scheduler had a `step_type` method that allowed stepping only agents of a specific type. To replicate this functionality using AgentSet:

Replace:

```
self.schedule.step_type(AgentType)
```

With:

```
self.agents_by_type[AgentType].shuffle_do("step")
```

#### 2.4.4.4.7.11 General Notes

1. The `Model.steps` counter is now automatically incremented. You don't need to manage it manually.
2. If you were using `self.schedule.agents`, replace it with `self.agents`.
3. If you were using `self.schedule.get_agent_count()`, replace it with `len(self.agents)`.
4. If you were using `self.schedule.agents_by_type`, replace it with `self.agents_by_type`.
5. Agents are now automatically added to or removed from the model's `AgentSet` (`model.agents`) when they are created or deleted, eliminating the need to manually call `self.schedule.add()` or `self.schedule.remove()`.
    - However, you still need to explicitly remove the Agent itself by using `Agent.remove()`. Typically, this means:
        - Replace `self.schedule.remove(agent)` with `agent.remove()` in the Model.
        - Replace `self.model.schedule.remove(self)` with `self.remove()` within the Agent.

From now on you're now not bound by 5 distinct schedulers, but can mix and match any combination of AgentSet methods (`do`, `shuffle`, `select`, etc.) to get the desired Agent activation.

Ref: Original discussion #1912, decision discussion #2231, example updates #183 and #201, PR #2306

#### 2.4.4.4.8 Visualisation

Mesa has adopted a new API for our frontend. If you already migrated to the experimental new SolaraViz you can still use the import from mesa.experimental. Otherwise here is a list of things you need to change.

> **Note:** SolaraViz is experimental and still in active development for Mesa 3.0. While we attempt to minimize them, there might be API breaking changes between Mesa 3.0 and 3.1. There won't be breaking changes between Mesa 3.0.x patch releases.

#### 2.4.4.4.8.1 Model Initialization

Previously SolaraViz was initialized by providing a `model_cls` and a `model_params`. This has changed to expect a model instance `model`. You can still provide (user-settable) `model_params`, but only if users should be able to change them. It is now also possible to pass in a "reactive model" by first calling `model = solara.reactive(model)`. This is useful for notebook environments. It allows you to pass the model to the SolaraViz Module, but continue to use the model. For example calling `model.value.step()` (notice the extra .value) will automatically update the plots. This currently only automatically works for the step method, you can force visualization updates by calling `model.value.force_update()`.

#### 2.4.4.4.9 Model Initialization with Keyword Arguments

With the introduction of SolaraViz in Mesa 3.0, models are now instantiated using `**model_parameters.value`. This means all inputs for initializing a new model must be keyword arguments. Ensure your model's `__init__` method accepts keyword arguments matching the keys in `model_params`.

```
class MyModel(mesa.Model):
    def __init__(self, n_agents=10, seed=None):
        super().__init__(seed=seed)
        # Initialize the model with N agents
```

#### 2.4.4.4.9.1 Default space visualization

Previously we included a default space drawer that you could configure with an `agent_portrayal` function. You now have to explicitly create a space drawer with the `agent_portrayal` function

```
# old
from mesa.experimental import SolaraViz

SolaraViz(model_cls, model_params, agent_portrayal=agent_portrayal)

# new
from mesa.visualization import SolaraViz, make_space_component

SolaraViz(model, components=[make_space_component(agent_portrayal)])
```

#### 2.4.4.4.9.2 Plotting "measures"

"Measure" plots also need to be made explicit here. Previously, measure could either be 1) A function that receives a model and returns a solara component or 2) A string or list of string of variables that are collected by the datacollector and are to be plotted as a line plot. 1) still works, but you can pass that function to "components" directly. 2) needs to explicitly call the `make_plot_measure()`function.

```
# old
from mesa.experimental import SolaraViz


def make_plot(model):
    ...


SolaraViz(model_cls, model_params, measures=[make_plot, "foo", ["bar", "baz"]])
```

(continues on next page)

(continued from previous page)

```
# new
from mesa.visualization import SolaraViz, make_plot_component

SolaraViz(model, components=[make_plot, make_plot_component("foo"), make_plot_component(
↪"bar", "baz")])
```

#### 2.4.4.4.9.3 Plotting text

To plot model-dependent text the experimental SolaraViz provided a `make_text` function that wraps another functions that receives the model and turns its string return value into a solara text component. Again, this other function can now be passed directly to the new SolaraViz components array. It is okay if your function just returns a string.

```
# old
from mesa.experimental import SolaraViz, make_text

def show_steps(model):
    return f"Steps: {model.steps}"

SolaraViz(model_cls, model_params, measures=make_text(show_steps))

# new
from mesa.visualisation import SolaraViz

def show_steps(model):
    return f"Steps: {model.steps}"

SolaraViz(model, components=[show_steps])
```

#### 2.4.4.4.10 Other changes

##### 2.4.4.4.10.1 Removal of Model.initialize_data_collector

The `initialize_data_collector` in the Model class is removed. In the Model class, replace:

Replace:

```
self.initialize_data_collector(...)
```

With:

```
self.datacollector = DataCollector(...)
```

- Ref: PR #2327, Mesa-examples PR #208)

### 2.4.5 APIs

#### 2.4.5.1 Model

The model class for Mesa framework.

Core Objects: Model

**class Model**(**args: Any*, *rng: RNGLike | SeedLike | None = None*, *scenario: S | type[S] = <class 'mesa.experimental.scenarios.scenario.Scenario'>*, ***kwargs: Any*)

Base class for models in the Mesa ABM library.

This class serves as a foundational structure for creating agent-based models. It includes the basic attributes and methods necessary for initializing and running a simulation model.

**Type Parameters:**
A: The agent type used in this model S: The scenario type used in this model

`running`

A boolean indicating if the model should continue running.

`steps`

the number of times *model.step()* has been called.

`time`

the current simulation time.

`random`

a seeded python.random number generator.

`rng`

a seeded numpy.random.Generator

`scenario`

the scenario instance containing model parameters

### Notes

Model.agents returns the AgentSet containing all agents registered with the model. Changing the content of the AgentSet directly can result in strange behavior. If you want change the composition of this AgentSet, ensure you operate on a copy.

Create a new model.

Overload this method with the actual code to initialize the model. Always start with super().__init__() to initialize the model object properly.

**Parameters**

- `args` – arguments to pass onto super
- `rng` – Seed for the random number generator. Accepts any value accepted by numpy.random.default_rng(). Ignored if a Scenario instance is passed; used to instantiate the scenario when a Scenario class is passed.
- `scenario` – A Scenario instance or subclass to use for this model. If a class is passed it is instantiated with rng. If an instance is passed, rng must not be set.
- `kwargs` – keyword arguments to pass onto super

### Notes

Pass either rng or a Scenario instance, not both. Passing rng alongside a Scenario class is valid — rng is forwarded to the class constructor.

`property scenario: S`

Return scenario instance.

**property agents: _HardKeyAgentSet[A]**

Provides a _HardKeyAgentSet of all agents in the model, combining agents from all types.

**Returns**
The agent set containing all agents with strong references.

**Return type**
_HardKeyAgentSet

> **Warning**
>
> This returns the actual internal _HardKeyAgentSet used by Mesa for agent registration and tracking. It uses strong references to prevent premature garbage collection and reduce performance overhead caused by weak reference management.
>
> **Do not modify this AgentSet directly** (e.g., by adding or removing agents manually). Direct modifications can break the model's agent tracking system and cause unexpected behavior. Instead:
>
> - Use `Agent()` to create new agents (automatically registers them)
> - Use `agent.remove()` to remove agents (automatically deregisters them)
> - For read-only operations or transformations, work on a copy: `model.agents.copy()`

### Notes

This is Mesa's core agent registration system. All agents created via `Agent.__init__` are automatically registered here.

**property agent_types: list[type]**

Return a list of all unique agent types registered with the model.

**property agents_by_type: dict[type[A], _HardKeyAgentSet[A]]**

A dictionary where keys are agent types and values are the corresponding _HardKeyAgentSets.

**Returns**
Dictionary mapping agent types to their AgentSets.

**Return type**
dict[type[A], _HardKeyAgentSet[A]]

> **Warning**
>
> Each AgentSet in this dictionary is a _HardKeyAgentSet with strong references, forming part of Mesa's core agent registration system.
>
> **Do not modify these AgentSets directly**. Direct modifications can break agent tracking and cause unexpected behavior. Instead:
>
> - Use `Agent()` to create new agents (automatically registers them)
> - Use `agent.remove()` to remove agents (automatically deregisters them)
> - For read-only operations, work on copies: `model.agents_by_type[AgentType].copy()`

### Notes

This is part of Mesa's core agent registration system. All agents are automatically registered in the appropriate type-specific AgentSet when created via `Agent.__init__`.

**register_agent**(*agent: A*)

Register the agent with the model.

**Parameters**
**agent** – The agent to register.

### Notes

This method is called automatically by `Agent.__init__`, so there is no need to use this if you are subclassing Agent and calling its super in the `__init__` method.

**deregister_agent**(*agent: A*)

Deregister the agent with the model.

**Parameters**
**agent** – The agent to deregister.

### Notes

This method is called automatically by `Agent.remove`

**run_model**() → None

Run the model until the end condition is reached.

Overload as needed.

**step**() → None

A single step. Fill in here.

**batch**()

Return a context manager that batches signals.

Signals emitted during the batch are buffered and aggregated on exit. Nested batches merge into the outer batch; only the outermost dispatches.

**Note**

Computed properties may return stale cached values during the batch. They will be updated when aggregated signals are dispatched on exit.

**classmethod clear_all_class_subscriptions**(*name: str | _AllSentinel | Iterable[str]*)

Clears all class-level subscriptions for the observable <name>.

if name is ALL, all subscriptions are removed

**Parameters**
**name** – name of the Observable to unsubscribe for all signal types

**clear_all_subscriptions**(*name: str | _AllSentinel | Iterable[str]*)

Clears all instance-level subscriptions for the observable <name>.

if name is ALL, all subscriptions are removed

**Parameters**
**name** – name of the Observable to unsubscribe for all signal types

**notify**(*observable: str*, *signal_type: str | SignalType*, ***kwargs*)

Emit a signal.

**Parameters**

- **observable** – the public name of the observable emitting the signal
- **signal_type** – the type of signal to emit
- **kwargs** – additional keyword arguments to include in the signal

**observe**(*observable_name: str | _AllSentinel | Iterable[str]*, *signal_type: str | SignalType | _AllSentinel | Iterable[str | SignalType]*, *handler: Callable*)

Subscribe to the Observable <name> for signal_type.

**Parameters**

- **observable_name** – name of the Observable to subscribe to
- **signal_type** – the type of signal on the Observable to subscribe to
- **handler** – the handler to call

**Raises**

- **ValueError** – if the Observable <name> is not registered or if the Observable
- **does not emit the given signal_type** –

**classmethod observe_class**(*observable_name: str | _AllSentinel | Iterable[str]*, *signal_type: str | SignalType | _AllSentinel | Iterable[str | SignalType]*, *handler: Callable*)

Subscribe at the class level to the Observable <name> for signal_type.

All instances of this class will trigger the handler when they emit this signal. Handlers are stored as weak references to prevent memory leaks during experiments.

**Parameters**

- **observable_name** – name of the Observable to subscribe to
- **signal_type** – the type of signal on the Observable to subscribe to
- **handler** – the handler to call

**remove_all_agents**()

Remove all agents from the model.

### Notes

This method calls agent.remove for all agents in the model. If you need to remove agents from e.g., a SingleGrid, you can either explicitly implement your own agent.remove method or clean this up near where you are calling this method.

**suppress**()

Return a context manager that suppresses all signals.

No signals are emitted, buffered, or dispatched during suppression.

> **Note**
>
> Computed properties may become permanently stale because their triggering signals are dropped entirely.

**unobserve**(*observable_name: str | _AllSentinel | Iterable[str]*, *signal_type: str | SignalType | _AllSentinel | Iterable[str | SignalType]*, *handler: Callable*)

Unsubscribe to the Observable <name> for signal_type.

**Parameters**

- **observable_name** – name of the Observable to unsubscribe from
- **signal_type** – the type of signal on the Observable to unsubscribe to
- **handler** – the handler that is unsubscribing

**classmethod unobserve_class**(*observable_name: str | _AllSentinel | Iterable[str]*, *signal_type: str | SignalType | _AllSentinel | Iterable[str | SignalType]*, *handler: Callable*)

Unsubscribe at the class level to the Observable <name> for signal_type.

**Parameters**

- **observable_name** – name of the Observable to unsubscribe from
- **signal_type** – the type of signal on the Observable to unsubscribe to
- **handler** – the handler that is unsubscribing

**schedule_event**(*function: Callable*, ** (Keyword-only parameters separator (PEP 3102))*, *at: float | None = None*, *after: float | None = None*, *priority: Priority = Priority.DEFAULT*) → *Event*

Schedule a one-off event.

**Parameters**

- **function** – The callable to execute
- **at** – Absolute time to execute (mutually exclusive with after)
- **after** – Relative time from now to execute (mutually exclusive with at)
- **priority** – Priority level for the event

**Returns**
The scheduled Event (can be used to cancel)

**Raises**

- **ValueError** – If both or neither of at/after are specified
- **ValueError** – If both or neither of at/after are specified, or if the scheduled time is in the past.

**schedule_recurring**(*function: Callable*, *schedule: Schedule*, *priority: Priority = Priority.DEFAULT*) → *EventGenerator*

Schedule a recurring event based on a Schedule.

**Parameters**

- **function** – The callable to execute repeatedly
- **schedule** – The Schedule defining when events occur
- **priority** – Priority level for generated events

**Returns**
The EventGenerator (can be used to stop)

**Raises**
**ValueError** – If the schedule start time is in the past.

**run_for**(*duration: float | int*) → None

Run the model for the specified duration.

**Parameters**
**duration** – Time units to advance

**run_until**(*end_time: float | int*) → None

Run the model until the specified time.

**Parameters**
**end_time** – Absolute time to run until

If model.time is larger than end_time, the method returns directly.

#### 2.4.5.2 Agent

Agent related classes.

Core Objects: Agent.

**class Agent**(*model: M*, **args*, ***kwargs*)

Base class for a model agent in Mesa.

**model**

A reference to the model instance.

**Type**
Model

**unique_id**

A unique identifier for this agent.

**Type**
int

##### Notes

Agents must be hashable to be used in an AgentSet. In Python 3, defining *__eq__* without *__hash__* makes an object unhashable, which will break AgentSet usage. unique_id is unique relative to a model instance and starts from 1

Create a new agent.

**Parameters**

- **model** (*Model*) – The model instance in which the agent exists.
- **args** – Passed on to super.
- **kwargs** – Passed on to super.

##### Notes

to make proper use of python's super, in each class remove the arguments and keyword arguments you need and pass on the rest to super

**remove**() → None

Remove and delete the agent from the model.

If the agent is currently performing an action, the action's scheduled completion event is cancelled silently. The action's on_interrupt() callback is NOT fired, because the agent is being destroyed — not making a behavioral decision. The action moves to no defined end state; it is simply abandoned.

If your action holds external resources (e.g., a Resource slot, a reservation, a lock), override Agent.remove() and call self.cancel_action() before super().remove() to ensure on_interrupt() fires and cleanup logic runs:

**def remove(self):**
self.cancel_action() # Fires on_interrupt for cleanup super().remove()

### Notes

This is a deliberate design choice. The default silent cleanup is safe and avoids callbacks touching agent state during teardown. Models that need cleanup should opt in explicitly.

`step()` → None

A single step of the agent.

`classmethod create_agents`(*model: Model*, *n: int*, **args*, ***kwargs*) → *AgentSet*[T]

Create N agents.

**Parameters**

- **`model`** – the model to which the agents belong
- **`args`** – arguments to pass onto agent instances each arg is either a single object or a sequence of length n
- **`n`** – the number of agents to create
- **`kwargs`** – keyword arguments to pass onto agent instances each keyword arg is either a single object or a sequence of length n

**Returns**
AgentSet containing the agents created.

`classmethod from_dataframe`(*model: Model*, *df: pd.DataFrame*, ***kwargs*) → *AgentSet*[T]

Create agents from a pandas DataFrame.

Each row of the DataFrame represents one agent. The DataFrame columns are mapped to the agent's constructor as keyword arguments. Additional keyword arguments (***kwargs*) can be used to set constant attributes for all agents.

**Parameters**

- **`model`** – The model instance.
- **`df`** – The pandas DataFrame. Each row represents an agent.
- **`**kwargs`** – Constant values to pass to every agent's constructor. Only non-sequence data is allowed in kwargs to avoid ambiguity with DataFrame columns.

**Returns**
AgentSet containing the agents created.

> **Note**
>
> If you need to pass variable data or sequences, add them as columns to the DataFrame before calling this method.

`property random:` `Random`

Return a seeded stdlib rng.

property rng: Generator

Return a seeded np.random rng.

property scenario

Return the scenario associated with the model.

start_action(*action: Action*) → Action

Start performing an action.

The action must be in PENDING or INTERRUPTED state and the agent must not be currently performing another action.

**Parameters**
**action** – The Action to perform. Must have been created with this agent as its agent.

**Returns**
The started Action.

**Raises**
**ValueError** – If the agent is already performing an action, or if the action doesn't belong to this agent.

interrupt_for(*new_action: Action*) → bool

Interrupt the current action and start a new one.

If there is no current action, simply starts the new one. If the current action is non-interruptible, returns False and does nothing.

**Parameters**
**new_action** – The Action to perform instead.

**Returns**
True if the new action was started (either no current action, or the current one was successfully interrupted). False if the current action is non-interruptible.

cancel_action() → bool

Cancel the current action, ignoring interruptible flag.

Calls on_interrupt with partial progress. Returns False only if there is no current action.

**Returns**
True if an action was cancelled, False if idle.

property is_busy: bool

Whether the agent is currently performing an action.

#### 2.4.5.3 AgentSet

AgentSet related classes.

Core Objects: AgentSet, AbstractAgentSet, _HardKeyAgentSet, GroupBy.

class AbstractAgentSet

An abstract base collection class that represents an ordered set of agents within an agent-based model (ABM).

This class defines the minimal interface that all AgentSet implementations must follow. Subclasses are free to override methods with optimized implementations based on their storage mechanism (weak references vs strong references).

**model**

The ABM model instance to which this AbstractAgentSet belongs.

**Type**
Model

**select**(*filter_func: Callable[[A], bool] | None = None*, *at_most: int | float = inf*, *inplace: bool = False*, *agent_type: type[A] | None = None*) → *AbstractAgentSet*[A]

Select a subset of agents from the AbstractAgentSet based on a filter function and/or quantity limit.

**Parameters**

- **filter_func** (*Callable[[Agent], bool], optional*) – A function that takes an Agent and returns True if the agent should be included in the result. Defaults to None, meaning no filtering is applied.
- **at_most** (*int | float, optional*) – The maximum amount of agents to select. Defaults to infinity. - If an integer, at most the first number of matching agents are selected. - If a float between 0 and 1, at most that fraction of original the agents are selected.
- **inplace** (*bool, optional*) – If True, modifies the current AbstractAgentSet; otherwise, returns a new AbstractAgentSet. Defaults to False.
- **agent_type** (*type[Agent], optional*) – The class type of the agents to select. Defaults to None, meaning no type filtering is applied.

**Returns**
A new AbstractAgentSet containing the selected agents, unless inplace is True, in which case the current AbstractAgentSet is updated.

**Return type**
*AbstractAgentSet*

### Notes

- at_most just return the first n or fraction of agents. To take a random sample, shuffle() beforehand.
- at_most is an upper limit. When specifying other criteria, the number of agents returned can be smaller.

**agg**(*attribute: str*, *func: Callable | Iterable[Callable]*) → Any | list[Any]

Aggregate an attribute of all agents in the AgentSet using one or more functions.

**Parameters**

- **attribute** (*str*) – The name of the attribute to aggregate.
- **func** (*Callable | Iterable[Callable]*) –
  - If Callable: A single function to apply to the attribute values (e.g., min, max, sum, np.mean)
  - If Iterable: Multiple functions to apply to the attribute values

**Returns**
Result of applying the function(s) to the attribute values.

**Return type**
Any | [Any, ...]

### Examples

# Single function avg_energy = model.agents.agg("energy", np.mean)

# Multiple functions min_wealth, max_wealth, total_wealth = model.agents.agg("wealth", [min, max, sum])

**get**(*attr_names: str*, *handle_missing: Literal['error', 'default'] = 'error'*, *default_value: Any = None*) → list[Any]

**get**(*attr_names: list[str]*, *handle_missing: Literal['error', 'default'] = 'error'*, *default_value: Any = None*) → list[list[Any]]

Retrieve the specified attribute(s) from each agent in the AgentSet.

