**Programming in Python with the Pi2Go Simulator: Part 4**

**Chapter 1: Introduction to Classes and Objects**



**AIM:** After completing this chapter you should be able to explain what a class and an object are and create classes and objects in Python.

**You Need:** To complete this chapter you should be familiar with the material in parts 1-3.

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

Just as it is sometimes useful to package up sequences of code that you use often into functions, it can also be useful to package together sets of functions and data. We do this using *classes* and *objects*.

A **class** is some code that describes how some data and functions can be grouped together.

An **object** is a specific instance of a class that can be used in a program.

Below is an example of a DataLogger class.

import simclient.simrobot as pi2go

class DataLogger:

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

 self.readings = []

 def take\_reading(self):

 self.readings.append(pi2go.getDistance())

 def print\_readings(self):

 print(self.readings)

Create a file containing this program and run it in idle.

Now type the following at the Python Command line:

>>> pi2go.init()

>>> data1 = DataLogger()

>>> datal.print\_readings()

**Question 1:** What happens?

Data1 is an object of class DataLogger. The three functions in the class are referred to as *methods.* Methods in Python *always* take self as their first argument. self refers to the object that instantiates the class.

All classes must have an \_\_init\_\_ method that is used to create an object. When you typed

>>> data1 = DataLogger()

You created an object of class DataLogger called data\_l. This was done by executing the \_\_init\_\_(self) method.

**Question 2:** Looking at the code for \_\_init\_\_(self) what happened when you created the object?

self.readings is data associated with the object. We refer to the data associated with objects as *fields*. So self.readings is one of DataLogger’s fields. In this case self.readings is a list of readings from the distance sensor. The class contains one method for adding a reading to this list and another method for printing out the readings in the list.

When we execute a method associated with an object we use the syntax *object.method().* Notice that we don’t supply the self argument. This is *object.* So, when you typed

>>> datal.print\_readings()

You called the function print\_readings(self) in the DataLogger class and passed in data1 as the first argument.

Recall that data stored in an object is referred to as a *field.* So, readings, is a field in the DataLogger class. You can access fields directly using the syntax *object.field*

Type:

>>> data1.readings

**Question 3:** What happens?

**Question 4:** Which method adds a reading to the list?

**Question 4:** Which method prints out the readings in the list?

Add a box to your simulation from the edit window, add some readings to data1 for different distances of this box from the Pi2Go.

**Question 5:** Now print out the readings. What do you see?

You can have several objects of the same class. Create an object data2 of the DataLogger class.

**Question 6:** What Python command do you use to do this?

Take some readings for different distances of object using both data1 and data2. Move the objects between taking a reading with data1 and taking a reading with data2.

**Question 7:** Explain why data1.print\_readings() and data2.print\_readings() are not the same.

**Exercise 1:** Create a class that will log the readings from the left and right infra-red sensors. It should have the following methods:

take\_irRight\_reading(self) this should take a reading from the right infra-red sensor and add it to a log for that sensor.

take\_irLeft\_reading(self) this should take a reading from the left infra-red sensor and add it to a log for that sensor.

take\_reading(self) this should take a reading both infra-red sensors and add them to the appropriate logs.

print\_irRight\_log(self) this print the log for the right infra-red sensor.

print\_irLeft\_log(self) this print the log for the left infra-red sensor.

print\_readings(self) this print the log for both sensors.

**Using Arguments when Creating Objects**

Just like functions can take arguments, so can methods, including the \_\_init\_\_ method of an object. Say for instance we wanted our data logger objects to have names: We could change the init method to:

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

 self.readings = []

 self.logger\_name = name

You can see we have added a new field self.logger\_name to our class. When we want to create an object of this class, we will have to supply a name as an argument, for instance we might call

>>> logger1 = DataLogger(‘logger1’)

It is very common to have two methods associated with a field which are called the *getter* and *setter* methods. The first method returns the value of the field and the second method sets the value of the field. The setter will need to take a second argument, as well as self, but when it is called only one argument will be given - the second argument – self is still ignored.

