ROS2 Tutorial

Release October 03, 2023

Murilo M. Marinho

Oct 03, 2023

PREAMBLE

Note: If you're looking for the official documentation, this is **NOT** it. For the official ROS documentation, refer to this link.

Hint: You can download this tutorial as a PDF .

About this tutorial

ROS2 Humble tutorials by Murilo M. Marinho, focusing on Ubuntu 22.04 x64 LTS and the programming practices of successful state-of-the-art robotics implementations such as the SmartArmStack and the AISciencePlatform.

Using this tutorial

This is a tutorial that supposes that the user will follow it linearly. Some readers can skip the *Preamble* if they are somewhat already comfortable in Python and Ubuntu. Otherwise, all steps can be considered as dependent on the prior ones, starting from *ROS2 Setup*.

Quick overview

1. ***Preamble: Ubuntu Basics***
   A few tips on Ubuntu/terminal usage.
2. ***Preamble: Python Basics***
   A quick memory refresher for the Python stuff we'll use in ROS2.
3. ***ROS2 Setup*** **(start here)**
   Installing ROS2 and setting up its environment for use.
4. ***ROS2 Python Package/Build Basics***
   Creating our first ROS2 package with **`ament_python`** and building it with **`colcon`**.
5. ***ROS2 Python Node Basics***
   Creating a `rclpy` Node and figuring out what all that means.
6. ***ROS2 Python Library Basics***
   Create a Python library and importing/using it in another **`ament_python`** package.
7. ***ROS2 Python Interface Basics***
   Making ROS2 messages, services, publishers, subscribers, service servers, and service clients.
8. ***ROS2 Parameter/Launch Basics***
   Making configurable ROS2 Nodes using parameters and launch files.

Note: This section is optional, the ROS2 tutorial starts at *ROS2 Installation*.

CHAPTER

ONE

UBUNTU TERMINAL BASICS

You already know how to turn on your computer and press some keys to make bits flip and colorful pixels shine on your monitor. Here, we'll go through a few tips on Ubuntu.

Note: The world is full of smart people, and they've done some amazing stuff, like Ubuntu and Linux. There are endless tutorials for those and this is not a complete one. In this section, we'll go through some basic tools available in Ubuntu's terminal that help with our quest to learn/use ROS2.

1.1 Who cares about the `terminal` anyways, are you like 100 years old or something?

Besides the unintended upside that if you're typing into a terminal fast enough with a black hoodie, you're cosplaying Mr. Robot at a very low cost, there wouldn't be another way to make a tutorial like this within the current age of the Universe without relying on Ubuntu's **`terminal`**.

GUIs (Graphical User Interfaces) change faster than long tutorials like this one can keep up with and **`terminal`** is our reliable partner in crime and unlikely to change much in the foreseeable future.

For the whole tutorial, you can copy and paste the commands in **`terminal`**. If it doesn't work, it's either your fault or mine, but surely not the **`terminal`**'s.

1.2 The `terminal`

Note: Check out Canonical's Tutorial on **`terminal`** for the complete story.

Hint: You can open a new terminal window by pressing `CTRL+ALT+T`.

Warning: This section is about the default terminal in Ubuntu 22.04. If you prefer to use some other terminal instead (there are many), then this might not be useful to you, and you might be happier referring to its documentation instead.

The **`terminal`** is one of those things with many names. Some call it **`shell`**, some **`console`**, some **`command line`**, some **`terminal`**. I'm sure there's someone furiously typing right now saying that I'm wrong and describing in detail what those differences might be. The truth is that, in the wild (a.k.a. the Internet), those terms are used pretty much as synonyms.

For all intents and purposes, Tom Hanks is not stuck in this terminal. Instead, we use it to send commands to Ubuntu and make stuff happen.

Table 1: (Murilo's) List of Useful Command Line Programs

Program	Example usage	What it does
pwd	`pwd`	Outputs the absolute path to the current directory.
mkdir	`mkdir a_folder`	**Ma**kes a **dir**ectory called `a_folder` in the current directory.
cd	`cd a_folder`	**C**hanges **d**irectory to a specified target.
touch	`touch a_file.whatever`	Creates an empty file called `a_file.whatever`.
cat	`cat a_file.whatever`	Outputs into the console the contents of `a_file.whatever`.
rm	`rm a_file.whatever`	**Rem**oves a file or directory (with the `-r` option).
ls	`ls`	**Li**sts the contents of the current directory.
grep	`cat a_file.whatever | grep robocop`	Outputs the lines of `a_file.whatever` that contain the string `robocop`.
nano	`nano a_file.whatever`	Helps you edit a file using a (relatively?) user-friendly program so that you don't get stuck into vim.
sudo	`sudo touch a_sudo_made_file.whatever`	With the powers of a **s**uper **u**ser, **do** something. It allows a given user to modify sensitive files in Ubuntu.
apt	`sudo apt install git`	Installs Ubuntu packages, in this case, **git**.
alias	`alias say_hello="echo hello"`	Creates an alias for a command, i.e. another way to refer to it.

1.3 Let's use it. (!?)

The thing is, we'll be using the terminal throughout the entire tutorial, so don't worry about going too deep right now.

To warm up, let's start by creating an empty file inside a new directory, as follows

Hint: The path ~ stands for the currently logged-in user's home folder.

Hint: You can open a new terminal window by pressing `CTRL+ALT+T`.

Warning: For copying from the terminal use `CTRL+SHIFT+C`. For pasting to the terminal, use `CTRL+SHIFT+V`. Be careful with `CTRL+C`, in particular. It is used to, in simple terms, close running programs on the terminal.

```
cd ~
mkdir a_folder
cd a_folder
touch an_empty_file.txt
```

Then, we can use **nano** to create another file with some contents

```
nano file_with_stuff.txt
```

Then, **nano** will run. At this point we can start typing, so let's just type

```
stuff
```

then you can exit with the following keys

1. `CTRL+X`
2. `Y`
3. `ENTER`

you can also look at the bottom side of the window to know what keys to press. As an example, in **nano**, `^X` stands for `CTRL+X`.

Then, if you run

```
ls
```

the output will be

```
an_empty_file.txt  file_with_stuff.txt
```

we can, for example, get the contents of `file_with_stuff.txt` with

```
cat file_with_stuff.txt
```

whose output will be

```
stuff
```

So, enough of this example, let's get rid of everything with

> **Warning:** **ALWAYS** be careful when using **rm**. The files removed this way do NOT go to the trash can, if you use it you pretty much said bye bye bye to those files/directories.

```
cd ~
rm -r a_folder
```

1.4 bash redirections

Hint: Before defaulting to writing a 300-lines-long Python script for the simplest and most common of tasks, it is always good to check if there is something already available in **bash** that can do the same thing in an easier and more stable way.

In a time long long ago, before ChatGPT became the new Deep Magic, **bash** was already tilting heads and leaving Ubuntu users in awe.

Among many powerful features, the *redirection operator*, >, stands out. It can be used to, unsurprisingly, *redirect* the output of a command to a file.

Warning: The operator > overwrites the target file with the output of the preceding command, it does not ask for permission, it just goes and does it.

The operator >> appends to the target file with the output of the preceding command.

Don't mix these up, there is no way to undo.

For example, if we want to store the result of the command `ls` to a file called `result_of_ls.txt`, the following will do

```
cd ~
ls > result_of_ls.txt
```

As a default in this version of Ubuntu, if the file does not exist it is created.

1.5 Tab completion

Hint: Use `TAB` completion extensively.

Whenever I have to look at a novice's shoulders while they interact with the terminal it gives me a certain level of anxiety. That is because they are trying to perfectly type even the longest and meanest paths for files, directories, and programs.

The terminal has `TAB` completion, so use it extensively. You can press `TAB` at any time to complete the name of a program, folder, file, or pretty much anything.

For example, we can move to a folder

```
cd ~
```

Then type a partial command or a part of its arguments. For example,

```
rm result_o
```

then, by pressing `TAB`, it should autocomplete to

```
rm result_of_ls.txt
```

1.6 Be careful with sudo

Warning: **DO NOT**, I repeat, **DO NOT** play around with **sudo**.

With great power, comes great opportunity to destroy your Ubuntu. It turns out that **sudo** is the master key of destruction, it will allow you to do basically anything in the system as far as the software is concerned.

So, don't.

For these tutorials, only use **sudo** when installing system-wide packages. Otherwise, do not use it.

1.7 Be careful even when not using `sudo`

With regular user privileges, the major system folders will be protected from tampering. However, our home folder, e.g. `/home/<YOU>` will not. In our home folder, we are the lords, so a mistake can be fatal for your files/directories.

1.8 File permissions

Warning: **DO NOT**, I repeat, **DO NOT** play around with **sudo**, **chmod**, or **chown**.

One of the reasons that using **sudo** indiscriminately will destroy your Ubuntu is file permissions. For example, if you *simply* open a file and save it as **sudo**, you'll change its permissions, and that might be enough to even block you from logging into Ubuntu via the GUI (Graphics User Interface).

I will not get into detail here about programs to change permissions because we won't need them extensively in these tutorials. However, it is important to be aware that this exists and might cause problems.

1.9 `nautilus`: browsing files with a GUI

To some extent similar to **explorer** in Windows and **finder** in macOS, **nautilus** is the default file manager in Ubuntu.

One tip is that it can be opened from the **terminal** as well, so that you don't have to find whatever folder you are again. For example,

Hint: The path `.` means the current folder.

```
cd ~
nautilus .
```

will open the currently logged-in user's home folder in **nautilus**.

Note: This section is optional, the ROS2 tutorial starts at *ROS2 Installation*.

CHAPTER

TWO

PYTHON BASICS

Note: This section is optional, the ROS2 tutorial starts at *ROS2 Installation*.

2.1 Installing Python on Ubuntu

Warning: If you change or try to tinker with the default Python version of Ubuntu, your system will most likely **BREAK COMPLETELY**. Do not play around with the default Python installation, because Ubuntu depends on it to work properly (or work at all).

In Ubuntu 22.04, Python is already installed! In fact, Ubuntu would not work without it. Let's check its version by running

```
python3 --version
```

which should output

```
Python 3.10.6
```

If the `3.10` part of your version is different (e.g. `3.9` or `3.11`), get this fixed because this tutorial will not work for you.

Warning: Note that the command is **`python3`** and not **`python`**. In fact, the result of

```
python
```

is

```
 Command 'python' not found, did you mean:
command 'python3' from deb python3
command 'python' from deb python-is-python3
```

2.1.1 A quick Python check

Run

```
python3
```

which should output something similar to

```
Python 3.10.6 (main, Mar 10 2023, 10:55:28) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

in particular, if the `GCC 11` is different, e.g. `GCC 9` or `GCC 12`, then get this fixed because this tutorial will not work for you.

As you already know, to exit the interative shell you can use `CTRL+D` or type `quit()` and press `ENTER`.

2.1.2 Some Python packages must be installed through `apt`

> **Warning:** Aside from these packages that you **MUST** install from **`apt`**, it is best to use **`venv`** and **`pip`** to install packages only for your user without using `sudo`.

For some Python packages to work well with the default Python in Ubuntu, they must be installed through **`apt`**. If you deviate from this, you can cause issues that might not be easy to recover from.

For the purposes of this tutorial, let us install `pip` and `venv`

```
sudo apt install -y python3-pip python3-venv
```

2.1.3 When you want to isolate your environment, use `venv`

> **Warning:** At the time of this writing, there was no support for **`venv`** on ROS2 (More info). Until that is handled, we are not going to use **`venv`** for the ROS2 tutorials. However, we will use **`venv`** to protect our ROS2 environment from these Python preamble tutorials.

Using **`venv`** (More info) is quite straightforward.

Create a `venv`

```
cd ~
python3 -m venv ros2tutorial_venv
```

where the only argument, `ros2tutorial_venv`, is the name of the folder in which the `venv` will be created.

Activate a venv

Whenever we want to use a venv, it must be explicitly activated.

```
cd ~
source ros2tutorial_venv/bin/activate
```

The terminal will change to have the prefix (ros2tutorial_venv) to let us know that we are using a venv, as follows

```
(ros2tutorial_venv) murilo@murilos-toaster:~$
```

Deactivate a venv

To deactivate, run

```
deactivate
```

We'll know that we're no longer using the ros2tutorial_venv because the prefix will disappear back to

```
murilo@murilos-toaster:~$
```

2.1.4 Installing libraries

Warning: In these tutorials, we rely either on **apt** or **pip** to install packages. There are other package managers for Python and plenty of other ways to install and manage packages. They are, in general, not compatible with each other so, like cleaning products, **DO NOT** mix them.

Hint: Using python3 -m pip instead of calling just pip allows more control over which version of **pip** is being called. The need for this becomes more evident when several Python versions have to coexist in a system.

As an example, let us install the best robot modeling and control library ever conceived, DQ Robotics.

First, we activate the virtual environment

```
cd ~
source ros2tutorial_venv/bin/activate
```

then, we install

```
python3 -m pip install dqrobotics
```

which will result in something similar to (might change depending on future versions)

```
Collecting dqrobotics
Downloading dqrobotics-23.4.0a15-cp310-cp310-manylinux1_x86_64.whl (551 kB)
     ---------------------------------------- 551.4/551.4 KB 6.3 MB/s eta 0:00:00
Collecting numpy
  Downloading numpy-1.25.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
↪(17.6 MB)
```

(continues on next page)

(continued from previous page)

```
     ---------------------------------------- 17.6/17.6 MB 7.4 MB/s eta 0:00:00
Installing collected packages: numpy, dqrobotics
Successfully installed dqrobotics-23.4.0a15 numpy-1.25.0
```

2.1.5 Removing libraries (installed with pip)

We can remove the library we just installed with

```
python3 -m pip uninstall dqrobotics
```

resulting in

```
Found existing installation: dqrobotics 23.4.0a15
Uninstalling dqrobotics-23.4.0a15:
  Would remove:
    /home/murilo/ros2tutorial_venv/lib/python3.10/site-packages/dqrobotics-23.4.0a15.
↪dist-info/*
    /home/murilo/ros2tutorial_venv/lib/python3.10/site-packages/dqrobotics/*
Proceed (Y/n)?
```

Hint: If in the terminal a question is made, the option with an uppercase letter, in this case `Y`, will be the default. If you want the default, just press `ENTER`.

Then, press `ENTER`, which results in

```
Successfully uninstalled dqrobotics-23.4.0a15
```

2.1.6 When using pip, do NOT use sudo

Using `sudo` without knowing what one is doing is *the* easiest way to wreak havoc in a Ubuntu installation. Even seemingly innocuous operations such as copying files with `sudo` can cause irreparable damage to your Ubuntu environment.

When installing Python packages that are not available on **apt**, use **pip**.

Note: This section is optional, the ROS2 tutorial starts at *ROS2 Installation*.

2.2 Editing Python source (with PyCharm)

There are near-infinite ways to manage your Python code and, for this tutorial, we will use **PyCharm**. Namely, the free community version.

2.2.1 Installing PyCharm

PyCharm is a great program for managing one's Python sources that is frequently updated and has a free edition. However, precisely because it is frequently updated, there is no way for this tutorial to keep up with future changes.

What we will do, instead, is to download a specific version of **PyCharm** for these tutorials, so that its behavior/looks/menus are predictable. If you'd prefer using the shiniest new version, be sure to wear sunglasses and not stare directly into the light.

Run

```
cd ~
mkdir ros2_workspace_pycharm
cd ros2_workspace_pycharm
wget https://download.jetbrains.com/python/pycharm-community-2023.1.1.tar.gz
tar -xzvf pycharm-community-2023.1.1.tar.gz
```

2.2.2 Create an alias for pycharm_ros2

To simplify the use of this version of **PyCharm**, let us create a **bash** alias for it.

```
echo "# Alias for PyCharm, as instructed in https://ros2-tutorial.readthedocs.io" >> ~/.
↪bashrc
echo "alias pycharm_ros2=~/ros2_workspace_pycharm/pycharm-community-2023.1.1/bin/pycharm.
↪sh" >> ~/.bashrc
source ~/.bashrc
```

Then, you can run **PyCharm** with

```
pycharm_ros2
```

Note: This section is optional, the ROS2 tutorial starts at *ROS2 Installation*.

2.3 (Murilo's) Python Best Practices

Warning: This tutorial expects prior knowledge in Python and object-oriented programming. As such, this section is not meant to be a comprehensive Python tutorial. You have better resources made by smarter people available online, e.g. The Python Tutorial.

2.3.1 Terminology

Let's go through the Python terminology used in this tutorial. This terminology is not necessarily uniform with other sources/tutorials you might find elsewhere. It is based on my interpretation of The Python Tutorial on Modules, the Python Glossary, and my own experience.

Table 1: (Murilo's) Python Glossary

Term	Book Definition	Use in the wild
script	A Python file that can be executed.	Any Python file *meant to be* executed.
module	A file with content that is meant to be imported by other modules and scripts.	This term is used very loosely and can basically mean any Python file, but usually a Python file *meant to be* imported from.
package	A collection of modules.	A folder with an `__init__.py`, even if it doesn't have more than one module. When people say Python Packaging it refers instead to making your package installable (e.g. with a `setup.py` or `pyproject.toml`), so be ready for that ambiguity.

2.3.2 Use a venv

We already know that it is a good practice to *When you want to isolate your environment, use venv*. So, let's turn that into a reflex and do so for this whole section.

```
cd ~
source ros2tutorial_venv/bin/activate
```

2.3.3 Minimalist package: something to start with

In this step, we'll work on these.

```
python/minimalist_package/
  └── minimalist_package/
        └── __init__.py
```

First, let's make a folder for our project

Hint: The `-p` option for **mkdir** creates all parent folders as well, when they do not exist.

