Intelligent Agents

This notebook serves as supporting material for topics covered in **Chapter 2 - Intelligent Agents** from the book *Artificial Intelligence: A Modern Approach*. This notebook uses implementations from agents.py module. Let's start by importing everything from agents module.

In [1]:
```
from agents import *
from notebook import psource
```

CONTENTS

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

OVERVIEW

An agent, as defined in 2.1, is anything that can perceive its **environment** through sensors, and act upon that environment through actuators based on its **agent program**. This can be a dog, a robot, or even you. As long as you can perceive the environment and act on it, you are an agent. This notebook will explain how to implement a simple agent, create an environment, and implement a program that helps the agent act on the environment based on its percepts.

AGENT

Let us now see how we define an agent. Run the next cell to see how `Agent` is defined in agents module.

In [2]:
```
psource(Agent)
```

```
class Agent(Thing):
    """An Agent is a subclass of Thing with one required instance attribute
    (aka slot), .program, which should hold a function that takes one argument,
    the percept, and returns an action. (What counts as a percept or action
    will depend on the specific environment in which the agent exists.)
    Note that 'program' is a slot, not a method. If it were a method, then the
    program could 'cheat' and look at aspects of the agent. It's not supposed
    to do that: the program can only look at the percepts. An agent program
    that needs a model of the world (and of the agent itself) will have to
    build and maintain its own model. There is an optional slot, .performance,
    which is a number giving the performance measure of the agent in its
    environment."""

    def __init__(self, program=None):
        self.alive = True
        self.bump = False
        self.holding = []
        self.performance = 0
        if program is None or not isinstance(program, collections.abc.Callable):
            print("Can't find a valid program for {}, falling back to default.".format(self.__class_
_.__name__))

            def program(percept):
                return eval(input('Percept={}; action? '.format(percept)))

        self.program = program

    def can_grab(self, thing):
        """Return True if this agent can grab this thing.
        Override for appropriate subclasses of Agent and Thing."""
        return False
```

The `Agent` has two methods.

- `__init__(self, program=None)` : The constructor defines various attributes of the Agent. These include
    - `alive` : which keeps track of whether the agent is alive or not
    - `bump` : which tracks if the agent collides with an edge of the environment (for eg, a wall in a park)
    - `holding` : which is a list containing the `Things` an agent is holding,

    - `performance` : which evaluates the performance metrics of the agent
    - `program` : which is the agent program and maps an agent's percepts to actions in the environment. If no implementation is provided, it defaults to asking the user to provide actions for each percept.
- `can_grab(self, thing)` : Is used when an environment contains things that an agent can grab and carry. By default, an agent can carry nothing.

ENVIRONMENT

Now, let us see how environments are defined. Running the next cell will display an implementation of the abstract `Environment` class.

In [3]:
```
psource(Environment)
```

```
class Environment:
    """Abstract class representing an Environment. 'Real' Environment classes
    inherit from this. Your Environment will typically 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."""

    def __init__(self):
        self.things = []
        self.agents = []

    def thing_classes(self):
        return []  # List of classes that can go into environment

    def percept(self, agent):
        """Return the percept that the agent sees at this point. (Implement this.)"""
        raise NotImplementedError

    def execute_action(self, agent, action):
        """Change the world to reflect this action. (Implement this.)"""
        raise NotImplementedError

    def default_location(self, thing):
        """Default location to place a new thing with unspecified location."""
        return None

    def exogenous_change(self):
        """If there is spontaneous change in the world, override this."""
        pass

    def is_done(self):
        """By default, we're done when we can't find a live agent."""
        return not any(agent.is_alive() for agent in self.agents)

    def step(self):
        """Run the environment for one time step. If the
        actions and exogenous changes are independent, this method will
        do. If there are interactions between them, you'll need to
        override this method."""
        if not self.is_done():
            actions = []
            for agent in self.agents:
                if agent.alive:
                    actions.append(agent.program(self.percept(agent)))
                else:
                    actions.append("")
            for (agent, action) in zip(self.agents, actions):
                self.execute_action(agent, action)
            self.exogenous_change()

    def run(self, steps=1000):
        """Run the Environment for given number of time steps."""
        for step in range(steps):
            if self.is_done():
                return
            self.step()

    def list_things_at(self, location, tclass=Thing):
        """Return all things exactly at a given location."""
        if isinstance(location, numbers.Number):
            return [thing for thing in self.things
                    if thing.location == location and isinstance(thing, tclass)]
        return [thing for thing in self.things
                if all(x == y for x, y in zip(thing.location, location)) and isinstance(thing, tclass)]

    def some_things_at(self, location, tclass=Thing):
        """Return true if at least one of the things at location
        is an instance of class tclass (or a subclass)."""
        return self.list_things_at(location, tclass) != []

    def add_thing(self, thing, location=None):
        """Add a thing to the environment, setting its location. For
        convenience, if thing is an agent program we make a new agent
        for it. (Shouldn't need to override this.)"""
        if not isinstance(thing, Thing):
            thing = Agent(thing)
        if thing in self.things:
            print("Can't add the same thing twice")
        else:
```

```
        thing.location = location if location is not None else self.default_location(thing)
        self.things.append(thing)
        if isinstance(thing, Agent):
            thing.performance = 0
            self.agents.append(thing)

def delete_thing(self, thing):
    """Remove a thing from the environment."""
    try:
        self.things.remove(thing)
    except ValueError as e:
        print(e)
        print("  in Environment delete_thing")
        print("  Thing to be removed: {} at {}".format(thing, thing.location))
        print("  from list: {}".format([(thing, thing.location) for thing in self.things]))
    if thing in self.agents:
        self.agents.remove(thing)
```

