# How to create a ROS package (In Python) in ?? Easy Steps

## Workspace Setup

Make sure you have ~duckietown/catkin_ws setup and sourced

    laptop $ source ~/duckietown/environment.sh

(Refer to laptop setup from last lab)

## Where to put your package

Let's define the **<package_path>**. This depends on whether you are a core developer or a student completing a module

**Core development:**

packages should be placed in

```
<package_path> = $(DUCKIETOWN_ROOT)/src/catkin_ws/src
```

**Student modules**

packages should be placed in

```
<package_path> =
$(DUCKIETOWN_ROOT)/src/catkin_ws/src/spring2016/<handle>
```

and your package should be named with a name that ends in _**<handle>**

From this point forward we will refer to the name of your package as **<package_name>**.

**Explanation**: Things are setup this way so that when merges are performed all of the code lives on forever. Some of the code from student modules may be incorporated into the core development code by the development team.

## Package Creation

Use the catkin_create_pkg script to create your package. The syntax is:

catkin_create_pkg **<package_name>** dependency1 dependency2 …

Do

```
      laptop $ cd ~/duckietown/catkin_ws/src/
      laptop $ catkin_create_pkg <package_name> rospy roscpp
duckietown_msgs
```

You will see

```
      Created file <package_name>/CMakeLists.txt
      Created file <package_name>/package.xml
      Created folder <package_name>/include/pkg_your_handle
      Created folder <package_name>/src
      Successfully created files in <package_path>/<package_name>.
      Please adjust the values in package.xml.
```

Check that the package is actually created

```
      laptop $ ls
```

You should see at least this folder

```
      <package_name>
```

Let’s see what’s in that folder

```
      laptop $ cd <package_path>/<package_name>
      laptop $ ls
```

You should see

```
      CMakeLists.txt  include  package.xml  src
```

## Understanding CMakeList.txt

Yes, you still need to have a `CMakeLists.txt` file even if you are just using python code in your package. But don't worry, it's pretty straight-forward.

For a simple package, you only have to pay attention to the following parts.

You can open the file by nano (or you can use emacs, vim, or **sublime**)

```
      laptop $ nano CMakeLists.txt
```

Edit line 2 to be the name of your package:

```
project(<package_name>)
```

this defines the name of the project.

Line 7 to 10,

```
find_package(catkin REQUIRED COMPONENTS
  duckietown_msgs
  roscpp
  rospy
)
```

specifies the packages on which your package depends. In duckietown, most packages should depend on `duckietown_msgs` to make use of the customized messages.

Line 20,

```
# catkin_python_setup()
```

configure python related settings for this pkg. Uncomment this by removing the #. Save the File.

## Understanding package.xml

package.xml defines the meta data of the package. Catkin makes use of it to build the dependency tree to determine build order. Pay attention to the following parts.

Line 3:

```
<name><package_name></name>
```

defines the name of the pkg. It has to match the project name in `CMakeLists.txt`.

Line 5:

```
<description>The <package_name> package</description>
```

Describe the purpose and functionality of this pkg concisely.

Line 10:

```
<maintainer email="your_email">first_name last_name</maintainer>
```

Describe the maintainer. Put down your name and email.

Line 42 to Line 48

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>duckietown_msgs</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<run_depend>duckietown_msgs</run_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
```

The catkin packages this package depends on. These should match the `find_package` section in `CMakeLists.txt`.

Save and close the file.

## Creating setup.py

setup.py file help making your python modules in the include/pkg_your_handle folder availalbe to other packages in the workspace. By default this is not created by the catkin_create_pkg script. So let's create one using nano by:

```
laptop $ nano <package_path>/<package_name>/setup.py
```

Copy and paste the following to the setup.py file (to paste into a terminal, Ctrl+Shift+V)

```
## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD
from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup

# fetch values from package.xml
setup_args = generate_distutils_setup(
     packages=['<package_name>'],
     package_dir={'': 'include'},
)
setup(**setup_args)
```

The

```
packages = [...],
```

is set to a list of strings of the name of the folders inside the `include` folder. The convention is to set the folder name the same as the pkg name. Here it's the `include/**<package_name>**` folder.    This configures the **<package_name>**`/include/`**<package_name>** folder as a python module available to the whole workspace. You should put ROS-independent and/or reusable code module (for this, and other modules) in the `include/`**<package_name>** folder.

Save and close the file.s

For a folder to be treated as a python module, the __init__.py file must exist. Let's create one by

```
    laptop $ touch
<package_path>/<package_name>/include/<package_name>/__init__.py
```

## Write a python modulels

Now, let’s write a simple utility module in `/include/`**<package_name>**`/`:

```
laptop $ nano <package_path>/<package_name>/include/<package_name>/util.py
```

Type or copy and paste the following code:

```
import random
def getName():
    return "your_name"
def getStatus():
    return random.choice(["happy","awesome"])
```

The getName() returns “**your_name**”. The getStatus() returns “happy” or “awesome” randomly.

Save and close the file

## Make the module available to the workspace

You need to run catkin_make to make the module available to the whole workspace. (catkin_make invokes the setup.py. Do not invoke the setup.py by yourself.)

