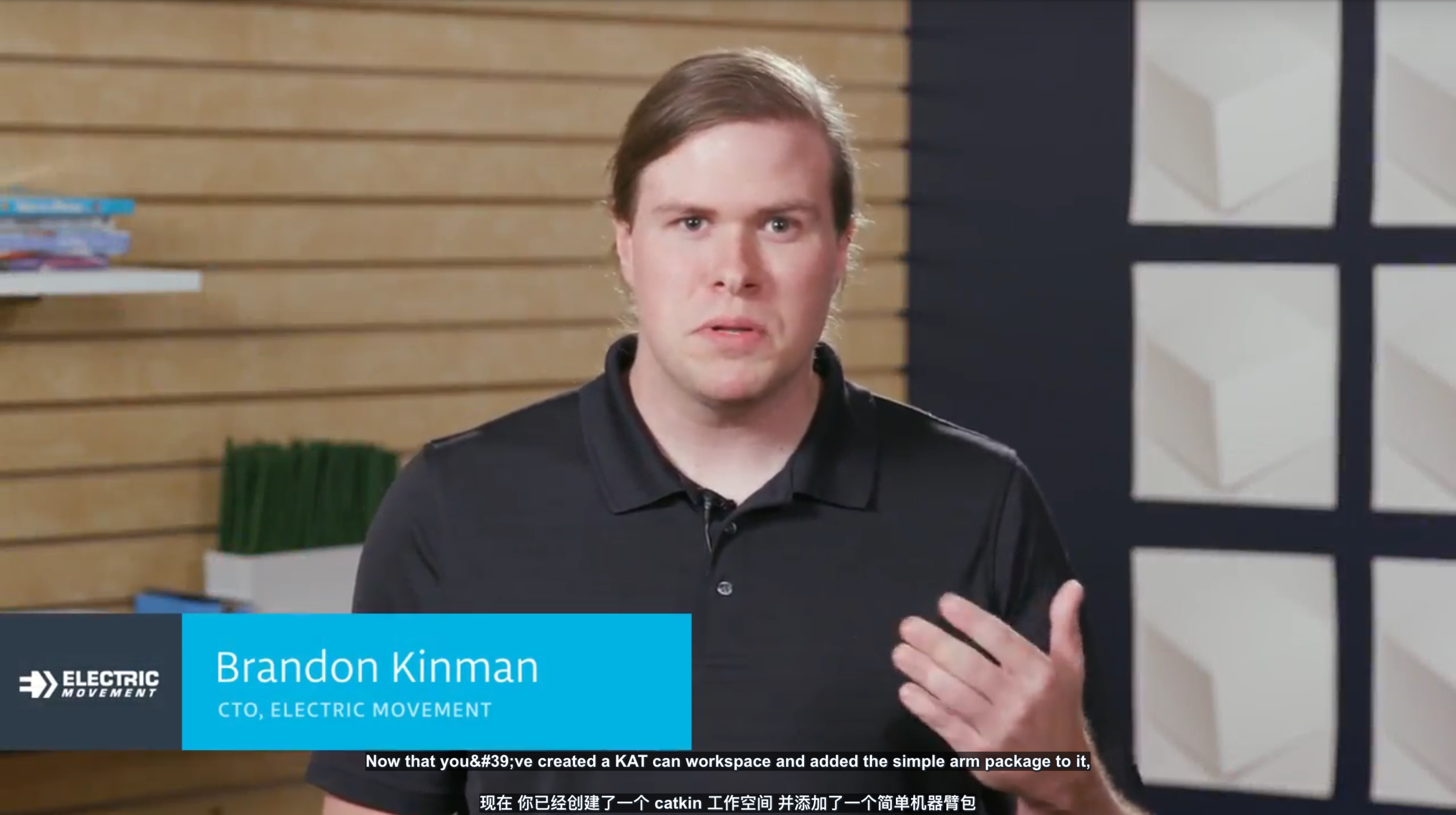
1. Closing In

Courage and recap from Sebastian



1. Overview



1. ROS Publishers

Before you see the code for simple\_mover, it may be helpful to see how ROS Publishers work in Python.

Publishers allow a node to send messages to a topic, so that data from the node can be used in other parts of the ROS system. In Python, ROS publishers typically have the following definition format, although other parameters and arguments are possible:

pub1 = rospy.Publisher("/topic\_name", message\_type, queue\_size=size)

The "/topic\_name" indicates which topic the publisher will be publishing to. The message\_type is the type of message being published on "/topic\_name".

