**Virtual Pi2Go Programming: WS33 Sample Answers and Troubleshooting**

**Note:** This worksheet is supposed to give some idea how complex programs/classes can be built up from simpler ones by creating a class that draws upon multiple objects. While it serves as an example, the worksheet doesn’t really seek to *teach* how this may be done and students may benefit from discussion and encouragement to think about what is going on here.

**Question 1:** The students should get varying results depending upon whether they supply the values for left, right and centre or the objects gets the values from the virtual Pi2Go’s sensors. Students may need some help importing the pi2go module, initialising the pi2go and passing it as an argument to the class.

**Exercise 1:**

import simclient.simrobot as pi2go

import time

class LineState:

 def \_\_init\_\_(self, robot, left=None, right=None):

 if (left is None):

 self.left = robot.irLeftLine()

 else:

 self.left = left

 if (right is None):

 self.right = robot.irRightLine()

 else:

 self.right = right

 def print\_state(self):

 string = "(" + str(self.left) + " ," + str(self.right) + ")"

 print(string)

**Question 2:** It prints False.

**Exercise 2:**

import simclient.simrobot as pi2go

import time

class ObstacleState:

 def \_\_init\_\_(self, robot, left=None, right=None, centre=None):

 if (left is None):

 self.left = robot.irLeft()

 else:

 self.left = left

 if (right is None):

 self.right = robot.irRight()

 else:

 self.right = right

 if (centre is None):

 self.centre = robot.irCentre()

 else:

 self.centre = centre

 def \_\_eq\_\_(self, other):

 if self.left == other.left and self.right == other.right and self.centre == other.centre:

 return True

 else:

 return False

 def \_\_ne\_\_(self, other):

 if self.left != other.left or self.right != other.right or self.centre != other.centre:

 return True

 else:

 return False

 def print\_state(self):

 string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

 print(string)

**Exercise 3 & 4:** This is the answer to exercise 3 including the hash function from exercise 4. Going into depth about hash functions is outside the scope of these worksheets and has not been explained in detail. The Wikipedia page on hash functions is respectable, if quite technical, if students want to learn more.

class State:

 def calculate\_reward(self):

 return 1

 def print\_state(self):

 print("This is the base class")

class ObstacleState(State):

 def \_\_init\_\_(self, robot, left=None, right=None, centre=None):

 if (left is None):

 self.left = robot.irLeft()

 else:

 self.left = left

 if (right is None):

 self.right = robot.irRight()

 else:

 self.right = right

 if (centre is None):

 self.centre = robot.irCentre()

 else:

 self.centre = centre

 def calculate\_reward(self):

 if self.right and not self.centre:

 return 4

 elif not self.left:

 return 3

 elif self.centre:

 return 2

 else:

 return 0

 def \_\_eq\_\_(self, other):

 if self.left == other.left and self.right == other.right and self.centre == other.centre:

 return True

 else:

 return False

 def \_\_ne\_\_(self, other):

 if self.left != other.left or self.right != other.right or self.centre != other.centre:

 return True

 else:

 return False

 def \_\_hash\_\_(self):

 return self.left+3\*self.right+7\*self.centre

 def print\_state(self):

 string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

 print(string)

class LineState(State):

 def \_\_init\_\_(self, robot, left=None, right=None):

 if (left is None):

 self.left = robot.irLeftLine()

 else:

 self.left = left

 if (right is None):

 self.right = robot.irRightLine()

 else:

 self.right = right

 def calculate\_reward(self):

 if self.right and not self.left:

 return 2

 elif self.right == self.left:

 return 1

 else:

 return 0

 def \_\_eq\_\_(self, other):

 if self.left == other.left and self.right == other.right:

 return True

 else:

 return False

 def \_\_ne\_\_(self, other):

 if self.left != other.left or self.right != other.right:

 return True

 else:

 return False

 def \_\_hash\_\_(self):

 return self.left+3\*self.right

 def print\_state(self):

 string = "(" + str(self.left) + " ," + str(self.right) + ")"

 print(string)