```
laptop $ cd ~/duckietown/catkin_ws
laptop $ catkin_make
```

After catkin_make, it’s usually a good idea to

```
laptop $ rospack profile
```

to reindex the packages so you can autocomplete packages related commands.

## Writing a node that make use of the module

Now let’s write a simple publisher node that says something interesting.

First let’s create and open a file in the <package_name>`/src` folder by

```
laptop $ nano <package_path><package_name>/src/publisher_node.py
```

Type or copy and paste the following code into that file:

```
#!/usr/bin/env python
import rospy
from <package_name> import util : from std_msgs.msg import String
# Initialize the node with rospy
```

```
    rospy.init_node('publisher_node')
    # Create publisher
    publisher = rospy.Publisher("~topic",String,queue_size=1)
    # Define Timer callback
    def callback(event):
        msg = String()
        msg.data = "%s is %s!" %(util.getName(),util.getStatus())
        publisher.publish(msg)
    # Read parameter
    pub_period = rospy.get_param("~pub_period",1.0)
    # Create timer
    rospy.Timer(rospy.Duration.from_sec(pub_period),callback)
    # spin to keep the script for exiting
    rospy.spin()
```

Note that the line:

```
    from <package_name> import util
```

imports the module defined in util.py, and the line

```
    msg.data = "%s is %s!" %(util.getName(),util.getStatus())
```

utilizes the functions `getName()` and `getStatus()` in that module.

Save and close the file.

To run a python script using rosrun, the script has to be executable. You can make publisher_node.py exectuable by:

```
    laptop $ chmod +x <package_path>/<package_name>/src/publisher_node.py
```

Now Let’s run our newly written module utilizing node!

First start a ros master by:

```
    laptop $ roscore
```

In a new terminal, run the script using rosrun

```
    laptop $ rosrun <package_name> publisher_node.py
```

Now in another new terminal, use rostopic echo to see what it’s sayin

```
    laptop $ rostopic echo /publisher_node/topic
```

It should say something like these:

data: **your_name** is happy

```
---
data: your_name is happy
---
data: your_name is awesome
---
data: your_name is happy!
---
data: your_name is happy!
---
data: your_name is awesome!
```

Kill the echo, the node, and the master when you’re done.

# Extra: Defining new messages in your package

If you are defining new message in your package, besides from putting .msg files into the msg folder, you also need to edit CMakeList.txt and package.xml for the messages to be compiled and made available to the workspace.

## CMakeLists.txt

**The find_package section:**
You need to add `message_generation` to the find_package list.

The add_message_files section.

```
add_message_files(
    FILES
    MessageName1.msg
    MessageName2.msg
)
```

You need to add name of the .msg files in the msg folder here.

**The generate_messages section:**

```
generate_messages(
    DEPENDENCIES
    std_msgs
)
```

If your message uses messages from other packages, such as std_msgs/Float32, you need to declare the dependency here. You should at least have std_msgs here.

**The catkin_package section**
You need to add message_runtime to the CATKIN_DEPENDS field of the catkin_package section. Also you should include all the msgs packages in the generate_messages section here too.

## package.xml

Add

```
        <build_depend>message_generation</build_depend>
        <run_depend>message_runtime</run_depend>
```

And also

```
        <build_depend>std_msgs</build_depend>
        <run_depend>std_msgs</run_depend>
```

For all the messages your customized messages depend on.

## Note on compiling messages

You will need to catkin_make if you add a new .msg or edit an existing .msg. If you get message related error during catkin_make, you might need to remove the /devel and /build folder under your catkin_ws and then reinvoke catkin_make.

# Old tutorial (

# Writing a node
Let's look at `src/talker.py` as an example. ROS nodes are put under the `src` folder and they have to be made executable to function properly. You can do so by`chmod +x talker.py`.

## Header
```python
#!/usr/bin/env python
import rospy
from pkg_name.util import HelloGoodbye #Imports module. Not limited to modules in this pkg.
from std_msgs.msg import String #Imports msg
```

`#!/usr/bin/env python`, this specify that the script is written in python. Every ROS node in python should start with this line (or else it won't work properly.)

`import rospy` imports the rospy module necessary for all ROS nodes in python.

`from pkg_name.util import HelloGoodbye` imports HelloGoodby defined in the file `pkg_name/include/pkg_name/util.py`. Note that you can also include modules provided by other pkgs giving that you specify dependency in `CMakeLists.txt` and `package.xml`.

`from std_msgs.msg import String` imports the `String` msg defined in the `std_msgs` pkg. Note that you can use `rosmsg show std_msgs/String `
in a terminal to lookup the definition of `String.msg`.

## Main
```python
if __name__ == '__main__':
    # Initialize the node with rospy
    rospy.init_node('talker', anonymous=False)