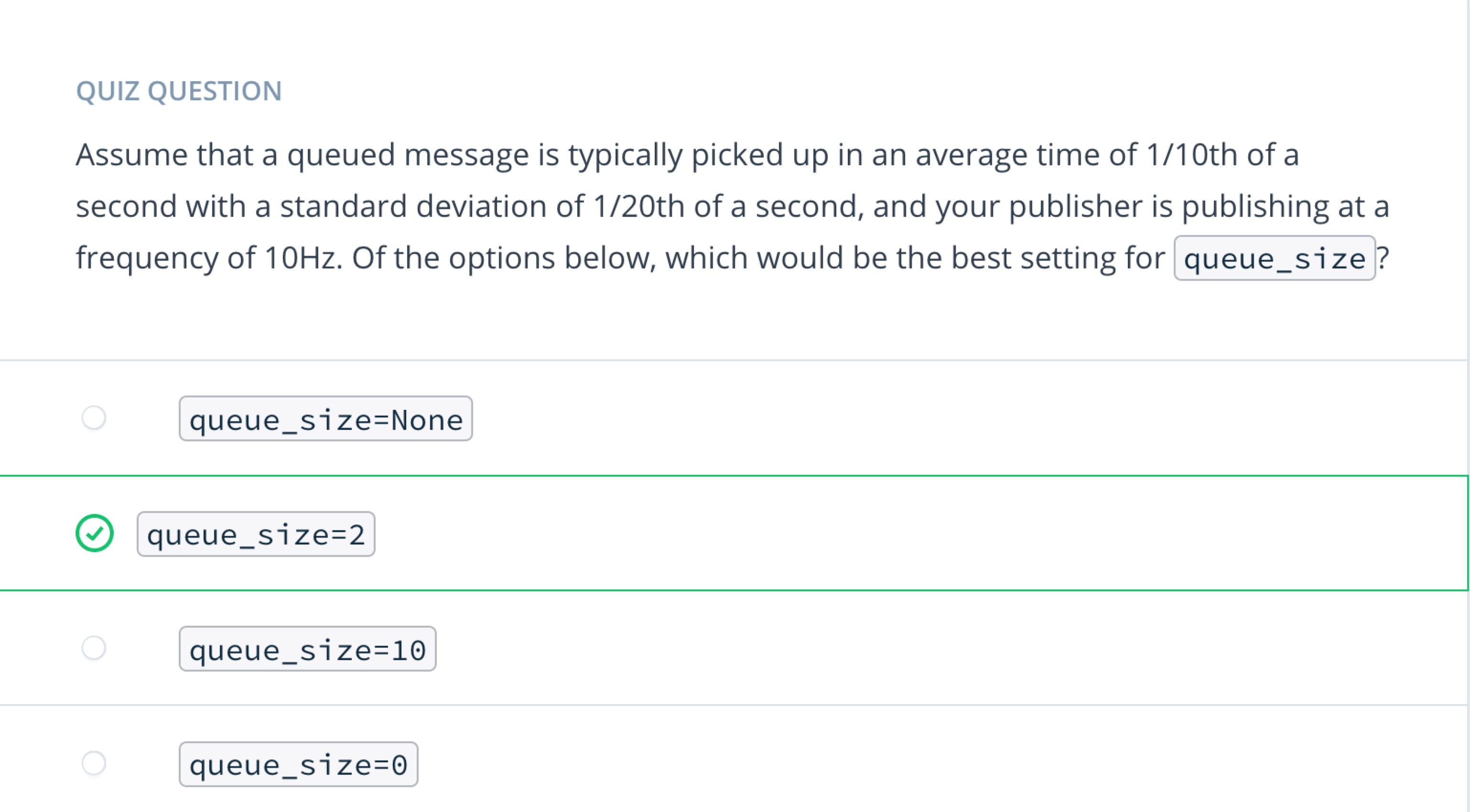
ROS publishing can be either synchronous or asynchronous:

* Synchronous publishing means that a publisher will attempt to publish to a topic but may be blocked if that topic is being published to by a different publisher. In this situation, the second publisher is blocked until the first publisher has serialized all messages to a buffer and the buffer has written the messages to each of the topic's subscribers. This is the default behavior of a rospy.Publisher if the queue\_size parameter is not used or set to None.
* Asynchronous publishing means that a publisher can store messages in a queue until the messages can be sent. If the number of messages published exceeds the size of the queue, the oldest messages are dropped. The queue size can be set using the queue\_size parameter.

