**Virtual Pi2Go Programming: Bringing it all Together**

**AIM:** After completing this worksheet you should be able to use optional arguments in functions and methods and use None for when a variable has no value. You should also have integrated your knowledge of objects, classes, inheritance, cognitive agents and machine learning into one complex and flexible piece of code.

**You Need:** To complete this worksheet you need to have a virtual Pi2Go simulator (see WS1), and to be able to use files to store Programs (WS5). You also need to know the commands to operate the Pi2Go motors, LEDs and sensors (WS3 & WS4). You should be able to use Python’s time module (WS6), variables (WS12), functions (WS16) and objects (WS27 & WS31). You should also understand how to use rules (WS29 & WS30) and goal (WS32) with cognitive agents. You need to understand simple machine learning algorithms (WS24) and a wall following algorithm (WS18). You will benefit from having programs that result from the exercises in exercise sheet 32.

**If the simulator isn’t already running: Start the Simulator, Select the Pi2Go Simulation and square.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

**Optional Arguments and None**

Sometimes we want a class to have two different initialisation methods. For instance, imagine wanting to create a State class for a machine learning program. We might want to be able to initialise the class directly by calling the sensors, or we might want to pass in the sensor values. To do this we use *optional* arguments to the class’s methods.

Consider the following class:

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 print\_state(self):

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

print(string)

The \_\_init\_\_ method has three optional arguments, left, right and centre and a compulsory argument, robot. The optional arguments are given a *default* value which is the value they will take if no argument is supplied. In this case it is a special value, None. We can test if a variable is None by using the keyword is.

So this class gives the fields self.left, self.right and self.centre values from some robot’s sensors (presumably the Pi2Go’s). Otherwise it takes the value supplied to the \_\_init\_\_ method.

**Question 1:** Experiment with this class to see what happens when you supply 1, 2, 3 or 4 arguments when you create an object of this class. Explain the output you see. Don’t forget to initialise the Pi2Go with pi2go.init() before creating an object that will try to access its sensors.

**Note:** Normal functions can also have optional arguments as well as methods appearing in classes.

**Exercise 1:** Write a LineState class. Objects in this class are instantiated with a robot when they are created. The objects then either take values from the two line sensors of the robot or they get passed the values when the object is created and store these values as fields.

**Comparing Objects**

We are used to being able to compare two objects to see if they are equal using == but this won’t work with objects. Try loading the ObstacleState class and typing the following:

>>> pi2go.init()

>>> state1 = ObstacleState(0, 0, 0)

>>> state2 = ObstacleState(0, 0, 0)

>>> state1 == state2

**Question 2:** What happens?

In order for == (and !=) to work properly we need to implement two methods:

1. \_\_eq\_\_(self, other) which returns True if other equals the object
2. \_\_ne\_\_(self, other) which returns True if other doesn’t equal the object

**Exercise 2:** Implement and test \_\_eq\_\_(self, other) and \_\_ne\_\_(self, other) for the ObstacleState class.

**Inheritance from a Base Class** Often when we are use objects and inheritance in programs we create a base class that we don’t actually use but which sets out all the methods that its sub-classes must have and then create sub-classes for specific things. Consider the following State class

class State:

def calculate\_reward(self):

return 1

def print\_state(self):

print("This is the base class")

**Exercise 3:** Create two sub-classes of State:

1. ObstacleState: This uses the values of the three infra-red obstacle sensors from a robot (supplied when an object is created) to create the state (or the values can be supplied when the object is created). Its calculate\_reward method should return: 4 if there is an obstacle on the right and no obstacle in front of the robot; otherwise it returns 3 if there is no obstacle to the left of the robot; otherwise it returns 2 if there is an obstacle in front of the robot, and 0 otherwise.
2. LineState: This uses the values of the two line sensors from a robot (supplied when an object is created) to create the state (or the values can be supplied when the object is created). Its calculate\_reward method should return: 2 if the right sensor detects a line and the left sensor does not; 1 if both sensors return the same value; and 0 otherwise.

We are going to want to use a tuple of our State classes with an action (a string) as a key in a dictionary. In order to use objects as keys in dictionaries you have to implement one further method \_\_hash\_\_(self). This helps the programming language efficiently look up keys. This *hash function* should return an integer and the number it returns for any object shouldn’t change during the lifetime of an object. If two objects are equal, then they should return the same hash number. Objects which are not equal *do not have to* return different hash numbers, but you do not want all objects to return the same hash number.

**Exercise 4:** Add the following hash function to ObstacleState and LineState:

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

Return self.left + 3\*self.right

**Exercise 5:** Create a base class called RewardDictionary() which has three fields:

1. reward\_dictionary which is a dictionary of (State, action) pairs as keys and integers as values
2. learning\_rate is a float
3. robot is a robot to be used to get sensor values from.

and four methods:

1. newState(self, robot) should do nothing except print a message saying this is a method in the base class.
2. max\_reward(self) should do nothing except print a message saying this is a method in the base class.
3. best\_action(self, state) should return the best action to take in state based on the values in reward\_dictionary
4. update\_reward(self, state, action, reward) which updates the current value for (state, action) in reward\_dictionary by adding (reward – self.reward\_dictionary([(state, action)])\*self.learning\_rate to it.

Create two subclasses of RewardDictionary():

1. ObstacleRewardDictionary: This is initialised with a robot and a list of actions and should have a field that contains a dictionary of all possible (ObstacleState, action) pairs as keys with 2 as the value.
2. LineRewardDictionary: This is initialised with a robot and a list of actions and should have a field that contains a dictionary of all possible (LineState, action) pairs as keys with 1 as the value.

In both cases newState(self, robot) should return the current state of the robot (either its ObstacleState) or its (LineState) and max\_reward(self) should return the highest reward possible for the dictionary (4 for ObstacleRewardDictionary and 2 for LineRewardDictionary).

**Exercise 6:** Create a MachineLearner class that is initialised with a reward dictionary and then learns to maximise its reward according to that dictionary. You may want to use the machine learning program from WS24 as a starting point, or the MachineLearner class from exercise sheet 27 as your starting point.

**Exercise 7:** Create a cognitive agent machine learner that asks whether it should follow an oval or follow a wall. If it doesn’t know how to do one of these it then uses the appropriate RewardDictionary class to learn the action and then it follows the best policy. If the switch is pressed while it is following a policy then the program ends.



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/).