**Parameters**

- **attr_names** (*str | list[str]*) – The name(s) of the attribute(s) to retrieve from each agent.
- **handle_missing** (*str, optional*) – How to handle missing attributes. Can be: - 'error' (default): raises an AttributeError if attribute is missing. - 'default': returns the specified default_value.
- **default_value** (*Any, optional*) – The default value to return if 'handle_missing' is set to 'default' and the agent does not have the attribute.

**Returns**

A list with the attribute value for each agent if attr_names is a str. list[list[Any]]: A list with a lists of attribute values for each agent if attr_names is a list of str.

**Return type**

list[Any]

**Raises**

- **AttributeError** – If 'handle_missing' is 'error' and the agent does not have the specified attribute(s).
- **ValueError** – If an unknown 'handle_missing' option is provided.

**set**(*attr_name: str*, *value: Any*) → *AgentSet*[A]

Set a specified attribute to a given value for all agents in the AgentSet.

**Parameters**

- **attr_name** (*str*) – The name of the attribute to set.
- **value** (*Any*) – The value to set the attribute to.

**Returns**

The AgentSet instance itself, after setting the attribute.

**Return type**

*AgentSet*

**to_list**() → list[A]

Convert the AbstractAgentSet to a list.

**Returns**

A list containing all agents in the AbstractAgentSet.

**Return type**

list[Agent]

### Notes

This method provides an explicit way to convert the AgentSet to a list. It is the recommended approach when list operations (indexing, slicing) are needed, as direct sequence operations on AgentSet are deprecated and will be removed in Mesa 4.0.

**abstractmethod add**(*agent: A*)

Add an agent to the AbstractAgentSet.

**Parameters**
**agent** (*Agent*) – The agent to add to the set.

> **Note**
>
> This method is an implementation of the abstract method from MutableSet.

**abstractmethod discard**(*agent: A*)

Remove an agent from the AbstractAgentSet if it exists.

This method does not raise an error if the agent is not present.

**Parameters**
**agent** (*Agent*) – The agent to remove from the set.

> **Note**
>
> This method is an implementation of the abstract method from MutableSet.

**abstractmethod remove**(*agent: A*)

Remove an agent from the AbstractAgentSet.

**Raises**
**An Exception if the agent is not present.** –

**Parameters**
**agent** (*Agent*) – The agent to remove from the set.

> **Note**
>
> This method is an implementation of the abstract method from MutableSet.

**groupby**(*by: Callable | str*, *result_type: Literal['agentset', 'list'] = 'agentset'*) → *GroupBy*

Group agents by the specified attribute or return from the callable.

**Parameters**

- **by** (*Callable, str*) – used to determine what to group agents by
  - if by is a callable, it will be called for each agent and the return is used for grouping
  - if by is a str, it should refer to an attribute on the agent and the value of this attribute will be used for grouping
- **result_type** (*str, optional*) – The datatype for the resulting groups {“agentset”, “list”}

**Returns**
GroupBy

### Notes

There might be performance benefits to using *result_type='list'* if you don't need the advanced functionality of an AbstractAgentSet.

**abstractmethod shuffle**(*inplace: bool = False*) → *AbstractAgentSet*[A]

Randomly shuffle the order of agents in the AbstractAgentSet.

**abstractmethod sort**(*key: Callable[[A], Any] | str*, *ascending: bool = False*, *inplace: bool = False*) → *AbstractAgentSet*[A]

Sort the agents in the AbstractAgentSet based on a specified attribute or custom function.

**abstractmethod do**(*method: str | Callable*, **args*, ***kwargs*) → *AbstractAgentSet*[A]

Invoke a method or function on each agent in the AbstractAgentSet.

**abstractmethod shuffle_do**(*method: str | Callable*, **args*, ***kwargs*) → *AbstractAgentSet*[A]

Shuffle the agents in the AbstractAgentSet and then invoke a method or function on each agent.

**abstractmethod map**(*method: str | Callable*, **args*, ***kwargs*) → list[Any]

Invoke a method or function on each agent in the AbstractAgentSet and return the results.

**clear**()

This is slow (creates N new iterators!) but effective.

**isdisjoint**(*other*)

Return True if two sets have a null intersection.

**pop**()

Return the popped value. Raise KeyError if empty.

**class AgentSet**(*agents: Iterable[A]*, *random: Random | None = None*)

A collection class that represents an ordered set of agents using weak references.

This implementation uses weak references to agents, allowing for efficient management of agent lifecycles without preventing garbage collection.

**random**

The random number generator for this agent set.

**Type**
Random

### Notes

The AgentSet maintains weak references to agents, which means that agents not referenced elsewhere in the program may be automatically removed from the AgentSet. This is the default implementation for most use cases where automatic cleanup is desired.

Performance-critical methods are optimized to work directly with weak references, avoiding the overhead of creating strong references during iteration.

Initialize the AgentSet with weak references to agents.

**Parameters**

- **agents** (*Iterable[Agent]*) – An iterable of Agent objects to be included in the set.
- **random** (*Random | None*) – The random number generator for this agent set.

**shuffle**(*inplace: bool = False*) → *AgentSet*[A]

Randomly shuffle the order of agents in the AgentSet.

**Parameters**

**inplace** (*bool, optional*) – If True, shuffles the agents in the current AgentSet; otherwise, returns a new shuffled AgentSet. Defaults to False.

**Returns**

A shuffled AgentSet. Returns the current AgentSet if inplace is True.

**Return type**

*AgentSet*

> **Note**
>
> Using inplace = True is more performant

**sort**(*key: Callable[[A], Any] | str, ascending: bool = False, inplace: bool = False*) → *AgentSet*[A]

Sort the agents in the AgentSet based on a specified attribute or custom function.

**Parameters**

- **key** (*Callable[[Agent], Any] | str*) – A function or attribute name based on which the agents are sorted.
- **ascending** (*bool, optional*) – If True, the agents are sorted in ascending order. Defaults to False.
- **inplace** (*bool, optional*) – If True, sorts the agents in the current AgentSet; otherwise, returns a new sorted AgentSet. Defaults to False.

**Returns**

A sorted AgentSet. Returns the current AgentSet if inplace is True.

**Return type**

*AgentSet*

**do**(*method: str | Callable, *args, **kwargs*) → *AgentSet*[A]

Invoke a method or function on each agent in the AgentSet.

**Parameters**

- **method** (*str, callable*) – the callable to do on each agent
  - in case of str, the name of the method to call on each agent.
  - in case of callable, the function to be called with each agent as first argument
- ***args** – Variable length argument list passed to the callable being called.
- ****kwargs** – Arbitrary keyword arguments passed to the callable being called.

**Returns**

The AgentSet instance itself.

**Return type**

*AgentSet*

**shuffle_do**(*method: str | Callable, *args, **kwargs*) → *AgentSet*[A]

Shuffle the agents in the AgentSet and then invoke a method or function on each agent.

It's a fast, optimized version of calling shuffle() followed by do().

**map**(*method: str | Callable*, **args*, ***kwargs*) → list[Any]

Invoke a method or function on each agent in the AgentSet and return the results.

**Parameters**

- **method** (*str, callable*) – the callable to apply on each agent
  - in case of str, the name of the method to call on each agent.
  - in case of callable, the function to be called with each agent as first argument
- ***args** – Variable length argument list passed to the callable being called.
- ****kwargs** – Arbitrary keyword arguments passed to the callable being called.

**Returns**

The results of the callable calls

**Return type**

list[Any]

**add**(*agent: A*)

Add an agent to the AgentSet.

**Parameters**

**agent** (*Agent*) – The agent to add to the set.

> **Note**
>
> This method is an implementation of the abstract method from MutableSet.

**discard**(*agent: A*)

Remove an agent from the AgentSet if it exists.

This method does not raise an error if the agent is not present.

**Parameters**

**agent** (*Agent*) – The agent to remove from the set.

> **Note**
>
> This method is an implementation of the abstract method from MutableSet.

**remove**(*agent: A*)

Remove an agent from the AgentSet.

This method raises an error if the agent is not present.

**Parameters**

**agent** (*Agent*) – The agent to remove from the set.

> **Note**
>
> This method is an implementation of the abstract method from MutableSet.

**agg**(*attribute: str*, *func: Callable | Iterable[Callable]*) → Any | list[Any]

Aggregate an attribute of all agents in the AgentSet using one or more functions.

**Parameters**

- **attribute** (*str*) – The name of the attribute to aggregate.
- **func** (*Callable | Iterable[Callable]*) –
  - If Callable: A single function to apply to the attribute values (e.g., min, max, sum, np.mean)
  - If Iterable: Multiple functions to apply to the attribute values

**Returns**

Result of applying the function(s) to the attribute values.

**Return type**

Any | [Any, ...]

### Examples

# Single function avg_energy = model.agents.agg("energy", np.mean)

# Multiple functions min_wealth, max_wealth, total_wealth = model.agents.agg("wealth", [min, max, sum])

**clear**()

This is slow (creates N new iterators!) but effective.

**count**(*value*) → integer -- return number of occurrences of value

**get**(*attr_names*, *handle_missing='error'*, *default_value=None*)

Retrieve the specified attribute(s) from each agent in the AgentSet.

**Parameters**

- **attr_names** (*str | list[str]*) – The name(s) of the attribute(s) to retrieve from each agent.
- **handle_missing** (*str, optional*) – How to handle missing attributes. Can be: - 'error' (default): raises an AttributeError if attribute is missing. - 'default': returns the specified default_value.
- **default_value** (*Any, optional*) – The default value to return if 'handle_missing' is set to 'default' and the agent does not have the attribute.

**Returns**

A list with the attribute value for each agent if attr_names is a str. list[list[Any]]: A list with a lists of attribute values for each agent if attr_names is a list of str.

**Return type**

list[Any]

**Raises**

- **AttributeError** – If 'handle_missing' is 'error' and the agent does not have the specified attribute(s).
- **ValueError** – If an unknown 'handle_missing' option is provided.

**groupby**(*by: Callable | str*, *result_type: Literal['agentset', 'list'] = 'agentset'*) → *GroupBy*

Group agents by the specified attribute or return from the callable.

**Parameters**

- **by** (*Callable, str*) – used to determine what to group agents by
  - if by is a callable, it will be called for each agent and the return is used for grouping
  - if by is a str, it should refer to an attribute on the agent and the value of this attribute will be used for grouping
- **result_type** (*str, optional*) – The datatype for the resulting groups {"agentset", "list"}

**Returns**

GroupBy

### Notes

There might be performance benefits to using *result_type='list'* if you don't need the advanced functionality of an AbstractAgentSet.

**index**(*value*[, *start*[, *stop* ]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

**isdisjoint**(*other*)

Return True if two sets have a null intersection.

**pop**()

Return the popped value. Raise KeyError if empty.

**select**(*filter_func: Callable[[A], bool] | None = None*, *at_most: int | float = inf*, *inplace: bool = False*, *agent_type: type[A] | None = None*) → *AbstractAgentSet*[A]

Select a subset of agents from the AbstractAgentSet based on a filter function and/or quantity limit.

**Parameters**

- **filter_func** (*Callable[[Agent], bool], optional*) – A function that takes an Agent and returns True if the agent should be included in the result. Defaults to None, meaning no filtering is applied.
- **at_most** (*int | float, optional*) – The maximum amount of agents to select. Defaults to infinity. - If an integer, at most the first number of matching agents are selected. - If a float between 0 and 1, at most that fraction of original the agents are selected.
- **inplace** (*bool, optional*) – If True, modifies the current AbstractAgentSet; otherwise, returns a new AbstractAgentSet. Defaults to False.
- **agent_type** (*type[Agent], optional*) – The class type of the agents to select. Defaults to None, meaning no type filtering is applied.

**Returns**

A new AbstractAgentSet containing the selected agents, unless inplace is True, in which case the current AbstractAgentSet is updated.

**Return type**

*AbstractAgentSet*

### Notes

- at_most just return the first n or fraction of agents. To take a random sample, shuffle() beforehand.
- at_most is an upper limit. When specifying other criteria, the number of agents returned can be smaller.

**set**(*attr_name: str*, *value: Any*) → *AgentSet*[A]

Set a specified attribute to a given value for all agents in the AgentSet.

**Parameters**

- **attr_name** (*str*) – The name of the attribute to set.
- **value** (*Any*) – The value to set the attribute to.

**Returns**
The AgentSet instance itself, after setting the attribute.

**Return type**
*AgentSet*

**to_list**() → list[A]

Convert the AbstractAgentSet to a list.

**Returns**
A list containing all agents in the AbstractAgentSet.

**Return type**
list[Agent]

### Notes

This method provides an explicit way to convert the AgentSet to a list. It is the recommended approach when list operations (indexing, slicing) are needed, as direct sequence operations on AgentSet are deprecated and will be removed in Mesa 4.0.

**class GroupBy**(*groups: dict[Any, list | AbstractAgentSet]*)

Helper class for AgentSet.groupby.

**groups**

A dictionary with the group_name as key and group as values

**Type**
dict

Initialize a GroupBy instance.

**Parameters**
**groups** (*dict*) – A dictionary with the group_name as key and group as values

**get_group**(*name: Hashable*, *default: Any = <object object>*) → list | *AbstractAgentSet* | Any

Return the group for the given name.

**Parameters**

- **name** (*Hashable*) – The group name to retrieve.
- **default** (*Any, optional*) – Value to return when the group is missing.

**Raises**
**KeyError** – If the group does not exist and no default is provided.

**map**(*method: Callable | str*, **args*, ***kwargs*) → dict[Any, Any]

Apply the specified callable to each group and return the results.

**Parameters**

- **method** (*Callable, str*) – The callable to apply to each group,
  - if `method` is a callable, it will be called it will be called with the group as first argument
  - if `method` is a str, it should refer to a method on the group

  Additional arguments and keyword arguments will be passed on to the callable.
- **args** – arguments to pass to the callable
- **kwargs** – keyword arguments to pass to the callable

**Returns**
dict with group_name as key and the return of the method as value

### Notes

this method is useful for methods or functions that do return something. It will break method chaining. For that, use `do` instead.

**do**(*method: Callable | str*, **args*, ***kwargs*) → *GroupBy*

Apply the specified callable to each group.

**Parameters**

- **method** (*Callable, str*) – The callable to apply to each group,
  - if `method` is a callable, it will be called it will be called with the group as first argument
  - if `method` is a str, it should refer to a method on the group

  Additional arguments and keyword arguments will be passed on to the callable.
- **args** – arguments to pass to the callable
- **kwargs** – keyword arguments to pass to the callable

**Returns**
the original GroupBy instance

### Notes

this method is useful for methods or functions that don't return anything and/or if you want to chain multiple do calls

**count**() → dict[Any, int]

Return the count of agents in each group.

**Returns**
A dictionary mapping group names to the number of agents in each group.

**Return type**
dict

**agg**(*attr_name: str*, *func: Callable*) → dict[Hashable, Any]

Aggregate the values of a specific attribute across each group using the provided function.

**Parameters**

- **attr_name** (*str*) – The name of the attribute to aggregate.

- **func** (*Callable*) – The function to apply (e.g., sum, min, max, mean).

**Returns**
A dictionary mapping group names to the result of applying the aggregation function.

**Return type**
dict[Hashable, Any]

#### 2.4.5.4 Time

Underlying modules for event scheduling and time advancement.

This module provides the foundational data structures and classes needed for event-based simulation in Mesa. The EventList class is a priority queue implementation that maintains simulation events in chronological order while respecting event priorities. Key features:

- Priority-based event ordering
- Weak references to prevent memory leaks from canceled events
- Efficient event insertion and removal using a heap queue
- Support for event cancellation without breaking the heap structure

**class Event**(*time: int | float*, *function: Callable[[...], None]*, *priority:* Priority = *Priority.DEFAULT*, *function_args: list[Any] | None = None*, *function_kwargs: dict[str, Any] | None = None*)

A simulation event.

The callable is wrapped using weakref, so there is no need to explicitly cancel event if e.g., an agent is removed from the simulation.

**time**

The simulation time of the event

**Type**
float

**fn**

The function to execute for this event

**Type**
Callable

**priority**

The priority of the event

**Type**
*Priority*

**unique_id**

**Type**
int

**function_args list[Any]**

Argument for the function

**function_kwargs dict[str, Any]**

Keyword arguments for the function

### Notes

Simulation events use a weak reference to the callable. If the callback no longer exists at execution time (e.g., because an agent has been removed), execution will fail silently. Lambda callbacks are rejected at Event creation.

Initialize a simulation event.

**Parameters**

- **time** – the instant of time of the simulation event
- **function** – the callable to invoke
- **priority** – the priority of the event
- **function_args** – arguments for callable
- **function_kwargs** – keyword arguments for the callable

**execute**() → None

Execute this event.

**cancel**() → None

Cancel this event.

**class EventGenerator**(*model: Model*, *function: Callable[..., None]*, *schedule:* Schedule, *priority:* Priority = *Priority.DEFAULT*)

A generator that creates recurring events based on a Schedule.

Unlike a single Event, an EventGenerator is persistent and can be stopped or configured with stop conditions.

**model**

The model this generator belongs to

**function**

The callable to execute for each generated event

**schedule**

The Schedule defining when events occur

**priority**

Priority level for generated events

### Notes

Event generators use a weak reference to the callable. Therefore, you cannot pass a lambda function in fn. A simulation event where the callable no longer exists (e.g., because the agent has been removed from the model) will fail silently. If you want to use functools.partial, please assign the partial function to a variable prior to creating the generator.

Initialize an EventGenerator.

**Parameters**

- **model** – The model this generator belongs to
- **function** – The callable to execute for each generated event. Use functools.partial to bind arguments.
- **schedule** – The Schedule defining timing
- **priority** – Priority level for generated events

**property is_active: bool**

Return whether the generator is currently active.

**property execution_count: int**

Return the number of times this generator has executed.

**property next_scheduled_time: float | None**

Return the time of the next scheduled execution, or None if not scheduled.

**start**() → *EventGenerator*

Start the event generator.

**Returns**
Self for method chaining

**stop**() → None

Stop the event generator immediately.

**pause**() → None

Pause the event generator temporarily.

This cancels the currently scheduled event but keeps the generator active in the model. Execution can be resumed later using resume().

**resume**() → None

Resume a paused event generator.

**class EventList**

An event list.

This is a heap queue sorted list of events. Events are always removed from the left, so heapq is a performant and appropriate data structure. Events are sorted based on their time stamp, their priority, and their unique_id as a tie-breaker, guaranteeing a complete ordering.

Initialize an event list.

**add_event**(*event:* Event)

Add the event to the event list.

**Parameters**
**event** (Event) – The event to be added

**peek_ahead**(*n: int = 1*) → list[*Event*]

Look at the first n non-canceled event in the event list.

**Parameters**
**n** (*int*) – The number of events to look ahead

**Returns**
list[Event]

**Raises**
**IndexError** – If the eventlist is empty

### Notes

this method can return a list shorted then n if the number of non-canceled events on the event list is less than n.

**pop_event**() → *Event*

Pop the first element from the event list.

**compact**() → None

Remove all canceled events from the heap and rebuild it.

If there are many canceled events, compaction can speed up performance substantially.

**is_empty**() → bool

Return whether the event list is empty.

**remove**(*event:* Event) → None

Remove an event from the event list.

**Parameters**
**event** (Event) – The event to be removed

**clear**() → None

Clear the event list.

**class Priority**(**values*)

Enumeration of priority levels.

**conjugate**()

Returns self, the complex conjugate of any int.

**bit_length**()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

**bit_count**()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

**to_bytes**(*length=1*, *byteorder='big'*, ***, *signed=False*)

Return an array of bytes representing an integer.

**length**
Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

**byteorder**
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use sys.byteorder as the byte order value. Default is to use 'big'.

**signed**
Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

`classmethod from_bytes`(*bytes*, *byteorder='big'*, ***, *signed=False*)

Return the integer represented by the given array of bytes.

**bytes**
Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

**byteorder**
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use sys.byteorder as the byte order value. Default is to use 'big'.

**signed**
Indicates whether two's complement is used to represent the integer.

`as_integer_ratio`()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

`is_integer`()

Returns True. Exists for duck type compatibility with float.is_integer.

`real`

the real part of a complex number

`imag`

the imaginary part of a complex number

`numerator`

the numerator of a rational number in lowest terms

`denominator`

the denominator of a rational number in lowest terms

`class Schedule`(*interval: float | int | Callable[[Model], float | int] = 1.0*, *start: float | None = None*, *end: float | None = None*, *count: int | None = None*)

Defines when something should happen repeatedly.