**Exercise 2:** Write a new version of the DataLogger class which has the \_\_init\_\_ method shown above and two additional methods:

 get\_name(self) returns the Data Logger object’s name

 set\_name(self, name) sets the Data Logger’s name to name

Once you have written your program run it and type the following:

>>> logger1 = DataLogger('logger1')

>>> logger1.get\_name()

>>> logger1.set\_name('logger2')

>>> logger1.get\_name()

**Question 8:** What happens and why?

**Chapter 2: Introduction to Classes and Objects Exercises**

**AIM:** This chapter provides additional exercises using Objects and Classes.

**Exercise 1:** Consider the Wall Following program you wrote for WS18. Convert this into a class WallFollower() with methods follow\_wall(side) (which follows a wall on side until the switch is pressed) and drive\_to\_wall(). Illustrate the use of this class by writing programs to create a WallFollower() object to get the Pi2Go robot to follow the wall into the house where the robot starts next to the wall and a second where it starts some way from the wall.

**Note:** When an object calls one of its own methods it needs to use self.*method\_name()* not just the method name.

**Exercise 2:** Adapt your WallFollower class so it is initialised with the side on which it should follow the wall and has a method follow\_wall(self) that only takes self as an argument. A program can use both a right wall follower and a left wall follower object. Illustrate the use of this class with a program that follows the wall on its right until it detects a black floor and then follows the wall on its left.

**Hint:** Note that direction still needs to be supplied to spin because it has to spin in both directions.

**Exercise 3:** Create a MachineLearner class from the program you wrote for WS24 This should have a learn() method that will learn to drive on around the black oval from **oval.xml** world and a follow\_policy() method that will just drive according to the learned reward table.

**Chapter 3: Introduction to Cognitive Agents**

**AIM:** After completing this chapter you should be able to describe what a cognitive agent is.

**You Need:** To complete this chapter you should be familiar with the material in parts 1-3 and Objects (chapter 1)

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

### What are Cognitive Agents?

Cognitive agents (sometimes called *Beliefs-Desires-Intentions* or *BDI* agents) are an artificial intelligence technology. A cognitive agent has beliefs about the world and goals it wants to achieve. Its actions are governed by its beliefs and goals. In this worksheet we will explore a cognitive agent module for use with the Pi2Go simulator.

At the idle command line type the following:

import bdi.pi2goagent as cognitive

This imports a Python module that implements cognitive agent programming. The first thing we need to do is create an agent. This is an object of Pi2GoAgent class.

Type:

agent = cognitive.Pi2GoAgent()

agent.init()

The agent will control our Initio simulator. This means we still need to initialise our simulation.

### Beliefs

The agent can check all its sensors for *perceptions*. It can do this using the getPercepts() method. When it checks its sensors, it stores their values in its *BeliefBase.*

Type

agent.getPercepts()

print(agent.beliefbase)

**Question 1:** What is printed?

The belief base is a dictionary of pairs of strings and values. So we have the following strings for each sensor

|  |  |
| --- | --- |
| String | Sensor |
| distance | Ultrasonic distance Sensor |
| Obstacle\_centre | Centre infra-red sensor |
| obstacle\_right | Right infra-red sensor |
| obstacle\_left | Left infra-red sensor |
| line\_right | Right line sensor |
| line\_left | Left line sensor |
| switch\_pressed | The switch |
| lightFL | Front left light sensor |
| lightFR | Front right light sensor |
| lightBL | Back left light sensor |
| lightBR | Back right light sensor |

**Question 2:** What does your agent believe is the current value of the ultrasonic distance sensor?

You can add your own beliefs to the belief base using the methods add\_belief and change\_belief.

You use add\_belief if the belief you are adding has a true/false value. So, for instance you could add a belief that you are currently testing the agent:

Type:

agent.add\_belief(‘test’)

**Question 3:** Now print out the contents of the belief base. How has it changed?

You use change\_belief if the belief you are adding as some other value, such as a string. So, for instance, you could add a belief that the agent’s name is “pi2go”.

Type:

agent.change\_belief(‘name’,’pi2go’)

**Quesiton 4:** Now print out the contents of the belief base. How has it changed?

### You can remove a belief from the agent’s belief base using remove\_belief

Type:

agent.remove\_belief(‘name’)

**Question 5:** Now print out the contents of the belief base. How has it changed?

### Important: Since the belief base is a dictionary you can check if something is in the belief base using dictionary functions. For instance

Type:

‘distance’ in agent.beliefbase

**Question 6:** What happens? Explain why.

Type:

agent.beliefbase[‘distance’]

**Question 7:** What happens? Explain why.

### Goals

The agents goals are things it wants to achieve. That is things it wants to believe are true. The agent’s *goalbase* is a list of strings representing its goals. You can add a goal using the method add\_goal.

Type:

agent.add\_goal(‘a\_goal’)

print(agent.goalbase)

**Question 8:** What is printed?

When an agent believes a goal then it can be removed. This is done using the method check\_goals().

