
ROS Basics in 5 days

Extra Unit Using Python Classes in ROS



- Summary -

Estimated time to completion: 3 hours

What will you learn with this unit?

- [Introduction to OOP and Python classes](#)
- [How to use Python classes in ROS](#)

- End of Summary -

- Notes -

Note for experienced programmers

If you already know about Object-Oriented Programming, and Python Classes, you can skip the introductory explanations and go straight for the exercises.

- End of Notes -

1 Object-Oriented Programming

In all the programs we've wrote until now, we have designed our programs around functions, which manipulate data. This is called the procedure-oriented way of programming. But let me tell you that there is another way of programming! This one is based on combining data and functionality, and wrapping them inside "things" known as objects. This way of programming is known as object-oriented programming, or OOP. You may be asking... why are you telling me about this? And why now? And those are great questions!

Most of the time, you will still be able to use procedural programming, but when writing larger and more complex programs, this method can become a pain. In these cases, it is much better to use OOP, since your code will be better organized and it will be much easier to understand (and debug).

Anyway, let me remind you that the goal of this unit is not to learn what OOP is, or what a Python class is, but how to apply these concepts to your ROS code. This a ROS course, isn't it? So, we'll try to summarize these concepts in the easiest and fastest way possible, just to put everything in context.

So, as I've already said before, OOP is based on the concept of "objects." These objects are usually defined by two things:

- Attributes: This is the data associated with the object, defined in fields.
- Methods: These are the procedures or functions associated with the object.

The most important feature of objects is that the methods associated with them can access and modify the data fields of the object with which they are associated.

2 What is a Python Class?

So, everything sounds super cool and interesting, but I haven't read a word yet about Python classes. What the heck are Python classes? Let's see!

A Python class is, basically, a code template for creating an object. An object is created using the constructor of the class. This object will then be called the instance of the class. Well, what about seeing an example of a Python class, so that we can understand everything better? Let's see then!

****Python File: jedi_class.py****

```
In [ ]: class Jedi:
        def __init__(self, name):
            self.name = name

        def say_hi(self):
            print('Hello, my name is ', self.name)

j = Jedi('ObiWan')
j.say_hi()
```

****END Python File: jedi_class.py****

Let's quickly analyze the above class.

```
In [ ]: class Jedi:
```

Here, we are creating a new class named Jedi.

```
In [ ]: def __init__(self, name):  
        self.name = name
```



This is the `__init__` method of the class. It is also known as the constructor because it will be called as soon as an object of a class is created. It is usually used for the initialization of the attributes. In this case, we are initializing an attribute of the class, which is the name of the Jedi.

```
In [ ]: def say_hi(self):  
        print('Hello, my name is', self.name)
```



This is another method of the class. In this case, it just contains a simple print inside. So, when this method is called, the print will be executed.

```
In [ ]: j = Jedi('ObiWan')
```



Here we are creating an object of the Jedi class, which will be stored on the variable `j`. Remember that when this is executed, the `__init__` method of the class will also be automatically executed. So, in this case, the string `'ObiWan'` will be stored in the `self.name` attribute of the class.

```
In [ ]: j.say_hi()
```



Finally, we are calling the `say_hi()` method from the class. So, this will result in the following string being printed: **Hello, my name is Obiwan.**

Note that we are using the `self` keyword. This keyword refers to the object itself. Each time we define a new method of the class, we will need to pass this keyword as the first argument. Also, when accessing the attributes of the class, we will use this keyword, as you can see in:

```
In [ ]: self.name
```



And that's it! As I've already mentioned before, this unit is not meant to exhaustively teach you how to work with Python classes, but to introduce you to them so that you can use them on your ROS projects. If you are interested in learning more about them, you can refer to the many online tutorials you will find on the web.

You can test the class above by creating the Python file, and executing using the **python** keyword.

► Execute in Terminal #1

```
In [ ]: python jedi_class.py
```



If everything goes fine, you should get the following output:

```
In [ ]: ('Hello, my name is', 'ObiWan')
```



3 Using Python classes in ROS

Great! So now that you already know what a Python Class is, and how it works, let's try to apply this to a ROS code. For instance, we can start by creating a simple class that will move the BB-8 robot in a circular movement, just as you did in the previous unit.

- Example P1 -

Below you can have a look at a class that will control the movement of the BB-8 robot.