`interval`

Time between executions (fixed value or callable returning value)

**Type**
float | int | Callable[[Model], float | int]

**start**

Absolute time to begin (None = use current model time + interval)

**Type**
float | None

**end**

Absolute time to stop (None = no end)

**Type**
float | None

**count**

Maximum executions (None = unlimited)

**Type**
int | None

#### 2.4.5.5 Discrete Space

Cell spaces for active, property-rich spatial modeling in Mesa.

Cell spaces extend Mesa's spatial modeling capabilities by making the space itself active - each position (cell) can have properties and behaviors rather than just containing agents. This enables more sophisticated environmental modeling and agent-environment interactions.

Key components: - Cells: Active positions that can have properties and contain agents - CellAgents: Agents that understand how to interact with cells - Spaces: Different cell organization patterns (grids, networks, etc.)

This is particularly useful for models where the environment plays an active role, like resource growth, pollution diffusion, or infrastructure networks. The cell space system is experimental and under active development.

**class Cell**(*coordinate: tuple[int, ...]*, *position: ndarray | None = None*, *capacity: int | None = None*, *random: Random | None = None*)

The cell represents a position in a discrete space.

**coordinate**

the logical position(or index) of the cell in the discrete space

**Type**
Coordinate

**position**

the physical position of the cell in the discrete space

**Type**
np.ndarray | None

**agents**

the agents occupying the cell

**Type**
List[Agent]

**capacity**

the maximum number of agents that can simultaneously occupy the cell

**Type**
int

**random**
the random number generator

**Type**
Random

Initialise the cell.

**Parameters**

- **coordinate** – coordinates of the cell
- **position** – physical coordinates of the cell
- **capacity** (*int*) – the capacity of the cell. If None, the capacity is infinite
- **random** (*Random*) – the random number generator to use

**property position: ndarray**
Get the physical position of the cell.

**Returns**
Physical position of the cell

**Return type**
np.ndarray

**connect**(*other:* Cell, *key:* *tuple[int, ...] | None = None*) → None
Connects this cell to another cell.

**Parameters**

- **other** (Cell) – other cell to connect to
- **key** (*Tuple[int, ...]*) – key for the connection. Should resemble a relative coordinate

**disconnect**(*other:* Cell) → None
Disconnects this cell from another cell.

**Parameters**
**other** (Cell) – other cell to remove from connections

**add_agent**(*agent:* CellAgent) → None
Adds an agent to the cell.

**Parameters**
**agent** (CellAgent) – agent to add to this Cell

**remove_agent**(*agent:* CellAgent) → None
Removes an agent from the cell.

**Parameters**
**agent** (CellAgent) – agent to remove from this cell

**property is_empty: bool**
Returns a bool of the contents of a cell.

**property is_full: bool**
Returns a bool of the contents of a cell.

**property agents: list[*CellAgent*]**
Returns a list of the agents occupying the cell.

**property neighborhood: *CellCollection*[*Cell*]**

Returns the direct neighborhood of the cell.

This is equivalent to cell.get_neighborhood(radius=1)

**get_neighborhood**(*radius: int = 1*, *include_center: bool = False*) → *CellCollection*[*Cell*]

Returns a list of all neighboring cells for the given radius.

For getting the direct neighborhood (i.e., radius=1) you can also use the *neighborhood* property.

**Parameters**

- **radius** (*int*) – the radius of the neighborhood
- **include_center** (*bool*) – include the center of the neighborhood

**Returns**

a list of all neighboring cells

**class CellAgent**(*model: M*, **args*, ***kwargs*)

Cell Agent is an extension of the Agent class and adds behavior for moving in discrete spaces.

**cell**

The cell the agent is currently in.

**Type**

*Cell*

Create a new agent.

**Parameters**

- **model** (*Model*) – The model instance in which the agent exists.
- **args** – Passed on to super.
- **kwargs** – Passed on to super.

### Notes

to make proper use of python's super, in each class remove the arguments and keyword arguments you need and pass on the rest to super

**remove**()

Remove the agent from the model.

**class CellCollection**(*cells: Mapping[T, list[*CellAgent*]] | Iterable[T]*, *random: Random | None = None*)

An immutable collection of cells.

**cells**

The list of cells this collection represents

**Type**

List[*Cell*]

**agents**

List of agents occupying the cells in this collection

**Type**

List[*CellAgent*]

**random**

The random number generator

**Type**
Random

### Notes

A *UserWarning* is issued if *random=None*. You can resolve this warning by explicitly passing a random number generator. In most cases, this will be the seeded random number generator in the model. So, you would do *random=self.random* in a *Model* or *Agent* instance.

Initialize a CellCollection.

**Parameters**

- **cells** – cells to add to the collection
- **random** – a seeded random number generator.

**select_random_cell**(*default=<object object>*) → T | None

Select a random cell from the collection.

**Parameters**
**default** – Value to return if the collection is empty. If not provided, raises LookupError.

**Returns**
A random cell, or the default value if provided and collection is empty.

**Return type**
T

**Raises**
**LookupError** – If collection is empty and no default is provided.

**select_random_agent**(*default=<object object>*) → *CellAgent* | None

Select a random agent from the collection.

**Parameters**
**default** – Value to return if the collection is empty. If not provided, raises LookupError.

**Returns**
A random agent, or the default value if provided and collection is empty.

**Return type**
*CellAgent*

**Raises**
**LookupError** – If collection is empty and no default is provided.

**select**(*filter_func: Callable[[T], bool] | None = None*, *at_most: int | float = inf*)

Select cells based on filter function.

**Parameters**

- **filter_func** – filter function
- **at_most** – The maximum amount of cells to select. Defaults to infinity. - If an integer, at most the first number of matching cells is selected. - If a float between 0 and 1, at most that fraction of original number of cells

**Returns**
CellCollection

class DiscreteSpace(*capacity: int | None = None*, *cell_klass: type[T] = <class 'mesa.discrete_space.cell.Cell'>*, *random: Random | None = None*)

Base class for all discrete spaces.

**capacity**

The capacity of the cells in the discrete space

**Type**
int

**all_cells**

The cells composing the discrete space

**Type**
*CellCollection*

**random**

The random number generator

**Type**
Random

**cell_klass**

the type of cell class

**Type**
Type

**empties**

collection of all cells that are empty

**Type**
*CellCollection*

**property_layers**

property_layer of the discrete space

**Type**
dict[str, np.ndarray]

### Notes

A *UserWarning* is issued if *random=None*. You can resolve this warning by explicitly passing a random number generator. In most cases, this will be the seeded random number generator in the model. So, you would do *random=self.random* in a *Model* or *Agent* instance.

Instantiate a DiscreteSpace.

**Parameters**

- **capacity** – capacity of cells
- **cell_klass** – base class for all cells
- **random** – random number generator

property agents: *AgentSet*

Return an AgentSet with the agents in the space.

abstractmethod find_nearest_cell(*position: ndarray*) → T

Find the cell at or nearest to the given position.

**add_cell**(*cell: T*)

Add a cell to the space.

**Parameters**
**cell** – cell to add

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

> **Warning**
>
> Coordinate Collision: If a cell already exists at the specified coordinates (e.g., cell1), it will be overwritten silently by the new cell (cell2).
>
> Ensure the target coordinates are vacant before calling this method.

**remove_cell**(*cell: T*)

Remove a cell from the space.

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

**add_connection**(*cell1: T*, *cell2: T*)

Add a connection between the two cells.

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

**remove_connection**(*cell1: T*, *cell2: T*)

Remove a connection between the two cells.

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

**property all_cells**

Return all cells in space.

**property empties: *CellCollection*[T]**

Return all empty in spaces.

**select_random_empty_cell**() → T

Select random empty cell.

**class FixedAgent**(*model: M*, **args*, ***kwargs*)

A patch in a 2D grid.

Create a new agent.

**Parameters**

- **model** (*Model*) – The model instance in which the agent exists.
- **args** – Passed on to super.
- **kwargs** – Passed on to super.

### Notes

to make proper use of python's super, in each class remove the arguments and keyword arguments you need and pass on the rest to super

**remove**()

Remove the agent from the model.

**class Grid**(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)

Base class for all grid classes.

**dimensions**

the dimensions of the grid

**Type**
Sequence[int]

**torus**

whether the grid is a torus

**Type**
bool

**capacity**

the capacity of a grid cell

**Type**
int

**random**

the random number generator

**Type**
Random

**_try_random**

whether to get empty cell be repeatedly trying random cell

**Type**
bool

### Notes

width and height are accessible via property_layers, higher dimensions can be retrieved via dimensions

Initialise the grid class.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell
- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

**property width: int**

Convenience access to the width of the grid.

**property height: int**

Convenience access to the height of the grid.

**create_property_layer**(*name: str*, *default_value=0.0*, *dtype=<class 'float'>*, *read_only: bool = False*) → ndarray

Create a property layer array and attach it to cells.

Warning: Do not reassign *grid.name* directly — this will detach it from *property_layers* and break cell-level access. Use in-place operations.

grid.sugar[:] = 0 # correct grid.sugar = array # wrong — breaks cell access

**add_property_layer**(*name: str*, *array: ndarray*, *read_only: bool = False*) → None

Attach an existing array as a property layer. Shape must match *self.dimensions*.

Warning: Do not reassign *grid.name* directly — this will detach it from *property_layers* and break cell-level access. Use in-place operations.

grid.sugar[:] = 0 # correct grid.sugar = array # wrong — breaks cell access

**remove_property_layer**(*name: str*) → None

Remove a property_layer.

**Parameters**
**name** – property_layer name.

**get_neighborhood_mask**(*coordinate*, *include_center: bool = True*, *radius: int = 1*) → ndarray

Return a boolean mask shaped like `self.dimensions` for a neighborhood.

**find_nearest_cell**(*position: ndarray*) → T

Find the cell containing the given position.

**Parameters**
**position** – Physical position [x, y]

**Returns**
The cell containing the position

**Return type**
*Cell*

**Raises**
**ValueError** – If position is outside grid bounds and not a torus

**select_random_empty_cell**() → T

Select random empty cell.

**property cells_with_capacity: *CellCollection*[T]**

Return all cells that have available capacity (i.e. are not full).

A cell is considered *available* if `not cell.is_full`.

This is meaningfully different from *empties*: `empties` only includes cells with **zero** agents. If a cell has `capacity=5` and currently holds 3 agents it is **not** empty, but it **is** still available. `available_cells` is therefore the correct API for models where agents share cells up to a finite limit.

For cells with `capacity=None` (unlimited), every cell is always available regardless of how many agents it holds.

**Returns**
All cells where `len(agents) < capacity` (or all cells when capacity is None).

**Return type**
*CellCollection*[T]

Example:

```
# Place an agent in any non-full cell
agent.move_to(grid.select_random_cell_with_capacity())

# Count how many cells still have room
len(list(grid.cells_with_capacity))
```

**select_random_cell_with_capacity**() → *Cell*

Select a random cell that has remaining capacity.

**Raises**
**IndexError** – If every cell in the grid is at full capacity.

Example:

```
# Safe placement that respects capacity limits
free_cell = grid.select_random_cell_with_capacity()
agent.move_to(free_cell)
```

**class Grid2DMovingAgent**(*model: M*, **args*, ***kwargs*)

Mixin for moving agents in 2D grids.

Create a new agent.

**Parameters**

- **model** (*Model*) – The model instance in which the agent exists.
- **args** – Passed on to super.
- **kwargs** – Passed on to super.

### Notes

to make proper use of python's super, in each class remove the arguments and keyword arguments you need and pass on the rest to super

**move**(*direction: str*, *distance: int = 1*)

Move the agent in a cardinal direction.

**Parameters**

- **direction** – The cardinal direction to move in.
- **distance** – The distance to move.

**class HexGrid**(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)

A Grid with hexagonal tilling of the space.

**Note**

When torus=True, both width and height must be even.

**Raises**

**ValueError** – If torus=True and either width or height is odd.

Initialize the hex grid.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell
- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

**find_nearest_cell**(*position: ndarray*) → T

Find the hex cell at the given position.

**class Network**(*G: Any*, *capacity: int | None = None*, *random: Random | None = None*, *cell_klass: type[Cell] = <class 'mesa.discrete_space.cell.Cell'>*, *layout: Mapping | Callable | None = None*)

A networked discrete space.

A Networked grid.

**Parameters**

- **G** – a NetworkX Graph instance.
- **capacity** (*int*) – the capacity of the cell
- **random** (*Random*) – a random number generator
- **cell_klass** (*type[Cell]*) – The base Cell class to use in the Network
- **layout** – A dictionary mapping node IDs to physical positions (x, y), or a callable that generates them (e.g. nx.spring_layout). It defaults to nx.circular_layout This ensures all nodes

possess physical (x, y) positions for visualization and spatial queries without introducing performance bottlenecks on large graphs

**find_nearest_cell**(*position: ndarray*) → *Cell*

Find the network node nearest to the given position.

Only works for spatial networks (networks with node positions).

> **Note**
>
> The KD-Tree index is rebuilt lazily. If cells were added or removed since the previous spatial query, this method rebuilds the KD-Tree before performing the nearest-neighbor lookup.

**Parameters**
**position** – Physical position [x, y]

**Returns**
The node closest to the position

**Return type**
*Cell*

**Raises**
**ValueError** – If network is not spatial

**add_cell**(*cell:* Cell)

Add a cell to the space.

**remove_cell**(*cell:* Cell)

Remove a cell from the space.

**add_connection**(*cell1:* Cell, *cell2:* Cell)

Add a connection between the two cells.

**remove_connection**(*cell1:* Cell, *cell2:* Cell)

Remove a connection between the two cells.

**class OrthogonalMooreGrid**(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)

Grid where cells are connected to their 8 neighbors.

Example for two dimensions: directions = [

(-1, -1), (-1, 0), (-1, 1), ( 0, -1), ( 0, 1), ( 1, -1), ( 1, 0), ( 1, 1),

]

Initialise the grid class.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell
- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

class OrthogonalVonNeumannGrid(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)

Grid where cells are connected to their 4 neighbors.

Example for two dimensions: directions = [

(0, -1),

**(-1, 0), ( 1, 0),**
(0, 1),

]

Initialise the grid class.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell
- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

class VoronoiGrid(*centroids_coordinates: ~collections.abc.Sequence[~collections.abc.Sequence[float]], capacity: int | ~collections.abc.Callable | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.cell.Cell] = <class 'mesa.discrete_space.cell.Cell'>*)

Voronoi meshed GridSpace.

A Voronoi Tessellation Grid.

Given a set of points, this class creates a grid where a cell is centered in each point, its neighbors are given by Voronoi Tessellation cells neighbors and the capacity by the polygon area.

**Parameters**

- **centroids_coordinates** – coordinates of centroids to build the tessellation space
- **capacity** (*int*) – capacity of the cells in the discrete space
- **random** (*Random*) – random number generator
- **cell_klass** (*type[Cell]*) – type of cell class
- **capacity_function** (*Callable*) – function to compute (int) capacity according to (float) area

find_nearest_cell(*position: ndarray*) → *Cell*

Find the Voronoi cell nearest to the given position.

**Parameters**
**position** – Physical position [x, y]

**Returns**
The Voronoi cell whose centroid is nearest to position

**Return type**
*Cell*

Cells are positions in space that can have properties and contain agents.

A cell represents a location that can: - Have properties (like temperature or resources) - Track and limit the agents it contains - Connect to neighboring cells - Provide neighborhood information

Cells form the foundation of the cell space system, enabling rich spatial environments where both location properties and agent behaviors matter. They're useful for modeling things like varying terrain, infrastructure capacity, or environmental conditions.

**class Cell**(*coordinate: tuple[int, ...]*, *position: ndarray | None = None*, *capacity: int | None = None*, *random: Random | None = None*)

The cell represents a position in a discrete space.

**coordinate**

the logical position(or index) of the cell in the discrete space

**Type**
Coordinate

**position**

the physical position of the cell in the discrete space

**Type**
np.ndarray | None

**agents**

the agents occupying the cell

**Type**
List[Agent]

**capacity**

the maximum number of agents that can simultaneously occupy the cell

**Type**
int

**random**

the random number generator

**Type**
Random

Initialise the cell.

**Parameters**

- **coordinate** – coordinates of the cell
- **position** – physical coordinates of the cell
- **capacity** (*int*) – the capacity of the cell. If None, the capacity is infinite
- **random** (*Random*) – the random number generator to use

**property position: ndarray**

Get the physical position of the cell.

**Returns**
Physical position of the cell

**Return type**
np.ndarray

**connect**(*other:* Cell, *key: tuple[int, ...] | None = None*) → None

Connects this cell to another cell.

**Parameters**

- **other** (Cell) – other cell to connect to
- **key** (*Tuple[int, ...]*) – key for the connection. Should resemble a relative coordinate

**disconnect**(*other:* Cell) → None

Disconnects this cell from another cell.

**Parameters**
**other** (Cell) – other cell to remove from connections

**add_agent**(*agent:* CellAgent) → None

Adds an agent to the cell.

**Parameters**
**agent** (CellAgent) – agent to add to this Cell

**remove_agent**(*agent:* CellAgent) → None

Removes an agent from the cell.

**Parameters**
**agent** (CellAgent) – agent to remove from this cell

**property is_empty: bool**

Returns a bool of the contents of a cell.

**property is_full: bool**

Returns a bool of the contents of a cell.

**property agents: list[*CellAgent*]**

Returns a list of the agents occupying the cell.

**property neighborhood: *CellCollection*[*Cell*]**

Returns the direct neighborhood of the cell.

This is equivalent to cell.get_neighborhood(radius=1)

**get_neighborhood**(*radius: int = 1*, *include_center: bool = False*) → *CellCollection*[*Cell*]

Returns a list of all neighboring cells for the given radius.

For getting the direct neighborhood (i.e., radius=1) you can also use the *neighborhood* property.

**Parameters**

- **radius** (*int*) – the radius of the neighborhood
- **include_center** (*bool*) – include the center of the neighborhood

**Returns**
a list of all neighboring cells

Agents that understand how to exist in and move through cell spaces.

Provides specialized agent classes that handle cell occupation, movement, and proper registration: - CellAgent: Mobile agents that can move between cells - FixedAgent: Immobile agents permanently fixed to cells - Grid2DMovingAgent: Agents with grid-specific movement capabilities

These classes ensure consistent agent-cell relationships and proper state management as agents move through the space. They can be used directly or as examples for creating custom cell-aware agents.

class HasCellProtocol(*args, **kwargs)

Protocol for discrete space cell holders.

class HasCell

Descriptor for cell movement behavior.

class BasicMovement

Mixin for moving agents in discrete space.

move_to(*cell:* Cell) → None

Move to a new cell.

move_relative(*direction: tuple[int, ...]*)

Move to a cell relative to the current cell.

**Parameters**

**direction** – The direction to move in.

class FixedCell

Mixin for agents that are fixed to a cell.

Once assigned to a cell, the agent cannot be moved to a different cell. The assignment is atomic: if cell.add_agent() raises (e.g. capacity reached), the agent's _mesa_cell reference is left unchanged.

property cell: *Cell* | None

The cell the agent is fixed to.

class CellAgent(*model: M*, *args, **kwargs)

Cell Agent is an extension of the Agent class and adds behavior for moving in discrete spaces.

cell

The cell the agent is currently in.

**Type**

*Cell*

Create a new agent.

**Parameters**

- **model** (*Model*) – The model instance in which the agent exists.
- **args** – Passed on to super.
- **kwargs** – Passed on to super.

### Notes

to make proper use of python's super, in each class remove the arguments and keyword arguments you need and pass on the rest to super

remove()

Remove the agent from the model.

class FixedAgent(*model: M*, *args, **kwargs)

A patch in a 2D grid.

Create a new agent.

**Parameters**

- **model** (*Model*) – The model instance in which the agent exists.

- **args** – Passed on to super.
- **kwargs** – Passed on to super.

### Notes

to make proper use of python's super, in each class remove the arguments and keyword arguments you need and pass on the rest to super

**remove**()

Remove the agent from the model.

**class Grid2DMovingAgent**(*model: M*, **args*, ***kwargs*)

Mixin for moving agents in 2D grids.

Create a new agent.

**Parameters**

- **model** (*Model*) – The model instance in which the agent exists.
- **args** – Passed on to super.
- **kwargs** – Passed on to super.

### Notes

to make proper use of python's super, in each class remove the arguments and keyword arguments you need and pass on the rest to super

**move**(*direction: str*, *distance: int = 1*)

Move the agent in a cardinal direction.

**Parameters**

- **direction** – The cardinal direction to move in.
- **distance** – The distance to move.

