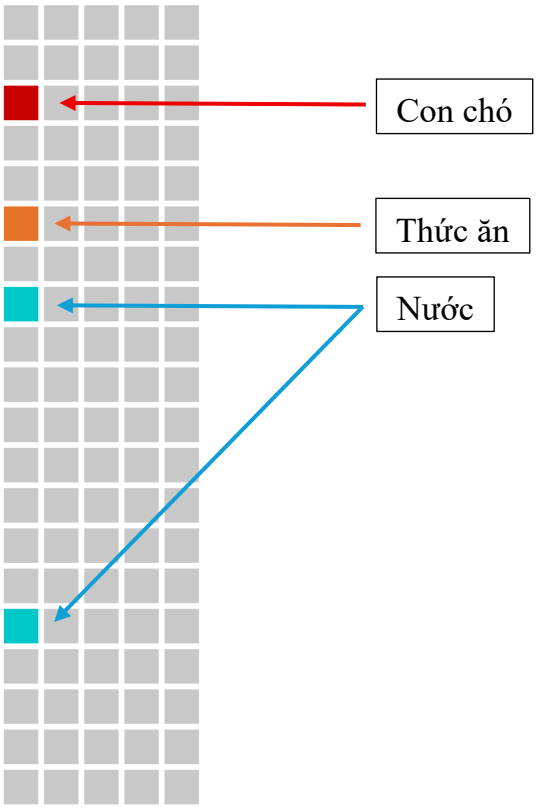
TRÍ TUỆ NHÂN TẠO

LAB 02 – AGENTS 2D

Trong bài thực hành này, chúng ta muốn xây dựng một tác nhân (Agent) là một con Chó, tuy nhiên con Chó này bị mù.




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

Chúng ta bổ sung thêm một vài đối tượng như vật cản (Obstacle), tường (Wall):

```
class Obstacle(Thing):
    """Something that can cause a bump, preventing an agent from
    moving into the same square it's in."""
    pass


class Wall(Obstacle):
    pass
```

Lớp **Direction** – để các Agents có thể di chuyển trong mặt phẳng 2D.

```
class Direction:
    R = "right"
    L = "left"
    U = "up"
    D = "down"

    def __init__(self, direction):
        self.direction = direction

    def __add__(self, heading):
        """
        >>> d = Direction('right')
        >>> l1 = d.__add__(Direction.L)
        >>> l2 = d.__add__(Direction.R)
        >>> l1.direction
        'up'
        >>> l2.direction
        'down'
        >>> d = Direction('down')
        >>> l1 = d.__add__('right')
        >>> l2 = d.__add__('left')
        >>> l1.direction == Direction.L
        True
        >>> l2.direction == Direction.R
        True
        """
        if self.direction == self.R:
            return {
                self.R: Direction(self.D),
                self.L: Direction(self.U),
            }.get(heading, None)
        elif self.direction == self.L:
            return {
                self.R: Direction(self.U),
                self.L: Direction(self.D),
            }.get(heading, None)
        elif self.direction == self.U:
            return {
                self.R: Direction(self.R),
                self.L: Direction(self.L),
            }.get(heading, None)
        elif self.direction == self.D:
            return {
                self.R: Direction(self.L),
                self.L: Direction(self.R),
            }.get(heading, None)

    def move_forward(self, from_location):
        """
        >>> d = Direction('up')
        >>> l1 = d.move_forward((0, 0))
```

```
        >>> l1
        (0, -1)
        >>> d = Direction(Direction.R)
        >>> l1 = d.move_forward((0, 0))
        >>> l1
        (1, 0)
        """
        # get the iterable class to return
        iclass = from_location.__class__
        x, y = from_location
        if self.direction == self.R:
            return iclass((x + 1, y))
        elif self.direction == self.L:
            return iclass((x - 1, y))
        elif self.direction == self.U:
            return iclass((x, y - 1))
        elif self.direction == self.D:
            return iclass((x, y + 1))
```