Once the publisher has been created as above, a message with the specified data type can be published as follows:

pub1.publish(message)

For more information about ROS publishers, see [the documentation here](http://docs.ros.org/kinetic/api/rospy/html/rospy.topics.Publisher-class.html).



1. Simple Mover

You will now go through the process of implementing your first ROS node in python. This node is called simple\_mover. As it’s name implies, this node only has one responsibility, and that is to command joint movements for simple\_arm.

To do so, it must publish joint angle command messages to the following topics:

| **Topic Name** | **/simple\_arm/joint\_1\_position\_controller/command** |
| --- | --- |
| **Message Type** | std\_msgs/Float64 |
| **Description** | Commands joint 1 to move counter-clockwise, units in radians |
| **Topic Name** | **/simple\_arm/joint\_2\_position\_controller/command** |
| **Message Type** | std\_msgs/Float64 |
| **Description** | Commands joint 2 to move counter-clockwise, units in radians |

**Note**: If you no longer have the catkin workspace from the previous lesson, you can download a copy of it [here](https://github.com/udacity/simple_arm_01). Alternately, If you’d prefer to skip to the punch, you can download the entire, complete simple\_arm package from [here](https://github.com/udacity/simple_arm).

**Adding the scripts directory**

In order to create a new node in python, you must first create the scripts directory within the simple\_arm package, as it does not yet exist.

$ cd ~/catkin\_ws/src/simple\_arm/

$ mkdir scripts

## Creating a new script

Once the scripts directory has been created, executable scripts can be added to the package. However, in order for rosrun to find them, their permissions must be changed to allow execution. Let’s add a simple bash script that prints “Hello World” to the console.

$ cd scripts

$ echo '#!/bin/bash' >> hello

$ echo 'echo Hello World' >> hello

After setting the appropriate execution permissions on the file, rebuilding the workspace, and sourcing the newly created environment, you will be able to run the script.

$ chmod u+x hello

$ cd ~/catkin\_ws

$ catkin\_make

$ source devel/setup.bash

$ rosrun simple\_arm hello



And there you have it! You have now added a script

## Creating the empty simple\_mover node script

To create the simple\_mover node script, you will must simply follow the same basic routine introduced a moment ago.

$ cd ~/catkin\_ws/src/simple\_arm

$ cd scripts

$ touch simple\_mover

$ chmod u+x simple\_mover

You can now edit the empty simple\_mover script with your favorite text editor.

Let’s write the code!

1. Simple Mover: The Code

Below is the complete code for the simple\_mover node, in it’s entirety, followed by a step-by-step explanation of what is happening. You can copy and paste this code into the simple\_mover script you created in the ~/catkin\_ws/src/simple\_arm/scripts/ directory like this:

First, open a new terminal, next:

$ cd ~/catkin\_ws/src/simple\_arm/scripts/

$ nano simple\_mover

You have opened the simple\_mover script with the **nano** editor, now copy and paste the code below into the script and use ctrl-x followed by y then enter to save the script.

## simple\_mover

#!/usr/bin/env python

import math

import rospy

from std\_msgs.msg import Float64

def mover():

pub\_j1 = rospy.Publisher('/simple\_arm/joint\_1\_position\_controller/command',

Float64, queue\_size=10)

pub\_j2 = rospy.Publisher('/simple\_arm/joint\_2\_position\_controller/command',

Float64, queue\_size=10)

rospy.init\_node('arm\_mover')

rate = rospy.Rate(10)

start\_time = 0

while not start\_time:

start\_time = rospy.Time.now().to\_sec()

while not rospy.is\_shutdown():

elapsed = rospy.Time.now().to\_sec() - start\_time

pub\_j1.publish(math.sin(2\*math.pi\*0.1\*elapsed)\*(math.pi/2))

pub\_j2.publish(math.sin(2\*math.pi\*0.1\*elapsed)\*(math.pi/2))

rate.sleep()

if \_\_name\_\_ == '\_\_main\_\_':

try:

mover()

except rospy.ROSInterruptException:

pass

## The code: Explained

#!/usr/bin/env python

import math

import rospy

rospy is the official Python client library for ROS. It provides most of the fundamental functionality required for interfacing with ROS via Python. It has interfaces for creating Nodes, interfacing with Topics, Services, Parameters, and more. It will certainly be worth your time to check out the API documentation [here](http://docs.ros.org/kinetic/api/rospy/html/). General information about rospy, including other tutorials may be found on the [ROS Wiki](http://wiki.ros.org/rospy_tutorials/Tutorials/WritingPublisherSubscriber).

