**INTELLIGENT AGENTS**

* **Contents**
* **Overview**
* **Agent**
* **Environment**
* **Simple Agent and Environment**
* **Agents in a 2-D Environment**
* **Wumpus Environment**

|  |  |
| --- | --- |
| **AGENT** | **ENVIRONMENT** |
| psource(Agent) | psource(Environment) |
| 1. **\_\_**init\_\_(self,program = none) |  |
| * **Alive** * **Bump** * **Holding** * **Performance** * **program** | * thing\_classes(self) * add\_thing(self,things, location=none) * run(self,steps) * is\_done(self) |
| 1. can\_grab(self, thing) | 1. percept(self , agent) |
|  | 1. execute\_action(self , agent , action) |
|  |  |
|  |  |

**Simple Agent and Environment**

Agent class to creating our first agent - a blind dog.

**#** In the blind dog class i.e agent class dog can either eat or drink so we use “ can\_grab(self, thing) “ method and then created object of dog of BlindDog class.

# We create a dog who can only feel what's in his location (since he's blind), and can eat or drink and check whether it is alive

--------------------------------------------------------------------------------------------------------------------

class BlindDog(Agent):

def eat(self, thing):

print("Dog: Ate food at {}.".format(self.location))

def drink(self, thing):

print("Dog: Drank water at {}.".format( self.location))

dog = BlindDog()

print(dog.alive)

* Now, we create a environment because our dog can perceive and act upon it.

**Environment**

The **Environment** class is an abstract class, so we will have to create our own subclass from it before we can use it. We create **park** environment

# Abstract class representing an Environment. 'Real' Environment classes inherit from this. We need to implement:

percept: Define the percept that an agent sees.

execute\_action: Define the effects of executing an action. Also update the agent.performance slot.

The environment keeps a list of .things and .agents (which is a subset of .things). Each agent has a .performance slot, initialized to 0.

Each thing has a .location slot, even though some environments may not

need this.

--------------------------------------------------------------------------------------------------------------------

class Food(Thing):

pass

class Water(Thing):

pass

class Park(Environment):

def percept(self, agent):

'''return a list of things that are in our agent's location'''

things = self.list\_things\_at(agent.location)

return things

def execute\_action(self, agent, action):

'''changes the state of the environment based on what the agent does.'''

if action == "move down":

print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action, agent.location))

agent.movedown()

elif action == "eat":

items = self.list\_things\_at(agent.location, tclass=Food)

if len(items) != 0:

if agent.eat(items[0]): #Have the dog eat the first item

print('{} ate {} at location: {}'

.format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))

self.delete\_thing(items[0]) #Delete it from the Park after.

elif action == "drink":

items = self.list\_things\_at(agent.location, tclass=Water)

if len(items) != 0:

if agent.drink(items[0]): #Have the dog drink the first item

print('{} drank {} at location: {}'

.format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))

self.delete\_thing(items[0]) #Delete it from the Park after.

def is\_done(self):

'''By default, we're done when we can't find a live agent,

but to prevent killing our cute dog, we will stop before itself - when there is no more food or water'''

no\_edibles = not any(isinstance(thing, Food) or isinstance(thing, Water) for thing in self.things)

dead\_agents = not any(agent.is\_alive() for agent in self.agents)

return dead\_agents or no\_edibles

--------------------------------------------------------------------------------------------------------------------

* We re-implement our BlindDog to be able to **move down and eat food or drink water** only if it is present.

class BlindDog(Agent):

location = 1

def movedown(self):

self.location += 1

def eat(self, thing):

'''returns True upon success or False otherwise'''

if isinstance(thing, Food):

return True

return False

def drink(self, thing):

''' returns True upon success or False otherwise'''

if isinstance(thing, Water):

return True

return False

--------------------------------------------------------------------------------------------------------------------

* **Program controls how the dog acts upon its environment**

|  |  |  |  |
| --- | --- | --- | --- |
| **Percept:** | Feel Food | Feel Water | Feel Nothing |
| **Action:** | eat | drink | move down |

#Return the percept that the agent sees at the point

def program(percepts):

'''Returns an action based on the dog's percepts'''

for p in percepts:

if isinstance(p, Food):

return 'eat'

elif isinstance(p, Water):

return 'drink'

return 'move down'

* **Run our simulation by creating a park with some food, water, and our dog.**

We notice that the dog moved from location 1 to 4, over 4 steps, and ate food at location 5 in the 5th step.

#Run the Environment for given number of time steps.