```
mkdir -p ~/ros2_tutorials_preamble/python/minimalist_package
```

Then, let's create a folder with the same name within it for our package. A Python package is a folder that has an `__init__.py`, so for now we add an empty `__init__.py` by doing so

```
cd ~/ros2_tutorials_preamble/python/minimalist_package
mkdir minimalist_package
cd minimalist_package
touch __init__.py
```

The (empty) package is done!

Hint: In PyCharm, open the ~/ros2_tutorials_preamble/python/minimalist_package folder to correctly interact with this project.

Warning: It is confusing to have two nested folders with the same name. However, this is quite common and starts to make sense after getting used to it (it is also the norm in ROS2). The first folder is supposed to be how your file system sees your package, i.e. the *project* folder, and the other contains the actual Python package, with the __init__.py and other source code.

2.3.4 Minimalist script

In this step, we'll work on this.

```
python/minimalist_package/
  └── minimalist_package/
        └── __init__.py
        └── minimalist_script.py
```

Let's start with a minimalist script that prints a string periodically, as follows. Create a file in ~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package called minimalist_script.py with the following contents.

minimalist_script.py

```python
#!/bin/python3
import time


def main() -> None:
    """An example main() function that prints 'Howdy!' twice per second."""
    try:
        while True:
            print("Howdy!")
            time.sleep(0.5)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


if __name__ == "__main__":
    """When this module is run directly, it's __name__ property will be '__main__'."""
    main()
```

2.3.5 Running a Python script on the terminal

There are a few ways to run a script/module in the command line. Without worrying about file permissions, specifying that the file must be interpreted by Python (and which version of Python) is the most general way to run a script

```
cd ~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package
python3 minimalist_script.py
```

which will output

Hint: You can end the **minimalist_script.py** by pressing CTRL+C in the terminal in which it is running.

```
Howdy!
Howdy!
Howdy!
```

Another way to run a Python script is to execute it directly in the terminal. This can be done with

```
cd ~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package
./minimalist_script.py
```

which will result in

```
bash: ./minimalist_script.py: Permission denied
```

because our file does not have the permission to run as an executable. To give it that permission, we must run **ONCE**

```
cd ~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package
chmod +x minimalist_script.py
```

and now we can run it properly with

```
cd ~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package
./minimalist_script.py
```

resulting in

```
Howdy!
Howdy!
Howdy!
```

Note that for this second execution strategy to work, we **MUST** have the *#!*, called shebang, at the beginning of the first line. The path after the shebang specifies what program will be used to interpret that file. In general, differently from Windows, Ubuntu does not guess the file type by the extension when running it.

```
#!/bin/python3
```

If we remove the shebang line and try to execute the script, it will return the following errors, because Ubuntu doesn't know what to do with that file.

```
./minimalist_script.py: line 2: import: command not found
./minimalist_script.py: line 5: syntax error near unexpected token `('
./minimalist_script.py: line 5: `def main() -> None:'
```

2.3.6 When using `if __name__=="__main__":`, just call the real `main()`

There are multiple ways of running a Python script. In the one we just saw, the name of the module becomes `__main__`, but in others that does not happen, meaning that the `if` can be completely skipped. So, write the `main()` function of a script as something standalone and, in the condition, just call it and do nothing else, as shown below

```
if __name__ == "__main__":
    """When this module is run directly, it's __name__ property will be '__main__'."""
    main()
```

2.3.7 It's dangerous to go alone: Always wrap the contents of `main` function on a *try–except* block

It is good practice to wrap the contents of `main()` call in a `try--except` block with at least the `KeyboardInterrupt` clause. This allows the user to shutdown the module cleanly either through the terminal or through **PyCharm**. We have done so in the example as follows

```
def main() -> None:
    """An example main() function that prints 'Howdy!' twice per second."""
    try:
        while True:
            print("Howdy!")
            time.sleep(0.5)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)
```

This is of particular importance when hardware is used, otherwise, the connection with it might be left in an undefined state causing difficult-to-understand problems at best and physical harm at worst.

The `Exception` clause in our example is very broad, but a **MUST** in code that is still under development. Exceptions of all sorts can be generated when there is a communication error with the hardware, software (internet, etc), or other issues.

This broad `Exception` clause could be replaced for a less broad exception handling if that makes sense in a given application, but that is usually not necessary nor safe. When handling hardware, it is, in general, **IMPOSSIBLE** to test the code of all combinations of inputs and states. As they say,

> *Be wary, for overconfidence is a slow and insidious [source for terrible bugs and failed demos]*

Hint: Catching all `Exceptions` might make debugging more difficult in some cases. At your own risk, you can remove this clause temporarily when trying to debug a stubborn bug, at the risk of forgetting to put it back and ruining your hardware.

2.3.8 Minimalist class: Use classes profusely

In this step, we'll work on these.

```
python/minimalist_package/
  └── minimalist_package/
        └── __init__.py
        └── minimalist_script.py
        └── _minimalist_class.py
```

As you are familiar with object-oriented programming, you know that classes are central to this paradigm. As a memory refresher, let's make a class that honestly does nothing really useful but illustrates all the basic points in a Python class.

Create a file in `~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package` called `_minimalist_class.py` with the following contents.

`_minimalist_class.py`

```python
class MinimalistClass:
    """
    A minimalist class example with the most used elements.
    https://docs.python.org/3/tutorial/classes.html
    """
    # Attribute reference, accessed with MinimalistClass.attribute_reference
    attribute_reference: str = "Hello "

    def __init__(self,
                 attribute_arg: float = 10.0,
                 private_attribute_arg: float = 20.0):  # With a default value of 20.0
        """The __init__ works together with __new__ (not shown here) to
        construct a class. Loosely it is called the Python 'constructor' in
        some references, although it is officially an 'initializer' hence
        the name.
        https://docs.python.org/3/reference/datamodel.html#object.__init__
        It customizes an instance with input arguments.
        """
        # Attribute that can be accessed externally
        self.attribute: float = attribute_arg

        # Attribute that should not be accessed externally
        # a name prefixed with an underscore (e.g. _spam) should be treated
        # as a non-public part of the API (whether it is a function, a method or a data member).
        # It should be considered an implementation detail and subject to change without notice.
        self._private_attribute: float = private_attribute_arg

    def method(self) -> float:
        """Methods with 'self' should use at least one statement in which 'self' is required."""
        return self.attribute + self._private_attribute

    def set_private_attribute(self, private_attribute_arg: float) -> None:
```

(continues on next page)

(continued from previous page)

```python
        """If a private attribute should be writeable, define a setter."""
        self._private_attribute = private_attribute_arg

    def get_private_attribute(self) -> float:
        """If a private attribute should be readable, define a getter."""
        return self._private_attribute

    @staticmethod
    def static_method():
        """
        Methods that do not use the 'self' should be decorated with the @staticmethod.
        It will only have access to attribute references.
        https://docs.python.org/3.10/library/functions.html#staticmethod
        """
        return MinimalistClass.attribute_reference + "World!"
```

then, let's modify the `__init__.py` with the following contents

`__init__.py`

```python
"""
Having an __init__.py file within a directory turns it into a Python Package.
A package within a package is called a subpackage.
https://docs.python.org/3/tutorial/modules.html#packages
"""
from minimalist_package._minimalist_class import MinimalistClass
```

Note: When adding imports to the `__init__.py`, the folder that we use to open in Pycharm and that we call to execute the scripts is *extremely* relevant. When packages are deployed (e.g. in PyPI or ROS2), the "correct" way to import in `__init__.py` is to use `import <PACKAGE_NAME>.<THING_TO_IMPORT>`, which is why we're doing it this way.

Note: Relative imports such as `.<THING_TO_IMPORT>` might work in some cases, and that is fine. It is a supported and valid way to import. However, don't be surprised when it doesn't work in ROS2, PyPI packages, etc, and generates a lot of frustration.

2.3.9 Not a matter of taste: Code style

It might be parsing through jibber-jabber code in l__tcode lessons with weird C-pointer logic and nested dereference operators that gets you through the door into one of those fancy companies with no dress code and free snacks, perks that I'm totally not envious of one bit. In the ideal world, at least, writing easy-to-understand code with the proper style is what should keep you in that job.

So, always pay attention to the naming of classes (PascalCase), files and functions (snake_case), etc.

Thankfully, Python has a bunch of style rules builtin the language and PEP (Python Enhancement Proposal), such as PEP8. Take this time to read it and get inspired by The Zen of Python

> *Beautiful is better than ugly.*
> *Explicit is better than implicit.*
> *Simple is better than complex.*

> *Complex is better than complicated.*
> *Flat is better than nested.*
> *Sparse is better than dense.*
> *Readability counts.*
> *Special cases aren't special enough to break the rules.*
> *Although practicality beats purity.*
> *Errors should never pass silently.*
> *Unless explicitly silenced.*
> *In the face of ambiguity, refuse the temptation to guess.*
> *There should be one– and preferably only one –obvious way to do it.*
> *Although that way may not be obvious at first* *unless you're Dutch*.
> *Now is better than never.*
> *Although never is often better than *right* now.*
> *If the implementation is hard to explain, it's a bad idea.*
> *If the implementation is easy to explain, it may be a good idea.*
> *Namespaces are one honking great idea – let's do more of those!*

2.3.10 Take the (type) hint: Always use type hints

Note: For more info, check out the documentation on Python typing and the type hints cheat sheet

Before you flood my inbox with complaints, let me vent for you. A *preemptive* vent.

> *But, you know, one of the cool things in Python is that we don't have to explicitly type variables. Do you want to turn Python into C?? Why do you love C++ so much you unpythonic Python hater????*

The dynamic typing nature of Python is, no doubt, a strong point of the language. Note that adding type hints does not impede your code to be used with other types as arguments. Type hints are, to no one's surprise, hints to let users (and some automated tools) know what types your functions were made for, e.g. to allow your favorite IDE (Integrated Development Environment) to help you with code suggestions.

In these tutorials, we are not going to use any complex form of type hints. We're basically going to attain ourselves to the simplest two forms, the (attribute, argument, etc) type, and the return types.

For attributes we use `<attribute>: type`, as shown below

```
        self.attribute: float = attribute_arg
```

For method arguments we use `<argument>: <type>` and for return types we use `def <method>(<params>) -> <type>`, as shown below in our example

```
    def set_private_attribute(self, private_attribute_arg: float) -> None:
        """If a private attribute should be writeable, define a setter."""
        self._private_attribute = private_attribute_arg
```

2.3.11 Document your code with Docstrings

You do not need to document every single line you code, that would in fact be quite obnoxious

```
# c stores the sum of a and b
c = a + b

# d stores the square of c
d = c**2

# check if d is zero
if d == 0:
   # Print warning
   print("Warning")
```

But, on the other side of the coin, it doesn't take too long for us to forget what the parameters of a function mean. *Take the (type) hint: Always use type hints* helps a lot, but additional information is always welcome. If you get used to using docstrings for every new method, your programming will be better in general because documenting your code makes you think about it.

The example below shows a quick explanation of what the class does using a docstring

```
class MinimalistClass:
    """
    A minimalist class example with the most used elements.
    https://docs.python.org/3/tutorial/classes.html
    """
```

The PEP 257 talks about docstrings but does not define too much beyond saying that we should use it. My recommendation as of now would be the Sphinx markup, because of the many Python libraries using it for Sphinx documentation/tutorials like this one.

The sample code shown in this section has docstrings everywhere, but they are being used to explain the general usage of some Python syntax. When documenting your code, obviously, the documentation should be about what the method/class/attribute does.

Hint: Ideally, all documentation is perfect from the start. In reality, however, that rarely ever happens so some documentation is always better than none. My advice would be to write something as it goes and possibly adjust it to more stable or cleaner documentation when the need arises.

2.3.12 Unit tests: always test your code

Note: For a comprehensive tutorial on unit testing go through the unittest docs.

In this step, we'll work on these.

```
python/minimalist_package/
  └── minimalist_package/
        └── __init__.py
        └── minimalist_script.py
```

(continues on next page)

(continued from previous page)

```
        └── _minimalist_class.py
  └── test/
        └── test_minimalist_class.py
```

Unit testing is a flag that has been waved by programming enthusiasts and is often a good measurement of code maturity.

The elephant in the room is that writing unit tests is **boring**. Yes, we know, *very* boring.

Unit tests are boring because they are an *investment*. Unit testing won't necessarily make your code [. . .] better, faster, [. . .] *right now*. However, without tests, don't be surprised after some point if your implementations make you drown in tech debt. Dedicating a couple of minutes now to make a couple of tests when your codebase is still in its infancy makes it more manageable and less boresome.

Back to the example, a good practice is to create a folder name `test` at the same level as the packages to be tested, like so

```
cd ~/ros2_tutorials_preamble/python/minimalist_package
mkdir test
```

Then, we create a file named `test_minimalist_class.py` with the contents below in the `test` folder.

Note: The prefix `test_` is important as it is used by some frameworks to automatically discover tests. So it is better not to use that prefix if that file does not contain a unit test.

test_minimalist_class.py

```python
import unittest
from minimalist_package import MinimalistClass


class TestMinimalistClass(unittest.TestCase):
    """For each `TestCase`, we create a subclass of `unittest.TestCase`."""

    def setUp(self):
        self.minimalist_instance = MinimalistClass(attribute_arg=15.0,
                                                   private_attribute_arg=35.0)

    def test_attribute(self):
        self.assertEqual(self.minimalist_instance.attribute, 15.0)

    def test_private_attribute(self):
        self.assertEqual(self.minimalist_instance._private_attribute, 35.0)

    def test_method(self):
        self.assertEqual(self.minimalist_instance.method(), 15.0 + 35.0)

    def test_get_set_private_attribute(self):
        self.minimalist_instance.set_private_attribute(20.0)
        self.assertEqual(self.minimalist_instance.get_private_attribute(), 20.0)

    def test_static_method(self):
        self.assertEqual(MinimalistClass.static_method(), "Hello World!")
```

(continues on next page)

(continued from previous page)

```python


def main():
    unittest.main()
```

Running the tests

For a quick jolt of instant gratification, let's run the tests before we proceed with the explanation.

There are many ways to run tests written with `unittest`. The following will run all tests found in the folder `test`

```
cd ~/ros2_tutorials_preamble/python/minimalist_package
python -m unittest discover -v test
```

which will output

```
test_attribute (test_minimalist_class.TestMinimalistClass) ... ok
test_get_set_private_attribute (test_minimalist_class.TestMinimalistClass) ... ok
test_method (test_minimalist_class.TestMinimalistClass) ... ok
test_private_attribute (test_minimalist_class.TestMinimalistClass) ... ok
test_static_method (test_minimalist_class.TestMinimalistClass) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Yay! We've done it!

Start with use `unittest`

Note: ROS2 uses `pytest` as default, but that doesn't mean you also have to use it in every Python code you ever write.

There are many test frameworks for Python. Nonetheless, the unittest module is built into Python so, unless you have a very good reason not to use it, just [use] it.

We import the `unittest` module along with the class that we want to test, namely `MinimalistClass`.

```python
import unittest
from minimalist_package import MinimalistClass
```

Test them all

Note: Good unit tests will not only let you know when something broke but also *where* it broke. A failed test of a high-level function might not give you too much information, whereas a failed test of a lower-level (more fundamental) function will allow you to pinpoint the issue.

Unit tests are somewhat like insurance. The more coverage you have, the better. In this example, we test all the elements in the class. Each test will be based on one or more asserts. For more info check the unittest docs.

In a few words, we make a subclass of `unittest.TestCase` and create methods within it that test one part of the code, hence the name unit tests.

```
    def test_attribute(self):
        self.assertEqual(self.minimalist_instance.attribute, 15.0)

    def test_private_attribute(self):
        self.assertEqual(self.minimalist_instance._private_attribute, 35.0)

    def test_method(self):
        self.assertEqual(self.minimalist_instance.method(), 15.0 + 35.0)

    def test_get_set_private_attribute(self):
        self.minimalist_instance.set_private_attribute(20.0)
        self.assertEqual(self.minimalist_instance.get_private_attribute(), 20.0)

    def test_static_method(self):
        self.assertEqual(MinimalistClass.static_method(), "Hello World!")
```

If one of the `asserts` fails, then the related test will fail, and the test framework will let us know which one.

The test's main function

Generally, a test script based on *unittest* will have the following main function. It will run all available tests in our test class. For more info and alternatives check the unittest docs.

```
def main():
    unittest.main()
```

Note: This section is optional, the ROS2 tutorial starts at *ROS2 Installation*.

2.4 Python's `asyncio`

Note: Asynchronous code is not the same as code that runs in parallel, even more so in Python because of the GIL (Global Interpreter Lock) (More info). Basically, the `async` framework allows us to not waste time waiting for results that we don't know when will arrive. It either allows us to attach a `callback` for when the result is ready, or to run many service calls and `await` for them all, instead of running one at a time.

There are two main ways to interact with `async` code, the first being by `awaiting` the results or by handling those results through `callbacks`. Let's go through both of them with examples.

2.4.1 Use a venv

We already know that it is a good practice to *When you want to isolate your environment, use venv*. So, let's turn that into a reflex and do so for this whole section.

```
cd ~
source ros2tutorial_venv/bin/activate
```

2.4.2 Create the minimalist_async package

In this step, we'll work on these.

```
python/minimalist_package/minimalist_package/
  └── minimalist_async/
        └── __init__.py
```

As we learned in *Minimalist package: something to start with*, let's make a package called `minimalist_async`.

```
cd ~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package
mkdir minimalist_async
cd minimalist_async
```

we then create an `__init__.py` file with the following contents

`__init__.py`

```python
from minimalist_package.minimalist_async._unlikely_to_return import unlikely_to_return
```

2.4.3 Create the async function

In this step, we'll work on this.

```
python/minimalist_package/minimalist_package/
  └── minimalist_async/
        └── __init__.py
        └── _unlikely_to_return.py
```

Let's create a module called `_unlikely_to_return.py` to hold a function used for this example at the `~/ros2_tutorials_preamble/python/minimalist_package/minimalist_package/minimalist_async` folder with the following contents

`_unlikely_to_return.py`

```python
import asyncio
import random
from textwrap import dedent


async def unlikely_to_return(tag: str, likelihood: float = 0.1) -> float:
    """
    A function that is unlikely to return.
    :return: When it returns, the successful random roll as a float.
    """
    while True:
        a = random.uniform(0.0, 1.0)
        if a < likelihood:
            print(f"{tag} Done.")
            return a
        else:
            print(f"{tag} retry needed (roll = {a} > {likelihood})")
            await asyncio.sleep(0.1)
```

Because we're using `await` in the function, we start by defining an `async` function.