Collection class for managing and querying groups of cells.

The CellCollection class provides a consistent interface for operating on multiple cells, supporting: - Filtering and selecting cells based on conditions - Random cell and agent selection - Access to contained agents - Group operations

This is useful for implementing area effects, zones, or any operation that needs to work with multiple cells as a unit. The collection handles efficient iteration and agent access across cells. The class is used throughout the cell space implementation to represent neighborhoods, selections, and other cell groupings.

**class CellCollection**(*cells: Mapping[T, list[*CellAgent*]] | Iterable[T]*, *random: Random | None = None*)

An immutable collection of cells.

**cells**

The list of cells this collection represents

**Type**
List[*Cell*]

**agents**

List of agents occupying the cells in this collection

**Type**
List[*CellAgent*]

**random**

The random number generator

**Type**
Random

### Notes

A *UserWarning* is issued if *random=None*. You can resolve this warning by explicitly passing a random number generator. In most cases, this will be the seeded random number generator in the model. So, you would do *random=self.random* in a *Model* or *Agent* instance.

Initialize a CellCollection.

**Parameters**

- **cells** – cells to add to the collection
- **random** – a seeded random number generator.

**select_random_cell**(*default=<object object>*) → T | None

Select a random cell from the collection.

**Parameters**
**default** – Value to return if the collection is empty. If not provided, raises LookupError.

**Returns**
A random cell, or the default value if provided and collection is empty.

**Return type**
T

**Raises**
**LookupError** – If collection is empty and no default is provided.

**select_random_agent**(*default=<object object>*) → *CellAgent* | None

Select a random agent from the collection.

**Parameters**
**default** – Value to return if the collection is empty. If not provided, raises LookupError.

**Returns**
A random agent, or the default value if provided and collection is empty.

**Return type**
*CellAgent*

**Raises**
**LookupError** – If collection is empty and no default is provided.

**select**(*filter_func: Callable[[T], bool] | None = None*, *at_most: int | float = inf*)

Select cells based on filter function.

**Parameters**

- **filter_func** – filter function
- **at_most** – The maximum amount of cells to select. Defaults to infinity. - If an integer, at most the first number of matching cells is selected. - If a float between 0 and 1, at most that fraction of original number of cells

**Returns**
CellCollection

Abstract Base class for building cell-based spatial environments.

DiscreteSpace provides the core functionality needed by all cell-based spaces: - Cell creation and tracking - Agent-cell relationship management - Property Layer support - Random selection capabilities - Capacity management

This serves as the foundation for specific space implementations like grids and networks, ensuring consistent behavior and shared functionality across different space types. All concrete cell space implementations (grids, networks, etc.) inherit from this class.

**class DiscreteSpace**(*capacity: int | None = None*, *cell_klass: type[T] = <class 'mesa.discrete_space.cell.Cell'>*, *random: Random | None = None*)

Base class for all discrete spaces.

**capacity**

The capacity of the cells in the discrete space

**Type**
int

**all_cells**

The cells composing the discrete space

**Type**
*CellCollection*

**random**

The random number generator

**Type**
Random

**cell_klass**

the type of cell class

**Type**
Type

**empties**

collection of all cells that are empty

**Type**
*CellCollection*

**property_layers**

property_layer of the discrete space

**Type**
dict[str, np.ndarray]

### Notes

A *UserWarning* is issued if *random=None*. You can resolve this warning by explicitly passing a random number generator. In most cases, this will be the seeded random number generator in the model. So, you would do *random=self.random* in a *Model* or *Agent* instance.

Instantiate a DiscreteSpace.

**Parameters**

- **capacity** – capacity of cells
- **cell_klass** – base class for all cells

- **random** – random number generator

**property agents:** *AgentSet*

Return an AgentSet with the agents in the space.

**abstractmethod find_nearest_cell**(*position: ndarray*) → T

Find the cell at or nearest to the given position.

**add_cell**(*cell: T*)

Add a cell to the space.

**Parameters**
**cell** – cell to add

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

> **Warning**
>
> Coordinate Collision: If a cell already exists at the specified coordinates (e.g., cell1), it will be overwritten silently by the new cell (cell2).
>
> Ensure the target coordinates are vacant before calling this method.

**remove_cell**(*cell: T*)

Remove a cell from the space.

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

**add_connection**(*cell1: T*, *cell2: T*)

Add a connection between the two cells.

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

**remove_connection**(*cell1: T*, *cell2: T*)

Remove a connection between the two cells.

> **Note**
>
> Discrete spaces rely on caching neighborhood relations for speedups. Adding or removing cells and connections at runtime is possible. However, only the caches of cells directly affected will be cleared. So if you rely on getting neighborhoods of cells with a radius higher than 1, these might not be cleared correctly if you are adding or removing cells and connections at runtime.

**property all_cells**

Return all cells in space.

**property empties: *CellCollection*[T]**

Return all empty in spaces.

**select_random_empty_cell**() → T

Select random empty cell.

Grid-based cell space implementations with different connection patterns.

Provides several grid types for organizing cells: - OrthogonalMooreGrid: 8 neighbors in 2D, (3^n)-1 in nD - OrthogonalVonNeumannGrid: 4 neighbors in 2D, 2n in nD - HexGrid: 6 neighbors in hexagonal pattern (2D only)

Each grid type supports optional wrapping (torus) and cell capacity limits. Choose based on how movement and connectivity should work in your model - Moore for unrestricted movement, Von Neumann for orthogonal-only movement, or Hex for more uniform distances.

**pickle_gridcell**(*obj*)

Helper function for pickling GridCell instances.

**unpickle_gridcell**(*parent*, *fields*)

Helper function for unpickling GridCell instances.

**class Grid**(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)

Base class for all grid classes.

**dimensions**

the dimensions of the grid

**Type**
Sequence[int]

**torus**

whether the grid is a torus

**Type**
bool

**capacity**

the capacity of a grid cell

**Type**
int

**random**

the random number generator

**Type**
Random

**_try_random**

whether to get empty cell be repeatedly trying random cell

**Type**
bool

### Notes

width and height are accessible via property_layers, higher dimensions can be retrieved via dimensions

Initialise the grid class.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell
- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

**property width: int**

Convenience access to the width of the grid.

**property height: int**

Convenience access to the height of the grid.

**create_property_layer**(*name: str*, *default_value=0.0*, *dtype=<class 'float'>*, *read_only: bool = False*) → ndarray

Create a property layer array and attach it to cells.

Warning: Do not reassign *grid.name* directly — this will detach it from *property_layers* and break cell-level access. Use in-place operations.

grid.sugar[:] = 0 # correct grid.sugar = array # wrong — breaks cell access

**add_property_layer**(*name: str*, *array: ndarray*, *read_only: bool = False*) → None

Attach an existing array as a property layer. Shape must match *self.dimensions*.

Warning: Do not reassign *grid.name* directly — this will detach it from *property_layers* and break cell-level access. Use in-place operations.

grid.sugar[:] = 0 # correct grid.sugar = array # wrong — breaks cell access

**remove_property_layer**(*name: str*) → None

Remove a property_layer.

**Parameters**
**name** – property_layer name.

**get_neighborhood_mask**(*coordinate*, *include_center: bool = True*, *radius: int = 1*) → ndarray

Return a boolean mask shaped like `self.dimensions` for a neighborhood.

**find_nearest_cell**(*position: ndarray*) → T

Find the cell containing the given position.

**Parameters**
**position** – Physical position [x, y]

**Returns**
The cell containing the position

**Return type**
*Cell*

**Raises**
**ValueError** – If position is outside grid bounds and not a torus

**select_random_empty_cell**() → T
Select random empty cell.

**property cells_with_capacity: *CellCollection*[T]**
Return all cells that have available capacity (i.e. are not full).

A cell is considered *available* if `not cell.is_full`.

This is meaningfully different from `empties`: `empties` only includes cells with **zero** agents. If a cell has `capacity=5` and currently holds 3 agents it is **not** empty, but it **is** still available. `available_cells` is therefore the correct API for models where agents share cells up to a finite limit.

For cells with `capacity=None` (unlimited), every cell is always available regardless of how many agents it holds.

**Returns**
All cells where `len(agents) < capacity` (or all cells when capacity is None).

**Return type**
*CellCollection*[T]

Example:

```
# Place an agent in any non-full cell
agent.move_to(grid.select_random_cell_with_capacity())

# Count how many cells still have room
len(list(grid.cells_with_capacity))
```

**select_random_cell_with_capacity**() → *Cell*
Select a random cell that has remaining capacity.

**Raises**
**IndexError** – If every cell in the grid is at full capacity.

Example:

```
# Safe placement that respects capacity limits
free_cell = grid.select_random_cell_with_capacity()
agent.move_to(free_cell)
```

**class OrthogonalMooreGrid**(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)
Grid where cells are connected to their 8 neighbors.

Example for two dimensions: directions = [

(-1, -1), (-1, 0), (-1, 1), ( 0, -1), ( 0, 1), ( 1, -1), ( 1, 0), ( 1, 1),

]

Initialise the grid class.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell
- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

**class OrthogonalVonNeumannGrid**(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)

Grid where cells are connected to their 4 neighbors.

Example for two dimensions: directions = [

(0, -1),

**(-1, 0), ( 1, 0),**
(0, 1),

]

Initialise the grid class.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell
- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

**class HexGrid**(*dimensions: ~collections.abc.Sequence[int], torus: bool = False, capacity: float | None = None, random: ~random.Random | None = None, cell_klass: type[~mesa.discrete_space.grid.T] = <class 'mesa.discrete_space.cell.Cell'>*)

A Grid with hexagonal tilling of the space.

**Note**

When torus=True, both width and height must be even.

**Raises**

**ValueError** – If torus=True and either width or height is odd.

Initialize the hex grid.

**Parameters**

- **dimensions** – the dimensions of the space
- **torus** – whether the space wraps
- **capacity** – capacity of the grid cell

- **random** – a random number generator
- **cell_klass** – the base class to use for the cells

**find_nearest_cell**(*position: ndarray*) → T

Find the hex cell at the given position.

Network-based cell space using arbitrary connection patterns.

Creates spaces where cells connect based on network relationships rather than spatial proximity. Built on NetworkX graphs, this enables: - Arbitrary connectivity patterns between cells - Graph-based neighborhood definitions - Logical rather than physical distances - Dynamic connectivity changes - Integration with NetworkX's graph algorithms

Useful for modeling systems like social networks, transportation systems, or any environment where connectivity matters more than physical location.

**class Network**(*G: Any*, *capacity: int | None = None*, *random: Random | None = None*, *cell_klass: type[Cell] = <class 'mesa.discrete_space.cell.Cell'>*, *layout: Mapping | Callable | None = None*)

A networked discrete space.

A Networked grid.

**Parameters**

- **G** – a NetworkX Graph instance.
- **capacity** (*int*) – the capacity of the cell
- **random** (*Random*) – a random number generator
- **cell_klass** (*type[Cell]*) – The base Cell class to use in the Network
- **layout** – A dictionary mapping node IDs to physical positions (x, y), or a callable that generates them (e.g. nx.spring_layout). It defaults to nx.circular_layout This ensures all nodes possess physical (x, y) positions for visualization and spatial queries without introducing performance bottlenecks on large graphs

**find_nearest_cell**(*position: ndarray*) → *Cell*

Find the network node nearest to the given position.

Only works for spatial networks (networks with node positions).

**Note**

The KD-Tree index is rebuilt lazily. If cells were added or removed since the previous spatial query, this method rebuilds the KD-Tree before performing the nearest-neighbor lookup.

**Parameters**
**position** – Physical position [x, y]

**Returns**
The node closest to the position

**Return type**
*Cell*

**Raises**
**ValueError** – If network is not spatial

**add_cell**(*cell:* Cell)

Add a cell to the space.

**remove_cell**(*cell:* Cell)

Remove a cell from the space.

**add_connection**(*cell1:* Cell, *cell2:* Cell)

Add a connection between the two cells.

**remove_connection**(*cell1:* Cell, *cell2:* Cell)

Remove a connection between the two cells.

Cell spaces based on Voronoi tessellation around seed points.

Creates irregular spatial divisions by building cells around seed points, where each cell contains the area closer to its seed than any other. Features: - Organic-looking spaces from point sets - Automatic neighbor determination - Area-based cell capacities - Natural regional divisions

Useful for models requiring irregular but mathematically meaningful spatial divisions, like territories, service areas, or natural regions.

**class Delaunay**(*center: tuple = (0, 0)*, *radius: int = 9999*)

Class to compute a Delaunay triangulation in 2D.

ref: http://github.com/jmespadero/pyDelaunay2D

Init and create a new frame to contain the triangulation.

**Parameters**

- **center** – Optional position for the center of the frame. Default (0,0)
- **radius** – Optional distance from corners to the center.

**add_point**(*point: Sequence*) → None

Add a point to the current DT, and refine it using Bowyer-Watson.

**export_triangles**() → list

Export the current list of Delaunay triangles.

**export_voronoi_regions**()

Export coordinates and regions of Voronoi diagram as indexed data.

**class VoronoiGrid**(*centroids_coordinates: ~collections.abc.Sequence[~collections.abc.Sequence[float]]*, *capacity: int | ~collections.abc.Callable | None = None*, *random: ~random.Random | None = None*, *cell_klass: type[~mesa.discrete_space.cell.Cell] = <class 'mesa.discrete_space.cell.Cell'>*)

Voronoi meshed GridSpace.

A Voronoi Tessellation Grid.

Given a set of points, this class creates a grid where a cell is centered in each point, its neighbors are given by Voronoi Tessellation cells neighbors and the capacity by the polygon area.

**Parameters**

- **centroids_coordinates** – coordinates of centroids to build the tessellation space
- **capacity** (*int*) – capacity of the cells in the discrete space
- **random** (*Random*) – random number generator
- **cell_klass** (*type[Cell]*) – type of cell class

- **capacity_function** (*Callable*) – function to compute (int) capacity according to (float) area

**find_nearest_cell**(*position: ndarray*) → *Cell*

Find the Voronoi cell nearest to the given position.

**Parameters**
**position** – Physical position [x, y]

**Returns**
The Voronoi cell whose centroid is nearest to position

**Return type**
*Cell*

#### 2.4.5.6 Data collection

Mesa Data Collection Module.

DataCollector is meant to provide a simple, standard way to collect data generated by a Mesa model. It collects four types of data: model-level data, agent-level data, agent-type-level data, and tables.

A DataCollector is instantiated with three dictionaries of reporter names and associated variable names or functions for each, one for model-level data, one for agent-level data, and one for agent-type-level data; a fourth dictionary provides table names and columns. Variable names are converted into functions which retrieve attributes of that name.

When the collect() method is called, each model-level function is called, with the model as the argument, and the results associated with the relevant variable. Then the agent-level functions are called on each agent, and the agent-type-level functions are called on each agent of the specified type.

Additionally, other objects can write directly to tables by passing in an appropriate dictionary object for a table row.

**The DataCollector then stores the data it collects in dictionaries:**

- model_vars maps each reporter to a list of its values
- tables maps each table to a dictionary, with each column as a key with a list as its value.
- _agent_records maps each model step to a list of each agent's id and its values.
- _agenttype_records maps each model step to a dictionary of agent types, each containing a list of each agent's id and its values.

Finally, DataCollector can create a pandas DataFrame from each collection.

**class DataCollector**(*model_reporters=None*, *agent_reporters=None*, *agenttype_reporters=None*, *tables=None*)

Class for collecting data generated by a Mesa model.

A DataCollector is instantiated with dictionaries of names of model-, agent-, and agent-type-level variables to collect, associated with attribute names or functions which actually collect them. When the collect(...) method is called, it collects these attributes and executes these functions one by one and stores the results.

Instantiate a DataCollector with lists of model, agent, and agent-type reporters.

Both model_reporters, agent_reporters, and agenttype_reporters accept a dictionary mapping a variable name to either an attribute name, a function, a method of a class/instance, a partial function, or a function with parameters placed in a list.

Model reporters can take five types of arguments: 1. Lambda function:

{"agent_count": lambda m: len(m.agents)}

2. Method of a class/instance: {"agent_count": self.get_agent_count} # self here is a class instance {"agent_count": Model.get_agent_count} # Model here is a class

3. Class attributes of a model: {“model_attribute”: “model_attribute”}
4. Partial function: {“agent_count”: functools.partial(count_agents, multiplier=2)}
5. Functions with parameters that have been placed in a list: {“Model_Function”: [function, [param_1, param_2]]}

Agent reporters can similarly take: 1. Attribute name (string) referring to agent’s attribute:

{“energy”: “energy”}

2. Lambda function: {“energy”: lambda a: a.energy}
3. Method of an agent class/instance: {“agent_action”: self.do_action} # self here is an agent class instance {“agent_action”: Agent.do_action} # Agent here is a class
4. Partial function: {“energy”: functools.partial(get_energy, scale=2)}
5. Functions with parameters placed in a list: {“Agent_Function”: [function, [param_1, param_2]]}

Agenttype reporters take a dictionary mapping agent types to dictionaries of reporter names and attributes/funcs/methods, similar to agent_reporters:

{Wolf: {“energy”: lambda a: a.energy}}

The tables arg accepts a dictionary mapping names of tables to lists of columns. For example, if we want to allow agents to write their age when they are destroyed (to keep track of lifespans), it might look like:

{“Lifespan”: [“unique_id”, “age”]}

**Parameters**

- **model_reporters** – Dictionary of reporter names and attributes/funcs/methods.
- **agent_reporters** – Dictionary of reporter names and attributes/funcs/methods.
- **agenttype_reporters** – Dictionary of agent types to dictionaries of reporter names and attributes/funcs/methods.
- **tables** – Dictionary of table names to lists of column names.

### Notes

- If you want to pickle your model you must not use lambda functions.
- If your model includes a large number of agents, it is recommended to use attribute names for the agent reporter, as it will be faster.

**collect**(*model*)

Collect all the data for the given model object.

**add_table_row**(*table_name*, *row*, *ignore_missing=False*)

Add a row dictionary to a specific table.

**Parameters**

- **table_name** – Name of the table to append a row to.
- **row** – A dictionary of the form {column_name: value...}
- **ignore_missing** – If True, fill any missing columns with Nones; if False, throw an error if any columns are missing

`get_model_vars_dataframe()`

Create a pandas DataFrame from the model variables.

The DataFrame has one column for each model variable, and the index is (implicitly) the model tick.

`get_agent_vars_dataframe()`

Create a pandas DataFrame from the agent variables.

The DataFrame has one column for each variable, with two additional columns for tick and agent_id.

`get_agenttype_vars_dataframe(agent_type)`

Create a pandas DataFrame from the agent-type variables for a specific agent type.

The DataFrame has one column for each variable, with two additional columns for tick and agent_id.

**Parameters**
**agent_type** – The type of agent to get the data for.

`get_table_dataframe(table_name)`

Create a pandas DataFrame from a particular table.

**Parameters**
**table_name** – The name of the table to convert.

#### 2.4.5.7 Visualization

**Important note for SolaraViz users**

When using **SolaraViz**, Mesa models must be instantiated **using keyword arguments only**. SolaraViz creates model instances internally via keyword-based parameters, and positional arguments are **not supported**.

**Not supported:**

```
MyModel(10, 10)
```

**Supported:**

```
MyModel(width=10, height=10)
```

To avoid errors, it is recommended to define your model constructor with keyword-only arguments, for example:

```
class MyModel(Model):
    def __init__(self, *, width, height, seed=None):
        ...
```

For detailed tutorials, please refer to:

- *Basic Visualization*
- *Dynamic Agent Visualization*
- *Custom Agent Visualization*

##### 2.4.5.7.1 Jupyter Visualization

Mesa visualization module for creating interactive model visualizations.

This module provides components to create browser- and Jupyter notebook-based visualizations of Mesa models, allowing users to watch models run step-by-step and interact with model parameters.

**Key features:**

- SolaraViz: Main component for creating visualizations, supporting grid displays and plots
- ModelController: Handles model execution controls (step, play, pause, reset)
- UserInputs: Generates UI elements for adjusting model parameters

The module uses Solara for rendering in Jupyter notebooks or as standalone web applications. It supports various types of visualizations including matplotlib plots, agent grids, and custom visualization components.

**Usage:**

1. Define an agent_portrayal function to specify how agents should be displayed
2. Set up model_params to define adjustable parameters
3. Create a SolaraViz instance with your model, parameters, and desired measures
4. Display the visualization in a Jupyter notebook or run as a Solara app

See the Visualization Tutorial and example models for more details.

`create_space_component`(*renderer:* SpaceRenderer)

Create a space visualization component for the given renderer.

