1. Creating a workspace (https://wiki.ros.org/ROS/Tutorials/CreatingPackage):

In */home/pi* create a folder:

mkdir -p my_workspace/src

Using *catkin_create_pkg* we can create a new package and add any dependencies using the syntax:

catkin_create_pkg package_name dependency1 dependency2 …

For example:

catkin_create_pkg pkg_1 std_msgs rospy roscpp

This will create the following folder structure:

/home/pi:

- **my_workspace:**
    - **src:**
        - **pkg_1:**
            - **CMakeLists.txt**
            - **include**
            - **package.xml**
            - **src**

package.xml contains metadata about the package such as the package name, version, author details dependencies, and exports. The file can also be edited in any editor if we need to add more details.

Under the **src** folder we can place *cpp* files that will be compiled.

We will create the **scripts** folder for adding python scripts. In **/home/pi/workspace_1/src/pkg_1** execute:

mkdir scripts

To build the package, we will move to the root folder of the workspace (**cd /home/pi/workspace_1**) and execute:

catkin_make

This will add 2 folders: ***/home/pi/workspace_1/devel*** and **/home/pi/workspace_1/build**

To make the package detectable by ros, in folder **/home/pi/workspace_1/** run:

source devel/setup.bash

2. Creating your first ROS script

In **/home/pi/workspace_1/src/pkg_1/scripts** create a python file:

touch my_script.py

You can edit it using any text editor (**note: puppy pi only has vim and nano**):

Add the following code:

```
#!/usr/bin/env python3
import rospy
if __name__ == '__main__':
        print("Hello world")
```

- The first line is a directive (similar to *!/bin/bash* in bash scripts), it tells the operating system what interpreter should be used to run the file.
- The second line imports the ROS-specific functions available for python
- The third line checks if the script was launched for execution (in which case it will execute the following code) or just imported into another script (in which case the code within the if will not be executed)

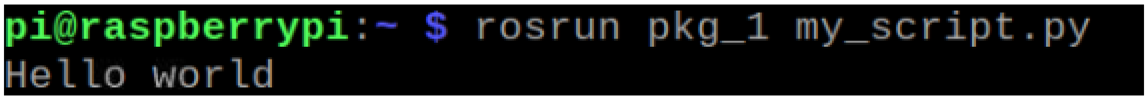
Finally, to allow the script to be run, it must have execution permissions:

chmod +x my_script.py

Running the script is done via the **rosrun** command:

rosrun pkg_1 my_script.py

This should print the text **"Hello world"** on the screen



```
pi@raspberrypi:~ $ rosrun pkg_1 my_script.py
Hello world
```

The advantage of using python scripts is that we do not need to rebuild the package if we add a new script or if we modify the existing ones. We will need, however, to rebuild the package if we add new dependencies, messages, services, or cpp files.

3. Create a ROS node from our script

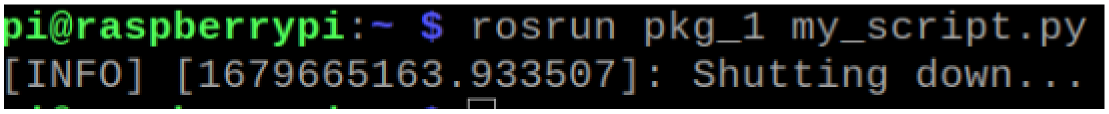
A ROS node is an executable that uses ROS to communicate with other nodes. We need to turn our script in a node before we can start interacting with other topics and services.

A node should have a name, should be initialized with **rospy.init_node** and it should be able to perform some cleanup on exit.