**Exercise 5:** Showing just the RewardDictionary classes – but the State classes will also be needed.

class RewardDictionary():

 def \_\_init\_\_(self, robot):

 self.reward\_dictionary = {}

 self.learning\_rate = 0.5

 self.robot = robot

 def newState(self, robot):

 print("Base Class: Not Implemented")

 def max\_reward(self):

 print("Base Class: Not Implemented")

 def best\_action(self, state):

 max\_reward = 0

 for act in self.actions:

 if (self.reward\_dictionary[(state, act)] > max\_reward):

 action = act

 max\_reward = self.reward\_dictionary[(state, act)]

 return action

 def update\_reward(self, state, action, reward):

 self.reward\_dictionary([(state, action)]) = self.reward\_dictionary([state, action]) + (reward - self.reward\_dictionary([state, action]))\*self.learning\_rate

class ObstacleRewardDictionary():

 def \_\_init\_\_(self, robot, actions):

 super().\_\_init\_\_()

 for i in range(0, 2):

 for j in range(0, 2):

 for k in range(0, 2):

 for a in (actions):

 self.reward\_dictionary[(ObstacleState(robot, i, j, k), a)] = 2

 def newState(self):

 return ObstacleState(self.robot)

 def max\_reward(self):

 return 4

class LineRewardDictionary():

 def \_\_init\_\_(self, robot, actions):

 super().\_\_init\_\_(robot)

 for i in range(0, 2):

 for j in range(0, 2):

 for a in (actions):

 self.reward\_dictionary[(LineState(robot, i, j), a)] = 2

 def newState(self):

 return LineState(self.robot)

 def max\_reward(self):

 return 2

**Exercise 6:**

import simclient.simrobot as pi2go

import time

import random

class State:

 def calculate\_reward(self):

 return 1

 def print\_state(self):

 print("This is the base class")

class ObstacleState(State):

 def \_\_init\_\_(self, robot, left=None, right=None, centre=None):

 if (left is None):

 self.left = robot.irLeft()

 else:

 self.left = left

 if (right is None):

 self.right = robot.irRight()

 else:

 self.right = right

 if (centre is None):

 self.centre = robot.irCentre()

 else:

 self.centre = centre

 def calculate\_reward(self):

 if self.right and not self.centre:

 return 4

 elif not self.left:

 return 3

 elif self.centre:

 return 2

 else:

 return 0

 def \_\_eq\_\_(self, other):

 if self.left == other.left and self.right == other.right and self.centre == other.centre:

 return True

 else:

 return False

 def \_\_ne\_\_(self, other):

 if self.left != other.left or self.right != other.right or self.centre != other.centre:

 return True

 else:

 return False

 def \_\_hash\_\_(self):

 return self.left+3\*self.right+7\*self.centre

 def print\_state(self):

 string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

 print(string)

class LineState(State):

 def \_\_init\_\_(self, robot, left=None, right=None):

 if (left is None):

 self.left = robot.irLeftLine()

 else:

 self.left = left

 if (right is None):

 self.right = robot.irRightLine()

 else:

 self.right = right

 def calculate\_reward(self):

 if self.right and not self.left:

 return 2

 elif self.right == self.left:

 return 1

 else:

 return 0

 def \_\_eq\_\_(self, other):

 if self.left == other.left and self.right == other.right:

 return True

 else:

 return False

 def \_\_ne\_\_(self, other):

 if self.left != other.left or self.right != other.right:

 return True

 else:

 return False

 def \_\_hash\_\_(self):

 return self.left+3\*self.right

 def print\_state(self):

 string = "(" + str(self.left) + " ," + str(self.right) + ")"

 print(string)

class RewardDictionary():

 def \_\_init\_\_(self, robot, actions):

 self.reward\_dictionary = {}

 self.learning\_rate = 0.5

 self.actions = actions

 self.robot = robot

 def newState(self):

 print("Base Class: Not Implemented")

 def max\_reward(self):

 print("Base Class: Not Implemented")

 def best\_action(self, state):

 max\_reward = 0

 for act in self.actions:

 if (self.reward\_dictionary[(state, act)] > max\_reward):

 action = act

 max\_reward = self.reward\_dictionary[(state, act)]

 return action

 def update\_reward(self, state, action, reward):

 self.reward\_dictionary[(state, action)] = self.reward\_dictionary[state, action] + (reward - self.reward\_dictionary[state, action])\*self.learning\_rate

class ObstacleRewardDictionary(RewardDictionary):

 def \_\_init\_\_(self, robot, actions):

 super().\_\_init\_\_(robot, actions)

 for i in range(0, 2):

 for j in range(0, 2):

 for k in range(0, 2):

 for a in (actions):

 self.reward\_dictionary[(ObstacleState(robot, i, j, k), a)] = 2

 def newState(self):

 return ObstacleState(self.robot)

 def max\_reward(self):

 return 4

class LineRewardDictionary(RewardDictionary):

 def \_\_init\_\_(self, robot, actions):

 super().\_\_init\_\_(robot, actions)

 for i in range(0, 2):

 for j in range(0, 2):

 for a in (actions):

 self.reward\_dictionary[(LineState(robot, i, j), a)] = 1

 def newState(self):

 return LineState(self.robot)