`Environment` class has lot of methods! But most of them are incredibly simple, so let's see the ones we'll be using in this notebook.

- `thing_classes(self)` : Returns a static array of `Thing` sub-classes that determine what things are allowed in the environment and what aren't
- `add_thing(self, thing, location=None)` : Adds a thing to the environment at location
- `run(self, steps)` : Runs an environment with the agent in it for a given number of steps.
- `is_done(self)` : Returns true if the objective of the agent and the environment has been completed

The next two functions must be implemented by each subclasses of `Environment` for the agent to recieve percepts and execute actions

- `percept(self, agent)` : Given an agent, this method returns a list of percepts that the agent sees at the current time
- `execute_action(self, agent, action)` : The environment reacts to an action performed by a given agent. The changes may result in agent experiencing new percepts or other elements reacting to agent input.

SIMPLE AGENT AND ENVIRONMENT

Let's begin by using the `Agent` class to creating our first agent - a blind dog.

In [4]:
```
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()
```

```
Can't find a valid program for BlindDog, falling back to default.
```

What we have just done is create a dog who can only feel what's in his location (since he's blind), and can eat or drink. Let's see if he's alive...

In [5]:
```
print(dog.alive)
```

```
True
```

This is our dog. How cool is he? Well, he's hungry and needs to go search for food. For him to do this, we need to give him a program. But before that, let's create a park for our dog to play in.

ENVIRONMENT - Park

A park is an example of an environment because our dog can perceive and act upon it. The **Environment** class is an abstract class, so we will have to create our own subclass from it before we can use it.

In [6]:
```
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]
            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], age
                    self.delete_thing(items[0]) #Delete it from the Park aft
        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], age
                    self.delete_thing(items[0]) #Delete it from the Park aft

    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 - wh
        no_edibles = not any(isinstance(thing, Food) or isinstance(thing, Wa
        dead_agents = not any(agent.is_alive() for agent in self.agents)
        return dead_agents or no_edibles
```

PROGRAM - BlindDog

Now that we have a **Park** Class, we re-implement our **BlindDog** to be able to move down and eat food or drink water only if it is present.

In [7]:
```
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
```

Now its time to implement a **program** module for our dog. A program controls how the dog acts upon its environment. Our program will be very simple, and is shown in the table below.

Percept:	Feel Food	Feel Water	Feel Nothing
Action:	eat	drink	move down

In [8]:
```
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'
```

Let's now run our simulation by creating a park with some food, water, and our dog.

In [9]:
```
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)
```

```
BlindDog decided to move down at location: 1
BlindDog decided to move down at location: 2
BlindDog decided to move down at location: 3
BlindDog decided to move down at location: 4
BlindDog ate Food at location: 5
```

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

Let's continue this simulation for 5 more steps.

In [10]:
```
park.run(5)
```

```
BlindDog decided to move down at location: 5
BlindDog decided to move down at location: 6
BlindDog drank Water at location: 7
```

Perfect! Note how the simulation stopped after the dog drank the water - exhausting all the food and water ends our simulation, as we had defined before. Let's add some more water and see if our dog can reach it.