Thus, we turn our initial script in:

```
#!/usr/bin/env python3
import rospy
ROS_NODE_NAME = "my_ros_node" # global variable in case we will need it in multiple places
def cleanup():
    rospy.loginfo("Shutting down...") # the cleanup function currently doesn't have anything to do
if __name__ == '__main__':
    rospy.init_node(ROS_NODE_NAME, log_level=rospy.INFO) # init node
    rospy.on_shutdown(cleanup) # register the cleanup function on shutdown
```

Running the node with *rosrun* will produce the following result:



```
pi@raspberrypi:~ $ rosrun pkg_1 my_script.py
[INFO] [1679665163.933507]: Shutting down...
```

The node starts and very quickly terminates. If we want our node to persist, we have 2 options:

- *rospy.spin()* keeps the main thread alive until an exit is triggered (by an error, keyboard interrupt, exit);

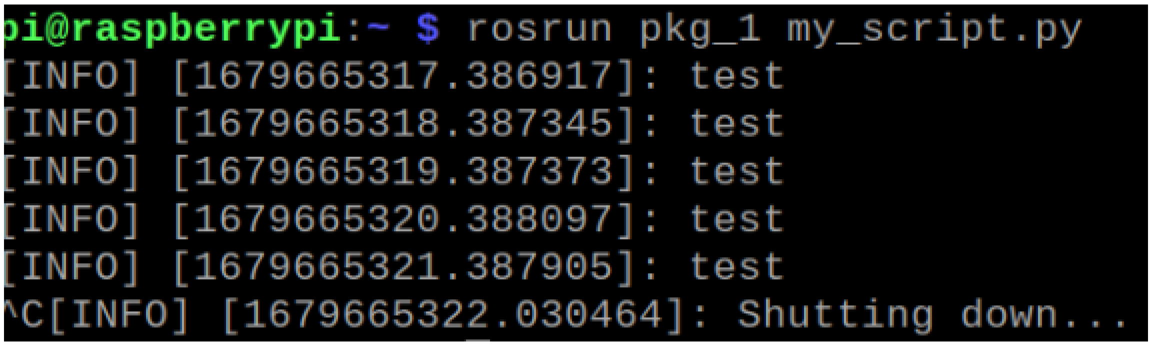
This is generally used if the node is listening on some topic and reacts to new data coming in

- *rospy.Rate* and *rate.sleep()* in conjunction with a loop in which we check if a shutdown was triggered;

This is used if the node is publishing data, Rate controls how often the node is pushing the data (measured in Hz).

```
#!/usr/bin/env python3
import rospy
ROS_NODE_NAME = “my_ros_node” # global variable in case we will need it in multiple places
def workFunction():
        rate = rospy.Rate(1) # this defines a rate of 1 Hz (once per second)
        while not rospy.is_shutdown():
                rospy.loginfo(“test”)
                rate.sleep() # sleeps enough time to ensure the frequency specified
def cleanup():
        rospy.loginfo("Shutting down...") # the cleanup function currently doesn’t have anything
to do
if __name__ == '__main__':
        rospy.init_node(ROS_NODE_NAME, log_level=rospy.INFO) # init node
        rospy.on_shutdown(cleanup) # register the cleanup function on shutdown
        try:
                workFunction()
        except KeyboardInterrupt:
                pass # on Ctrl+C, ROS will already intercept the signal and trigger a shutdown
```

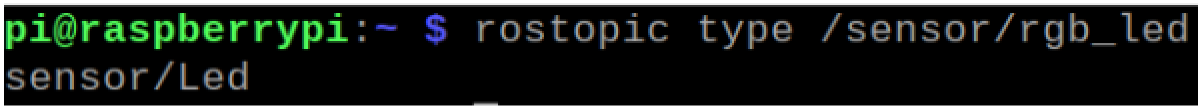
Running the node with rosrun will produce the following result:



```
pi@raspberrypi:~ $ rosrun pkg_1 my_script.py
[INFO] [1679665317.386917]: test
[INFO] [1679665318.387345]: test
[INFO] [1679665319.387373]: test
[INFO] [1679665320.388097]: test
[INFO] [1679665321.387905]: test
^C[INFO] [1679665322.030464]: Shutting down...
```

