第1部　モデル作りの基本を身につける

第3章：シミュレーションの準備をする

❐3.0 準備をすると後が楽です

- 飛ぶ鳥のモデルを作り始めます。
- モデルを組み立てるための設定方法を学びます。
- シミュレーションを見るための設定方法を学びます。

❒3.1 シミュレーションの3要素

- シミュレーションのプロセスの3要素
  - 1) シミュレーションするためのモデルを作る
  - 2) モデルを実際に動かす（シミュレーションの実行）
  - 3) 実行の過程・結果を見る（シミュレーションの出力）

  - 各プロセスには以下のような準備が必要
    - ➡ 1) モデルの枠組（土台）の設定
    - ➡ 2) シミュレーションを実行させる環境の設定
    - ➡ 3) 実行過程を見るための出力表示の設定

❒3.2 artisoc Cloudモデルの基本(1)

- artisoc Cloudの基本的な要素
  - 「エージェント」と呼ばれる行動主体
  - 個々のエージェントの属性を表す「変数」
  - エージェントが行動する「ルール」
  - エージェントが行動する「空間」
  - シミュレーションやモデル全体に関わる「変数」や「ルール」

❒3.2 artisoc Cloudモデルの基本(2)

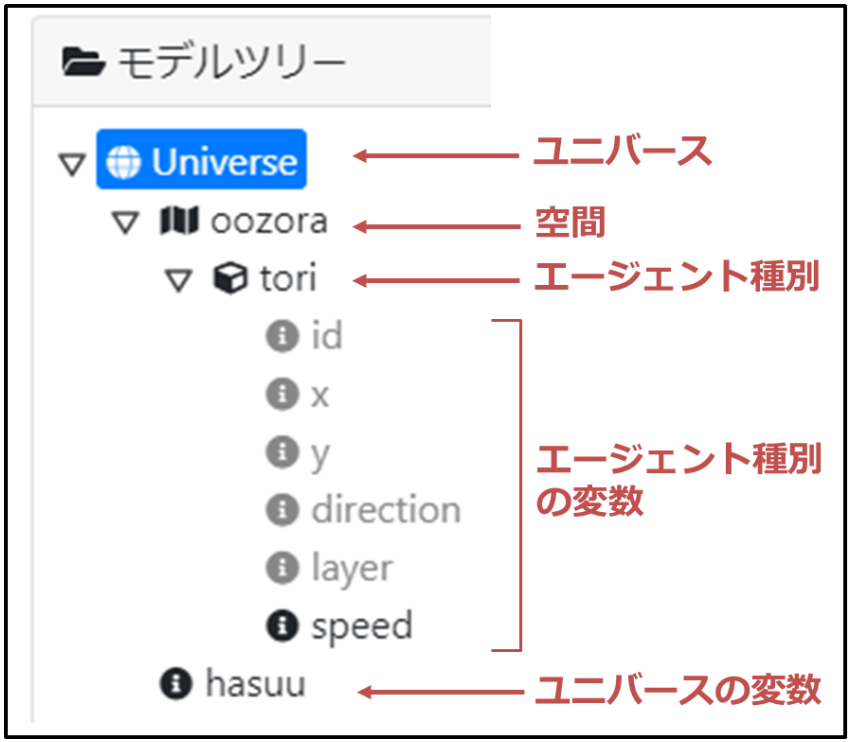
- Universe（全体）の中に、
  エージェントが相互作用する場を作り、
  相互作用のルールを指定する。

  - モデルツリー画面
    - ➡ 3階層構造：
      - ☑「Universe(全体)」
      - ☑「空間」
      - ☑「エージェント」

  - Unverse、空間、エージェントそれぞれに指定できる「変数」

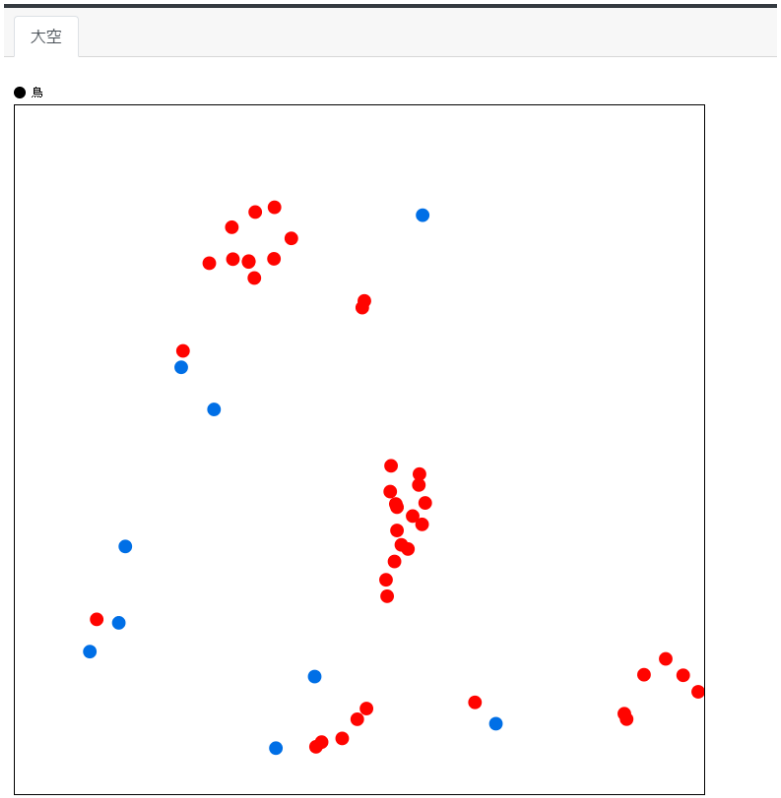
  - Universeとエージェントにシミュレーションのルールを書き込むための「ルールエディタ」

  - ルール



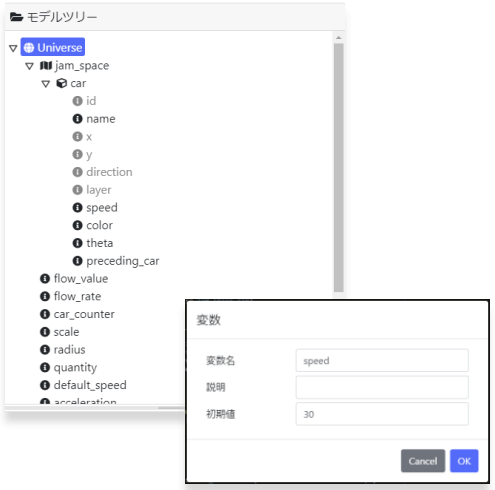
❒3.3 モデルの枠組みを作る(1)

- 概要
  - 『飛ぶ鳥モデル』の作成を通じて、artisoc Cloudの基本的テクニックを習得
    - ➡ 飛ぶ鳥モデル：「大空(oozora)」という空間上に「鳥(tori)」というエージェントが存在し、様々な向きに飛んでいく



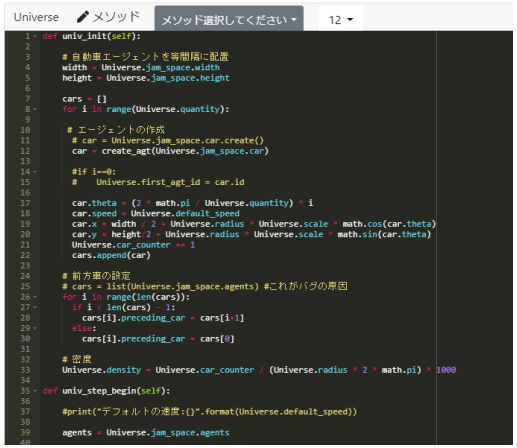
❑3.3 モデルの枠組みを作る(2)　モデル構築の流れ

① 空間／エージェントの
種類・属性を作成

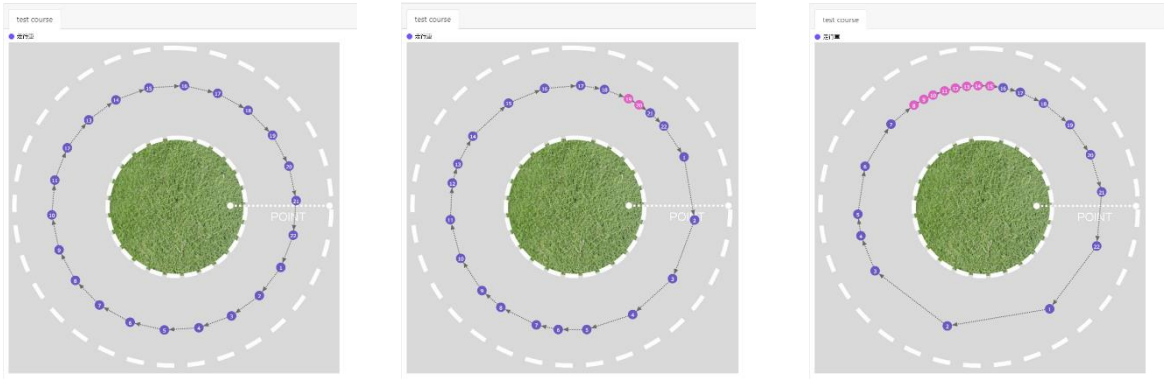


② シミュレーション結果の
出力形式を決定

③ エージェントの
行動ルールを作成

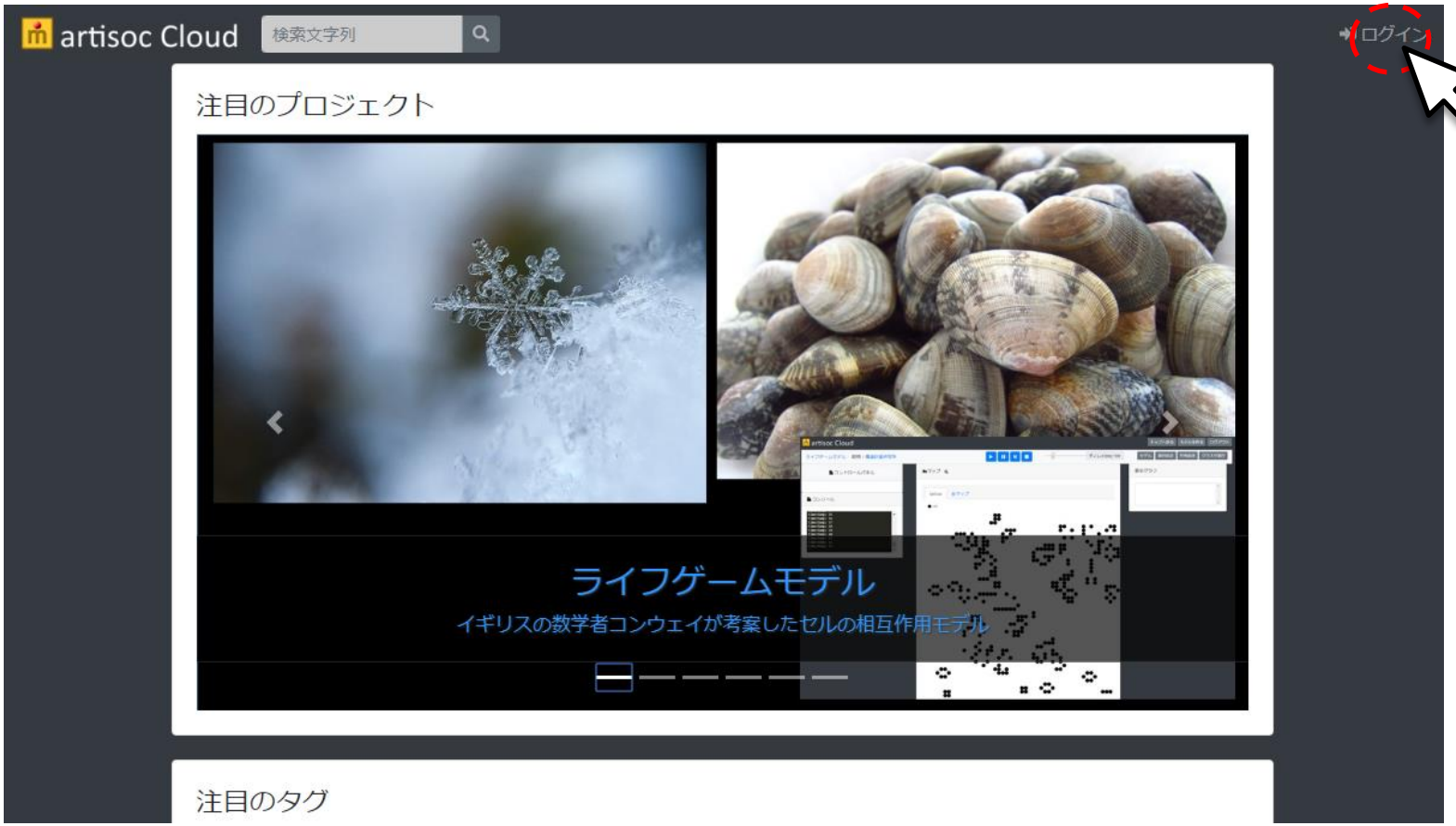


実行ボタンをクリック！



❑3.3 モデルの枠組みを作る(3)　準備

- 下記URLにアクセスし、画面右上からログインします
  - https://artisoc-cloud.kke.co.jp/

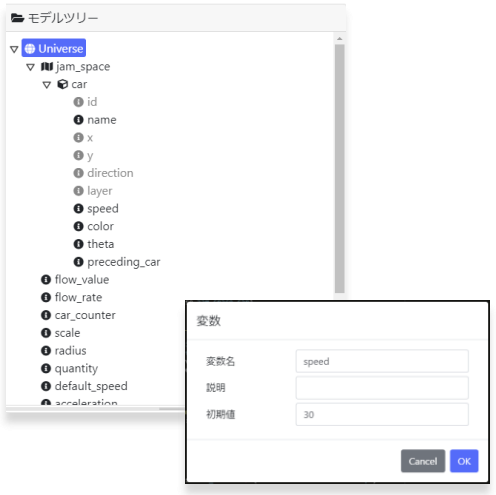


❑3.3 モデルの枠組みを作る(4)　準備

- 右上から「新規モデルの作成」をクリックし基本情報をクリックして、モデル名を入力します。
  - 「飛ぶ鳥モデル」と名付けましょう

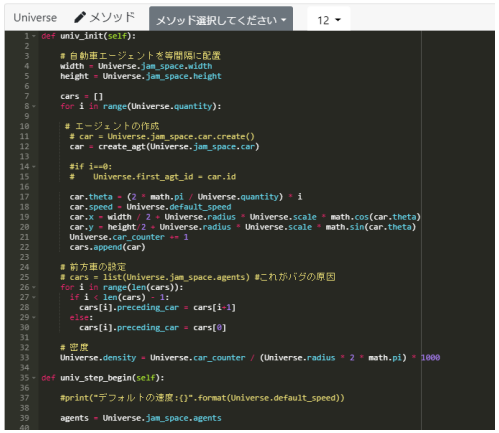
❑3.3 モデルの枠組みを作る(5)

① 空間／エージェントの
種類・属性を作成



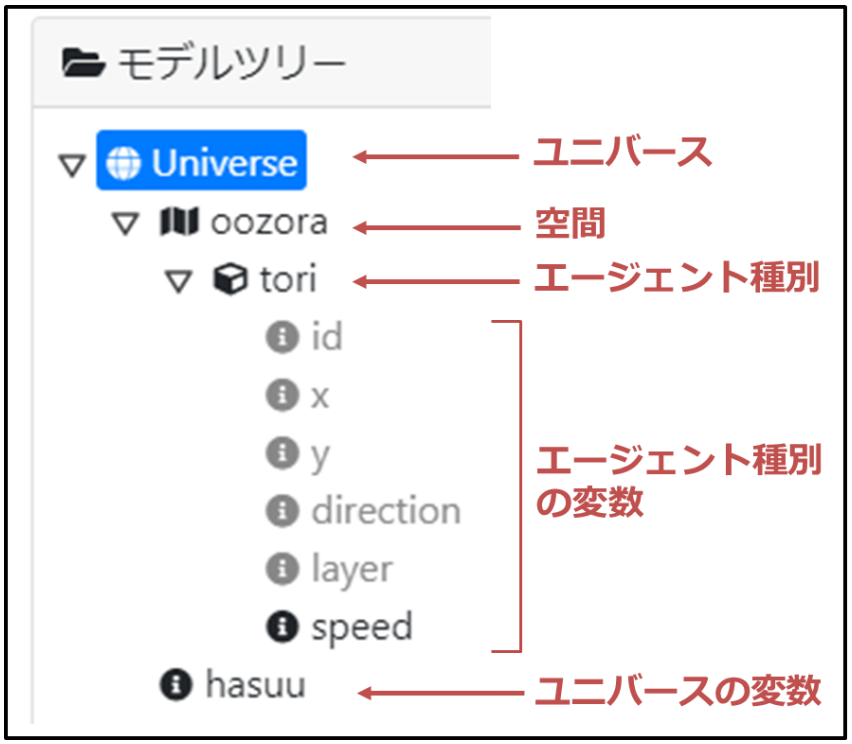
② シミュレーション結果の
出力形式を決定

③ エージェントの
行動ルールを作成



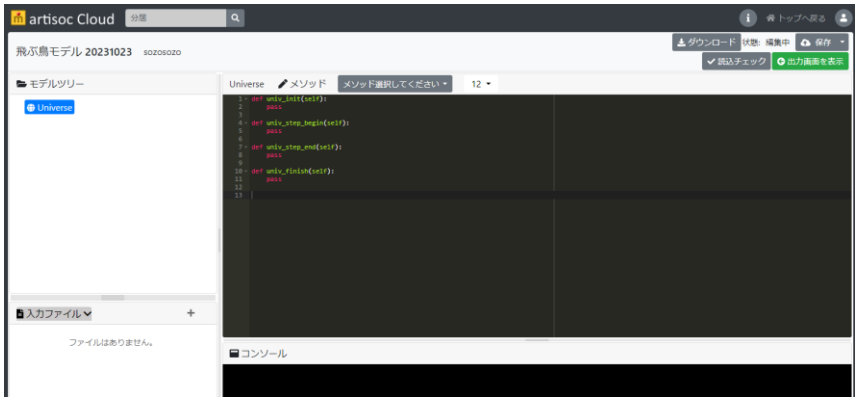
❑3.3 モデルの枠組みを作る(6)　空間/エージェントの種類・属性を作成

- 「モデルツリー」を編集することでモデルの枠組みを構築します
  - 最初はモデル全体を表す「Universe」だけがあります
  - ここに「空間」「エージェント種別」「変数」などを追加していきます



❑3.3 モデルの枠組みを作る(7)　ルール画面の表示

■「ルール画面を表示」をクリックしてモデルのルール画面に遷移します

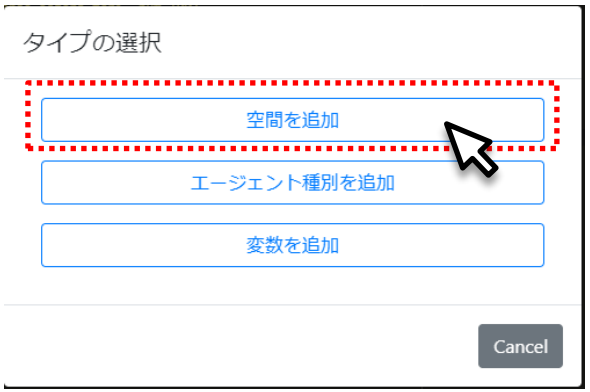


出力画面

ルール画面

❒3.3 モデルの枠組みを作る(8)　空間の定義

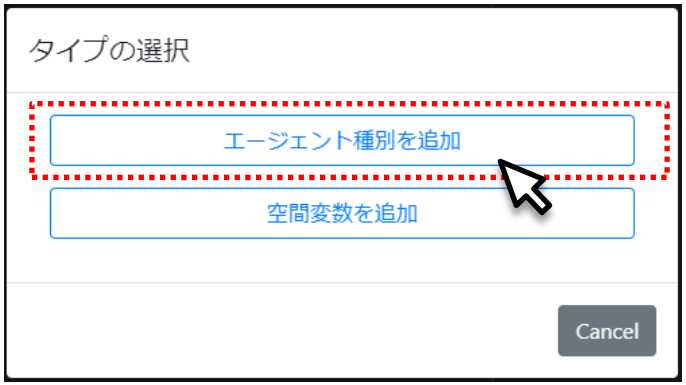
- Universe右に表示される「＋」をクリックして空間を作成
  - 空間名「oozora」・・・・大空
  - その他はデフォルト値
    - 説明は自由に入力

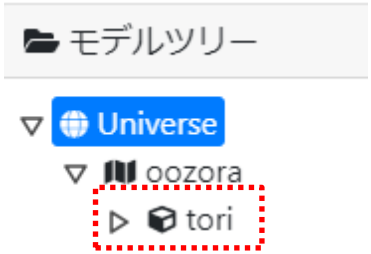


❑3.3 モデルの枠組みを作る(9) エージェント種別の作成

- 空間（oozora）右に表示される「＋」をクリックしてエージェント種別を作成
  - エージェント種別名「tori」・・・鳥エージェント
    - 「説明」は自由に、「記憶数」は0のままでOK

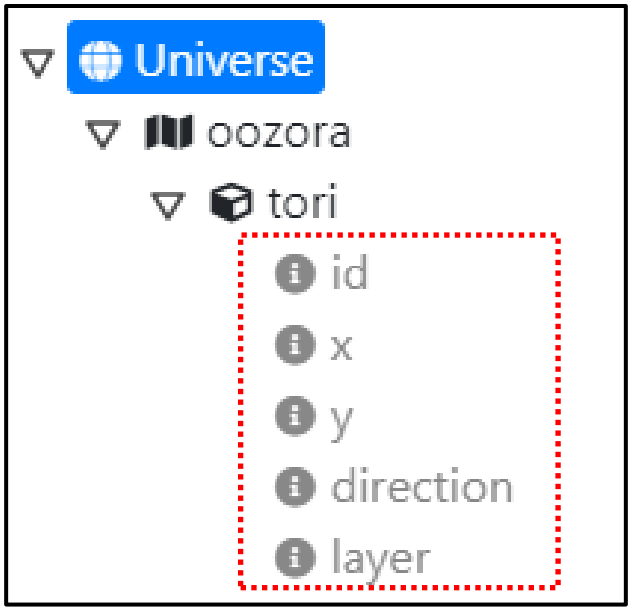
    ※この定義はエージェント種別でエージェントそのものではありません





❑3.3 モデルの枠組みを作る(10) エージェントの変数

- エージェントは以下の変数を持ちます
  - id … 識別番号
  - x … x座標
  - y … y座標
  - direction … 向き
  - layer … 空間レイヤー（今回は使いません）
- 変数はエージェントの性質を表します



❒TIPS：シミュレーションモデルの保存

シミュレーションモデルに
不具合があって、artisoc　Cloudが
停止してしまった‥

ルールを変更したら
動かなくて、元に戻せなく
なった‥

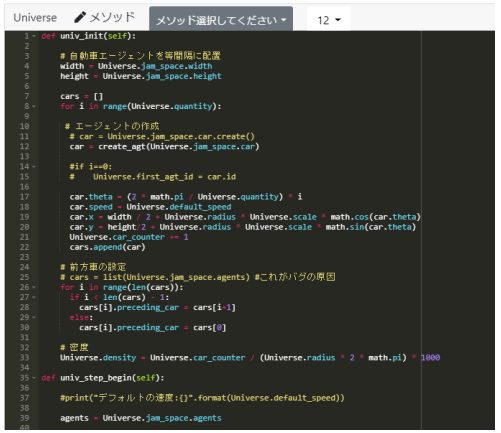
こまめに保存

❑3.4 シミュレーションの過程を見るための準備をする(1)

① 空間／エージェントの
種類・属性を作成

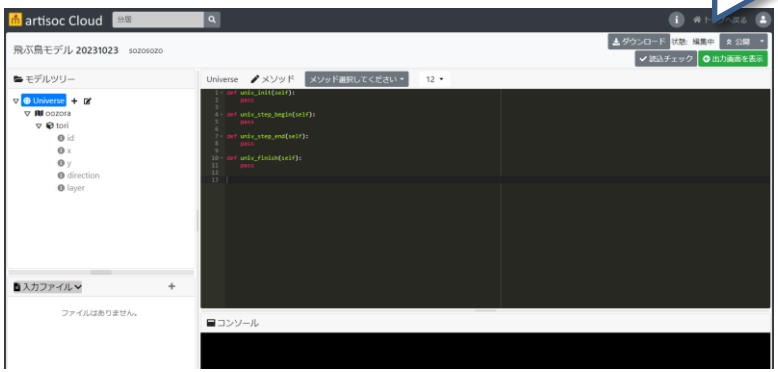
② シミュレーション結果の
出力形式を決定

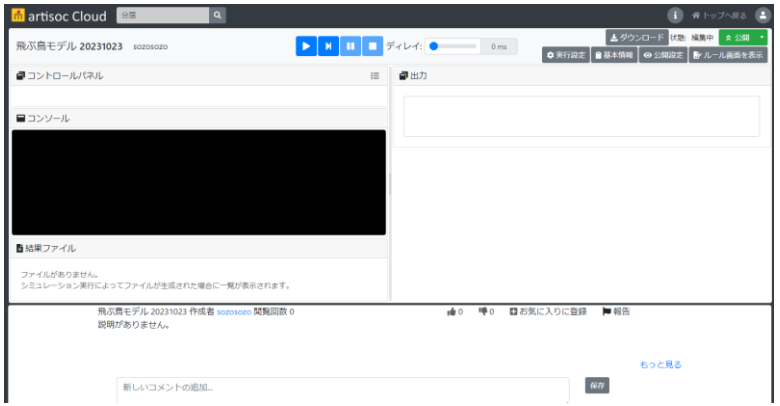
③ エージェントの
行動ルールを作成



❑3.4 シミュレーションの過程を見るための準備をする(2) 出力設定

■「出力画面を表示」をクリックして出力画面へ移動





ルール画面

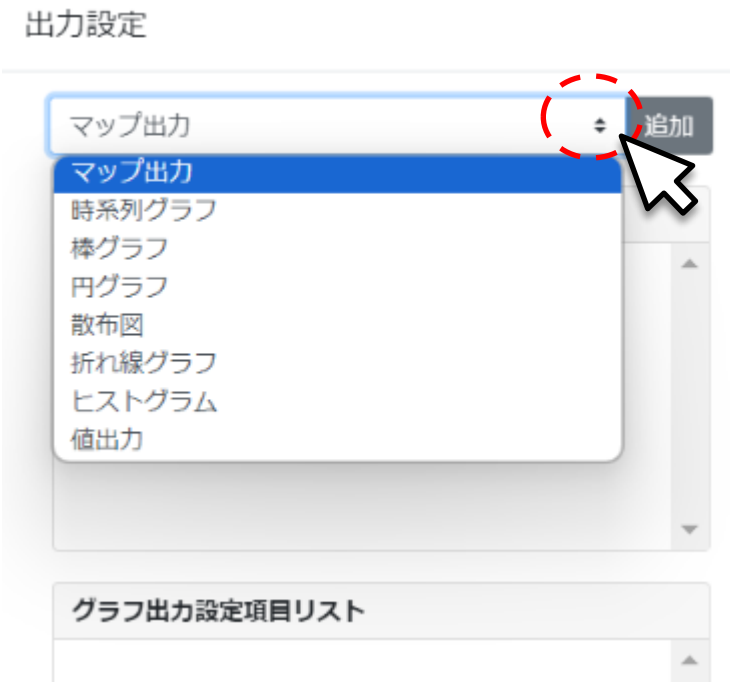
出力画面

❑3.4 シミュレーションの過程を見るための準備をする(3) 出力設定

- 空間とエージェントを「マップ」として出力
  - 下図のようにマップ出力を選択

カーソルをここに合わせると が表示される

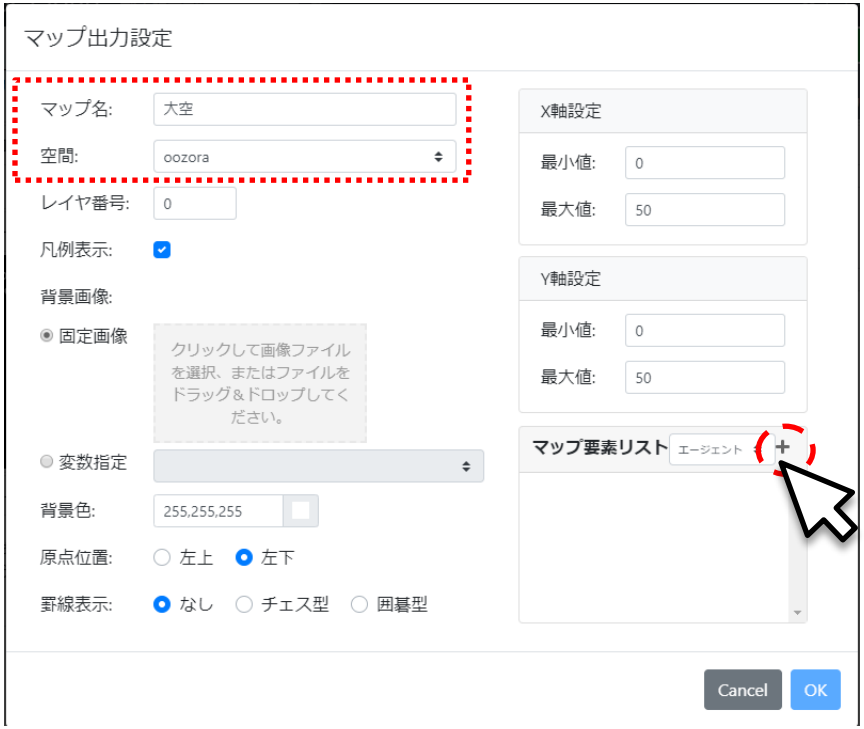
結果の出力の仕方を設定する画面を表示



結果を「マップ」とする出力項目を追加

❑3.4 シミュレーションの過程を見るための準備をする(4) 出力設定

■ 空間とエージェントを「マップ」として出力



マップ名：大空
空間：oozora
→空間oozoraを「大空」という名前で出力

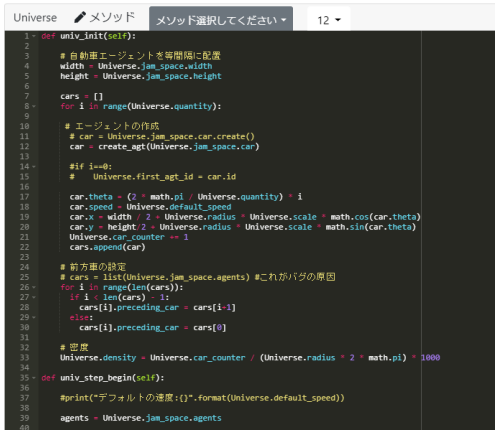
要素名：鳥
エージェント：tori
→エージェントtoriを「鳥」という名前で出力
※エージェントを表示する形状、色なども設定できるが、デフォルトでOK

❑3.4 シミュレーションの過程を見るための準備をする(5)

① 空間／エージェントの
種類・属性を作成

② シミュレーション結果の
出力形式を決定

③ エージェントの
行動ルールを作成

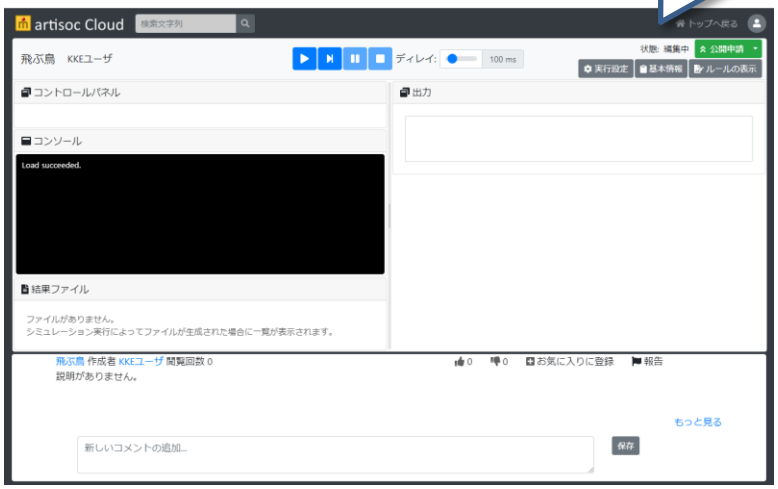


❑3.5 エージェントの行動ルールを作成する(1) ルールを作成

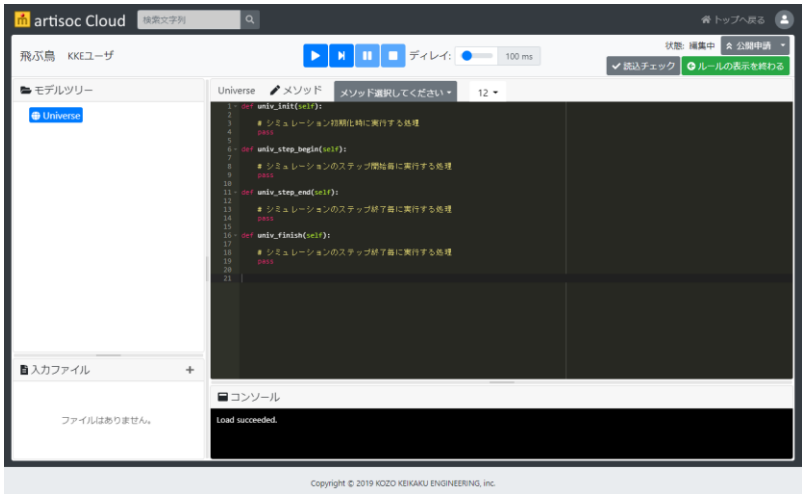
- 以下のようなモデルのルールを作成します
  - シミュレーション開始時に鳥エージェントを1体だけ作成
  - 鳥エージェントの初期位置は空間の中央
  - 鳥エージェントは毎ステップ、前方に1進む

❑3.5 エージェントの行動ルールを作成する(2) ルールを作成

■「ルール画面を表示」をクリックしてモデルのルール画面に遷移します



出力画面



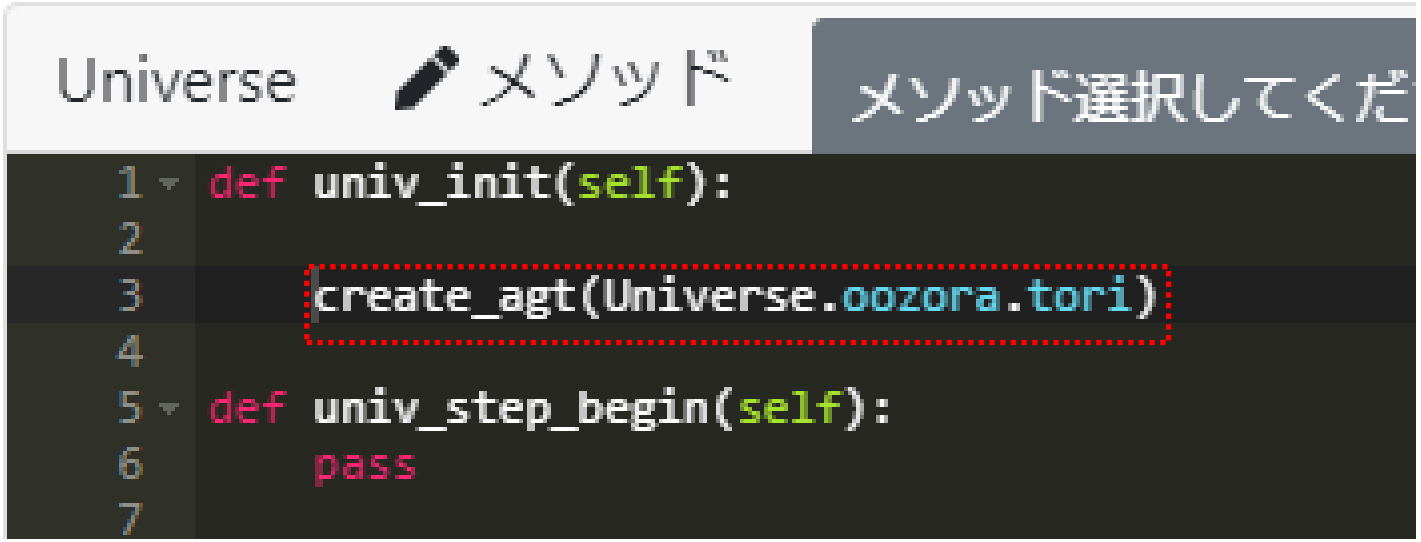
ルール画面

❑3.5 エージェントの行動ルールを作成する(3) エージェントを生成

- シミュレーション開始時に鳥エージェントを1体生成するルールを作成します
  - モデル全体に関わるルール→Universeのルールエディタに記述します
  - モデルツリーのUniverseをクリックするとUniverseのルールエディタが開きます
  - Universeのルールは以下の部分からなります
    - univ_init … シミュレーション開始時に一度だけ実行するルール
    - univ_step_begin … シミュレーションの各ステップ開始時に実行するルール
    - univ_step_end … シミュレーションの各ステップ終了時に実行するルール
    - univ_finish … シミュレーション終了時に一度だけ実行するルール