****Python File: bb8_move_circle_class.py****

In []: `#!/usr/bin/env python`



```
import rospy
from geometry_msgs.msg import Twist

class MoveBB8():

    def __init__(self):
        self.bb8_vel_publisher = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
        self.cmd = Twist()
        self.ctrl_c = False
        self.rate = rospy.Rate(10) # 10hz
        rospy.on_shutdown(self.shutdownhook)

    def publish_once_in_cmd_vel(self):
        """
        This is because publishing in topics sometimes fails the first time you publish.
        In continuous publishing systems, this is no big deal, but in systems that publish only
        once, it IS very important.
        """
        while not self.ctrl_c:
            connections = self.bb8_vel_publisher.get_num_connections()
            if connections > 0:
                self.bb8_vel_publisher.publish(self.cmd)
                rospy.loginfo("Cmd Published")
                break
            else:
                self.rate.sleep()

    def shutdownhook(self):
        # works better than the rospy.is_shutdown()
        self.ctrl_c = True

    def move_bb8(self, linear_speed=0.2, angular_speed=0.2):
```

```

        self.cmd.linear.x = linear_speed
        self.cmd.angular.z = angular_speed

        rospy.loginfo("Moving BB8!")
        self.publish_once_in_cmd_vel()

if __name__ == '__main__':
    rospy.init_node('move_bb8_test', anonymous=True)
    movebb8_object = MoveBB8()
    try:
        movebb8_object.move_bb8()
    except rospy.ROSInterruptException:
        pass

```

****END Python File: bb8_move_circle_class.py****

Let's analyze some of the most important parts of the above class.

In []:

```

def __init__(self):
    self.bb8_vel_publisher = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
    self.cmd = Twist()
    self.ctrl_c = False
    self.rate = rospy.Rate(10) # 10hz
    rospy.on_shutdown(self.shutdownhook)

```



This is the constructor of the class. Here, we are defining several attributes:

- **bb8_vel_publisher**: Creates a Publisher for the **/cmd_vel** topic
- **cmd**: Contains a Twist message
- **ctrl_c**: Contains a boolean indicating if we have pressed Ctrl+C in order to stop the program
- **rate**: Contains the rate frequency

Also, as you can see in the last line of the `__init__` method, we are setting that when our ROS program is closed (**rospy.on_shutdown**), the class method **self.shutdownhook** will be triggered.

```
In [ ]: def publish_once_in_cmd_vel(self):  
        """  
        This is because publishing in topics sometimes fails the first time you publish.  
        In continuous publishing systems, this is no big deal, but in systems that publish only  
        once, it IS very important.  
        """  
        while not self.ctrl_c:  
            connections = self.bb8_vel_publisher.get_num_connections()  
            if connections > 0:  
                self.bb8_vel_publisher.publish(self.cmd)  
                rospy.loginfo("Cmd Published")  
                break  
            else:  
                self.rate.sleep()
```

This class method is used in order to make sure that the first message we publish into a topic is successfully received, as you can read on the commented code. Don't pay too much attention to the code inside it, since it's not important right now. Just keep in mind that you can use this class method for situations where you will publish one single command into a topic.

```
In [ ]: def shutdownhook(self):  
        self.ctrl_c = True
```

This method will be called when our ROS program is closed, as you saw on the constructor of the class. Therefore, we are setting the **ctrl_c** attribute to True.

```
In [ ]: def move_bb8(self, linear_speed=0.2, angular_speed=0.2):  
  
        self.cmd.linear.x = linear_speed  
        self.cmd.angular.z = angular_speed  
  
        rospy.loginfo("Moving BB8!")  
        self.publish_once_in_cmd_vel()
```

This method is used to move the robot. As you can see, we just fill in the **cmd** attribute with the values we want, and we then call the **publish_once_in_cmd_vel()** method.

```
In [ ]: if __name__ == '__main__':  
        rospy.init_node('move_bb8_test', anonymous=True)  
        movebb8_object = MoveBB8()  
        try:  
            movebb8_object.move_bb8()  
        except rospy.ROSInterruptException:  
            pass
```

Finally, in the main function, we are doing three things:

- We create a ROS node named **move_bb8_test**
- We create an object of the MoveBB8 class, which is stored into a variable named **movebb8_object**.
- We call the **move_bb8()** method of the class in order to start moving the BB-8 robot.

Note that when calling the **move_bb8()** method, we are using a **try/except** structure. This is because if the call to the method fails, the error would be caught by the **ROSInterruptException**, avoiding error messages.

Great! So now, we already have a class that moves our BB-8 robot in a circle movement. Let's test it. In this case, since we are using a ROS node, we are going to create a ROS package in which to place the class.