In [11]:
```
park.add_thing(water, 15)
park.run(10)
```

```
BlindDog decided to move down at location: 7
BlindDog decided to move down at location: 8
BlindDog decided to move down at location: 9
BlindDog decided to move down at location: 10
BlindDog decided to move down at location: 11
BlindDog decided to move down at location: 12
BlindDog decided to move down at location: 13
BlindDog decided to move down at location: 14
BlindDog drank Water at location: 15
```

Above, we learnt to implement an agent, its program, and an environment on which it acts. However, this was a very simple case. Let's try to add complexity to it by creating a 2-Dimensional environment!

AGENTS IN A 2D ENVIRONMENT

For us to not read so many logs of what our dog did, we add a bit of graphics while making our Park 2D. To do so, we will need to make it a subclass of **GraphicEnvironment** instead of Environment. Parks implemented by subclassing **GraphicEnvironment** class adds these extra properties to it:

- Our park is indexed in the 4th quadrant of the X-Y plane.
- Every time we create a park subclassing **GraphicEnvironment**, we need to define the colors of all the things we plan to put into the park. The colors are defined in typical **RGB digital 8-bit format**, common across the web.
- Fences are added automatically to all parks so that our dog does not go outside the park's boundary - it just isn't safe for blind dogs to be outside the park by themselves! **GraphicEnvironment** provides `is_inbounds` function to check if our dog tries to leave the park.

First let us try to upgrade our 1-dimensional `Park` environment by just replacing its superclass by `GraphicEnvironment`.

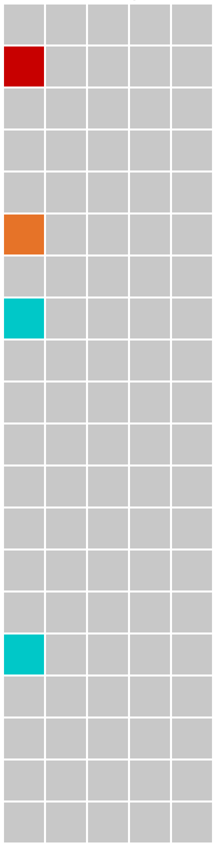
In [12]:
```
class Park2D(GraphicEnvironment):
    def percept(self, agent):
        '''return a list of things that are in our agent's location'''
        things = self.list_things_at(agent.location)
        return things

    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]
            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], age
                    self.delete_thing(items[0]) #Delete it from the Park aft
        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], age
                    self.delete_thing(items[0]) #Delete it from the Park aft

    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 - wh
        no_edibles = not any(isinstance(thing, Food) or isinstance(thing, Wa
        dead_agents = not any(agent.is_alive() for agent in self.agents)
        return dead_agents or no_edibles

class BlindDog(Agent):
    location = [0,1] # change location to a 2d value
    direction = Direction("down") # variable to store the direction our dog

    def movedown(self):
        self.location[1] += 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
```

Now let's test this new park with our same dog, food and water. We color our dog with a nice red and mark food and water with orange and blue respectively.

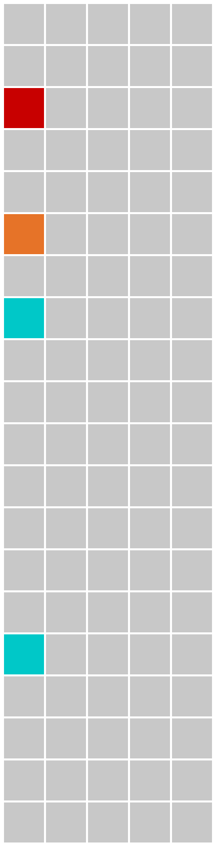
In [13]:
```
park = Park2D(5,20, color={'BlindDog': (200,0,0), 'Water': (0, 200, 200), 'F
dog = BlindDog(program)
dogfood = Food()
water = Water()
park.add_thing(dog, [0,1])
park.add_thing(dogfood, [0,5])
park.add_thing(water, [0,7])
morewater = Water()
```

```
park.add_thing(morewater, [0,15])
print("BlindDog starts at (1,1) facing downwards, lets see if he can find ar
park.run(20)
```