❑3.5 エージェントの行動ルールを作成する(4) エージェントを生成

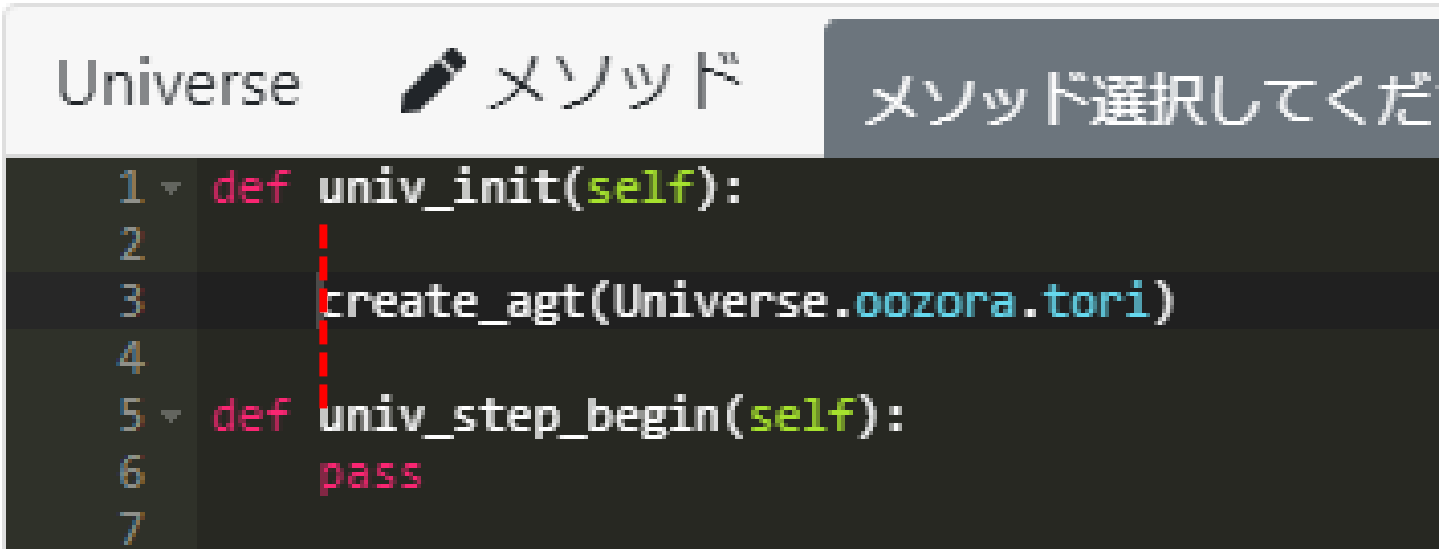
- シミュレーション開始時に鳥エージェントを1体生成
  - univ_initに鳥エージェントを生成するルールを記述します
  - エージェントの生成→create_agt：エージェントを生成する**関数**
    - 関数：ルールをひとまとめにしたもの
    - 「create_agt(Universe.oozora.tori)」でtoriエージェントを生成します
  - 「pass」を消し、下図のように記述します
    - インデントに注意→次ページ



❒TIPS：ルール作成時の注意「インデント」

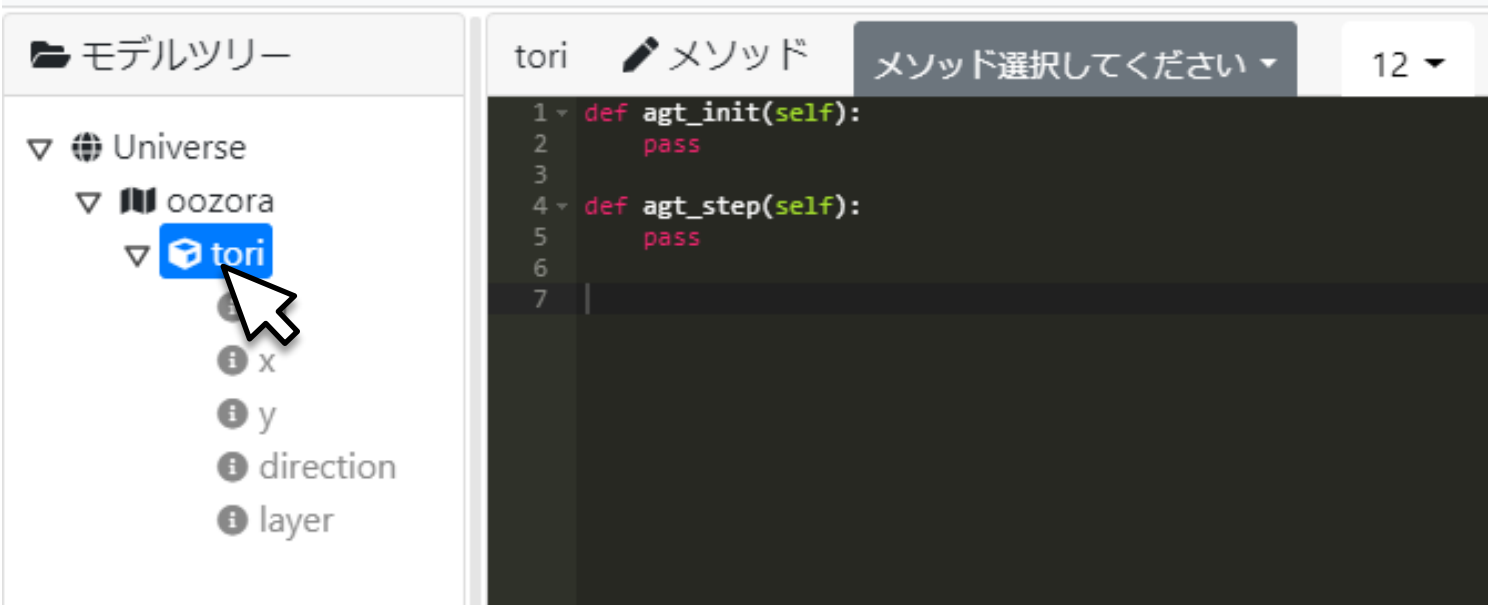
- ルールは、tabキーを使って1段下げた部分に記述することに注意
  - これをインデントといいます
  - インデントをするにはtabキーを使うと便利です
    - 「shift + tab」でインデントが戻ります
  - エンターキーで改行してもインデントは維持されます

※空白行は自由に挿入してかまいません



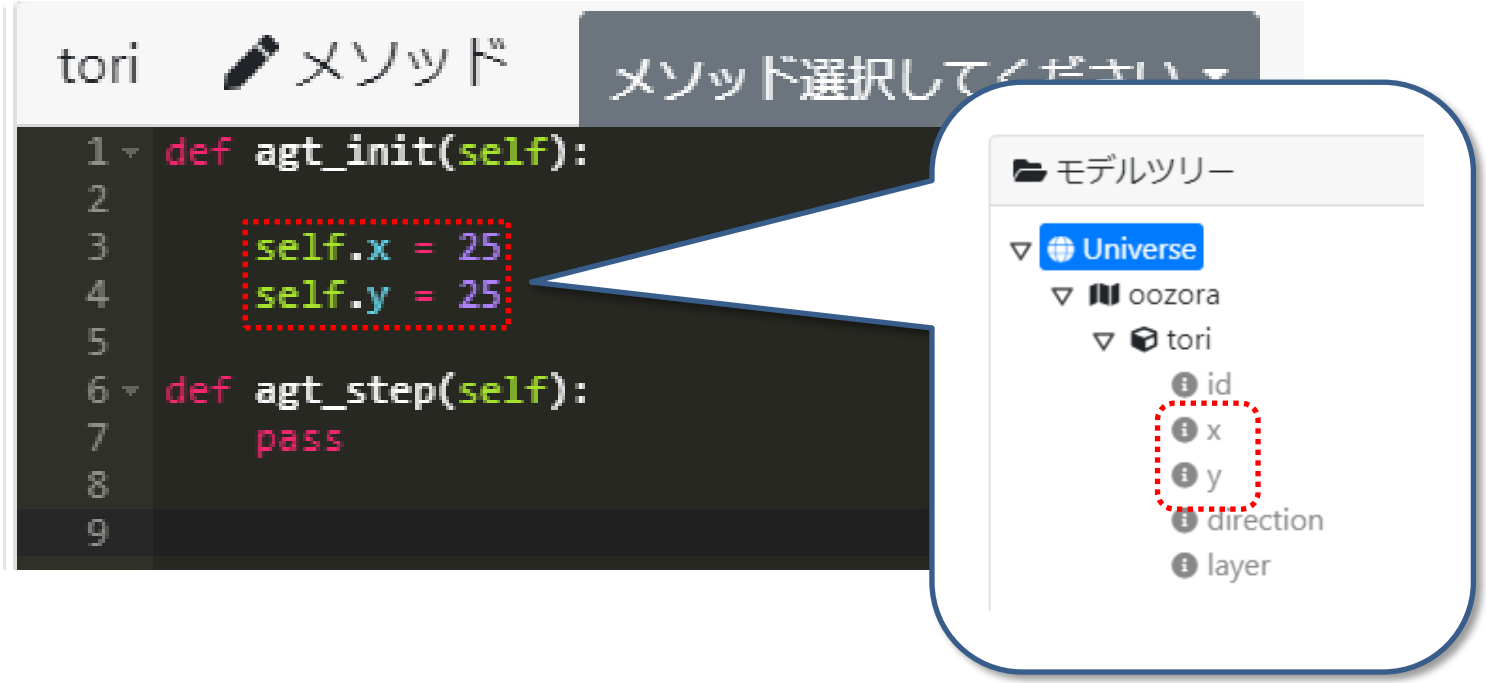
❒3.5 エージェントの行動ルールを作成する(5) 行動ルールを作成

- エージェントの行動ルールを作成します
  - エージェントの行動ルール→エージェントのルールエディタに記述します
  - ツリー上のエージェント種別名をクリックするとエージェントのルールエディタが開きます
  - エージェントのルールは以下の部分からなります
    - agt_init … エージェントが生成されたとき一度だけ実行されるルール
    - agt_step … エージェントが毎ステップ実行するルール



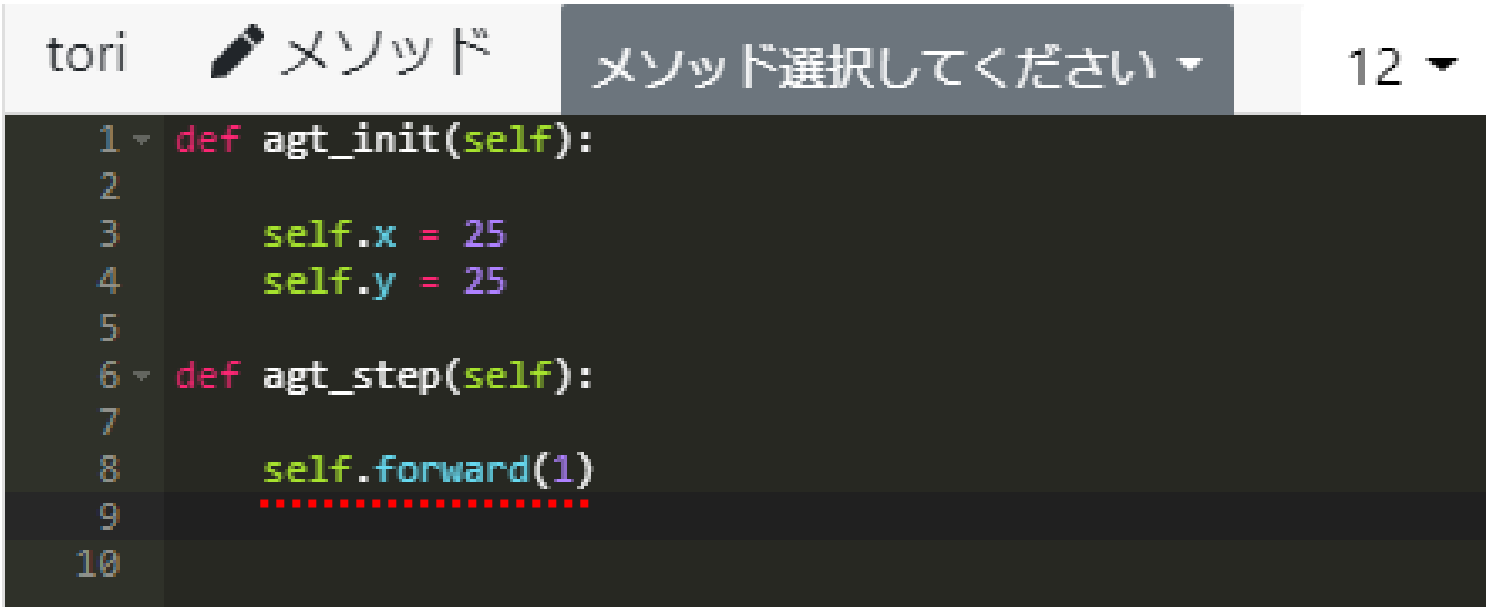
❒3.5 エージェントの行動ルールを作成する(6) 行動ルールを作成

- 鳥エージェントを初期位置として空間の中央に配置
  - agt_initにルールを書きます
  - 自分自身のx座標とy座標に25を代入します
    - 自分自身の変数を呼び出すときには「self.[変数名]」と記述します
    - 「変数名 = 数値」と記述することで、「変数に数値を代入する」という意味になります
      - ☑「=」前後の半角スペースはなくてもよいが、あったほうが見やすい
  - 「pass」を消し、下図のように記述してください
    - pass（パス）は何もしないという意味



❑3.5 エージェントの行動ルールを作成する(7) 行動ルールを作成

- 鳥エージェントは毎ステップ前に進む
  - agt_stepにルールを書きます
  - forward：前に進む関数
    - 「self.forward([数値])」と書くことで数値だけ前に進みます
      - ☑「self」は自分自身なので、自分自身に「進め」と命令するイメージ

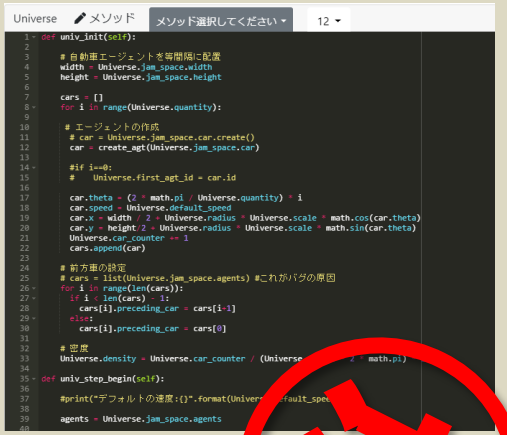


❑3.5 エージェントの行動ルールを作成する(8)

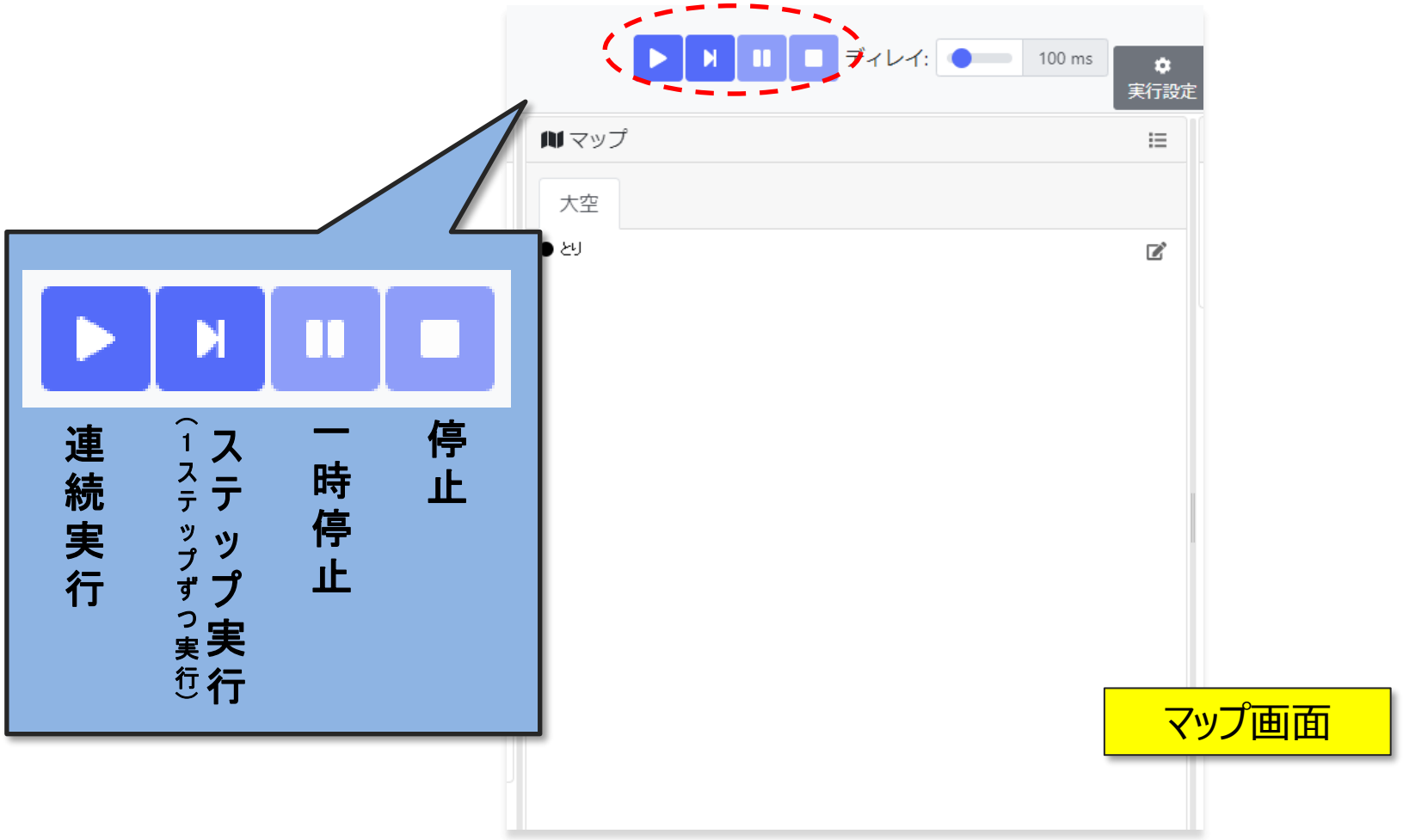
① 空間／エージェントの
種類・属性を作成

② シミュレーション結果の
出力形式を決定

③ エージェントの
行動ルールを作成



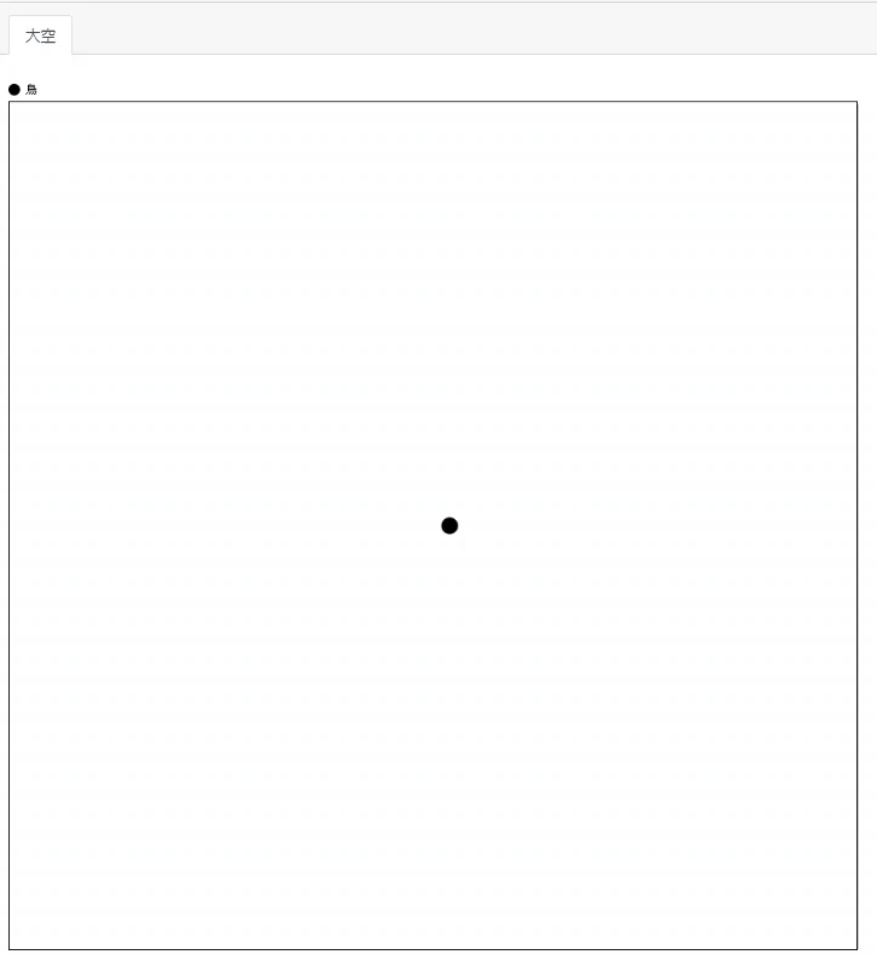
❑3.6 モデルの動作を確認する(1) シミュレーションを実行



ルールの編集、出力の設定を行うときは **「停止」** 状態である必要があります

❒3.6 モデルの動作を確認する(2)

- マップ上を鳥が左から右に動いていれば成功です
  - 10000ステップで自動終了するようになっています



❐「飛ぶ鳥モデル 3.5終了時点」のモデル

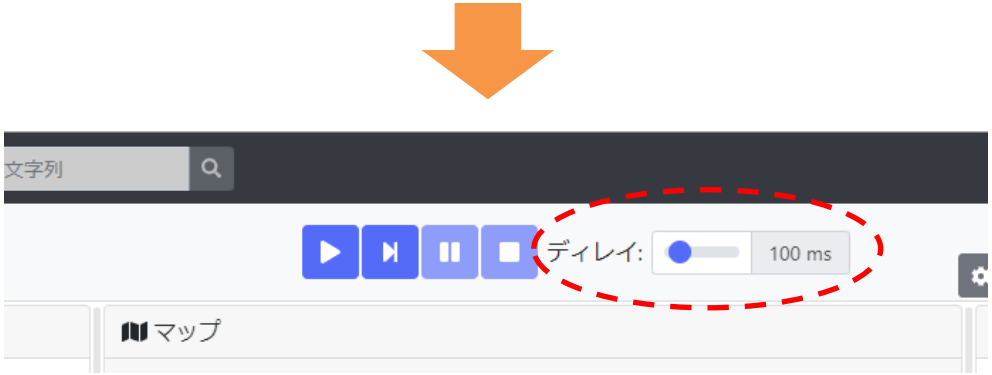
途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/cL2BjrAESiCrg5L8xpEcrQ
  - モデル名「飛ぶ鳥モデル 3.5終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❒TIPS：シミュレーション速度を調整する

シミュレーション速度を変更したい・・・

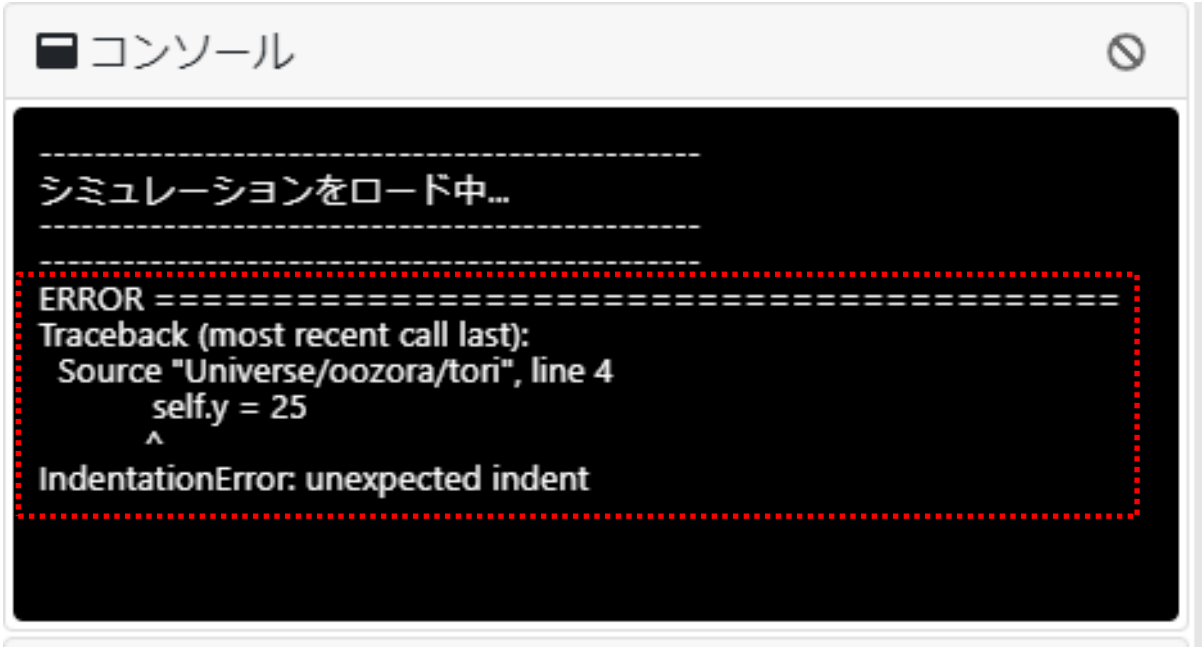
スライドバーで実行速度を調整する



右に動かすと実行速度が遅くなり、左に動かすと速くなる

❒TIPS：うまく動かないときは誰かに質問する

- うまく動かなければ、Q&Aで質問してください
  - コンソール画面にエラーメッセージが出ていたら、コピーしてお知らせください

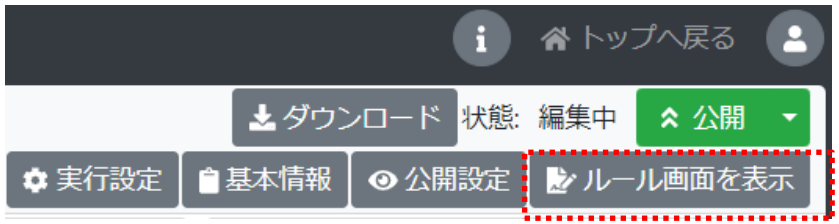


※ どうしても解決できない場合は、[質問掲示板](#) に問合せてください

❐TIPS：空間のループを変更する

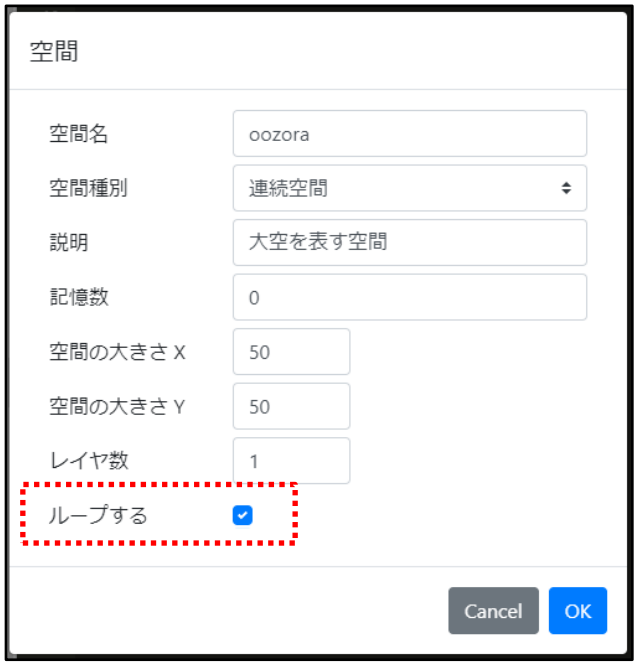
■ 空間がループ ＝上端-下端・左端-右端が繋がった空間

ループした空間のイメージ



「ルール画面を表示」をクリック

「Edit」をクリック



❑学んだ事項

- 新規モデルに名前をつけて保存する。
- ツリーに空間とエージェントを作る。
- ルールを定義する。
- 空間を見るためにマップ出力設定をする。
- 実行ボタン、停止ボタンを押す。
- こまめにモデルを保存する。

第4章：エージェントを動かす（基本編）

❐4.0 ルールが肝腎です

- 前章の続きとして、この章で鳥を飛ばすモデルを完成させましょう。
- エージェントのルールエディタに行動ルールを書き込みます。
- 動かすルールの初歩を学びます。
- 乱数を利用します。
- 実際に動かして、シミュレーションの過程を観察しましょう。
- ルールの変更や設定の変更に慣れましょう。

❒4.1 エージェントの自律性とはどういうものか？

- MASの基本＝個々のエージェントが自律的に行動すること。
  - 自律的行動の弱いとらえ方（※強いとらえ方は次章以降）
    - 100羽のtoriを動かすケースで、
      - ☑ 1羽ごとにどのように動くべきかを厳密に指示しない
      - ☑ 行動の「ルール」を教えておいて、エージェントはそのルールに従って行動
    - Toriがどこに動くのかを事細かに指定せず、
      動き方のルールさえ指定すれば良い。

❐「飛ぶ鳥モデル 3.5終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

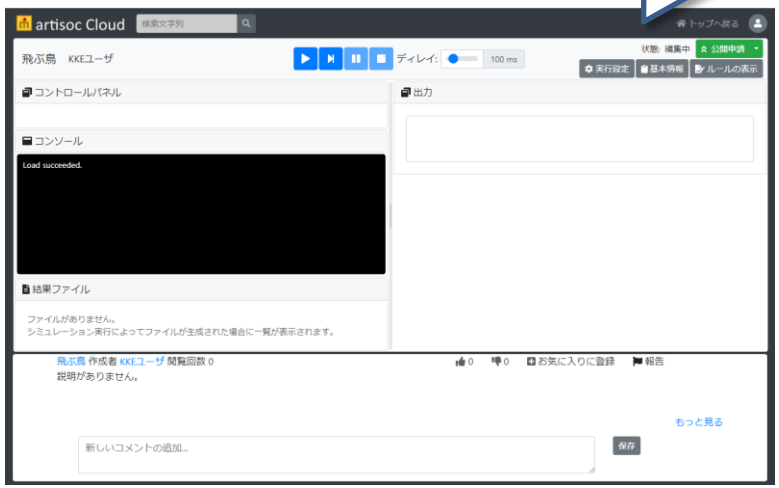
- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/cL2BjrAESiCrg5L8xpEcrQ
  - モデル名「飛ぶ鳥モデル 3.5終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❑4.2 エージェントのルールエディタを開く(1)

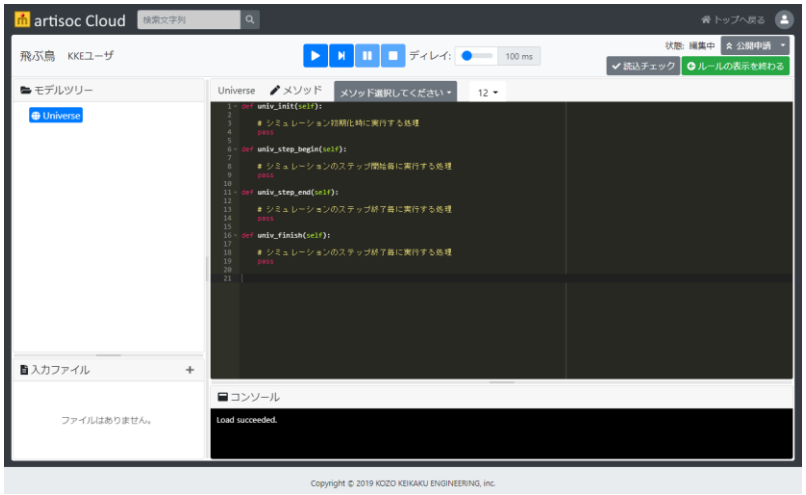
- artisoc Cloudにアクセス
  - https://artisoc-cloud.kke.co.jp/
- ログインしていない場合、右上の[ログイン]をクリックしてログイン
- トップページ画面右上の人型アイコンをクリック → [ユーザページ]をクリック
  → ユーザページに遷移
- ユーザページの作成したモデルから、「飛ぶ鳥」のモデルを選択
  ※継承して作成した「飛ぶ鳥」モデル

❑4.2 エージェントのルールエディタを開く(2)

■「ルール画面を表示」をクリックしてモデルのルール画面に遷移します。



出力画面



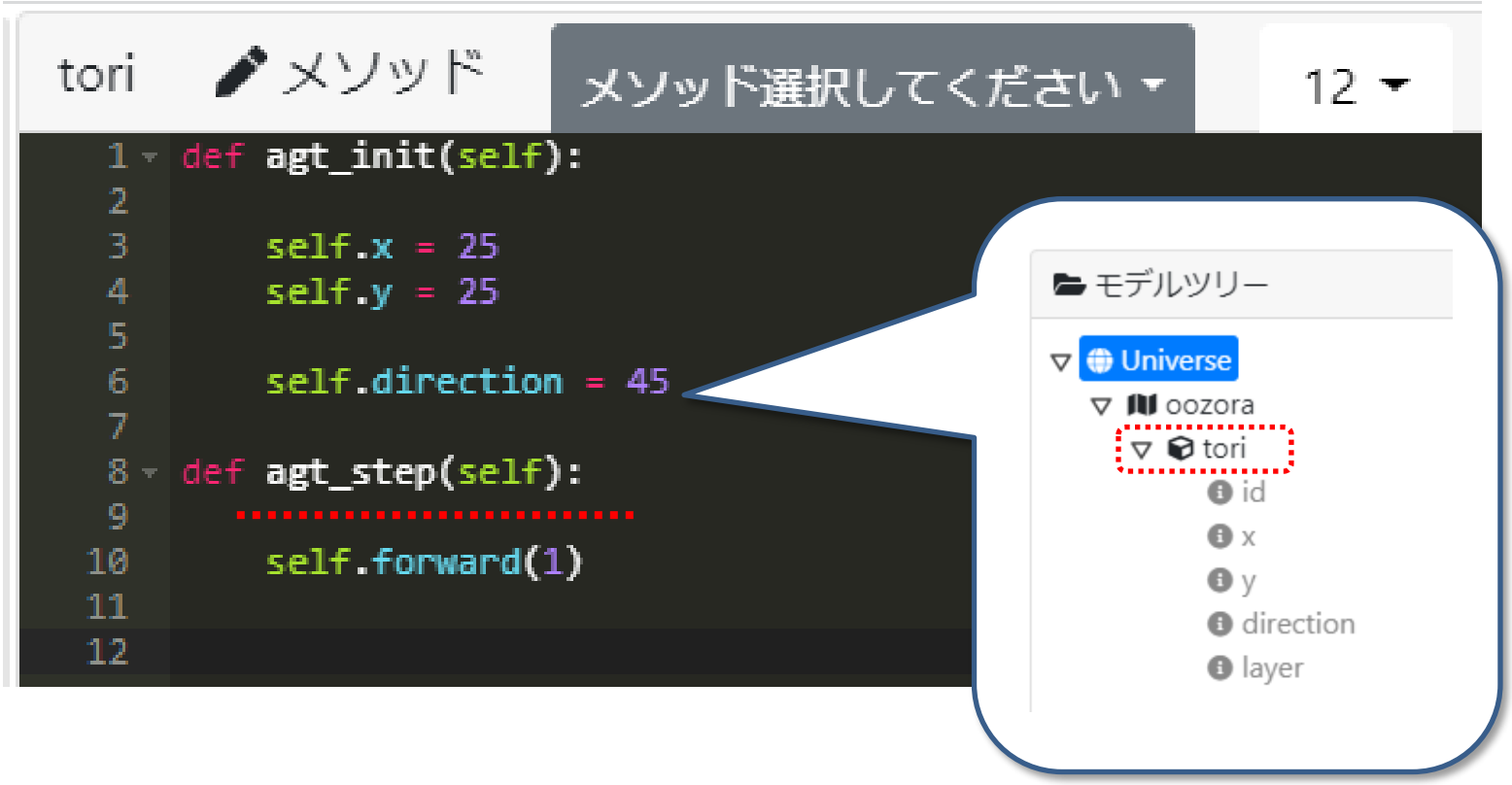
ルール画面

❐4.3「動かす」ルールを書き込む(1)

- 行動ルール
  - (1) 全てのtoriがoozoraの中央部（X=25, Y=25）にいる。
    - ➡ 左下隅が原点（0, 0）
  - (2) 各toriは、勝手な（ランダムな）方向を向いている。
  - (3) 各toriは、毎ステップ、「1」ずつ、その方角に飛び続ける。
    - ➡ ステップ=シミュレーション上の時間単位

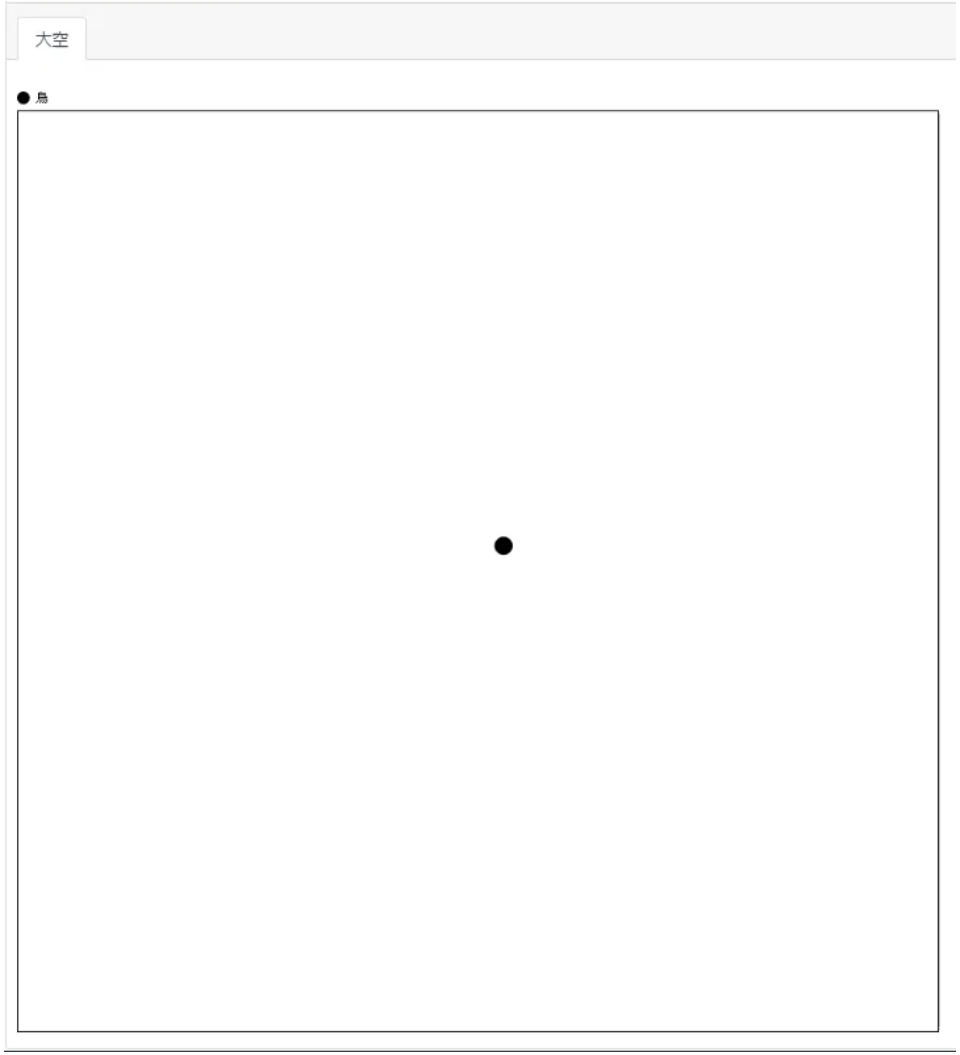
❐4.3「動かす」ルールを書き込む(2)　進む方向を変える

- ルール画面に戻り、モデルツリー上のtoriをクリックしてルールエディタを開きます。
- agt_initで変数「direction」に45を代入して実行します。