`split_model_params`(*model_params*)

Split model parameters into user-adjustable and fixed parameters.

**Parameters**
`model_params` – Dictionary of all model parameters

**Returns**
(user_adjustable_params, fixed_params)

**Return type**
tuple

`check_param_is_fixed`(*param*)

Check if a parameter is fixed (not user-adjustable).

**Parameters**
`param` – Parameter to check

**Returns**
True if parameter is fixed, False otherwise

**Return type**
bool

`make_initial_grid_layout`(*num_components*)

Create an initial grid layout for visualization components.

**Parameters**
`num_components` – Number of components to display

**Returns**
Initial grid layout configuration

**Return type**
list

`copy_renderer`(*renderer:* SpaceRenderer, *model: Model*)

Create a new renderer instance with the same configuration as the original.

Custom visualization components.

**class AgentPortrayalStyle**(*x: float | None = None, y: float | None = None, color: str | tuple | int | float | None = 'tab:blue', marker: str | None = 'o', size: int | float | None = 50, zorder: int | None = 1, alpha: float | None = 1.0, edgecolors: str | tuple | None = None, linewidths: float | int | None = 1.0, tooltip: dict | None = None*)

Bases: `object`

Represents the visual styling options for an agent in a visualization.

User facing component to control how agents are drawn. Allows specifying properties like color, size, marker shape, position, and other plot attributes.

x, y are determined automatically according to the agent's type (normal/CellAgent) and position in the space if not manually declared.

### Example

```
>>> def agent_portrayal(agent):
>>>     return AgentPortrayalStyle(
>>>         x=agent.cell.coordinate[0],
>>>         y=agent.cell.coordinate[1],
>>>         color="red",
>>>         marker="o",
>>>         size=20,
>>>         zorder=2,
>>>         alpha=0.8,
>>>         edgecolors="black",
>>>         linewidths=1.5
>>>     )
>>>
>>> # or for a default agent portrayal
>>> def agent_portrayal(agent):
>>>     return AgentPortrayalStyle()
```

**x:** `float | None = None`

**y:** `float | None = None`

**color:** `str | tuple | int | float | None = 'tab:blue'`

**marker:** `str | None = 'o'`

**size:** `int | float | None = 50`

**zorder:** `int | None = 1`

**alpha:** `float | None = 1.0`

**edgecolors:** `str | tuple | None = None`

**linewidths:** `float | int | None = 1.0`

**tooltip:** `dict | None = None`

**update**(**updates_fields: tuple[str, Any]*)

Updates attributes from variable (field_name, new_value) tuple arguments.

### Example

```
>>> def agent_portrayal(agent):
>>>     primary_style = AgentPortrayalStyle(color="blue", marker="^", size=10,␣
↪x=agent.pos[0], y=agent.pos[1])
>>>     if agent.type == 1:
>>>         primary_style.update(("color", "red"), ("size", 30))
>>>     return primary_style
```

**class PropertyLayerStyle**(*colormap: str | None = None*, *color: str | None = None*, *alpha: float = 0.8*, *colorbar: bool = True*, *vmin: float | None = None*, *vmax: float | None = None*)

Bases: `object`

Represents the visual styling options for a property layer in a visualization.

User facing component to control how property layers are drawn. Allows specifying properties like colormap, single color, value limits (vmin, vmax), transparency (alpha) and colorbar visibility.

Note: vmin and vmax are the lower and upper bounds for the colorbar and the data is normalized between these values for color/colormap rendering. If they are not declared the values are automatically determined from the data range.

Note: You can specify either a 'colormap' (for varying data) or a single 'color' (for a uniform layer appearance), but not both simultaneously.

### Example

```
>>> def property_layer_portrayal(layer):
>>>     return PropertyLayerStyle(colormap="viridis", vmin=0, vmax=100, alpha=0.5,␣
↪colorbar=True)
>>> # or for a uniform color layer
>>> def property_layer_portrayal(layer):
>>>     return PropertyLayerStyle(color="lightblue", alpha=0.8, colorbar=False)
```

**colormap:** `str | None = None`

**color:** `str | None = None`

**alpha:** `float = 0.8`

**colorbar:** `bool = True`

**vmin:** `float | None = None`

**vmax:** `float | None = None`

**make_altair_space**(*agent_portrayal*, *property_layer_portrayal=None*, *post_process=None*, ***space_drawing_kwargs*)

Create an Altair-based space visualization component.

**Parameters**

- **agent_portrayal** – Function to portray agents.
- **property_layer_portrayal** – Dictionary of property_layer portrayal specifications
- **post_process** – A user specified callable that will be called with the Chart instance from Altair. Allows for fine tuning plots (e.g., control ticks)
- **space_drawing_kwargs** – not yet implemented

agent_portrayal is called with an agent and should return a dict. Valid fields in this dict are "color", "size", "marker", and "zorder". Other field are ignored and will result in a user warning.

**Returns**
A function that creates a SpaceMatplotlib component

**Return type**
*function*

**make_mpl_plot_component**(*measure: str | dict[str, str] | list[str] | tuple[str], post_process: Callable | None = None, page: int = 0, save_format='png'*)

Create a plotting function for a specified measure.

**Parameters**

- **measure** (*str | dict[str, str] | list[str] | tuple[str]*) – Measure(s) to plot.
- **post_process** – a user-specified callable to do post-processing called with the Axes instance.
- **page** – Page number where the plot should be displayed.
- **save_format** – save format of figure in solara backend

**Returns**
A tuple of a function that creates a PlotMatplotlib component and a page number.

**Return type**
(*function*, page)

**make_mpl_space_component**(*agent_portrayal: Callable | None = None, property_layer_portrayal: dict | None = None, post_process: Callable | None = None, **space_drawing_kwargs*) → SpaceMatplotlib

Create a Matplotlib-based space visualization component.

**Parameters**

- **agent_portrayal** – Function to portray agents.
- **property_layer_portrayal** – Dictionary of property_layer portrayal specifications
- **post_process** – a callable that will be called with the Axes instance. Allows for fine tuning plots (e.g., control ticks)
- **space_drawing_kwargs** – additional keyword arguments to be passed on to the underlying space drawer function. See the functions for drawing the various spaces for further details.

agent_portrayal is called with an agent and should return a dict. Valid fields in this dict are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

**Returns**
A function that creates a SpaceMatplotlib component

**Return type**
*function*

**make_plot_component**(*measure: str | dict[str, str] | list[str] | tuple[str], post_process: Callable | None = None, backend: str = 'matplotlib', page: int = 0, **plot_drawing_kwargs*)

Create a plotting function for a specified measure using the specified backend.

**Parameters**

- **measure** (*str | dict[str, str] | list[str] | tuple[str]*) – Measure(s) to plot.
- **post_process** – a user-specified callable to do post-processing called with the Axes instance.
- **backend** – the backend to use {"matplotlib", "altair"}
- **page** – Page number where the plot should be displayed (default 0).
- **plot_drawing_kwargs** – additional keyword arguments to pass onto the backend specific function for making a plotting component

**Returns**
A tuple of a function and page number that creates a plot component on that specific page.

**Return type**
(*function*, page)

**make_space_component**(*agent_portrayal: Callable | None = None*, *property_layer_portrayal: dict | None = None*, *post_process: Callable | None = None*, *backend: str = 'matplotlib'*, ***space_drawing_kwargs*) → SpaceMatplotlib | SpaceAltair

Create a Matplotlib-based space visualization component.

**Parameters**

- **agent_portrayal** – Function to portray agents.
- **property_layer_portrayal** – Dictionary of property_layer portrayal specifications
- **post_process** – a callable that will be called with the Axes instance. Allows for fine-tuning plots (e.g., control ticks)
- **backend** – the backend to use {"matplotlib", "altair"}
- **space_drawing_kwargs** – additional keyword arguments to be passed on to the underlying backend specific space drawer function. See the functions for drawing the various spaces for the appropriate backend further details.

**Returns**
A function that creates a space component

**Return type**
*function*

#### 2.4.5.7.2 User Parameters

Solara visualization related helper classes.

**class UserParam**

Bases: `object`

UserParam.

**maybe_raise_error**(*param_type*, *valid*)

**class Slider**(*label=''*, *value=None*, *min=None*, *max=None*, *step=1*, *dtype=None*)

Bases: *UserParam*

A number-based slider input with settable increment.

Example: slider_option = Slider("My Slider", value=123, min=10, max=200, step=0.1)

**Parameters**

- **label** – The displayed label in the UI
- **value** – The initial value of the slider
- **min** – The minimum possible value of the slider
- **max** – The maximum possible value of the slider
- **step** – The step between min and max for a range of possible values
- **dtype** – either int or float

Initializes a slider.

**Parameters**

- **label** – The displayed label in the UI
- **value** – The initial value of the slider
- **min** – The minimum possible value of the slider
- **max** – The maximum possible value of the slider
- **step** – The step between min and max for a range of possible values
- **dtype** – either int or float

**get**(*attr*)

#### 2.4.5.7.3 Matplotlib-based visualizations

Matplotlib based solara components for visualization MESA spaces and plots.

**make_space_matplotlib**(**args*, ***kwargs*)

**make_mpl_space_component**(*agent_portrayal: Callable | None = None*, *property_layer_portrayal: dict | None = None*, *post_process: Callable | None = None*, ***space_drawing_kwargs*) → SpaceMatplotlib

Create a Matplotlib-based space visualization component.

**Parameters**

- **agent_portrayal** – Function to portray agents.
- **property_layer_portrayal** – Dictionary of property_layer portrayal specifications
- **post_process** – a callable that will be called with the Axes instance. Allows for fine tuning plots (e.g., control ticks)
- **space_drawing_kwargs** – additional keyword arguments to be passed on to the underlying space drawer function. See the functions for drawing the various spaces for further details.

`agent_portrayal` is called with an agent and should return a dict. Valid fields in this dict are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

**Returns**
A function that creates a SpaceMatplotlib component

**Return type**
*function*

**make_plot_measure**(**args*, ***kwargs*)

make_mpl_plot_component(*measure: str | dict[str, str] | list[str] | tuple[str], post_process: Callable | None = None, page: int = 0, save_format='png'*)

Create a plotting function for a specified measure.

**Parameters**

- **measure** (*str | dict[str, str] | list[str] | tuple[str]*) – Measure(s) to plot.
- **post_process** – a user-specified callable to do post-processing called with the Axes instance.
- **page** – Page number where the plot should be displayed.
- **save_format** – save format of figure in solara backend

**Returns**
A tuple of a function that creates a PlotMatplotlib component and a page number.

**Return type**
(*function*, page)

Helper functions for drawing mesa spaces with matplotlib.

These functions are used by the provided matplotlib components, but can also be used to quickly visualize a space with matplotlib for example when creating a mp4 of a movie run or when needing a figure for a paper.

collect_agent_data(*space: OrthogonalMooreGrid | OrthogonalVonNeumannGrid | HexGrid | Network | ContinuousSpace | VoronoiGrid, agent_portrayal: Callable, default_size: float | None = None*) → dict

Collect the plotting data for all agents in the space.

**Parameters**

- **space** – The space containing the Agents.
- **agent_portrayal** – A callable that is called with the agent and returns a AgentPortrayalStyle
- **default_size** – default size

agent_portrayal should return a AgentPortrayalStyle, limited to size (size of marker), color (color of marker), zorder (z-order), marker (marker style), alpha, linewidths, and edgecolors.

draw_space(*space, agent_portrayal: Callable, property_layer_portrayal: Callable | None = None, ax: Axes | None = None, **space_drawing_kwargs*)

Draw a Matplotlib-based visualization of the space.

**Parameters**

- **space** – the space of the mesa model
- **agent_portrayal** – A callable that returns a AgnetPortrayalStyle specifying how to show the agent
- **property_layer_portrayal** – A callable that returns a PropertyLayerStyle specifying how to show the property layer
- **ax** – the axes upon which to draw the plot
- **space_drawing_kwargs** – any additional keyword arguments to be passed on to the underlying function for drawing the space.

**Returns**
Returns the Axes object with the plot drawn onto it.

agent_portrayal is called with an agent and should return a AgentPortrayalStyle. Valid fields in this object are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

**draw_property_layers**(*space*, *property_layer_portrayal: dict[str, dict[str, Any]] | Callable*, *ax: Axes*)

Draw Property Layers on the given axes.

**Parameters**

- **space** – The space having the property_layer.
- **property_layer_portrayal** (*Callable*) – A function that accepts a property layer object and returns either a *PropertyLayerStyle* object defining its visualization, or *None* to skip drawing this particular layer.
- **ax** (*matplotlib.axes.Axes*) – The axes to draw on.

**draw_orthogonal_grid**(*space: OrthogonalMooreGrid | OrthogonalVonNeumannGrid*, *agent_portrayal: Callable*, *ax: Axes | None = None*, *draw_grid: bool = True*, ***kwargs*)

Visualize a orthogonal grid.

**Parameters**

- **space** – the space to visualize
- **agent_portrayal** – a callable that is called with the agent and returns a AgentPortrayalStyle
- **ax** – a Matplotlib Axes instance. If none is provided a new figure and ax will be created using plt.subplots
- **draw_grid** – whether to draw the grid
- **kwargs** – additional keyword arguments passed to ax.scatter

**Returns**
Returns the Axes object with the plot drawn onto it.

agent_portrayal is called with an agent and should return a AgentPortrayalStyle. Valid fields in this object are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

**draw_hex_grid**(*space: HexGrid*, *agent_portrayal: Callable*, *ax: Axes | None = None*, *draw_grid: bool = True*, ***kwargs*)

Visualize a hex grid.

**Parameters**

- **space** – the space to visualize
- **agent_portrayal** – a callable that is called with the agent and returns a AgentPortrayalStyle
- **ax** – a Matplotlib Axes instance. If none is provided a new figure and ax will be created using plt.subplots
- **draw_grid** – whether to draw the grid
- **kwargs** – additional keyword arguments passed to ax.scatter

**Returns**
Returns the Axes object with the plot drawn onto it.

agent_portrayal is called with an agent and should return a AgentPortrayalStyle. Valid fields in this object are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

**draw_network**(*space: Network*, *agent_portrayal: Callable*, *ax: Axes | None = None*, *draw_grid: bool = True*, ***kwargs*)

Visualize a network space.

**Parameters**

- **space** – the space to visualize
- **agent_portrayal** – a callable that is called with the agent and returns a AgentPortrayalStyle
- **ax** – a Matplotlib Axes instance. If none is provided a new figure and ax will be created using plt.subplots
- **draw_grid** – whether to draw the grid
- **kwargs** – additional keyword arguments passed to ax.scatter

**Returns**
Returns the Axes object with the plot drawn onto it.

`agent_portrayal` is called with an agent and should return a AgentPortrayalStyle. Valid fields in this object are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

**draw_continuous_space**(*space: ContinuousSpace*, *agent_portrayal: Callable*, *ax: Axes | None = None*, ***kwargs*)

Visualize a continuous space.

**Parameters**

- **space** – the space to visualize
- **agent_portrayal** – a callable that is called with the agent and returns a AgentPortrayalStyle
- **ax** – a Matplotlib Axes instance. If none is provided a new figure and ax will be created using plt.subplots
- **kwargs** – additional keyword arguments passed to ax.scatter

**Returns**
Returns the Axes object with the plot drawn onto it.

`agent_portrayal` is called with an agent and should return a AgentPortrayalStyle. Valid fields in this object are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

**draw_voronoi_grid**(*space: VoronoiGrid*, *agent_portrayal: Callable*, *ax: Axes | None = None*, *draw_grid: bool = True*, ***kwargs*)

Visualize a voronoi grid.

**Parameters**

- **space** – the space to visualize
- **agent_portrayal** – a callable that is called with the agent and returns a AgentPortrayalStyle
- **ax** – a Matplotlib Axes instance. If none is provided a new figure and ax will be created using plt.subplots
- **draw_grid** – whether to draw the grid or not
- **kwargs** – additional keyword arguments passed to ax.scatter

**Returns**
Returns the Axes object with the plot drawn onto it.

`agent_portrayal` is called with an agent and should return a AgentPortrayalStyle. Valid fields in this object are "color", "size", "marker", "zorder", alpha, linewidths, and edgecolors. Other field are ignored and will result in a user warning.

#### 2.4.5.7.4 Altair-based visualizations

Altair based solara components for visualization mesa spaces.

**make_space_altair**(**args*, ***kwargs*)

**make_altair_space**(*agent_portrayal*, *property_layer_portrayal=None*, *post_process=None*, ***space_drawing_kwargs*)

Create an Altair-based space visualization component.

**Parameters**

- **agent_portrayal** – Function to portray agents.
- **property_layer_portrayal** – Dictionary of property_layer portrayal specifications
- **post_process** – A user specified callable that will be called with the Chart instance from Altair. Allows for fine tuning plots (e.g., control ticks)
- **space_drawing_kwargs** – not yet implemented

`agent_portrayal` is called with an agent and should return a dict. Valid fields in this dict are "color", "size", "marker", and "zorder". Other field are ignored and will result in a user warning.

**Returns**
A function that creates a SpaceMatplotlib component

**Return type**
*function*

**chart_property_layers**(*space*, *property_layer_portrayal*, *chart_width*, *chart_height*)

Creates Property Layers in the Altair Components.

**Parameters**

- **space** – the ContinuousSpace instance
- **property_layer_portrayal** – Dictionary of property_layer portrayal specifications
- **chart_width** – width of the agent chart to maintain consistency with the property_layer charts
- **chart_height** – height of the agent chart to maintain consistency with the property_layer charts

**Returns**
Altair Chart

**make_altair_plot_component**(*measure: str | dict[str, str] | list[str] | tuple[str]*, *post_process: Callable | None = None*, *page: int = 0*, *grid=False*)

Create a plotting function for a specified measure.

**Parameters**

- **measure** (*str | dict[str, str] | list[str] | tuple[str]*) – Measure(s) to plot.
- **post_process** – a user-specified callable to do post-processing called with the Axes instance.

- **page** – Page number where the plot should be displayed.
- **grid** – Bool to draw grid or not.

**Returns**
A tuple of a function that creates a PlotAltair component and a page number.

**Return type**
(*function*, page)

#### 2.4.5.7.5 Command Console

A command console interface for interactive Python code execution in the browser.

This module provides a set of classes and functions to create an interactive Python console that can be embedded in a web browser. It supports command history, multi-line code blocks, and special commands for console management.

### Notes

- The console supports multi-line code blocks with proper indentation
- Output is captured and displayed with appropriate formatting
- Error messages are displayed in red with distinct styling
- The console maintains a history of commands and their outputs

**class ConsoleEntry**(*command: str*, *output: str = ''*, *is_error: bool = False*, *is_continuation: bool = False*)

Bases: `object`

A class to store command console entries.

**command**
The command entered

**Type**
str

**output**
The output of the command

**Type**
str

**is_error**
Whether the entry represents an error

**Type**
bool

**is_continuation**
Whether the entry is a continuation of previous command

**Type**
bool

**command:** `str`

**output:** `str = ''`

**is_error:** `bool = False`

**is_continuation: bool = False**

**class CaptureOutput**

Bases: object

A context manager for capturing stdout and stderr output.

This class provides a way to capture output that would normally be printed to stdout and stderr during the execution of code within its context.

Initialize the CaptureOutput context manager with empty string buffers.

**get_output()**

Retrieve and clear the captured output.

**Returns**
A pair of strings (stdout_output, stderr_output)

**Return type**
tuple

**class InteractiveConsole(*locals_dict=None*)**

Bases: InteractiveConsole

A custom interactive Python console with output capturing capabilities.

This class extends code.InteractiveConsole to provide output capturing functionality when executing Python code interactively.

**Parameters**
**locals_dict** (*dict, optional*) – Dictionary of local variables. Defaults to None.

Initialize the InteractiveConsole with the provided locals dictionary.

**push(*line*)**

Push a line to the command interpreter and execute it.

This method captures the output of the executed command and returns both the 'more' flag and the captured output.

**Parameters**
**line** (*str*) – The line of code to be executed.

**Returns**

**A tuple containing:**

- more (bool): Flag indicating if more input is needed
- str: The captured output from executing the command

**Return type**
tuple

**class ConsoleManager(*model=None*, *additional_imports=None*)**

Bases: object

A console manager for executing Python code interactively.

This class provides functionality to execute Python code in an interactive console environment, maintain command history, and handle multi-line code blocks.