Hint: If the function/method uses `await` anywhere, it should be `async` (More info).

This function was thought this way to emulate, for example, us waiting for something external without actually having to. To do so, we add a `while True:` and return only with 10% chance. Instead of using a `time.sleep()` we use `await asyncio.sleep(0.1)` to unleash the power of `async`. The main difference is that `time.sleep()` is synchronous (blocking), meaning that the interpreter will be locked here until it finishes. With `await`, the interpreter is free to do other things and come back to this one later after the desired amount of time has elapsed.

The function by itself doesn't do much, so let's use it in another module.

2.4.4 Using `await`

TL;DR Using `await`

1. Run multiple `Tasks`.
2. Use `await` for them, **after they were executed**.

In this step, we'll work on this.

```
python/minimalist_package/minimalist_package/
  └── minimalist_async/
        └── __init__.py
        └── _unlikely_to_return.py
        └── async_await_example.py
```

Differently from synchronous programming, using `async` needs us to reflect on several tasks being executed at the same time(-ish). The main use case is for programs with multiple tasks that can run concurrently and, at some point, we need the result of those tasks to either end the program or further continue with other tasks.

The `await` strategy we're seeing now is suitable when either we need the results from all tasks before proceeding or when the order of results matters.

To illustrate this, let's make a file called `async_await_example.py` in `minimalist_async` with the following contents.

async_await_example.py

```python
import asyncio
from minimalist_package.minimalist_async import unlikely_to_return


async def async_main() -> None:
    tags: list[str] = ["task1", "task2"]
    tasks: list[asyncio.Task] = []

    # Start all tasks before awaiting them, otherwise the code
    # will not be concurrent.
    for task_tag in tags:
        task = asyncio.create_task(
            unlikely_to_return(tag=task_tag)
        )
        tasks.append(task)

    # Alternatively, use asyncio.gather()
    # At this point, the functions are already running concurrently. We are now (a)waiting for the
    # results, IN THE ORDER OF THE AWAIT, even if the other task ends first.
    print("Awaiting results...")
    for (tag, task) in zip(tags, tasks):
        result = await task
        print(f"The result of task={tag} was {result}.")


def main() -> None:
    try:
        asyncio.run(async_main())
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

We start by importing the `async` method we defined in the other module

```python
from minimalist_package.minimalist_async import unlikely_to_return
```

The function will be run by an instance of `asyncio.Task`. When the task is created, it is equivalent to calling the function and it starts running concurrently to the script that created the task. The example is a bit on the fancy side to make it easier to read and mantain, but the concept is simple. When using the `await` paradigm, focus on the following

1. Make the function it should run, like our `unlikely_to_return()`.

2. Run all concurrent tasks and keep a reference to them as `asyncio.Task`.
3. `await` on each `asyncio.Task`, in the order in which you want those results.

```
async def async_main() -> None:
    tags: list[str] = ["task1", "task2"]
    tasks: list[asyncio.Task] = []

    # Start all tasks before awaiting them, otherwise the code
    # will not be concurrent.
    for task_tag in tags:
        task = asyncio.create_task(
            unlikely_to_return(tag=task_tag)
        )
        tasks.append(task)

    # Alternatively, use asyncio.gather()
    # At this point, the functions are already running concurrently. We are now␣
↪(a)waiting for the
    # results, IN THE ORDER OF THE AWAIT, even if the other task ends first.
    print("Awaiting results...")
    for (tag, task) in zip(tags, tasks):
        result = await task
        print(f"The result of task={tag} was {result}.")
```

Ok, enough with the explanation, let's go to the endorphin rush of actually running the program with

```
cd ~/ros2_tutorials_preamble/python/minimalist_package/
python3 -m minimalist_package.minimalist_async.async_await_example
```

Which will result in something like shown below. The function is stochastic, so it might take more or less time to return and the order of the tasks ending might also be different.

However, in the `await` framework, the results will **ALWAYS** be processed in the order that was specified by the `await`, **EVEN WHEN THE OTHER TASK ENDS FIRST**, as in the example below. This is neither good nor bad, it will be proper for some cases and not proper for others.

We can also see that both tasks are running concurrently until `task2` finishes, then only `task1` is executed.

```
Awaiting results...
task1 retry needed (roll = 0.36896762068176037 > 0.1).
task2 retry needed (roll = 0.8429002838770375 > 0.1).
task1 retry needed (roll = 0.841018521652675 > 0.1).
task2 retry needed (roll = 0.1351152094825686 > 0.1).
task1 retry needed (roll = 0.9484654265361889 > 0.1).
task2 retry needed (roll = 0.3167046796566366 > 0.1).
task1 retry needed (roll = 0.7519672365071198 > 0.1).
task2 retry needed (roll = 0.38440407016827005 > 0.1).
task1 retry needed (roll = 0.23155484384953284 > 0.1).
task2 retry needed (roll = 0.6418306170261009 > 0.1).
task1 retry needed (roll = 0.532161975008607 > 0.1).
task2 Done.
task1 retry needed (roll = 0.448132225703992 > 0.1).
task1 retry needed (roll = 0.13504700640433664 > 0.1).
```

(continues on next page)

(continued from previous page)

```
task1 retry needed (roll = 0.7404815278498079 > 0.1).
task1 retry needed (roll = 0.9830081693068259 > 0.1).
task1 retry needed (roll = 0.4070546146764875 > 0.1).
task1 retry needed (roll = 0.7474267487174882 > 0.1).
task1 Done.
The result of task=task1 was 0.038934769861482144.
The result of task=task2 was 0.06380247590535493.

Process finished with exit code 0
```

Hooray! May there be concurrency!

2.4.5 Using callback

TL;DR Using callbacks

1. Run multiple Tasks.
2. Add a callback to handle the result **as soon as it is ready**.
3. Use await for each Task just so that the main loop does not return prematurely.

In this step, we'll work on this.

```
python/minimalist_package/minimalist_package/
  └── minimalist_async/
        └── __init__.py
        └── async_await_example.py
        └── async_callback_example.py
```

Differently from awaiting for each task and then processing their result, we can define callbacks in such a way that each result will be processed as they come. In that way, the results can be processed in an arbitrary order. Once again, this is inherently neither a good strategy nor a bad one. Some frameworks will work with callbacks, for example ROS1, ROS2, and Qt, but some others will prefer to use await.

Enough diplomacy, let's make a file called async_callback_example.py in minimalist_async with the following contents.

async_callback_example.py

```python
from functools import partial
import asyncio
from minimalist_package.minimalist_async import unlikely_to_return


def handle_return_callback(tag: str, future: asyncio.Future) -> None:
    """
    Callback example for asyncio.Future
    :param tag: An example parameter, in this case, a tag
    :param future: A asyncio.Future is expected to be the last parameter
```

(continues on next page)

(continued from previous page)

```python
    of the callback.
    :return: Nothing.
    """
    if future is not None and future.done():
        print(f"The result of task={tag} was {future.result()}.")
    else:
        print(f"Problem with task={tag}.")


async def async_main() -> None:
    tags: list[str] = ["task1", "task2"]
    tasks: list[asyncio.Task] = []

    # Start all tasks before adding the callback
    for task_tag in tags:
        task = asyncio.create_task(
            unlikely_to_return(tag=task_tag)
        )
        task.add_done_callback(
            partial(handle_return_callback, task_tag)
        )
        tasks.append(task)

    # Alternatively, use asyncio.gather()
    # At this point, the functions are already running concurrently. And the result will be processed
    # by the callback AS "SOON" AS THEY ARE AVAILABLE.
    print("Awaiting results...")
    for task in tasks:
        await task


def main() -> None:
    try:
        asyncio.run(async_main())
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

In the `callback` paradigm, besides the function that does the actual task, as in the prior example, we have to make a, to no one's surprise, callback function to process the results as they come.

We do so with

```python
def handle_return_callback(tag: str, future: asyncio.Future) -> None:
    """
    Callback example for asyncio.Future
    :param tag: An example parameter, in this case, a tag
```

(continues on next page)

(continued from previous page)

```
    :param future: A asyncio.Future is expected to be the last parameter
    of the callback.
    :return: Nothing.
    """
    if future is not None and future.done():
        print(f"The result of task={tag} was {future.result()}.")
    else:
        print(f"Problem with task={tag}.")
```

In this case, the `callback` must receive a `asyncio.Future` and process it. Test the future for `None` in case the task fails for any reason.

Aside from that, there are only two key differences with the `await` logic example we showed before,

1. The callback must be added with `task.add_done_callback()`, remember to use `partial()` if the callback has other parameters besides the `Future`
2. `await` for the tasks at the end, not because this script will process it (it will be processed as they come by its `callback`), but because otherwise the main script will return and (most likely) nothing will be done.

```
async def async_main() -> None:
    tags: list[str] = ["task1", "task2"]
    tasks: list[asyncio.Task] = []

    # Start all tasks before adding the callback
    for task_tag in tags:
        task = asyncio.create_task(
            unlikely_to_return(tag=task_tag)
        )
        task.add_done_callback(
            partial(handle_return_callback, task_tag)
        )
        tasks.append(task)

    # Alternatively, use asyncio.gather()
    # At this point, the functions are already running concurrently. And the result will␣
↪be processed
    # by the callback AS "SOON" AS THEY ARE AVAILABLE.
    print("Awaiting results...")
    for task in tasks:
        await task
```

But enough talk… Have at you! Let's run the code with

```
cd ~/ros2_tutorials_preamble/python/
python3 -m minimalist_package.minimalist_async.async_callback_example
```

Depending on our luck, we will have a very illustrative result like the one below. This example shows that, with the `callback` logic, when the second task ends before the first one, it will be automatically processed by its `callback`.

```
Awaiting results...
task1 retry needed (roll = 0.6248308966234916 > 0.1).
task2 retry needed (roll = 0.2425971403299036 > 0.1).
task1 retry needed (roll = 0.1996764883575476 > 0.1).
```

(continues on next page)

(continued from previous page)

```
task2 Done.
The result of task=task2 was 0.09069407383542283.
task1 retry needed (roll = 0.6700777523785147 > 0.1).
task1 retry needed (roll = 0.7344216907108979 > 0.1).
task1 retry needed (roll = 0.4907223062034761 > 0.1).
task1 retry needed (roll = 0.20026037098687932 > 0.1).
task1 Done.
The result of task=task1 was 0.09676678954317675.
```

Can you feel the new synaptic connections?

Note: This section is optional, the ROS2 tutorial starts at *ROS2 Installation*.

2.5 Making your Python package installable

Warning: There is some movement towards having Python deployable packages configurable with `pyproject.toml` as a default. However, in ROS2 and many other frameworks, the `setup.py` approach using setuptools is ingrained. So, we'll do that for these tutorials but it doesn't necessarily mean it's the best approach.

2.5.1 Use a venv

We already know that it is a good practice to *When you want to isolate your environment, use venv*. So, let's turn that into a reflex and do so for this whole section.

```
cd ~
source ros2tutorial_venv/bin/activate
```

2.5.2 The setup.py

In this step, we'll work on this.

```
python/minimalist_package/
  setup.py
```

Has Python Packaging ever looked daunting to you? Of course not, but let's go through a quick overview of how we can get this done.

First, we create a `setup.py` at `~/ros2_tutorials_preamble/python/minimalist_package` with the following contents

`setup.py`

```
from setuptools import setup, find_packages

package_name = 'minimalist_package'
```

(continues on next page)

(continued from previous page)

```
setup(
    name=package_name,
    version='23.6.0',
    packages=find_packages(exclude=['test']),
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='Murilo M. Marinho',
    maintainer_email='murilomarinho@ieee.org',
    description='A minimalist package',
    license='MIT',
    entry_points={
        'console_scripts': [
            'minimalist_script = minimalist_package.minimalist_script:main',
            'async_await_example = minimalist_package.minimalist_async.async_await_
↪example:main',
            'async_callback_example = minimalist_package.minimalist_async.async_callback_
↪example:main'
        ],
    },
)
```

Note: By no coincidence, the `setup.py` is a Python file. We use Python to interpret it, meaning that we can process information using Python to define the arguments for the `setup()` function.

All arguments defined above are quite self-explanatory and are passed to the `setup()` function available at the `setuptools` module built into Python.

The probably most unusual part of it is the `entry_points` dictionary. In the key `console_scripts`, we can list up scripts in the package that can be used as console programs after the package is installed. Indeed, `setuptools` is rich, has a castle, and can do magic.

2.5.3 Installing `wheel`

Warning: In the current version of Python, if you do not install `wheel` as described herein, the following warning will be output.

```
DEPRECATION: minimalist-package is being installed using the legacy 'setup.py install
↪' method because it does not have a 'pyproject.toml'
and the 'wheel' package is not installed. pip 23.1 will enforce this behaviour change.
↪ A possible replacement is to enable the '--use-pep517'
option. Discussion can be found at https://github.com/pypa/pip/issues/8559
```

To install the package in the recommended way in this tutorial, we need `wheel`. While using the `venv`, we install it

```
python3 -m pip install wheel
```

2.5.4 Installing the Python package

We first go to the folder containing our *project* folder and we build and install the *project* folder within it using **pip** as follows

```
cd ~/ros2_tutorials_preamble/python
python3 -m pip install ./minimalist_package
```

which results in

```
Processing ./minimalist_package
  Preparing metadata (setup.py) ... done
Requirement already satisfied: setuptools in ~ros2tutorial_venv/lib/python3.10/site-
↪packages (from minimalist-package==23.6.0) (65.6.3)
Building wheels for collected packages: minimalist-package
  Building wheel for minimalist-package (setup.py) ... done
  Created wheel for minimalist-package: filename=minimalist_package-23.6.0-py3-none-any.
↪whl size=8608 sha256=929446a2fa81fc99fc5dec239a9f3e4439bc8fa8fe49cc4deb987d6f31b3d8b9
  Stored in directory: /private/var/folders/4k/20khytt17blf21lptscczbl00000gn/T/pip-
↪ephem-wheel-cache-j3a0f5xy/wheels/00/16/ef/
↪863b898c6ea4d32d47a24fda31f80cbc9cb1063742032b7d49
Successfully built minimalist-package
Installing collected packages: minimalist-package
Successfully installed minimalist-package-23.6.0
```

Done!

2.5.5 Running the newly available scripts

After installing, we have access to the scripts (and packages). For instance, we can do

```
minimalist_script
```

which will return the friendly

```
Howdy!
Howdy!
Howdy!
```

The other two scripts are also available, for instance, we can do

```
async_await_example
```

which will return something similar to

```
Awaiting results...
task1 retry needed (roll = 0.1534174185325745 > 0.1).
task2 retry needed (roll = 0.35338687437350913 > 0.1).
task1 Done.
task2 retry needed (roll = 0.3877920607121429 > 0.1).
The result of task=task1 was 0.07646509818952207.
task2 retry needed (roll = 0.7010015915930288 > 0.1).
task2 retry needed (roll = 0.8907576123834621 > 0.1).
```

(continues on next page)

(continued from previous page)

```
task2 retry needed (roll = 0.4233577578392548 > 0.1).
task2 retry needed (roll = 0.7512028176843422 > 0.1).
task2 retry needed (roll = 0.33501957024540663 > 0.1).
task2 Done.
The result of task=task2 was 0.09239734738421612.
```

2.5.6 Importing things from the installed package

We first run an interactive session with

```
python3
```

we can then interact with it as any other installed package

```
>>> from minimalist_package import MinimalistClass
>>> mc = MinimalistClass()
>>> print(mc.get_private_attribute())
20.0
```

Hooray!

2.5.7 Uninstalling packages

Given that we installed it using **pip**, removing it is also a breeze. We do

```
python3 -m pip uninstall minimalist_package
```

which will return something similar to

```
Found existing installation: minimalist-package 23.6.0
Uninstalling minimalist-package-23.6.0:
  Would remove:
    /home/murilo/ros2tutorial_venv/bin/async_await_example
    /home/murilo/ros2tutorial_venv/bin/async_callback_example
    /home/murilo/ros2tutorial_venv/bin/minimalist_script
    /home/murilo/ros2tutorial_venv/lib/python3.10/site-packages/minimalist_package-23.6.
↪0.dist-info/*
    /home/murilo/ros2tutorial_venv/lib/python3.10/site-packages/minimalist_package/*
Proceed (Y/n)?
```

and just press `ENTER`, resulting in the package being uninstalled

```
Successfully uninstalled minimalist-package-23.6.0
```

CHAPTER

THREE

ROS2 INSTALLATION

Note: This tutorial is an abridged version of the original ROS 2 Documentation. This tutorial considers a fresh Ubuntu Desktop (not Server) 22.04 LTS x64 (not arm64) installation, that you have super user access and common sense. It might work in other cases, but those have not been tested in this tutorial.

Warning: All commands must be followed to the letter, in the precise order described herein. Any deviation from what is described might cause unspecified problems and not all of them are easily solvable.

3.1 Update `apt` packages

Hint: You can quickly open a new terminal window by pressing `CTRL+ATL+T`.

After a fresh install, update and upgrade all **`apt`** packages.

```
sudo apt update && sudo apt upgrade -y
```

3.2 Install a few pre-requisites

```
sudo apt install -y software-properties-common curl terminator git
```

Namely:

software-properties-common	Allows us to access the ROS2 packages using **`apt`**.
curl	Helps download installation/configuration files from the terminal.
terminator	ROS uses plenty of terminals, so this helps keep one's sanity intact by enabling the management of several terminals in a single window. Despite what some might say, this particular terminator has no interest whatsoever in Sarah Connor.
git	The trendy source control program everyone mentions in their CV. You might be interested in knowing why it's called `git`.

