Workspaces and using colcon to build packages

First, If you don't have colcon installed on your computers ROS system, do so with the following:

```
sudo apt install python3-colcon-common-extensions
```

Note: we are still using apt here, (all these notes are taken in reference to running on a linux machine, or wsl)

First things first — in ROS 2, a **workspace** is the core directory where you'll develop your entire robotic system. Think of it as the project folder that contains all the software packages (nodes, libraries, launch files, etc.) your robot needs to operate.

"A workspace is a directory containing ROS 2 packages. Before using ROS 2, it's necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2's packages available for you to use in that terminal." - more on that in the workflow...

ROS 2 workspaces follow a **standard structure**, which is important because many ROS 2 tools (like `colcon`, `rosdep`, and `ros2 launch`) rely on this consistency to locate and build your packages correctly. If the structure is incorrect or incomplete, your build or runtime tools may fail silently or throw confusing errors.

Best practice is to create a new directory for every new workspace.

1. Source ROS as the underlay

```
source /opt/ros/humble/setup.bash
```

2. Create a new directory WS

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

* Best practice is to create a new directory for every new workspace. The name doesn't matter, but it is helpful to have it indicate the purpose of the workspace.

A typical ROS 2 workspace contains three key directories:

```
my_ros2_ws/
├── src/      <-- Your packages go here
├── install/  <-- Populated after building; contains executables, configs
├── build/    <-- Temporary build files created by colcon
```

To clone packages into the source (src) folder from the examples on github:

```
cd ~/ros2_ws
git clone https://github.com/ros2/examples src/examples -b humble
```

Good practice to to check that you have all the dependencies for something you clone off github, to do this run the following from the ws (cd .. out of src if in it)

```
rosdep install -i --from-path src --rosdistro humble -y
```

To build your ROS 2 workspace, use `colcon build --symlink-install.` This command compiles all packages inside your `src/` folder and sets up the `install/` and `build/` directories. The `--symlink-install` flag tells `colcon` to create symbolic links to source files instead of copying them, which is especially useful during development — it lets you edit Python scripts, launch files, or configs and see the changes immediately without rebuilding. After building, always run `source install/setup.bash` to update your environment with the new packages.

*Think of **colcon** like a project manager on a construction site:* it reads the blueprints (your packages), figures out the correct build order based on dependencies, and then coordinates all the workers (compilers, scripts) to build your robot system efficiently. In ROS 2, `colcon` is the official build tool that replaces older systems like `catkin_make`, and it supports both CMake and Python packages. When you run `colcon build`, it scans the `src/` folder, builds each package in order, and outputs compiled binaries into the `install/` directory. It also handles logging, dependency tracking, and partial rebuilds intelligently. Combined with flags like `--symlink-install`, it makes developing and debugging ROS 2 systems much smoother.

ROS 2 supports multiple build types, like CMake for C++ and `ament_python` for Python. Since I'm focusing on **Python**, my builds will be using the `ament_python` build type, meaning packages will be interpreted (not compiled), and rely on Python's packaging system under the hood. Perhaps this is a bit slower, but for the purposes of learning it's what I'm sticking with.

Workflow for Creating a Python Package (more detail on this to come)

(Python-based ROS 2 package):

Navigate to your workspace's `src/` directory (best practice is to have packages in src):

```
cd ~/ros2_ws/src
```

1. **Create a new package with rclpy:**

```
ros2 pkg create my_python_pkg --build-type ament_python --dependencies
rclpy

''' build-type ament_python tells ROS that this is a Python package.
dependencies rclpy includes ROS 2's core Python client library (like rospy
in ROS 1)'''
```

Understand the package layout:

You'll get a structure like the following layout after creating a new package skeleton:

```
my_python_pkg/
├── my_python_pkg/          # Python module (must match the package name)
│   └── __init__.py
├── setup.py                # Defines how the package is installed
├── package.xml             # ROS 2 metadata and dependencies
└── resource/
    └── my_python_pkg       # Required empty marker file
```

