Ejercicio 1

En este sistema multiagente se han definido tres agentes: GrassPatch, Sheep y Wolf.

Los tres agentes son de tipo reactivo. Frente a un conjunto de entradas (variables) reaccionan con acciones de forma aislada e individual para cada instancia sin usar ningún razonador o motor de inferencia. Vemos a continuación esas variables de entrada y las acciones a realizar por los agentes.

GrassPatch (implementa la clase Agent)

Las variables implementadas son

self: representa la propia instancia de GrassPatch
pos: indica la posición en el plano con un vector de dos dimensiones (x,y)
model: variable de la clase model que almacena las variables globales del modelo (ver al final del ejercicio variables globales del modelo)
fully_grown: Variable tipo boolean que indica si se sigue generando hierba (potencial comida para las ovejas) o se detiene el crecimiento.
countdown: contador de tipo (número entero). Si la variable fully_grown es *false* el contador decrece por un valor determinado por *self.model.grass_regrowth_time*

Las acciones que implementa son

Step(self): recibiéndose a si mismo como parámetro, el agente decrece su contador en función de *grass_regrowth_time* cuando la varible *fully_grown* es *false* y el contador es mayor que cero. En caso contrario no hay acción.

Sheep (implementa la clase Randomwalker,que implementa a su vez agent)

Tiene las siguientes variables

self: representa la propia instancia de Sheep
pos: indica la posición en el plano con un vector de dos dimensiones (x,y)
model: variable de la clase model que almacena las variables globales del modelo (ver al final del ejercicio)
moore: determina la posibilidad de moverse en diagonal o no
energy: cantidad de energía de Sheep. Cuando decrece hasta cero, la oveja muere y desaparece. Se incrementa comiendo hierba

Las acciones que implementa son

init(): crea y coloca una oveja (self) en el espacio bidimensional (Grid)
step(): realiza el conjunto de acciones en cada paso en función de las condicones del entorno.

- La oveja cambia de posición a una adyacente en el plano de forma aleatoria
- Si no hay hierba en la pos(x,y) decrece la propia energía
- Si hay hierba se la come, interactuando con el agente GrassPatch (fully_grown=false)
- Si la energía llega a cero muere. La variable local de step Living se pone en false
- Si no muere ejecuta un método aleatorio que compara con la variable model.sheep_reproduce. Si el resultado es menor crea una nueva oveja.

Wolf implementa la clase Randomwalker (que implementa a su vez agent) y tiene las siguientes variables

self: representa la propia instancia de Sheep
pos: indica la posición en el plano con un vector de dos dimensiones (x,y)
model: variable de la clase model que almacena las variables globales del modelo (ver al final del ejercicio)
moore: determina la posibilidad de moverse en diagonal o no
energy: cantidad de energía de Wolf. Cuando decrece hasta cero, el lobo muere y desaparece. Se incrementa matando ovejas

Las acciones que implementa son

init: crea y coloca una oveja (self) en el espacio bidimensional

step: realiza el conjunto de acciones en cada paso en función de las condiciones del entorno.

- El lobo cambia de posición a una adyacente en el plano de forma aleatoria
- Si no hay oveja en la pos(x,y) decrece la propia energía
- Si hay oveja se la come, interactuando con el agente Sheep que desaparece y se incrementa la energía de wolf en un valor definido en la variable global *wolf_gain_from_food*
- Si la energía del lobo llega a cero muere y desaparece
- Si no muere ejecuta un método aleatorio que compara con la variable model.wolf_reproduce. Si el resultado es menor crea una nueva instancia de Wolf.

Adicionalmente para el funcionamiento del modelo se definen las variables siguientes en la clase Model:

```
height = 20
width = 20
```

Medidas (x,y) del terreno modelado. Se mide en unidades (cuadrículas)

`initial_sheep = 100`
Cantidad inicial de objetos de la clase Sheep

`initial_wolves = 50`
Catidad inicial de objetos clase Wolf

`sheep_reproduce = 0.04`
Valor para el cálculo aleatorio que determina si ay reproducción o no de Sheep

`wolf_reproduce = 0.05`
Valor para el cálculo aleatorio que determina si ay reproducción o no de Wolf

`wolf_gain_from_food = 20`
Cantidad de enrgía ganada por wun lobo al matar una oveja

`grass = False`
Determina la presencia o no de hierba en un cuadrante

`grass_regrowth_time = 30`
índice para determianr la tasa de crecimiento de la hierba

`sheep_gain_from_food = 4`
Cantidad de energía ganada por una oveja al comer hierba