3.3 Add ROS2 sources

Your **apt** needs to know where the ROS2 packages can be found and to be able to verify their authenticity. After setting up the **apt** sources, the local package list must be updated. The following commands will do all that magic.

```
sudo add-apt-repository universe
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/
↪share/keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-
↪keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_
↪CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
sudo apt update && sudo apt upgrade -y
```

3.4 Install ROS2 packages

There are plenty of ways to install ROS2, the following will suffice for now.

```
sudo apt install -y ros-humble-desktop ros-dev-tools
```

3.5 Set up system environment to find ROS2

ROS2 packages are implemented in such a way that they live peacefully in the `/opt/ros/{ROS_DISTRO}` folder in your Ubuntu. A given terminal window or program will only know that ROS2 exists, and which version you want to use, if you run a setup file *for each terminal, every time you open a new one*.

The `~/.bashrc` file can be used for that exact purpose as, in Ubuntu, that is the file that configures each terminal window for a given user.

TL;DR just run this **ONCE AND ONLY ONCE**

```
echo "# Source ROS2 Humble, as instructed in https://ros2-tutorial.readthedocs.io" >> ~/.
↪bashrc
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

3.6 Check if it works

If the following command

```
ros2
```

outputs something similar to what is shown below, then it worked! Otherwise, it didn't!

```
   usage: ros2 [-h] [--use-python-default-buffering]
            Call `ros2 <command> -h` for more detailed usage. ...

ros2 is an extensible command-line tool for ROS 2.

```

(continues on next page)

(continued from previous page)

```
options:
  -h, --help            show this help message and exit
  --use-python-default-buffering
                        Do not force line buffering in stdout and instead use
                        the python default buffering, which might be affected
                        by PYTHONUNBUFFERED/-u and depends on whatever stdout
                        is interactive or not

Commands:
  action     Various action related sub-commands
  bag        Various rosbag related sub-commands
  component  Various component related sub-commands
  daemon     Various daemon related sub-commands
  doctor     Check ROS setup and other potential issues
  interface  Show information about ROS interfaces
  launch     Run a launch file
  lifecycle  Various lifecycle related sub-commands
  multicast  Various multicast related sub-commands
  node       Various node related sub-commands
  param      Various param related sub-commands
  pkg        Various package related sub-commands
  run        Run a package specific executable
  security   Various security related sub-commands
  service    Various service related sub-commands
  topic      Various topic related sub-commands
  wtf        Use `wtf` as alias to `doctor`

  Call `ros2 <command> -h` for more detailed usage.
```

CHAPTER

FOUR

TERMINATOR IS LIFE

Note: You can refer to the project's documentation for more info.

After installing **terminator** as instructed in the last section, the default terminal window will be automatically updated to use it.

4.1 Shortcuts

To prevent repetition, let's go through the most relevant terminator shortcuts only once, here, now.

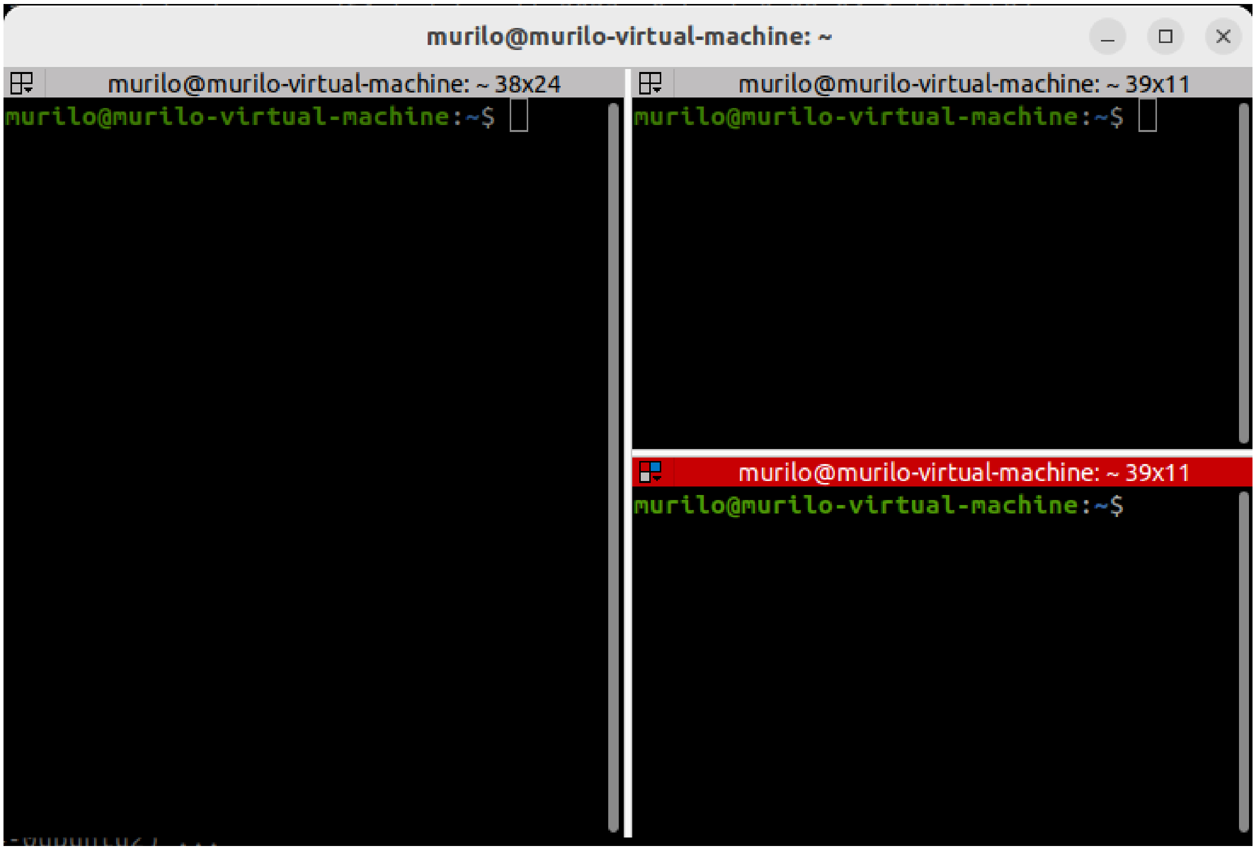
Table 1: Terminator Shortcuts

Shortcut	Description
CTRL+ALT+T	Open a new terminal window using your default viewer.
SHIFT+CTRL+E	**Horizontally** split the currently focused window by adding a new terminal.
SHIFT+CTRL+O	**Vertically** split the currently focused window by adding a new terminal.

For example, pressing the following combination:

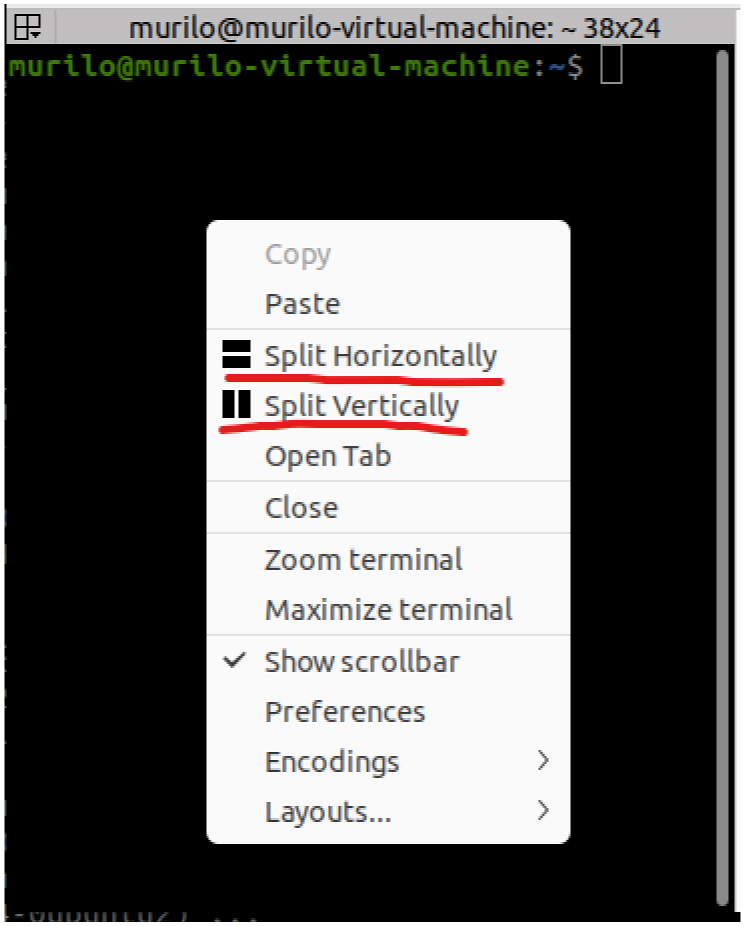
1. CTRL+ALT+T
2. SHIFT+CTRL+E
3. SHIFT+CTRL+O

Will result in three terminal windows that look like so.



4.2 OK, but what if shortcuts scare me

Instead of using shortcuts, a context menu can be opened by right-clicking a terminal window. Then, you can choose to *Split Horizontally* or *Split Vertically* to achieve the same results.


murilo@murilo-virtual-machine: ~ 38x24
murilo@murilo-virtual-machine:~$
Copy
Paste
Split Horizontally
Split Vertically
Open Tab
Close
Zoom terminal
Maximize terminal
Show scrollbar
Preferences
Encodings
Layouts...

CHAPTER

FIVE

WORKSPACE SETUP

Similar to how ROS2 files are installed in /opt/ros/{ROS_DISTRO} so that you can have several distributions installed simultaneously, you can also have many separate workspaces in your system.

In addition, because files in the /opt folder require superuser privileges (for good reasons), having a user-wide workspace is the accepted practice. They call this an **overlay**.

5.1 Setting up

In ROS2, a workspace is nothing more than a folder in which all your packages are contained.

No, really, you just need to make a folder, e.g. the one we will use throughout these tutorials.

```
cd ~
mkdir -p ros2_tutorial_workspace/src
```

It is common practice to have all source files inside the src folder, so we will also do so for these tutorials. Nonetheless, it is not a strict requirement.

5.2 First build

Regardless of it being a currently empty project, we run **colcon** once to set up the environment and illustrate a few things. The program **colcon** is the build system of ROS2 and will be described in more detail later.

For now, run

```
cd ~/ros2_tutorial_workspace
colcon build
```

for which the output will be something similar to

```
Summary: 0 packages finished [0.17s]
```

given that we have an empty workspace, no surprise here.

The folders build, install, and log have been generated automatically by **colcon**. The project structure becomes as follows.

```
ros2_tutorial_workspace/
  └── src/
  └── build/
```

(continues on next page)

(continued from previous page)

```
└── install/
└── log/
```

Inside the `install` folder lie all programs etc generated by the project that can be accessed by the users.

Do the following just once, so that all terminal windows automatically source this new workspace for you.

```
echo "# Source the ROS2 overlay, as instructed in https://ros2-tutorial.readthedocs.io" >
↪> ~/.bashrc
echo "source ~/ros2_tutorial_workspace/install/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

However, since our workspace is currently empty, there's not much we can do with it. Let's add some content.

CHAPTER SIX

CREATE PACKAGES (ROS2 PKG CREATE)

ROS2 has a tool to help create package templates. We can get all available options by running

```
ros2 pkg create -h
```

which outputs a list of handy options to populate the package template with useful files. Namely, the four emphasized ones.

```
usage: ros2 pkg create [-h] [--package-format {2,3}] [--description DESCRIPTION]
                       [--license LICENSE]
                       [--destination-directory DESTINATION_DIRECTORY]
                       [--build-type {cmake,ament_cmake,ament_python}]
                       [--dependencies DEPENDENCIES [DEPENDENCIES ...]]
                       [--maintainer-email MAINTAINER_EMAIL]
                       [--maintainer-name MAINTAINER_NAME] [--node-name NODE_NAME]
                       [--library-name LIBRARY_NAME]
                       package_name

Create a new ROS 2 package

positional arguments:
  package_name          The package name

options:
  -h, --help            show this help message and exit
  --package-format {2,3}, --package_format {2,3}
                        The package.xml format.
  --description DESCRIPTION
                        The description given in the package.xml
  --license LICENSE     The license attached to this package; this can be an arbitrary
                        string, but a LICENSE file will only be generated if it is one
                        of the supported licenses (pass '?' to get a list)
  --destination-directory DESTINATION_DIRECTORY
                        Directory where to create the package directory
  --build-type {cmake,ament_cmake,ament_python}
                        The build type to process the package with
  --dependencies DEPENDENCIES [DEPENDENCIES ...]
                        list of dependencies
  --maintainer-email MAINTAINER_EMAIL
                        email address of the maintainer of this package
  --maintainer-name MAINTAINER_NAME
                        name of the maintainer of this package
```

(continues on next page)

(continued from previous page)

```
  --node-name NODE_NAME
                        name of the empty executable
  --library-name LIBRARY_NAME
                        name of the empty library
```

CHAPTER

SEVEN

CREATING A PYTHON PACKAGE (FOR AMENT_PYTHON)

Note: This is **NOT** the only way to build Python packages in ROS2.

Packages in ROS2 can either rely on `CMake` or directly use setup tools available in Python. For pure Python projects, it might be easier to use `ament_python`, so we start this tutorial with it.

Let us build the simplest of Python packages and start from there.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create the_simplest_python_package \
--build-type ament_python
```

Hint: If you don't explicitly define the mantainer name and email, `ros2 pkg create` will try to:

1. Define the mantainer's name as the currently logged-in user's name (see source and source).
2. Define the mantainer's email by getting it from `git` (see source). It will get whatever is defined with `git config --global user.email`.

which will result in the output below, meaning the package has been generated successfully.

```
going to create a new package
package name: the_simplest_python_package
destination directory: /home/murilo/ros2_tutorial_workspace/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['murilo <murilomarinho@ieee.org>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
creating folder ./the_simplest_python_package
creating ./the_simplest_python_package/package.xml
creating source folder
creating folder ./the_simplest_python_package/the_simplest_python_package
creating ./the_simplest_python_package/setup.py
creating ./the_simplest_python_package/setup.cfg
creating folder ./the_simplest_python_package/resource
creating ./the_simplest_python_package/resource/the_simplest_python_package
creating ./the_simplest_python_package/the_simplest_python_package/__init__.py
```

(continues on next page)

(continued from previous page)

```
creating folder ./the_simplest_python_package/test
creating ./the_simplest_python_package/test/test_copyright.py
creating ./the_simplest_python_package/test/test_flake8.py
creating ./the_simplest_python_package/test/test_pep257.py

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the␣
↪package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

We can build the workspace that now has this empty package using **colcon**

```
cd ~/ros2_tutorial_workspace
colcon build
```

which will now output

```
Starting >>> the_simplest_python_package
--- stderr: the_simplest_python_package
/usr/lib/python3/dist-packages/setuptools/command/install.py:34:␣
↪SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip and␣
↪other standards-based tools.
  warnings.warn(
---
Finished <<< the_simplest_python_package [1.72s]

Summary: 1 package finished [1.89s]
  1 package had stderr output: the_simplest_python_package
```

meaning that **colcon** successfully built the example package.

> **Warning:** In this version of ROS2, all **ament_python** packages will output a `SetuptoolsDeprecationWarning`. This is related to this issue on Github. Until that is fixed, just ignore it.

CHAPTER EIGHT

CREATING A PYTHON NODE WITH A TEMPLATE (FOR AMENT_PYTHON)

It is always good to rely on the templates available in **ros2 pkg create**, mostly because the best practices for packaging might change between ROS2 versions.

Let us use the template for creating a package with a Node, as follows.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create python_package_with_a_node \
--build-type ament_python \
--node-name sample_python_node
```

Which will output many things in common with the prior example, but with two major differences.

1. It generates a template Node
2. The `setup.py` has information about the Node.

```
going to create a new package
package name: python_package_with_a_node
destination directory: ~/ros2_tutorial_workspace/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['murilo <murilomarinho@ieee.org>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
node_name: sample_python_node
creating folder ./python_package_with_a_node
creating ./python_package_with_a_node/package.xml
creating source folder
creating folder ./python_package_with_a_node/python_package_with_a_node
creating ./python_package_with_a_node/setup.py
creating ./python_package_with_a_node/setup.cfg
creating folder ./python_package_with_a_node/resource
creating ./python_package_with_a_node/resource/python_package_with_a_node
creating ./python_package_with_a_node/python_package_with_a_node/__init__.py
creating folder ./python_package_with_a_node/test
creating ./python_package_with_a_node/test/test_copyright.py
creating ./python_package_with_a_node/test/test_flake8.py
creating ./python_package_with_a_node/test/test_pep257.py
creating ./python_package_with_a_node/python_package_with_a_node/sample_python_node.py
```

(continues on next page)

(continued from previous page)

```
[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the␣
↪package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

Then, we can build the workspace as usual to consider the new package as well.

```
cd ~/ros2_tutorial_workspace
colcon build
```

which will result in going through the package we created in the prior example and the current one.

```
Starting >>> python_package_with_a_node
Starting >>> the_simplest_python_package
--- stderr: python_package_with_a_node
/usr/lib/python3/dist-packages/setuptools/command/install.py:34:␣
↪SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip and␣
↪other standards-based tools.
  warnings.warn(
---
Finished <<< python_package_with_a_node [1.16s]
--- stderr: the_simplest_python_package
/usr/lib/python3/dist-packages/setuptools/command/install.py:34:␣
↪SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip and␣
↪other standards-based tools.
  warnings.warn(
---
Finished <<< the_simplest_python_package [1.17s]

Summary: 2 packages finished [1.36s]
  2 packages had stderr output: python_package_with_a_node the_simplest_python_package
```

CHAPTER NINE

ALWAYS SOURCE AFTER YOU BUILD

When creating new packages or modifying existing ones, many changes will not be visible by the system unless our workspace is re-sourced.