--------------------------------------------------------------------------------------------------------------------

park = Park()

dog = BlindDog(program)

dogfood = Food()

water = Water()

park.add\_thing(dog, 1)

park.add\_thing(dogfood, 5)

park.add\_thing(water, 7)

park.run(5)

-----------------------------------------------------------------------------------------------------------------

**Agents in a 2-D Environment**

We need a subclass of **GraphicEnvironment** instead of Environment. Parks implemented by subclassing **GraphicEnvironment** class adds these extra properties to it:

1. 4 quadrant of the X-Y plane.
2. colors are defined in typical [**RGB digital 8-bit format**](https://en.wikipedia.org/wiki/RGB_color_model#Numeric_representations),
3. Fences are added . **GraphicEnvironment** provides “ is\_inbounds”  function to check if our dog tries to leave the park.

We will make dog to make Turns and moves forward, instead of always moving down.

|  |  |  |  |
| --- | --- | --- | --- |
| **Percept:** | Feel Food | Feel Water | Feel Nothing |
| **Action:** | eat | drink | |  |  |  | | --- | --- | --- | | **Remember being at Edge :** | At Edge | Not at Edge | | **Action :** | Turn Left/Right ( 50% - 50% chance ) | Turn Left / Right / Move Forward( 25% - 25% - 50% chance ) | |

This class is for environments on a 2D plane, with locations labelled by (x, y) points, either discrete or continuous.

Agents perceive things within a radius. Each agent in the environment has a .location slot which should be a location such as (0, 1), and a .holding slot, which should be a list of things that are held.

\_

class Park2D(GraphicEnvironment):

def percept(self, agent):

' 'return a list of things that are in our agent's location'''

things = self.list\_things\_at(agent.location)

loc = copy.deepcopy(agent.location) # find out the target location

#Check if agent is about to bump into a wall

if agent.direction.direction == Direction.R:

loc[0] += 1

elif agent.direction.direction == Direction.L:

loc[0] -= 1

elif agent.direction.direction == Direction.D:

loc[1] += 1

elif agent.direction.direction == Direction.U:

loc[1] -= 1

if not self.is\_inbounds(loc):

things.append(Bump())

return things

def execute\_action(self, agent, action):

'''changes the state of the environment based on what the agent does.'''

if action == 'turnright':

print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action, agent.location))

agent.turn(Direction.R)

elif action == 'turnleft':

print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action, agent.location))

agent.turn(Direction.L)

elif action == 'moveforward':

print('{} decided to move {}wards at location: {}'.format(str(agent)[1:-1], agent.direction.direction, agent.location))

agent.moveforward()

elif action == "eat":

items = self.list\_things\_at(agent.location, tclass=Food)

if len(items) != 0:

if agent.eat(items[0]):

print('{} ate {} at location: {}'

.format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))

self.delete\_thing(items[0])

elif action == "drink":

items = self.list\_things\_at(agent.location, tclass=Water)

if len(items) != 0:

if agent.drink(items[0]):

print('{} drank {} at location: {}'

.format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))

self.delete\_thing(items[0])

def is\_done(self):

'''By default, we're done when we can't find a live agent,

but to prevent killing our cute dog, we will stop before itself - when there is no more food or water'''

no\_edibles = not any(isinstance(thing, Food) or isinstance(thing, Water) for thing in self.things)

dead\_agents = not any(agent.is\_alive() for agent in self.agents)

return dead\_agents or no\_edibles

**Wumpus Environment**

from ipythonblocks import BlockGrid

from agents import \*

color = {"Breeze": (225, 225, 225),

"Pit": (0,0,0),

"Gold": (253, 208, 23),

"Glitter": (253, 208, 23),

"Wumpus": (43, 27, 23),

"Stench": (128, 128, 128),

"Explorer": (0, 0, 255),

"Wall": (44, 53, 57)

}

def program(percepts):

'''Returns an action based on it's percepts'''

print(percepts)

return input()

w = WumpusEnvironment(program, 7, 7)

grid = BlockGrid(w.width, w.height, fill=(123, 234, 123))

def draw\_grid(world):

global grid

grid[:] = (123, 234, 123)

for x in range(0, len(world)):

for y in range(0, len(world[x])):

if len(world[x][y]):

grid[y, x] = color[world[x][y][-1].\_\_class\_\_.\_\_name\_\_]

def step():

global grid, w

draw\_grid(w.get\_world())

grid.show()

w.step()

---------------------------------------------------------------------------------------------------------------------