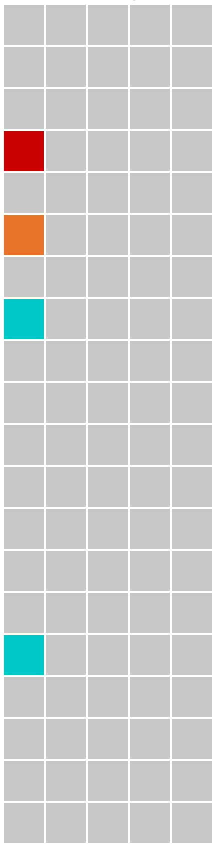
```
BlindDog starts at (1,1) facing downwards, lets see if he can find any food!
```



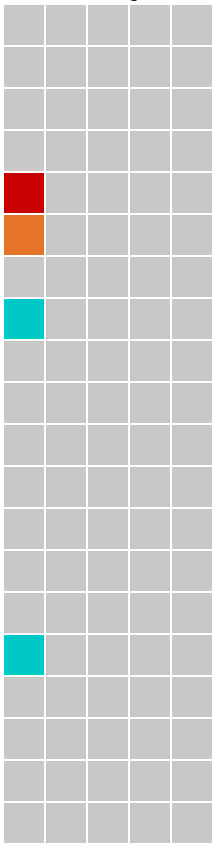
```
BlindDog decided to move down at location: [0, 1]
```



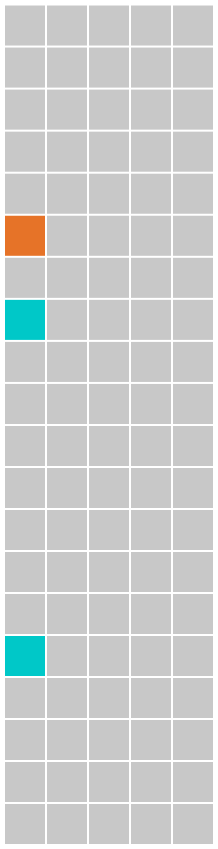
```
BlindDog decided to move down at location: [0, 2]
```



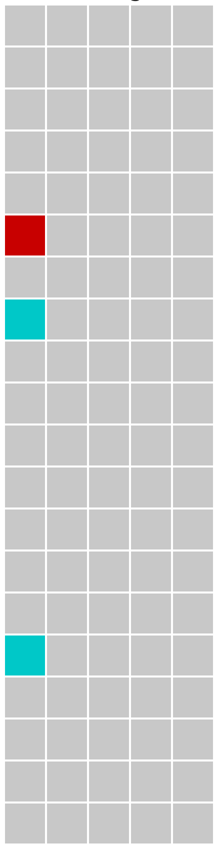
```
BlindDog decided to move down at location: [0, 3]
```



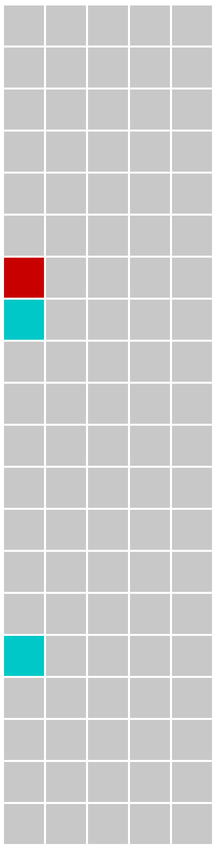
```
BlindDog decided to move down at location: [0, 4]
```



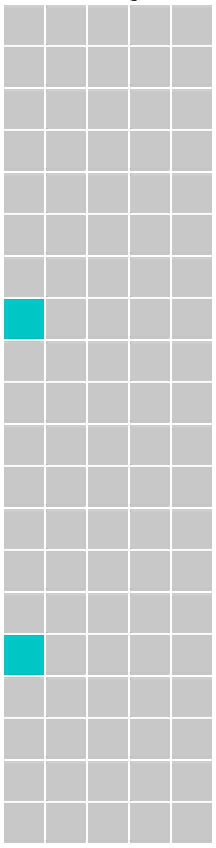
```
BlindDog ate Food at location: [0, 5]
```



