Using parameters in a class (Python)

Goal: Create and run a class with ROS parameters using Python.

Tutorial level: Beginner

Time: 20 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Create a package
  - 2 Write the Python node
  - 3 Build and run
- Summary
- Next steps

Background

When making your own nodes you will sometimes need to add parameters that can be set from the launch file.

This tutorial will show you how to create those parameters in a Python class, and how to set them in a launch file.

Prerequisites

In previous tutorials, you learned how to create a workspace and create a package. You have also learned about parameters and their function in a ROS 2 system.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that `ros2` commands will work.

Follow these instructions to create a new workspace named `ros2_ws`.

Recall that packages should be created in the `src` directory, not the root of the workspace. Navigate into `ros2_ws/src` and create a new package:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 python_parameters --dependencies
rclpy
```

Your terminal will return a message verifying the creation of your package `python_parameters` and all its necessary files and folders.

The `--dependencies` argument will automatically add the necessary dependency lines to `package.xml` and `CMakeLists.txt`.

1.1 Update `package.xml`

Because you used the `--dependencies` option during package creation, you don't have to manually add dependencies to `package.xml` or `CMakeLists.txt`.

As always, though, make sure to add the description, maintainer email and name, and license information to `package.xml`.

```
<description>Python parameter tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

2 Write the Python node

Inside the `ros2_ws/src/python_parameters/python_parameters` directory, create a new file called `python_parameters_node.py` and paste the following code within:

```
import rclpy
import rclpy.node

class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        self.declare_parameter('my_parameter', 'world')

        self.timer = self.create_timer(1, self.timer_callback)

    def timer_callback(self):
        my_param = self.get_parameter('my_parameter').get_parameter_value().string_value

        self.get_logger().info('Hello %s!' % my_param)

        my_new_param = rclpy.parameter.Parameter(
            'my_parameter',
            rclpy.Parameter.Type.STRING,
            'world'
        )
        all_new_parameters = [my_new_param]
        self.set_parameters(all_new_parameters)

def main():
    rclpy.init()
    node = MinimalParam()
    rclpy.spin(node)

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

2.1 Examine the code

The `import` statements at the top are used to import the package dependencies.

The next piece of code creates the class and the constructor. The line `self.declare_parameter('my_parameter', 'world')` of the constructor creates a parameter with the name `my_parameter` and a default value of `world`. The parameter type is inferred from the default value, so in this case it would be set to a string type. Next the `timer` is initialized with a period of 1, which causes the `timer_callback` function to be executed once a second.

```
class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        self.declare_parameter('my_parameter', 'world')

        self.timer = self.create_timer(1, self.timer_callback)
```

The first line of our `timer_callback` function gets the parameter `my_parameter` from the node, and stores it in `my_param`. Next the `get_logger` function ensures the event is logged. The `set_parameters` function then sets the parameter `my_parameter` back to the default string value `world`. In the case that the user changed the parameter externally, this ensures it is always reset back to the original.

```
def timer_callback(self):
    my_param = self.get_parameter('my_parameter').get_parameter_value().string_value

    self.get_logger().info('Hello %s!' % my_param)

    my_new_param = rclpy.parameter.Parameter(
        'my_parameter',
        rclpy.Parameter.Type.STRING,
        'world'
    )
    all_new_parameters = [my_new_param]
    self.set_parameters(all_new_parameters)
```

Following the `timer_callback` is our `main`. Here ROS 2 is initialized, an instance of the `MinimalParam` class is constructed, and `rclpy.spin` starts processing data from the node.

```
def main():
    rclpy.init()
    node = MinimalParam()
    rclpy.spin(node)

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

2.1.1 (Optional) Add ParameterDescriptor

Optionally, you can set a descriptor for the parameter. Descriptors allow you to specify a text description of the parameter and its constraints, like making it read-only, specifying a range, etc. For that to work, the `__init__` code has to be changed to:

```
# ...

class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        from rcl_interfaces.msg import ParameterDescriptor
        my_parameter_descriptor = ParameterDescriptor(description='This parameter is mine!')

        self.declare_parameter('my_parameter', 'world', my_parameter_descriptor)

        self.timer = self.create_timer(1, self.timer_callback)
```

The rest of the code remains the same. Once you run the node, you can then run `ros2 param describe /minimal_param_node my_parameter` to see the type and description.