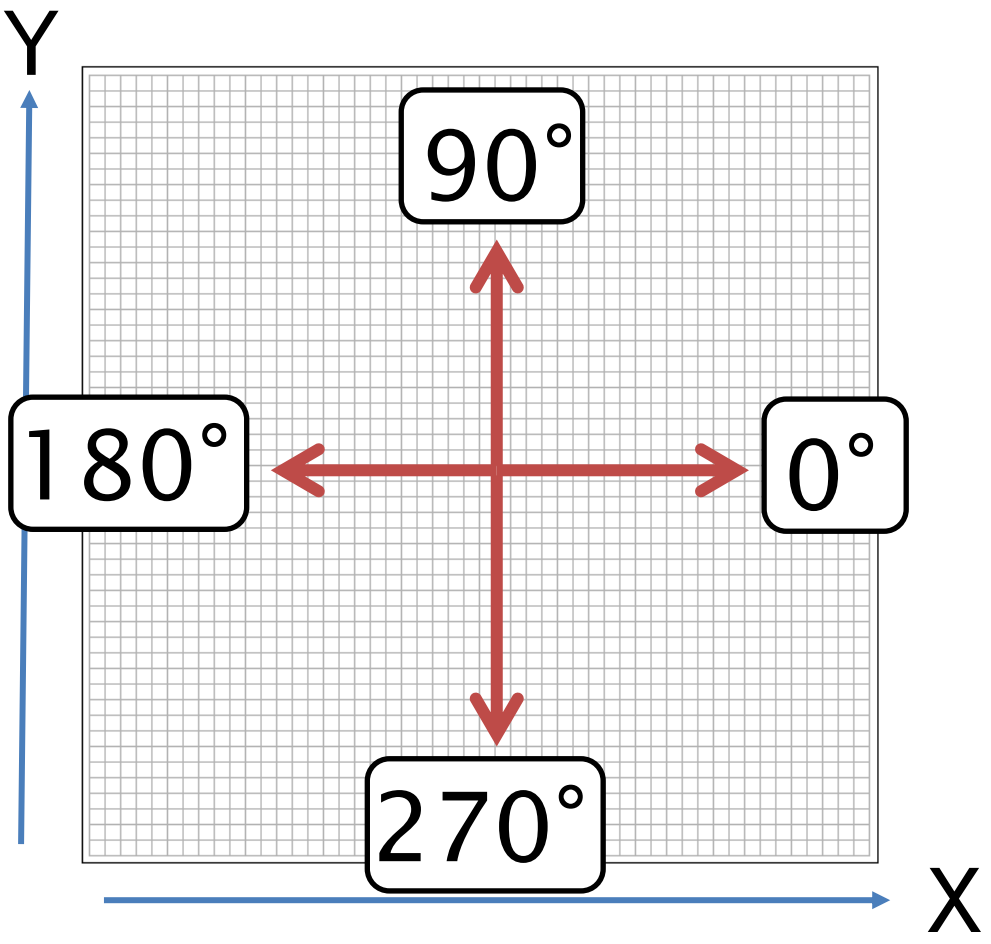


❐4.3「動かす」ルールを書き込む(3)

- 鳥が右上に飛んでいきます。



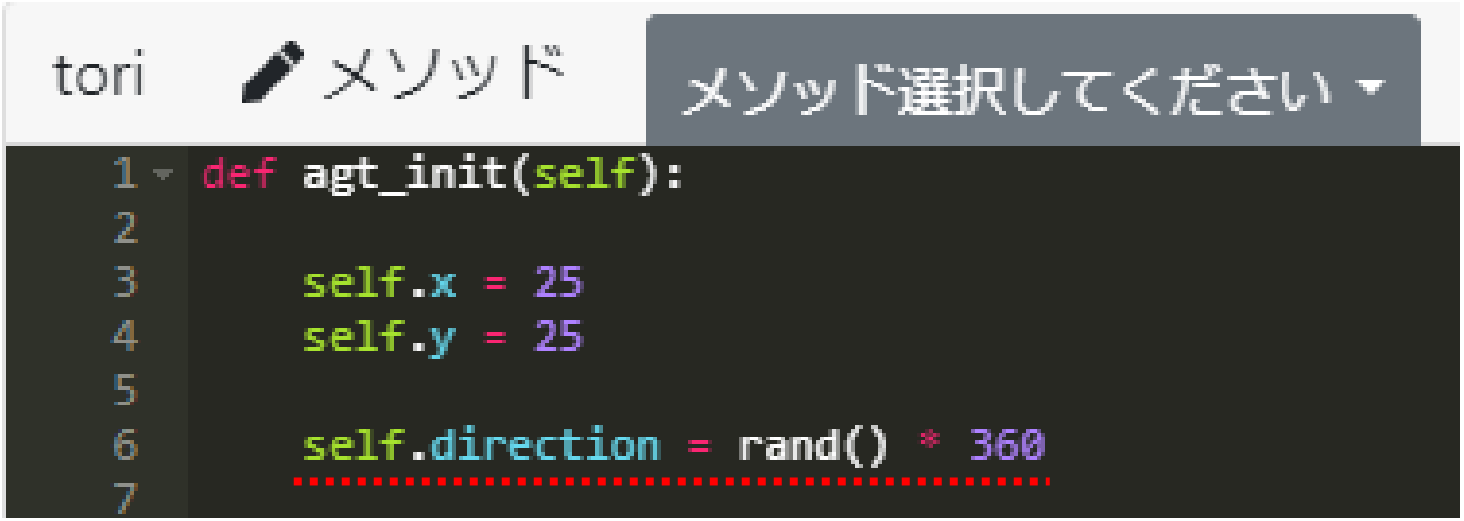
❑4.3「動かす」ルールを書き込む(4)　方向について



※左下原点の場合

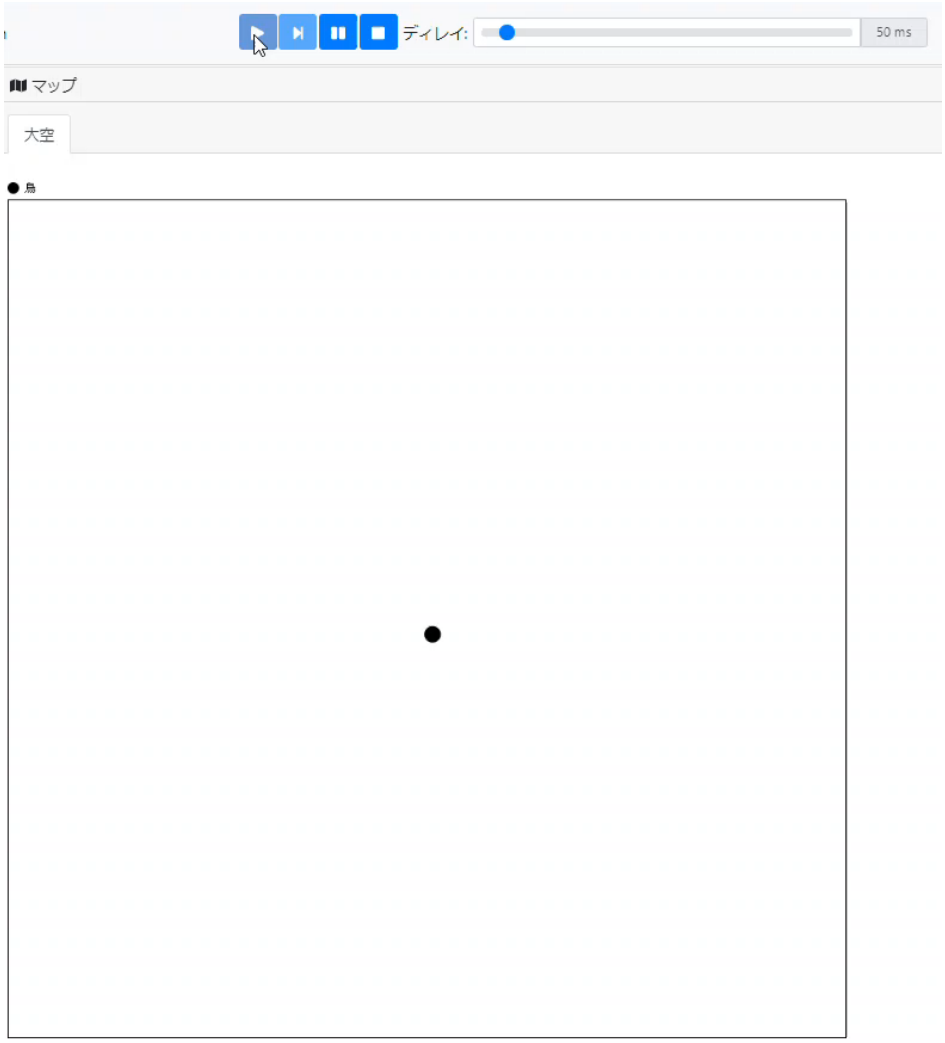
❑4.3「動かす」ルールを書き込む(5)　進む方向をランダムに変える

- 属性「direction」に進む方向の値をランダムに設定（0~360度）して実行します。
  - rand：0~1のランダムな値（一様乱数）を取得する関数
  - 「rand() * 360」で0~360のランダムな値を意味します。
    - 「*」は掛け算の記号です。
    - 関数には必ず()がつきます。
    - 「=」は右の値を左の変数の値に代入するという操作を示す。←要注意



❐4.3「動かす」ルールを書き込む(6)

■ 実行と停止を繰り返すと、実行するたびに鳥の飛ぶ向きが変わります。



❒関数とは

- 関数：処理をひとまとめにしたもの
  - 引数：関数へ渡す値
  - 処理：引数をもとに関数が行う動作
  - 返り値（戻り値）：処理をもとに関数が返す値

  ※引数は複数あったり、なかったりします

  ※返り値がない（使わない）関数もあります

- 関数の例：forward
  - 引数：進む距離
  - 処理：引数の値だけ前に進む
  - 返り値：正常終了時は-1（今は使っていません）

```
self.forward(10)　# 引数の数10だけ前に進む
```

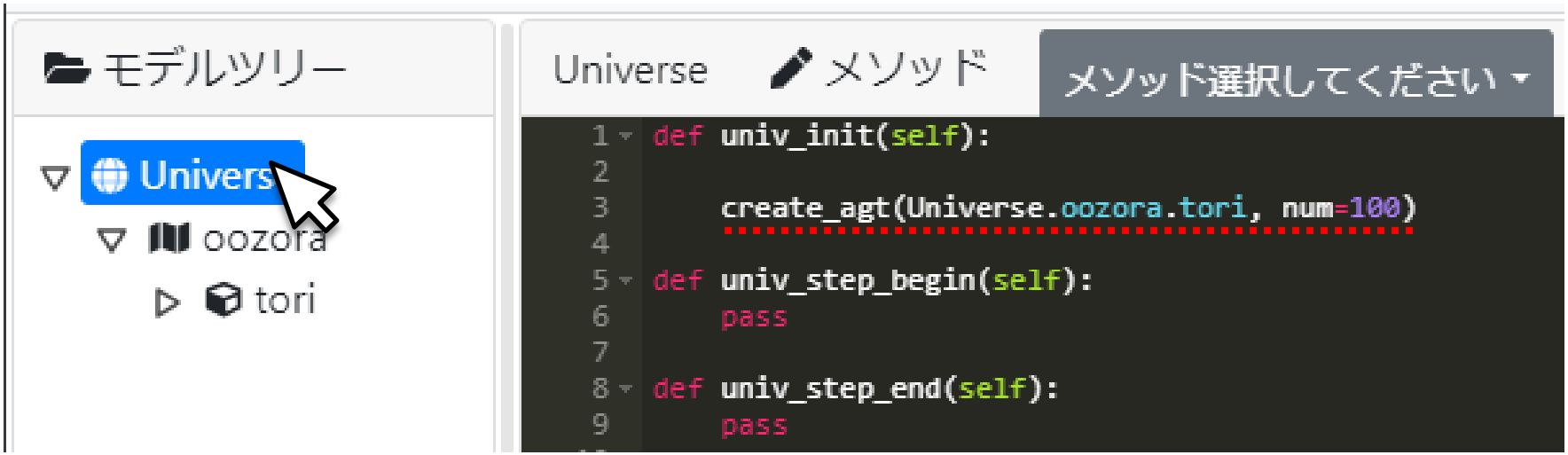
- 関数の例：rand
  - 引数：なし
  - 処理：0~1の一様乱数を発生させる
  - 返り値：0~1の一様乱数

```
r = rand()　# rand関数の返り値（0~1の一様乱数）を変数rに代入する
```

※引数がないときも()が必要

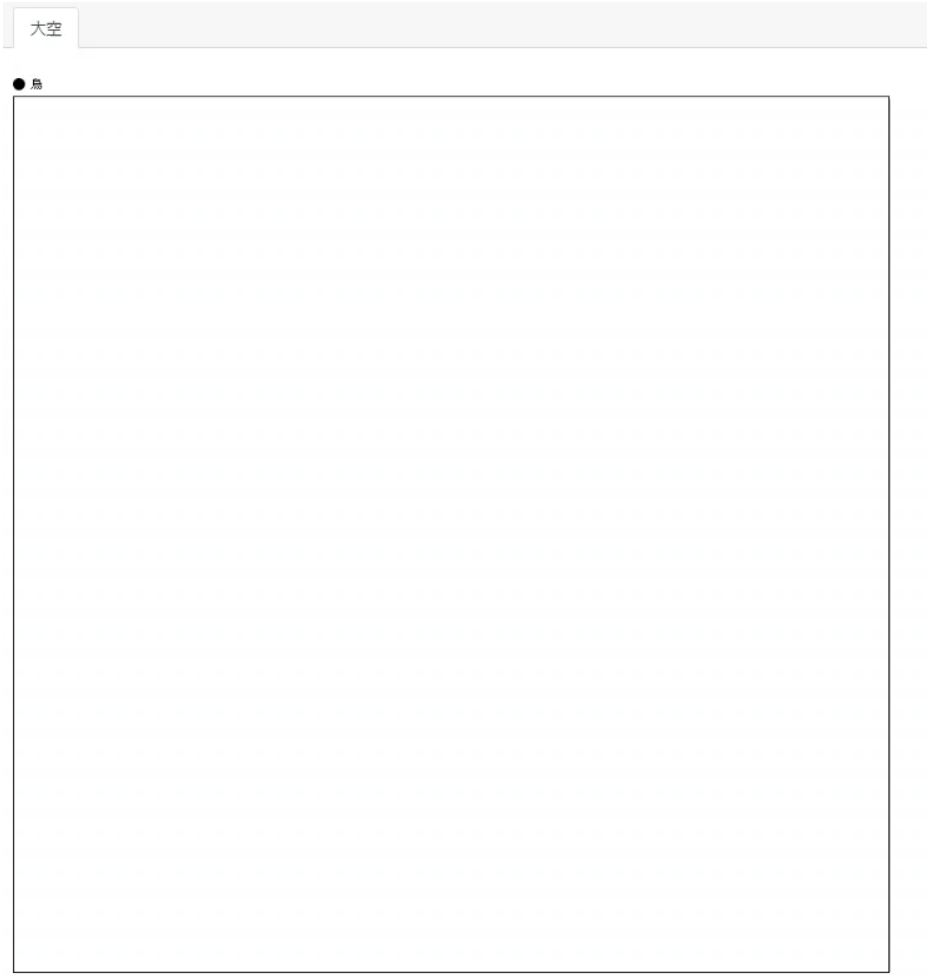
❒4.4 いよいよ実行(1) エージェントの生成の仕方を変える

- エージェントを一度に複数生成する。
  - 「create_agt([エージェント種別], num=[エージェント数]）」で指定した数のエージェントを生成します。
  - 100個生成します。



❑4.4 いよいよ実行(2) エージェントの生成の仕方を変える

- 100羽の鳥が円形に飛んでいきます。



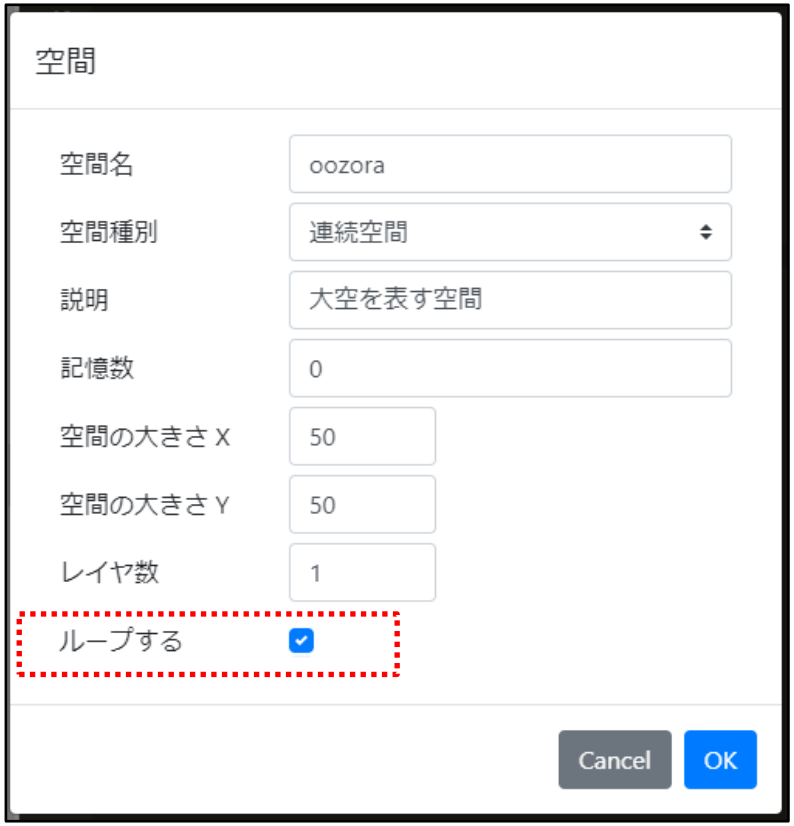
❐「飛ぶ鳥モデル 4.4 終了時点」のモデル

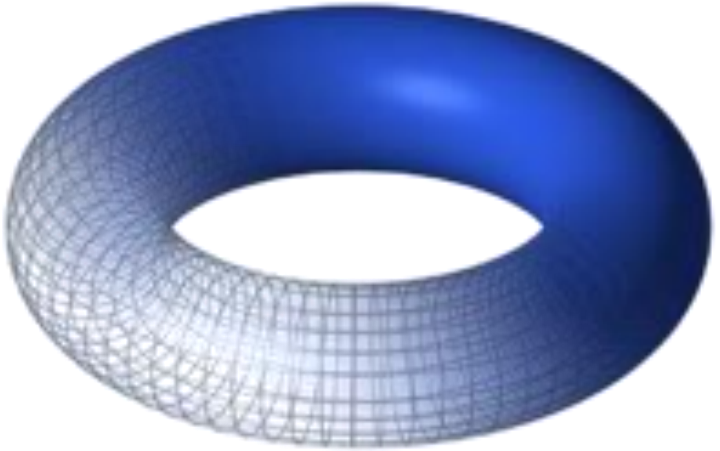
途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/BGDukBgKSmGNYNBTNPTyxg
  - モデル名「飛ぶ鳥モデル 4.4 終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❑4.4 いよいよ実行(4)　空間のループ

- 空間がループする ＝上端と下端、左端と右端が繋がった空間





ループした空間のイメージ

❐4.5 動かすルールや設定変更に慣れる(1)

- toriの数を100羽から1000羽に増やす。
  - 「ルール画面を表示」を選択。
  - 「ツリー」の中の「Universe」を選択。
  - 「ルールエディタ」から以下のように変更して、エージェント数を1000にする。
    - 変更前： one = create_agt(Universe.oozora.tori, num=100)
    - 変更後: one = create_agt(Universe.oozora.tori, num= 1000)
  - 保存ボタンをクリック

- toriを左上端から飛び立たせる。
  - 「ルール画面を表示」を選択。
  - 「ツリー」の中の「tori」を選択。
  - 「ルールエディタ」で、以下のように変更する
  - self.x、self.yの値をそれぞれ0、50に変える。

❑4.5 動かすルールや設定変更に慣れる(2)

Universeのルール

```
def univ_init(self):
     create_agt(Universe.oozora.tori, num=1000)

def univ_step_begin(self):
    pass

def univ_step_end(self):
    pass

def univ_finish(self):
    pass
```

toriのルール

```
def agt_init(self):

    self.x = 0
    self.y = 50

    self.direction = rand() * 360

def agt_step(self):

    self.forward(1)
```

❒4.5 動かすルールや設定変更に慣れる(3)

- toriは飛ぶ方向を毎ステップでたらめに選ぶ。
  - 「ルール画面を表示」を選択。
  - 「ツリー」の中の「tori」を選択。
  - 「ルールエディタ」で、以下のように変更する
  - agt_init(self): の中のself.direction = rand() * 360を、
    agt_step(self): 内に移す。

```
1  def agt_init(self):
2
3      self.x = 0
4      self.y = 50
5
6
7  def agt_step(self):
8
9      self.direction = rand() * 360
10
11     self.forward(1)
12
13
```

❐「飛ぶ鳥モデル 4.5 終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/GpySUNkhTsmIRwqWGJQjmw
  - モデル名「飛ぶ鳥モデル 4.5 終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

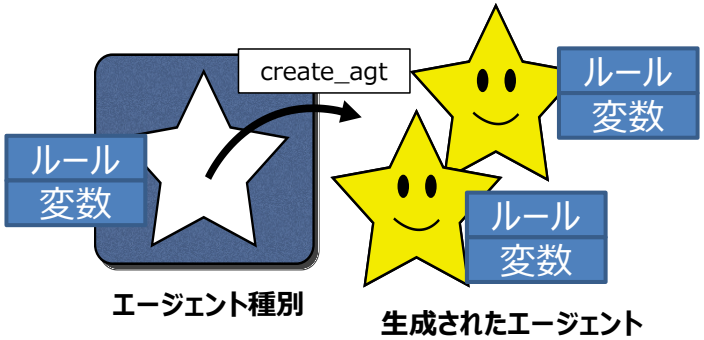
❑新しく学んだ事項

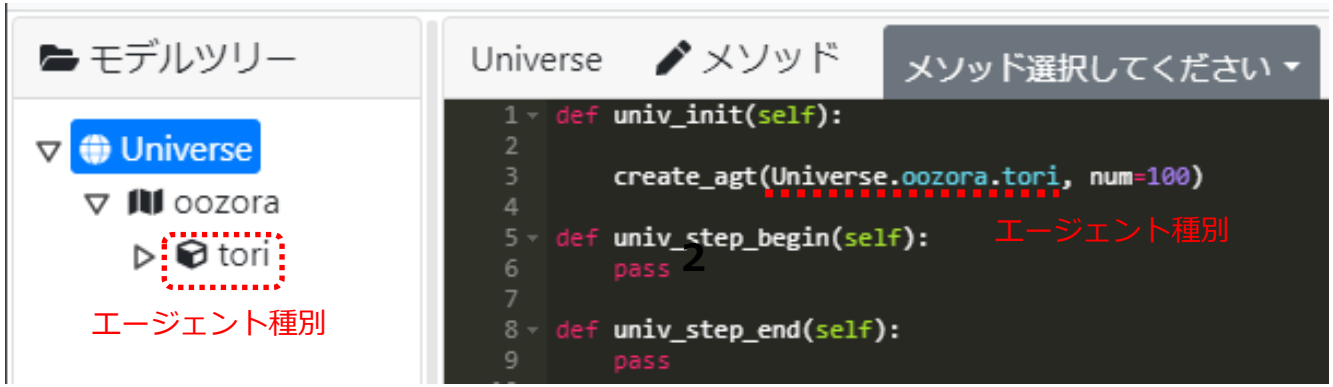
- エージェントのルールエディタにルールを書く。
- self.変数の入力支援機能。
- agt_init(self): とagt_step(self):の違い。
- self. に続くエージェントの変数x、y、directionの使い方。
- 右辺の値を左辺の変数に代入する代入文(=)
- かけ算の記号*
- rand()
- forward()
- 空間はループしている（デフォルト状態）
- ルールエディタの中で、ローマ字の大文字と小文字の区別は見やすいように。
- シミュレーションの実行をわざと遅くする（「実行ウェイト」の変更）
- 別の名前をつけて保存する。

第4章：エージェントを動かす（応用編1）

❒4.6 エージェント種別とエージェント

- エージェント種別：エージェントの「ひながた」のようなものです。
  - ツリー上に存在するのはエージェントでなくエージェント種別です。
- create_agtはエージェント種別を指定して、エージェントを生成します。
- 生成されたエージェントは共通のルールと変数を持ちます。変数の値はそれぞれのエージェントで異なります。
  - 共通のルール：
    「directionをランダムに設定する」→direction変数の値は個々のエージェントで異なる。





❒4.7 飛ぶ速さを変える(1)

- 個々のエージェントで飛ぶ速度を変えてみましょう。
  - 飛ぶ速さを表す新たなエージェントの変数「speed」を追加します。

❒4.7 飛ぶ速さを変える(2)

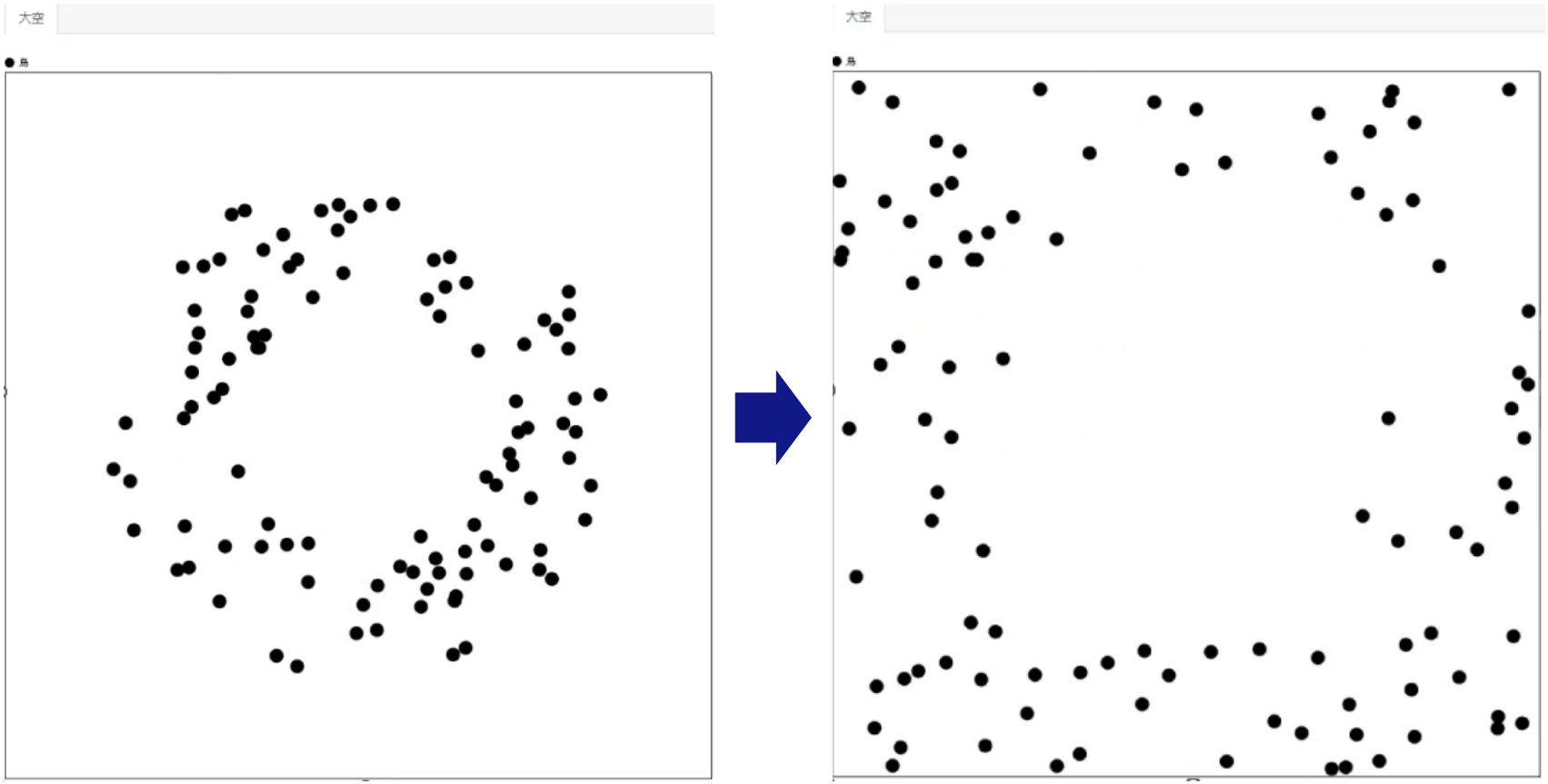
- speedの値をランダムに設定し、飛ぶ速さとして指定します。
  - agt_initでspeedにランダムな値を代入する。
    - 「1 + rand()」とすることで1~2の間のランダムな値をとる。
  - agt_stepでforwardの引数にspeedを指定する。

tori　メソッド　メソッド選択してください

```
1  def agt_init(self):
2
3      self.x = 25
4      self.y = 25
5
6      self.direction = rand() * 360
7
8      self.speed = 1 + rand()
9
10 def agt_step(self):
11
12     self.forward(self.speed)
13
```

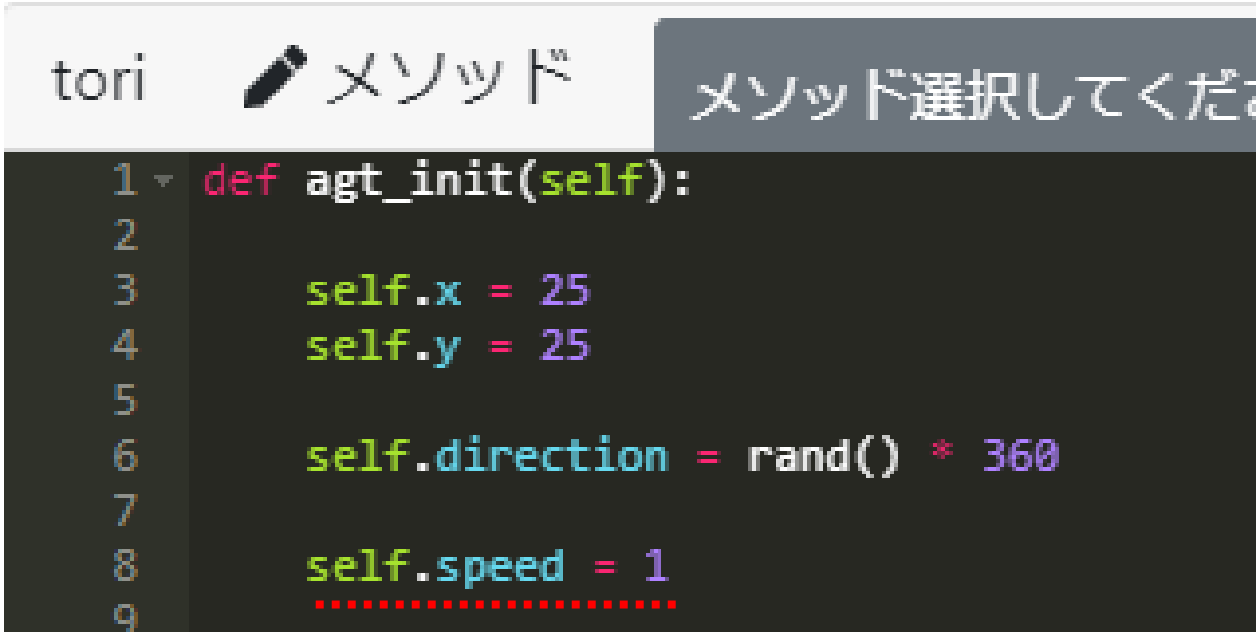
❐4.7 飛ぶ速さを変える(3)

- 速度が異なるため、散らばって飛んでいきます。



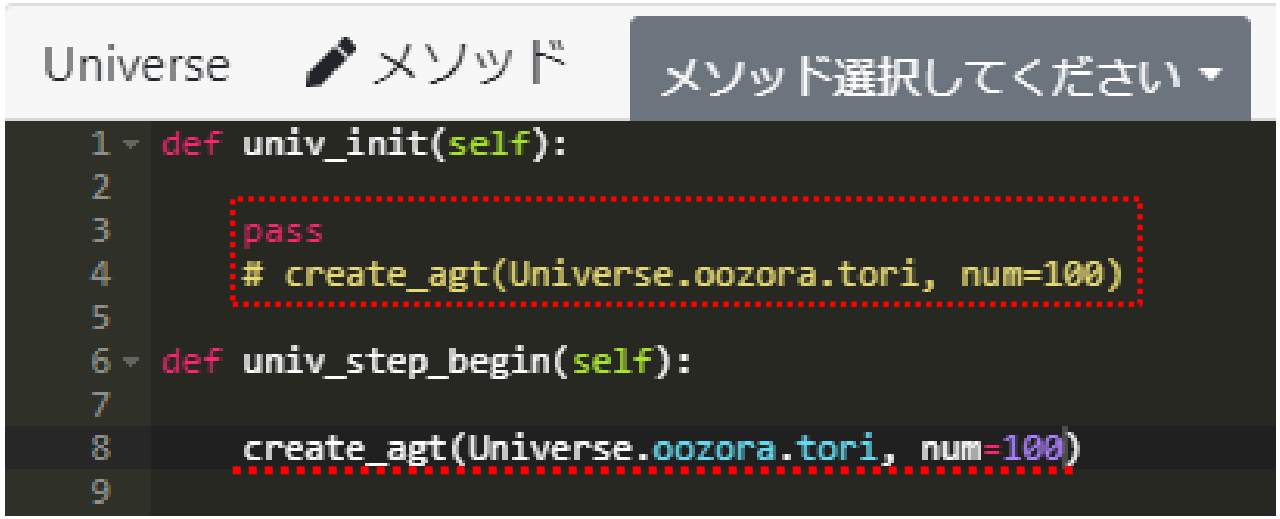
❐4.7 飛ぶ速さを変える(4)

- 速度は1に戻しておきましょう。



❒4.8 エージェントの生成の仕方を変える(1)

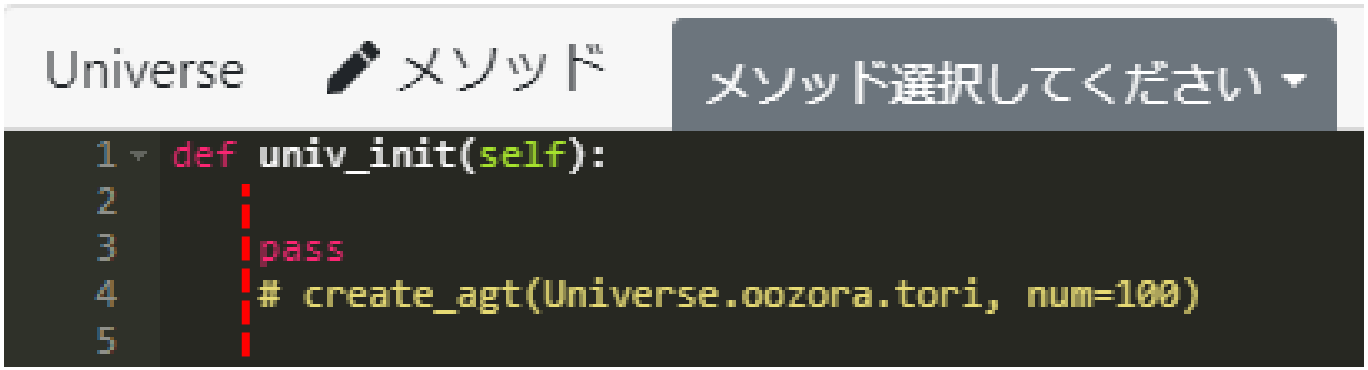
- エージェントを毎ステップ生成する。
  - create_agtをuniv_step_beginに移します。
    - これで、各ステップの開始時にエージェントを生成します。
    - univ_initに書いてあったルールはコメントアウトし、「pass」を追記します（次ページ参照）。



❒artisoc Cloudのヒント　コメントアウトとpass

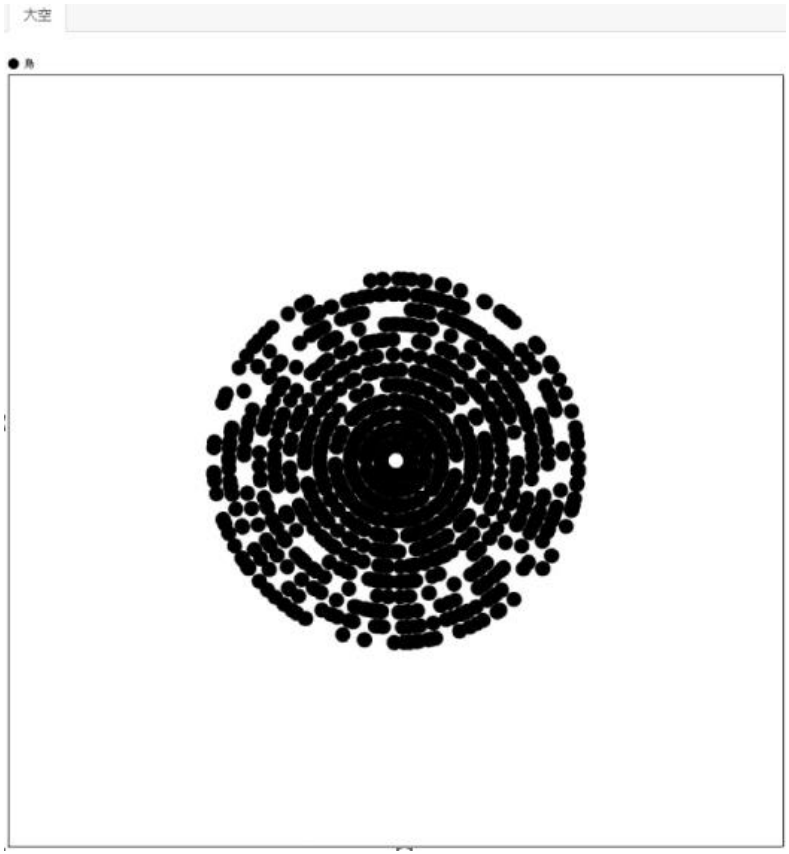
- コメントアウトについて
  - コードを無効化します。
  - 行の先頭に「＃」
    - 行にカーソルを合わせたり選択した状態で「ctrl ＋ /」でもコメントアウトできます。
  - コードを一時的に変更したり、モデルの説明を加える場合にも使います。