 def max\_reward(self):

 return 2

class MachineLearner():

 def \_\_init\_\_(self, reward\_dictionary):

 self.reward\_dictionary = reward\_dictionary

 self.epsilon = 1

 self.epsilon\_reduce = 0.05

 def execute\_action(self, action):

 if (action == "forward"):

 pi2go.forward(10)

 elif (action == "left"):

 pi2go.spinLeft(10)

 elif (action == "right"):

 pi2go.spinRight(10)

 else:

 pi2go.stop()

 time.sleep(3)

 def learn(self):

 while (self.epsilon > 0):

 explore = random.random()

 state = self.reward\_dictionary.newState()

 if (explore < self.epsilon):

 action = random.choice(self.reward\_dictionary.actions)

 print("Random Action: " + action)

 else:

 action = self.reward\_dictionary.best\_action(state)

 print("Best Action: " + action)

 self.execute\_action(action)

 new\_state = self.reward\_dictionary.newState()

 reward = new\_state.calculate\_reward()

 self.reward\_dictionary.update\_reward(state, action, reward)

 if (reward == self.reward\_dictionary.max\_reward()):

 self.epsilon = self.epsilon - self.epsilon\_reduce

 print("New epsilon: " + str(self.epsilon))

 pi2go.stop()

**Exericse 7:** Students should be encouraged to take their results from exercise 6 and adapt for this exercise by adding in cognitive agent aspects. They will need to take care to add self. in all the right places. They may also want to look at some of the exercise answers for WS32 and Ex32 to get ideas for how to do things like controlling getting input from the user.

import bdi.pi2goagent as cognitive

import time

import random

class State:

 def calculate\_reward(self):

 return 1

 def print\_state(self):

 print("This is the base class")

class ObstacleState(State):

 def \_\_init\_\_(self, pi2go, left=None, right=None, centre=None):

 if (left is None):

 self.left = pi2go.irLeft()

 else:

 self.left = left

 if (right is None):

 self.right = pi2go.irRight()

 else:

 self.right = right

 if (centre is None):

 self.centre = pi2go.irCentre()

 else:

 self.centre = centre

 def calculate\_reward(self):

 if self.right and not self.centre:

 return 4

 elif not self.left:

 return 3

 elif self.centre:

 return 2

 else:

 return 0

 def \_\_eq\_\_(self, other):

 if self.left == other.left and self.right == other.right and self.centre == other.centre:

 return True

 else:

 return False

 def \_\_ne\_\_(self, other):

 if self.left != other.left or self.right != other.right or self.centre != other.centre:

 return True

 else:

 return False

 def \_\_hash\_\_(self):

 return self.left+3\*self.right

 def print\_state(self):

 string = "(" + str(self.centre) + " ," + str(self.left) + " ," + str(self.right) + ")"

 print(string)

class LineState(State):

 def \_\_init\_\_(self, pi2go, left=None, right=None):

 if (left is None):

 self.left = pi2go.irLeftLine()

 else:

 self.left = left

 if (right is None):

 self.right = pi2go.irRightLine()

 else:

 self.right = right

 def calculate\_reward(self):

 if self.right and not self.left:

 return 2

 elif self.right == self.left:

 return 1

 else:

 return 0

 def \_\_eq\_\_(self, other):

 if self.left == other.left and self.right == other.right:

 return True

 else:

 return False

 def \_\_ne\_\_(self, other):

 if self.left != other.left or self.right != other.right:

 return True

 else:

 return False

 def \_\_hash\_\_(self):

 return self.left+3\*self.right

 def print\_state(self):

 string = "(" + str(self.left) + " ," + str(self.right) + ")"

 print(string)

class RewardDictionary():

 def \_\_init\_\_(self, actions, robot):

 self.reward\_dictionary = {}

 self.learning\_rate = 0.5

 self.actions = actions

 self.robot = robot

 def newState(self):

 print("Base Class: Not Implemented")

 def max\_reward(self):

 print("Base Class: Not Implemented")

 def best\_action(self, state):

 max\_reward = 0

 for act in self.actions:

 if (self.reward\_dictionary[(state, act)] > max\_reward):

 action = act

 max\_reward = self.reward\_dictionary[(state, act)]

 return action

 def update\_reward(self, state, action, reward):

 self.reward\_dictionary[(state, action)] = self.reward\_dictionary[state, action] + (reward - self.reward\_dictionary[state, action])\*self.learning\_rate

class ObstacleRewardDictionary(RewardDictionary):

 def \_\_init\_\_(self, actions, robot):

 super().\_\_init\_\_(actions, robot)

 for i in range(0, 2):

 for j in range(0, 2):