**locals_dict**
Dictionary containing local variables available to the console

**Type**
dict

**console**
Python's interactive console instance

**Type**
*InteractiveConsole*

**buffer**
Buffer for storing multi-line code blocks

**Type**
list

**history**
List of console entries containing commands and their outputs

**Type**
list[*ConsoleEntry*]

**Special Commands:**

1. *history* : Shows the command history
2. *cls* : Clears the console screen
3. *tips* : Shows available console commands and usage tips

### Example

```
>>> console = ConsoleManager(model=my_model)
>>> console.execute_code("print('hello world')", set_input_callback)
```

Initialize the console manager with the provided model and imports.

**history:** list[*ConsoleEntry*]

**execute_code**(*code_line: str*, *set_input_text: Callable[[str], None]*) → None
Execute the provided code line and update the console history.

**clear_console**() → None
Clear the console history and reset the console state.

**get_entries**() → list[*ConsoleEntry*]
Get the list of console entries.

**prev_command**(*current_text: str*, *set_input_text: Callable[[str], None]*) → None
Navigate to previous command in history.

**next_command**(*set_input_text: Callable[[str], None]*) → None
Navigate to next command in history.

**format_command_html**(*entry*)
Format the command part of a console entry as HTML.

**format_output_html**(*entry*)
Format the output part of a console entry as HTML.

#### 2.4.5.7.6 Portrayal Components

Portrayal Components Module.

This module defines data structures for styling visual elements in Mesa agent-based model visualizations. It provides user-facing classes to specify how agents and property layers should appear in the rendered space.

Classes: 1. AgentPortrayalStyle: Controls the appearance of individual agents (e.g., color, shape, size, etc.). 2. PropertyLayerStyle: Controls the appearance of background property layers (e.g., color gradients or uniform fills).

These components are designed to be passed into Mesa visualizations to customize and standardize how data is presented.

**class AgentPortrayalStyle**(*x: float | None = None*, *y: float | None = None*, *color: str | tuple | int | float | None = 'tab:blue'*, *marker: str | None = 'o'*, *size: int | float | None = 50*, *zorder: int | None = 1*, *alpha: float | None = 1.0*, *edgecolors: str | tuple | None = None*, *linewidths: float | int | None = 1.0*, *tooltip: dict | None = None*)

Bases: `object`

Represents the visual styling options for an agent in a visualization.

User facing component to control how agents are drawn. Allows specifying properties like color, size, marker shape, position, and other plot attributes.

x, y are determined automatically according to the agent's type (normal/CellAgent) and position in the space if not manually declared.

### Example

```
>>> def agent_portrayal(agent):
>>>     return AgentPortrayalStyle(
>>>         x=agent.cell.coordinate[0],
>>>         y=agent.cell.coordinate[1],
>>>         color="red",
>>>         marker="o",
>>>         size=20,
>>>         zorder=2,
>>>         alpha=0.8,
>>>         edgecolors="black",
>>>         linewidths=1.5
>>>     )
>>>
>>> # or for a default agent portrayal
>>> def agent_portrayal(agent):
>>>     return AgentPortrayalStyle()
```

**x:** `float | None = None`

**y:** `float | None = None`

**color:** `str | tuple | int | float | None = 'tab:blue'`

**marker:** `str | None = 'o'`

**size:** `int | float | None = 50`

**zorder:** `int | None = 1`

**alpha:** *float | None = 1.0*

**edgecolors:** *str | tuple | None = None*

**linewidths:** *float | int | None = 1.0*

**tooltip:** *dict | None = None*

**update**(**updates_fields: tuple[str, Any]*)

Updates attributes from variable (field_name, new_value) tuple arguments.

### Example

```
>>> def agent_portrayal(agent):
>>>     primary_style = AgentPortrayalStyle(color="blue", marker="^", size=10,␣
↪x=agent.pos[0], y=agent.pos[1])
>>>     if agent.type == 1:
>>>         primary_style.update(("color", "red"), ("size", 30))
>>>     return primary_style
```

**class PropertyLayerStyle**(*colormap: str | None = None, color: str | None = None, alpha: float = 0.8, colorbar: bool = True, vmin: float | None = None, vmax: float | None = None*)

Bases: `object`

Represents the visual styling options for a property layer in a visualization.

User facing component to control how property layers are drawn. Allows specifying properties like colormap, single color, value limits (vmin, vmax), transparency (alpha) and colorbar visibility.

Note: vmin and vmax are the lower and upper bounds for the colorbar and the data is normalized between these values for color/colormap rendering. If they are not declared the values are automatically determined from the data range.

Note: You can specify either a 'colormap' (for varying data) or a single 'color' (for a uniform layer appearance), but not both simultaneously.

### Example

```
>>> def property_layer_portrayal(layer):
>>>     return PropertyLayerStyle(colormap="viridis", vmin=0, vmax=100, alpha=0.5,␣
↪colorbar=True)
>>> # or for a uniform color layer
>>> def property_layer_portrayal(layer):
>>>     return PropertyLayerStyle(color="lightblue", alpha=0.8, colorbar=False)
```

**colormap:** *str | None = None*

**color:** *str | None = None*

**alpha:** *float = 0.8*

**colorbar:** *bool = True*

**vmin:** *float | None = None*

**vmax:** *float | None = None*

#### 2.4.5.7.7 Backends

Visualization backends for Mesa space rendering.

This module provides different backend implementations for visualizing Mesa agent-based model spaces and components.

> **Note**
>
> These backends are used internally by the space renderer and are not intended for direct use by end users. See *SpaceRenderer* for actual usage and setting up visualizations.

**Available Backends:**

1. AltairBackend
2. MatplotlibBackend

**class AltairBackend**(*space_drawer*)

Bases: *AbstractRenderer*

Altair-based renderer for Mesa spaces.

This module provides an Altair-based renderer for visualizing Mesa model spaces, agents, and property layers with interactive charting capabilities.

Initialize the renderer.

**Parameters**

- **space_drawer** – Object responsible for drawing space elements. Checkout *SpaceDrawer*
- **functions.** (*for more details on the detailed implementations of the drawing*)

**initialize_canvas**() → None

Initialize the Altair canvas.

**draw_structure**(***kwargs*) → Chart

Draw the space structure using Altair.

**Parameters**

- ****kwargs** – Additional arguments passed to the space drawer.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**

The Altair chart representing the space structure.

**Return type**

alt.Chart

**collect_agent_data**(*space*, *agent_portrayal: Callable*, *default_size: float | None = None*)

Collect plotting data for all agents in the space for Altair.

**Parameters**

- **space** – The Mesa space containing agents.
- **agent_portrayal** – Callable that returns AgentPortrayalStyle for each agent.
- **default_size** – Default marker size if not specified in portrayal.

**Returns**
Dictionary containing agent plotting data arrays.

**Return type**
dict

**draw_agents**(*arguments*, *chart_width: int = 450*, *chart_height: int = 350*, ***kwargs*)

Draw agents using Altair backend.

**Parameters**

- **arguments** – Dictionary containing agent data arrays.
- **chart_width** – Width of the chart.
- **chart_height** – Height of the chart.
- ****kwargs** – Additional keyword arguments for customization.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The Altair chart representing the agents, or None if no agents.

**Return type**
alt.Chart

**draw_property_layer**(*space*, *property_layers: dict[str, ndarray]*, *property_layer_portrayal: Callable*, *chart_width: int = 450*, *chart_height: int = 350*)

Draw property layers using Altair backend.

**Parameters**

- **space** – The Mesa space object containing the property layers.
- **property_layers** – A dictionary mapping property_layer names to numpy arrays.
- **property_layer_portrayal** – A function that returns PropertyLayerStyle that contains the visualization parameters.
- **chart_width** – The width of the chart.
- **chart_height** – The height of the chart.

**Returns**
A tuple containing the base chart and the color bar chart.

**Return type**
alt.Chart

**class MatplotlibBackend**(*space_drawer*)

Bases: *AbstractRenderer*

Matplotlib-based renderer for Mesa spaces.

Provides visualization capabilities using Matplotlib for rendering space structures, agents, and property layers.

Initialize the Matplotlib backend.

**Parameters**
**space_drawer** – An instance of a SpaceDrawer class that handles the drawing of the space structure.

**initialize_canvas**(*ax=None*)

Initialize the matplotlib canvas.

**Parameters**
**ax** (*matplotlib.axes.Axes, optional*) – Existing axes to use. If None, creates new figure and axes.

**draw_structure**(***kwargs*)

Draw the space structure using matplotlib.

**Parameters**

- ****kwargs** – Additional arguments passed to the space drawer.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The matplotlib axes with the drawn structure.

**collect_agent_data**(*space*, *agent_portrayal*, *default_size=None*)

Collect plotting data for all agents in the space.

**Parameters**

- **space** – The Mesa space containing agents.
- **agent_portrayal** (*Callable*) – Function that returns AgentPortrayalStyle for each agent.
- **default_size** (*float, optional*) – Default marker size if not specified in portrayal.

**Returns**
Dictionary containing agent plotting data arrays.

**Return type**
dict

**draw_agents**(*arguments*, ***kwargs*)

Draw agents on the backend's axes - optimized version.

**Parameters**

- **arguments** – Dictionary containing agent data arrays.
- ****kwargs** – Additional keyword arguments for customization.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The Matplotlib Axes with the agents drawn upon it.

**Return type**
matplotlib.axes.Axes

**draw_property_layer**(*space*, *property_layers*, *property_layer_portrayal*)

Draw property layers using matplotlib backend.

**Parameters**

- **space** – The Mesa space object.
- **property_layers** (*dict*) – Dictionary of property layers to visualize.

- **property_layer_portrayal** (*Callable*) – Function that returns PropertyLayerStyle.

**Returns**
(matplotlib.axes.Axes, colorbar) - The matplotlib axes and colorbar objects.

**Return type**
tuple

Abstract base class for visualization backends in Mesa.

This module provides the foundational interface for implementing various visualization backends for Mesa agent-based models.

**class AbstractRenderer**(*space_drawer*)

Bases: ABC

Abstract base class for visualization backends.

This class defines the interface for rendering Mesa spaces and agents. For details on the methods checkout specific backend implementations.

Initialize the renderer.

**Parameters**

- **space_drawer** – Object responsible for drawing space elements. Checkout *SpaceDrawer*
- **functions.** (*for more details on the detailed implementations of the drawing*)

**abstractmethod initialize_canvas**()

Set up the drawing canvas.

**abstractmethod draw_structure**(***kwargs*)

Draw the space structure.

**Parameters**
****kwargs** – Structure drawing configuration options.

**abstractmethod collect_agent_data**(*space*, *agent_portrayal*, *default_size=None*)

Collect plotting data for all agents in the space.

**Parameters**

- **space** – The Mesa space containing agents.
- **agent_portrayal** (*Callable*) – Function that returns AgentPortrayalStyle for each agent.
- **default_size** (*float, optional*) – Default marker size if not specified in portrayal.

**Returns**
Dictionary containing agent plotting data arrays with keys:

**Return type**
dict

**abstractmethod draw_agents**(*arguments*, ***kwargs*)

Drawing agents on space.

**Parameters**

- **arguments** (*dict*) – Dictionary containing agent data.
- ****kwargs** – Additional drawing configuration options.

**abstractmethod draw_property_layer**(*space*, *property_layers*, *property_layer_portrayal*)

Draw property layers on the visualization.

**Parameters**

- **space** – The model's space object.
- **property_layers** (*dict*) – Dictionary of property layers to visualize.
- **property_layer_portrayal** (*Callable*) – Function that returns PropertyLayerStyle.

**class AltairBackend**(*space_drawer*)

Bases: *AbstractRenderer*

Altair-based renderer for Mesa spaces.

This module provides an Altair-based renderer for visualizing Mesa model spaces, agents, and property layers with interactive charting capabilities.

Initialize the renderer.

**Parameters**

- **space_drawer** – Object responsible for drawing space elements. Checkout *SpaceDrawer*
- **functions.** (*for more details on the detailed implementations of the drawing*)

**initialize_canvas**() → None

Initialize the Altair canvas.

**draw_structure**(***kwargs*) → Chart

Draw the space structure using Altair.

**Parameters**

- ****kwargs** – Additional arguments passed to the space drawer.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The Altair chart representing the space structure.

**Return type**
alt.Chart

**collect_agent_data**(*space*, *agent_portrayal: Callable*, *default_size: float | None = None*)

Collect plotting data for all agents in the space for Altair.

**Parameters**

- **space** – The Mesa space containing agents.
- **agent_portrayal** – Callable that returns AgentPortrayalStyle for each agent.
- **default_size** – Default marker size if not specified in portrayal.

**Returns**
Dictionary containing agent plotting data arrays.

**Return type**
dict

**draw_agents**(*arguments*, *chart_width: int = 450*, *chart_height: int = 350*, ***kwargs*)

Draw agents using Altair backend.

**Parameters**

- **arguments** – Dictionary containing agent data arrays.
- **chart_width** – Width of the chart.
- **chart_height** – Height of the chart.
- ****kwargs** – Additional keyword arguments for customization.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The Altair chart representing the agents, or None if no agents.

**Return type**
alt.Chart

**draw_property_layer**(*space*, *property_layers: dict[str, ndarray]*, *property_layer_portrayal: Callable*, *chart_width: int = 450*, *chart_height: int = 350*)

Draw property layers using Altair backend.

**Parameters**

- **space** – The Mesa space object containing the property layers.
- **property_layers** – A dictionary mapping property_layer names to numpy arrays.
- **property_layer_portrayal** – A function that returns PropertyLayerStyle that contains the visualization parameters.
- **chart_width** – The width of the chart.
- **chart_height** – The height of the chart.

**Returns**
A tuple containing the base chart and the color bar chart.

**Return type**
alt.Chart

**class MatplotlibBackend**(*space_drawer*)

Bases: *AbstractRenderer*

Matplotlib-based renderer for Mesa spaces.

Provides visualization capabilities using Matplotlib for rendering space structures, agents, and property layers.

Initialize the Matplotlib backend.

**Parameters**
**space_drawer** – An instance of a SpaceDrawer class that handles the drawing of the space structure.

**initialize_canvas**(*ax=None*)

Initialize the matplotlib canvas.

**Parameters**
**ax** (*matplotlib.axes.Axes, optional*) – Existing axes to use. If None, creates new figure and axes.

**draw_structure**(***kwargs*)

Draw the space structure using matplotlib.

**Parameters**

- ****kwargs** – Additional arguments passed to the space drawer.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The matplotlib axes with the drawn structure.

**collect_agent_data**(*space*, *agent_portrayal*, *default_size=None*)

Collect plotting data for all agents in the space.

**Parameters**

- **space** – The Mesa space containing agents.
- **agent_portrayal** (*Callable*) – Function that returns AgentPortrayalStyle for each agent.
- **default_size** (*float, optional*) – Default marker size if not specified in portrayal.

**Returns**
Dictionary containing agent plotting data arrays.

**Return type**
dict

**draw_agents**(*arguments*, ***kwargs*)

Draw agents on the backend's axes - optimized version.

**Parameters**

- **arguments** – Dictionary containing agent data arrays.
- ****kwargs** – Additional keyword arguments for customization.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The Matplotlib Axes with the agents drawn upon it.

**Return type**
matplotlib.axes.Axes

**draw_property_layer**(*space*, *property_layers*, *property_layer_portrayal*)

Draw property layers using matplotlib backend.

**Parameters**

- **space** – The Mesa space object.
- **property_layers** (*dict*) – Dictionary of property layers to visualize.
- **property_layer_portrayal** (*Callable*) – Function that returns PropertyLayerStyle.

**Returns**
(matplotlib.axes.Axes, colorbar) - The matplotlib axes and colorbar objects.

**Return type**
tuple

#### 2.4.5.7.8 Space Renderer

Space rendering module for Mesa visualizations.

This module provides functionality to render Mesa model spaces with different backends, supporting various space types and visualization components.

**class SpaceRenderer**(*model: Model*, *backend: Literal['matplotlib', 'altair'] | None = 'matplotlib'*)

Bases: `object`

Renders Mesa spaces using different visualization backends.

Supports multiple space types and backends for flexible visualization of agent-based models.

Initialize the space renderer.

**Parameters**

- **model** (*mesa.Model*) – The Mesa model to render.
- **backend** (*Literal["matplotlib", "altair"] | None*) – The visualization backend to use.

**setup_structure**(***kwargs*) → *SpaceRenderer*

Setup the space structure without drawing.

**Parameters**

- ****kwargs** – Additional keyword arguments for the setup function. For ContinuousSpace, you may pass `viz_dims=(i, j)` to select which two dimensions are projected to x/y.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The current instance for method chaining.

**Return type**
*SpaceRenderer*

**setup_agents**(*agent_portrayal: Callable*, ***kwargs*) → *SpaceRenderer*

Setup agents on the space without drawing.

**Parameters**

- **agent_portrayal** (*Callable*) – Function that takes an agent and returns AgentPortrayalStyle.
- ****kwargs** – Additional keyword arguments for the setup function.
- ****kwargs.** (*Checkout respective SpaceDrawer class on details how to pass*) –

**Returns**
The current instance for method chaining.

**Return type**
*SpaceRenderer*

**setup_property_layer**(*property_layer_portrayal: Callable | dict | PropertyLayerStyle*) → *SpaceRenderer*

Setup property layers on the space without drawing.

**Parameters**
**property_layer_portrayal** (*Callable | dict | PropertyLayerStyle*) – A PropertyLayerStyle, a function that produces a PropertyLayerStyle instance, or a dictionary specifying portrayal parameters.

**Returns**
The current instance for method chaining.

**Return type**
*SpaceRenderer*

**draw_structure**(***kwargs*)

Draw the space structure.

**Parameters**
****kwargs** – (Deprecated) Additional keyword arguments for drawing. Use setup_structure() instead.

**Returns**
The visual representation of the space structure.

**draw_agents**(*agent_portrayal=None*, ***kwargs*)

Draw agents on the space.

**Parameters**

- **agent_portrayal** – (Deprecated) Function that takes an agent and returns AgentPortrayalStyle. Use setup_agents() instead.
- ****kwargs** – (Deprecated) Additional keyword arguments for drawing.

**Returns**
The visual representation of the agents.

**draw_property_layer**(*property_layer_portrayal=None*)

Draw property layers on the space.

**Parameters**

- **property_layer_portrayal** – (Deprecated) A PropertyLayerStyle, a function that produces
- **instance** (*a PropertyLayerStyle*)
- **parameters.** (*or a dictionary specifying portrayal*)
- **instead.** (*Use setup_property_layer()*)

**Returns**
The visual representation of the property layers.

**Raises**
**Exception** – If no property layers are found on the space.

**render**(*agent_portrayal=None*, *property_layer_portrayal=None*, ***kwargs*)

Render the complete space with structure, agents, and property layers.

**Parameters**

- **agent_portrayal** – (Deprecated) Function for agent portrayal. Use setup_agents() instead.
- **property_layer_portrayal** – (Deprecated) Function for property layer portrayal. Use setup_property_layer() instead.

- ****kwargs** – (Deprecated) Additional keyword arguments.

**property canvas**

Get the current canvas object.

**Returns**
The backend-specific canvas object.

**property post_process**

Get the current post-processing function.

**Returns**
The post-processing function, or None if not set.

**Return type**
Callable | None

#### 2.4.5.7.9 Space Drawers

Mesa visualization space drawers.

This module provides the core logic for drawing spaces in Mesa, supporting orthogonal grids, hexagonal grids, networks, continuous spaces, and Voronoi grids. It includes implementations for both Matplotlib and Altair backends.

**class BaseSpaceDrawer**(*space*)

Bases: object

Base class for all space drawers.

Initialize the base space drawer.

**Parameters**
**space** – Grid/Space type to draw.

**get_viz_limits**()

Get visualization limits for the space.

**Returns**
A tuple of (xmin, xmax, ymin, ymax) for visualization limits.

**class OrthogonalSpaceDrawer**(*space:* OrthogonalMooreGrid | OrthogonalVonNeumannGrid)

Bases: *BaseSpaceDrawer*

Drawer for orthogonal grid spaces (SingleGrid, MultiGrid, Moore, VonNeumann).

Initialize the orthogonal space drawer.

**Parameters**
**space** – The orthogonal grid space to draw

**draw_matplotlib**(*ax=None*, ***draw_space_kwargs*)

Draw the orthogonal grid using matplotlib.

**Parameters**

- **ax** – Matplotlib axes object to draw on
- ****draw_space_kwargs** – Additional keyword arguments for styling.

### Examples

figsize=(10, 10), color="blue", linewidth=2.

**Returns**
The modified axes object

**draw_altair**(*chart_width=450*, *chart_height=350*, ***draw_chart_kwargs*)

Draw the orthogonal grid using Altair.

**Parameters**

- **chart_width** – Width for the shown chart
- **chart_height** – Height for the shown chart
- ****draw_chart_kwargs** – Additional keyword arguments for styling the chart.