    # Create the NodeName object
    node = Talker()

    # Setup proper shutdown behavior
    rospy.on_shutdown(node.on_shutdown)

    # Keep it spinning to keep the node alive
    rospy.spin()
```

`rospy.init_node('talker', anonymous=False)` initialize a node named `talker`. Note that this name can be overwritten by a launch file. The launch file can also push this node down namespaces. If the `anonymous` argument is set to `True` then a random string of numbers will be append to the name of the node. Usually we don't use anonymous nodes.

`node = Talker()` creates an instance of the Talker object. More details in the next section.

`rospy.on_shutdown(node.on_shutdown)` ensures that the `node.on_shutdown` will be called when the node is shutdown.

`rospy.spin()` blocks to keep the script alive. This makes sure the node stays alive and all the publication/subscriptions work correctly.

## Talker
```python
class Talker(object):
    def __init__(self):
        # Save the name of the node
        self.node_name = rospy.get_name()

        rospy.loginfo("[%s] Initialzing." %(self.node_name))

        # Setup publishers
        self.pub_topic_a = rospy.Publisher("~topic_a",String, queue_size=1)
        # Setup subscriber
        self.sub_topic_b = rospy.Subscriber("~topic_b", String, self.cbTopic)
        # Read parameters
        self.pub_timestep = setupParameter("~pub_timestep",1.0)
        # Create a timer that calls the cbTimer function every 1.0 second
        self.timer = rospy.Timer(rospy.Duration.from_sec(self.pub_timestep),self.cbTimer)

        rospy.loginfo("[%s] Initialzed." %(self.node_name))

    def setupParameter(self,param_name,default_value):
        value = rospy.get_param(param_name,default_value)
        rospy.set_param(param_name,value) #Write to parameter server for transparancy
        rospy.loginfo("[%s] %s = %s " %(self.node_name,param_name,value))
        return value

    def cbTopic(self,msg):
        rospy.loginfo("[%s] %s" %(self.node_name,msg.data))

    def cbTimer(self,event):
        singer = HelloGoodbye()
        # Simulate hearing something
        msg = String()
        msg.data = singer.sing("duckietown")
        self.pub_topic_name.publish(msg)

    def on_shutdown(self):
        rospy.loginfo("[%s] Shutting down." %(self.node_name))
```
### constructor
```python
self.node_name = rospy.get_name()
```

saves the name of the node. Including the name of the node in printouts makes them more informative.

```python
rospy.loginfo("[%s] Initialzing." %(self.node_name))
```
prints to ROS info.

```python
self.pub_topic_a = rospy.Publisher("~topic_a",String, queue_size=1)
```
defines a publisher which publishes a `String` msg to the topic `~topic_a`. Note that the `~` in the name of topic under the namespace of the node. More specifically, this will actually publish to `talker/topic_a` instead of just `topic_a`. The `queue_size` is usually set to 1 on all publishers. For more details see [rospy overview: publisher and subscribers](http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers)

```python
self.sub_topic_b = rospy.Subscriber("~topic_b", String, self.cbTopic)
```
defines a subscriber which expects a `String` message and subscribes to `~topic_b`. The message will be handled by the `self.cbTopic` callback function. Note that similar to the publisher, the `~` in the topic name puts the topic under the namespace of the node. In this case the subscriber actually subscribes to the topic `talker/topic_b`.

It is strongly encouraged that a node always publishes and subscribes to topics under their `node_name` namespace. In other words, always put a `~` in front of the topic names when you define a publisher or a subscriber. They can be easily remapped in a launch file. This makes the node more modular and minimizes the possibility of confusion and naming conflicts. See the launch file section for how remapping works.

```python
self.pub_timestep = self.setupParameter("~pub_timestep",1.0)
```
Sets the value of self.pub_timestep to the value of the parameter `~pub_timestep`. If the parameter doesn't exist (not set in the launch file), then set it to the default value `1.0`. The `setupParameter` function also writes the final value to the parameter server. This means that you can `rosparam list` in a terminal to check the actual values of parameters being set.