4. Extend the ROS node to interact with an existing topic (/sensor/led)

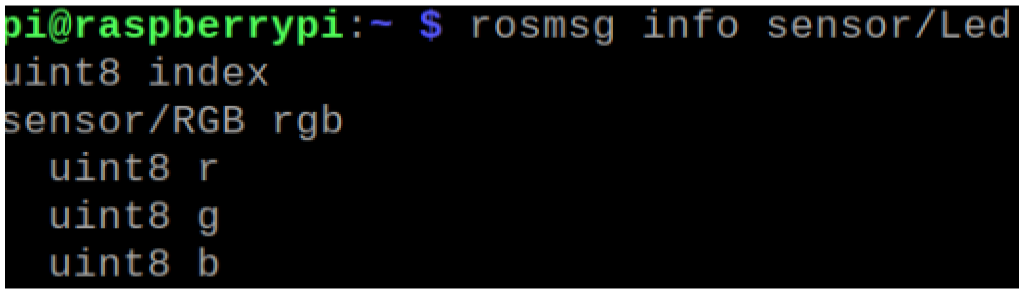
We want to send messages to an existing topic, so we need to identify the message type that is accepted on it. Run **rostopic type /sensor/led** to get the message type that we need to use:



```
pi@raspberrypi:~ $ rostopic type /sensor/rgb_led
sensor/Led
```

The message is **sensor/Led**, which tells us that it belongs to the package **sensor**.

To see the structure of the message, run **rosmsg info sensor/Led**.



```
pi@raspberrypi:~ $ rosmsg info sensor/Led
uint8 index
sensor/RGB rgb
  uint8 r
  uint8 g
  uint8 b
```

The steps we need to take to extend our ROS node are:

- import the necessary message types
- create a publisher for an existing topic
- build and send a message

Per the naming convention, we can note that the package containing the **sensor/Led** message is **sensor**.
To import messages from any package, the syntax is *from package.msg import message*
In our case:

from sensor.msg import Led

Defining a publisher on a topic is done using the rospy.Publisher class:

led_pub = rospy.Publisher(topic_name, message_type, queue_size)

- The topic_name is a string, as it is printed by rostopic (in our case **/sensor/rgb_led**)
- The message_type is a class, as imported in the script (in our case **Led**)
- The queue_size indicates how many messages should be kept if they are not consumed by anyone. If the queue is full, the oldest message in the queue is dropped to make space for a newer one.

To build a message we either use an empty constructor and populate each property separately or we pass all the properties as arguments to the constructor.
Sending the message is just one function call:

led_pub.publish(msg)

```
#!/usr/bin/env python3
import rospy
from sensor.msg import Led, RGB

ROS_NODE_NAME = "my_ros_node"
led_pub = None

def changeColor():
    global led_pub
    led_pub = rospy.Publisher("/sensor/rgb_led", Led, queue_size=1)
    colors = [(255, 0, 0), (255, 100, 0), (255, 255, 0), (0, 255, 0), (0, 0, 255), (75, 0, 130), (148, 0,
211)]
    size = len(colors)
    i = 0
    rate = rospy.Rate(1)
    led = Led()
    led.index = 0
    while not rospy.is_shutdown():
        led.rgb.r = colors[i][0]
        led.rgb.g = colors[i][1]
        led.rgb.b = colors[i][2]
        led_pub.publish(led)
        i = (i + 1) % size
        rate.sleep()

def cleanup():
    global led_pub
    led = Led(0, RGB(0, 0, 0))
    if led_pub != None:
        led_pub.publish(led)
    rospy.loginfo("Shutting down...")

if __name__ == '__main__':
    rospy.init_node(ROS_NODE_NAME, log_level=rospy.INFO)
    rospy.on_shutdown(cleanup)
    try:
        changeColor()
    except KeyboardInterrupt:
        pass
```