### Examples

width=500, height=500, title="Grid".

**Returns**
Altair chart object

**class HexSpaceDrawer**(*space:* HexGrid)

Bases: *BaseSpaceDrawer*

Drawer for hexagonal grid spaces.

Initialize the hexagonal space drawer.

**Parameters**
**space** – The hexagonal grid space to draw

**draw_matplotlib**(*ax=None*, ***draw_space_kwargs*)

Draw the hexagonal grid using matplotlib.

**Parameters**

- **ax** – Matplotlib axes object to draw on
- ****draw_space_kwargs** – Additional keyword arguments for styling.

### Examples

figsize=(8, 8), color="red", alpha=0.5.

**Returns**
The modified axes object

**draw_altair**(*chart_width=450*, *chart_height=350*, ***draw_chart_kwargs*)

Draw the hexagonal grid using Altair.

**Parameters**

- **chart_width** – Width for the shown chart
- **chart_height** – Height for the shown chart
- ****draw_chart_kwargs** – Additional keyword arguments for styling the chart.

### Examples

- Line properties like color, strokeDash, strokeWidth, opacity.
- Other kwargs (e.g., width, title) apply to the chart.

**Returns**
Altair chart object representing the hexagonal grid.

**class NetworkSpaceDrawer**(*space:* Network, *layout_alg=<function spring_layout>*, *layout_kwargs=None*)

Bases: *BaseSpaceDrawer*

Drawer for network-based spaces.

Initialize the network space drawer.

**Parameters**

- **space** – The network space to draw
- **layout_alg** – NetworkX layout algorithm to use
- **layout_kwargs** – Keyword arguments for the layout algorithm

**draw_matplotlib**(*ax=None*, ***draw_space_kwargs*)

Draw the network using matplotlib.

**Parameters**

- **ax** – Matplotlib axes object to draw on.
- ****draw_space_kwargs** – Dictionaries of keyword arguments for styling. Can also handle zorder for both nodes and edges if passed. * `node_kwargs`: A dict passed to nx.draw_networkx_nodes. * `edge_kwargs`: A dict passed to nx.draw_networkx_edges.

**Returns**
The modified axes object.

**draw_altair**(*chart_width=450*, *chart_height=350*, ***draw_chart_kwargs*)

Draw the network using Altair.

**Parameters**

- **chart_width** – Width for the shown chart
- **chart_height** – Height for the shown chart
- ****draw_chart_kwargs** – Dictionaries for styling the chart. * `node_kwargs`: A dict of properties for the node's mark_point. * `edge_kwargs`: A dict of properties for the edge's mark_rule. * Other kwargs (e.g., title, width) are passed to chart.properties().

**Returns**
Altair chart object representing the network.

**class ContinuousSpaceDrawer**(*space: ContinuousSpace*, *viz_dims: tuple[int, int] = (0, 1)*)

Bases: *BaseSpaceDrawer*

Drawer for continuous spaces.

Initialize the continuous space drawer.

**Parameters**

- **space** – The continuous space to draw.

- **viz_dims** – The pair of dimension indices to project onto the x and y axes.

**Raises**

- **ValueError** – If the space has fewer than two dimensions.
- **ValueError** – If viz_dims does not contain exactly two distinct valid indices.

**set_viz_dims**(*viz_dims: tuple[int, int]*) → None

Set which dimensions are visualized on the x and y axes.

**Parameters**
**viz_dims** – Tuple of two distinct dimension indices.

**Raises**
**ValueError** – If viz_dims is invalid for the underlying space.

**project**(*position*)

Project an n-dimensional position onto the configured 2D plane.

**draw_matplotlib**(*ax=None*, ***draw_space_kwargs*)

Draw the continuous space using matplotlib.

**Parameters**

- **ax** – Matplotlib axes object to draw on.
- ****draw_space_kwargs** – Keyword arguments for styling the axis frame. You may optionally pass `viz_dims=(i, j)` to select which two dimensions of an n-dimensional space are projected to x/y.

### Examples

linewidth=3, color="green"

**Returns**
The modified axes object.

**draw_altair**(*chart_width=450*, *chart_height=350*, ***draw_chart_kwargs*)

Draw the continuous space using Altair.

**Parameters**

- **chart_width** – Width for the shown chart.
- **chart_height** – Height for the shown chart.
- ****draw_chart_kwargs** – Keyword arguments for styling the chart's view properties. You may optionally pass `viz_dims=(i, j)` to select which two dimensions of an n-dimensional space are projected to x/y.

**Returns**
An Altair Chart object representing the space.

**class VoronoiSpaceDrawer**(*space: VoronoiGrid*)

Bases: *BaseSpaceDrawer*

Drawer for Voronoi diagram spaces.

Initialize the Voronoi space drawer.

**Parameters**
**space** – The Voronoi grid space to draw

**draw_matplotlib**(*ax=None*, ***draw_space_kwargs*)

Draw the Voronoi diagram using matplotlib.

**Parameters**

- **ax** – Matplotlib axes object to draw on
- ****draw_space_kwargs** – Keyword arguments passed to matplotlib's LineCollection.

#### Examples

lw=2, alpha=0.5, colors='red'

**Returns**
The modified axes object

**draw_altair**(*chart_width=450*, *chart_height=350*, ***draw_chart_kwargs*)

Draw the Voronoi diagram using Altair.

**Parameters**

- **chart_width** – Width for the shown chart
- **chart_height** – Height for the shown chart
- ****draw_chart_kwargs** – Additional keyword arguments for styling the chart.

#### Examples

- Line properties like color, strokeDash, strokeWidth, opacity.
- Other kwargs (e.g., width, title) apply to the chart.

**Returns**
An Altair Chart object representing the Voronoi diagram.

#### 2.4.5.8 Experimental

This namespace contains experimental features. These are under development, and their API is not necessarily stable.

##### 2.4.5.8.1 Continuous Space

A Continuous Space class.

**class ContinuousSpace**(*dimensions: ArrayLike*, *torus: bool = False*, *random: Random | None = None*, *n_agents: int = 100*)

Continuous space where each agent can have an arbitrary position.

Create a new continuous space.

**Parameters**

- **dimensions** – a numpy array like object where each row specifies the minimum and maximum value of that dimension.
- **torus** – boolean for whether the space wraps around or not
- **random** – a seeded stdlib random.Random instance
- **n_agents** – the expected number of agents in the space

Internally, a numpy array is used to store the positions of all agents. This is resized if needed, but you can control the initial size explicitly by passing n_agents.

**property agents:** *AgentSet*

Return an AgentSet with the agents in the space.

**calculate_difference_vector**(*point: ndarray*, *agents=None*) → ndarray

Calculate the difference vector between the point and all agenents.

**Parameters**

- **point** – the point to calculate the difference vector for
- **agents** – the agents to calculate the difference vector of point with. By default, all agents are considered.

**calculate_distances**(*point: ArrayLike*, *agents: Iterable[Agent] | None = None*, ***kwargs*) → tuple[ndarray, list]

Calculate the distance between the point and all agents.

**Parameters**

- **point** – the point to calculate the difference vector for
- **agents** – the agents to calculate the difference vector of point with. By default, all agents are considered.
- **kwargs** – any additional keyword arguments are passed to scipy's cdist, which is used only if torus is False. This allows for non-Euclidian distance measures.

**get_agents_in_radius**(*point: ArrayLike*, *radius: float | int = 1*) → tuple[list, ndarray]

Return the agents and their distances within a radius for the point.

**get_k_nearest_agents**(*point: ArrayLike*, *k: int = 1*) → tuple[list, ndarray]

Return the k nearest agents and their distances to the point.

### Notes

This method returns exactly k agents, ignoring ties. In case of ties, the earlier an agent is inserted the higher it will rank.

If fewer than k agents are present in the space, all agents are returned and a UserWarning is emitted to indicate that the requested k could not be satisfied. If the space is empty or k <= 0, an empty result is returned without a warning.

**in_bounds**(*point: ArrayLike*) → bool

Check if point is inside the bounds of the space.

**torus_correct**(*point: ArrayLike*) → ndarray

Apply a torus correction to the point.

Continuous space agents.

**class HasPositionProtocol**(**args*, ***kwargs*)

Protocol for continuous space position holders.

**class ContinuousSpaceAgent**(*space: ContinuousSpace*, *model*)

A continuous space agent.

**space**

the continuous space in which the agent is located

**Type**
*ContinuousSpace*

**position**

the position of the agent

**Type**
np.ndarray

Initialize a continuous space agent.

**Parameters**

- **space** – the continuous space in which the agent is located
- **model** – the model to which the agent belongs

**property position: ndarray**

Position of the agent.

**remove**() → None

Remove and delete the agent from the model and continuous space.

**get_neighbors_in_radius**(*radius: float | int = 1*) → tuple[list, ndarray]

Get neighbors within radius.

**Parameters**
**radius** – radius within which to look for neighbors

**get_nearest_neighbors**(*k: int = 1*) → tuple[list, ndarray]

Get neighbors within radius.

**Parameters**
**k** – the number of nearest neighbors to return

#### 2.4.5.8.2 Continuous Space

A Continuous Space class.

**class ContinuousSpace**(*dimensions: ArrayLike*, *torus: bool = False*, *random: Random | None = None*, *n_agents: int = 100*)

Continuous space where each agent can have an arbitrary position.

Create a new continuous space.

**Parameters**

- **dimensions** – a numpy array like object where each row specifies the minimum and maximum value of that dimension.
- **torus** – boolean for whether the space wraps around or not
- **random** – a seeded stdlib random.Random instance
- **n_agents** – the expected number of agents in the space

Internally, a numpy array is used to store the positions of all agents. This is resized if needed, but you can control the initial size explicitly by passing n_agents.

**property agents: *AgentSet***

Return an AgentSet with the agents in the space.

**calculate_difference_vector**(*point: ndarray*, *agents=None*) → ndarray

Calculate the difference vector between the point and all agenents.

**Parameters**

- **point** – the point to calculate the difference vector for
- **agents** – the agents to calculate the difference vector of point with. By default, all agents are considered.

**calculate_distances**(*point: ArrayLike*, *agents: Iterable[Agent] | None = None*, ***kwargs*) → tuple[ndarray, list]

Calculate the distance between the point and all agents.

**Parameters**

- **point** – the point to calculate the difference vector for
- **agents** – the agents to calculate the difference vector of point with. By default, all agents are considered.
- **kwargs** – any additional keyword arguments are passed to scipy's cdist, which is used only if torus is False. This allows for non-Euclidian distance measures.

**get_agents_in_radius**(*point: ArrayLike*, *radius: float | int = 1*) → tuple[list, ndarray]

Return the agents and their distances within a radius for the point.

**get_k_nearest_agents**(*point: ArrayLike*, *k: int = 1*) → tuple[list, ndarray]

Return the k nearest agents and their distances to the point.

### Notes

This method returns exactly k agents, ignoring ties. In case of ties, the earlier an agent is inserted the higher it will rank.

If fewer than k agents are present in the space, all agents are returned and a UserWarning is emitted to indicate that the requested k could not be satisfied. If the space is empty or k <= 0, an empty result is returned without a warning.

**in_bounds**(*point: ArrayLike*) → bool

Check if point is inside the bounds of the space.

**torus_correct**(*point: ArrayLike*) → ndarray

Apply a torus correction to the point.

Continuous space agents.

**class HasPositionProtocol**(**args*, ***kwargs*)

Protocol for continuous space position holders.

**class ContinuousSpaceAgent**(*space: ContinuousSpace*, *model*)

A continuous space agent.

**space**

the continuous space in which the agent is located

**Type**
*ContinuousSpace*

**position**

the position of the agent

**Type**
np.ndarray

Initialize a continuous space agent.

**Parameters**

- **space** – the continuous space in which the agent is located
- **model** – the model to which the agent belongs

**property position: ndarray**

Position of the agent.

**remove**() → None

Remove and delete the agent from the model and continuous space.

**get_neighbors_in_radius**(*radius: float | int = 1*) → tuple[list, ndarray]

Get neighbors within radius.

**Parameters**
**radius** – radius within which to look for neighbors

**get_nearest_neighbors**(*k: int = 1*) → tuple[list, ndarray]

Get neighbors within radius.

**Parameters**
**k** – the number of nearest neighbors to return

#### 2.4.5.8.3 Scenarios

Base Scenario class.

**rescale_samples**(*samples: ndarray*, *ranges: ndarray*, *, *inplace: bool = False*) → ndarray

Rescale samples from the unit interval [0, 1] to parameter ranges.

**Parameters**

- **samples** (*ndarray (n, d)*) – Samples drawn from the unit interval.
- **ranges** (*ndarray (d, 2)*) – Parameter ranges given as [[min, max], ...].
- **inplace** (*bool, optional*) – If True, the input `samples` array is modified in place. If False (default), a new array containing the rescaled samples is returned.
- **Returns**
- **-------**
- **(n** (*ndarray*) – Rescaled samples.
- **d)** – Rescaled samples.
- **Notes**
- **-----**
- **inplace=True** (*The rescaling is performed using NumPy broadcasting. If*)

:param : :param the original `samples` array is overwritten.:

**class Scenario**(*, *rng: Generator | BitGenerator | int | integer | Sequence[int] | SeedSequence | None = None*, *scenario_id: int | None = None*, *replication_id: int | None = None*, ***kwargs*)

A Scenario class for defining model parameters and experiments.

Supports both simple instantiation and type-hinted subclassing:

# Simple usage scenario = Scenario(rng=42, density=0.8, minority_pc=0.5)

# Type-hinted subclass (recommended for complex models) class MyScenario(Scenario):

citizen_density: float = 0.7 cop_vision: int = 7 movement: bool = True

scenario = MyScenario(rng=42, cop_vision=10) # Override defaults

**scenario_id**

A unique identifier for this scenario, auto-generated starting from 0

**experiment_id**

Identifies the design point (e.g., row in a QMC sample matrix)

**replication_id**

Identifies the stochastic replication within a design point

**rng**

Random number generator seed value

### Notes

All parameters are accessible via attribute access (scenario.param). Class-level attributes in subclasses serve as default values. Scenario instances are frozen after initialisation; parameters cannot be modified. To create replications with derived seeds, use replicate().

Initialize a Scenario.

**Parameters**

- **rng** – Seed for the random number generator. Accepts any value accepted by numpy.random.default_rng(). scenario.rng is always a Generator after initialisation. The initial rng state is stored in scenario.initial_rng_state and used by spawn_replications() to derive child seeds.
- **scenario_id** – Index of the design point in the experiment matrix.
- **replication_id** – Index of the stochastic replication for this design point.
- ****kwargs** – All other scenario parameters (override class-level defaults).

**to_dict**() → dict[str, Any]

Return dict representation of the scenario.

**spawn_replications**(*n: int*) → list[*Scenario*]

Spawn n replications of this scenario with deterministically derived seeds.

Each replication has identical user provided parameters but a unique random number generator and replication_id. The rng is spawned from the original rng of the base scenario instance.

**Parameters**
**n** – Number of replications to create.

**Returns**
A list of n Scenario instances with replication_id 0..n-1.

**classmethod from_dataframe**(*experiments: DataFrame*, *, *rng: int | integer | Sequence[int] | SeedSequence | None = None*, *replications: int | None = None*) → list[*Scenario*]

Turn a dataframe into a list of scenarios.

**Parameters**

- **experiments** – Dataframe containing the parameters for the scenarios.
- **rng** – the number of random seeds to use or a list of seeds.

- **replications** – the number of replications to create for each scenario

**Returns**
a list of scenario instances

If rng is an integer, numpy will be used to generate that many seed values.

**classmethod from_ndarray**(*experiments: ndarray*, *parameter_names: list[str]*, ***, *rng: int | integer | Sequence[int] | SeedSequence | None = None*, *replications: int | None = None*) → list[*Scenario*]

Turn a numpy array into a list of scenarios.

**Parameters**

- **experiments** – Dataframe containing the parameters for the scenarios.
- **parameter_names** – the names of the parameters
- **rng** – the number of random seeds to use or a list of seeds.
- **replications** – the number of replications to create for each scenario

**Returns**
a list of scenario instances

If rng is an integer, numpy will be used to generate that many seed values.

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## d

datacollection, 243

## e

experimental.continuous_space.continuous_space, 276
experimental.continuous_space.continuous_space_agents, 277
experimental.scenarios.scenario, 278

## m

mesa.agentset, 202
mesa.discrete_space.__init__, 218
mesa.discrete_space.cell, 230
mesa.discrete_space.cell_agent, 231
mesa.discrete_space.cell_collection, 233
mesa.discrete_space.discrete_space, 235
mesa.discrete_space.grid, 237
mesa.discrete_space.network, 241
mesa.discrete_space.voronoi, 242
mesa.time, 213
mesa.visualization.backends.__init__, 261
mesa.visualization.backends.abstract_renderer, 264
mesa.visualization.backends.altair_backend, 265
mesa.visualization.backends.matplotlib_backend, 266
mesa.visualization.command_console, 256
mesa.visualization.components.__init__, 246
mesa.visualization.components.altair_components, 255
mesa.visualization.components.matplotlib_components, 251
mesa.visualization.components.portrayal_components, 259
mesa.visualization.mpl_space_drawing, 252
mesa.visualization.solara_viz, 245
mesa.visualization.space_drawers, 270
mesa.visualization.space_renderer, 268
mesa.visualization.user_param, 250

# INDEX

## Symbols


_try_random (*Grid attribute*), 224, 237


## A


AbstractAgentSet (*class in mesa.agentset*), 202
AbstractRenderer (*class in mesa.visualization.backends.abstract_renderer*), 264
add() (*AbstractAgentSet method*), 205
add() (*AgentSet method*), 208
add_agent() (*Cell method*), 219, 231
add_cell() (*DiscreteSpace method*), 222, 236
add_cell() (*Network method*), 228, 241
add_connection() (*DiscreteSpace method*), 223, 236
add_connection() (*Network method*), 228, 242
add_event() (*EventList method*), 215
add_point() (*Delaunay method*), 242
add_property_layer() (*Grid method*), 225, 238
add_table_row() (*DataCollector method*), 244
Agent (*class in mesa.agent*), 200
agent_types (*Model property*), 196
AgentPortrayalStyle (*class in mesa.visualization.components.__init__*), 246
AgentPortrayalStyle (*class in mesa.visualization.components.portrayal_components*), 259
agents (*Cell attribute*), 218, 230
agents (*Cell property*), 219, 231
agents (*CellCollection attribute*), 220, 233
agents (*ContinuousSpace property*), 274, 276
agents (*DiscreteSpace property*), 222, 236
agents (*Model property*), 195
agents_by_type (*Model property*), 196
AgentSet (*class in mesa.agentset*), 206
agg() (*AbstractAgentSet method*), 203
agg() (*AgentSet method*), 208
agg() (*GroupBy method*), 212
all_cells (*DiscreteSpace attribute*), 222, 235
all_cells (*DiscreteSpace property*), 224, 237
alpha (*AgentPortrayalStyle attribute*), 247, 259
alpha (*PropertyLayerStyle attribute*), 248, 260
AltairBackend (*class in mesa.visualization.backends.__init__*), 261
AltairBackend (*class in mesa.visualization.backends.altair_backend*), 265
as_integer_ratio() (*Priority method*), 217


## B


BaseSpaceDrawer (*class in mesa.visualization.space_drawers*), 270
BasicMovement (*class in mesa.discrete_space.cell_agent*), 232
batch() (*Model method*), 197
bit_count() (*Priority method*), 216
bit_length() (*Priority method*), 216
buffer (*ConsoleManager attribute*), 258