Ejercicio 2

A continuación, en rojo, las modificaciones efectuadas en la clase Agents para tener en cuenta la edad de los agentes en el modelo. Consideramos 1 step=1mes, 12 meses=1año

```
class Sheep(RandomWalker):
    '''
    A sheep that walks around, reproduces (asexually) and gets eaten.

    The init is the same as the RandomWalker.
    '''

    energy = None
    age=None

    def __init__(self, pos, model, moore, energy=None, age=0):
        super().__init__(pos, model, moore=moore)
        self.energy = energy
        self.age=0

    def step(self):
        '''
        A model step. Move, then eat grass and reproduce.
```

```
        '''
        self.random_move()
        living = True
        self.age +=1

        if self.model.grass:
            # Reduce energy
            self.energy -= 1

            # If there is grass available, eat it
            this_cell =
self.model.grid.get_cell_list_contents([self.pos])
            grass_patch = [obj for obj in this_cell
                           if isinstance(obj, GrassPatch)][0]
            if grass_patch.fully_grown:
                self.energy += self.model.sheep_gain_from_food
                grass_patch.fully_grown = False

            # Death
            if self.energy < 0:
                self.model.grid._remove_agent(self.pos, self)
                self.model.schedule.remove(self)
                living = False
            if self.age>=120:
                living = False

        if living and self.age>=12 and random.random() <
self.model.sheep_reproduce:
            # Create a new sheep:
            if self.model.grass:
                self.energy /= 2
            lamb = Sheep(self.pos, self.model, self.moore, self.energy)
            self.model.grid.place_agent(lamb, self.pos)
            self.model.schedule.add(lamb)


class Wolf(RandomWalker):
    '''
    A wolf that walks around, reproduces (asexually) and eats sheep.
    '''

    energy = None
    age=None

    def __init__(self, pos, model, moore, energy=None, age=0):
        super().__init__(pos, model, moore=moore)
        self.energy = energy
        self.age=0

    def step(self):
```

```
        self.random_move()
        self.energy -= 1
        self.age +=1

        # If there are sheep present, eat one
        x, y = self.pos
        this_cell = self.model.grid.get_cell_list_contents([self.pos])
        sheep = [obj for obj in this_cell if isinstance(obj, Sheep)]
        if len(sheep) > 0:
            sheep_to_eat = random.choice(sheep)
            self.energy += self.model.wolf_gain_from_food

            # Kill the sheep
            self.model.grid._remove_agent(self.pos, sheep_to_eat)
            self.model.schedule.remove(sheep_to_eat)

        # Death or reproduction
        if self.energy < 0 or self.age>=120:
            self.model.grid._remove_agent(self.pos, self)
            self.model.schedule.remove(self)
        else:
            if random.random() < self.model.wolf_reproduce and
self.age<=12:
                # Create a new wolf cub
                self.energy /= 2
                cub = Wolf(self.pos, self.model, self.moore, self.energy)
                self.model.grid.place_agent(cub, cub.pos)
                self.model.schedule.add(cub)
```

Ejercicio 3

Si Woolf y Sheep pudiesen comunicarse, lo más lógico, atendiendo a la demanda de realismo del enunciado sería que lo hiciesen entre agentes de la misma clase (wolf—>wolf y sheep—> sheep).

1ª acción: avisar de la presencia de lobos u ovejas (según sea el caso)

Siguiendo este razonamientosupondremos que las ovejas establecerán comunicación para avisar de la presencia de lobos adyacentes a su cuadrante balando y que los lobos advertirán de la presencia de ovejas a otros lobos aullando.

La información transmitida en leguaje FIPA-ACL seguiría una secuencia pregunta-respuesta.

```
#Pregunta-respuesta agentes clase Sheep
#pregunta
(query
      :sender sheep 1
      :receiver: {adjacent sheep 1, adjacen sheep2,…adjacent sheep (n<8)}
```

```
      :content
            (report (closeWolves(x,y)?))
      :language  Lisp
      :ontology wolf_sheep_predation
)

#respuesta
(inform
      :sender sheep 2
      :receiver sheep1
      :content
            (inform {(12, 8), (13,8),(14,7)})
      :in-reply-to: query
      :languge Lisp
      :ontology wolf_sheep_predation
)

#Pregunta-respuesta agentes clase Wolf
#pregunta
(query
      :sender wolf 1
      :receiver: {adjacent Wolf 1, adjacent Wolf 2,…adjacent Wolf n
      (n<8)}
      :content
            (report (close_sheep(x,y)?))
      :language  Lisp
      :ontology wolf_sheep_predation
)

#respuesta
(inform
      :sender wolf 2
      :receiver wolf 1
      :content
            (inform {(15, 3), (17,3)})
      :in-reply-to: query
      :languge Lisp
      :ontology wolf_sheep_predation
)
```

Para la comunicación deberá utilizarse el protocolo FIPA-Query. En el ejemplo de mensajes se ha supuesto que los mensajes no fallan en la transmisión, que se entienden por el receptor y que no son rechazados (ej: falta de privilegios,…).