Lớp **XYEnvironment** cho môi trường trong mặt phẳng 2D với các vị trí là các điểm (x, y).

```
class XYEnvironment(Environment):
    """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."""

    def __init__(self, width=10, height=10):
        super().__init__()

        self.width = width
        self.height = height
        self.observers = []
        # Sets iteration start and end (no walls).
        self.x_start, self.y_start = (0, 0)
        self.x_end, self.y_end = (self.width, self.height)

    perceptible_distance = 1

    def things_near(self, location, radius=None):
        """Return all things within radius of location."""
        if radius is None:
            radius = self.perceptible_distance
        radius2 = radius * radius
        return [(thing, radius2 - distance_squared(location,
thing.location))
                for thing in self.things if distance_squared(
                location, thing.location) <= radius2]
```

```
    def percept(self, agent):
        """By default, agent perceives things within a default radius."""
        return self.things_near(agent.location)

    def execute_action(self, agent, action):
        agent.bump = False
        if action == 'TurnRight':
            agent.direction += Direction.R
        elif action == 'TurnLeft':
            agent.direction += Direction.L
        elif action == 'Forward':
            agent.bump = self.move_to(agent,
agent.direction.move_forward(agent.location))
        elif action == 'Grab':
            things = [thing for thing in self.list_things_at(agent.location)
if agent.can_grab(thing)]
            if things:
                agent.holding.append(things[0])
                print("Grabbing ", things[0].__class__.__name__)
                self.delete_thing(things[0])
        elif action == 'Release':
            if agent.holding:
                dropped = agent.holding.pop()
                print("Dropping ", dropped.__class__.__name__)
                self.add_thing(dropped, location=agent.location)

    def default_location(self, thing):
        location = self.random_location_inbounds()
        while self.some_things_at(location, Obstacle):
            # we will find a random location with no obstacles
            location = self.random_location_inbounds()
        return location

    def move_to(self, thing, destination):
        """Move a thing to a new location. Returns True on success or False
if there is an Obstacle.
        If thing is holding anything, they move with him."""
        thing.bump = self.some_things_at(destination, Obstacle)
        if not thing.bump:
            thing.location = destination
            for o in self.observers:
                o.thing_moved(thing)
            for t in thing.holding:
                self.delete_thing(t)
                self.add_thing(t, destination)
                t.location = destination
        return thing.bump

    def add_thing(self, thing, location=None,
exclude_duplicate_class_items=False):
        """Add things to the world. If (exclude_duplicate_class_items) then
the item won't be
        added if the location has at least one item of the same class."""
        if location is None:
            super().add_thing(thing)
        elif self.is_inbounds(location):
            if (exclude_duplicate_class_items and
```

```
                    any(isinstance(t, thing.__class__) for t in
self.list_things_at(location))):
                return
            super().add_thing(thing, location)

    def is_inbounds(self, location):
        """Checks to make sure that the location is inbounds (within walls
if we have walls)"""
        x, y = location
        return not (x < self.x_start or x > self.x_end or y < self.y_start
or y > self.y_end)

    def random_location_inbounds(self, exclude=None):
        """Returns a random location that is inbounds (within walls if we
have walls)"""
        location = (random.randint(self.x_start, self.x_end),
                    random.randint(self.y_start, self.y_end))
        if exclude is not None:
            while location == exclude:
                location = (random.randint(self.x_start, self.x_end),
                            random.randint(self.y_start, self.y_end))
        return location

    def delete_thing(self, thing):
        """Deletes thing, and everything it is holding (if thing is an
agent)"""
        if isinstance(thing, Agent):
            del thing.holding

        super().delete_thing(thing)
        for obs in self.observers:
            obs.thing_deleted(thing)

    def add_walls(self):
        """Put walls around the entire perimeter of the grid."""
        for x in range(self.width):
            self.add_thing(Wall(), (x, 0))
            self.add_thing(Wall(), (x, self.height - 1))
        for y in range(1, self.height - 1):
            self.add_thing(Wall(), (0, y))
            self.add_thing(Wall(), (self.width - 1, y))

        # Updates iteration start and end (with walls).
        self.x_start, self.y_start = (1, 1)
        self.x_end, self.y_end = (self.width - 1, self.height - 1)

    def add_observer(self, observer):
        """Adds an observer to the list of observers.
        An observer is typically an EnvGUI.

        Each observer is notified of changes in move_to and add_thing,
        by calling the observer's methods thing_moved(thing)
        and thing_added(thing, loc)."""
        self.observers.append(observer)

    def turn_heading(self, heading, inc):
        """Return the heading to the left (inc=+1) or right (inc=-1) of
```

```
heading."""
        return turn_heading(heading, inc)
```

Lớp GraphicEnvironment để hiển thị ra giao diện:

```
from ipythonblocks import BlockGrid
from IPython.display import HTML, display, clear_output
from time import sleep

from agents import  XYEnvironment

class GraphicEnvironment(XYEnvironment):
    def __init__(self, width=10, height=10, boundary=True, color={},
display=False):
        """Define all the usual XYEnvironment characteristics,
        but initialise a BlockGrid for GUI too."""
        super().__init__(width, height)
        self.grid = BlockGrid(width, height, fill=(200, 200, 200))
        if display:
            self.grid.show()
            self.visible = True
        else:
            self.visible = False
        self.bounded = boundary
        self.colors = color

    def get_world(self):
        """Returns all the items in the world in a format
        understandable by the ipythonblocks BlockGrid."""
        result = []
        x_start, y_start = (0, 0)
        x_end, y_end = self.width, self.height
        for x in range(x_start, x_end):
            row = []
            for y in range(y_start, y_end):
                row.append(self.list_things_at((x, y)))
            result.append(row)
        return result

    """
    def run(self, steps=1000, delay=1):
        "" "Run the Environment for given number of time steps,
        but update the GUI too." ""
        for step in range(steps):
            sleep(delay)
            if self.visible:
                self.reveal()
            if self.is_done():
                if self.visible:
                    self.reveal()
                return
            self.step()
```

```
        if self.visible:
            self.reveal()
    """

    def run(self, steps=1000, delay=1):
        """Run the Environment for given number of time steps,
        but update the GUI too."""
        for step in range(steps):
            self.update(delay)
            if self.is_done():
                break
            self.step()
        self.update(delay)

    def update(self, delay=1):
        sleep(delay)
        self.reveal()

    def reveal(self):
        """Display the BlockGrid for this world - the last thing to be added
        at a location defines the location color."""
        self.draw_world()
        # wait for the world to update and
        # apply changes to the same grid instead
        # of making a new one.
        clear_output(1)
        self.grid.show()
        self.visible = True

    def draw_world(self):
        self.grid[:] = (200, 200, 200)
        world = self.get_world()
        for x in range(0, len(world)):
            for y in range(0, len(world[x])):
                if len(world[x][y]):
                    self.grid[y, x] = self.colors[world[x][y][-
1].__class__.__name__]

    def conceal(self):
        """Hide the BlockGrid for this world"""
        self.visible = False
        display(HTML(''))
```

Lớp Công viên trong mặt phẳng 2 chiều (Park2D):

```
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], 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
```

Lớp BlindDog

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

    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
```

Chạy trên Colab:

```
if __name__ == "__main__":
    park = Park2D(5, 20, color={'BlindDog': (200, 0, 0), 'Water': (0, 200, 200),
                        'Food': (230, 115, 40)})  # park width is set to 5, and height to 20
    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 any food!")
    park.run(20)
```