► Execute in Terminal #1

```
In [ ]: catkin_create_pkg my_python_class rospy
```



Now, add the [bb8_move_circle_class.py](#) file to your package and execute it by running the following command:

► Execute in Terminal #1

```
In [ ]: rosrn my_python_class bb8_move_circle_class.py
```



If everything is OK, you should see your BB-8 robot start moving in circles.

Excellent!! But now you may be asking yourself... what happens with the ROS Service? I don't see any ROS Service in the Python class above. It's just a Python script that moves the robot, right? Well yes, you're totally right!

What happens with the ROS Service is... nothing. Or everything. It's up to you. What I mean is that now you have a class that can be used in order to move the BB-8 robot. And it can be used by anybody, included a ROS Service, of course. In fact, now it's extremely easy to add this Python class to a Service Server. Let me show you an example below:

****Python File: bb8_move_circle_service_server.py****

```
In [ ]: #!/usr/bin/env python

import rospy
from std_srvs.srv import Empty, EmptyResponse
from bb8_move_circle_class import MoveBB8

def my_callback(request):
    rospy.loginfo("The Service move_bb8_in_circle has been called")
    movebb8_object = MoveBB8()
    movebb8_object.move_bb8()
    rospy.loginfo("Finished service move_bb8_in_circle")
    return EmptyResponse()

rospy.init_node('service_move_bb8_in_circle_server')
my_service = rospy.Service('/move_bb8_in_circle', Empty , my_callback)
rospy.loginfo("Service /move_bb8_in_circle Ready")
rospy.spin() # keep the service open.
```

END Python File: bb8_move_circle_service_server.py

So basically, we just have the following lines:

```
In [ ]: from move_bb8_circle_class import MoveBB8
```

This line of code imports the **MoveBB8** class we've created from the Python file **move_bb8_circle_class.py**.

```
In [ ]: movebb8_object = MoveBB8()
        movebb8_object.move_bb8()
```

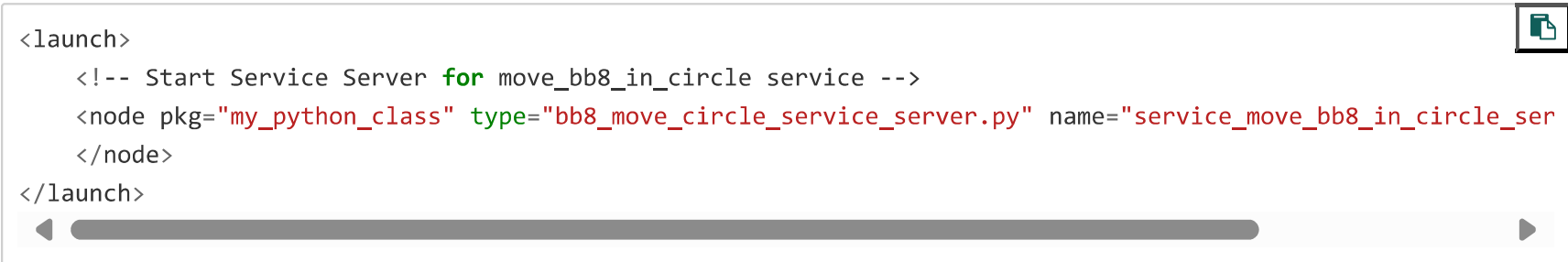
Then here, we are creating an object of the **MoveBB8**, and we are calling the **move_bb8()** method in order to start moving the robot. Quite simple, right?

So, let's test all this together. First, add the [bb8_move_circle_service_server.py](#) file to your package. Add also a launch file that starts the Service Server, like the one below:

****Launch File: bb8_move_circle_service_server.launch****

In []:

```
<launch>
  <!-- Start Service Server for move_bb8_in_circle service -->
  <node pkg="my_python_class" type="bb8_move_circle_service_server.py" name="service_move_bb8_in_circle_ser
  </node>
</launch>
```



****END Launch File: bb8_move_circle_service_server.launch****

Finally, execute the launch file and check that everything works as expected. That is, that the ROS service **/move_bb8_in_circle** is available, and that when you call it, the BB-8 robot starts moving in circles.

- End of Example P1 -

- Exercise P1 -

Modify the Python scripts you created in **Example P1** so that now they include the custom Service Message you used for **Exercise 3.3**, in the previous unit.

- You will need to modify the class so that now, the BB-8 robot stops moving after the specified time in the message has passed.
- You will also need to modify the Service Server code so that you pass the specified duration to the class.

- End of Exercise P1 -

4 Additional Materials to Learn More

Python Classes: <https://docs.python.org/2/tutorial/classes.html> (<https://docs.python.org/2/tutorial/classes.html>)



English
proofread