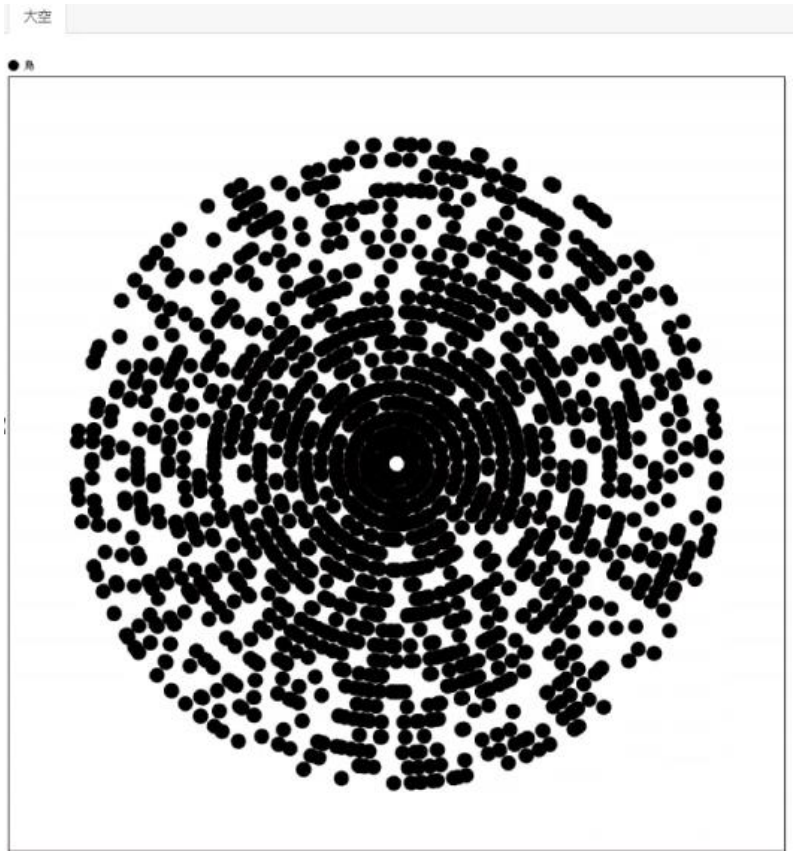
- passについて
  - 「何もしない」ことを意味するルールです。
  - univ_initなどのルールエディタの要素内では、インデントの中に何かを記述する必要があります。
    - でないと、「インデントがない」というエラーになる。
  - 何もしないときは、「何もしない」ということをartisoc Cloudに教えるためにpassを記述します。



❐4.8 エージェントの生成の仕方を変える(2)

■ 毎ステップ、エージェントが生成されます。





❒4.9 ルールの構成

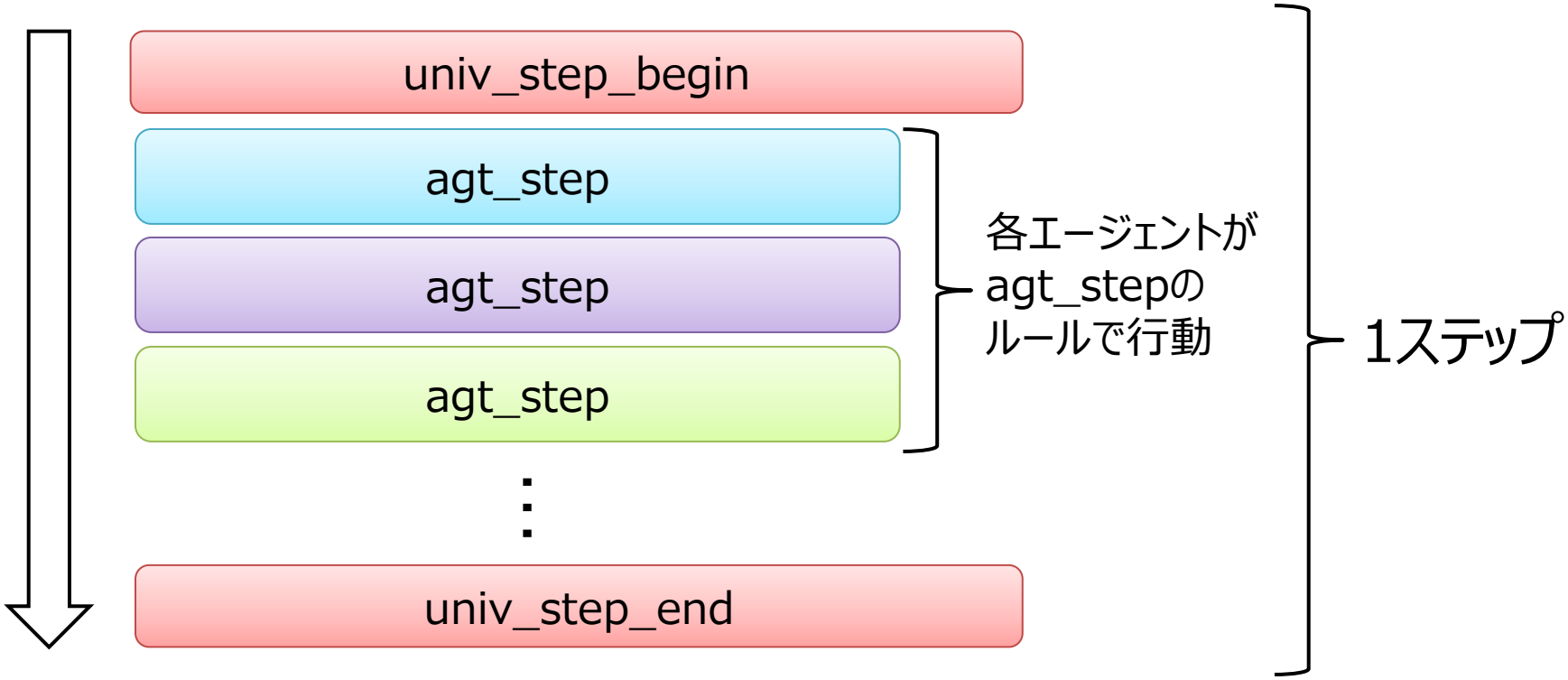
- エージェントのルールエディタ→エージェントの行動ルール
  - agt_init ・・・ エージェント生成時に一度だけ実行
  - agt_step ・・・ 各ステップで実行

- Universeのルールエディタ→モデル全体のルール
  - univ_init ・・・ シミュレーション開始時に一度だけ実行
  - univ_step_begin ・・・ 各ステップの最初に実行
  - univ_step_end ・・・ 各ステップの最後に実行
  - univ_finish ・・・ シミュレーション終了時に一度だけ実行

❒4.10 ステップについて

- ステップ
  - artisoc Cloudの時刻単位
  - 1ステップの間に、全てのエージェントがルールにしたがって行動する。

- 1ステップの流れ



❒4.11 条件分岐文(1)

- 「10ステップ目までエージェントを毎ステップ生成する」というルールを考えます。
  - 「現在のステップ数が10以下の場合、エージェントを生成」という場合分けのルールを書く
  - 場合分けの処理→if文で記述

条件文が真の場合に実行する処理

```
if 条件文 :
    処理
```

※処理はインデント内に記述

条件文の例

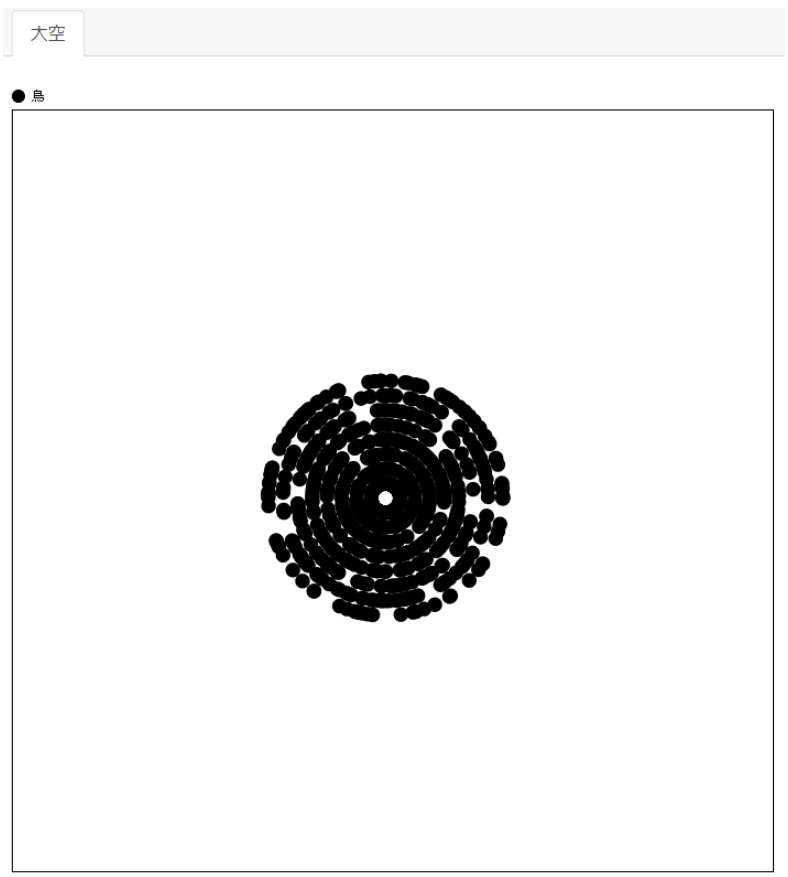
1 < 5 [1は5未満である] ⇒ 真(True)
4 >= 8 [4は8以上である] ⇒ 偽(False)
3 == 4 [3と4は等しい] ⇒ 偽(False)

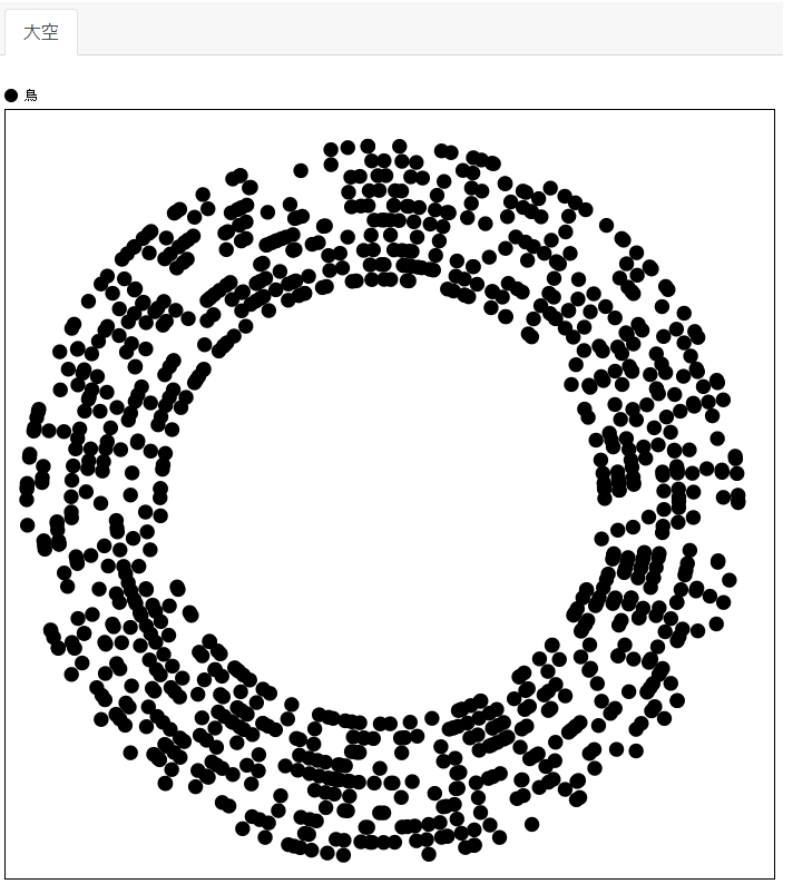
```
def univ_step_begin(self):

    if count_step() <= 10:
        create_agt(Universe.oozora.tori, num=100)
```

※count_step: 現在のステップ数を取得する関数

❒4.11 条件分岐文(2)

- 10ステップ目までエージェントが生成されます。





❒4.11 条件分岐文(3)

- 毎ステップ0.2の確率でエージェントを生成する。
  - rand()の値が0.2未満の場合、エージェントを生成する。

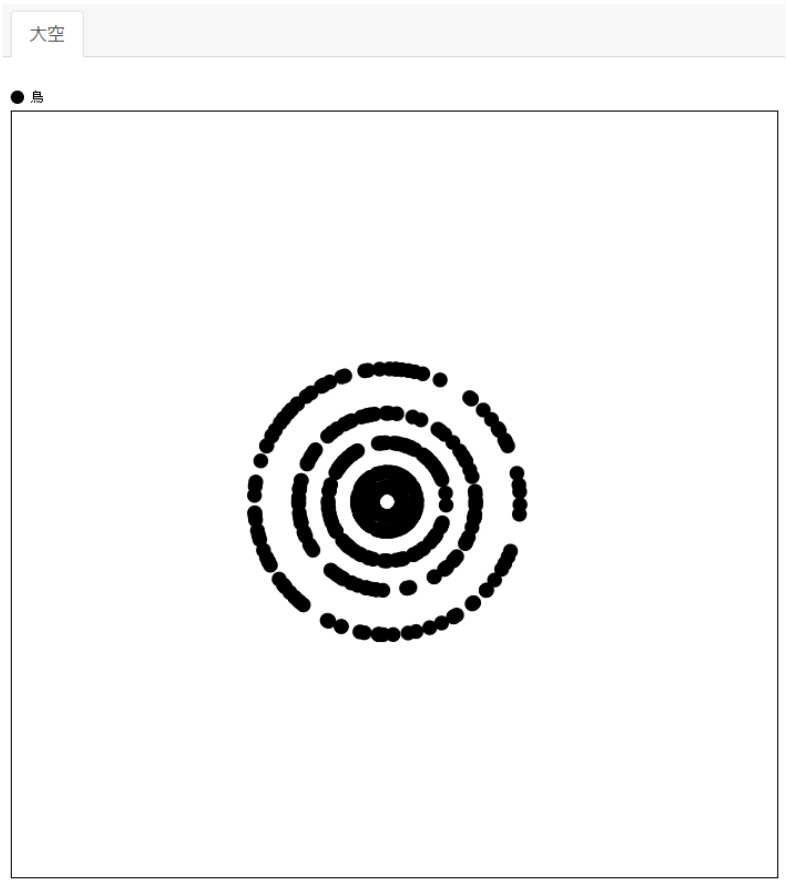
確率rで実行する処理

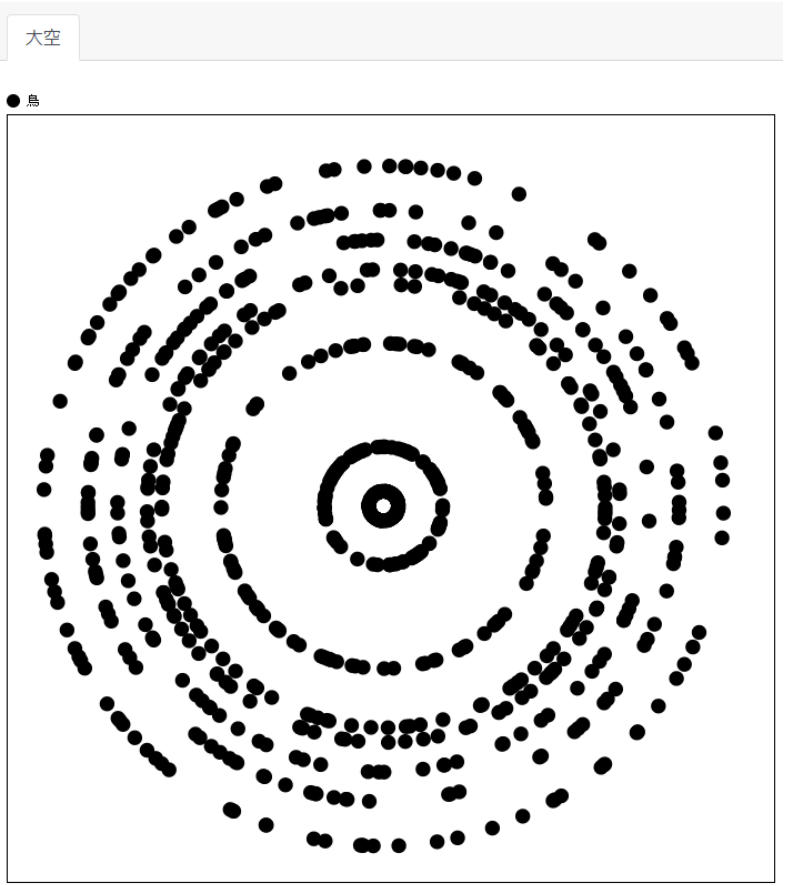
```
if rand() < r :
    処理
```

```
def univ_step_begin(self):

    if rand() < 0.2:
        create_agt(Universe.oozora.tori, num=100)
```

❒4.11 条件分岐文(4)

- 5回に1回の確率でエージェントが生成されます。



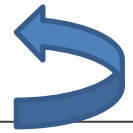


❒4.12 繰り返し文(1)

- 「シミュレーション開始時に100エージェントをまとめて生成する」というルールを考える。
  - 先ほどは「num=100」でまとめて生成しましたが、今回はエージェントを1つ作る処理を100回繰り返します。
  - 繰り返しの処理→for文を用います。

n回繰り返す処理

```
for 変数 in range(n) :
    処理
```

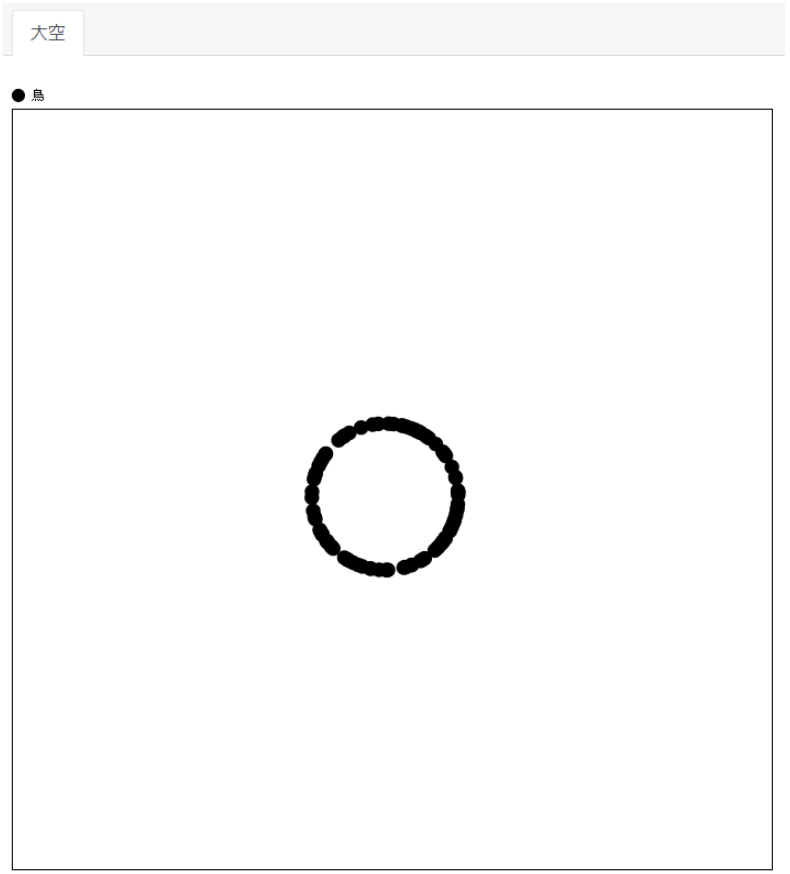


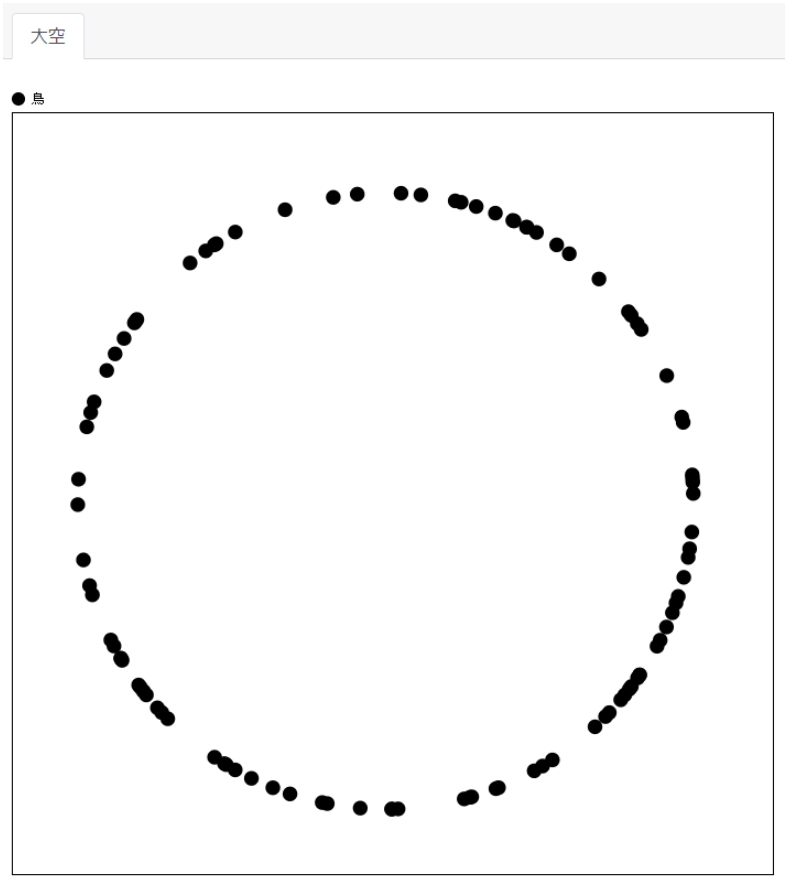
※処理はインデント内に記述
※変数は何でもかまいませんが、慣習としてiをよく使います。

  - univ_initに記述し、univ_step_beginの内容はコメントアウトします。

❒4.12 繰り返し文(2)

■ 鳥が円形に飛んでいきます。





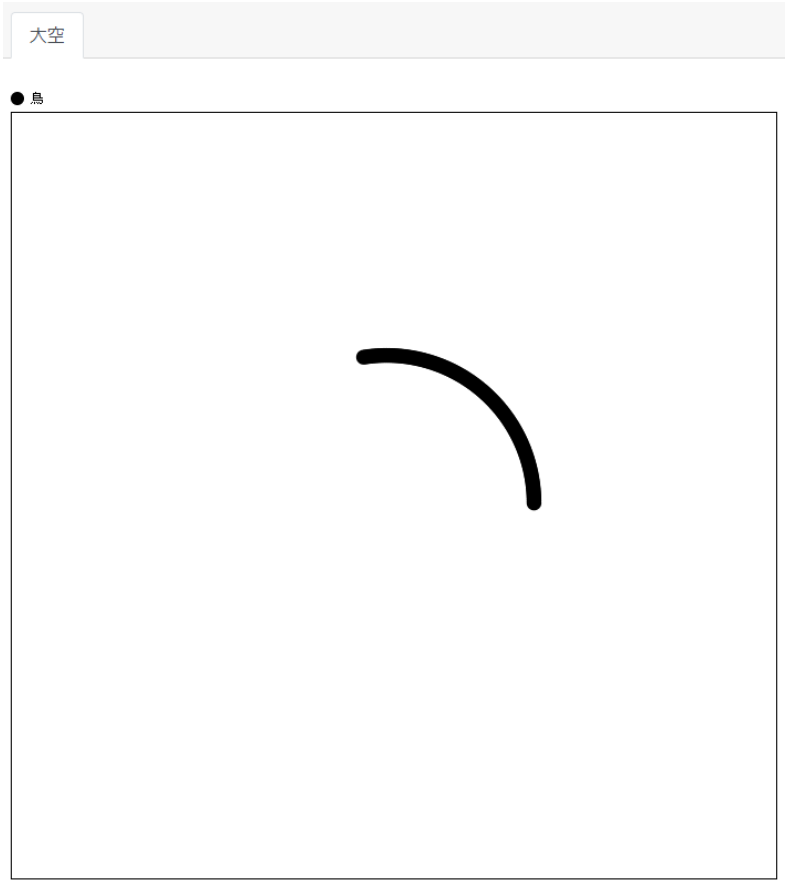
❒4.12 繰り返し文(3)

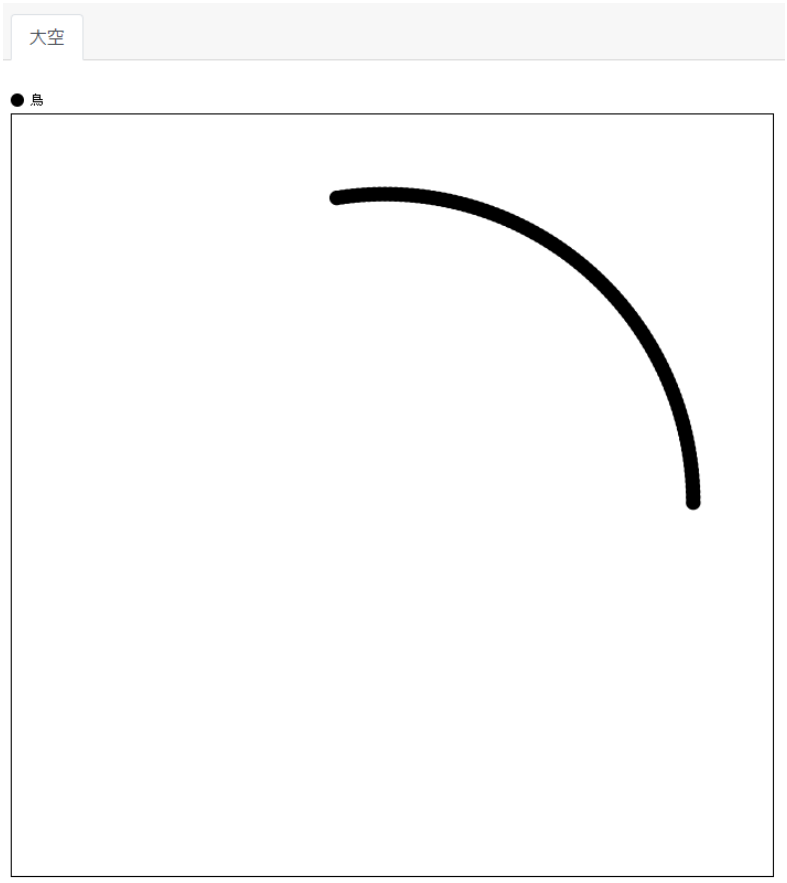
- 「シミュレーション開始時に100エージェントを生成し、それぞれに0度～99度の向きを与える」というルール
  - for文を用いて以下のように書いてみましょう。

```
def univ_init(self):

    for i in range(100):
        one = create_agt(Universe.oozora.tori)
        one.direction = i
```

❒4.12 繰り返し文(4)

■ 鳥が扇形に飛んでいきます。



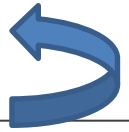


❒4.12 繰り返し文(5)

- for文はより一般的には、変数の値を変化させながら繰り返す処理を意味します。

変数を0からn-1まで変化させながらn回繰り返す処理

```
for 変数 in range(n) :
    変数を用いた処理
```



※処理はインデント内に記述
※変数には慣習としてiをよく使う
（integerの頭文字）

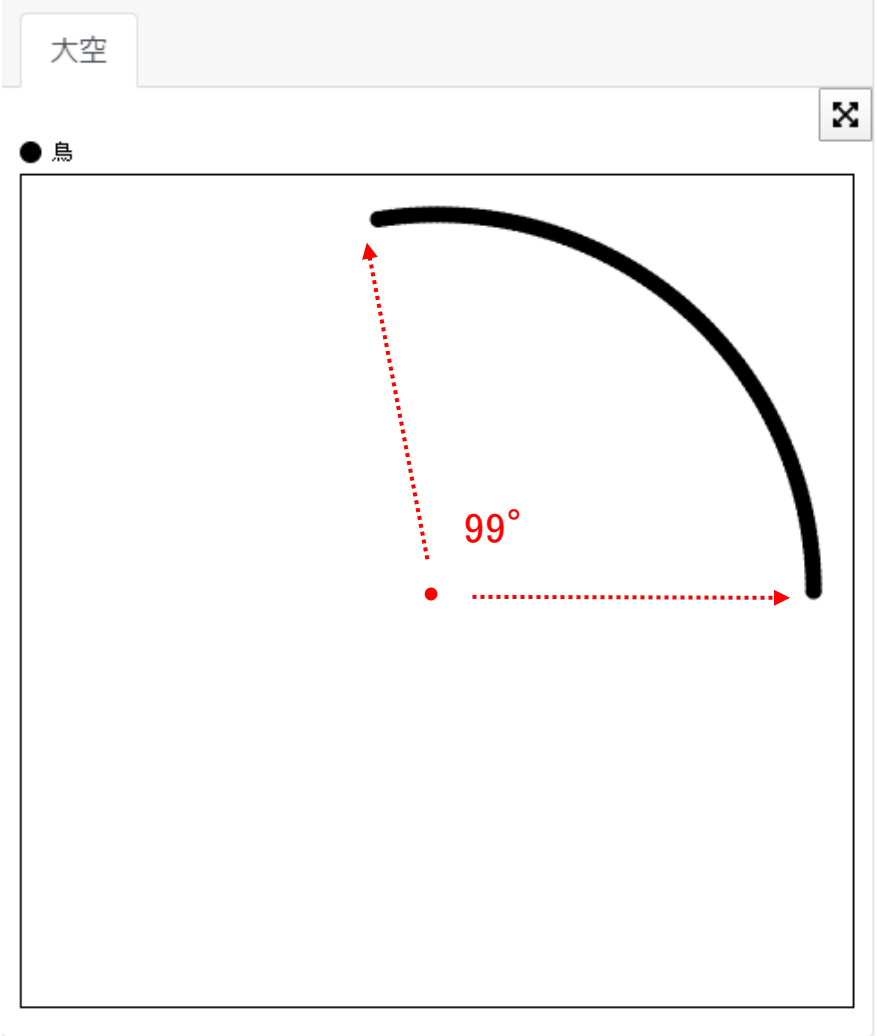
❒4.12 繰り返し文(6)

```
def univ_init(self):

 ①  for i in range(100):
      ②  one = create_agt(Universe.oozora.tori)
      ③  one.direction = i
```

①：iの値を0~99まで変えながら②③を100回繰り返す
②：生成したエージェントを変数oneに代入（create_agt関数の戻り値）
③：エージェントoneの変数directionにiを代入

```
one = create_agt(Universe.oozora.tori) 0回目
one.direction = 0

one = create_agt(Universe.oozora.tori) 1回目
one.direction = 1

…

one = create_agt(Universe.oozora.tori) 99回目
one.direction = 99
```

❐4.12 繰り返し文(7)



❒4.13 変数のデータ型について

- データ型：変数が持つデータの種類のこと
  - 数値だけでなく、様々な種類がある。

主なデータ型

- 整数
- 実数（浮動小数点数）
- 文字列
- 論理値 TrueまたはFalseで表す

- エージェント
- エージェント種別
- 空間

etc・・・

データの集まりを扱うための型

- 集合（セット）
- リスト
- 辞書
- タプル

etc・・・

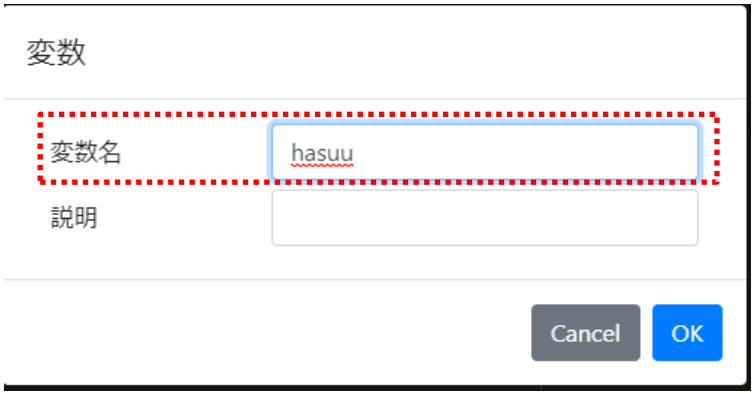
```
a = 3  # 変数aに整数型の値3を代入
b = 4.5  # 変数bに実数型の値4.5を代入
c = “artisoc Cloud”  # 変数cに文字列型の値”artisoc Cloud”を代入
one = create_agt(Universe.oozora.tori)  # toriエージェント（エージェント型）を生成して、変数oneに代入
```

❐4.14 コントロールパネルによる初期値設定(1)

■ 鳥の数をスライドバーの操作で設定できるようにする。

❐4.14 コントロールパネルによる初期値設定(2)

■ 鳥の数を表す変数としてUniverseの下に変数「hasuu」を作成する。



❐4.14 コントロールパネルによる初期値設定(3)

- 変数「hasuu」を鳥の数に設定する。

```
def univ_init(self):

    for i in range(Universe.hasuu):
        one = create_agt(Universe.oozora.tori)
        one.direction = i
```

❒4.14 コントロールパネルによる初期値設定(4)

- Universe変数「hasuu」をコントロールパネルの操作対象に設定する。
  - 出力画面に移動し、実行画面左上の「コントロールパネル」で設定する。
  - 「種類」で「スライドバー」を選択する。

❒4.14 コントロールパネルによる初期値設定(5)

- 鳥の数をスライドバーの操作で設定できるようにする。
  - Universe変数「hasuu」をコントロールパネルの操作対象に設定する。
    - 下図のように項目を設定します。

コントロールパネルで鳥の数を設定できます。

❒4.15 artisoc Cloudにおける変数の種類

- artisoc Cloudの変数は作成する場所でも分類され、以下のように使い分けます。
  - Universe変数　・・・ モデル全体の性質を表す（例：鳥の数）。
  - エージェント変数・・エージェントの性質を表す（例：鳥の位置、向き、速さ）。
  - ローカル変数　・・・ルールエディタ上で一時的に使う変数（例：鳥エージェントone)。
  - 空間変数　　・・・・空間の性質を表す(今回は使いません)。


```
def univ_init(self):

    for i in range(Universe.hasuu):
        one = create_agt(Universe.oozora.tori)
        one.direction = i
```

ローカル変数

第4章：エージェントを動かす（応用編2）

❒4.16 相互作用を含むモデルを作成する(1)

- 鳥が群れながら飛べるのはなぜ？
  - ×リーダーの鳥が指示している。
  - ○個別の鳥が周囲の鳥に飛び方を合わせている。

  →ボイドモデル（Craig Raynolds, 1987）

- ごく単純なボイドモデルを作り、群れながら飛ぶ鳥を表現しましょう。

❒4.16 相互作用を含むモデルを作成する(2) 準備

- Universeのルールを以下のように修正します。
  - 向きを指定しない（コメントアウトする）。
    - 向きはagt_initでランダムに指定される。

- 鳥のルールを以下のように修正します。
  - 初期位置をランダムに
    - 「rand() * 50」で0~50のランダムな値

Universe　メソッド　メソッド選択してくださ

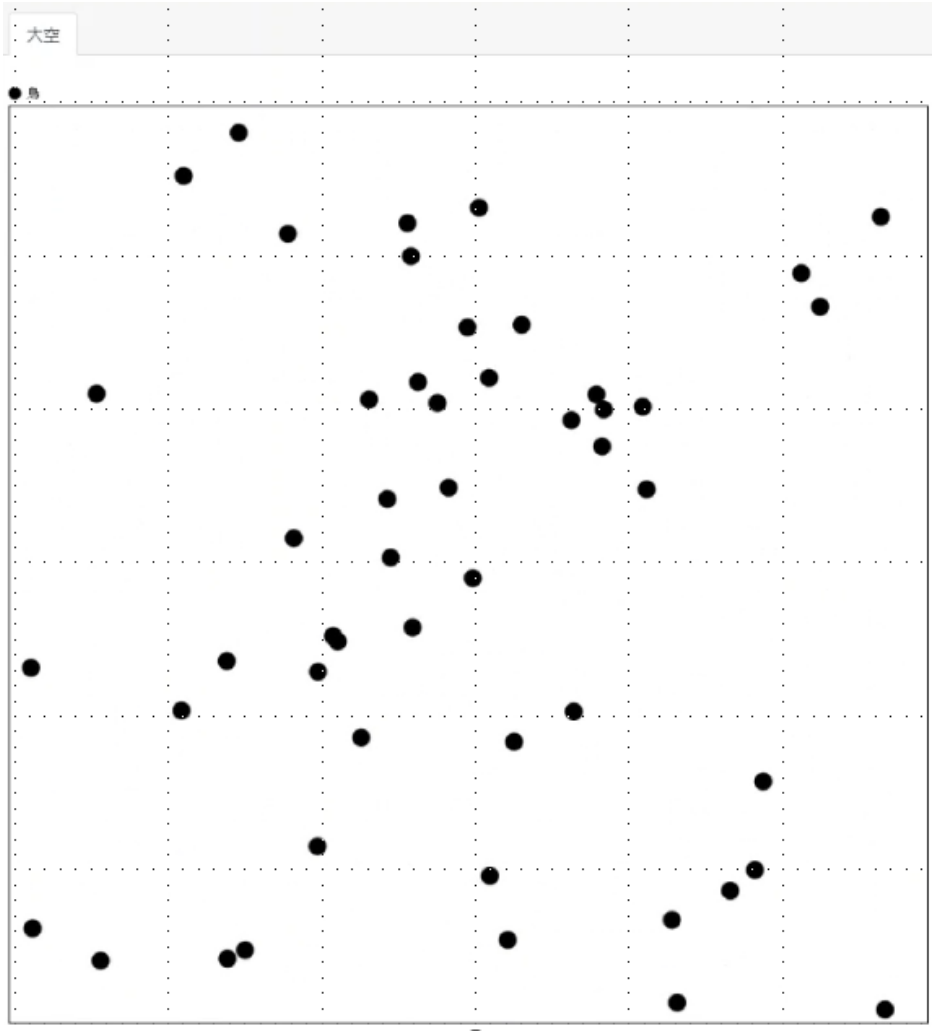
```
1  def univ_init(self):
2
3      for i in range(Universe.hasuu):
4          one = create_agt(Universe.oozora.tori)
5          # one.direction = i
6
```

tori　メソッド　メソッド選択してください

```
1  def agt_init(self):
2
3      self.x = rand() * 50  # ランダムな初期位置
4      self.y = rand() * 50  # ランダムな初期位置
5
6      self.direction = rand() * 360
7
8      self.speed = 1
9
10 def agt_step(self):
11
12     self.forward(self.speed)
13
```