Type:

agent.add\_belief(‘a\_goal’)

agent.check\_goals()

print(agent.goalbase)

**Question 9:**What is printed and why has the goal base changed?

**Controlling the Robot**

You may wish to control the robot through the agent. The Simulated Pi2Go is stored as a data field, robot.

Type:

agent.robot.forward(10)

agent.robot.stop()

**Question 10:** What happens and why?

**Chapter 4: Exercises with Beliefs and Goals**

**AIM:** This chapter provides additional exercises programming with cognitive/BDI agents, beliefs and goals.

**Exercise 1**: Write an obstacle avoidance agent. So, if the agent believes there is an obstacle it turns and if it does not believe there is an obstacle it moves forward. If it believes that the switch is pressed it should stop.

**Exercise 2:** Program up obstacle avoidance behaviour for a cognitive agent using the idea that if there is something closer than 50cm from the robot then it should believe there is an obstacle. Then it should turn or move forward depending upon whether or not it believes there is an obstacle

**Exercise 3:** Write an agent that has a goal to find an obstacle. While it has this goal it should move forward. When it believes the goal has been achieved (i.e., it believes there is an obstacle) it should stop.

**Exercise 4:** Write a square following agent that will move around the edge of the square in **square.xml** keeping one line sensor in the square and one line sensor out of the square and turning if both sensors are out of the square. If it believes that the switch is pressed it should stop.

**Exercise 5:** Write an agent that has a goal to ‘edge\_square’ which it can achieve by first achieving ‘line\_left’ and then executing the algorithm to go around the edge of the square. When the user presses the switch the agent should drop the goal and stop.

**Chapter 5: The Cognitive Agent Reasoning Cycle**

**AIM:** After completing this chapter you should be able to explain the Reasoning cycle for your Python agent and program simple rules for it.

**You Need:** To complete this chapter you should be familiar with parts 1-3. You should know about objects (chapter 1) and be familiar with basic Cognitive Agent constructs (chapter 3).

**If the simulator isn’t already running: Start the Simulator, Select the Pi2go Simulation and default\_world.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

**The Reasoning Cycle:** As well as beliefs and goals the Pi2Go Agent has a *reasoning cycle.* When the cycle is active the agent does four things in turn:

1. It checks all its sensors and updates its belief base
2. It checks to see if any goals have been achieved and removes them
3. It checks to see if any BDI *rules* are applicable. If some are it picks one and,
4. executes it

**How do you make the reasoning cycle active?** By calling the method run\_agent (run\_agent will also initialise the agent if you have not already done so)

**What is a BDI rule?** A BDI rule is a python function that has been added to the agent.

**How do you add a rule to the agent?** By calling the method add\_rule

**When is a rule applicable?** Rules are always applicable unless they have a condition. We will talk about conditions in the next section.

Consider the following program:

import bdi.pi2goagent as cognitive

agent = cognitive.Pi2GoAgent()

def print\_beliefs():

 print(agent.beliefbase)

 return

agent.add\_rule(print\_beliefs)

agent.run\_agent()

**Question 1:** What will happen when it is run?

**Exercise 1:** Modify the program so that it prints out just the value of the distance sensor on each cycle through the reasoning cycle.

**Adding conditions to rules:** It isn’t very useful having a rule that is always applicable. It would be better if rules were only executed in certain situations. In order to do this, we the method add\_condition\_rule this takes two arguments – the first is a function that be executed to decide if the rule is applicable and the second is a rule function.

Consider the following program:

import bdi.pi2goagent as cognitive

agent = cognitive.Pi2GoAgent()

def check\_distance():

 if (agent.beliefbase['distance'] < 50):

 return True

 return False

def print\_beliefs():

 print(agent.beliefbase)

 return

agent.add\_condition\_rule(check\_distance, print\_beliefs)

agent.run\_agent()

**Question 2:** What will happen when it is run?

**Belief Support:** The Pi2GoAgent has methods that will help with referring to beliefs in conditions:

*agent*.believe(*key*)creates a function that checks whether *key* takes the value 1 or True in the agent’s belief base

*agent*.B(*key*)is the same as *agent.*believe(*key*) but is quicker to type.

**Exercise 2:** Write a program that will make the robot reverse if it believes there is an obstacle in front of the centre infra-red sensor. **Remember:** you can use agent.robot to access the robot’s functions.

**Stopping the Reasoning Cycle:** You can stop the agent reasoning cycle by calling the function done()

**Exercise 3:** Modify your program from the last exercise that will make the robot reverse for five seconds if it believes there is an obstacle in front of it and then the robot will stop, and the agent will stop the reasoning cycle.

