**Virtual Pi2Go Programming: Inheritance**



**AIM:** After completing this worksheet 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 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). You should also understand how to use rules with cognitive agents (WS29 & WS30).

**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()

What happens?

Explain why?

Now Type

>>> bob.changeName('bob')

>>> bob.getName()

What happens?

Explain why?

Now Type

bob.robot.init()

>>> bob.robot.forward(10)

>>> bob.robot.stop()

What happens?

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()

What does it do?

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:** 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:** 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")

Which methods from which class are being overriden?

What does it do?



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/).