2. **Add the Python nodes:**

   Inside the `my_python_pkg/` folder (the inner one), create a Python script like `talker.py` or `listener.py`. Don't forget the shebang (`#!/usr/bin/env python3`) and to make the script executable:

   (**shebang** is used at the beginning of a script file to specify the interpreter that should be used to execute the script. For bash scripts, the shebang line typically looks like #!/bin/bash) so above we are saying to run using python3 compiler!

```
chmod +x talker.py
```

The chmod command in Linux and Unix-like operating systems is used to change the access permissions to be executable (like a flag saying, “hey you can run me to do x”

3. **Update `setup.py (sorta optional part not in official docs)`:**

   Add your script to the `entry_points` so ROS 2 can run it with `ros2 run`. Example:

```
entry_points={
    'console_scripts': [
        'talker = my_python_pkg.talker:main',
    ],
},
```

Steps 4 and 5 are procedural and very important, they are expanded upon below this section in fine detail.

4. **Build your workspace (re-reference all the nodes ROS has to work with) :**

   Do this by going back to the root of your workspace and run:

```
cd ~/ros2_ws
colcon build --symlink-install
```

5. **Source the environment (tell your OS where you are working from, like how in volume one we always sourced ROS2 in each new shell (as an underlay):**
   Do this by running in a NEW SHELL (same shell as your build = very bad):

```
source install/setup.bash
```

6. **Test run the node in a ROS2 sourced shell:**

```
ros2 run my_python_pkg talker
```

Details of step 4 above, Building using colcon.

- *Colcon is like the construction site manager.* It looks over the building materials (your Python nodes) and the blueprints (your ROS 2 Humble environment), then organizes everything, deciding what gets built, in what order, and where it all goes. It ensures the right dependencies are included, sets up symbolic links if asked, and makes sure everything is ready to run. In this way, Colcon is the master organizer of your ROS 2 workspace, coordinating all parts of the system to work together seamlessly.

Once you've created or added a package to your workspace's `src/` folder, it's time to **build** everything so ROS 2 can recognize and use your nodes from the **root of your workspace.**

From the root, (for the running example its ~/ros2_ws) we call colcon to build. It automatically looks at our folders and either copies or symbolically links packages.

There are 2 options to build packages. When adding the flags --symlink-install you **don't** need to **rebuild for simple script edits** the changes are picked up automatically. This is because the flag asks colcon to build a symbolic link to the file, referencing it each time, rather than copying it over

```
colcon build
colcon build --symlink-install
```

When to Rebuild? You should re-run colcon build any time you:

- Add new packages,
- Edit setup.py, package.xml, or python source code,
- Change dependencies.

For Python packages with --symlink-install, you don't need to rebuild for simple script edits — the changes are picked up automatically.

Details of step 5 above, sourcing the overlay.

What does sourcing mean?

When you source a setup file (like install/setup.bash), you're telling your current terminal session:

“Hey there pal, load all the environment variables, paths, and package info from this ROS 2 workspace so I can use it.”

Without sourcing, ROS 2 won’t know where to find your packages, nodes, or commands.

Under and overlays of ROS:

In ROS 2, you can **layer** workspaces on top of each other, usually the layers follow:

- The **underlay** is usually your base ROS 2 installation (e.g. `/opt/ros/humble`).
- The **overlay** is your personal workspace (e.g. `~/ros2_ws`), which adds packages you built or cloned from github.

Correct Workflow of steps 4&5

Terminal 1 – Build Cleanly

```
cd ~/ros2_ws
colcon build --symlink-install
```

Terminal 2 – Run Your Code

```
source /opt/ros/humble/setup.bash      # source the underlay (ros2)
source ~/ros2_ws/install/setup.bash    # source the overlay (your ws)

ros2 run my_python_pkg my_node
```

Note that in Volume 1 we made it such that each new shell automatically sources ROS2 Humble each time it opens, meaning that you would only need to source the ws.