For example, if we try the following in the terminal window we used to first build this example package

```
ros2 run python_package_with_a_node sample_python_node
```

it will not work and will output

```
Package 'python_package_with_a_node' not found
```

As the workspace grows bigger and the packages more complex, figuring out such errors becomes a considerable hassle. My suggestion is to always source after a build, so that sourcing errors can always be ruled out.

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Hint for the future you

In rare cases, the workspace can be left in an unclean state in which older build artifacts cause build and runtime issues, such as failed builds and programs that do not seem to match their intended source code. These artifacts might include old files that should have been removed, issues with dependencies, and so on. In those cases, it might be good to remove the `build`, `install`, and `log` folders before rebuilding and re-sourcing.

Hint for the future you

It might also be the case that certain packages fail to build after `build`, `install`, and `log` are removed, or that the build only works after **`colcon`** is called twice in a row. This is usually because the dependencies of the packages in your workspace are poorly configured and, in consequence, ROS2 is not building them in the correct order. If your workspace does not build properly after being cleaned as mentioned above, you must correct its dependencies until it builds properly.

CHAPTER

TEN

RUNNING A NODE (ROS2 RUN)

The most basic way of running a Node is using the ROS2 tool **ros2 run**.

More information on it can be obtained through

```
ros2 run -h
```

which returns the most relevant arguments `package_name` and `executable_name`.

```
usage: ros2 run [-h] [--prefix PREFIX] package_name executable_name ...

Run a package specific executable

positional arguments:
  package_name     Name of the ROS package
  executable_name  Name of the executable
  argv             Pass arbitrary arguments to the executable

options:
  -h, --help       show this help message and exit
  --prefix PREFIX  Prefix command, which should go before the executable. Command must␣
↪be wrapped
                   in quotes if it contains spaces (e.g. --prefix 'gdb -ex run --args').
```

Back to our example, with a properly sourced terminal, the example node can be executed with

```
ros2 run python_package_with_a_node sample_python_node
```

which will now correctly output

```
Hi from python_package_with_a_node.
```

CHAPTER

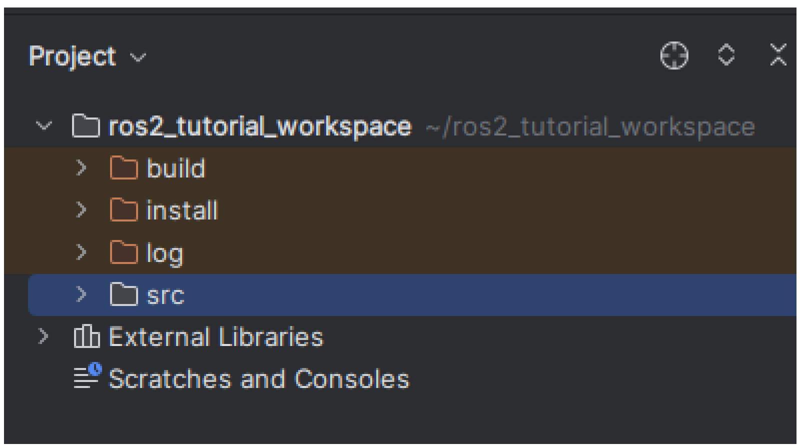
ELEVEN

USING PYCHARM FOR ROS2 SOURCES

With **PyCharm** opened as instructed in *Editing Python source (with PyCharm)*, here are a few tips to make your life easier.

1. Go to *File* → *Open...* and browse to our workspace folder `~/ros2_tutorial_workspace`
2. Right-click the folder `install` and choose *Mark Directory as* → *Excluded*. Do the same for `build` and `log`

Your project view should look like so



11.1 Running a Node from PyCharm

With the project correctly configured, you can

1. move to *src* → *python_package_with_a_node* → *python_package_with_a_node*.
2. double (left) click **sample_python_node.py** to open the source code, showing the contents of the Node. It is minimal to the point that it doesn't have anything related to **ROS** at all.
3. right click **sample_python_node.py** and choose *Run sample_python_node*

It will output in **PyCharm**'s console

```
Hi from python_package_with_a_node.
```

Note: You should extensively use the Debugger in **PyCharm** when developing code. If you're still adding `print` functions to figure out what is wrong with your code, now is the opportunity you always needed to stop doing that and join the adult table.

Note: You can read more about debugging with **PyCharm** at the official documentation.

11.2 What to do when PyCharm does not find the dependencies

Note: This section is meant to help you troubleshoot if this ever happens to you. It can be safely skipped if you're following the tutorial for the first time.

Note: There might be ways to adjust the settings of **PyCharm** or other IDEs to save us from the trouble of having to do this. Nonetheless, this is the *one-size-fits-most* solution, which should work for all past and future versions of **PyCharm**.

If you have ruled out all issues related to your own code, it might be the case that the terminal in which you initially ran **PyCharm** is unaware of certain changes to your ROS2 workspace.

To be sure that the current **PyCharm** session is updated without changes to any settings, do

1. Close **PyCharm**.
2. Build and source the ROS2 workspace.

   ```
   cd ~/ros2_tutorial_workspace
   colcon build
   source install/setup.bash
   ```

   Note: If you don't remember why we're building with these commands, see *Always source after you build*.

3. Re-open **PyCharm**.

   ```
   pycharm_ros2
   ```

CHAPTER

TWELVE

CREATING A PYTHON NODE FROM SCRATCH (FOR AMENT_PYTHON)

TL;DR Making an ament_python Node

1. Modify package.xml with any additional dependencies.
2. Create the Node.
3. Modify the setup.py file.

Let us add an additional Node to our **ament_python** package that actually uses ROS2 functionality. These are the steps that must be taken, in general, to add a new Node.

12.1 Handling dependencies (package.xml)

The package.xml was automatically generated by **ros2 pkg create** and holds basic information about the package.

One important role of package.xml is to declare dependencies with other ROS2 packages. It is common for new Nodes to have additional dependencies, so we will cover that here. For any ROS2 package, we must modify the package.xml to add new dependencies.

In this toy example, let us add rclpy as a dependency because it is the Python implementation of the RCL (ROS Client Library). All Nodes that use anything related to ROS2 will directly or indirectly depend on that library.

By no coincidence, the package.xml has the .xml extension, meaning that it is written in XML (Extensible Markup Language).

Let us add the dependency between the <license> and <test_depend> tags. This is not a strict requirement but is where it commonly is for standard packages.

~/ros2_tutorial_workspace/src/python_package_with_a_node/package.xml

```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>python_package_with_a_node</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="murilomarinho@ieee.org">murilo</maintainer>
  <license>TODO: License declaration</license>

  <depend>rclpy</depend>
```

(continues on next page)

(continued from previous page)

```

  <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>
```

12.2 After you modify the workspace, build it once

After you add a new dependency to package.xml, nothing really changes in the workspace unless a new build is performed.

In addition, when programming with new dependencies, unless you rebuild the workspace, **PyCharm** will not recognize the libraries, and autocomplete will not work.

So,

1. close **PyCharm**.
2. **Run (in the terminal you used to run PyCharm before)**

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

3. **Re-open pycharm**

```
pycharm_ros2
```

12.3 Creating the Node

In the directory src/python_package_with_a_node/python_package_with_a_node, create a new file called print_forever_node.py. Copy and paste the following contents into the file.

~/ros2_tutorial_workspace/src/python_package_with_a_node/python_package_with_a_node/print_forever_node.py

```python
import rclpy
from rclpy.node import Node


class PrintForever(Node):
    """A ROS2 Node that prints to the console periodically."""
```

(continues on next page)

(continued from previous page)

```python

    def __init__(self):
        super().__init__('print_forever')
        timer_period: float = 0.5
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.print_count: int = 0

    def timer_callback(self):
        """Method that is periodically called by the timer."""
        self.get_logger().info(f'Printed {self.print_count} times.')
        self.print_count = self.print_count + 1


def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        print_forever_node = PrintForever()

        rclpy.spin(print_forever_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

By now, this should be enough for you to be able to run the node in **PyCharm**. You can right-click it and choose *Debug print_forever_node*. This will output

```
[INFO] [1683009340.877110693] [print_forever]: Printed 0 times.
[INFO] [1683009341.336559942] [print_forever]: Printed 1 times.
[INFO] [1683009341.836334639] [print_forever]: Printed 2 times.
[INFO] [1683009342.336555088] [print_forever]: Printed 3 times.
```

To finish, press the *Stop* button or press `CTRL+F2` on **PyCharm**. The node will exit gracefully with

```
Process finished with exit code 0
```

12.4 Making `ros2 run` work

Even though you can run the new node in **PyCharm**, we need an additional step to make it deployable in a place where **ros2 run** can find it.

To do so, we modify the `console_scripts` key in the `entry_points` dictionary defined in `setup.py`, to have our new node, as follows

Hint: `console_scripts` expects a `list` of `str` in a specific format. Hence, follow the format properly and don't forget the commas to separate elements in the `list`.

~/ros2_tutorial_workspace/src/python_package_with_a_node/setup.py

```python
from setuptools import setup

package_name = 'python_package_with_a_node'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
         ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='murilo',
    maintainer_email='murilomarinho@ieee.org',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'sample_python_node = python_package_with_a_node.sample_python_node:main',
            'print_forever_node = python_package_with_a_node.print_forever_node:main'
        ],
    },
)
```

The format is straightforward, as follows

print_forever_node	The name of the node when calling it through **ros2 run**.
python_package_with_a_node	The name of the package.
print_forever_node	The name of the script, without the `.py` extension.
main	The function, within the script, that will be called. In general, `main`.

Once again, we have to refresh the workspace so we run

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

And, with that, we can run

```
ros2 run python_package_with_a_node print_forever_node
```

which will output, as expected

```
[INFO] [1683010987.130432622] [print_forever]: Printed 0 times.
[INFO] [1683010987.622780292] [print_forever]: Printed 1 times.
[INFO] [1683010988.122731296] [print_forever]: Printed 2 times.
[INFO] [1683010988.622735422] [print_forever]: Printed 3 times.
```

To stop, press `CTRL+C` on the terminal and the Node will return gracefully.

CHAPTER

THIRTEEN

THE PYTHON NODE, EXPLAINED

Note: The way that a Python Node in ROS2 works, i.e. the explanation in this section, does not depend on the building with **ament_python** or **ament_cmake**.

In a strict sense, the print_forever_node.py is not a minimal Node, but it does showcase most good practices in a Node that actually does something.

13.1 The imports

```
import rclpy
from rclpy.node import Node
```

As in any Python code, we have to import the libraries that we will use and specific modules/classes within those libraries. With rclpy, there is no difference.

13.2 Making a subclass of Node

The current version of ROS2 behaves better when your custom Node is a subclass of rclpy.node.Node. That is achieved with

```
class PrintForever(Node):
    """A ROS2 Node that prints to the console periodically."""

    def __init__(self):
        super().__init__('print_forever')
        timer_period: float = 0.5
```

About inheritance in Python, you can check the official documentation on inheritance and on super().

In more advanced nodes, inheritance does not cut it, but that is an advanced topic to be covered some other time.

13.3 Use a `Timer` for periodic work (when using `rclpy.spin()`)

Tips for the future you

If the code relies on `rclpy.spin()`, a Timer must be used for periodic work.

In its most basic usage, periodic tasks in ROS2 must be handled by a Timer.

To do so, have the node create it with the `create_timer()` method, as follows.

```
    def __init__(self):
        super().__init__('print_forever')
        timer_period: float = 0.5
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.print_count: int = 0
```

The method that is periodically called by the Timer is, in this case, as follows. We use `self.get_logger().info()` to print to the terminal periodically.

```
    def timer_callback(self):
        """Method that is periodically called by the timer."""
        self.get_logger().info(f'Printed {self.print_count} times.')
```

In ROS2, the logging methods, i.e. `self.get_logger().info()`, are methods of the Node itself. So, the capability to log (print to the terminal) using ROS2 Nodes is dependent on the scope in which that Node exists.

13.4 Where the ROS2 magic happens: `rclpy.init()` and `rclpy.spin()`

All the ROS2 magic happens in some sort of `spin()` method. It is called this way because the `spin()` method will constantly loop (or spin) through **items of work**, e.g. scheduled Timer callbacks. All the **items of work** will only be effectively executed when an **executor** runs through it. For simple Nodes, such as the one in this example, the **global** executor is implicitly used. You can read a bit more about that here.

Anyhow, the point is that nothing related to ROS2 will happen unless the two following methods are called. First, `rclpy.init()` is going to initialize a bunch of ROS2 elements behind the curtains, whereas `rclpy.spin()` will block the program and, well, **spin** through Timer callbacks forever. There are alternative ways to `spin()`, but we will not discuss them right now.

```
def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        print_forever_node = PrintForever()

        rclpy.spin(print_forever_node)
```

(continues on next page)

(continued from previous page)

```
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)
```

13.5 Have a try-catch block for KeyboardInterrupt

In the current version of the official ROS2 examples, for reasons beyond my comprehension, this step is not followed.

However, when running Nodes either in the terminal or in **PyCharm**, catching a KeyboardInterrupt is the only reliable way to finish the Nodes cleanly. A KeyboardInterrupt is emitted at a terminal by pressing CTRL+C, whereas it is emitted by **PyCharm** when pressing *Stop*.

That is particularly important when real robots need to be gracefully shut down (otherwise they might inadvertently start the evil robot uprising), but it also looks unprofessional when all your Nodes return with an ugly stack trace.

```
def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        print_forever_node = PrintForever()

        rclpy.spin(print_forever_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)
```

13.6 Document your code with Docstrings

As simple as a code might look for you right now, it needs to be documented for anyone you work with, including the future you. In a few weeks/months/years time, the BeStNoDeYouEvErWrote (TM) might be indistinguishable from Yautja Language.

Add as much description as possible to classes and methods, using the Docstring Convention.

Example of a class:

```
class PrintForever(Node):
    """A ROS2 Node that prints to the console periodically."""
```

Example of a method:

```
    def timer_callback(self):
        """Method that is periodically called by the timer."""
```

CHAPTER

FOURTEEN

CREATING A PYTHON LIBRARY (FOR AMENT_PYTHON)

Let us start, as already recommended in this tutorial, with a template by **ros2 pkg create**.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create python_package_with_a_library \
--build-type ament_python \
--library-name sample_python_library
```

which outputs the forever beautiful wall of text we're now used to, with a minor difference regarding the additional library template, as highlighted below.

```
going to create a new package
package name: python_package_with_a_library
destination directory: /home/murilo/git/ROS2_Tutorial/ros2_tutorial_workspace/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['murilo <murilomarinho@ieee.org>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
library_name: sample_python_library
creating folder ./python_package_with_a_library
creating ./python_package_with_a_library/package.xml
creating source folder
creating folder ./python_package_with_a_library/python_package_with_a_library
creating ./python_package_with_a_library/setup.py
creating ./python_package_with_a_library/setup.cfg
creating folder ./python_package_with_a_library/resource
creating ./python_package_with_a_library/resource/python_package_with_a_library
creating ./python_package_with_a_library/python_package_with_a_library/__init__.py
creating folder ./python_package_with_a_library/test
creating ./python_package_with_a_library/test/test_copyright.py
creating ./python_package_with_a_library/test/test_flake8.py
creating ./python_package_with_a_library/test/test_pep257.py
creating folder ./python_package_with_a_library/python_package_with_a_library/sample_
↪python_library
creating ./python_package_with_a_library/python_package_with_a_library/sample_python_
↪library/__init__.py

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the␣
↪package.xml, but no LICENSE file has been created.
```

(continues on next page)

(continued from previous page)

```
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

14.1 The folders/files, Mason, what do they mean?

The ROS2 package created from the template has a structure like so. In particular, we can see that `python_package_with_a_library` is repeated twice in a row. This is a common source of error, so don't forget!

```
python_package_with_a_library
   └── python_package_with_a_library
      └── sample_python_library
         __init__.py
      __init__.py
   └── resource
      python_package_with_a_library
   └── test
   package.xml
   setup.cfg
   setup.py
```

We learned the meaning of most of those in the preamble, namely *(Murilo's) Python Best Practices*. To quickly clarify a few things, see the table below.

Table 1: ROS2 Python package folders/files explained

File/Directory	Meaning
`python_package_with_a_library`	The ROS2 package folder.
`python_package_with_a_library/python_package_with_a_library`	The Python package, as we saw in the preamble.
`sample_python_library`	The module corresponding to our sample library.
`resource/python_package_with_a_library`	A file for ROS2 to index this package correctly. See Resource file.
`test`	The folder contaning the tests, as we already saw in the preamble.
`setup.cfg`	Used by setup.py, see setup.cfg docs.
`setup.py`	The instructions to make the package installable, as we saw in the preamble.

14.2 Overview of the library

Hint: If you have created the bad habit of declaring all/too many things in your `__init__.py` file, take the hint and start breaking the definitions into different files and use the `__init__.py` just to export the relevant parts of your library.

For the sake of the example, let us create a library with a Python `function` and another one with a `class`. To guide our next steps, we first draw a quick overview of what our `python_package_with_a_library` will look like.