```python
self.timer = rospy.Timer(rospy.Duration.from_sec(self.pub_timestep),self.cbTimer)
```
defines a timer that calls the `self.cbTimer` function every `self.pub_timestep` seconds.

### Timer callback
```python
def cbTimer(self,event):
    singer = HelloGoodbye()
    # Simulate hearing something
    msg = String()
    msg.data = singer.sing("duckietown")
    self.pub_topic_name.publish(msg)
```
Everytime the timer ticks, a message is generated and published.

### Subscriber callback
```python
def cbTopic(self,msg):
    rospy.loginfo("[%s] %s" %(self.node_name,msg.data))
```
Everytime a message is published to `~topic_b`, the `cbTopic` function is called. It simply prints the msg using `rospy.loginfo`.

# Launch File
You should always write a launch file to launch a node. It also serves as a documentation on the IOs of the node. Let's take a look at `launch/test.launch`.
```
<launch>
    <node name="talker" pkg="pkg_name" type="talker.py" output="screen">
        <!-- Setup parameters -->
        <param name="~pub_timestep" value="0.5"/>
        <!-- Remapping topics -->
        <remap from="~topic_b" to="~topic_a"/>
    </node>
</launch>
```
For the `<node>`, the `name` specify the name of the node, which overwrites `rospy.init_node()` in the `__main__` of `talker.py`. The `pkg` and `type` specify the pkg and the script of the node, in this case it's `talke.py`. Don't forget the .py in the end (and remember to make the file executable through chmod). The `output="screen"` direct all the rospy.loginfo to the screen, without this you won't see any printouts (useful when you want to suppress a node that's too talkative.)

The `<param>` can be used to set the parameters. Here we set the `~pub_timestep` to `0.5`. Note that in this case this sets the value of `talker/pub_timestep` to `0.5`.

The `<remap>` is used to remap the topic names. In this case we are replacing `~topic_b` with `~topic_a` so that the subscriber of the node actually listens to its own publisher. Replace the line with
```
<remap from="~topic_b" to="talker/topic_a"/>
```
will have the same effect. This is redundant in this case but very useful when you want to subscribe to a topic published by another node.

# Testing the node
First of all, you have to `catkin_make` the pkg even if it only uses python. `catkin` makes sure that the modules in the include folder and the messages are available to the whole workspace. You can do so by
```bash
$ cd ~/duckietown/catkin_ws
$ catkin_make
```

Ask ROS to reindex the packages so that you can auto-complete most things.
```bash
$ rospack profile
```

Now you can launch the node by the launch file.
```bash
$ roslaunch pkg_name test.launch
```
You should see something like this in the terminal
```
... logging to
/home/shihyuan/.ros/log/d4db7c80-b272-11e5-8800-5c514fb7f0ed/roslaunch-Wolverine-15961.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://Wolverine.local:33925/

SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.16
 * /talker/pub_timestep: 0.5

NODES
  /
    talker (pkg_name/talker.py)

auto-starting new master
process[master]: started with pid [15973]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to d4db7c80-b272-11e5-8800-5c514fb7f0ed
process[rosout-1]: started with pid [15986]
started core service [/rosout]
process[talker-2]: started with pid [15993]
[INFO] [WallTime: 1451864197.775356] [/talker] Initialzing.
[INFO] [WallTime: 1451864197.780158] [/talker] ~pub_timestep = 0.5
[INFO] [WallTime: 1451864197.780616] [/talker] Initialzed.
[INFO] [WallTime: 1451864198.281477] [/talker] Goodbye, duckietown.
[INFO] [WallTime: 1451864198.781445] [/talker] Hello, duckietown.
[INFO] [WallTime: 1451864199.281871] [/talker] Goodbye, duckietown.
[INFO] [WallTime: 1451864199.781486] [/talker] Hello, duckietown.
[INFO] [WallTime: 1451864200.281545] [/talker] Goodbye, duckietown.
[INFO] [WallTime: 1451864200.781453] [/talker] Goodbye, duckietown.
```

Open another terminal and

```
$ rostopic list
```

You should see

```
/rosout
/rosout_agg
/talker/topic_a
```

In the same terminal

```
$ rosparam list
```

You should see
`/talker/pub_timestep`

You can see the parameters and the values of the `talker` node with
```
$ rosparam get /talker
```

# Documentation
You should document the parameters and the publish/subscribe topic names of each node in your package. The user should not have to look at the source code to figure out how to use the nodes.

# Guidelines
* Make sure to put all topics (publish or subscribe) and parameters under the namespace of the node with `~`. This makes sure that the IO of the node is crystal clear.
* Always include the name of the node in the printouts.
* Always provide a launch file that includes all the parameters (using `<param>`) and topics (using `<remap>`) with each node.