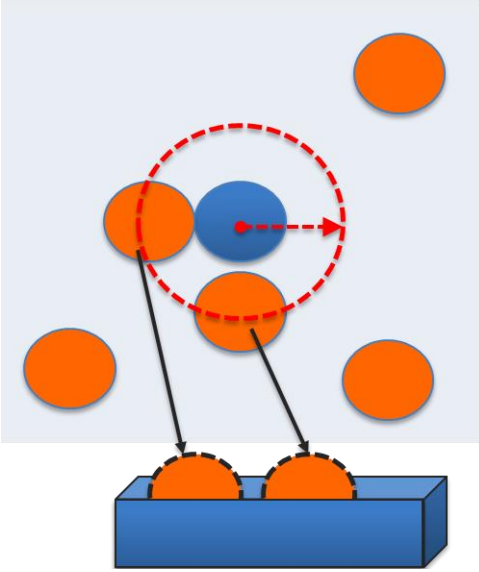
❒4.16 相互作用を含むモデルを作成する(3) 準備

■ 鳥がランダムに飛んでいきます（鳥の数=50）。



❒4.17 周囲の鳥に方向を合わせる(1)

- 鳥エージェントのルールに、周囲の鳥を認識し飛び方を合わせるルールを追記します。
  - 周囲のエージェントを探し、エージェント集合型の変数に格納する。
    - ➡ エージェント集合型変数：エージェントの集合を値として持つ変数
  - エージェント集合型の変数に入っているエージェント数を数える。
    - ➡ 1以上の場合、周囲にエージェントがいる。
  - 周囲にエージェントがいる場合、そのうち1つのエージェントを選び、自分の飛ぶ向きをそのエージェントに合わせる。



エージェント集合型変数
（複数のエージェントを格納するための変数）

❐4.17 周囲の鳥に方向を合わせる(2)

- 周囲のエージェントを取得し、エージェント集合を変数に格納します。
  - agt_stepに記述します。
  - 用いる関数：make_agtset_around_own
    - 視野の範囲内のエージェント集合を取得する。
    - 第1引数：視野
    - 第2引数：自分を含むかどうか（TrueかFalse）

```
def agt_step(self):

    # 自分から距離2以内のエージェントをtori_setに格納（自分自身を含まない）
    tori_set = self.make_agtset_around_own(2, False)

    self.forward(self.speed)
```

❒4.17 周囲の鳥に方向を合わせる(3)

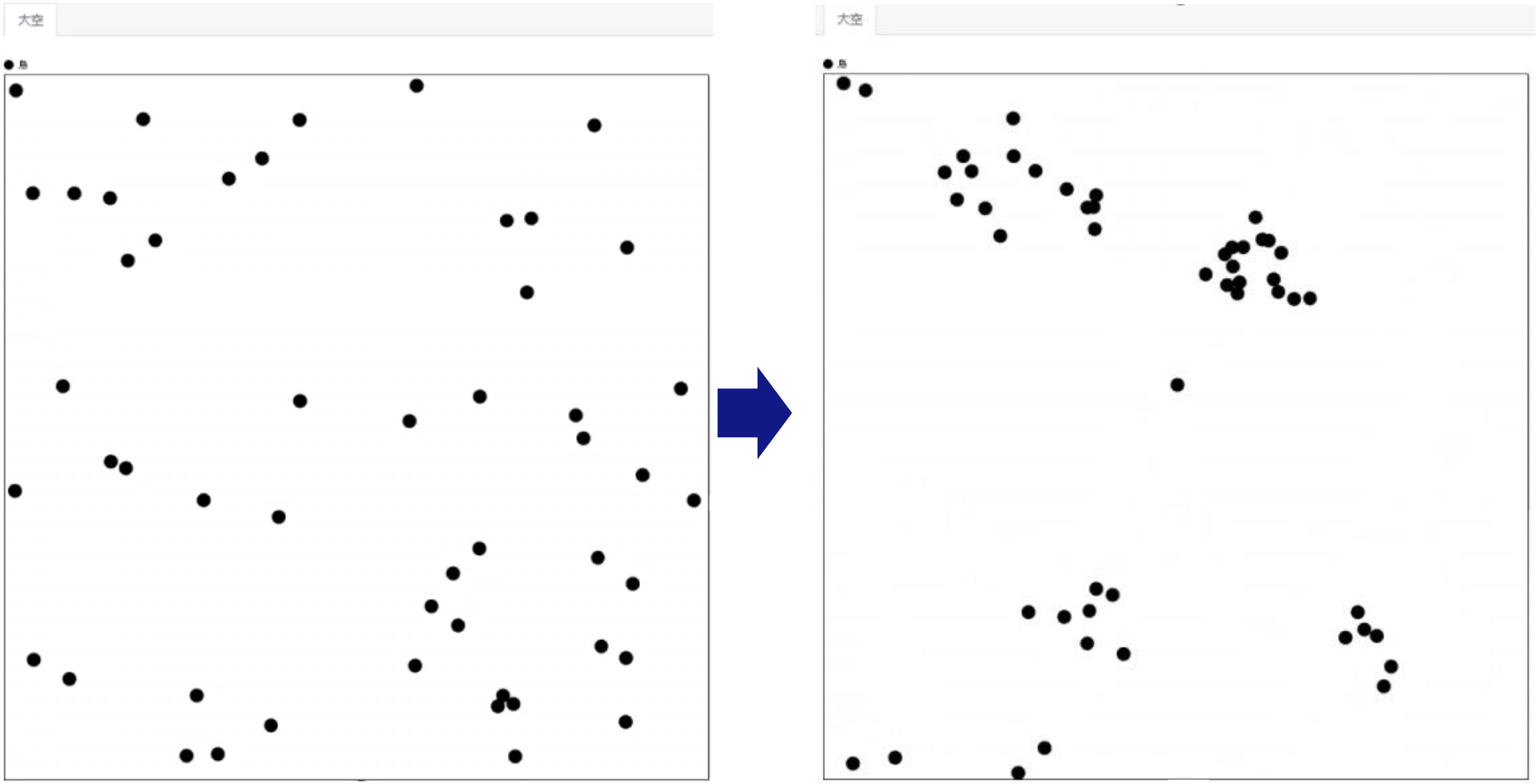
- もし周囲に鳥がいれば、そのうち1羽を選び、自分の方向をその鳥に合わせます。
  - count_agtset: エージェント集合の要素数を数える関数
  - randchoice: エージェント集合からランダムにエージェントを1つ取得する関数

```
def agt_step(self):

    # 自分から距離2以内のエージェントをtori_setに格納（自分自身を含まない）
    tori_set = self.make_agtset_around_own(2, False)

    if count_agtset(tori_set) > 0:  # もし周囲に鳥がいれば
        one = randchoice(tori_set)  # ランダムに1羽を選ぶ
        self.direction = one.direction  # 自分の向きをその鳥に合わせる

    self.forward(self.speed)
```

※ここで、tori_setは順序を持たない集合型。artisoc4でのAgtsetと異なり、位置を指定してエージェントを取り出すことができないことに注意。

❐4.17 周囲の鳥に方向を合わせる(4)

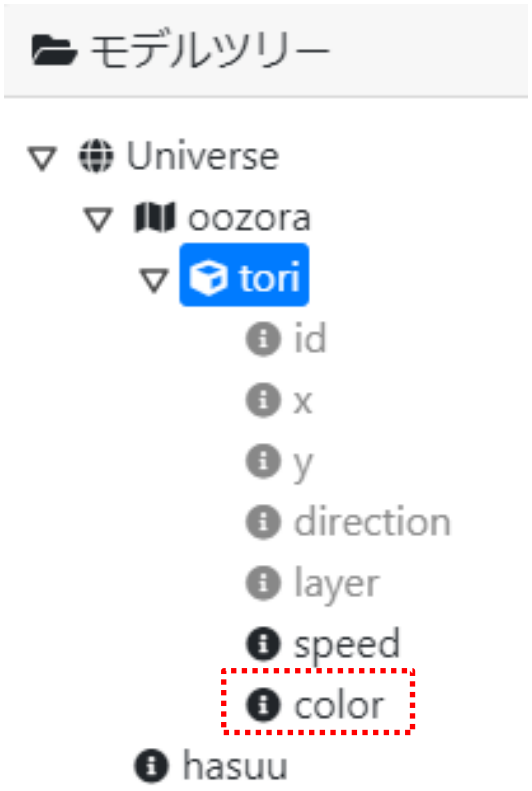
■ 鳥が群れを作ります（鳥の数=50）。



❒4.18 出力を工夫する(1)

- 以下のようなルールを追記します。
  - 群れを形成している場合、エージェントは赤色
  - 群れを形成していない場合、エージェントは青色

- 鳥エージェントに色を表す変数colorを追加します。



❒4.18 出力を工夫する(2)

- 出力設定で変数colorをエージェントの色に指定します。
  - マップの編集→マップ要素リスト（鳥）の編集→エージェント表示色
    - 「固定色」：設定画面で指定した色で表示
    - 「変数指定」：変数に格納された色で表示

❒4.18 出力を工夫する(3)

- ルールで変数colorの値を指定します。
  - 周囲に鳥がいるとき→COLOR_RED
  - 周囲に鳥がいないとき→COLOR_BLUE

～の場合に実行する処理

```
if (条件文) :
    ～条件に当てはまる場合の処理～
else:
    ～条件に当てはまらない場合の処理～
```

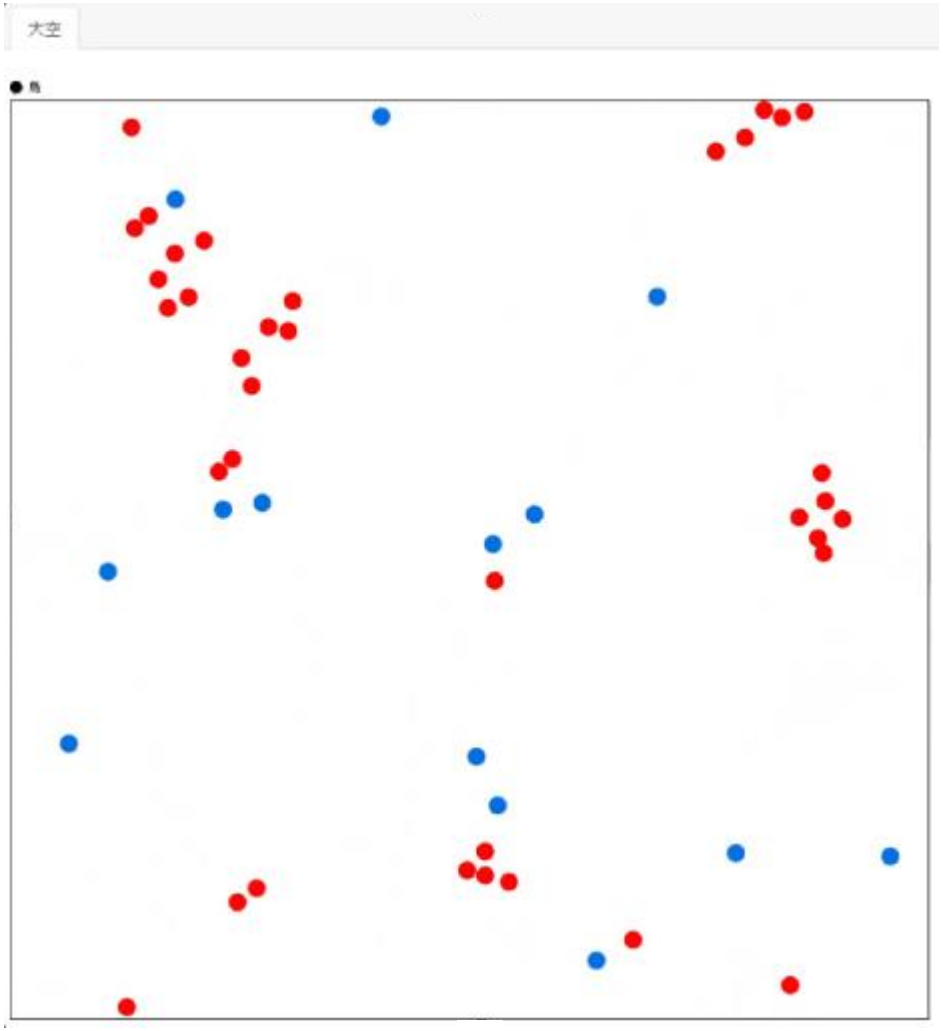
色を表す変数

COLOR_RED	赤色
COLOR_GREEN	緑色
COLOR_BLUE	青色
COLOR_YELLOW	黄色
COLOR_CYAN	水色
COLOR_MAGENTA	紫色
COLOR_BLACK	黒色
COLOR_WHITE	白色

```
def agt_step(self):

    # 自分から距離2以内のエージェントをtori_setに格納（自分自身を含まない）
    tori_set = self.make_agtset_around_own(2, False)

    if count_agtset(tori_set) > 0:  # もし周囲に鳥がいれば、
        one = randchoice(tori_set)  # ランダムに1羽を選ぶ
        self.direction = one.direction  # 自分の向きをその鳥に合わせる
        self.color = COLOR_RED  # 群れを形成するので赤色
    else:                       # 周囲に鳥がいなければ、
        self.color = COLOR_BLUE  # 群れを形成しないので青色

    self.forward(self.speed)
```

❒4.18 出力を工夫する(4)

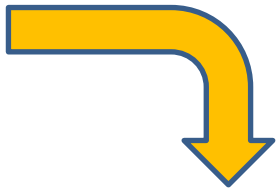
■ 群れを形成しているかどうかによって色が変わります（鳥の数=50）。

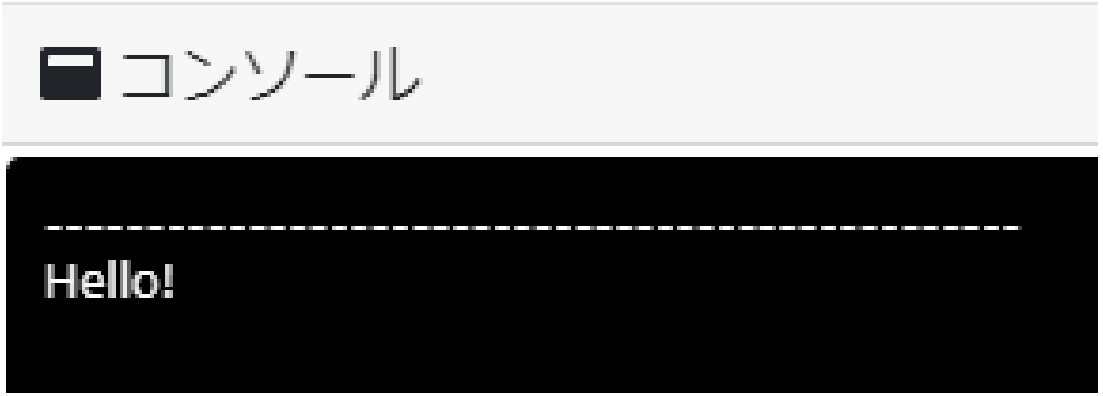


❒4.19 コンソール出力機能(1)

- コンソール画面に任意の文字列を表示します。
  - print: コンソールに文字を表示する関数
  - シミュレーションの状況把握に役立ちます。

```
def univ_init(self):
    print("Hello!")
```





❒4.19 コンソール出力機能(2)

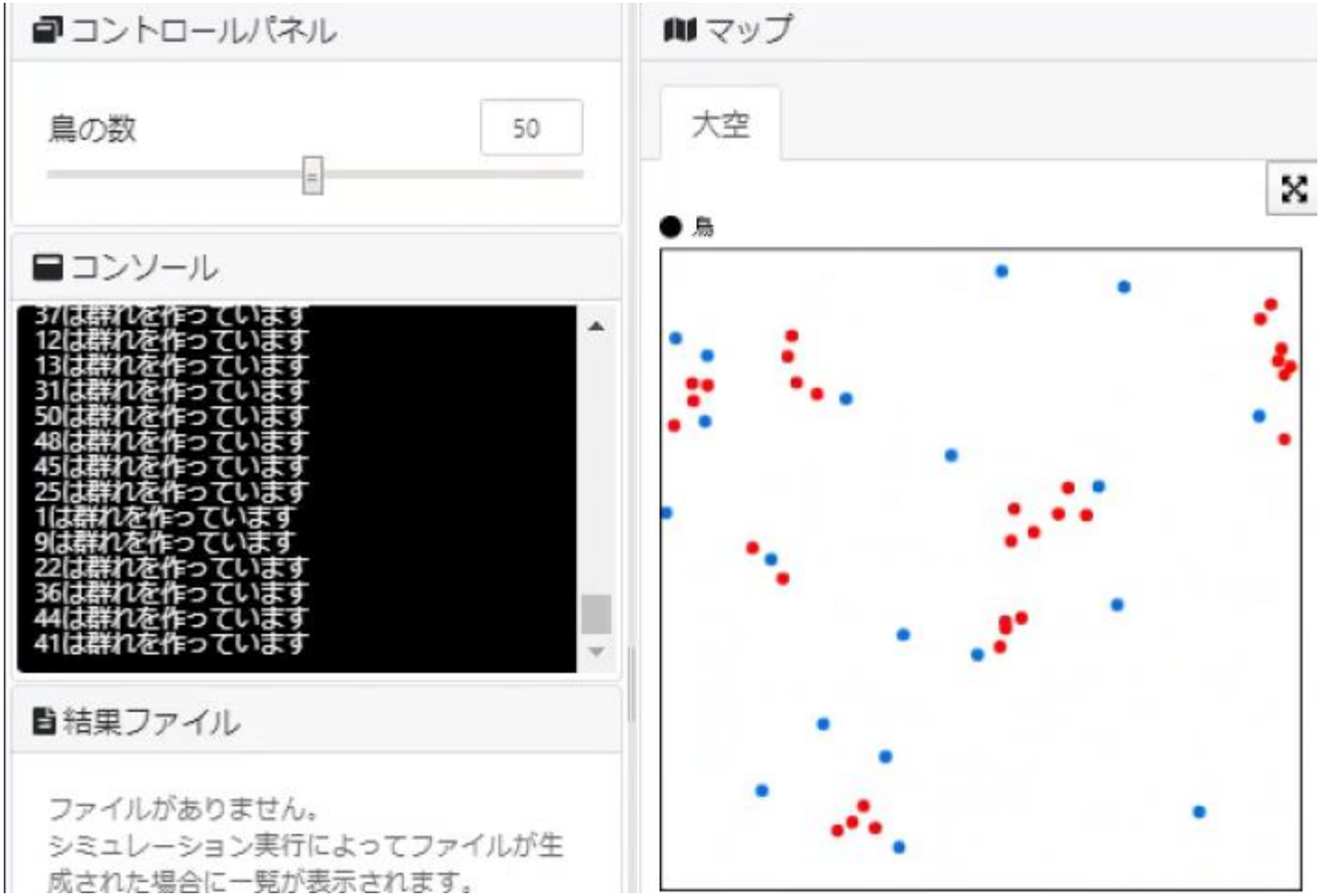
- 群れを作った場合はコンソールに表示する。
  - 「[自分のID]は群れを作っています」とコンソール画面に表示。

```
def agt_step(self):

    # 自分から距離2以内のエージェントをtori_setに格納（自分自身を含まない）
    tori_set = self.make_agtset_around_own(2, False)

    if count_agtset(tori_set) > 0:  # もし周囲に鳥がいれば
        one = randchoice(tori_set)  # ランダムに1羽を選ぶ
        self.direction = one.direction  # 自分の向きをその鳥に合わせる
        self.color = COLOR_RED  # 群れを形成するので赤色
        print(str(self.id) + "は群れを作っています")
    else:                           # 周囲に鳥がいなければ、
        self.color = COLOR_BLUE  # 群れを形成しないので青色

    self.forward(self.speed)
```

※文字列と数値を続けて表示する場合、数値を関数「str」で文字列に変換し、「+」で繋ぐ。
例）　print(“私のIDは” + str(self.id) + “です”)

❒4.19 コンソール出力機能(3)

■ コンソール画面に表示されます。



❐4.20 発展問題(1)

- 向きを完全に合わせるのでなく、少しゆらぎをつけます。
  - 「+=」で左辺に右辺を足します。
  - 「+= rand() * 10 – 5」で-5~5のランダムな値を足します。

```
def agt_step(self):

    # 自分から距離2以内のエージェントをtori_setに格納（自分自身を含まない）
    tori_set = self.make_agtset_around_own(2, False)

    if count_agtset(tori_set) > 0:  # もし周囲に鳥がいれば、
        one = randchoice(tori_set)  # ランダムに1羽を選ぶ
        self.direction = one.direction  # 自分の向きをその鳥に合わせる
        self.direction += rand() * 10 - 5  # 向きにゆらぎをつける
        self.color = COLOR_RED  # 群れを形成するので赤色
    else:                               # 周囲に鳥がいなければ、
        self.color = COLOR_BLUE  # 群れを形成しないので青色

    self.forward(self.speed)
```

❒4.20 発展問題(2)

- 鳥の速度の初期値をランダムにし、速度も周囲の鳥に合わせる。

- 鳥の認識できる範囲を操作できるようにする。
  - 視野の広さを表す変数をUniverseに追加し、コントロールパネルで操作できるようにする。
  - make_agtset_around_own関数でそれを用いる。

第5章：エージェントに判断させる

❒学んだ事項

- ツリーに空間とエージェントを作る（「追加」機能を使う）。
- 空間を見るためにマップ出力設定をする。
- 新規ファイルに名前をつけて保存する。
- ルールを定義する。
- ファイルを上書き保存する。
- 実行ボタン、停止ボタンを押す。

❐5.0 行動の選択が自律性の根本です

- 鳥の飛び方を色々と変えたモデルを作りましょう。
- 状況に応じて異なる行動をさせる基本は「場合分け」です。
- artisoc Cloudでの「場合分け」のルール表記「If文」を学びます。
- 「場合分け」のいくつかの技法を身につけましょう。
- 乱数のいろいろな利用法を学びます。

❐5.1 本格的な「自律的な行動」とは何か(1)

- 強い意味での自律性の例
  - P地点からQ地点への移動。J、K、Lの3つの方法がある。
    - 移動費用の高い順：J、K、L
    - 移動時間の短い順：J、K、L
    - 風景の良い順：K、J、L
  - エージェントが急いでいる場合 → J
  - エージェントに時間の余裕がある場合 → K
  - エージェントの懐が寂しい場合 → L

- エージェントにとっての望ましさの基準にしたがって複数の選択肢から特定のものが選ばれる。

❐5.1 本格的な「自律的な行動」とは何か(2)

- 強いとらえ方での自律的行動
  - エージェントが複数の可能性から1つを選択する。
  - モデル作成者のすること
    - 選択肢（行動のレパートリー）と、
      どのような場合にどの選択肢が選択されるのかというルールを明確に。

- 「場合分け」・「条件分岐」とは？
  - 様々な可能性を設定し、ある場合にはどれが選択されるのかを指定すること。
    - このことについて学ぶ。

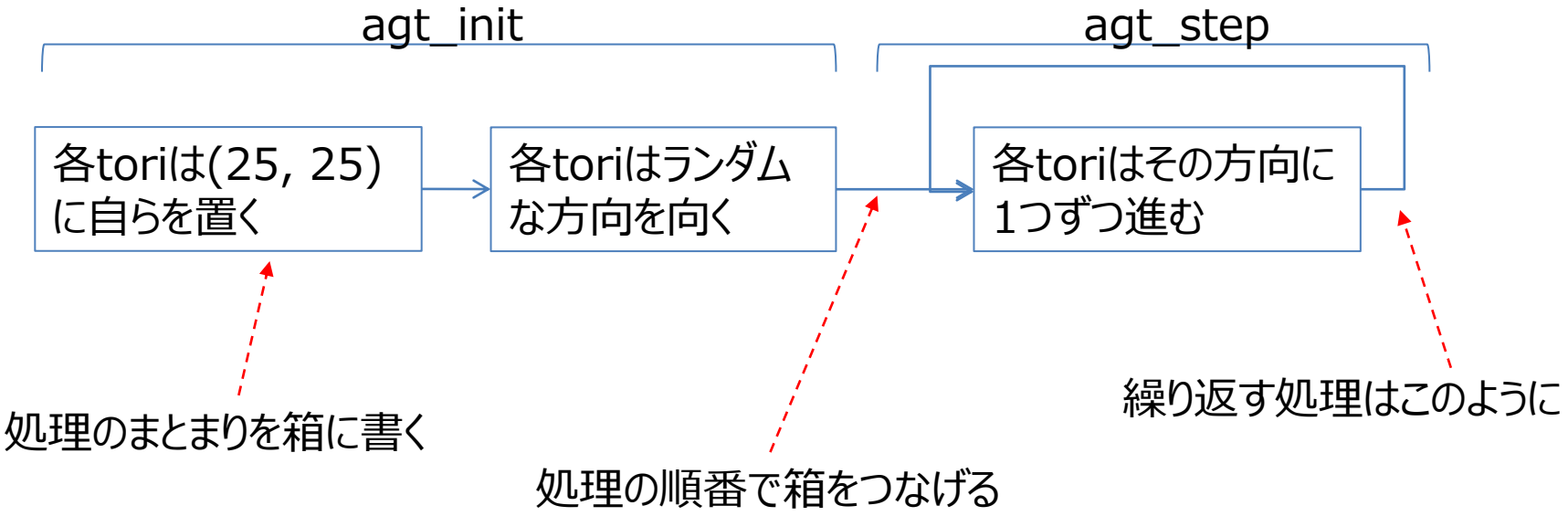
❒5.2 エージェントに何をさせたいのかを図ではっきりと表す(1)

- エージェントの行動ルールを明記すること。

- 2つの段階
  - 1) エージェントに何をさせたいかをはっきりとさせること。
  - 2) シミュレーション用のモデルの一部としてはっきりさせること。
    = artisoc Cloudのルールエディタに正しく書き込むこと。
    - ➡ 前章のtobutoriの場合
      - ☑ 1) toriの初期状態（位置と向き）や飛び方を決める。
      - ☑ 2) artisoc Cloudのルールエディタの中にルールを書き込む。

❐5.2 エージェントに何をさせたいのかを図ではっきりと表す(2)

- 第1段階の作業を、紙に図示する習慣をつける。
  →「フローチャート（流れ図）」
  - 「場合分け」を見やすく書く方法が標準化
  - 「飛ぶ鳥モデル」のフローチャート

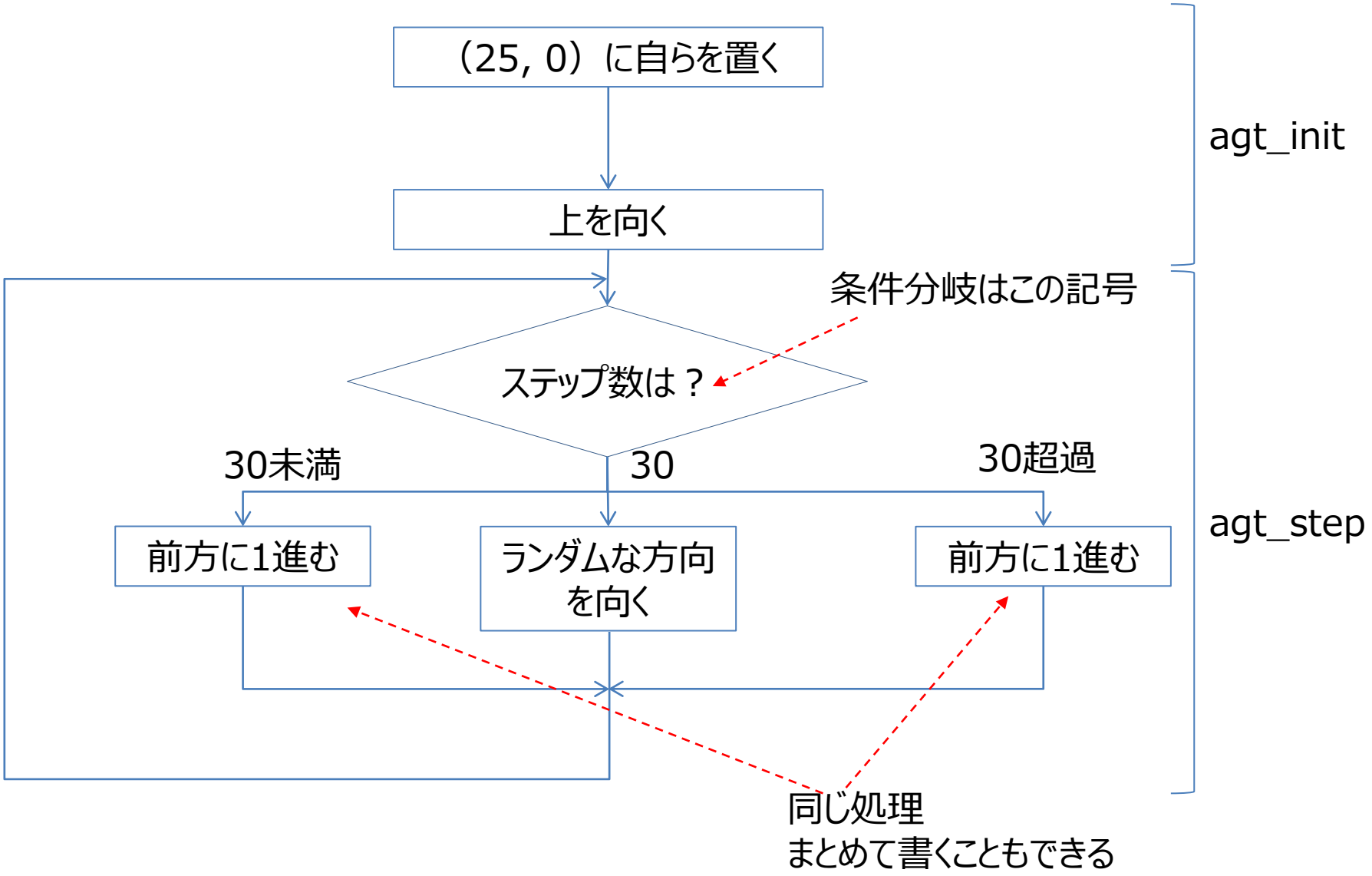


❑5.3「場合分け」を図示して、正しく理解する(1)

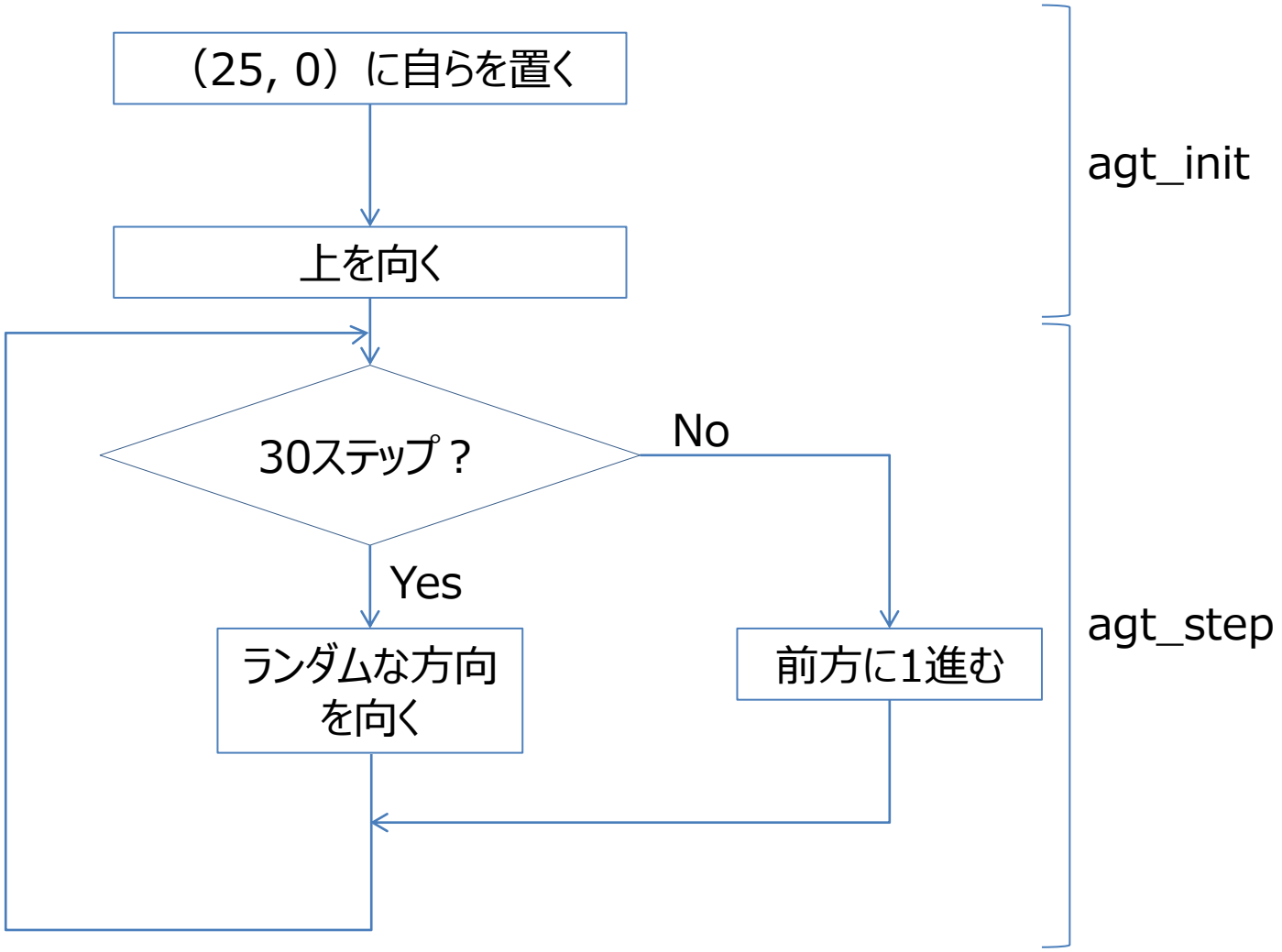
- Toriが中央下から上に飛び上がり、30ステップ以降は各toriは勝手な方向に飛ぶ。
  - 1) oozoraの下端中央（25, 0）にいる。
  - 2)そこで真上を向く。
  - 3) 30ステップになっていなければ、毎ステップ、「1」ずつ、前方に飛び続ける。
  - 4) 30ステップ目に、飛んでいく方向を勝手に決める。
  - 5) 30ステップを越えたら、毎ステップ「1」ずつ、前方に飛び続ける。

- フローチャート化してみる。

❑5.3「場合分け」を図示して、正しく理解する(2)



❐5.3「場合分け」を図示して、正しく理解する(3)



❑5.4「場合分け」ルールでエージェントに判断させる(1)

- フローチャートの内容をもとに、ルールを書いてみましょう。
  以下の手順で、前章のモデルを土台にフローチャートの内容に書き換えます。

- 前回のモデルをもとに、新しいモデルを準備する。
  - 前回のモデルを開く。
  - [継承して新規作成] →コピーを作成する。
  - [基本情報]を編集して、モデル名を花火モデルＡに変える。

良く使う手順！

- ルールエディタでtoriのルールを変更する。

- 実行

❑5.4「場合分け」ルールでエージェントに判断させる(2)
フローチャートの内容をルールエディタに書き込む

- artisoc Cloudにアクセス
  - https://artisoc-cloud.kke.co.jp/
- ログインしていない場合、右上の[ログイン]をクリックしてログイン
- トップページ画面右上の人型アイコンをクリック → [ユーザページ]をクリック → ユーザページに遷移
- ユーザページの作成したモデルから、「飛ぶ鳥モデル」を選択
  ※継承して作成した「飛ぶ鳥モデル」

❐「飛ぶ鳥モデル 4.5 終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

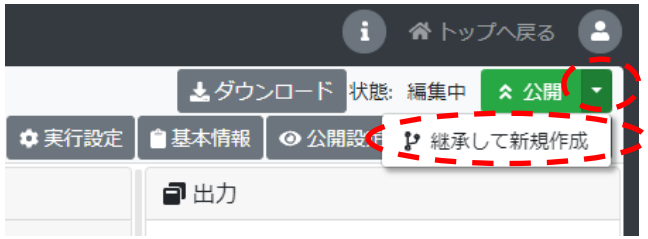
- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/GpySUNkhTsmIRwqWGJQjmw
  - モデル名「飛ぶ鳥モデル 4.5 終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❑5.4「場合分け」ルールでエージェントに判断させる(3)