**Chapter 6: Using Logic in BDI Rule Conditions**

**AIM:** After completing this chapter you should be able to use logical expressions in the conditions for BDI rules.

**You Need:** To complete this chapter you should be familiar with parts 1-3. You should understand about objects (chapter 1) and cognitive agents (chapters 3 & 4).

**If the simulator isn’t already running: Start the Simulator, Select the Pi2go Simulation and default\_world.xml, then start IDLE (open a *new IDLE window* if you have used IDLE to start the simulator).**

**BDI Rule Conditions:** You are accustomed when using if … then … else in normal Python programs to be able to use logic in the conditions – for instance using **and** or **not**. You can’t use the regular Python expressions for this in the conditions for BDI rules because the conditions are functions not expressions but the bdi.pi2goagent library provides substitutes for these which you can use instead.

|  |  |
| --- | --- |
| Function | Example |
| NOT | agent.NOT(agent.B(‘obstacle\_centre’)) |
| AND | agent.AND(agent.B(‘obstacle\_right’), agent.B(‘obstacle\_left’)) |
| OR | agent.OR(agent.B(‘obstacle\_right’), agent.B(‘obstacle\_left’)) |

Consider the following condition functions:

cond = agent.AND(agent.B('switch\_pressed'), agent.NOT(agent.B('obstacle\_centre')))

**Question 1:** When does the function **cond** return true?

cond = agent.AND(agent.B('switch\_pressed'), agent.NOT(agent.B('started')))

**Question 2:** When does the function **cond** return true? Remember that as well as getting beliefs from sensors agent programs can add beliefs to the belief base.

cond = agent.AND(agent.B('switch\_pressed'), agent.B('started'))

**Question 3:** When does the function **cond** return true?

cond = agent.B('started')

**Question 4:** When does the function **cond** return true?

Consider the following program.

import bdi.pi2goagent as cognitive

import time

agent = cognitive.Pi2GoAgent()

def start\_agent():

 agent.add\_belief('started')

 time.sleep(5)

 return

def stop\_agent():

 agent.drop\_belief('started')

 agent.add\_belief('stopping')

 time.sleep(5)

 return

def forward():

 agent.robot.forward(10)

 return

def stop\_rule():

 agent.robot.stop()

 agent.done()

 agent.drop\_belief('stopping')

 return

start = agent.AND(agent.B('switch\_pressed'), agent.NOT(agent.B('started')))

stop = agent.AND(agent.B('switch\_pressed'), agent.B('started'))

agent.add\_condition\_rule(start, start\_agent)

agent.add\_condition\_rule(stop, stop\_agent)

agent.add\_condition\_rule(agent.B('started'), forward)

agent.add\_condition\_rule(agent.B('stopping'), stop\_rule)

agent.run\_agent()

**Question 5:** What will happen when it is run?

**Exercise 1:** Modify the program so that it starts when the switch is briefly pressed and stops when it is pressed again. When it has started it moves forward when it does not believe there is an obstacle in front of it and turns when it does believe there is an obstacle in front of it.

**Chapter 7: Exercises with BDI Rules**

**AIM:** This chapter provides additional exercises programming with BDI rules.

**Exercise 1**: Write a set of BDI rules that will get your Pi2Go robot to follow the line in **line\_following.xml** world. The robot should move forward if its line sensors are either side of the line and turn left or right, as appropriate, if one of the line sensors detects the line. The agent will need to start on the line to work properly.

**Exercise 2:** Write a set of BDI rules for a wall following agent, that will move forward if the agent detects a wall on its right, turn left if it detects and obstacle to the front and a wall to its right, and turn right if it doesn’t detect a wall to its right. The agent will need to start next to a wall to work properly. The agent should stop if a black surface is detected. You can test this agent in **house.xml** world or **zigzag.xml** world.

**Chapter 8: Inheritance**



**AIM:** After completing this chapter you should be able to explain what inheritance means, what a sub-class and a super-class are and how methods can be overridden and create sub-classes in Python.

**You Need:** To complete this chapter you should be familiar with parts 1-3. You should understand objects (chapter 1). You should also understand how to use rules with cognitive agents (chapters 5 & 6).

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

We have established that classes are a convenient way to group data and functions together and that objects are particular instances of classes. One of the powerful tools that programming with objects gives us is the ability to make *sub-classes.* A sub-class *inherits* all the data and methods of its *super-class* but if it wishes to it can add its own data and methods or even *override* the methods of the super-class. We will look at each of these things in turn.