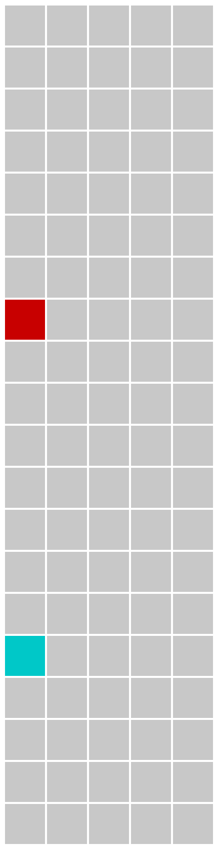
```
BlindDog decided to move down at location: [0, 5]
```



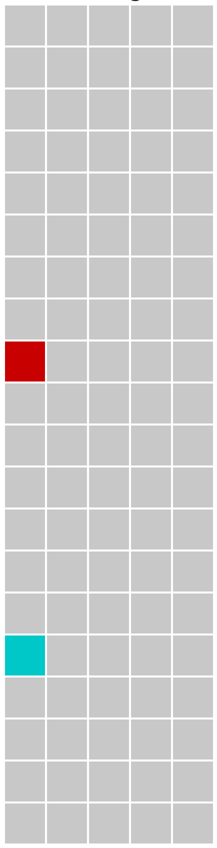
```
BlindDog decided to move down at location: [0, 6]
```



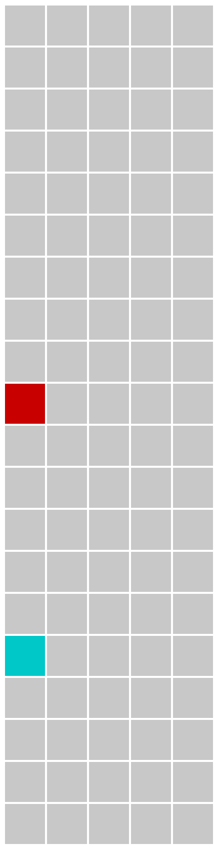
```
BlindDog drank Water at location: [0, 7]
```



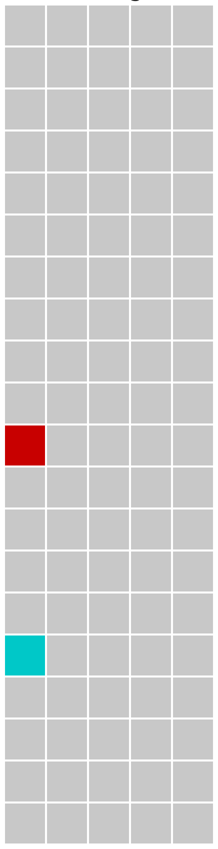
```
BlindDog decided to move down at location: [0, 7]
```



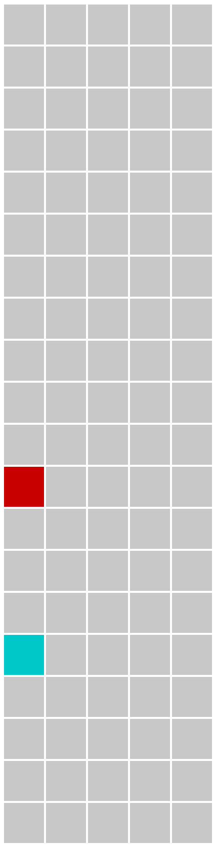
```
BlindDog decided to move down at location: [0, 8]
```



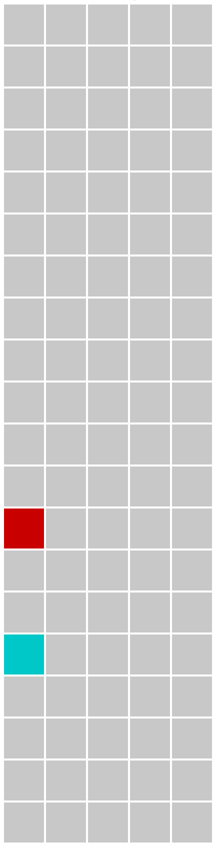
```
BlindDog decided to move down at location: [0, 9]
```