## C


calculate_difference_vector() (*ContinuousSpace method*), 275, 276
calculate_distances() (*ContinuousSpace method*), 275, 277
cancel() (*Event method*), 214
cancel_action() (*Agent method*), 202
canvas (*SpaceRenderer property*), 270
capacity (*Cell attribute*), 218, 230
capacity (*DiscreteSpace attribute*), 222, 235
capacity (*Grid attribute*), 224, 237
CaptureOutput (*class in mesa.visualization.command_console*), 257
cell (*CellAgent attribute*), 220, 232
Cell (*class in mesa.discrete_space.__init__*), 218
Cell (*class in mesa.discrete_space.cell*), 230
cell (*FixedCell property*), 232
cell_klass (*DiscreteSpace attribute*), 222, 235
CellAgent (*class in mesa.discrete_space.__init__*), 220
CellAgent (*class in mesa.discrete_space.cell_agent*), 232
CellCollection (*class in mesa.discrete_space.__init__*), 220
CellCollection (*class in mesa.discrete_space.cell_collection*), 233

cells (*CellCollection attribute*), 220, 233
cells_with_capacity (*Grid property*), 226, 239
chart_property_layers() (*in module mesa.visualization.components.altair_components*), 255
check_param_is_fixed() (*in module mesa.visualization.solara_viz*), 246
clear() (*AbstractAgentSet method*), 206
clear() (*AgentSet method*), 209
clear() (*EventList method*), 216
clear_all_class_subscriptions() (*Model class method*), 197
clear_all_subscriptions() (*Model method*), 197
clear_console() (*ConsoleManager method*), 258
collect() (*DataCollector method*), 244
collect_agent_data() (*AbstractRenderer method*), 264
collect_agent_data() (*AltairBackend method*), 261, 265
collect_agent_data() (*in module mesa.visualization.mpl_space_drawing*), 252
collect_agent_data() (*MatplotlibBackend method*), 263, 267
color (*AgentPortrayalStyle attribute*), 247, 259
color (*PropertyLayerStyle attribute*), 248, 260
colorbar (*PropertyLayerStyle attribute*), 248, 260
colormap (*PropertyLayerStyle attribute*), 248, 260
command (*ConsoleEntry attribute*), 256
compact() (*EventList method*), 216
conjugate() (*Priority method*), 216
connect() (*Cell method*), 219, 230
console (*ConsoleManager attribute*), 258
ConsoleEntry (*class in mesa.visualization.command_console*), 256
ConsoleManager (*class in mesa.visualization.command_console*), 257
ContinuousSpace (*class in experimental.continuous_space.continuous_space*), 274, 276
ContinuousSpaceAgent (*class in experimental.continuous_space.continuous_space_agents*), 275, 277
ContinuousSpaceDrawer (*class in mesa.visualization.space_drawers*), 272
coordinate (*Cell attribute*), 218, 230
copy_renderer() (*in module mesa.visualization.solara_viz*), 246
count (*Schedule attribute*), 218
count() (*AgentSet method*), 209
count() (*GroupBy method*), 212
create_agents() (*Agent class method*), 201
create_property_layer() (*Grid method*), 225, 238
create_space_component() (*in module mesa.visualization.solara_viz*), 246

## D

datacollection module, 243
DataCollector (*class in datacollection*), 243
Delaunay (*class in mesa.discrete_space.voronoi*), 242
denominator (*Priority attribute*), 217
deregister_agent() (*Model method*), 197
dimensions (*Grid attribute*), 224, 237
discard() (*AbstractAgentSet method*), 205
discard() (*AgentSet method*), 208
disconnect() (*Cell method*), 219, 231
DiscreteSpace (*class in mesa.discrete_space.__init__*), 221
DiscreteSpace (*class in mesa.discrete_space.discrete_space*), 235
do() (*AbstractAgentSet method*), 206
do() (*AgentSet method*), 207
do() (*GroupBy method*), 212
draw_agents() (*AbstractRenderer method*), 264
draw_agents() (*AltairBackend method*), 262, 265
draw_agents() (*MatplotlibBackend method*), 263, 267
draw_agents() (*SpaceRenderer method*), 269
draw_altair() (*ContinuousSpaceDrawer method*), 273
draw_altair() (*HexSpaceDrawer method*), 271
draw_altair() (*NetworkSpaceDrawer method*), 272
draw_altair() (*OrthogonalSpaceDrawer method*), 271
draw_altair() (*VoronoiSpaceDrawer method*), 274
draw_continuous_space() (*in module mesa.visualization.mpl_space_drawing*), 254
draw_hex_grid() (*in module mesa.visualization.mpl_space_drawing*), 253
draw_matplotlib() (*ContinuousSpaceDrawer method*), 273
draw_matplotlib() (*HexSpaceDrawer method*), 271
draw_matplotlib() (*NetworkSpaceDrawer method*), 272
draw_matplotlib() (*OrthogonalSpaceDrawer method*), 270
draw_matplotlib() (*VoronoiSpaceDrawer method*), 273
draw_network() (*in module mesa.visualization.mpl_space_drawing*), 253
draw_orthogonal_grid() (*in module mesa.visualization.mpl_space_drawing*), 253
draw_property_layer() (*AbstractRenderer method*), 264
draw_property_layer() (*AltairBackend method*), 262, 266

draw_property_layer() (*MatplotlibBackend method*), 263, 267
draw_property_layer() (*SpaceRenderer method*), 269
draw_property_layers() (*in module mesa.visualization.mpl_space_drawing*), 253
draw_space() (*in module mesa.visualization.mpl_space_drawing*), 252
draw_structure() (*AbstractRenderer method*), 264
draw_structure() (*AltairBackend method*), 261, 265
draw_structure() (*MatplotlibBackend method*), 263, 266
draw_structure() (*SpaceRenderer method*), 269
draw_voronoi_grid() (*in module mesa.visualization.mpl_space_drawing*), 254

## E

edgecolors (*AgentPortrayalStyle attribute*), 247, 260
empties (*DiscreteSpace attribute*), 222, 235
empties (*DiscreteSpace property*), 224, 237
end (*Schedule attribute*), 218
Event (*class in mesa.time*), 213
EventGenerator (*class in mesa.time*), 214
EventList (*class in mesa.time*), 215
execute() (*Event method*), 214
execute_code() (*ConsoleManager method*), 258
execution_count (*EventGenerator property*), 215
experiment_id (*Scenario attribute*), 279
experimental.continuous_space.continuous_space module, 274, 276
experimental.continuous_space.continuous_space_agents module, 275, 277
experimental.scenarios.scenario module, 278
export_triangles() (*Delaunay method*), 242
export_voronoi_regions() (*Delaunay method*), 242

## F

find_nearest_cell() (*DiscreteSpace method*), 222, 236
find_nearest_cell() (*Grid method*), 225, 238
find_nearest_cell() (*HexGrid method*), 227, 241
find_nearest_cell() (*Network method*), 228, 241
find_nearest_cell() (*VoronoiGrid method*), 229, 243
FixedAgent (*class in mesa.discrete_space.__init__*), 224
FixedAgent (*class in mesa.discrete_space.cell_agent*), 232
FixedCell (*class in mesa.discrete_space.cell_agent*), 232
fn (*Event attribute*), 213
format_command_html() (*in module mesa.visualization.command_console*), 258
format_output_html() (*in module mesa.visualization.command_console*), 258
from_bytes() (*Priority class method*), 217
from_dataframe() (*Agent class method*), 201
from_dataframe() (*Scenario class method*), 279
from_ndarray() (*Scenario class method*), 280
function (*EventGenerator attribute*), 214

## G

get() (*AbstractAgentSet method*), 204
get() (*AgentSet method*), 209
get() (*Slider method*), 251
get_agent_vars_dataframe() (*DataCollector method*), 245
get_agents_in_radius() (*ContinuousSpace method*), 275, 277
get_agenttype_vars_dataframe() (*DataCollector method*), 245
get_entries() (*ConsoleManager method*), 258
get_group() (*GroupBy method*), 211
get_k_nearest_agents() (*ContinuousSpace method*), 275, 277
get_model_vars_dataframe() (*DataCollector method*), 244
get_nearest_neighbors() (*ContinuousSpaceAgent method*), 276, 278
get_neighborhood() (*Cell method*), 220, 231
get_neighborhood_mask() (*Grid method*), 225, 238
get_neighbors_in_radius() (*ContinuousSpaceAgent method*), 276, 278
get_output() (*CaptureOutput method*), 257
get_table_dataframe() (*DataCollector method*), 245
get_viz_limits() (*BaseSpaceDrawer method*), 270
Grid (*class in mesa.discrete_space.__init__*), 224
Grid (*class in mesa.discrete_space.grid*), 237
Grid2DMovingAgent (*class in mesa.discrete_space.__init__*), 226
Grid2DMovingAgent (*class in mesa.discrete_space.cell_agent*), 233
GroupBy (*class in mesa.agentset*), 211
groupby() (*AbstractAgentSet method*), 205
groupby() (*AgentSet method*), 209
groups (*GroupBy attribute*), 211

## H

HasCell (*class in mesa.discrete_space.cell_agent*), 232
HasCellProtocol (*class in mesa.discrete_space.cell_agent*), 231
HasPositionProtocol (*class in experimental.continuous_space.continuous_space_agents*), 275, 277
height (*Grid property*), 225, 238

HexGrid (*class in mesa.discrete_space.__init__*), 227
HexGrid (*class in mesa.discrete_space.grid*), 240
HexSpaceDrawer (*class in mesa.visualization.space_drawers*), 271
history (*ConsoleManager attribute*), 258

## I

imag (*Priority attribute*), 217
in_bounds() (*ContinuousSpace method*), 275, 277
index() (*AgentSet method*), 210
initialize_canvas() (*AbstractRenderer method*), 264
initialize_canvas() (*AltairBackend method*), 261, 265
initialize_canvas() (*MatplotlibBackend method*), 262, 266
InteractiveConsole (*class in mesa.visualization.command_console*), 257
interrupt_for() (*Agent method*), 202
interval (*Schedule attribute*), 217
is_active (*EventGenerator property*), 214
is_busy (*Agent property*), 202
is_continuation (*ConsoleEntry attribute*), 256
is_empty (*Cell property*), 219, 231
is_empty() (*EventList method*), 216
is_error (*ConsoleEntry attribute*), 256
is_full (*Cell property*), 219, 231
is_integer() (*Priority method*), 217
isdisjoint() (*AbstractAgentSet method*), 206
isdisjoint() (*AgentSet method*), 210

## L

linewidths (*AgentPortrayalStyle attribute*), 247, 260
locals_dict (*ConsoleManager attribute*), 257

## M

make_altair_plot_component() (*in module mesa.visualization.components.altair_components*), 255
make_altair_space() (*in module mesa.visualization.components.__init__*), 248
make_altair_space() (*in module mesa.visualization.components.altair_components*), 255
make_initial_grid_layout() (*in module mesa.visualization.solara_viz*), 246
make_mpl_plot_component() (*in module mesa.visualization.components.__init__*), 249
make_mpl_plot_component() (*in module mesa.visualization.components.matplotlib_components*), 251
make_mpl_space_component() (*in module mesa.visualization.components.__init__*), 249
make_mpl_space_component() (*in module mesa.visualization.components.matplotlib_components*), 251
make_plot_component() (*in module mesa.visualization.components.__init__*), 249
make_plot_measure() (*in module mesa.visualization.components.matplotlib_components*), 251
make_space_altair() (*in module mesa.visualization.components.altair_components*), 255
make_space_component() (*in module mesa.visualization.components.__init__*), 250
make_space_matplotlib() (*in module mesa.visualization.components.matplotlib_components*), 251
map() (*AbstractAgentSet method*), 206
map() (*AgentSet method*), 207
map() (*GroupBy method*), 211
marker (*AgentPortrayalStyle attribute*), 247, 259
MatplotlibBackend (*class in mesa.visualization.backends.__init__*), 262
MatplotlibBackend (*class in mesa.visualization.backends.matplotlib_backend*), 266
maybe_raise_error() (*UserParam method*), 250
mesa.agent
    module, 200
mesa.agentset
    module, 202
mesa.discrete_space.__init__
    module, 218
mesa.discrete_space.cell
    module, 230
mesa.discrete_space.cell_agent
    module, 231
mesa.discrete_space.cell_collection
    module, 233
mesa.discrete_space.discrete_space
    module, 235
mesa.discrete_space.grid
    module, 237
mesa.discrete_space.network
    module, 241
mesa.discrete_space.voronoi
    module, 242
mesa.model
    module, 194
mesa.time

module, 213
mesa.visualization.backends.__init__
module, 261
mesa.visualization.backends.abstract_renderer
module, 264
mesa.visualization.backends.altair_backend
module, 265
mesa.visualization.backends.matplotlib_backend
module, 266
mesa.visualization.command_console
module, 256
mesa.visualization.components.__init__
module, 246
mesa.visualization.components.altair_components
module, 255
mesa.visualization.components.matplotlib_components
module, 251
mesa.visualization.components.portrayal_components
module, 259
mesa.visualization.mpl_space_drawing
module, 252
mesa.visualization.solara_viz
module, 245
mesa.visualization.space_drawers
module, 270
mesa.visualization.space_renderer
module, 268
mesa.visualization.user_param
module, 250
model (*AbstractAgentSet attribute*), 202
model (*Agent attribute*), 200
Model (*class in mesa.model*), 194
model (*EventGenerator attribute*), 214
module
datacollection, 243
experimental.continuous_space.continuous_space, 274, 276
experimental.continuous_space.continuous_space_agents, 275, 277
experimental.scenarios.scenario, 278
mesa.agent, 200
mesa.agentset, 202
mesa.discrete_space.__init__, 218
mesa.discrete_space.cell, 230
mesa.discrete_space.cell_agent, 231
mesa.discrete_space.cell_collection, 233
mesa.discrete_space.discrete_space, 235
mesa.discrete_space.grid, 237
mesa.discrete_space.network, 241
mesa.discrete_space.voronoi, 242
mesa.model, 194
mesa.time, 213
mesa.visualization.backends.__init__, 261
mesa.visualization.backends.abstract_renderer, 264
mesa.visualization.backends.altair_backend, 265
mesa.visualization.backends.matplotlib_backend, 266
mesa.visualization.command_console, 256
mesa.visualization.components.__init__, 246
mesa.visualization.components.altair_components, 255
mesa.visualization.components.matplotlib_components, 251
mesa.visualization.components.portrayal_components, 259
mesa.visualization.mpl_space_drawing, 252
mesa.visualization.solara_viz, 245
mesa.visualization.space_drawers, 270
mesa.visualization.space_renderer, 268
mesa.visualization.user_param, 250
move() (*Grid2DMovingAgent method*), 227, 233
move_relative() (*BasicMovement method*), 232
move_to() (*BasicMovement method*), 232

## N

neighborhood (*Cell property*), 219, 231
Network (*class in mesa.discrete_space.__init__*), 227
Network (*class in mesa.discrete_space.network*), 241
NetworkSpaceDrawer (*class in mesa.visualization.space_drawers*), 272
next_command() (*ConsoleManager method*), 258
next_scheduled_time (*EventGenerator property*), 215
notify() (*Model method*), 198
numerator (*Priority attribute*), 217

## O

observe() (*Model method*), 198
observe_class() (*Model class method*), 198
OrthogonalMooreGrid (*class in mesa.discrete_space.__init__*), 228
OrthogonalMooreGrid (*class in mesa.discrete_space.grid*), 239
OrthogonalSpaceDrawer (*class in mesa.visualization.space_drawers*), 270
OrthogonalVonNeumannGrid (*class in mesa.discrete_space.__init__*), 228
OrthogonalVonNeumannGrid (*class in mesa.discrete_space.grid*), 240
output (*ConsoleEntry attribute*), 256

## P

pause() (*EventGenerator method*), 215
peek_ahead() (*EventList method*), 215

pickle_gridcell() (*in module mesa.discrete_space.grid*), 237
pop() (*AbstractAgentSet method*), 206
pop() (*AgentSet method*), 210
pop_event() (*EventList method*), 215
position (*Cell attribute*), 218, 230
position (*Cell property*), 219, 230
position (*ContinuousSpaceAgent attribute*), 275, 277
position (*ContinuousSpaceAgent property*), 276, 278
post_process (*SpaceRenderer property*), 270
prev_command() (*ConsoleManager method*), 258
Priority (*class in mesa.time*), 216
priority (*Event attribute*), 213
priority (*EventGenerator attribute*), 214
project() (*ContinuousSpaceDrawer method*), 273
property_layers (*DiscreteSpace attribute*), 222, 235
PropertyLayerStyle (*class in mesa.visualization.components.__init__*), 248
PropertyLayerStyle (*class in mesa.visualization.components.portrayal_components*), 260
push() (*InteractiveConsole method*), 257

## R

random (*Agent property*), 201
random (*AgentSet attribute*), 206
random (*Cell attribute*), 218, 230
random (*CellCollection attribute*), 220, 233
random (*DiscreteSpace attribute*), 222, 235
random (*Grid attribute*), 224, 237
random (*Model attribute*), 195
real (*Priority attribute*), 217
register_agent() (*Model method*), 197
remove() (*AbstractAgentSet method*), 205
remove() (*Agent method*), 200
remove() (*AgentSet method*), 208
remove() (*CellAgent method*), 220, 232
remove() (*ContinuousSpaceAgent method*), 276, 278
remove() (*EventList method*), 216
remove() (*FixedAgent method*), 224, 233
remove_agent() (*Cell method*), 219, 231
remove_all_agents() (*Model method*), 198
remove_cell() (*DiscreteSpace method*), 223, 236
remove_cell() (*Network method*), 228, 242
remove_connection() (*DiscreteSpace method*), 223, 236
remove_connection() (*Network method*), 228, 242
remove_property_layer() (*Grid method*), 225, 238
render() (*SpaceRenderer method*), 269
replication_id (*Scenario attribute*), 279
rescale_samples() (*in module experimental.scenarios.scenario*), 278
resume() (*EventGenerator method*), 215
rng (*Agent property*), 201
rng (*Model attribute*), 195
rng (*Scenario attribute*), 279
run_for() (*Model method*), 199
run_model() (*Model method*), 197
run_until() (*Model method*), 200
running (*Model attribute*), 195

## S

scenario (*Agent property*), 202
Scenario (*class in experimental.scenarios.scenario*), 278
scenario (*Model attribute*), 195
scenario (*Model property*), 195
scenario_id (*Scenario attribute*), 279
Schedule (*class in mesa.time*), 217
schedule (*EventGenerator attribute*), 214
schedule_event() (*Model method*), 199
schedule_recurring() (*Model method*), 199
select() (*AbstractAgentSet method*), 203
select() (*AgentSet method*), 210
select() (*CellCollection method*), 221, 234
select_random_agent() (*CellCollection method*), 221, 234
select_random_cell() (*CellCollection method*), 221, 234
select_random_cell_with_capacity() (*Grid method*), 226, 239
select_random_empty_cell() (*DiscreteSpace method*), 224, 237
select_random_empty_cell() (*Grid method*), 226, 239
set() (*AbstractAgentSet method*), 204
set() (*AgentSet method*), 211
set_viz_dims() (*ContinuousSpaceDrawer method*), 273
setup_agents() (*SpaceRenderer method*), 268
setup_property_layer() (*SpaceRenderer method*), 268
setup_structure() (*SpaceRenderer method*), 268
shuffle() (*AbstractAgentSet method*), 206
shuffle() (*AgentSet method*), 207
shuffle_do() (*AbstractAgentSet method*), 206
shuffle_do() (*AgentSet method*), 207
size (*AgentPortrayalStyle attribute*), 247, 259
Slider (*class in mesa.visualization.user_param*), 250
sort() (*AbstractAgentSet method*), 206
sort() (*AgentSet method*), 207
space (*ContinuousSpaceAgent attribute*), 275, 277
SpaceRenderer (*class in mesa.visualization.space_renderer*), 268
spawn_replications() (*Scenario method*), 279
split_model_params() (*in module mesa.visualization.solara_viz*), 246

start (*Schedule attribute*), 217
start() (*EventGenerator method*), 215
start_action() (*Agent method*), 202
step() (*Agent method*), 201
step() (*Model method*), 197
steps (*Model attribute*), 195
stop() (*EventGenerator method*), 215
suppress() (*Model method*), 198

## T

time (*Event attribute*), 213
time (*Model attribute*), 195
to_bytes() (*Priority method*), 216
to_dict() (*Scenario method*), 279
to_list() (*AbstractAgentSet method*), 204
to_list() (*AgentSet method*), 211
tooltip (*AgentPortrayalStyle attribute*), 247, 260
torus (*Grid attribute*), 224, 237
torus_correct() (*ContinuousSpace method*), 275, 277

## U

unique_id (*Agent attribute*), 200
unique_id (*Event attribute*), 213
unobserve() (*Model method*), 198
unobserve_class() (*Model class method*), 199
unpickle_gridcell() (*in module mesa.discrete_space.grid*), 237
update() (*AgentPortrayalStyle method*), 247, 260
UserParam (*class in mesa.visualization.user_param*), 250

## V

vmax (*PropertyLayerStyle attribute*), 248, 260
vmin (*PropertyLayerStyle attribute*), 248, 260
VoronoiGrid (*class in mesa.discrete_space.__init__*), 229
VoronoiGrid (*class in mesa.discrete_space.voronoi*), 242
VoronoiSpaceDrawer (*class in mesa.visualization.space_drawers*), 273

## W

width (*Grid property*), 225, 238

## X

x (*AgentPortrayalStyle attribute*), 247, 259

## Y

y (*AgentPortrayalStyle attribute*), 247, 259

## Z

zorder (*AgentPortrayalStyle attribute*), 247, 259