The following is an example of how you create a sub-class

import bdi.pi2goagent as cognitive

class NameAgent(cognitive.Pi2GoAgent):

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

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

 self.name = "alice"

 def getName(self):

 return(self.name)

 def changeName(self,new\_name):

 self.name = new\_name

Here we have created a class called NameAgent by sub-classing Pi2GoAgent. We do this by supplying the name of the class we are sub-classing as an argument to the class name.

Next we make sure the first thing we do in \_\_init\_\_(self): is to create the super-class. This is the line:

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

In our name agent class, we have then created a new field which contains a name for the agent and two new methods, getName(self) and changeName(self) which can be used to find out the robot’s name and to change the robots name.

Run this module and then type the following at the Python Command line.

bob = NameAgent()

>>> bob.getName()

**Question 1:** What happens?

**Question 2:** Explain why?

Now Type

>>> bob.changeName('bob')

>>> bob.getName()

**Question 3:** What happens?

**Question 4:** Explain why?

Now Type

bob.robot.init()

>>> bob.robot.forward(10)

>>> bob.robot.stop()

**Question 5:** What happens?

**Question 6:** Explain why you can use bob.robot when robot isn’t mentioned in the NameAgent class code?

Consider the following program:

import bdi.pi2goagent as cognitive

import time

class ReverseAgent(cognitive.Pi2GoAgent):

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

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

 self.add\_condition\_rule(self.B('obstacle\_centre'), self.reverse\_rule)

 def reverse\_rule(self):

 self.robot.reverse(10)

 time.sleep(5)

 self.robot.stop()

**Question 7:** What does it do?

**Question 8:** What should you type at the command line to create an object of this class. What should you type in order to get that object to react to obstacles?

**Exercise 1:** Create a class called ReverseAgent which adds a rule to the agent during initialisation that will cause the robot to reverse for 5 seconds if it detects an obstacle in front of it and will exit its reasoning cycle if it the switch is pressed.

**Overriding Methods**

Suppose I wanted to have a reverse agent but instead of it reversing for 5 seconds and then stopping I wanted it to reverse for 1 second and then stop? In this case I can sub-class ReverseAgent and just replace its reverse\_rule method.

This is done in the following program

import bdi.pi2goagent as cognitive

import time

class ReverseAgent(cognitive.Pi2GoAgent):

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

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

 self.add\_condition\_rule(self.B('obstacle\_centre'), self.reverse\_rule)

 def reverse\_rule(self):

 self.robot.reverse(10)

 time.sleep(5)

 self.robot.stop()

class ShortReverseAgent(ReverseAgent):

 def reverse\_rule(self):

 self.robot.reverse(10)

 time.sleep(1)

 self.robot.stop()

**Notice That:** ShortReverseAgent doesn’t have an \_\_init\_\_ method. This is because we don’t want to change the \_\_init\_\_ method from its superclass.

We say that ShortReverseAgent’s reverse\_rule has *overridden* ReverseAgent’s reverse\_rule

**Exercise 2:** Create a class called ForwardAgent which adds a rule to the agent during initialisation that will cause the robot to move forward for 5 seconds if it doesn’t detect an obstacle in front of it and will exit its reasoning cycle if it detects an obstacle on the left. Create a second class called ShortForwardAgent that behaves like ForwardAgent except it only moves forward for 1 second.

**Calling the super-method when Overriding**

Suppose that we don’t want to simply replace a method from a super-class, but we want to execute that method and then do something extra either before or after executing it. In this case we need to call the super-method.

We do this by using the syntax super().*methodName()* (in fact when we initialise the super class we are calling the super-class’s \_\_init\_\_() method)

Consider the following program:

import bdi.pi2goagent as cognitive

import time

class ReverseAgent(cognitive.Pi2GoAgent):

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

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

 self.add\_condition\_rule(self.B('obstacle\_centre'), self.reverse\_rule)

 self.add\_condition\_rule(self.B(‘switch\_pressed’), self.done)

 def reverse\_rule(self):

 self.robot.reverse(10)

 time.sleep(5)

 self.robot.stop()

 def done(self):

 super().done()

 print("Exited the Reasoning Cycle")

**Question 9:** Which methods from which class are being overriden?

**Question 10:** What does it do?

**Chapter 9: Exercises with Inheritance**

**AIM:** This chapter provides additional exercises using inheritance.

**Exercise 1:** Create a class LineFollower() that is a sub-class of Pi2GoAgent. This class should follow the line in **line\_following.xml** world when the method run\_agent() is called. The robot should move forward if its line sensors are either side of the line and turn left or right, as appropriate, if one of the line sensors detects the line. The agent will need to start on the line to work properly.