```
python_package_with_a_library
   └── python_package_with_a_library
      └── sample_python_library
         __init__.py
         _sample_class.py
         _sample_function.py
      __init__.py
   └── resource
   └── test
```

With respect to the highlighted files, we will

1. Create the `_sample_function.py`.
2. Create the `_sample_class.py`.
3. Modify `__init__.py` to use the new function and class.

All other files and directories will remain as-is, in the way they were generated by **`ros2 pkg create`**.

14.3 Create the sample function

Create a new file with the following contents and name.

`~/ros2_tutorial_workspace/src/python_package_with_a_library/python_package_with_a_library/sample_python_library/_sample_function.py`

```python
def sample_function_for_square_of_sum(a: float, b: float) -> float:
    """Returns the square of a sum (a + b)^2 = a^2 + 2ab + b^2"""
    return a**2 + 2*a*b + b**2
```

The function has two parameters, `a` and `b`. For simplicity, we're expecting arguments of type `float` and returning a `float`, but it could be any Python function.

14.4 Create the sample class

Create a new file with the following contents and name.

~/ros2_tutorial_workspace/src/python_package_with_a_library/python_package_with_a_library/sample_python_library/_sample_class.py

```
class SampleClass:
    """A sample class to check how they can be imported by other ROS2 packages."""

    def __init__(self, name: str):
        self._name = name

    def get_name(self) -> str:
        """
        Gets the name of this instance.
        :return: This name.
        """
        return self._name
```

The class is quite simple with a private data member and a method to retrieve it.

14.5 Modify the `__init__.py` to export the symbols

With the necessary files created and properly organized, the last step is to `import` the function and the class. We modify proper `__init__.py` file with the following contents.

~/ros2_tutorial_workspace/src/python_package_with_a_library/python_package_with_a_library/sample_python_library/__init__.py

```
from python_package_with_a_library.sample_python_library._sample_class import SampleClass
from python_package_with_a_library.sample_python_library._sample_function import sample_
↪function_for_square_of_sum
```

14.6 Modify the `setup.py` to export the packages

Warning: This step might be unnecessary after this fix.

Note: This is a *one-size-fits-most* solution, which might not work for certain Python package structures. As a generic solution, we will export all Python packages in the ROS2 package excluding the *test* directory. For more information on **setuptools**, see the official Python packaging docs.

~/ros2_tutorial_workspace/src/python_package_with_a_library/setup.py

```
from setuptools import setup, find_packages

package_name = 'python_package_with_a_library'
```

(continues on next page)

(continued from previous page)

```python

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='murilo',
    maintainer_email='murilomarinho@ieee.org',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
        ],
    },
)
```

14.7 Build and source

No surprise here, right?

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

If it builds without any unexpected issues, we're good to go!

CHAPTER

FIFTEEN

USING A PYTHON LIBRARY FROM ANOTHER PACKAGE (FOR AMENT_PYTHON)

Let us create a package with a Node that uses the library we created in the prior example.

Note that we must add the python_package_with_a_library as a dependency to our new package. The easiest way to do so is through **ros2 pkg create**. We also add rclcpp as a dependency so that our Node can do something useful.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create python_package_that_uses_the_library \
--dependencies rclpy python_package_with_a_library \
--build-type ament_python \
--node-name node_that_uses_the_library
```

resulting in yet another version of our favorite wall of text

```
going to create a new package
package name: python_package_that_uses_the_library
destination directory: /home/murilo/ros2_tutorial_workspace/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['murilo <murilomarinho@ieee.org>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: ['rclpy', 'python_package_with_a_library']
node_name: node_that_uses_the_library
creating folder ./python_package_that_uses_the_library
creating ./python_package_that_uses_the_library/package.xml
creating source folder
creating folder ./python_package_that_uses_the_library/python_package_that_uses_the_
↪library
creating ./python_package_that_uses_the_library/setup.py
creating ./python_package_that_uses_the_library/setup.cfg
creating folder ./python_package_that_uses_the_library/resource
creating ./python_package_that_uses_the_library/resource/python_package_that_uses_the_
↪library
creating ./python_package_that_uses_the_library/python_package_that_uses_the_library/__
↪init__.py
creating folder ./python_package_that_uses_the_library/test
creating ./python_package_that_uses_the_library/test/test_copyright.py
creating ./python_package_that_uses_the_library/test/test_flake8.py
```

(continues on next page)

(continued from previous page)

```
creating ./python_package_that_uses_the_library/test/test_pep257.py
creating ./python_package_that_uses_the_library/python_package_that_uses_the_library/
↪node_that_uses_the_library.py

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the␣
↪package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

15.1 The sample Node

Given that it was created from a template, the file `python_package_that_uses_the_library/python_package_that_uses_the_library/node_that_uses_the_library.py` is currently *mostly* empty. Let us replace its contents with

`node_that_uses_the_library.py`

```python
import rclpy
from rclpy.node import Node
from python_package_with_a_library.sample_python_library import SampleClass, sample_
↪function_for_square_of_sum


class NodeThatUsesTheLibrary(Node):
    """A ROS2 Node that prints to the console periodically."""

    def __init__(self):
        super().__init__('node_that_uses_the_library')
        timer_period: float = 0.5
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        """
        Method that is periodically called by the timer.
        Prints out the result of sample_function_for_square_of_sum of two random numbers,
        followed by the result of SampleClass.get_name() for an instance created with
        a ten-character-long ascii string of random characters.
        """
        a: float = random.uniform(0, 1)
        b: float = random.uniform(1, 2)
        c: float = sample_function_for_square_of_sum(a, b)
        self.get_logger().info(f'sample_function_for_square_of_sum({a},{b}) returned {c}.
↪')
```

(continues on next page)

(continued from previous page)

```python

        random_name_ascii: str = ''.join(random.choice(string.ascii_letters) for _ in range(10))
        sample_class_with_random_name = SampleClass(name=random_name_ascii)
        self.get_logger().info(f'sample_class_with_random_name.get_name() '
                               f'returned {sample_class_with_random_name.get_name()}.')


def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        node_that_uses_the_library = NodeThatUsesTheLibrary()

        rclpy.spin(node_that_uses_the_library)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

Indeed, the most difficult part is to make and configure the library itself. After that, to use it in another package, it is straightforward. We `import` the library.

```python
import rclpy
from rclpy.node import Node
from python_package_with_a_library.sample_python_library import SampleClass, sample_function_for_square_of_sum
```

And then use the symbols we imported as we would with any other Python library.

```python
    def timer_callback(self):
        """
        Method that is periodically called by the timer.
        Prints out the result of sample_function_for_square_of_sum of two random numbers,
        followed by the result of SampleClass.get_name() for an instance created with
        a ten-character-long ascii string of random characters.
        """
        a: float = random.uniform(0, 1)
        b: float = random.uniform(1, 2)
        c: float = sample_function_for_square_of_sum(a, b)
        self.get_logger().info(f'sample_function_for_square_of_sum({a},{b}) returned {c}.')

```

(continues on next page)

(continued from previous page)

```
        random_name_ascii: str = ''.join(random.choice(string.ascii_letters) for _ in␣
↪range(10))
        sample_class_with_random_name = SampleClass(name=random_name_ascii)
        self.get_logger().info(f'sample_class_with_random_name.get_name() '
                               f'returned {sample_class_with_random_name.get_name()}.')
```

15.2 Build and source

As always, this is needed so that our new package and node can be recognized by **ros2 run**.

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

15.3 Run

Hint: Remember that you can stop the node at any time with CTRL+C.

```
ros2 run python_package_that_uses_the_library node_that_uses_the_library
```

Which outputs something similar to the shown below, but with different numbers and strings as they are randomized.

```
[INFO] [1683598288.149167944] [node_that_uses_the_library]: sample_function_for_square_
↪of_sum(0.19395834493833486,1.3891603395040568) returned 2.506264769030609.
[INFO] [1683598288.149643378] [node_that_uses_the_library]: sample_class_with_random_
↪name.get_name() returned qyOXLBEtzZ.
[INFO] [1683598288.616095880] [node_that_uses_the_library]: sample_function_for_square_
↪of_sum(0.7387236329957096,1.7650481260672202) returned 6.2688730214810775.
[INFO] [1683598288.616604769] [node_that_uses_the_library]: sample_class_with_random_
↪name.get_name() returned LCFNFyzwhk.
[INFO] [1683598289.116050219] [node_that_uses_the_library]: sample_function_for_square_
↪of_sum(0.003813494022560704,1.7056916575839387) returned 2.9224078633691604.
[INFO] [1683598289.116553899] [node_that_uses_the_library]: sample_class_with_random_
↪name.get_name() returned wrtTlOdanZ.
```

CHAPTER

SIXTEEN

MESSAGES AND SERVICES (ROS2 INTERFACE)

If by now you haven't particularly fallen in love with ROS2, fear not. Indeed, we haven't done much so far that couldn't be achieved more easily by other means.

ROS2 begins to shine most in its interprocess communication, through what are called ROS2 interfaces. In particular, the fact that we can easily interface Nodes written in Python and C++ is a strong selling point.

`Messages` are one of the three types of ROS2 interfaces. This will most likely be the standard of communication between Nodes in your packages. We will also see the bidirectional `Services` now. The last type of interface, `Actions`, is left for another section.

16.1 Description

In ROS2, interfaces are files written in the ROS2 IDL (Interface Description Language). Each type of interface is described in a `.msg` file (or `.srv` file), which is then built by **`colcon`** into libraries that can be imported into your Python programs.

When dealing with common robotics concepts such as geometric and sensor messages, it is good practice to use interfaces that already exist in ROS2, instead of creating new ones that serve the exact same purpose. In addition, for complicated interfaces, we can combine existing ones for simplicity.

16.2 Getting info on interfaces

We can get information about ROS2 interfaces available in our system with **`ros2 interface`**. Let us first get more information about the program usage with

```
ros2 interface -h
```

which results in

```
usage: ros2 interface [-h] Call `ros2 interface <command> -h` for more detailed usage. ..
↪.

Show information about ROS interfaces

options:
  -h, --help            show this help message and exit

Commands:
  list      List all interface types available
```

(continues on next page)

(continued from previous page)

```
  package   Output a list of available interface types within one package
  packages  Output a list of packages that provide interfaces
  proto     Output an interface prototype
  show      Output the interface definition

  Call `ros2 interface <command> -h` for more detailed usage.
```

This shows that with **ros2 interface list** we can get a list of all interfaces available in our workspace. That returns a huge list of interfaces, so it will not be replicated entirely here. Instead, we can run

```
ros2 interface packages
```

to get the list of packages with interfaces available, which returns something similar to

```
action_msgs
action_tutorials_interfaces
actionlib_msgs
builtin_interfaces
composition_interfaces
diagnostic_msgs
example_interfaces
geometry_msgs
lifecycle_msgs
logging_demo
map_msgs
nav_msgs
pcl_msgs
pendulum_msgs
rcl_interfaces
rmw_dds_common
rosbag2_interfaces
rosgraph_msgs
sensor_msgs
shape_msgs
statistics_msgs
std_msgs
std_srvs
stereo_msgs
tf2_msgs
trajectory_msgs
turtlesim
unique_identifier_msgs
visualization_msgs
```

From those, `sensor_msgs` and `geometry_msgs` are packages to always keep in mind when looking for a suitable interface. It will help to keep your Nodes compatible with the community.

> **Warning:** The `std_msgs` package, widely used in ROS1, is deprecated in ROS2 since Foxy. The `example_interfaces` somewhat takes its place, but the recommended practice is to create "semantically meaningful message types". They might remove both or either of these in future versions, so do not use them.

As an example, let us take a look into the `example_interfaces` package, containing, as the name implies, example interface types. We can do so with

```
ros2 interface package example_interfaces
```

which returns

```
example_interfaces/msg/String
example_interfaces/srv/AddTwoInts
example_interfaces/srv/SetBool
example_interfaces/msg/UInt8
example_interfaces/msg/Int64MultiArray
example_interfaces/msg/Byte
example_interfaces/msg/Float32
example_interfaces/msg/Int64
example_interfaces/msg/UInt32MultiArray
example_interfaces/msg/Int32MultiArray
example_interfaces/msg/Empty
example_interfaces/msg/Float32MultiArray
example_interfaces/msg/Int16MultiArray
example_interfaces/action/Fibonacci
example_interfaces/msg/UInt16MultiArray
example_interfaces/msg/Int8MultiArray
example_interfaces/msg/Bool
example_interfaces/msg/ByteMultiArray
example_interfaces/msg/MultiArrayLayout
example_interfaces/msg/UInt8MultiArray
example_interfaces/msg/UInt16
example_interfaces/msg/Int16
example_interfaces/msg/Int8
example_interfaces/msg/MultiArrayDimension
example_interfaces/msg/Char
example_interfaces/msg/Float64
example_interfaces/srv/Trigger
example_interfaces/msg/UInt64
example_interfaces/msg/WString
example_interfaces/msg/Int32
example_interfaces/msg/Float64MultiArray
example_interfaces/msg/UInt64MultiArray
example_interfaces/msg/UInt32
```

16.3 Messages

For example, let's say that we are interested in looking up the contents of `example_interfaces/msg/String`. We can do so with **ros2 interface show**, like so

```
ros2 interface show example_interfaces/msg/String
```

which returns the contents of the source file used to create this message

```
# This is an example message of using a primitive datatype, string.
# If you want to test with this that's fine, but if you are deploying
# it into a system you should create a semantically meaningful message type.
```

(continues on next page)

(continued from previous page)

```
# If you want to embed it in another message, use the primitive data type instead.
string data
```

Basically, the comments help to emphasize that interface types with too broad meaning are unloved in ROS2. Given that these example interfaces are either unsupported or only loosely supported, do not rely on them.

The real content of the message file is `string data`, showing that it contains a single string called `data`. Using `ros2 interface show` on other example interfaces, it is easy to see how to build interesting message types to fit our needs.

16.4 Services

In the case of a service, let's look up the contents of `example_interfaces/srv/AddTwoInts`.

We run

```
ros2 interface show example_interfaces/srv/AddTwoInts
```

that results in

```
int64 a
int64 b
---
int64 sum
```

Notice that the `---` is what separates the `Request`, above, from the `Response` below. Anyone using this service would expect that the result would be $sum = a + b$, but this logic needs to be implemented on the Node. The service itself is just a way of bidirectional communication.

CHAPTER

SEVENTEEN

CREATING A DEDICATED PACKAGE FOR CUSTOM INTERFACES

> **Warning:** Despite this push in ROS2 towards having the users define even the simplest of message types, to define new interfaces in ROS2 we must use an **ament_cmake** package. It **cannot** be done with an **ament_python** package.

All interfaces in ROS2 must be made in an **ament_cmake** package. We have so far not needed it, but for this scenario we cannot escape. Thankfully, for this we don't need to dig too deep into **CMake**, so fear not.

17.1 Creating the package

There isn't a template for message-only packages using **ros2 pkg create**. We'll need to build on top of a mostly empty **ament_cmake** package.

To take this chance to also learn how to nest messages on other interfaces, we also add the dependency on geometry_msgs.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create package_with_interfaces \
--build-type ament_cmake \
--dependencies geometry_msgs
```

which again shows our beloved wall of text, with a few highlighted differences because of it being a **ament_cmake** package.

```
going to create a new package
package name: package_with_interfaces
destination directory: /home/murilo/git/ROS2_Tutorial/ros2_tutorial_workspace/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['murilo <murilomarinho@ieee.org>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: [geometry_msgs]
creating folder ./package_with_interfaces
creating ./package_with_interfaces/package.xml
creating source and include folder
creating folder ./package_with_interfaces/src
creating folder ./package_with_interfaces/include/package_with_interfaces
```

(continues on next page)

(continued from previous page)

```
creating ./package_with_interfaces/CMakeLists.txt

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

The package.xml works the same way as when using ament_python. However, we no longer have a setup.py or setup.cfg, everything is handled by the CMakeLists.txt.

17.2 The package.xml dependencies

Whenever the package has any type of interface, the package.xml **must** include three specific dependencies. Namely, the ones highlighted below. Edit the package_with_interfaces/package.xml like so

package.xml

```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>package_with_interfaces</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="murilomarinho@ieee.org">murilo</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>geometry_msgs</depend>

  <buildtool_depend>rosidl_default_generators</buildtool_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

17.3 The message folder

The convention is to add all messages to a folder called `msg`. Let's follow that convention

```
cd ~/ros2_tutorial_workspace/src/package_with_interfaces
mkdir msg
```

17.4 The message file

Note: Here is a list of available built-in types for ROS2 interfaces.

Let us create a message file to transfer inspirational quotes between Nodes. For example, the one below.

> Use the force, Pikachu!
>
> —Uncle Ben

There are many ways to represent this, but for the sake of the example let us give each message an `id` and two rather obvious fields. Create a file called `AmazingQuote.msg` in the folder `msg` that we just created with the following contents.

`AmazingQuote.msg`

```
# AmazingQuote.msg from https://ros2-tutorial.readthedocs.io
# An inspirational quote a day keeps the therapist away
int32 id
string quote
string philosopher_name
```

17.5 The service folder

The convention is to add all services to a folder called `srv`. Let's follow that convention

```
cd ~/ros2_tutorial_workspace/src/package_with_interfaces
mkdir srv
```

17.6 The service file

With the `AmazingQuote.msg`, we have seen how to use built-in types. Let's use the service to learn two more possibilities. Let us use a message from the same package and a message from another package. Services cannot be used to define other services.

Add the file `WhatIsThePoint.srv` in the `srv` folder with the following contents

`WhatIsThePoint.srv`

```
# WhatIsThePoint.srv from https://ros2-tutorial.readthedocs.io
# Receives an AmazingQuote and returns what is the point
AmazingQuote quote
---
geometry_msgs/Point point
```

Note that if the message is defined in the same package, the package name does not appear in the service or message definition. If the message is defined elsewhere, we have to specify it.

17.7 The `CMakeLists.txt` directives

Note: The order of the **CMake** directives is very important and getting the order wrong can result in bugs with cryptic error messages.

If a package is dedicated to interfaces, there is no need to worry too much about the **CMake** details. We can follow the boilerplate as shown below. Edit the `package_with_interfaces/CMakeLists.txt` like so

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(package_with_interfaces)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(geometry_msgs REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)
find_package(rosidl_default_generators REQUIRED)

#### ROS2 Interface Directives ####
set(interface_files
  # Messages
  "msg/AmazingQuote.msg"

  # Services
  "srv/WhatIsThePoint.srv"

)

rosidl_generate_interfaces(${PROJECT_NAME}
  ${interface_files}
  DEPENDENCIES
  geometry_msgs

)
```