 for k in range(0, 2):

 for a in (actions):

 self.reward\_dictionary[(ObstacleState(robot, i, j, k), a)] = 2

 def newState(self):

 return ObstacleState(self.robot)

 def max\_reward(self):

 return 4

class LineRewardDictionary(RewardDictionary):

 def \_\_init\_\_(self, actions, robot):

 super().\_\_init\_\_(actions, robot)

 for i in range(0, 2):

 for j in range(0, 2):

 for a in (actions):

 self.reward\_dictionary[(LineState(robot, i, j), a)] = 1

 def newState(self):

 return LineState(self.robot)

 def max\_reward(self):

 return 2

class MachineLearner(cognitive.Pi2GoAgent):

 def \_\_init\_\_(self):

 super().\_\_init\_\_()

 self.epsilon = 1

 self.epsilon\_reduce = 0.05

 self.actions = ['forward', 'left', 'right']

 self.oval\_reward\_dictionary = LineRewardDictionary(self.actions, self.robot)

 self.wall\_reward\_dictionary = ObstacleRewardDictionary(self.actions, self.robot)

 self.add\_condition\_rule(self.G('end'), self.end)

 self.add\_condition\_rule(self.AND(self.NOT(self.G('end')), self.AND(self.NOT(self.G('follow\_oval')), self.NOT(self.G('follow\_wall')))), self.ask)

 self.add\_condition\_rule(self.B('switch\_pressed'), self.drop\_goals)

 self.add\_condition\_rule(self.G('learn\_oval'), self.learn\_oval)

 self.add\_condition\_rule(self.G('learn\_wall'), self.learn\_wall)

 self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.NOT(self.B('learn\_oval'))), self.first\_learn\_oval)

 self.add\_condition\_rule(self.AND(self.G('follow\_wall'), self.NOT(self.B('learn\_wall'))), self.first\_learn\_wall)

 self.add\_condition\_rule(self.AND(self.G('follow\_oval'), self.B('learn\_oval')), self.execute\_best\_oval)

 self.add\_condition\_rule(self.AND(self.G('follow\_wall'), self.B('learn\_wall')), self.execute\_best\_wall)

 def drop\_goals(self):

 self.drop\_goal('follow\_oval')

 self.drop\_goal('follow\_wall')

 self.robot.stop()

 def execute\_action(self, action):

 if (action == "forward"):

 self.robot.forward(10)

 elif (action == "left"):

 self.robot.spinLeft(10)

 elif (action == "right"):

 self.robot.spinRight(10)

 else:

 self.robot.stop()

 time.sleep(3)

 def ask(self):

 self.drop\_belief('end')

 goal = input('What would you like me to do? [O]follow\_oval,[W]follow\_wall,[E]end')

 if (goal == "O"):

 self.add\_goal('follow\_oval')

 elif (goal == "W"):

 self.add\_goal('follow\_wall')

 else:

 self.add\_goal('end')

 def end(self):

 self.add\_belief('end')

 self.done()

 def first\_learn\_oval(self):

 self.add\_goal('learn\_oval')

 def first\_learn\_wall(self):

 self.add\_goal('learn\_wall')

 def learn\_oval(self):

 self.epsilon = 1

 self.learn(self.oval\_reward\_dictionary)

 self.add\_belief('learn\_oval')

 def learn\_wall(self):

 self.epsilon = 1

 print("setting epsilon to 1")

 self.learn(self.wall\_reward\_dictionary)

 self.add\_belief('learn\_wall')

 def execute\_best(self, reward\_dictionary):

 state = reward\_dictionary.newState()

 action = reward\_dictionary.best\_action(state)

 self.execute\_action(action)

 def execute\_best\_oval(self):

 self.execute\_best(self.oval\_reward\_dictionary)

 def execute\_best\_wall(self):

 self.execute\_best(self.wall\_reward\_dictionary)

 def learn(self, reward\_dictionary):

 while (self.epsilon > 0):

 explore = random.random()

 state = reward\_dictionary.newState()

 if (explore < self.epsilon):

 action = random.choice(reward\_dictionary.actions)

 print("Random Action: " + action)

 else:

 action = reward\_dictionary.best\_action(state)

 print("Best Action: " + action)

 self.execute\_action(action)

 new\_state = reward\_dictionary.newState()

 reward = new\_state.calculate\_reward()

 reward\_dictionary.update\_reward(state, action, reward)

 if (reward == reward\_dictionary.max\_reward()):

 self.epsilon = self.epsilon - self.epsilon\_reduce

 print("New epsilon: " + str(self.epsilon))

 self.robot.stop()



 University of Liverpool, 2020

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).