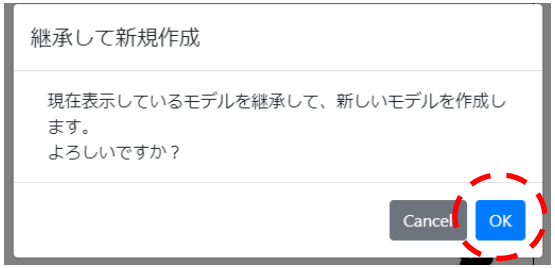
[継承して新規作成] →開いているモデルをもとにコピーを作成する

[継承して新規作成]ボタンを押します

OR



[継承して新規作成]が表示されていない場合、
緑のボタンの右▼をクリックして[継承して新規作成を選択]



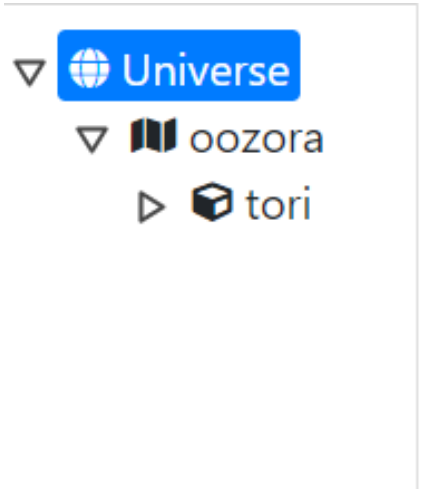
ＯＫを押すと、開いているモデルのコピーがつくられる

[基本情報]を押して、基本情報を編集

モデル名を
「花火モデルＡ」
に変更

❑5.4「場合分け」ルールでエージェントに判断させる(4)

- toriのルールを以下のように設定する。

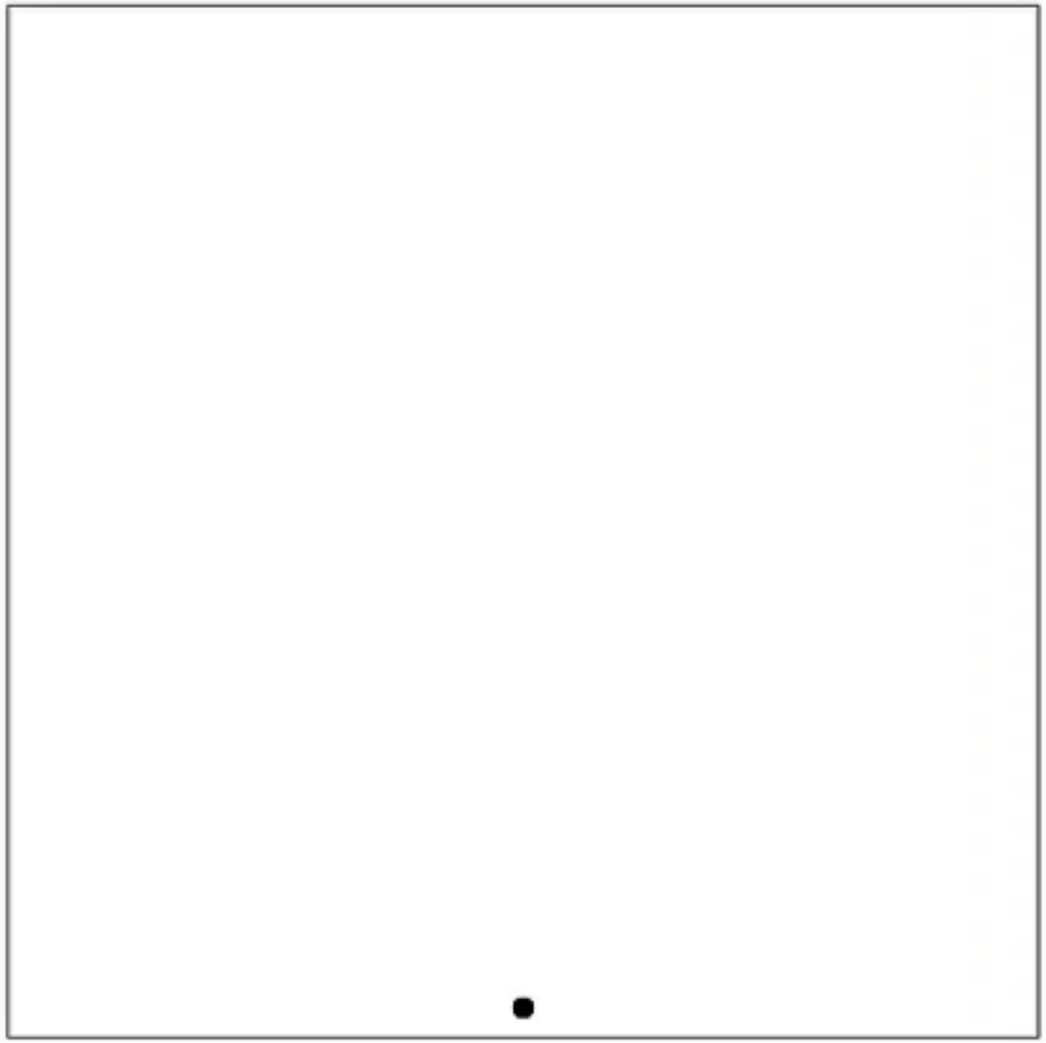



```
def univ_init(self):
    create_agt(Universe.oozora.tori, num=100)

def univ_step_begin(self):
    pass

def univ_step_end(self):
    pass

def univ_finish(self):
    pass
```

実行してみましょう！

```
def agt_init(self):
    self.x = 25
    self.y = 0

    self.direction = 90

def agt_step(self):

    if count_step() == 30:
        self.direction = rand() * 360
    else:
        self.forward(1)
```

❑5.4「場合分け」ルールでエージェントに判断させる(5)



❐5.4「場合分け」ルールでエージェントに判断させる(6)

- 「もし（if）XXXならばYYYを実行し、
  そうでなければ（else）ZZZを実行する」

```
if XXX :
    YYY    ～条件に当てはまる場合の処理～

else:
    ZZZ    ～条件に当てはまらない場合の処理～
```

- count_step() == 30
  - artisocがシミュレーションを実行し始めてからのステップ数
  - ステップ数が30ちょうどであることを示す。
    - “==”と“=”を混同しないこと。

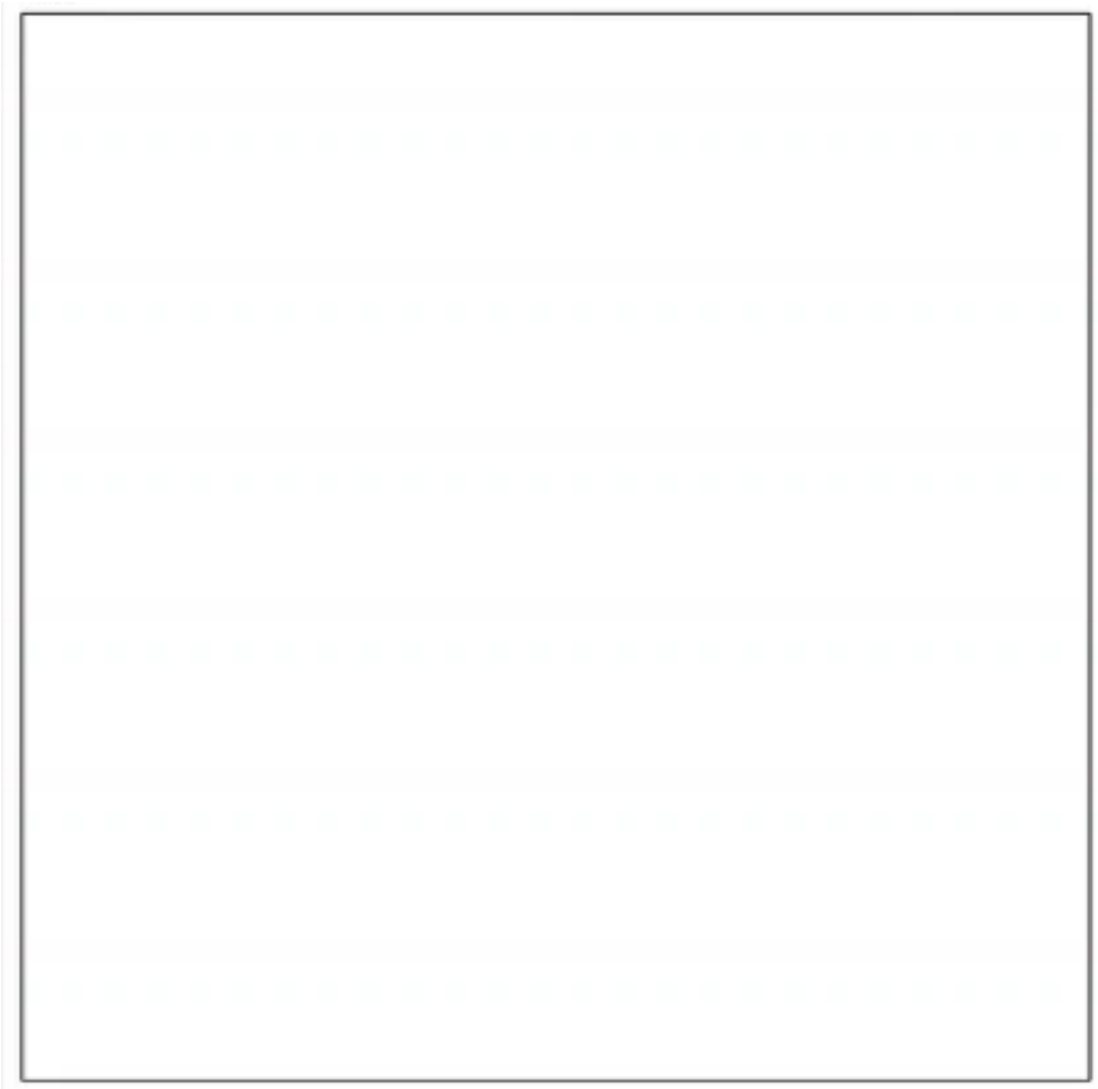
❐5.5「場合分け」のいろいろな技法(1)

- 前モデルをベースに改造
  - 継承して新規作成
  - モデル名は花火モデルB（基本情報で設定）

モデルを継承して新規作成は
p.13を参照

- 新ルール
  - 1) oozoraの下端中央（25, 0）にいる。
  - 2)そこで真上を向く。
  - 3) 30ステップでなければ毎ステップ、「1」ずつ、前方に飛び続ける。
  - 4) 30ステップ目に、 toriは左か右（50%の確率で）の水平から上方30度（つまり0度から30度の間または150度から180度の間）の方角を向く。

❑5.5「場合分け」のいろいろな技法(2)



❐5.5「場合分け」のいろいろな技法(3)

演習：p. 10のフローチャートをもとに、前ページの内容を実行する
フローチャートを書いてみましょう。

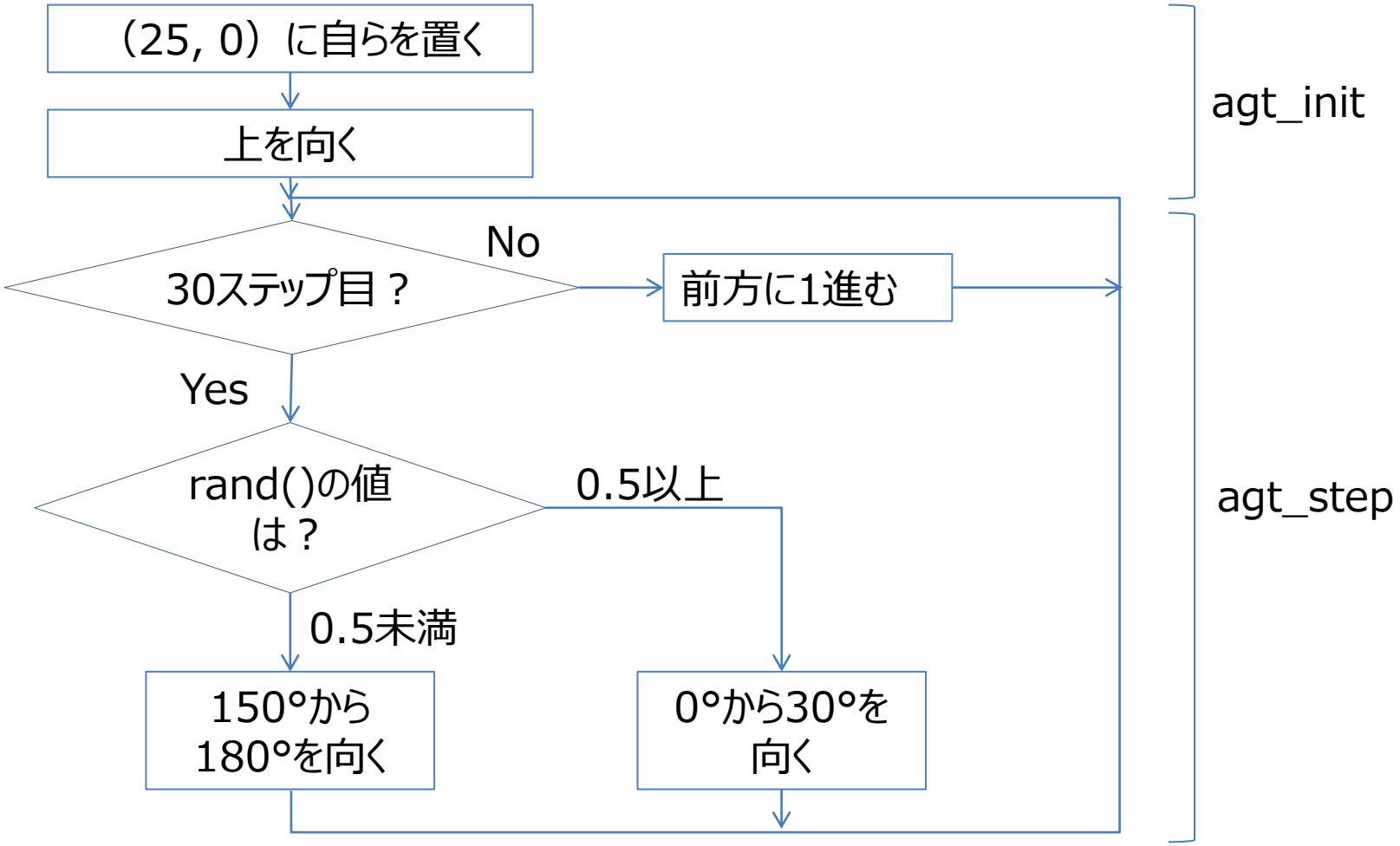
ヒント：ある確率で処理を実行する場合、
前回の講義で使った　rand()関数　を使います
（rand()は0 以上 1 未満の一様乱数を返します）

確率rで実行する処理

```
if rand() < r :
    処理
```

❐5.5「場合分け」のいろいろな技法(4)

回答例



❐5.5「場合分け」のいろいろな技法(5)

演習：作成したフローチャートをもとに、前頁の内容を実行するエージェントの
ルールを書いてみましょう。

ヒント：
フローチャートをよく見ると，場合分けが「入れ子」状態(2重)になっているのに気がつくでしょう。
イフ文を「入れ子」構造にすれば表現できます。

```
if 条件A:
    if 条件B:
        処理～～
    else:
        処理～～
else:
    処理～～
```

インデントに注意！

❐5.5「場合分け」のいろいろな技法(6)

回答例

```
 7 def agt_step(self):
 8
 9     if count_step() == 30:
10         if rand () < 0.5:
11             self.direction = 180 - rand () * 30
12         else:
13             self.direction = rand () * 30
14     else:
15         self.forward(1)
16
17
18
```

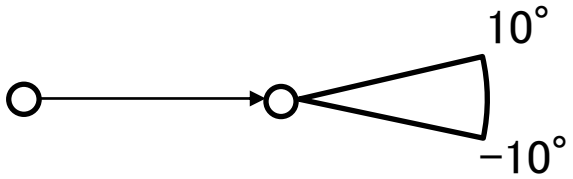
❐5.5「場合分け」のいろいろな技法(7)

- 前モデルをベースに改造
  - 継承して新規作成
  - モデル名は花火モデルＣ（基本情報で設定）

モデルを継承して新規作成は
p.13を参照

- 30ステップ目で左右に分かれ、さらに40ステップ目以降では、毎ステップ左右10度の範囲でランダムに方向を変える。
  - 30ステップ目も40ステップ目も前方に飛ぶようにする。

- 新しいルール表現
  - 条件部分の「＞＝」
    - ➡ 左辺の値が右辺の値“以上”
  - self. turn(rand() * 20 – 10)
    - ➡ self.turn：自分の方向を変える
    - ➡ -10以上10未満の乱数

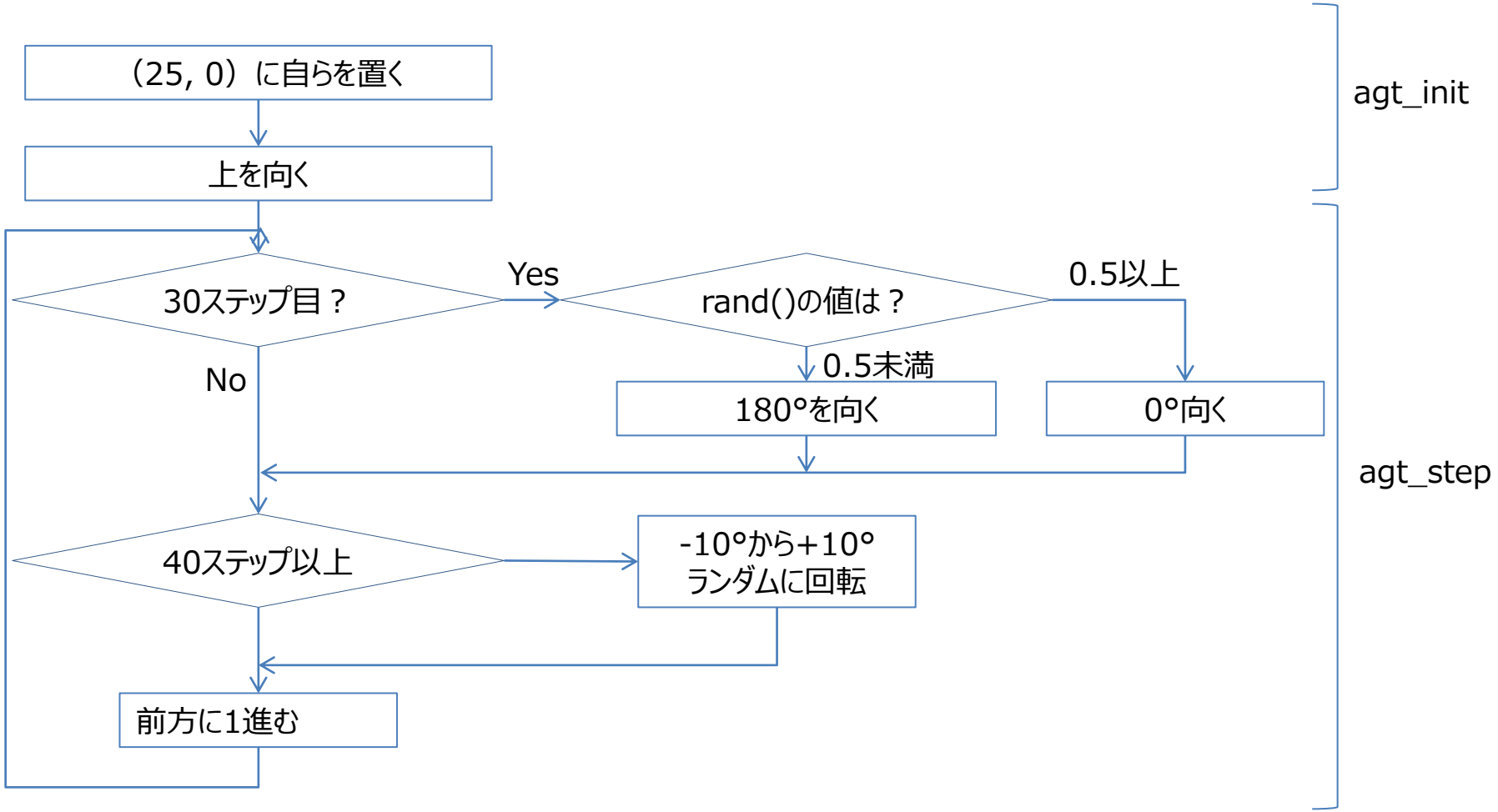


❐5.5「場合分け」のいろいろな技法(8)

演習：前ページの内容を実行するフローチャートを書いてみましょう。

❐5.5「場合分け」のいろいろな技法(9)

回答例



❐5.5「場合分け」のいろいろな技法(10)

演習：作成したフローチャートをもとに、前頁の内容を実行するエージェントの
ルールを書いてみましょう。

❐5.5「場合分け」のいろいろな技法(11)

回答例

```
 7 def agt_step(self):
 8
 9     if count_step() == 30:
10         if rand () < 0.5:
11             self.direction = 180
12         else:
13             self.direction = 0
14
15     if count_step() >= 40:
16         self.turn(rand() * 20 - 10)
17
18     self.forward(1)
```

❐「飛ぶ鳥モデル 5.5 終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/P2Ql4E1bTHSUl5numKRErg
  - モデル名「飛ぶ鳥モデル 5.5 終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❑新しく学んだ事項

- フローチャートの書き方
- イフ文の入れ子構
- 左辺と右辺の関係を表す記号　==　<　>=
- count_step()
- self.turn()
- rand()のいろいろな使い方

❒練習問題

次のようなルールのモデルをつくろう

30ステップ目で左右に分かれたtoriたちが、40ステップ目でランダムな方向を向いて、以降その方向に飛び続ける。

第6章：エージェントに周囲の環境を調べさせる

❐6.0 観察が肝腎です

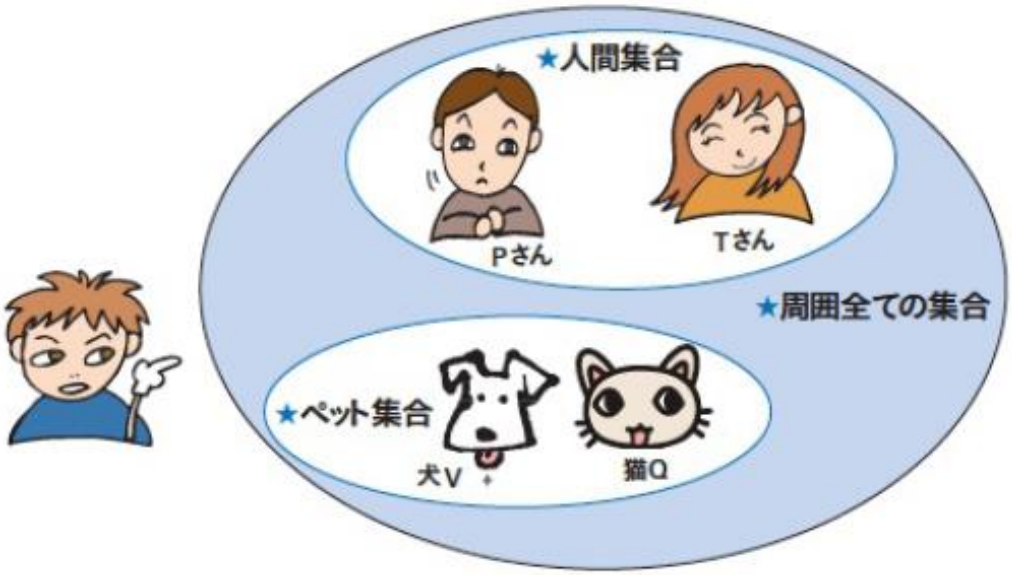
- 近くに人がいると立ち止まって集団ができるモデルを作りましょう。
- いよいよ相互作用をモデル化します。
- エージェント自身に自分の周囲にエージェントがいるかを調べます。
- エージェントに新しい変数を追加します。
- artisoc Cloudには便利な関数が多く用意されているので、簡単にモデルをつくることができます。

❐6.1 周囲の状況に応じた「自律的行動」

- 「エージェントの周囲の環境をエージェント自身が認識して、その認識状況にしたがって行動を選択する」というように、場合分けの条件を各エージェントの個別の条件にすることが重要。
  - MASの特徴のひとつ
    - エージェントは全体についての知識を持たず、自分の周囲の状況だけを知ることができる設定になっている。
    - エージェントごとに周囲の環境は異なるはず。
      → 周囲の環境に応じて異なった行動を各エージェントにとらせる。

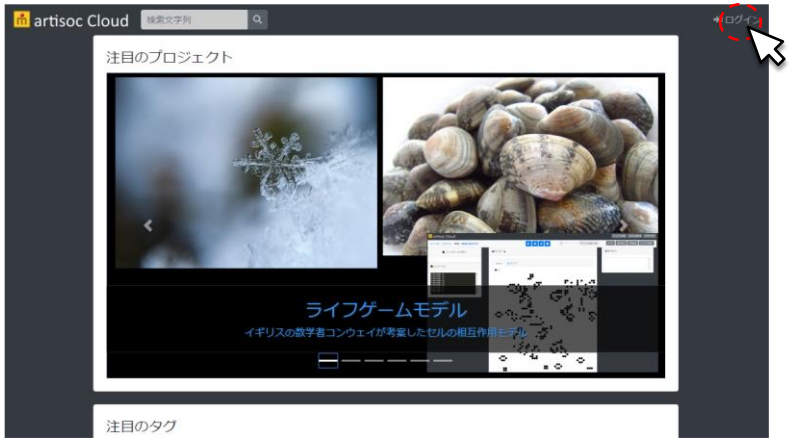
❐6.2 自分の周りのエージェントたち

- 自分の周囲にどのようなエージェントがいるかの認識手順
  1. 「周囲」の具体的な範囲（「視野の広さ」）
  2. 周囲を見回して、そこにいる他のエージェントを「認識」したことを表す変数
     - ➡ エージェント型集合変数
       - ☑ 人間集合 {P, T} 、ペット集合 {Q, V}
       - ☑ 「値」は数値ではなく、
         {P, T} や {Q, V} 、 {P, T, Q, V} といったエージェントの集合



❒準備

- 下記URLにアクセスし、画面右上からログインします。
  - https://artisoc-cloud.kke.co.jp/



- 右上から「新規モデルの作成」をクリックし、モデル名を入力します。
  - 「立ち話モデル」とでも名付けましょう。

❑6.3 新しい変数をエージェントに追加する（1）

- モデルツリーに空間hirobaを作成
- 空間hirobaの下にエージェントhitoを作成

3章参照

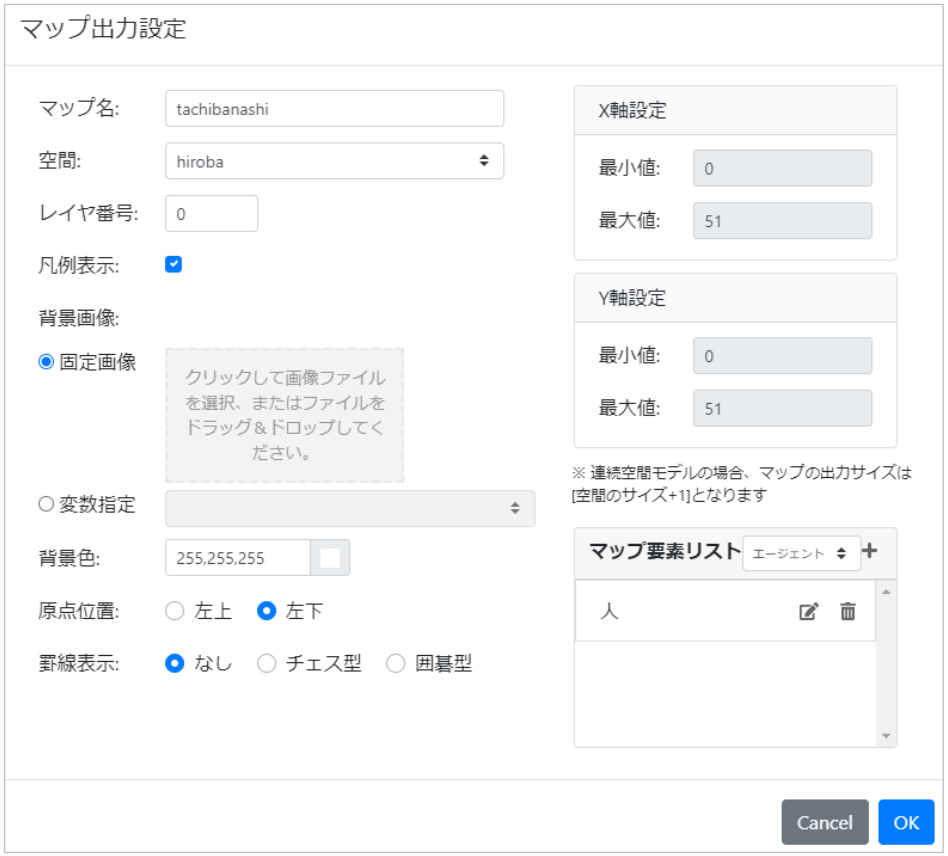
- univ_initで、hitoエージェントを100人生成
- エージェントにmawariという変数を追加

4章参照

  - 「エージェント集合型」
  - 自分の周りにどんなエージェントがいるかを調べた結果を記録しておくための変数

❑6.3 新しい変数をエージェントに追加する（2）

- 出力設定（3章参照）
  - マップ出力
    - 空間hirobaを出力
    - マップ要素リスト　hitoエージェントを出力



❐6.4 周囲にいる人と立ち話をさせる（1）

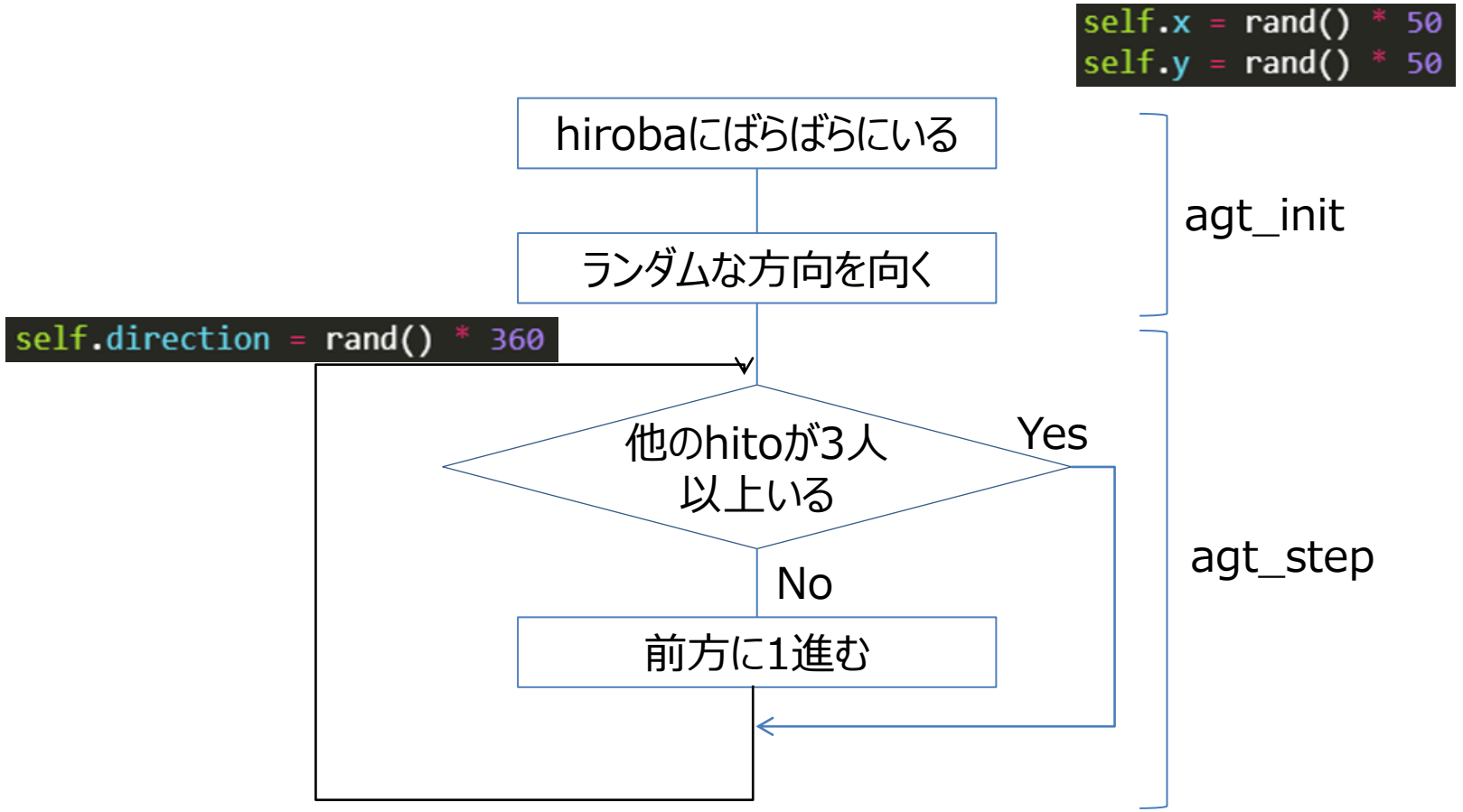
- 広場に歩いている人たちが、近くに集まると立ち止まる。
  - agt init
    - 1)hirobaにバラバラにいる。
    - 2) ランダムな方角を向く。
  - agt_step
    - 3) 毎ステップ、周囲（視野の広さ2）を観察する。
    - 4) 他のhitoが3人以上いるときには、そこで立ち止まる。いなければ「1」ずつ歩き続ける。

フローチャートを書いてみてください。

次ページから解説

❐6.4 周囲にいる人と立ち話をさせる（2）

■ フローチャート



❐6.4 周囲にいる人と立ち話をさせる（3）

- self.mawari = self.make_agtset_around_own(2, False)
  - mawariというエージェント集合型変数に、自分の視野2の範囲の周囲にいるエージェントたちを全て認識させる。
  - Falseは、自分自身を含めない認識方法。
  - “自分自身の”周囲にいるエージェントを探す関数なので、”self.”がつく。

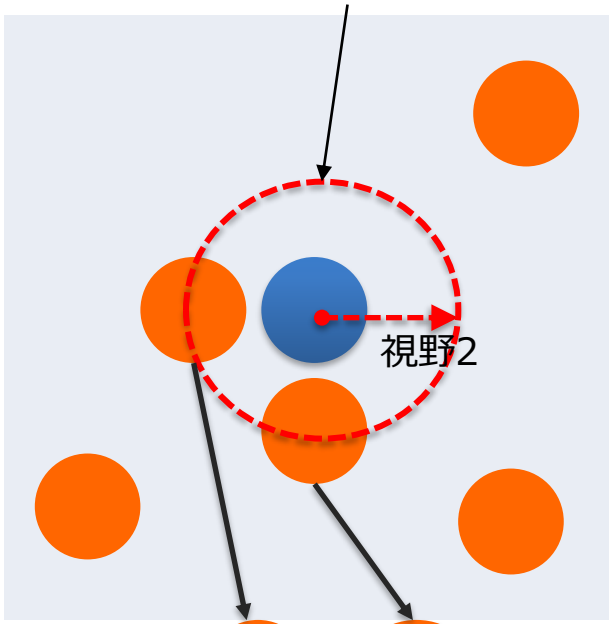
- count_agtset(self.mawari)
  - 括弧内のエージェント集合型変数に認識されているエージェントの数を求めるための関数（整数値をとる）。

- 空文
  - if文で「何もしない」場合、”pass”を入れる。

- グローバル関数とエージェント関数
  - グローバル関数
    - count_step(), count_agtset,など
  - エージェント関数　→自分自身に対して操作する関数
    - self.forward, self.make_agtset_around_ownなど

❐6.4 周囲にいる人と立ち話をさせる（4）

- self.mawari = self.make_agtset_around_own(2, False) のイメージ



エージェント集合型変数
(複数のエージェントを格納するための変数)

mawariというエージェント集合型変数に、自分の視野2の範囲の周囲にいるエージェントたちを全て認識させる。

※自分も視野2以内にいるエージェントだが、mawariには入れない

count_agtset(self.mawari)
→ 2が返る

フローチャートをもとに
ルールを書いてみてください

❑6.4 周囲にいる人と立ち話をさせる（5）

- hitoエージェントのルールを以下のように設定します。

```
1  def agt_init(self):
2      self.x = rand() * 50
3      self.y = rand() * 50
4      self.direction = rand() * 360
5
6
7  def agt_step(self):
8
9      self.mawari = self.make_agtset_around_own(2,False)
10     if count_agtset(self.mawari) >= 3:
11         pass
12     else:
13         self.forward(1)
14
```

エージェントが生成されたときに

ランダムに配置する

ランダムな方向を向かせる

自分の周囲2セルに存在するエージェントを self.mawariに入れる

3 人以上？

立ち止まる

一歩進む

❐6.5 パラメータを変えてみる（1）

- MASの目的の1つ
  - ミクロな状態（エージェントの自律的行動の仕方）が、
    マクロな状態（空間での集団行動のあり方）の変化にどのような影響を与えるか？
  - 視野の広さや立ち話をするときの最少人数を変えることで、
    立ち話集団の数や規模がどのように変わるか？

- パラメータ
  - モデルの基本を変えないで、結果がどのように変わるかを調べるために変化させる変数のこと。

❑6.5 パラメータを変えてみる（2）

- tachibanashiのルールエディタを開く。
  1. 視野の広さを1に減らす、2のままにする、3に増やす。
  2. 最少人数を2人に減らす、3人のままにする、4人に増やす。

- それぞれの場合について、何回か実行・終了してみて、hirobaの全体的な様子の違いを観察してみる。
  - ときたま、いったんたちどまってもまた歩き出すhitoがいる。
    - ➡ 毎ステップ、自分の周囲を観察して行動を変えるので、他のエージェントの行動次第では条件を満たさなくなり、歩き出すことになる。

❐「立ち話モデル 6.5終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/i8YvIo4oRECByP1g1mtkYg
  - モデル名「立ち話モデル 6.5終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❑6.6 モデルを複雑にしよう（1）