Each terminal has a clean, isolated context. **This helps prevent spooky scary bugs. Even the documentation doesn't get into how bad these errors can be if you dont source in a new shell, that is how you know they are seriously cursed.**

An in-depth look at Building Packages

The workflow above is a follow along for the general scheme we will use when writing ROS packages, but what does it all mean?

A package is an organizational unit for your ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package.

Package creation in ROS 2 uses *ament* as its build system and *colcon* as its build tool. (Seen in the previous workflow. As of now C and Python are supported package languages)

Makeup of a package:

- `package.xml` file containing meta information about the package
- `resource/<package_name>` marker file for the package
- `setup.cfg` is required when a package has executables, so `ros2 run` can find them
- `setup.py` containing instructions for how to install the package
- `<package_name>` - a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`

Which looks like the following structure in-shell

```
my_package/
      package.xml
      resource/my_package
      setup.cfg
      setup.py
      my_package/
```

Inside your workspace you can have as many packages as you want. (remember that best practice is to have all packages inside the source (src) folder. Each package is its one folder, within the src. What's nice is that all your packages do not have the same language, so of the two supported ones (python and c) you can have a mix of packages written in both languages. (You cannot nest packages however)

Once inside a ws navigate to the src folder, you can create a package with:

```
ros2 pkg create --build-type ament_python --license Apache-2.0
<package_name>
```

Note the Apache license, something I reference in my open science notes, and have on Laika code! Always good to license your work, even if its open

```
joshkraus@AX8Max:~/ros2_ws/src$ ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name my_node my_pa
ckage
going to create a new package
package name: my_package
destination directory: /home/joshkraus/ros2_ws/src
```

(Using --node-name defaults the node to the hello world example from the docs)

After you add a package, go back to the root (out of source) to build using colcon. Colcon is able to build all of your packages at once from the root!

```
colcon build
```

If you want to build just a sole package, you can run:

```
colcon build --packages-select <specific_package_to_build>
```

https://www.youtube.com/watch?v=iBGZ8LEvkCY

To use your new package and executable, first **open a new terminal and source your main ROS 2 installation.**
(Remember, cursed bugs will appear if you go and source the package you just built in the same directory)

cd back into your workspace in a new shell and then source using the following:

```
source install/local_setup.bash
```

When this returns you inr environment has sourced your packages, they are now running as the overlay on top of the ROS2 underlay.

```
joshkraus@AX8Max:~/ros2_ws/src/my_package$ ls
LICENSE  my_package  package.xml  resource  setup.cfg  setup.py  test
joshkraus@AX8Max:~/ros2_ws/src/my_package$
```

Opening up the package now we can see all the sections. Keep in mind that while parts are auto created they may still need some fill-in-the-blank work, like “who is the maintainer” or the details of your apache license.

Example on the next page of what package.xml (the metadata file) looks like:

```
  GNU nano 6.2                                                  package.
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.
<package format="3">
  <name>my_package</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="joshkraus@todo.todo">joshkraus</maintainer>
  <license>Apache-2.0</license>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

It's clear I need to add a package description, and my actual email address….

```
  GNU nano 6.2                                          package.xml *
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="
<package format="3">
  <name>my_package</name>
  <version>0.0.0</version>
  <description>This is a test package made while learning ros humble!</description>
  <maintainer email="jjk226@lehigh.edu">joshkraus</maintainer>
  <license>Apache-2.0</license> #you can edit this line too for any license!

