**Virtual Pi2Go Programming: Introduction to Classes and Objects**



**AIM:** After completing this worksheet 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 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 understand Python Functions (WS16), and Data Structures (WS19-21).

**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()

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.

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

What happens?

Which method adds a reading to the list?

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.

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.

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.

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()

What happens and why?



University of Liverpool, 2019

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