- 「立ち話モデル」をもとに新しいモデルを作る。
  - 「立ち話モデル」を[継承して新規作成] →コピーを作成する。
  - [基本情報]を編集して、モデル名を「立ち話モデルＣ」に変える。
  - 保存

- 広場には人だけでなくペットも走り回っている。
  1. 新しくpetというエージェントを100匹追加する。
     - モデルツリーでhirobaを選択>エージェントを追加。名前は”pet”
  2. 出力マップにpetエージェントの出力を追加する。
     - 設定>出力設定>マップ出力設定項目リストからhirobaの  ボタンをクリック
     - >マップ要素リストの追加> pet選択
  3. petエージェントは始めhirobaにランダムにいる。
  4. petの走る方向は毎ステップ「ランダム」。
  5. petは毎ステップ「1」だけ走る。

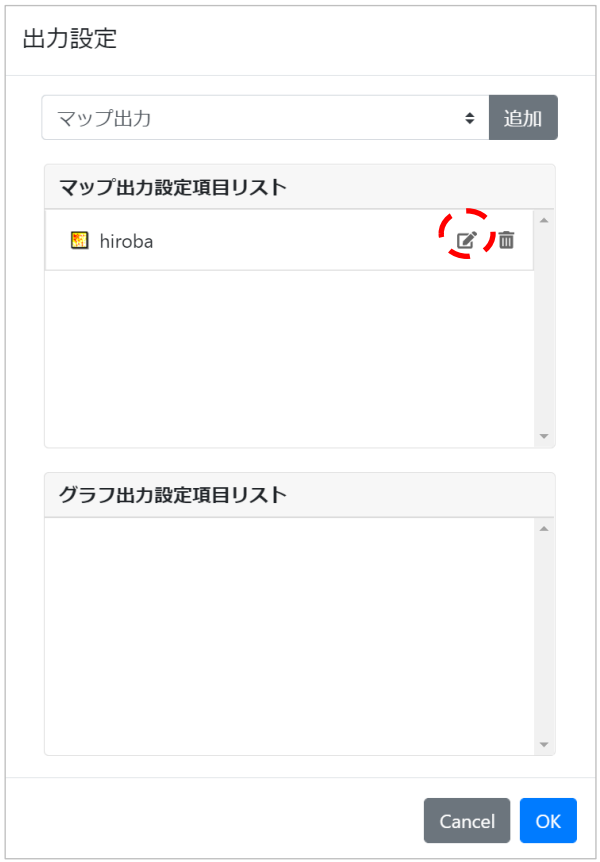
本章を参考に、
自分で作ってみてください

次ページから少し解説

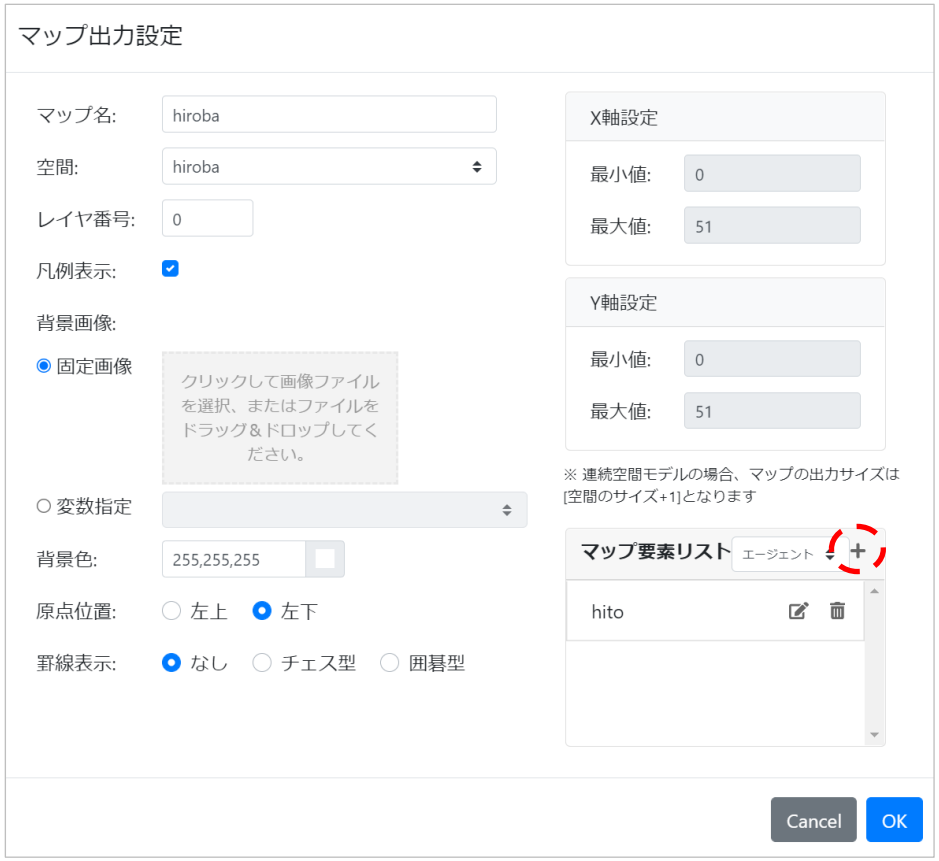
❐6.6 モデルを複雑にしよう（2）

■ 新しい操作：既存のマップ出力に要素を追加する。

出力設定ダイアログ
hirobaマップの編集ボタンを押す。



マップ出力設定ダイアログのマップ要素リスト
＋ボタン（追加）を押す。→マップ要素設定が開く。

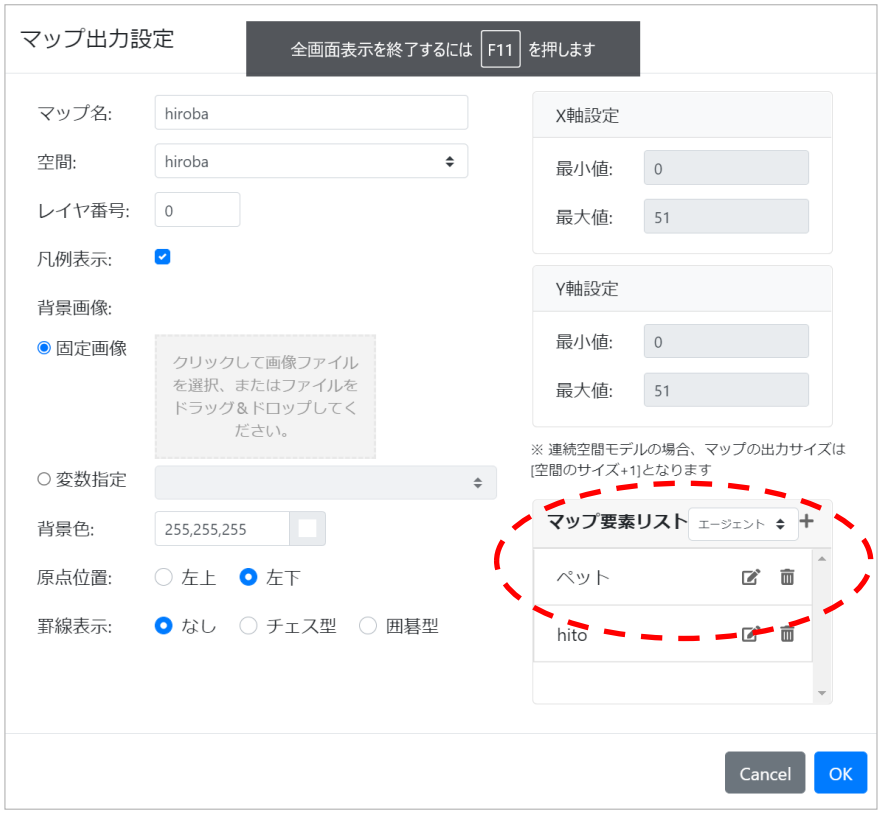


❐6.6 モデルを複雑にしよう（3）

■ 新しい操作：既存のマップ出力に要素を追加する（続き）

マップ要素設定（エージェント）
　要素名「ペット」を入力、エージェントは「pet」を選択
　エージェント表示色を固定色　赤に設定

マップ要素リストにペットが追加されます



❑6.7 1種類のエージェントだけに注目する（1）

- 新しい関数の使い方

```
self.mawari =
self.make_agtset_around_own(2, False, agttype=Universe.hiroba.hito)
```

  - これまでのルール
    - 周囲にいるエージェントすべてを認識していた（hitoとpetを合わせて）。
  - 新ルール
    - hitoのみを認識している。
    - hitoエージェント種の表記 Universe.hiroba.hito
      - ☑ Universeの下に作ったhirobaという空間で行動するhitoエージェント
        - ✓ ちゃんと書かないとコンピュータは実行してくれない。
        - ✓ “.”の前後に空白を入れないこと。

- 組み込み関数
  - artisoc Cloudに準備されている特定の行動をさせるためのルール（make_agtset_around_own やcount_agtsetなど）

- 入力支援機能
  - 途中まで入力すると、組み込み関数、変数がカーソルの下に表示される。

❑6.7 1種類のエージェントだけに注目する（2）

Universeのルール

```
def univ_init(self):
    create_agt(Universe.hiroba.hito, num=100)
    create_agt(Universe.hiroba.pet, num=100)

def univ_step_begin(self):
    pass

def univ_step_end(self):
    pass

def univ_finish(self):
    pass
```

Universe.hiroba.petのルール

```
def agt_init(self):
    self.x = rand() * 50
    self.y = rand() * 50

def agt_step(self):
    self.direction = rand() * 360
    self.forward(1)

```

Universe.hiroba.hitoのルール

```
def agt_init(self):
    self.x = rand() * 50
    self.y = rand() * 50
    self.direction = rand() * 360

def agt_step(self):
    self.mawari = self.make_agtset_around_own(2, False, agttype=Universe.hiroba.hito)
    if count_agtset(self.mawari) >= 3:
        pass
    else:
        self.forward(1)
```

❐「立ち話モデルC 6.7終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/WtfZwiDcSNuUQ4AygHxQMQ
  - モデル名「立ち話モデルC 6.7終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❑6.8「もっと現実的に」はどれだけ本質的か（1）

- 「立ち話モデルＣ」を継承して、「立ち話モデルＤ」を作る。

- 変更するルール

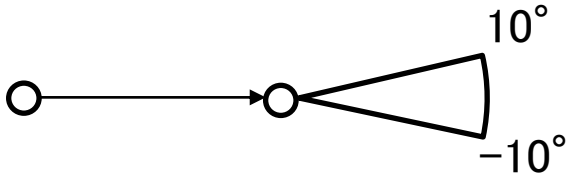
自分でルール変更してみてください

  - hito
    - 毎ステップ左右15度の範囲でランダムに進行方向を変える。
    - 毎ステップ「1」移動
  - pet
    - 毎ステップ左右30度の範囲でランダムに進行方向を変える。
    - petは毎ステップ「2」移動

- ヒント

5章参照

  - self. turn(rand() * 20 – 10)
    - self.turn: 自分の方向を変える。
    - -10以上10未満の乱数



❐6.8「もっと現実的に」はどれだけ本質的か（2）

- 結果
  - hitoとpetの動きの不自然さはなくなるように見える。
  - しかし、全体的な特徴には大きな変化は生まれていない。

- モデルの本質的な部分は、エージェントの細かな動き方ではなく、周囲の環境をどのように認識して行動に結び付けるかにある（視野の広さ、認識するエージェント種、立ち止まる際の人数）。
  - 必ずしも、現実らしいモデルにすればいいというわけではない。

❑新しく学んだ事項

- エージェントに変数を追加する方法
- エージェント集合型変数
- self.make_agtset_around_own
- count_agtset()
- 組み込み関数
- 組み込み関数の入力支援機能
- 出力設定の編集

第7章：モデルの設定値をモデルの外部から操作する

❑7.0「神様」になりましょう

- モデルを外部から操作することができます。
- 操作するためのコントロールパネルを作ります。
- モデルの内部を外部からの操作に対応できるように変えます。
- Universeが重要な「神様」の役割を果たします。
- 「繰り返し」をさせる「for文」を学びます。
- ツリーにない一時的変数を使います。

❑7.1 モデルの条件を簡単に設定し直したい(1)

- 前章までで学んだこと－MASの基本的考え方
  - 1)エージェントに行動させる。
    - 動くルール（方向と速さ）を書き込む。
  - 2)エージェントに自律的行動をさせる。
    - 「場合分け」で異なる行動様式を選択させる。
  - 3)エージェントに周囲の環境を調べさせる。
    - 周囲のエージェントを認識させる。
  - 4)周囲の環境の状態に応じて行動を変える。

- MASを本格的に行うこと
  - 基本的なモデルの中の一部をいろいろ変えてみながらシミュレーションを実行し、結果に現れる変化を観察すること。

❑7.1 モデルの条件を簡単に設定し直したい(2)

- artisoc Cloudに備えられている機能
  - 「コントロールパネル」
    - ➡ エージェント数をモデルの外から設定できる。
    - ➡ 場合分けの条件をモデルの外から設定できる。
  - Universe（エージェントが相互作用する空間よりも上のレベル）で様々な操作をする技法
    - ➡ 「神様」の立場から介入。

❑7.2 エージェント数を自由に変える(1)

- 立ち話モデルを最初から作ってみる。
  - 立ち話モデル２として新規モデルを作成
    - 6.5節までのモデル（petを入れる前まで）で結構です。

以下の説明を参考に
自分で作ってみてください！

参考：6章 p.6

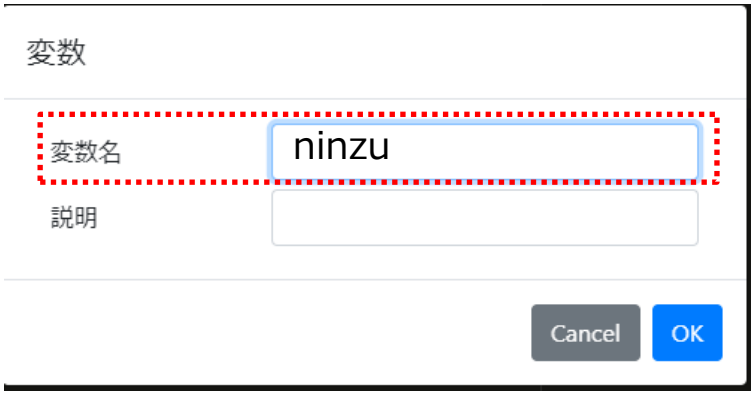
- 以下の変更点を加える。
  - 1)Universeにエージェント数を表す変数を作る。
    - Universe直下に変数を追加する：変数名「ninzu」

参考：4章（応用編1）

  - 2)その変数をモデルの外から操作できるように、コントロールパネルを作る。
    - コントロールパネルの追加
    - 名前：人数、設定対象ninzu、スライドバー、範囲10～200、目盛間隔1
  - 3) Universeのルールエディタに、コントロールパネルで指定したエージェント数だけのエージェントを生成させるルールを書き込む。

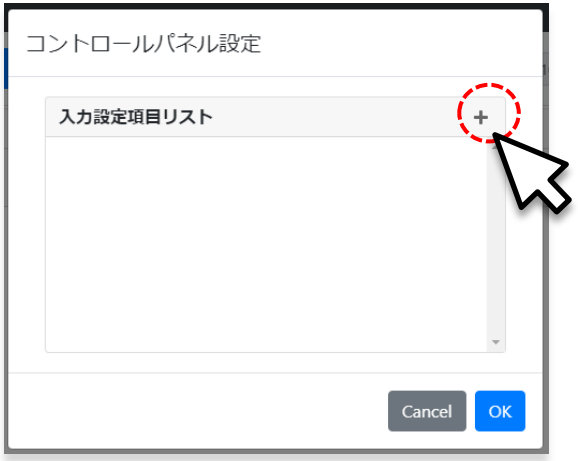
❒7.2 エージェント数を自由に変える(2) Universeにエージェント数を表す変数を作る

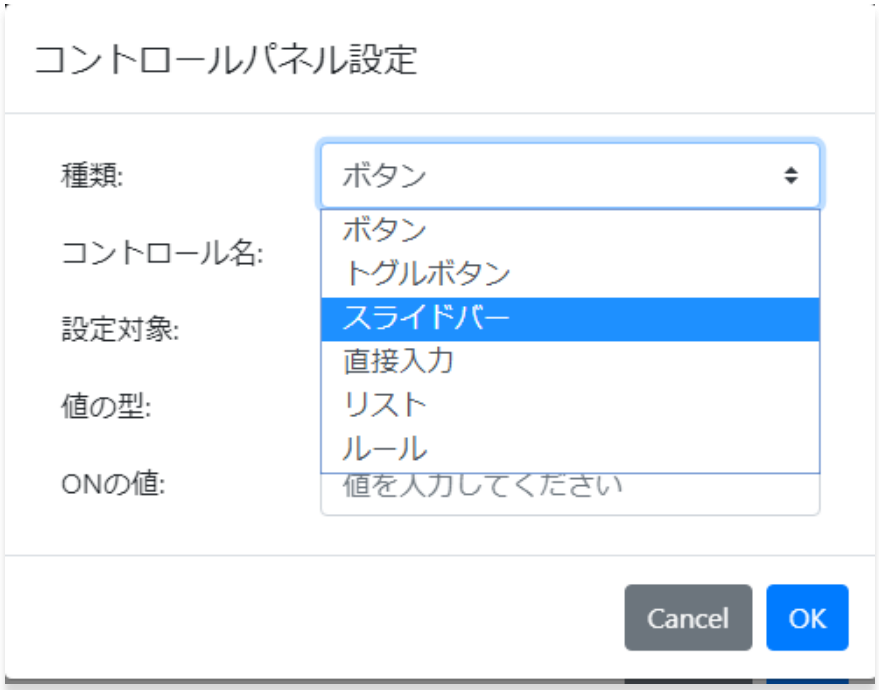
■ 鳥の数を表す変数としてUniverseの下に変数「ninzu」を作成する。



❑7.2 エージェント数を自由に変える(3)
その変数をモデルの外から操作できるように、コントロールパネルを作る

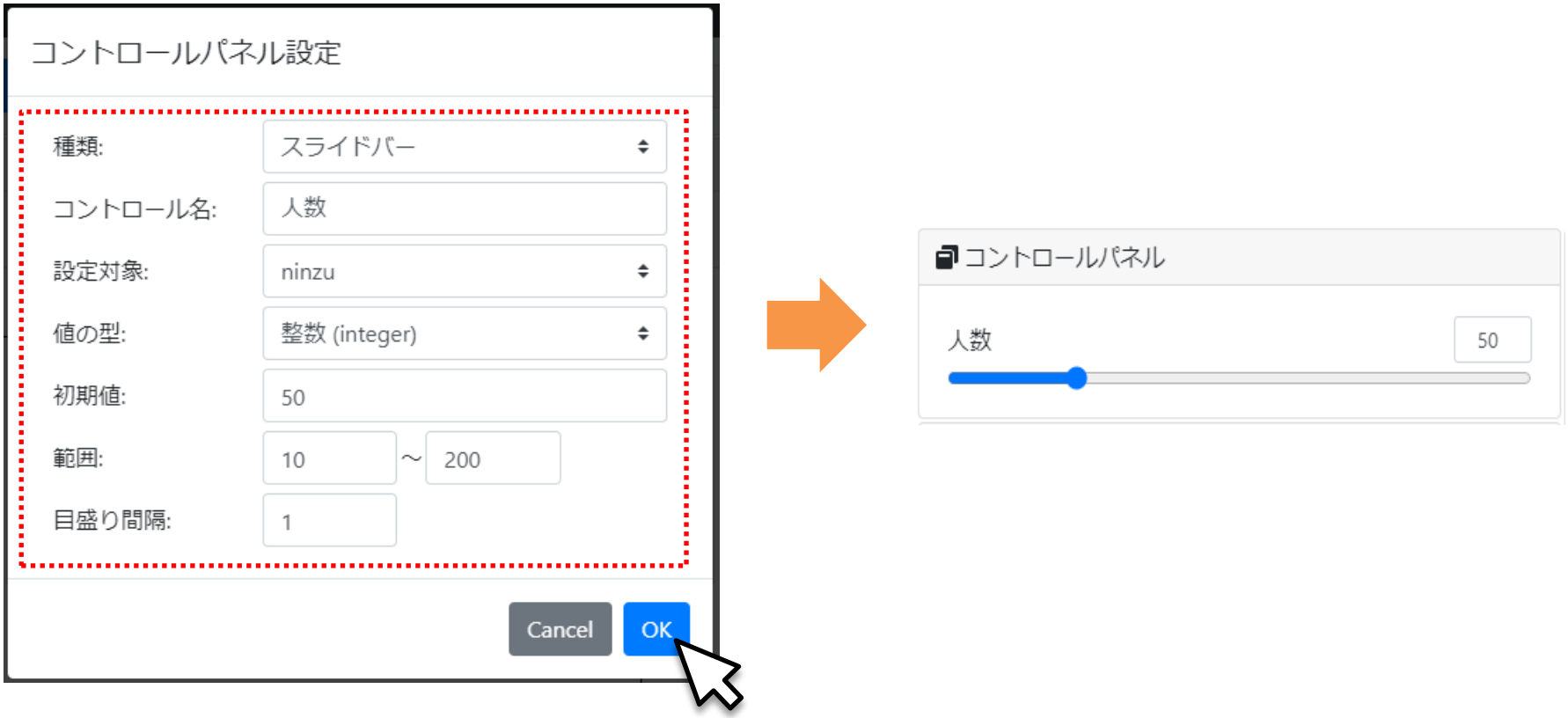
- Universe変数「ninzu」をコントロールパネルの操作対象に設定する。
  - 出力画面に移動し、実行画面左上の「コントロールパネル」で設定する。
  - 「種類」に「スライドバー」を選択する。





❑7.2 エージェント数を自由に変える(4)
その変数をモデルの外から操作できるように、コントロールパネルを作る

- 数をスライドバーの操作で設定できるようにする。
  - Universe変数「ninzu」をコントロールパネルの操作対象に設定する。
  - 下図のように項目を設定する。



❒7.3 モデルの中でエージェントを生まれさせる(1) Universeのルールエディタにルールを書き込む

- 「シミュレーション開始時に100エージェントをまとめて生成する」というルールを考える。
  - 先ほどは「num=100」でまとめて生成しましたが、今回はエージェントを1つ作る処理を100回繰り返します。
  - 繰り返しの処理→**for文**を用います。

n回繰り返す処理

```
for 変数 in range(n) :
    処理
```

※処理はインデント内に記述
※変数は何でもかまいませんが、慣習としてiをよく使います。

```
1  def univ_init(self):
2
3      for i in range(Universe.ninzu):
4          create_agt(Universe.hiroba.hito)
5
6  def univ_step_begin(self):
7      pass
8
9  def univ_step_end(self):
10     pass
11
12 def univ_finish(self):
13     pass
14
```

変数「ninzu」を
生成する人の数に設定

❑7.3 モデルの中でエージェントを生まれさせる(2)
繰り返し文

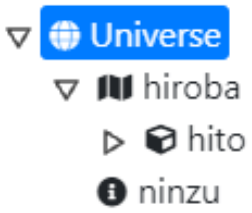
```
1  def univ_init(self):
2
3      ① for i in range(Universe.ninzu):
4          ② create_agt(Universe.hiroba.hito)
5
```

①：iの値を0~99まで変えながら②を100回繰り返す。
②：Universe.hiroba.hitoエージェントを生成

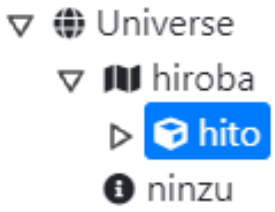
```
create_agt(Universe.hiroba.hito)  0回目
create_agt(Universe.hiroba.hito)  1回目
…
…
…

create_agt(Universe.hiroba.hito)  99回目
```

❑7.3 モデルの中でエージェントを生まれさせる(3)

ルール定義の例




```
def univ_init(self):

    for i in range(Universe.ninzu):
        create_agt(Universe.hiroba.hito)

def univ_step_begin(self):
    pass

def univ_step_end(self):
    pass

def univ_finish(self):
    pass

```




```
def agt_init(self):
    self.x = rand() * 50
    self.y = rand() * 50
    self.direction = rand() * 360

def agt_step(self):
    self.mawari = self.make_agtset_around_own(2, False, agttype=Universe.hiroba.hito)
    if count_agtset(self.mawari) >= 3:
        pass
    else:
        self.forward(1)

```

❐「立ち話モデル２ 7.3終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/BENM7uIaReaLVBD6i6lOcA
  - モデル名「立ち話モデル２ 7.3終了時点」で公開しています。
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る。

❑7.4 モデルの中のパラメータを簡単に変える(1)

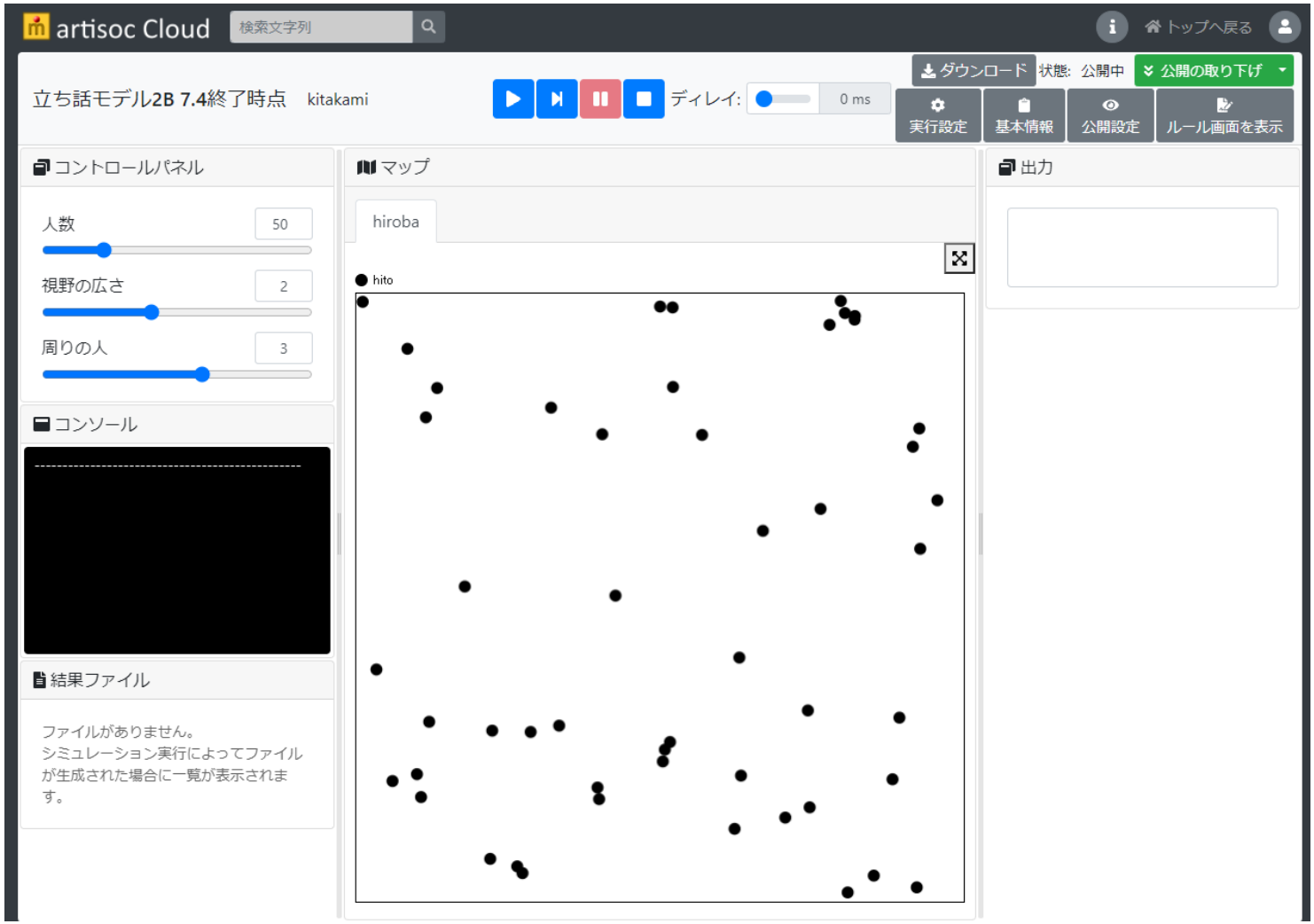
- 視野の広さshiya、行動を左右する人数nakamaを、コントロールパネルで設定できるようにする。
  - 立ち話モデルを継承し、立ち話モデル2Bとして保存する。

以下の説明を参考に
自分で作ってみてください。

- 変更内容
  - 1)Universe直下に「shiya」と「nakama」という変数を追加する。
  - 2)以下の２つのコントロールパネルを追加する。 7章 p.7
    - コントロール名：視野の広さ
      - ☑ 種類:スライドバー、設定対象: shiya, 値の型: 整数(integer) , 初期値: 2, 範囲:0～5, 目盛間隔:1
    - コントロール名：周りの人
      - ☑ 種類:スライドバー、設定対象: nakama, 値の型: 整数(integer) , 初期値: 3, 範囲:0～5, 目盛間隔:1
  - hitoエージェントのルールを開いて下記の通り変更する。
    - ☑ self.mawari = self.make_agtset_around_own(Universe.shiya, False, agttype=Universe.hiroba.hito)
    - ☑ if count_agtset(self.mawari) >= Universe.nakama:
  - コントロールパネルでshiyaやnakamaの値を変化させてみる。

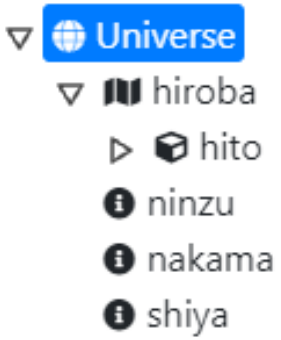
❑7.4 モデルの中のパラメータを簡単に変える(2)

完成したモデル画面

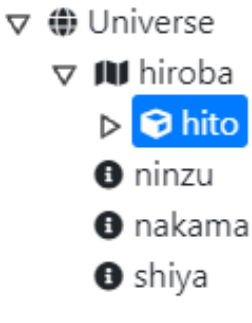


❑7.4 モデルの中のパラメータを簡単に変える(3)

モデルのルール例




```
def univ_init(self):

    for i in range(Universe.ninzu):
        create_agt(Universe.hiroba.hito)

def univ_step_begin(self):
    pass

def univ_step_end(self):
    pass

def univ_finish(self):
    pass
```




```
def agt_init(self):
    self.x = rand() * 50
    self.y = rand() * 50
    self.direction = rand() * 360

def agt_step(self):
    self.mawari = self.make_agtset_around_own(Universe.shiya, False, agttype=Universe.hiroba.hito)
    if count_agtset(self.mawari) >= Universe.nakama:
        pass
    else:
        self.forward(1)
```

❑7.4 モデルの中のパラメータを簡単に変える(4)

- 視野の広さshiya、行動を左右する人数nakamaを、コントロールパネルで設定できるようにする。
  - 立ち話モデルを継承、立ち話モデル2Bとして保存する。

以下の説明を参考に
自分で作ってみてください。

- 変更内容
  - 1)Universe直下に「shiya」と「nakama」という変数を追加する。
  - 2)以下の２つのコントロールパネルを追加する。 7章 p.7
    - ➡ コントロール名：視野の広さ
      - ☑ 種類:スライドバー、設定対象: shiya, 値の型: 整数(integer) , 初期値: 2, 範囲:0～5, 目盛間隔:1
    - ➡ コントロール名：周りの人
      - ☑ 種類:スライドバー、設定対象: nakama, 値の型: 整数(integer) , 初期値: 3, 範囲:0～5, 目盛間隔:1
  - hitoエージェントのルールを開いて下記の通り変更する。
    - ☑ self.mawari = self.make_agtset_around_own(Universe.shiya, False, agttype=Universe.hiroba.hito)
    - ☑ if count_agtset(self.mawari) >= Universe.nakama:
  - コントロールパネルでshiyaやnakamaの値を変化させてみる。

❑7.5 エージェントを生まれさせて、初期配置する(1)

- エージェントが個別に自分の居場所を最初に決めるのではなく、「神様」がエージェントを生み出すとともに配置するようにルールを変更する。
  - 立ち話モデル2Bを継承し、立ち話モデル2Cとして保存する。

- 変更内容
  - Universeのルールとして設定
  - 1)hitoエージェントのルールエディタを開いて、agt_initの右記赤点線内のルールをコメントアウトする。
  - 2)Universeのルールエディタの、univ_init内に右記赤点線内のルールをを付加する。

```
1  def agt_init(self):
2      # self.x = rand() * 50
3      # self.y = rand() * 50
4      self.direction = rand() * 360
5
```

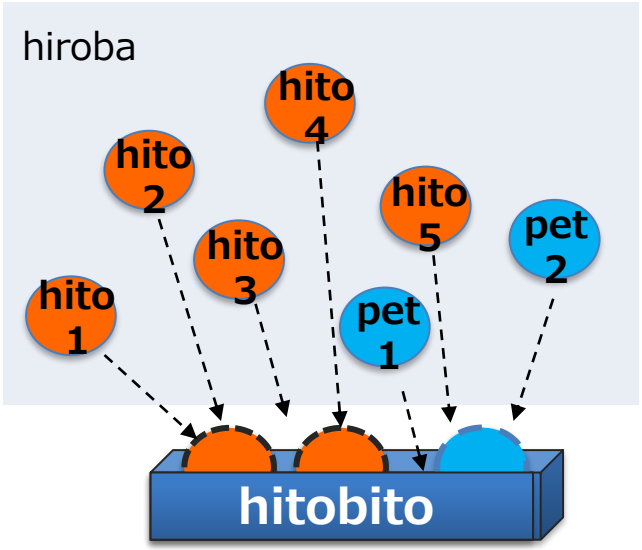
```
1  def univ_init(self):
2
3      for i in range(Universe.ninzu):
4          create_agt(Universe.hiroba.hito)
5
6      # Universe.hirobaにいる人をhitobito変数に入れる
7      hitobito = make_agtset(space=Universe.hiroba)
8
9      # ランダムに配置する
10     random_put_agtset(hitobito)
```

❑7.5 エージェントを生まれさせて、初期配置する(2)

- hitobito = make_agtset(space=Universe.hiroba)
  - hirobaにいる全てのエージェント種をhitobitoという変数の中にしまう操作
    - ➡ hitoエージェント種をすべて集める場合は
      hitobito = make_agtset(agttype =Universe.hiroba.hito)

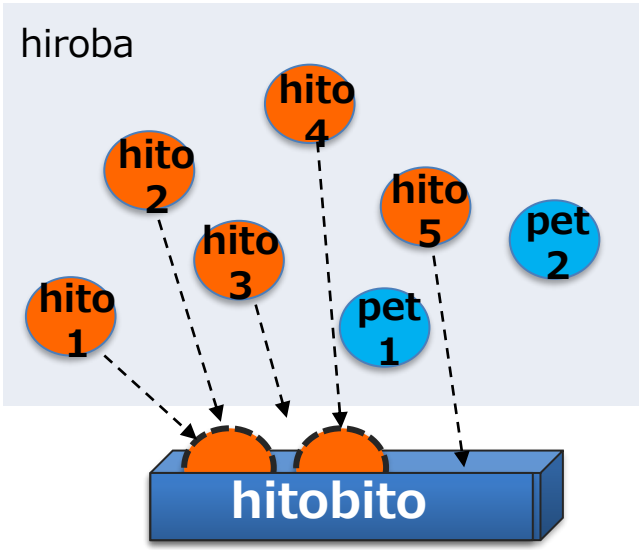
- random_put_agtset(hitobito)
  - hitobito内の全エージェントをランダムに配置

hitobito = make_agtset(space=Universe.hiroba)
hirobaにいるエージェントをすべて集める



エージェント集合型変数
(複数のエージェントを格納するための変数)

hitobito = make_agtset(agttype=Universe.hiroba.hito)
hitoエージェント種をすべて集める



エージェント集合型変数
(複数のエージェントを格納するための変数)

❑7.5 エージェントを生まれさせて、初期配置する(3)

- Universeで設定することのメリット
  - random_put_agtsetは、空間の大きさに依存しない。
    - 空間の大きさを変えても大丈夫。
    - なお、agtのルールとして初期配置をランダムに決めるやり方（7章p.17）の場合は、空間に依存する。

❑新しく学んだ事項

- for文
- create_agt()
- make_agtset()
- random_put_agtset()
- Universe変数の追加
- コントロールパネルの設定
- univ_initへのルールの書き込み
- コントロールパネルを操作してモデルのパラメータを変える
- シミュレーション実行中にコントロールパネルを操作する

❐「立ち話モデル2C 7.5終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/Bgt_bWz9RaGtX7YTrHrpyw
  - モデル名「立ち話モデル2C 7.5終了時点」で公開しています。
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❒練習問題

- 生成したいhitoエージェント数を自由に設定できるように設定を変える。

- 前章で作った立ち話 B モデルを参考にして、立ち話 2 モデルにpetエージェントを追加し、ペットの数やペットの移動速度をモデルの外部から設定できるように修正する。petを適当に動き回らせる。

第8章：シミュレーションの過程をいろいろ出力したり、管理したりする

❐8.0 見ているだけが能じゃない

- シミュレーションの出力はマップだけではありません。
- 時系列グラフに過程を出力させる方法を学びます。
- 1つの時系列グラフにいろいろな指標を出力できます。
- シミュレーション実行を管理することもできます。
- 終了条件を設定しましょう。
- 終了したらコンソール画面で知らせましょう。

❒8.1 シミュレーションの経過をもっと知るには

- これまで
  - 空間のマップ出力と観察
    - →正確な記録はできない？
- マップ出力以外の方法もある
  - 集計作業の実施
  - 時系列グラフとして出力
- シミュレーション管理の方法（初歩）
  - ある条件が満たされたら、シミュレーションを終了させる方法

❑8.2 モデルの中で集計させる(1)

- 立ち話モデル2Cを継承し、立ち話モデル3として保存する。

- シミュレーションの各ステップで、どれだけの人数が立ち止まっているのかを調べる。
  - 1) Universeの下にtachidomariという状態を調べるための変数を追加。
  - 2) tachidomariを時系列グラフとして出力するように設定。
    - ➡ 出力設定>出力項目リストで時系列グラフを追加（次ページ）。