**Exercise 2:** Create a class SwitchActivatedAgent() that is a sub-class of Pi2GoAgent. This agent should

1. add a belief started when the switch is pressed (then wait for a couple of seconds for the switch to be unpressed).
2. The agent should add a belief stopping when the switch is pressed *and* the agent believes started, and then it should drop the belief started.
3. Lastly the agent should stop the pigo and call the done() method in the Pi2GoAgent class, if it believes stopping.

**Exercise 3:** Adapt your agent from Exercise 1, so that it is a sub-class of SwitchActivatedAgent.

**Exercise 4:** Create a class WallFollower() that is a sub-class of SwitchActivatedAgent. This class should a wall when the method run\_agent() is called. The robot should move forward if the agent detects a wall on its right, turn left if it detects and obstacle to the front and a wall to its right, and turn right if it doesn’t detect a wall to its right. The agent will need to start next to a wall to work properly. The agent should stop if a black surface is detected. You can test this agent in **house.xml** world or **zigzag.xml** world.

**Chapter 10: BDI Goals**



**AIM:** After completing this chapter you should be able to explain how BDI goals differ from beliefs and use them in programs.

**You Need:** To complete this chapter you should be familiar with the material in parts 1-3. You should be able to use classes and objects (chapters 3 & 8). You should also understand how to use rules with cognitive agents (chapters 5 & 6).

**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).**

We have seen how we can use beliefs in rule conditions to control an agent’s behaviour. We can also use goals. Unlike beliefs, goals are sometimes automatically removed. This happens when they are *achieved.*

Consider the following program

import bdi.pi2goagent as cognitive

import time

class FindSquareAgent(cognitive.Pi2GoAgent):

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

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

 self.add\_condition\_rule(self.G('line\_left'), self.forward\_rule)

 self.add\_goal('line\_left')

 def forward\_rule(self):

 self.robot.forward(10)

 time.sleep(5)

 self.robot.stop()

**Question 1:** What does this agent do?

**Question 2:** Create an instance of this agent and run its reasoning cycle. What happens?

**Exercise 1:** Change to **house.xml** world. Create an agent that starts with a goal to enter the house (by achieving line\_left). While it has this goal, the agent should move forward until it detects the wall of the house and then follow the wall until it enters the house. You may want to look back at your answers to WS18 To help with this.

**Hint:** Consider creating the following variables to represent conditions you will need in your \_\_init\_\_ method

only\_line\_goal = self.AND(self.G('line\_left'), self.NOT(self.G('obstacle\_centre')))

goal\_and\_obstacle = self.AND(self.G('line\_left'), self.B('obstacle\_centre'))

goal\_and\_no\_wall = self.AND(self.G('line\_left'), self.NOT(self.B('obstacle\_right')))

goal\_and\_wall = self.AND(self.G('line\_left'), self.AND(self.B('obstacle\_right'), self.NOT(self.B('obstacle\_centre'))))

.

**Chapter 11: Exercises with Goals**

**AIM:** This chapter provides additional exerciss programming with goals in the BDI reasoning cycle.

**Exercise 1**: Create a cognitive agent that, when it has a goal to learn to follow an oval, learn\_oval, (as in Part 3, chapter 7) it selects an action (either at random or the current best action – depending upon epsilon), tests to see the outcome and modifies its reward dictionary accordingly. When epsilon equals zero it adds a belief that it has learned to follow the oval.

**Exercise 2:** Create a cognitive agent that has a goal to follow\_oval but can’t do this until it has first learned to follow the oval.

**Exercise 3:** Create a cognitive agent for the world **zigzag.xml**. If the agent has no goal, prompts the user to give it a goal (which can be to enter a black floor space or a clear floor space). When it has achieved a goal, it prompts the user for a new one. If the user enters end, then the agent stops.

**Chapter 12: Bringing it all Together**

**AIM:** After completing this chapter 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 chapter you should be familiar with the material in parts 1-3. You should be able to use objects (chapters 1 & 8). You should also understand how to use rules (chapters 5 & 6) and goal (chapter 10) with cognitive agents. You need to understand simple machine learning algorithms (part 3, chapter 7) and a wall following algorithm (part 2, chapter 10). You will benefit from having programs that result from the exercises in chapter 12.

**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/).