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

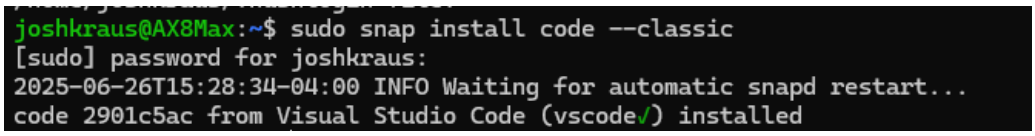
Other than package.xml, setup.py needs to be update to reflect similar info after build

Quick note. I have been wondering what the ~ (tilde) was used for when finding the workspace. It has to do with a little shorthand to get to your home directory without typing home/yourUser/. All systems shortcut the home folder with ~! That's also why going all the way "back" is cd ~ since you're going back to the home directory. The more you learn each day I guess. (And of course ros workspaces are created inside the home directory)

*****

Helpful when writing nodes and creating packages is an IDE like VS code:
Install VS code in the WSL using the command

```
sudo snap install code -classic
```



After running, you will see that VSCode is installed.

In some cases, WSL does not like running the linux version of VSCode - in this case, it may give you a warning to remove VS and install directly on your underlying device. You can remove that last snap download of VS by running:

```
sudo snap remove code
```

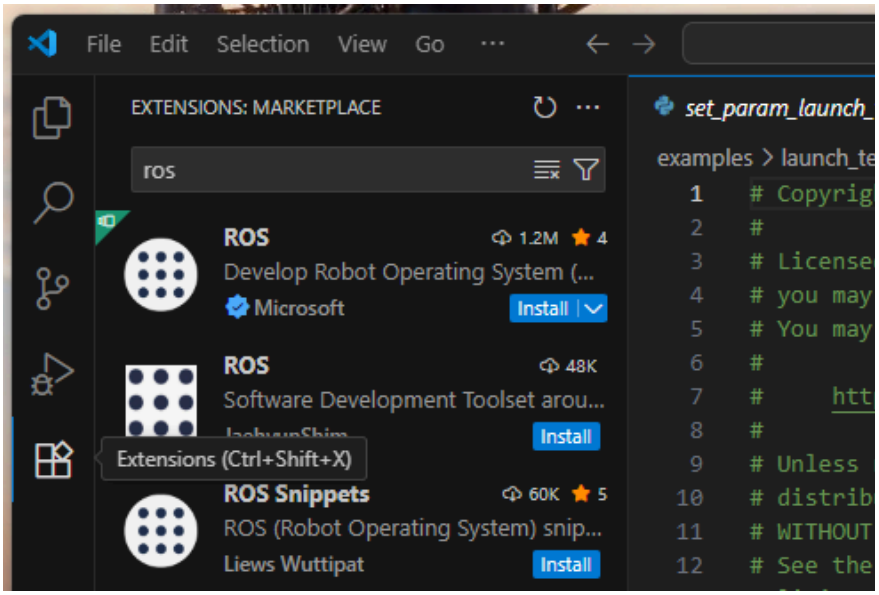
Re-open a *new* shell and all should be good to go. Click "trust author" as this is you.

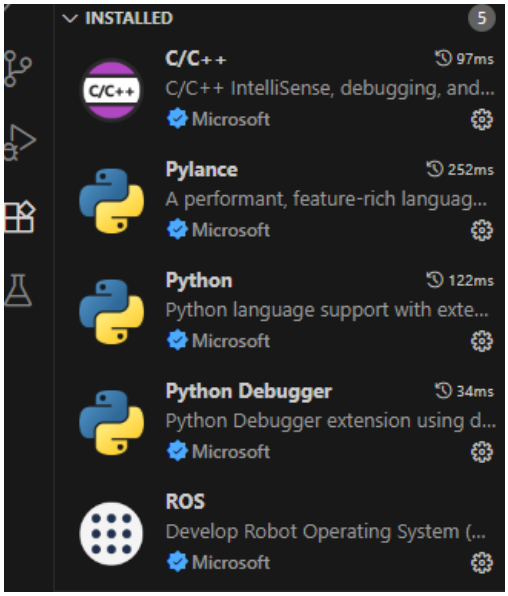
To open a folder in VS code, from within the folder run

```
code .
```

To get the full functionality of VS code we need to install ROS as an extension.