(continues on next page)

(continued from previous page)

```

ament_export_dependencies(
  rosidl_default_runtime
)
#### ROS2 Interface Directives [END] ####

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

17.8 What to do when adding new interfaces?

TL;DR Adding new interfaces

1. Add new dependencies to `package.xml`
2. Add each new interface file to `set(interface_files ...)`
3. Add new dependencies to `rosidl_generate_interfaces(... DEPENDENCIES ...)`

Yes, you **MUST** add the same dependency in two places!

If additional interfaces are required

1. Modify the `package.xml` to have any additional dependencies. See *Handling dependencies (package.xml)* for more details.
2. Add each new interface file to `set(interface_files ...)`

```
set(interface_files
  # Messages
  "msg/AmazingQuote.msg"

  # Services
  "srv/WhatIsThePoint.srv"

)
```

Note: There are ways to use `CMake` directives to automatically add all files in a given folder and provide other conveniences. In hindsight, that might seem to reduce our burden. However, the method described herein is the one used in the official ROS2 packages (e.g. geometry_msgs), so let us trust that they have good reasons for it.

17.9 Build and source

Before we proceed, let us build and source once.

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

17.10 Getting info on custom interfaces

As long as the package has been correctly built and sourced, we can easily get information on its interfaces using **ros2 interface**.

For instance, running

```
ros2 interface package package_with_interfaces
```

returns

```
package_with_interfaces/srv/WhatIsThePoint
package_with_interfaces/msg/AmazingQuote
```

and we can further get more specific info on `AmazingQuote.msg`

```
ros2 interface show package_with_interfaces/msg/AmazingQuote
```

which returns

```
# AmazingQuote.msg from https://ros2-tutorial.readthedocs.io
# An inspirational quote a day keeps the therapist away
int32 id
string quote
string philosopher_name
```

alternatively, we can do the same for `WhatIsThePoint.srv`

```
ros2 interface show package_with_interfaces/srv/WhatIsThePoint
```

which returns expanded information on each field of the service

```
# WhatIsThePoint.srv from https://ros2-tutorial.readthedocs.io
# Receives an AmazingQuote and returns what is the point
AmazingQuote quote
   int32 id
   string quote
```

(continues on next page)

(continued from previous page)

```
   string philosopher_name
---
geometry_msgs/Point point
   float64 x
   float64 y
   float64 z
```

CHAPTER

EIGHTEEN

PUBLISHERS AND SUBSCRIBERS: USING MESSAGES

Note: Except for the particulars of the `setup.py` file, the way that publishers and subscribers in ROS2 work in Python, i.e. the explanation in this section, does not depend on **ament_python** or **ament_cmake**.

Finally, we reached the point where ROS2 becomes appealing. As you saw in the last section, we can easily create complex interface types using an easy and generic description. We can use those to provide interprocess communication, i.e. two different programs talking to each other, which otherwise can be error-prone and very difficult to implement.

ROS2 works on a model in which any number of processes can communicate over a `Topic` that only accepts one message type. Each topic is uniquely identified by a string.

Then

- A program that sends (publishes) information to the topic has a `Publisher`.
- A program that reads (subscribes) information from a topic has a `Subscriber`.

Each Node can have any number of `Publishers` and `Subscribers` and a combination thereof, connecting to an arbitrary number of Nodes. This forms part of the connections in the so-called ROS graph.

18.1 Create the package

First, let us create an **ament_python** package that depends on our newly developed `packages_with_interfaces` and build from there.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create python_package_that_uses_the_messages \
--build-type ament_python \
--dependencies rclpy package_with_interfaces
```

18.2 Overview

Note: By no coincidence, I am using the terminology Node *with* a publisher, and Node *with* a subscriber. In general, each Node will have a combination of publishers, subscribers, and other interfaces.

Before we start exploring the elements of the package, let us

1. Create the Node with a publisher.

2. Create the Node with a subscriber.
3. Update the `setup.py` so that **`ros2 run`** finds these programs.

18.3 Create the Node with a publisher

TL;DR Creating a publisher

1. Add new dependencies to `package.xml`
2. Import new messages `from <package_name>.msg import <msg_name>`
3. In a subclass of `Node`
   1. Create a publisher with `self.publisher = self.create_publisher(...)`
   2. Send messages with `self.publisher.publish(....)`
4. Add the new Node to `setup.py`

For the publisher, create a file in `python_package_that_uses_the_messages/python_package_that_uses_the_messages` called `amazing_quote_publisher_node.py`, with the following contents

`amazing_quote_publisher_node.py`

```python
import rclpy
from rclpy.node import Node
from package_with_interfaces.msg import AmazingQuote


class AmazingQuotePublisherNode(Node):
    """A ROS2 Node that publishes an amazing quote."""

    def __init__(self):
        super().__init__('amazing_quote_publisher_node')

        self.amazing_quote_publisher = self.create_publisher(
            msg_type=AmazingQuote,
            topic='/amazing_quote',
            qos_profile=1)

        timer_period: float = 0.5
        self.timer = self.create_timer(timer_period, self.timer_callback)

        self.incremental_id: int = 0

    def timer_callback(self):
        """Method that is periodically called by the timer."""

        amazing_quote = AmazingQuote()
        amazing_quote.id = self.incremental_id
        amazing_quote.quote = 'Use the force, Pikachu!'
        amazing_quote.philosopher_name = 'Uncle Ben'
```

(continues on next page)

(continued from previous page)

```python

        self.amazing_quote_publisher.publish(amazing_quote)

        self.incremental_id = self.incremental_id + 1


def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        amazing_quote_publisher_node = AmazingQuotePublisherNode()

        rclpy.spin(amazing_quote_publisher_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

When we built our `package_with_interfaces` in the last section, what ROS2 did for us, among other things, was create a Python library called `package_with_interfaces.msg` containing the Python implementation of the `AmazingQuote.msg`. Because of that, we can use it by importing it like so

```python
import rclpy
from rclpy.node import Node
from package_with_interfaces.msg import AmazingQuote
```

The publisher must be created with the `Node.create_publisher(...)` method, which has the three parameters defined in the publisher and subscriber parameter table.

```python
        self.amazing_quote_publisher = self.create_publisher(
            msg_type=AmazingQuote,
            topic='/amazing_quote',
            qos_profile=1)
```

msg_type	A class, namely the message that will be used in the topic. In this case, `AmazingQuote`.
topic	The topic through which the communication will occur. Can be arbitrarily chosen, but to make sense `/amazing_quote`.
qos_pro	The simplest interpretation for this parameter is the number of messages that will be stored in the `spin(...)` takes too long to process them. (See more on docs for QoSProfile.)

> **Warning:** All the arguments in publisher and subscriber parameter table should be *EXACTLY* the same in the `Publishers` and `Subscribers` of the same topic.

Then, each message is handled much like any other class in Python. We instantiate and initialize the message as follows

```
        amazing_quote = AmazingQuote()
        amazing_quote.id = self.incremental_id
        amazing_quote.quote = 'Use the force, Pikachu!'
        amazing_quote.philosopher_name = 'Uncle Ben'
```

Lastly, the message needs to be published using `Node.publish(msg)`.

```
        self.amazing_quote_publisher.publish(amazing_quote)
```

Note: In general, the message will **NOT** be published instantaneously after `Node.publish()` is called. It is usually fast, but entirely dependent on `rclpy.spin()` and how much work it is doing.

18.4 Create the Node with a subscriber

TL;DR Creating a subscriber

1. Add new dependencies to `package.xml`
2. Import new messages `from <package_name>.msg import <msg_name>`
3. In a subclass of `Node`
   1. Create a callback `def callback(self, msg):`
   2. Create a subscriber `self.subscriber = self.create_subscription(...)`
4. Add the new Node to `setup.py`

For the subscriber Node, create a file in `python_package_that_uses_the_messages/python_package_that_uses_the_messages` called `amazing_quote_subscriber_node.py`, with the following contents

`amazing_quote_subscriber_node.py`

```
import rclpy
from rclpy.node import Node
from package_with_interfaces.msg import AmazingQuote


class AmazingQuoteSubscriberNode(Node):
    """A ROS2 Node that receives and prints an amazing quote."""

    def __init__(self):
        super().__init__('amazing_quote_subscriber_node')
        self.amazing_quote_subscriber = self.create_subscription(
            msg_type=AmazingQuote,
```

(continues on next page)

(continued from previous page)

```python
            topic='/amazing_quote',
            callback=self.amazing_quote_subscriber_callback,
            qos_profile=1)

    def amazing_quote_subscriber_callback(self, msg: AmazingQuote):
        """Method that is periodically called by the timer."""
        self.get_logger().info(f"""
        I have received the most amazing of quotes.
        It says

               '{msg.quote}'

        And was thought by the following genius

            -- {msg.philosopher_name}

        This latest quote had the id={msg.id}.
        """)


def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        amazing_quote_subscriber_node = AmazingQuoteSubscriberNode()

        rclpy.spin(amazing_quote_subscriber_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

Similarly to the publisher, in the subscriber, we start by importing the message in question

```python
import rclpy
from rclpy.node import Node
from package_with_interfaces.msg import AmazingQuote
```

Then, in our subclass of `Node`, we call `Node.create_publisher(...)` as follows

```python
        self.amazing_quote_subscriber = self.create_subscription(
            msg_type=AmazingQuote,
            topic='/amazing_quote',
```

(continues on next page)

(continued from previous page)

```
            callback=self.amazing_quote_subscriber_callback,
            qos_profile=1)
```

where the only difference with respect to the publisher is the third argument, namely `callback`, in which a method that receives a `msg_type` and returns nothing is expected. For example, the `amazing_quote_subscriber_callback`.

```
    def amazing_quote_subscriber_callback(self, msg: AmazingQuote):
        """Method that is periodically called by the timer."""
        self.get_logger().info(f"""
        I have received the most amazing of quotes.
        It says

               '{msg.quote}'

        And was thought by the following genius

            -- {msg.philosopher_name}

        This latest quote had the id={msg.id}.
        """)
```

That callback method will be automatically called by ROS2, as one of the tasks performed by `rclpy.spin(Node)`. Depending on the `qos_profile`, it will not necessarily be the latest message.

Note: The message will **ALWAYS** take some time between being published and being received by the subscriber. The speed in which that will happen will depend not only on this Node's `rclpy.spin()`, but also on the `rclpy.spin()` of the publisher node and the communication channel.

18.5 Update the `setup.py`

As we already learned in *Making ros2 run work*, we must adjust the `setup.py` to refer to the Nodes we just created.

`setup.py`

```
from setuptools import setup

package_name = 'python_package_that_uses_the_messages'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
         ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='murilo',
```

(continues on next page)

(continued from previous page)

```
    maintainer_email='murilomarinho@ieee.org',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'amazing_quote_publisher_node = python_package_that_uses_the_messages.
↪amazing_quote_publisher_node:main',
            'amazing_quote_subscriber_node = python_package_that_uses_the_messages.
↪amazing_quote_subscriber_node:main'
        ],
    },
)
```

18.6 Build and source

Before we proceed, let us build and source once.

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

18.7 Testing Publisher and Subscriber

Whenever we need to open two or more terminal windows, remember that *Terminator is life*.

Let us open two terminals.

In the first terminal, we run

```
ros2 run python_package_that_uses_the_messages amazing_quote_publisher_node
```

Nothing, in particular, should happen now. The publisher is sending messages through the specific topic we defined, but we need at least one subscriber to interact with those messages.

Hence, in the second terminal, we run

```
ros2 run python_package_that_uses_the_messages amazing_quote_subscriber_node
```

which outputs

```
[INFO] [1684215672.344532584] [amazing_quote_subscriber_node]:
     I have received the most amazing of quotes.
     It says

            'Use the force, Pikachu!'
```

(continues on next page)

(continued from previous page)

```
     And was thought by the following genius

         -- Uncle Ben

     This latest quote had the id=3.

[INFO] [1684215672.844618237] [amazing_quote_subscriber_node]:
     I have received the most amazing of quotes.
     It says

            'Use the force, Pikachu!'

     And was thought by the following genius

         -- Uncle Ben

     This latest quote had the id=4.

[INFO] [1684215673.344514856] [amazing_quote_subscriber_node]:
     I have received the most amazing of quotes.
     It says

            'Use the force, Pikachu!'

     And was thought by the following genius

         -- Uncle Ben

     This latest quote had the id=5.
```

Note: If there are any issues with either the publisher or the subscriber, this connection will not work. In the next section, we'll see strategies to help us troubleshoot and understand communication through topics.

Warning: Unless instructed otherwise, the publisher does **NOT** wait for a subscriber to connect before it starts publishing the messages. As shown in the case above, the first message we received started with *id=3*. If we delayed longer to start the publisher, we would have received later messages only.

Let's close each node with `CTRL+C` on each terminal before we proceed to the next tutorial.

CHAPTER

NINETEEN

INSPECTING TOPICS (ROS2 TOPIC)

ROS2 has a tool to help us inspect topics. This is used with considerable frequency in practice to troubleshoot and speed up the development of publishers and subscribers. As usual, we can get more information on this tool as follows.

```
ros2 topic -h
```

which outputs the detailed information of the tool, as shown below. In particular, the highlighted fields are used quite frequently in practice.

```
usage: ros2 topic [-h]
                  [--include-hidden-topics]
                  Call `ros2 topic <command>
                  -h` for more detailed usage.
                  ...

Various topic related sub-commands

options:
  -h, --help            show this help message
                        and exit
  --include-hidden-topics
                        Consider hidden topics
                        as well

Commands:
  bw     Display bandwidth used by topic
  delay  Display delay of topic from timestamp in header
  echo   Output messages from a topic
  find   Output a list of available topics of a given type
  hz     Print the average publishing rate to screen
  info   Print information about a topic
  list   Output a list of available topics
  pub    Publish a message to a topic
  type   Print a topic's type

  Call `ros2 topic <command> -h` for more detailed usage.
```

19.1 Start a publisher

During the development of a publisher, it is extremely useful to be able to check if topics are being properly made before we venture into making the subscribers. To see some of the tools for this job, we start by running the publisher Node we wrote in the last section.

> **Warning:** Be sure to terminate the Nodes we used in the past section before proceeding (e.g. with `CTRL+C`), otherwise, the output will look different from what is described here.

```
ros2 run python_package_that_uses_the_messages amazing_quote_publisher_node
```

19.2 Getting all topics with `ros2 topic list`

In particular, when there are many topics, it is difficult to remember every name. To see all currently active topics, we can run

```
ros2 topic list
```

which, in this case, outputs

```
/amazing_quote
/parameter_events
/rosout
```

showing, in particular, the `/amazing_quote` topic what we were looking for.

Hint: The **`ros2 topic info`** is one of the main tools to find out typos in the names of topics. For example, if there was a typo in our topic we might find, in fact, two topics being listed, when we only expected one. For instance,

```
/amazing_quote
/amazing_quotes
/parameter_events
/rosout
```

19.3 `grep` is your new best friend

Note: If you want more information on **`grep`**, check the Ubuntu Manpage

When the list of topics is too large, we can use **`grep`** to help filter the output. E.g.

```
ros2 topic list | grep quote
```

which outputs only the lines that contain `quote`, that is

```
/amazing_quote
```

19.4 Getting quick info with `ros2 topic info`

To get some quick information on a topic, we can run

```
ros2 topic info /amazing_quote
```

which outputs the message type and the number of publishers and subscribers connected to that topic

```
Type: package_with_interfaces/msg/AmazingQuote
Publisher count: 1
Subscription count: 0
```

19.5 Checking topic contents with `ros2 topic echo`

The **`ros2 topic echo`** is the main tool that we can use to inspect topic activity. We can check all the options of **`ros2 topic echo`** with the command below. The output is quite long so it's not replicated here.

```
ros2 topic echo -h
```

To inspect the topic whose name we already know, we run

```
ros2 topic echo /amazing_quote
```

which outputs the following

```
id: 6
quote: Use the force, Pikachu!
philosopher_name: Uncle Ben
---
id: 7
quote: Use the force, Pikachu!
philosopher_name: Uncle Ben
---
id: 8
quote: Use the force, Pikachu!
philosopher_name: Uncle Ben
---
id: 9
quote: Use the force, Pikachu!
philosopher_name: Uncle Ben
---
id: 10
quote: Use the force, Pikachu!
philosopher_name: Uncle Ben
---
id: 11
quote: Use the force, Pikachu!
```

(continues on next page)

(continued from previous page)

```
philosopher_name: Uncle Ben
---
```

19.6 `grep` is still your best friend

Whenever the topic is too crowded or the messages too fast, it might be difficult to pinpoint a single field we are looking for. In that case, **`grep`** can also help.

For example let us say that we want to see only the `id` fields of the messages. We can do

```
ros2 topic echo /amazing_quote | grep id
```

which will output only the lines with that pattern, e.g.

```
id: 1550
id: 1551
id: 1552
id: 1553
```

19.7 Measuring publishing frequency with `ros2 topic hz`

There are situations in which we are interested in knowing if the topics are receiving messages at an expected rate, without particular interest in the contents of the messages. We can do so with

```
ros2 topic hz /amazing_quote
```

which will output, after some time,

```
   WARNING: topic [/amazing_quote] does not appear to be published yet
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00007s window: 4
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00013s window: 7
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00011s window: 9
```

We must wait for a while until messages are received so that the tool can measure the frequency properly. You probably have noticed that the frequency measured by **`ros2 topic hz`** is compatible with the period of the `Timer` in our publisher Node.

19.8 Stop the publisher

Now we have exhausted all relevant tools that can give us information related to the publisher. Let us close the publisher with CTRL+C so that we can evaluate how these tools can help us analyze a subscriber.