La información contenida en el mensaje de respuesta es la un vector de posiciones ocupadas por lobos (u ovejas en el caso contrario)

El protocolo utilizado contempla otras tres posibles respuestas: *not understood, failiure y refuse*.

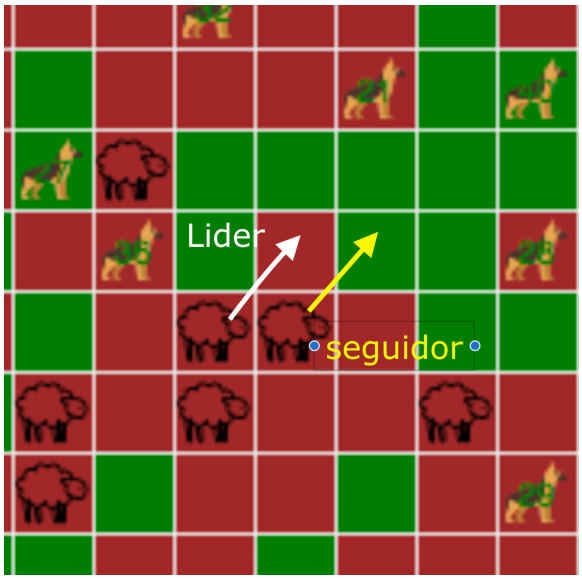
Para que tenga sentido la utilización de un sistema de avisos ente agentes los movimientos aleatorios de los agente por cada step deberán ser cambiados por movimientos dirigidos (al menos cuando se reciba respuesta afirmativa de otro agente a la interrogación sobre la presencia de agentes "contrarios")

Así, los lobos deberían aproximarse a los cuadrantes donde se detecte presencia de ovejas y las ovejas alejarse de los cuadrantes con presencia de lobos confirmada.

2ª acción: formar rebaño

La tendencia natural del ganado a formar rebaños puede modelarse utilizando una comunicación entre los agentes de clase *sheep*.

Mediante el protocolo *FIPA-request* cada oveja propondrá a las ovejas adyacentes ser el lider del rebaño, lo cual significará que quienes sigan un lider se moverán en el mismo sentido que este como se ve en el diagrama



Una oveja (sheep 2) que reciba un mensaje de de la oveja (sheep 1) proponiendose como lider lo aceptará si no es lider ella misma o es seguidora de otra oveja, en cuyo caso lo rechazará la propuesta y se propondrá como lider ella misma.

```
#Request formar rebaño. Sheep 1 se propone como lider
#pregunta
(request
      :sender sheep 1
      :receiver: {sheep 2}
      :content
```

```
            (flock(sheep 1)?)
      :language  Lisp
      :ontology wolf_sheep_predation
)

#respuesta. Sheep 2 ya sigue a otro agente
(refuse
      :sender sheep 2
      :receiver sheep1
      :content
            (refuse)
      :in-reply-to: request(sheep 1)
      :languge Lisp
      :ontology wolf_sheep_predation
)

#pregunta. Sheep 2 se propone como lider
(request
      :sender sheep 2
      :receiver: {sheep 1}
      :content
            (flock(sheep 2)?)
      :language  Lisp
      :ontology wolf_sheep_predation
)
#respuesta. Sheep 1 acepta a sheep 2 como lider
(agree
      :sender sheep 1
      :receiver sheep2
      :content
            (agree)
      :in-reply-to: request(sheep 2)
      :languge Lisp
      :ontology wolf_sheep_predation
)
```

De esta manera, cada agente tipo “oveja” que tenga otro agente de la misma clase adyacente o bien será lider o bien seguirá a uno de los agentes adyacentes.

Los datos enviados en el request son la solicitud del emisor para ser reconocido como lider por el receptor en forma de respuesta “agree” o “refuse”, y la información contenida en la respuesta es la aceptación (*agree*), rechazo *(refuse)* o mensaje no entendido *(not-understood)*