- Click the blocks to open extensions
- Download the Microsoft verified ROS (works with ROS1 and ROS2)
- When done this will install Python, C++ and ROS extensions





The main import used for ros is called rclpy. Now that ROS is installed in VS code, auto completion and user features can automatically complete and are recognized.

***

Below is the start of every python node.

- Line 1 tells the interpreter to use python (shebang of python3 since it's the newest)
- Line 2 we import the rclpy class provides the tools to initialize ROS 2, create nodes, publish/subscribe, etc

```
#!usr/bin/env python3
import rclpy
```

Below is the most basic structure of a python node program . This one does nothing but serves as a skeleton for how to write a node properly. Comments embedded.

```
#!usr/bin/env python3
import rclpy
from rclpy.node import Node #inheritable class


class MyNode(Node): #my own class (oriented programming) inherits all
functionalities

     def __init__(self): #constructor
        super().__init__("hello_world") # where you put the node name


        #here is where what the node DOES goes

def main(args=None):
    rclpy.init(args=args) #initializes ros2 communication (first line)
    node = MyNode() #node is now inside the main
    #how the node behaves is programmed here (eg staying alive or quitting after
running)


    rclpy.shutdown() #shuts down ros2 communication (last line)


if __name__ == "__main__":
    main()
```

To speak add: `self.get_logger().info("Hello from the Laika talker-node!")`
After super( )

Notes: hello_world is the name of the node in this file
I like to make the internal node name the same as the file name
In the next step i make this executable with the name hello_laika

Concept	Role in the Code
class MyNode(Node)	Object-oriented ROS 2 node
__init__()	Sets up the node's initial behavior
super().__init__()	Initializes base ROS 2 node and sets the name
main()	Central control function for your script
rclpy.init()	Starts the ROS 2 communication engine
rclpy.shutdown()	Gracefully shuts it down
__name__ == "__main__"	Makes the file safely runnable and importable

After creating a node (of which a more full node example will be walked through later in these notes) you need to *install* the node (best done in vs code still for step 2)

- Lets you run the node with ros2 run command
- Makes the node accessible for ros packages

1. First, inside the package terminal (twice there are 2 folders, package name and then package name again) make the python file executable with chmod

```
chmod +x <filename.py>
```

2. *Go to setup.py > find entry_points={, then inside console_scripts [*

```
setup.py > ...
1   setup.cfgfrom setuptools import find_packages, setup
2
3   package_name = 'my_package'
4
5   setup(
6       name=package_name,
7       version='0.0.0',
8       packages=find_packages(exclude=['test']),
9       data_files=[
10          ('share/ament_index/resource_index/packages',
11              ['resource/' + package_name]),
12          ('share/' + package_name, ['package.xml']),
13      ],
14      install_requires=['setuptools'],
15      zip_safe=True,
16      maintainer='joshkraus',
17      maintainer_email='jjk226@lehigh.edu',
18      description='TODO: test package from the ros2 humble docs',
19      license='Apache-2.0',
20      tests_require=['pytest'],
21      entry_points={
22          'console_scripts': [
23              'my_node = my_package.my_node:main'
24          ],
25      },
26  )
27
```

Inside console scripts we add ***"ros_executable_name == package_name.super_name:main"***

There are "QUOTES" here!

The structure from the example is shown is blown up as a skeleton below:

```
"<ros2_call_name> = <package_name.super_name>:main"
```

3. After installing the node program, go **back through the files to the main workspace** and use colcon to build.

```
Colcon build
```

Or for symbolic linking (remember auto updates)

```
Colcon build --symlink-install
```

4. AGAIN make sure to open a new terminal, go to the ws you have been working in and source with the command:

```
source install/setup.bash
```

At this point the node is now executable via `ros2 run pkg_name node_name`

If the package was a talker node, it will “talk” now!

```
joshkraus@AX8Max:~$ cd laika_testspace/
joshkraus@AX8Max:~/laika_testspace$ source install/setup.bash
joshkraus@AX8Max:~/laika_testspace$ ls
build  install  log  src
joshkraus@AX8Max:~/laika_testspace$ ros2 run talk_pkg hello_laika
[INFO] [1751310469.353263685] [hello_world]: Hello from the Laika talker-node!
```

Note: I have ROS auto sourcing each time I open a new shell as the underlay. You need to source your overlay each time! (this is why you see I “SOURCE INSTALL/SETUP.BASH”)

Use the tab key for auto completion as you start typing!