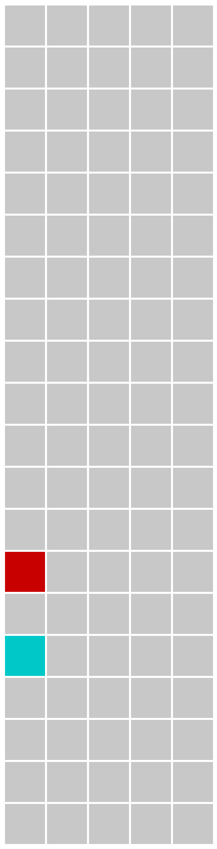
```
BlindDog decided to move down at location: [0, 10]
```



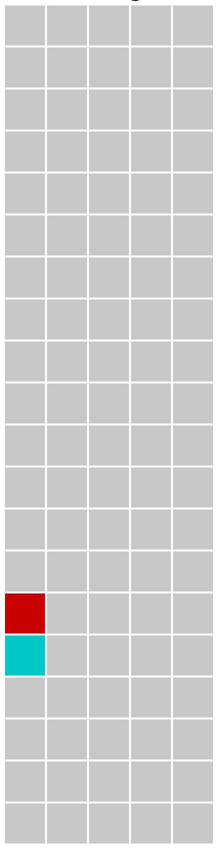
```
BlindDog decided to move down at location: [0, 11]
```



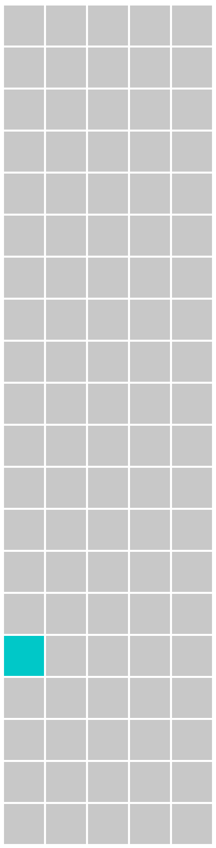
```
BlindDog decided to move down at location: [0, 12]
```



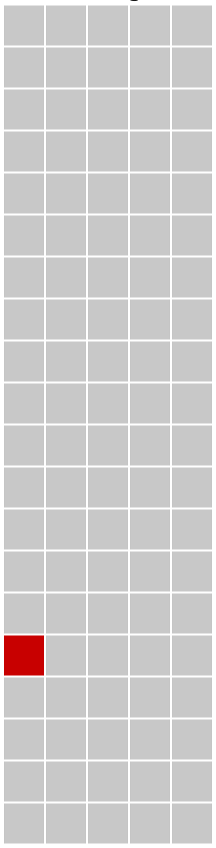
```
BlindDog decided to move down at location: [0, 13]
```

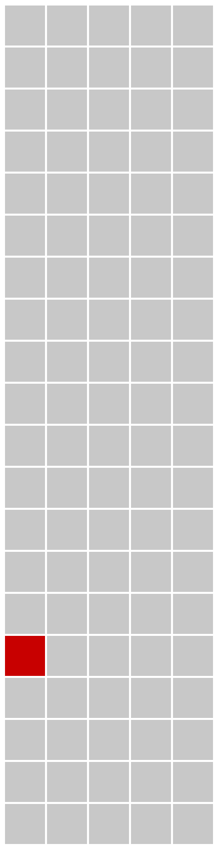


```
BlindDog decided to move down at location: [0, 14]
```



```
BlindDog drank Water at location: [0, 15]
```





Adding some graphics was a good idea! We immediately see that the code works, but our blind dog doesn't make any use of the 2 dimensional space available to him. Let's make our dog more energetic so that he turns and moves forward, instead of always moving down. In doing so, we'll also need to make some changes to our environment to be able to handle this extra motion.

PROGRAM - EnergeticBlindDog

Let's make our dog turn or move forwards at random - except when he's at the edge of our park - in which case we make him change his direction explicitly by turning to avoid trying to leave the park. However, our dog is blind so he wouldn't know which way to turn - he'd just have to try arbitrarily.