2.2 Add an entry point

Open the `setup.py` file. Again, match the `maintainer`, `maintainer_email`, `description` and `license` fields to your `package.xml`:

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Python parameter tutorial',
license='Apache License 2.0',
```

Add the following line within the `console_scripts` brackets of the `entry_points` field:

```
entry_points={
    'console_scripts': [
        'minimal_param_node = python_parameters.python_parameters_node:main',
    ],
},
```

Don't forget to save.

3 Build and run

It's good practice to run `rosdep` in the root of your workspace (`ros2_ws`) to check for missing dependencies before building:

Linux | macOS | Windows

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

Navigate back to the root of your workspace, `ros2_ws`, and build your new package:

Linux | macOS | Windows

```
colcon build --packages-select python_parameters
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

Linux | macOS | Windows

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

Now run the node:

```
ros2 run python_parameters minimal_param_node
```

The terminal should return the following message every second:

```
[INFO] [parameter_node]: Hello world!
```

Now you can see the default value of your parameter, but you want to be able to set it yourself. There are two ways to accomplish this.

3.1 Change via the console

This part will use the knowledge you have gained from the tutoral about parameters and apply it to the node you have just created.

Make sure the node is running:

```
ros2 run python_parameters minimal_param_node
```

Open another terminal, source the setup files from inside `ros2_ws` again, and enter the following line:

```
ros2 param list
```

There you will see the custom parameter `my_parameter`. To change it, simply run the following line in the console:

```
ros2 param set /minimal_param_node my_parameter earth
```

You know it went well if you get the output `Set parameter successful`. If you look at the other terminal, you should see the output change to `[INFO] [minimal_param_node]: Hello earth!`

Since the node afterwards set the parameter back to `world`, further outputs show `[INFO] [minimal_param_node]: Hello world!`

3.2 Change via a launch file

You can also set parameters in a launch file, but first you will need to add a launch directory. Inside the `ros2_ws/src/python_parameters/` directory, create a new directory called `launch`. In there, create a new file called `python_parameters_launch.py`

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='python_parameters',
            executable='minimal_param_node',
            name='custom_minimal_param_node',
            output='screen',
            emulate_tty=True,
            parameters=[
                {'my_parameter': 'earth'}
            ]
        )
    ])
```

Here you can see that we set `my_parameter` to `earth` when we launch our node `parameter_node`. By adding the two lines below, we ensure our output is printed in our console.

```
output="screen",
emulate_tty=True,
```

Now open the `setup.py` file. Add the `import` statements to the top of the file, and the other new statement to the `data_files` parameter to include all launch files:

```
import os
from glob import glob
# ...

setup(
  # ...
  data_files=[
      # ...
      (os.path.join('share', package_name), glob('launch/*launch.[pxy][yma]*')),
    ]
  )
```

Open a console and navigate to the root of your workspace, `ros2_ws`, and build your new package:

Linux | macOS | Windows

```
colcon build --packages-select python_parameters
```

Then source the setup files in a new terminal:

Linux | macOS | Windows

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

Now run the node using the launch file we have just created:

```
ros2 launch python_parameters python_parameters_launch.py
```

The terminal should return the following message every second:

```
[INFO] [custom_minimal_param_node]: Hello earth!
```

Summary

You created a node with a custom parameter that can be set either from a launch file or the command line. You added the dependencies, executables, and a launch file to the package configuration files so that you could build and run them, and see the parameter in action.

Next steps

Now that you have some packages and ROS 2 systems of your own, the next tutorial will show you how to examine issues in your environment and systems in case you have problems.