Listener Node:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):

        super().__init__("laika_listener")
        self.subscription = self.create_subscription(String, "topic",
self.listener_callback, 10) #note how its called a subscription not a subscriber
        self.subscription #prevents unused var alert
    '''No timer is needed since the subscriber simply respinses to publications,
so it is in a sense always on, and has no tick-rate'''

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data) #logger to express
data hearing on execution

def main (args=None):
    rclpy.init(args=args) #initialize ros comms
    node = MinimalSubscriber() #object orientyed node created
    rclpy.spin(node) #subs need to spin to collect info continuously with talker
    rclpy.shutdown()

if __name__ == "__main__":
    main()
```

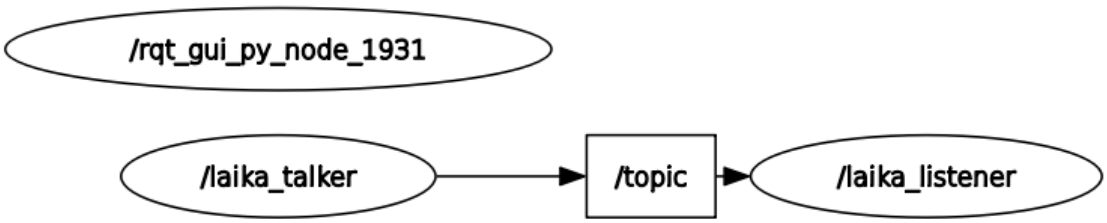
Talker node:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node #inheritable class
from std_msgs.msg import String #Publishing text to a topic class of strings
```

```
class MinimalPublisher(Node): #my own class (oriented programming) inherits all functionalities

    def __init__(self): #constructor
        super().__init__("laika_talker") # where you put the node name, name in ROS graph
        self.publisher = self.create_publisher(String, "topic", 10) #Creats itself and defines the topic to publish to as "topic"

        '''create_publisher declares that the node publishes messages of type String
        (imported from the std_msgs.msg module), over a topic named topic, and that the “queue size” is 10.'''

        timer_period = 0.5 # Sec.
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.count = 0

    '''timer_callback creates a message with the counter value appended'''

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello world from the Laika talking node! Itteration: %d' %self.count
        self.publisher.publish(msg)
        self.get_logger().info ('PUBLISHING: "%s"' %msg.data)
        self.count += 1


def main(args=None):
    rclpy.init(args=args) #initializes ros2 communication (first line)
    node = MinimalPublisher() #node is now inside the main
    rclpy.spin(node) #keeps node alive

    rclpy.shutdown() #shuts down ros2 communication (last line)


if __name__ == "__main__":
    main()
```

setup.py

```
from setuptools import find_packages, setup

package_name = 'talk_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='joshkraus',
    maintainer_email='jjk226@lehigh.edu',
    description='Simple test demonstration for project LAIKA',
    license='APACHE 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            "hello_laika = talk_pkg.hello_world:main",
            "listen_laika = talk_pkg.hello_listener:main"
        ],
    },
)
```



Note: Package.xml needs dependencies: `<depend>rclpy</depend>`

```
  <depend>std_msgs</depend>
```

Package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>talk_pkg</name>
  <version>0.0.0</version>
  <description>Simple test demonstration for project LAIKA</description>
  <maintainer email="jjk226@lehigh.edu">joshkraus</maintainer>
  <license>TODO: License declaration</license>

  <depend>rclpy</depend>
   <depend>std_msgs</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```