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

In [14]:
```python
from random import choice

class EnergeticBlindDog(Agent):
    location = [0,1]
    direction = Direction("down")

    def moveforward(self, success=True):
        '''moveforward possible only if success (i.e. valid destination loca
        if not success:
            return
        if self.direction.direction == Direction.R:
            self.location[0] += 1
        elif self.direction.direction == Direction.L:
            self.location[0] -= 1
        elif self.direction.direction == Direction.D:
            self.location[1] += 1
        elif self.direction.direction == Direction.U:
            self.location[1] -= 1

    def turn(self, d):
        self.direction = self.direction + d

    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

def program(percepts):
    '''Returns an action based on it's percepts'''

    for p in percepts: # first eat or drink - you're a dog!
        if isinstance(p, Food):
            return 'eat'
        elif isinstance(p, Water):
            return 'drink'
        if isinstance(p,Bump): # then check if you are at an edge and have t
            turn = False
            choice = random.choice((1,2));
        else:
            choice = random.choice((1,2,3,4)) # 1-right, 2-left, others-forw
    if choice == 1:
        return 'turnright'
    elif choice == 2:
        return 'turnleft'
    else:
        return 'moveforward'
```

ENVIRONMENT - Park2D

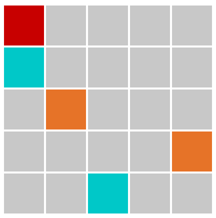
We also need to modify our park accordingly, in order to be able to handle all the new actions our dog wishes to execute. Additionally, we'll need to prevent our dog from moving to locations beyond our park boundary - it just isn't safe for blind dogs to be outside the park by themselves.

In [15]:
```
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]
            agent.turn(Direction.R)
        elif action == 'turnleft':
            print('{} decided to {} at location: {}'.format(str(agent)[1:-1]
            agent.turn(Direction.L)
        elif action == 'moveforward':
            print('{} decided to move {}wards at location: {}'.format(str(ag
            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], age
                    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], age
                    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 - wh
        no_edibles = not any(isinstance(thing, Food) or isinstance(thing, Wa
        dead_agents = not any(agent.is_alive() for agent in self.agents)
        return dead_agents or no_edibles
```

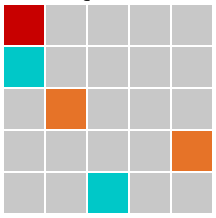
Now that our park is ready for the 2D motion of our energetic dog, lets test it!

In [16]:
```
park = Park2D(5,5, color={'EnergeticBlindDog': (200,0,0), 'Water': (0, 200,
dog = EnergeticBlindDog(program)
dogfood = Food()
water = Water()
park.add_thing(dog, [0,0])
park.add_thing(dogfood, [1,2])
park.add_thing(water, [0,1])
morewater = Water()
morefood = Food()
park.add_thing(morewater, [2,4])
park.add_thing(morefood, [4,3])
print("dog started at [0,0], facing down. Let's see if he found any food or
park.run(20)
```

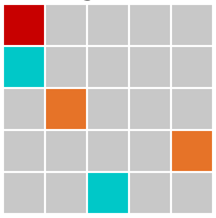
```
dog started at [0,0], facing down. Let's see if he found any food or water!
```



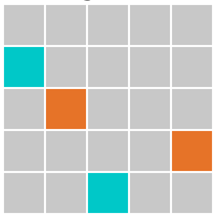
```
EnergeticBlindDog decided to turnright at location: [0, 0]
```



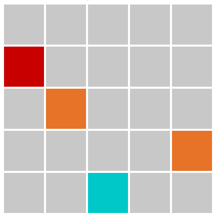
```
EnergeticBlindDog decided to turnleft at location: [0, 0]
```



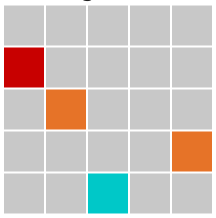
```
EnergeticBlindDog decided to move downwards at location: [0, 0]
```



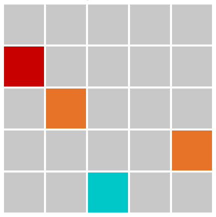
```
EnergeticBlindDog drank Water at location: [0, 1]
```



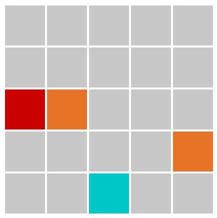
```
EnergeticBlindDog decided to turnright at location: [0, 1]
```



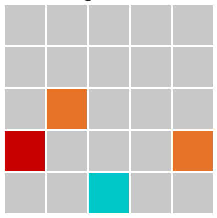
```
EnergeticBlindDog decided to turnleft at location: [0, 1]
```



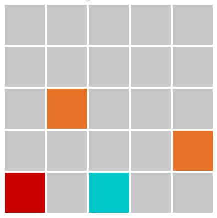
```
EnergeticBlindDog decided to move downwards at location: [0, 1]
```