from std\_msgs.msg import Float64

From the std\_msgs package, we import Float64, which is one of the primitive message types in ROS. The [std\_msgs](http://wiki.ros.org/std_msgs) package also contains all of the other primitive types. Later on in this script, we will be publishing Float64 messages to the position command topics for each joint.

def mover():

pub\_j1 = rospy.Publisher('/simple\_arm/joint\_1\_position\_controller/command',

Float64, queue\_size= 10)

pub\_j2 = rospy.Publisher('/simple\_arm/joint\_2\_position\_controller/command',

Float64, queue\_size=10)

At the top of the mover function, two publishers are declared, one for joint 1 commands, and one for joint 2 commands. Here, the queue\_size parameter is used to determine the maximum number messages that may be stored in the publisher queue before messages are dropped. More information about this parameter can be found [here](http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers#queue_size:_publish.28.29_behavior_and_queuing).

rospy.init\_node('arm\_mover')

Initializes a client node and registers it with the master. Here “arm\_mover” is the name of the node. init\_node() must be called before any other rospy package functions are called. The argument anonymous=True makes sure that you always have a unique name for your node

rate = rospy.Rate(10)

The rate object is created here with a value of 10 Hertz. Rates are used to limit the frequency at which certain loops spin in ROS. Choosing a rate which is too high may result in unnecessarily high CPU usage, while choosing a value too low could result in high overall system latency. Choosing sensible values for all of the nodes in a ROS system is a bit of a fine-art.

start\_time = 0

while not start\_time:

start\_time = rospy.Time.now().to\_sec()

start\_time is used to determine how much time has elapsed. When using ROS with simulated time (as we are doing here), rospy.Time.now() will initially return 0, until the first message has been received on the /clock topic. This is why start\_time is set and polled continuously until a nonzero value is returned (more information [here](http://wiki.ros.org/rospy/Overview/Time)).

while not rospy.is\_shutdown():

elapsed = rospy.Time.now().to\_sec() - start\_time

pub\_j1.publish(math.sin(2\*math.pi\*0.1\*elapsed)\*(math.pi/2))

pub\_j2.publish(math.sin(2\*math.pi\*0.1\*elapsed)\*(math.pi/2))

rate.sleep()

This the main loop. Due to the call to rate.sleep(), the loop is traversed at approximately 10 Hertz. Each trip through the body of the loop will result in two joint command messages being published. The joint angles are sampled from a sine wave with a period of 10 seconds, and in magnitude from [−π/2,+π/2-\pi/2, +\pi/2−π/2,+π/2]. When the node receives the signal to shut down (either from the master, or via SIGINT signal in a console window), the loop will be exited.

if \_\_name\_\_ == '\_\_main\_\_':

try:

mover()

except rospy.ROSInterruptException:

pass

If the name variable is set to “**main**”, indicating that this script is being executed directly, the mover() function will be called. The try/except blocks here are significant as rospy uses exceptions extensively. The particular exception being caught here is the ROSInterruptException. This exception is raised when the node has been signaled for shutdown. If there was perhaps some sort of cleanup needing to be done before the node shuts down, it would be done here. More information about rospy exceptions can be found [here](http://wiki.ros.org/rospy/Overview/Exceptions).

## Running simple\_mover

Assuming that your workspace has recently been built, and it’s setup.bash has been sourced, you can launch simple\_arm as follows:

$ cd ~/catkin\_ws

$ roslaunch simple\_arm robot\_spawn.launch

Once ROS Master, Gazebo, and all of our relevant nodes are up and running, we can finally launch simple\_mover. To do so, open a new terminal and type the following commands:

$ cd ~/catkin\_ws

$ source devel/setup.bash

$ rosrun simple\_arm simple\_mover

1. ROS Services
2. Arm Mover
3. Arm Mover: The Code
4. Arm Mover: Launch and Interact
5. ROS Subscribers
6. Look Away
7. Look Away: The Code
8. Look Away: Launch and Interact
9. Logging
10. Recap
11. Outtro