❒8.2 モデルの中で集計させる(2)

❑8.2 モデルの中で集計させる(3)

- （続き）
  - 3) Universeのルールエディタを開く。
    - univ_step_begin：に下記を書込。
      - ☑ univ_step_begin：毎ステップの冒頭に実行される処理を書く。
      - ☑ 毎ステップtachidomariをゼロに初期化。
        - ✓ → そのステップで何人のエージェントが立ち話をしているのかを数えることが可能になる。

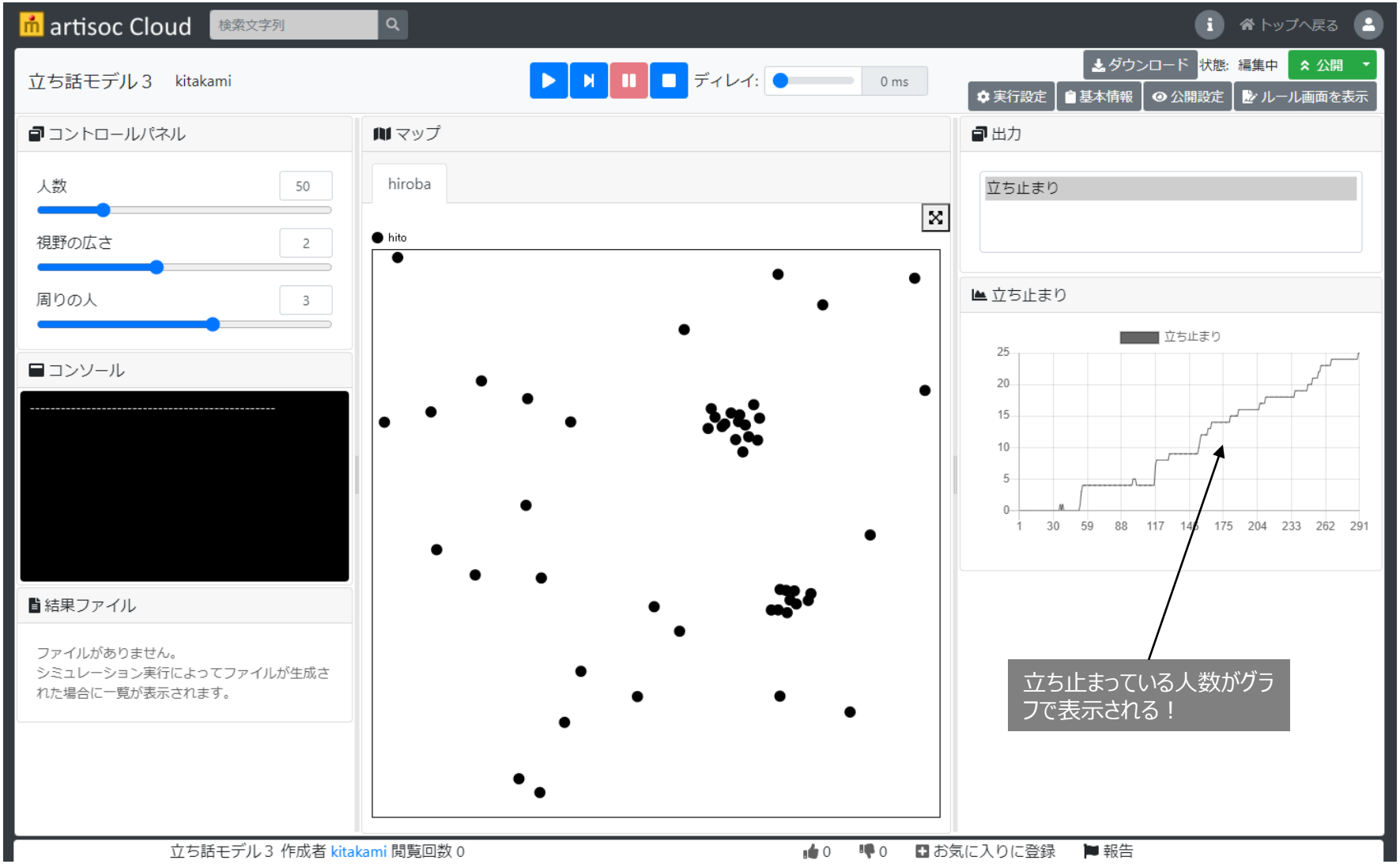
```
1  def univ_init(self):
2
3      for i in range(Universe.ninzu):
4          create_agt(Universe.hiroba.hito)
5
6      # Universe.hirobaにいる人をhitobito変数に入れる
7      hitobito = make_agtset(space=Universe.hiroba)
8
9      # ランダムに配置する
10     random_put_agtset(hitobito)
11
12 def univ_step_begin(self):
13
14     # Universe変数 tachidomariを毎ステップ初期化
15     Universe.tachidomari = 0
16
17 def univ_step_end(self):
18     pass
19
20 def univ_finish(self):
21     pass
22
23
```

❑8.2 モデルの中で集計させる(4)

- (続き)
  - 4) hitoのルールエディタを開いて、下記のルールを挿入する。
    - 自分のまわりに一定数以上の人がいたら、hiroba全体で立ち止まっている人数を１（自分の数）だけ増やすルール。
      - ☑ これは自己申告型のルール。hitoエージェントが「神様」に「＋1とカウントして下さい」と申告して集計するイメージ。
      - ☑ なお、「神様」がステップの最終時点で大域的に見渡して集計する方法もある（この場合は、自己申告型と違う結果になる可能性がある）。
    - 自分のまわりに一定数以上の人がいない場合は、そのまま1歩進む。
  - 保存して実行。

```
def agt_init(self):
    # self.x = rand() * 50
    # self.y = rand() * 50
    self.direction = rand() * 360

def agt_step(self):
    self.mawari = self.make_agtset_around_own(Universe.shiya, False, agttype=Universe.hiroba.hito)

    if count_agtset(self.mawari) >= Universe.nakama:

        # Universe変数 tachidomariを1増やす
        Universe.tachidomari = Universe.tachidomari + 1

    else:
        self.forward(1)

```

❒8.2 モデルの中で集計させる(5)



❐「立ち話モデル３ 8.2終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/92J2c9OcQ_2jA3Dlgmg6kA
  - モデル名「立ち話モデル３ 8.2終了時点」で公開しています。
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る。

❑8.3 時系列グラフの賢い利用法(1)

- 1つの時系列グラフに色々な数値を表示することも可能。
- 立ち止まっているエージェント数の割合(%)を出力する。
  - 立ち話モデル3を継承し、立ち話モデル3Bとして保存する。

- 変更内容
  - 出力設定>時系列グラフ・編集>新しい要素の追加
    - 要素名　wariai
    - 出力値　100 * Universe.tachidomari / Universe.ninzu
  - ※複数の要素を1つのグラフで表示すると見づらくなる可能性がある。
    - → 補正する必要がある（後者を100で割るなど）。

❒8.3 時系列グラフの賢い利用法(2)

❑8.4 自動的にシミュレーションを終了させる

- ある条件を満たせばシミュレーションが終了するようにする。
  - 立ち話モデル3Bを継承し、立ち話モデル3Cとして保存する。

- 「全エージェントが止まったら終了」とする。
  - univ_step_endに下記ルールを書込。
    - univ_step_end：毎ステップ終了時に実行されるルール
    - exit_simulation：シミュレーションを終了させるルール
    - hitoの数（ninzu）とtachidomariの数が等しい（==）ときに終了。

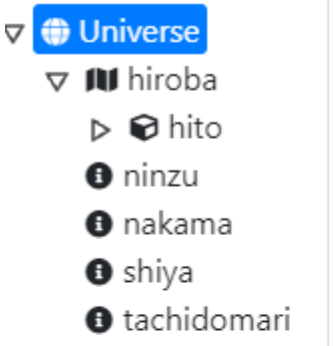
```
17 def univ_step_end(self):
18
19     # 全員が立ち止まったらシミュレーションを終了する
20     if Universe.tachidomari == Universe.ninzu:
21         exit_simulation()
```

❒8.5 終了時にartisoc Cloudに一仕事させる(1)

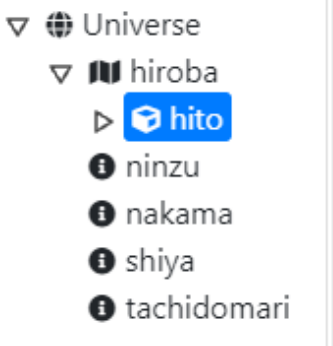
- 終了時に、何ステップで終了したかをコンソール画面に書き出してみる。
  - 立ち話モデル3Cを継承し、立ち話モデル3Dとして保存する。
  - univ_finishに下記を書込。
  - univ_finish：シミュレーション終了時のみに実行されるルール
  - print・・・：コンソール画面に “” で囲まれた文字および、 count_step()の数値を出力する。
    - ➡ +は、続けて表示するという意味
  - str：数値を文字列に変換する関数
    - ➡ count_step()でステップを示す整数が返ってくる。他の文字列と結合するために、str()で文字列に変換する。

```
23 def univ_finish(self):
24
25     # シミュレーション終了時にメッセージを出力
26     print("Simulation Completed after " + str(count_step()) + " Step")
27
28
```

❑8.5 終了時にartisoc Cloudに一仕事させる(2)




```python
def univ_init(self):

    for i in range(Universe.ninzu):
        create_agt(Universe.hiroba.hito)

    # Universe.hirobaにいる人をhitobito変数に入れる
    hitobito = make_agtset(space=Universe.hiroba)

    # ランダムに配置する
    random_put_agtset(hitobito)

def univ_step_begin(self):

    # Universe変数 tachidomariを毎ステップ初期化
    Universe.tachidomari = 0

def univ_step_end(self):

    # 全員が立ち止まったらシミュレーションを終了する
    if Universe.tachidomari == Universe.ninzu:
        exit_simulation()

def univ_finish(self):

    # シミュレーション終了時にメッセージを出力
    print("Simulation Completed after " + str(count_step()) + " Step")

```




```python
def agt_init(self):
    # self.x = rand() * 50
    # self.y = rand() * 50
    self.direction = rand() * 360

def agt_step(self):
    self.mawari = self.make_agtset_around_own(Universe.shiya, False, agttype=Universe.hiroba.hito)

    if count_agtset(self.mawari) >= Universe.nakama:

        # Universe変数 tachidomariを1増やす
        Universe.tachidomari = Universe.tachidomari + 1

    else:
        self.forward(1)

```

❑新しく学んだ事項

- 時系列グラフの設定
- 集計する技法
- コンソール画面の利用
- univ_step_begin, univ_step_end, univ_finishを利用する
- exit_simulation
- print

❒練習問題

- 以下の内容で立ち話モデル3Dを変更してみてください。
  - 1．歩き回っているhitoエージェント数を、新しい時系列グラフに出力する。
  - 2．立ち止まっている人が全体の90%を超えたら、シミュレーションを終了させる。

❒「立ち話モデル3D　8.4終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/xBWrwPm1R6O9WOiRp80vzQ
  - モデル名「立ち話モデル3D 8.4終了時点」で公開しています。
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

第9章：格子空間の構造を活用する

❐9.0 空間に「格子」の存在を想定しましょう

- artisoc Cloudの空間は、実は連続的です。
- 格子型空間を本来の格子のようにするには工夫が必要です。
- わざわざ「格子」上に空間を区切ってみます。
- 「格子」があるかのようにエージェントを振る舞わせましょう。
- 混み合う映画館での座り方をモデル化します。

❑9.1 ほんとうは「格子」になっていない(1)

- これまでのoozoraやhirobaという空間
  - 実は格子があることを気にしないで移動可能。

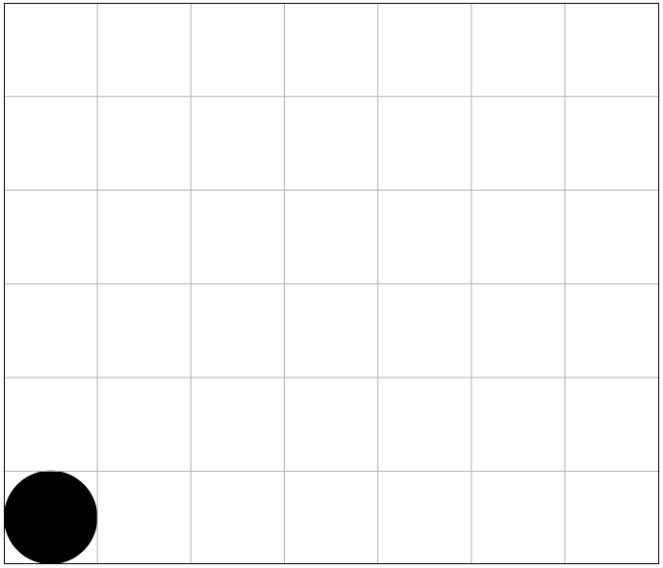
- 実験用の空間を新しく作る。
  - 新規モデルの作成　モデル名：celltest
  - 1)Universeの下に空間をつくる。
    - 空間名:akichi
    - 空間種別: 連続空間
    - 空間の大きさX:6、空間の大きさY:5
  - 2)akichiの下にhitoエージェントを作る。
  - 3)　出力設定＞マップ出力追加
    - マップ名: 空間
    - 空間: akichi
    - 罫線表示: チェス型に☑
    - マップ要素リストにhitoエージェントを追加。
  - 4) hitoエージェントの生成
    - univ_initでhitoエージェントを1体生成する。


```
def univ_init(self):
    a = create_agt(Universe.akichi.hito)
```

❒9.1 ほんとうは「格子」になっていない(2)

- hitoエージェントはX=0、Y=0にいるはず。
  - 左下端の格子の中にhitoエージェントが表示される。
- 空間の大きさは6×5のはず。
  - 格子の数は横に7個、縦に6個、表示される。

- エージェントをわかりやすく図示するため、右に0.5、上に0.5ずつずれている。
  - 右上端の(6，5)にいるエージェントは（6.5，5.5）に図示される。
    → 7×6としてのりしろを持たせることで表示可能。



❐9.2 空間は実数的だった(1)

- エージェントを動かしてみる（下記を書込）


```
def agt_init(self):
    self.x = rand() * 6
    self.y = rand() * 5

    self.direction = 30

def agt_step(self):
    self.forward(0.3)

```

❐9.2 空間は実数的だった(2)

- 観察のポイント
  - エージェントを表すアイコンと格子の罫線が重なることもある。
    - ➡ エージェントが存在している位置のX座標やY座標は整数ではない。
  - 右や上の端で一瞬消えて、同時に下や左の端に現れる。
    - ➡ 空間がループしている。
  - 動き方も細かい。
    - ➡ 小数単位で毎ステップ移動できる。

- → 格子上に罫線が入っていても、格子を無視してエージェントは動き回ることができる。

- → 空間を敢えて「格子型」としてモデル化する際には、これまでとは少し異なる設定を行う。

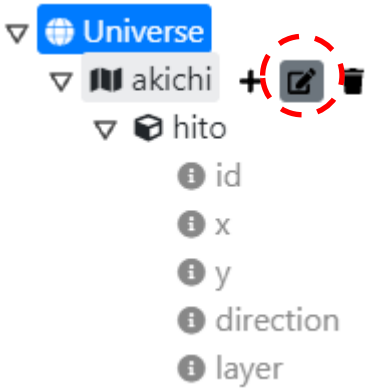
❒9.3「格子」があるかのようにエージェントを動かす(1)

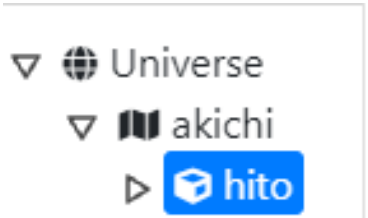
- 格子型の空間＝将棋盤のようなもの
  - セル（独房）型空間とも
    - ※囲碁型表示に☑をいれると、交点のみがエージェントの居場所になる。

- artisoc Cloudでの空間種別
  - 連続空間
    - エージェントの座標は実数
    - 設定した空間の大きさ ＋ 1
  - 四角格子空間
    - エージェントの座標は整数

❑9.3「格子」があるかのようにエージェントを動かす(2)

- 四角格子空間に変更してみる。
  - celltestを継承、celltest Bとして保存する。
- 変更内容
  - akichi 空間の空間種別を「四角格子空間」に変更する。
  - agtのルールを下のように変更する。






```
def agt_init(self):
    self.x = int(rand() * 6)
    self.y = int(rand() * 5)


def agt_step(self):
    self.forward_direction_sqgrid(3, 1)
```

❒9.3「格子」があるかのようにエージェントを動かす(3)

- self.x = int(rand() * 6)
- self.y = int(rand() * 5)
  - 座標の値を整数値化
  - int()：小数点以下を切り捨てて整数にする操作
    - int(rand() * 6)は、0から5までの整数値をとる。

- forward_direction_sqgrid(3,1)
  - エージェントを毎ステップ3の方向に1枠分動かす関数。

3		2		1
	3	2	1	
4	4	★	0	0
	5	6	7	
5		6		7

❑9.3「格子」があるかのようにエージェントを動かす(4)

- エージェントの動きを観察してみる。
  - 格子の中だけに位置し、飛び飛びに移動しているか？
  - 方向を他の数値（7以下の整数）に変えて、エージェントの動きがどのように変わるか？

- 空間種別を四角格子空間に設定すると、akichiののりしろ部分は表示されない。

❐「celltest 9.3終了時点」のモデル

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/6DVZuIgQRAWxId4T8Mb1VA
  - モデル名「celltest 9.3終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❑9.4 格子を利用するモデルの外枠を作る

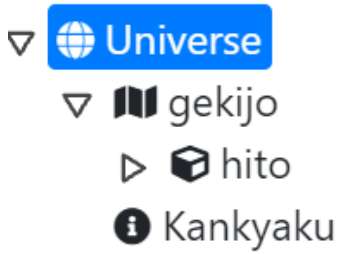
- 四角格子空間を使ったモデルを作成してみる。
  - 新規モデルを作成
  - モデル名：　映画館モデル

- ある劇場に10×20の椅子があり、そこにモデルの外から操作するn人が観劇に訪れるが、周囲に人がいて混んでくると空いている席に移動する。
  - 1) Universeの下にgekijo空間を作成する。
    - 空間種別：四角格子空間
    - 空間の大きさ：10×20
    - ループする：チェックを外す
  - 2) Universeの下に整数型変数kankyakuを作る。
  - 3）gekijoの下にhitoを作る（エージェント数は0）。
  - 4）マップ出力を設定する。
    - マップ名：劇場　空間：gekijo　罫線表示：チェス型を選択
    - マップ要素リストを追加　Universe.gekijo,hito　を出力
  - 5）kankyakuを10人から160人までの範囲で変えられるようにスライドバー コントロールパネルを設定（コントロール名: 観客数, 初期値: 20, 目盛間隔:10）。

この内容でモデルを作ってみましょう

❒9.5 格子を意識したルール表現を使う(1)

- univ_initにkankyakuの数だけhitoエージェントを生成し、ランダムにgekijoに座らせる。
  - 以下のルールを設定してください。

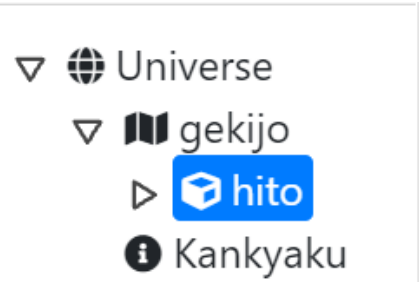



```
def univ_init(self):
    for i in range(Universe.Kankyaku):
        create_agt(Universe.gekijo.hito)

    hitobito = make_agtset(space=Universe.gekijo)
    random_put_agtset_sqgrid(hitobito, overlap=False)

def univ_step_begin(self):
    pass

def univ_step_end(self):
    pass

def univ_finish(self):
    pass
```

- 新しい関数
  - random_put_agtset_sqgrid(エージェント集合変数, overlap=False)
    - 格子に区切った空間にエージェントをランダムに配置する関数。
    - overlap=Falseにすると、1つの格子枠内に複数のエージェントを配置できない。
      - ☑ Trueにすると、重複も許す。

❒9.5 格子を意識したルール表現を使う(2)

- エージェントは周囲を見回して、視野1の範囲に4人以上他人がいると窮屈だと感じて、視野3の範囲で空席を探して、空席があればそこに移ります。




```
def agt_init(self):
    pass

def agt_step(self):

    # 距離1のセルにいるエージェントのエージェント集合を作る
    mawari = self.make_agtset_around_own_sqgrid(1, False)

    # 4人以上いれば、別なセルに移動する
    if count_agtset(mawari) >= 4:
        self.move_space_around_own_sqgrid(3)

```

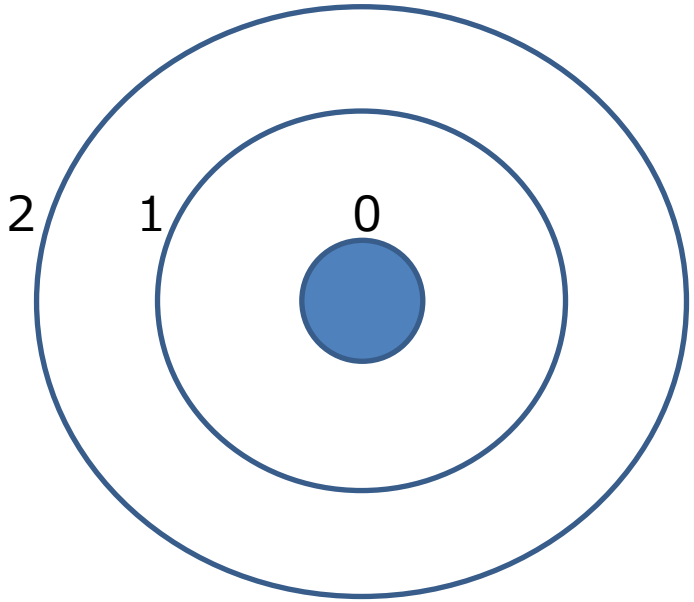
  - mawari = self.make_agtset_around_own_sqgrid(視野の広さ,自身を含むかどうか)
    - 自身の周りのセル（視野１）にいるエージェントを調べて、エージェント集合を作成する。
    - make_agtset_around_ownの四角格子空間版

- 保存・実行してみてください。

❒9.5 格子を意識したルール表現を使う(3)

- self.move_space_around_own_sqgrid(3)
  - 格子型視野「3」の範囲で、空いている格子があれば、そこに移るというルール。

2	2	2	2	2
2	1	1	1	2
2	1	0	1	2
2	1	1	1	2
2	2	2	2	2



❑新しく学んだ事項

- int()
- forward_direction_sqgrid()
- random_put_agtset_sqgrid()
- make_agtset_around_own_sqgrid()
- move_space_around_own_sqgrid()
- 格子（セル）型空間の方向の表し方
- マップ出力設定による罫線表示の情報（チェス盤と囲碁）
- 「視野」のとらえ方の違い

❒「映画館モデル」のモデル

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/u271BeGXQ7C5f7QCPUyH1Q
  - モデル名「映画館モデル」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❒練習問題

- 格子に区切られている空間を想定したエージェントの動かし方に慣れましょう。

- 1．この章で作ったcelltest Bモデルで、エージェントは毎ステップ、ランダムな方向に動く。
  - 定石のrand()*360は使えません。

- 2．まず、 celltest Bモデルで罫線を囲碁盤のようにマップ出力するように設定し直して下さい。出力設定の編集機能を使います。
- その上で、エージェントは毎ステップ、ランダムに上下左右に（つまり罫線に沿って） 1目しか動けないようにして下さい。

第10章：シェリングの「分居モデル」を作る

❐10.0 artisoc Cloudの有り難さをじっくりと味わいましょう

- artisoc Cloudの基本を一通り学びました。
- この章では新しく学ぶ技法はありません。
- 格子型に区切られた空間をエージェントが動き回り、やがて落ち着きます。
- アイデアをモデルに結びつけるところが肝腎です。
- モデルの改良も簡単です。

❒10.1 分居モデルとは(1)

- トマス・シェリング
  - 2005年ノーベル経済学賞受賞
  - おそらく初めてMASの手法を社会科学に取り入れた研究者
  - アメリカの都市で人々が民族集団ごとに別れて生活する現象に注目
    - 通説：差別意識や相互の排他的意識が地域社会の分居を促す。
    - シェリングはこの通説を疑問視
      → シミュレーションで研究
      → 個々人の意識がさほど排他的でなくても結果として地域社会の分居が生じてしまうことを鮮やかに示す。
  - 簡単なルールで社会現象の本質を捉え、個々人の意思とその相互作用で生まれる社会現象との直観に反する関連を見事に明らかにした古典。

❒10.1 分居モデルとは(2)

- シェリングの研究方法
  - 1960年代=コンピュータなし
    → 22枚の10セント硬貨（ダイムと呼ばれる銀色のコイン）と
    23枚の1セント硬貨（ペニーと呼ばれる銅色のコイン）、
    チェス盤（正確にはチェッカー盤）を使用。
  - 64軒の住宅が8×8に並んでいる地域社会に、
    銅色人種が23名、銀色人種が22名住んでいる状況をモデル化。

	#		#	0	#		0
#	#	#	0		0	#	0
	#	0			#	0	#
	0	#	0	#	0	#	0
0	0	0	#	0	0	0	
#		#	#	#			0
	#	0	#	0	#	0	
	0		0			#	

	#	#		0	#	#	
#	#	#	0	0	0	#	#
#	#	0	0			0	#
#	0		0		0	0	0
0	0	0	#	0	0	0	
	0	#	#	#	0	0	0
		#	#	#	#		
	0					#	

❐10.1 分居モデルとは(3)

- モデル内容
  - 個々の居住者は、近隣住民に占める同色居住者の比率が一定限度（閾値）以上ならば満足して、そこに住み続ける。
  - そうでなければ、空き地（空き家）に引っ越す。
    - ➡ 近隣とは、その家の東西南北4軒だけでなく、北東・南東・北西・南西の4軒も含む。
  - シミュレーションを簡単にするために、閾値は1/3に統一。
  - サイコロを振って45個のコインの場所をランダムに決定。
  - 個々のコインについて満足か不満かを計算し、不満なコインを周囲の空き地の中からサイコロを振って決めた場所へと移す作業を、不満なコインがなくなるまで繰り返す。
  - 最終状態の各コインにとっての近隣の同種コインの比率を調べる。
  - 個々の居住者の許容限度が低くても（1/3）、地域全体では分居が進むことが明らかになる。

❐10.2 コンピュータのなかの分居モデル(1)

- 住宅が8×8の状態に立ち並んでいる地域社会に、銅色人種の住民が23家族と銀色人種の住民が22家族住んでいる。両人種とも異人種に寛容で、同人種が近隣住民の内、1/3以上ならば満足して住み続ける。そうでない場合に限り、近所の空き地に引っ越しする。
  - 新しいモデルを作成。モデル名: schelling
  - Universeの下にchessboardという名前の四角格子空間（ループせず）を8×8でつくる。
  - chessboardの下にpennyエージェント、dimeエージェントをつくる（エージェントはuniv_init():で生成する。penny:23家族, dime:22家族）。
  - 出力設定の作業として、chessboardのマップ出力（罫線表示：チェス型）を設定。 表示色はpenny:青、dime:赤
  - 実行して、設定が正常かを確認する。

❐10.2 コンピュータのなかの分居モデル(2)

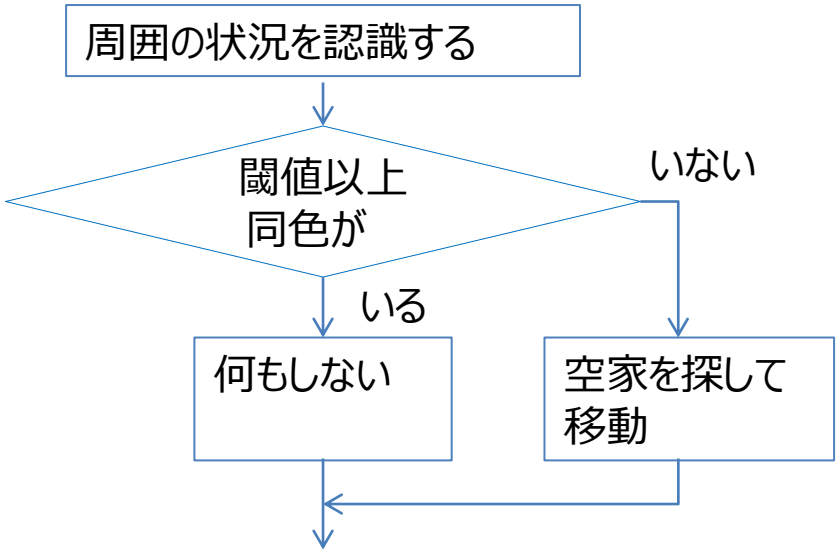
- コインをチェス盤にランダムに置くルールを書き込む

```
def univ_init(self):
    create_agt(Universe.chessboard.penny, num=23)
    create_agt(Universe.chessboard.dime, num=22)

    coin = make_agtset(space=Universe.chessboard)
    random_put_agtset_sqgrid(coin)

```

❒10.2 コンピュータのなかの分居モデル(3)

- 1セント硬貨（penny）のルールを書き込む。
  - 周囲の状況を認識し、満足かどうか判断させる。
    不満な場合は空き家を探して移動する（視野２）。

この内容でモデルを作ってみましょう



➡ 注意点：rateを計算させる部分でゼロで割ることをしないように工夫する。

❑10.2 コンピュータのなかの分居モデル(4)

```
def agt_step(self):

    # 自分のまわりのpennyエージェントをneighborに集める
    neighbor = self.make_agtset_around_own_sqgrid(1, False, agttype=Universe.chessboard.penny)
    # pennyエージェントの数を数えてnpに入れる
    np = count_agtset(neighbor)

    # 自分のまわりのdimeエージェントをneighborに集める
    neighbor = self.make_agtset_around_own_sqgrid(1, False, agttype=Universe.chessboard.dime)
    # dimeエージェントの数を数えてndに入れる
    nd = count_agtset(neighbor)

    rate = 0
    if 0 < np + nd: # npとndの合計が0超過の場合
        rate = np / (np+nd) #penny数の割合をrateに入れる

    # 閾値以上同色がいない場合、視野２で周辺のセルに引っ越す
    if rate < 1 / 3:
        self.move_space_around_own_sqgrid(2)
```

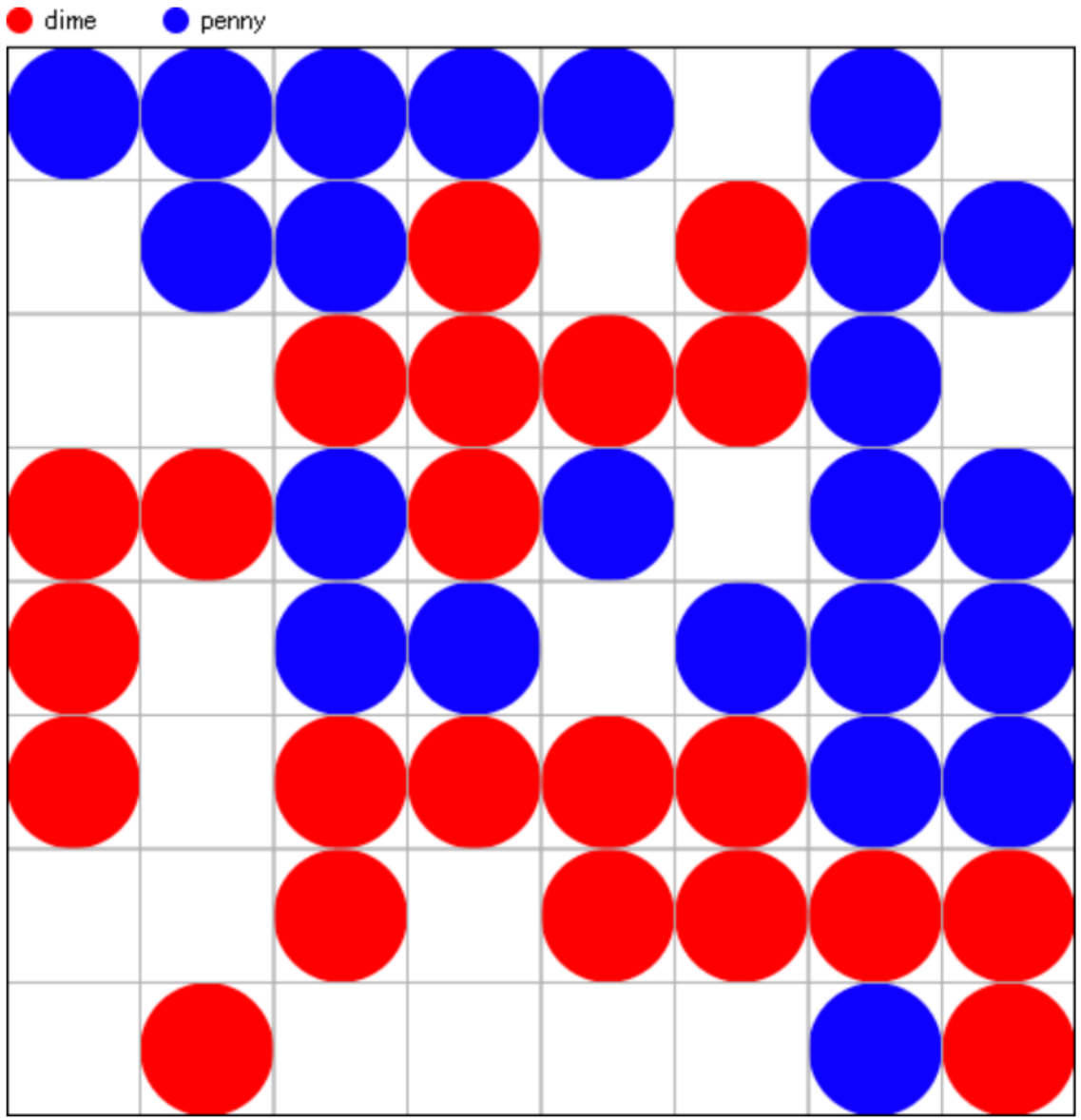
❐10.2 コンピュータのなかの分居モデル(5)

- 10セント硬貨（dime）についても同様のルールを書き込む。
  - pennyのルールと、下の1箇所だけ違います。

```
rate = nd / (np + nd)
```

- 上書き保存して実行する。

❒schelling



❑若干の補足

- シェリングのオリジナルモデルとの違い
  - シェリング：不満な場合、チェス盤全体のどこかの空き地に移動するルール。
    - コインが空き地を探し回る“過程”ではなく、分居してしまうという“結果”を分析することを狙いとしていたため、視野の大きさは重要な問題ではない。
  - ここでのモデル：視野2で空き地を探すルール
    - 広い視野を設定すると、すぐにどこかの空き家に引っ越しし、全てのコインが満足する。
    - あえて視野を狭めることで、過程を観察できるようにする。

❐「schelling 10.3終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/uMmPbUBDTOSwUpb3DV5gPg
  - モデル名「schelling 10.3終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❐10.3 シェリングの分析を再現する(1)

- 個々のエージェントにとっての分居度ではなく、全体の分居度を時系列グラフで表す。
  - “schelling”を継承し、“schelling B”という名前をつけて保存する。
  - 1)集計用の変数（実数型）averagelevelをUniverseに追加する。
    （averageとlevelはくっつける）
  - 2)毎ステップ初期化する。

```
 8 def univ_step_begin(self):
 9     Universe.averagelevel = 0
10
```

  - 3)各ステップ、個々のエージェントの分居度を集計する。
    (以下をどこに入れるか考えてみる)
    - Universe.averagelevel = Universe.averagelevel + rate
  - 4)時系列グラフの出力設定をする。
    - グラフ名「分居度」（Y軸設定の最大値を1に、目盛間隔を0.1にする）
    - Universe.averagelevel/45 を出力値とする（要するに平均）。

❐10.3 シェリングの分析を再現する(2)

- 満足しているエージェントの割合を時系列グラフに表示する。
  - “schelling B”を継承し、“schelling C”という名前をつけて保存する。
  - 1)集計用の変数satisfiedをUniverseに追加する。
  - 2)毎ステップ初期化。
  - 3)各ステップ、個々のエージェントについて、満足していればカウント。
    - ➡ どこにルールを書き込むか考えてみる。
      - ☑ Universe.satisfied = Universe.satisfied + 1
  - 4)時系列グラフの出力設定。
    - ➡ Universe.satisfied / 45 を出力値とする。
  - 5)schelling(C)として保存・実行する。

❐10.3 シェリングの分析を再現する(3)

- “schelling C”を継承し、“schelling D”という名前をつけて保存する。
- 全員が満足したら終了する。
- もし、不満な人が残っていても50ステップ経ったら終了する。

```
12  def univ_step_end(self):
13      # 全員が満足したらメッセージを出して終了
14      if Universe.satisfied == 45:
15          print("Every one satisfied after " + str(count_step()) + " steps")
16          exit_simulation()
17
18      # 全員満足していなくても、50ステップ経過後 終了
19      if 50 < count_step():
20          print("Not all satisfied until 50 steps")
21          exit_simulation()
```

❐「schelling D 10.4終了時点」のモデル

途中でついていけなくなった人・モデルがうまく動かない人

- 下記のモデルにアクセスしモデルを継承してください
  - https://artisoc-cloud.kke.co.jp/models/ZEgGmNwiSMaay8uvxdbMyg
  - モデル名「schelling D 10.4終了時点」で公開しています
  - 継承：他のユーザが作ったモデルをコピーして、自分のモデルを新しく作る

❒練習問題

- モデルを発展させてみましょう。
  - ① 空間を広くしてエージェントを増やす。
  - ② エージェント数をコントロールパネルで操作できるようにする。
  - ③ 視野をコントロールパネルで操作できるようにする。
  - ④ エージェントの種類を２から３にする。
  - ⑤ エージェント種毎に異なるルールを設定してみる。
  - ⑥ 分居モデルを発展させてオリジナルのモデルを作ってみる。
  - ⑦ ⑥のモデルのパラメータを変えて実験してみる。