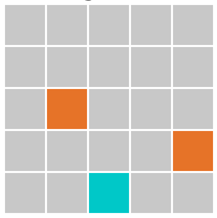
```
EnergeticBlindDog decided to move downwards at location: [0, 2]
```



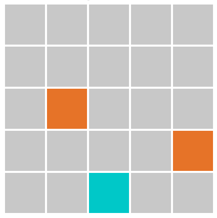
```
EnergeticBlindDog decided to move downwards at location: [0, 3]
```



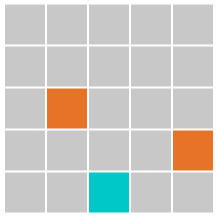
```
EnergeticBlindDog decided to move downwards at location: [0, 4]
```



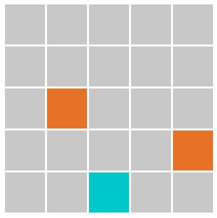
```
EnergeticBlindDog decided to turnleft at location: [0, 5]
```



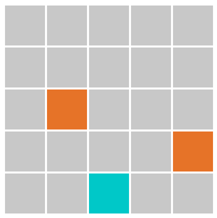
```
EnergeticBlindDog decided to turnright at location: [0, 5]
```



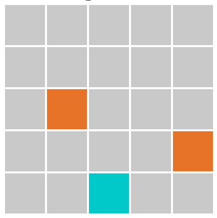
```
EnergeticBlindDog decided to turnleft at location: [0, 5]
```



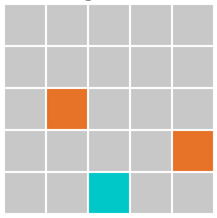
```
EnergeticBlindDog decided to move rightwards at location: [0, 5]
```



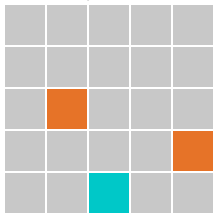
```
EnergeticBlindDog decided to turnleft at location: [1, 5]
```



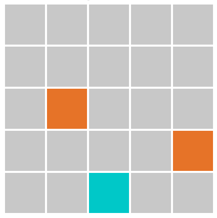
```
EnergeticBlindDog decided to turnleft at location: [1, 5]
```



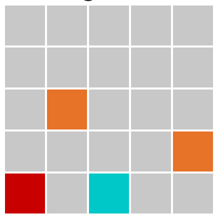
```
EnergeticBlindDog decided to move leftwards at location: [1, 5]
```



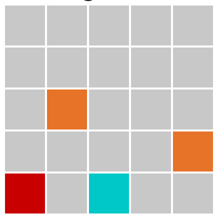
```
EnergeticBlindDog decided to turnright at location: [0, 5]
```



```
EnergeticBlindDog decided to move upwards at location: [0, 5]
```



```
EnergeticBlindDog decided to turnright at location: [0, 4]
```

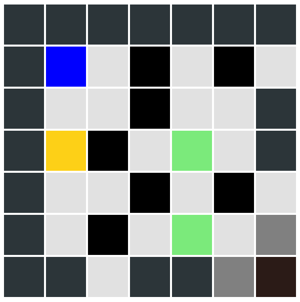


Wumpus Environment

```
In [27]: 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():
             print('Type any action into the bottom input box, such as "Left", and pr
             global grid, w
             draw_grid(w.get_world())
             grid.show()
             w.step()
```

```
In [25]: step()
```

```
Type any action into the bottom input box, such as "Left", and press "Enter"
```



```
[[<Bump>], [<Breeze>, <Breeze>], [<Bump>], [<Breeze>], [<Breeze>, None]]
```

```
In [26]: from collections import deque

         class Node:
             def __init__(self, value):
                 self.value = value
```

```
        self.left = None
        self.right = None

def bfs_traversal(root):
    if not root:
        return

    # Init the queue
    queue = deque([root])

    while queue:
        node = queue.popleft()

        # popped node's value
        print(node.value, end=" ")

        # add children to the queue (Left then Right)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

        print([n.value for n in queue])

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("BFS Traversal:")
bfs_traversal(root)
```

```
BFS Traversal:
1 [2, 3]
2 [3, 4, 5]
3 [4, 5]
4 [5]
5 []
```

```
In [ ]:
```