19.9 Start the subscriber and get basic info

```
ros2 run python_package_that_uses_the_messages amazing_quote_subscriber_node
```

When only the subscriber is running, we can still get the basic info on the topic, e.g.

```
ros2 topic list
```

which also outputs

```
/amazing_quote
/parameter_events
/rosout
```

and

```
ros2 topic info /amazing_quote
```

which, differently from before, outputs

```
Type: package_with_interfaces/msg/AmazingQuote
Publisher count: 0
Subscription count: 1
```

19.10 Testing your subscribers with `ros2 topic pub`

To somewhat quickly evaluate a subscriber, we can use the **ros2 topic pub**. It allows us to publish messages to check the behavior of our subscribers.

In our case, we can send an **AmazingQuote** using YAML (YAML Ain't Markup Language) (More info). You can also refer to the YAML Cheat Sheet at QuickRef.ME.

```
ros2 topic pub /amazing_quote \
package_with_interfaces/msg/AmazingQuote \
'{
id: 1994,
quote: So you’re telling me there’s a chance,
philosopher_name: Lloyd
}'
```

Note: To improve readability, the command above uses the escape character \. You can see more on this at the bash docs. You can also refer to the **bash** Cheat Sheet at QuickRef.ME.

which will result in our subscriber outputting

```
[INFO] [1684222464.960446589] [amazing_quote_subscriber_node]:
        I have received the most amazing of quotes.
        It says

               'So you’re telling me there’s a chance'

        And was though by the following genius

            -- Lloyd

        This latest quote had the id=1994.

[INFO] [1684222465.953452826] [amazing_quote_subscriber_node]:
        I have received the most amazing of quotes.
        It says

               'So you’re telling me there’s a chance'

        And was though by the following genius

            -- Lloyd

        This latest quote had the id=1994.
```

For complicated messages, properly writing the message on the terminal can be a handful. In that case, it might be better to make a minimal script to test the subscriber instead. Refer to *Create the Node with a publisher*.

CHAPTER

TWENTY

AT YOUR SERVICE: SERVERS AND CLIENTS

Note: Except for the particulars of the setup.py file, the way that services in ROS2 work in Python, i.e. the explanation in this section, does not depend on **ament_python** or **ament_cmake**.

In some cases, we need means of communication in which each command has an associated response. That is where Services come into play.

20.1 Create the package

We start by creating a package to use the Service we first created in *The service file*.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create python_package_that_uses_the_services \
--build-type ament_python \
--dependencies rclpy package_with_interfaces
```

20.2 Overview

Before we start exploring the elements of the package, let us

1. Create the Node with a Service Server.
2. Create the Node with a Service Client.
3. Update the setup.py so that **ros2 run** finds these programs.

20.3 Create the Node with a Service Server

TL;DR Creating a service server

1. Add new dependencies to package.xml
2. Import new services from <package_name>.srv import <srv_name>
3. In a subclass of Node
    1. create a callback def callback(self, request, response):

    2. create a service server with `self.service_server = self.create_service(...)`

4. Add the new Node to `setup.py`

Let's start by creating a `what_is_the_point_service_server_node.py` in `~/ros2_tutorial_workspace/src/python_package_that_uses_the_services/python_package_that_uses_the_services` with the following contents

`what_is_the_point_service_server_node.py`

```
import random
from textwrap import dedent  # https://docs.python.org/3/library/textwrap.html#textwrap.
↪dedent

import rclpy
from rclpy.node import Node
from package_with_interfaces.srv import WhatIsThePoint


class WhatIsThePointServiceServerNode(Node):
    """A ROS2 Node with a Service Server for WhatIsThePoint."""

    def __init__(self):
        super().__init__('what_is_the_point_service_server')

        self.service_server = self.create_service(
            srv_type=WhatIsThePoint,
            srv_name='/what_is_the_point',
            callback=self.what_is_the_point_service_callback)

        self.service_server_call_count: int = 0

    def what_is_the_point_service_callback(self,
                                           request: WhatIsThePoint.Request,
                                           response: WhatIsThePoint.Response
                                           ) -> WhatIsThePoint.Response:
        """Analyses an AmazingQuote and returns what is the point.
           If the quote contains 'life', it returns a point whose sum of coordinates is␣
↪42.
           Otherwise, it returns a random point whose sum of coordinates is not 42.
        """

        # Generate the x,y,z of the point
        if "life" in request.quote.quote.lower():
            x: float = random.uniform(0, 42)
            y: float = random.uniform(0, 42 - x)
            z: float = 42 - (x + y)
        else:
            x: float = random.uniform(0, 100)
            y: float = random.uniform(0, 100)
            z: float = random.uniform(0, 100)
            if x + y + z == 42:  # So you're telling me there's a chance? Yes!
                x = x + 1  # Not anymore :(
```

(continues on next page)

(continued from previous page)

```
        # Assign to the response
        response.point.x = x
        response.point.y = y
        response.point.z = z

        # Increase the call count
        self.service_server_call_count = self.service_server_call_count + 1

        self.get_logger().info(dedent(f"""
            This is the call number {self.service_server_call_count} to this Service␣
↪Server.
            The analysis of the AmazingQuote below is complete.

                    {request.quote.quote}

            -- {request.quote.philosopher_name}

            The point has been sent back to the client.
        """))

        return response


def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        what_is_the_point_service_server_node = WhatIsThePointServiceServerNode()

        rclpy.spin(what_is_the_point_service_server_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

The code begins with an import to the service we created. No surprise here.

```
import random
from textwrap import dedent  # https://docs.python.org/3/library/textwrap.html#textwrap.
↪dedent

import rclpy
from rclpy.node import Node
```

(continues on next page)

(continued from previous page)

```
from package_with_interfaces.srv import WhatIsThePoint
```

The Service Server must be initialised with the `create_service()`, as follows, with parameters that should by now be quite obvious to us.

```
        self.service_server = self.create_service(
            srv_type=WhatIsThePoint,
            srv_name='/what_is_the_point',
            callback=self.what_is_the_point_service_callback)
```

The Service Server receives a `WhatIsThePoint.Request` and returns a `WhatIsThePoint.Response`.

```
    def what_is_the_point_service_callback(self,
                                           request: WhatIsThePoint.Request,
                                           response: WhatIsThePoint.Response
                                           ) -> WhatIsThePoint.Response:
        """Analyses an AmazingQuote and returns what is the point.
           If the quote contains 'life', it returns a point whose sum of coordinates is␣
↪42.
           Otherwise, it returns a random point whose sum of coordinates is not 42.
        """
```

> **Warning:** The API for the Service Server callback is a bit weird in that needs the `Response` as an argument. This API might change, but for now this is what we got.

We play around with the `WhatIsThePoint.Request` a bit and use that result to populate a `WhatIsThePoint.Response`, as follows

```
        # Assign to the response
        response.point.x = x
        response.point.y = y
        response.point.z = z
```

At the end of the callback, we must return that `WhatIsThePoint.Request`, like so

```
        return response
```

The Service Server was quite painless, but it doesn't do much. The Service Client might be a bit more on the painful side for the uninitiated.

20.4 Service Clients

ROS2 `rclpy` Service Clients are implemented using an `asyncio` logic (More info). In this tutorial, we briefly introduce unavoidable `async` concepts in *Python's asyncio*, but for any extra understanding, it's better to check the official documentation.

20.5 Create the Node with a Service Client (using a callback)

TL;DR Creating a Service Client (using a callback)

1. Add new dependencies to package.xml
2. Import new services from <package_name>.srv import <srv_name>
3. In a subclass of Node
    1. (*recommended*) wait for service to be available service_client.wait_for_service(...).
    2. (*if periodic*) add a Timer with a proper timer_callback()
    3. create a callback for the future def service_future_callback(self, future: Future):
    4. create a Service Client with self.service_client = self.create_client(...)
4. Add the new Node to setup.py

20.5.1 The Node

Note: This example deviates somewhat from what is done in the official examples. This implementation shown herein uses a callback and rclpy.spin(). It has many practical applications, but it's no *panacea*.

We start by adding a what_is_the_point_service_client_node.py at python_package_that_uses_the_services/python_package_that_uses_the_services with the following contents.

what_is_the_point_service_client_node.py

```python
import random
from textwrap import dedent  # https://docs.python.org/3/library/textwrap.html#textwrap.
↪dedent

import rclpy
from rclpy.task import Future
from rclpy.node import Node

from package_with_interfaces.srv import WhatIsThePoint


class WhatIsThePointServiceClientNode(Node):
    """A ROS2 Node with a Service Client for WhatIsThePoint."""

    def __init__(self):
        super().__init__('what_is_the_point_service_client')

        self.service_client = self.create_client(
            srv_type=WhatIsThePoint,
            srv_name='/what_is_the_point')

        while not self.service_client.wait_for_service(timeout_sec=1.0):
            self.get_logger().info(f'service {self.service_client.srv_name} not␣
```

(continues on next page)

(continued from previous page)

```
available, waiting...')

        self.future: Future = None

        timer_period: float = 0.5
        self.timer = self.create_timer(
            timer_period_sec=timer_period,
            callback=self.timer_callback)

    def timer_callback(self):
        """Method that is periodically called by the timer."""

        request = WhatIsThePoint.Request()
        if random.uniform(0, 1) < 0.5:
            request.quote.quote = "I wonder about the Ultimate Question of Life, the Universe, and Everything."
            request.quote.philosopher_name = "Creators of Deep Thought"
            request.quote.id = 1979
        else:
            request.quote.quote = """[...] your living... it is always potatoes. I dream of potatoes."""
            request.quote.philosopher_name = "a young Maltese potato farmer"
            request.quote.id = 2013

        if self.future is not None and not self.future.done():
            self.future.cancel()  # Cancel the future. The callback will be called with Future.result == None.
            self.get_logger().info("Service Future canceled. The Node took too long to process the service call."
                                   "Is the Service Server still alive?")
        self.future = self.service_client.call_async(request)
        self.future.add_done_callback(self.process_response)

    def process_response(self, future: Future):
        """Callback for the future, that will be called when it is done"""
        response = future.result()
        if response is not None:
            self.get_logger().info(dedent(f"""
                We have thus received the point of our quote.

                            {(response.point.x, response.point.y, response.point.z)}
            """))
        else:
            self.get_logger().info(dedent("""
                    The response was None. :(
            """))


def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
```

(continues on next page)

(continued from previous page)

```python
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        what_is_the_point_service_client_node = WhatIsThePointServiceClientNode()

        rclpy.spin(what_is_the_point_service_client_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

20.5.2 Imports

To have access to the service, we import it with `from <package>.srv import <Service>`.

```python
from package_with_interfaces.srv import WhatIsThePoint
```

20.5.3 Instantiate a Service Client

We instantiate a Service Client with `Node.create_client()`. The values of `srv_type` and `srv_name` must match the ones used in the Service Server.

```python
        self.service_client = self.create_client(
            srv_type=WhatIsThePoint,
            srv_name='/what_is_the_point')
```

20.5.4 (Recommended) Wait for the Service Server to be available

> **Warning:** The order of execution and speed of Nodes depend on a complicated web of relationships between ROS2, the operating system, and the workload of the machine. It would be naive to expect the server to always be active before the client, even if the server Node is started before the client Node.

In many cases, having the result of the service is of particular importance (hence the use of a service and not messages). In that case, we have to wait until `service_client.wait_for_service()`, as shown below.

```python
        while not self.service_client.wait_for_service(timeout_sec=1.0):
            self.get_logger().info(f'service {self.service_client.srv_name} not available, waiting...')
```

20.5.5 Instantiate a Future as a class attribute

As part of the async framework, we instantiate a Future (More info). In this example it is important to have it as an attribute of the class so that we do not lose the reference to it after the callback.

```
        self.future: Future = None
```

20.5.6 Instantiate a Timer

Whenever periodic work must be done, it is recommended to use a Timer, as we already learned in *Use a Timer for periodic work (when using rclpy.spin())*.

```
        timer_period: float = 0.5
        self.timer = self.create_timer(
            timer_period_sec=timer_period,
            callback=self.timer_callback)
```

The need for a callback for the Timer, should also be no surprise.

```
    def timer_callback(self):
        """Method that is periodically called by the timer."""
```

20.5.7 Service Clients use <srv>.Request()

Given that services work in a request-response model, the Service Client must instantiate a suitable <srv>.Request() and populate its fields before making the service call, as shown below. To make the example more interesting, it randomly switches between two possible quotes.

```
        request = WhatIsThePoint.Request()
        if random.uniform(0, 1) < 0.5:
            request.quote.quote = "I wonder about the Ultimate Question of Life, the Universe, and Everything."
            request.quote.philosopher_name = "Creators of Deep Thought"
            request.quote.id = 1979
        else:
            request.quote.quote = """[...] your living... it is always potatoes. I dream of potatoes."""
            request.quote.philosopher_name = "a young Maltese potato farmer"
            request.quote.id = 2013
```

20.5.8 Make service calls with call_async()

The async framework in ROS2 is based on Python's asyncio that we already saw in *Python's asyncio*.

Note: At first glance, it might feel that all this trouble to use async is unjustified. However, Nodes in practice will hardly ever do one service call and be done. Many Nodes in a complex system will have a composition of many service servers, service clients, publishers, and subscribers. Blocking the entire Node while it waits for the result of a service is, in most cases, a bad design.

The recommended way to call a service is through `call_async()`, which is the reason why we are working with `async` logic. In general, the result of `call_async()`, a `Future`, will not have the result of the service call at the next line of our program.

There are many ways to address the use of a `Future`. One of them, specially tailored to interface `async` with callback-based frameworks is the `Future.add_done_callback()`. If the `Future` is already done by the time we call `add_done_callback()`, it is supposed to call the callback for us.

The benefit of this is that the callback will not block our resources until the response is ready. When the response is ready, and the ROS2 executor gets to processing `Future` callbacks, our callback will be called *automagically*.

```
        if self.future is not None and not self.future.done():
            self.future.cancel()  # Cancel the future. The callback will be called with␣
↪Future.result == None.
            self.get_logger().info("Service Future canceled. The Node took too long to␣
↪process the service call."
                                   "Is the Service Server still alive?")
        self.future = self.service_client.call_async(request)
        self.future.add_done_callback(self.process_response)
```

Given that we are periodically calling the service, before replace the class `Future` with the next service call, we can check if the service call was done with `Future.done()`. If it is not done, we can use `Future.cancel()` so that our callback can handle this case as well. For instance, if the Service Server has been shutdown, the `Future` would never be done.

```
        if self.future is not None and not self.future.done():
            self.future.cancel()  # Cancel the future. The callback will be called with␣
↪Future.result == None.
            self.get_logger().info("Service Future canceled. The Node took too long to␣
↪process the service call."
                                   "Is the Service Server still alive?")
        self.future = self.service_client.call_async(request)
        self.future.add_done_callback(self.process_response)
```

20.5.9 The Future callback

The callback for the `Future` must receive a `Future` as an argument. Having it as an attribute of the Node's class allows us to access ROS2 method such as `get_logger()` and other contextual information.

The result of the `Future` is obtained using `Future.result()`. The response might be `None` in some cases, so we must check it before trying to use the result, otherwise we will get a nasty exception.

```
    def process_response(self, future: Future):
        """Callback for the future, that will be called when it is done"""
        response = future.result()
        if response is not None:
            self.get_logger().info(dedent(f"""
                We have thus received the point of our quote.

                            {(response.point.x, response.point.y, response.point.z)}
            """))
        else:
            self.get_logger().info(dedent("""
                    The response was None. :(
```

(continues on next page)

(continued from previous page)

```
            """))
```

20.6 Update the setup.py

As we already learned in *Making ros2 run work*, we must adjust the `setup.py` to refer to the Nodes we just created.

setup.py

```python
from setuptools import setup

package_name = 'python_package_that_uses_the_services'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='murilo',
    maintainer_email='murilomarinho@ieee.org',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'what_is_the_point_service_client_node = '
            'python_package_that_uses_the_services.what_is_the_point_service_client_
↪node:main',
            'what_is_the_point_service_server_node = '
            'python_package_that_uses_the_services.what_is_the_point_service_server_
↪node:main'
        ],
    },
)
```

20.7 Build and source

Before we proceed, let us build and source once.

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

20.8 Testing Service Server and Client

```
ros2 run python_package_that_uses_the_services what_is_the_point_service_client_node
```

when running the client Node, the server is still not active. In that case, the client node will keep waiting for it, as follows

```
[INFO] [1684293008.888276849] [what_is_the_point_service_client]: service /what_is_the_
↪point not available, waiting...
[INFO] [1684293009.890589539] [what_is_the_point_service_client]: service /what_is_the_
↪point not available, waiting...
[INFO] [1684293010.892778194] [what_is_the_point_service_client]: service /what_is_the_
↪point not available, waiting...
```

In another terminal, we run the **python_package_uses_the_service_node**, as follows

```
ros2 run python_package_that_uses_the_services what_is_the_point_service_server_node
```

The server Node will then output, periodically,

```
[INFO] [1684485151.608507798] [what_is_the_point_service_server]:
This is the call number 1 to this Service Server.
The analysis of the AmazingQuote below is complete.

        [...] your living... it is always potatoes. I dream of potatoes.

-- a young Maltese potato farmer

The point has been sent back to the client.

[INFO] [1684485152.092508332] [what_is_the_point_service_server]:
This is the call number 2 to this Service Server.
The analysis of the AmazingQuote below is complete.

        I wonder about the Ultimate Question of Life, the Universe, and Everything.

-- Creators of Deep Thought

The point has been sent back to the client.
```

(continues on next page)

(continued from previous page)

```
[INFO] [1684485152.592516148] [what_is_the_point_service_server]:
This is the call number 3 to this Service Server.
The analysis of the AmazingQuote below is complete.

        I wonder about the Ultimate Question of Life, the Universe, and Everything.

-- Creators of Deep Thought

The point has been sent back to the client.
```

and the client Node will output, periodically,

```
[INFO] [1684485151.609611689] [what_is_the_point_service_client]:
We have thus received the point of our quote.

            (18.199457100225292, 33.14595477433704, 52.65262570058381)

[INFO] [1684485152.093228181] [what_is_the_point_service_client]:
We have thus received the point of our quote.

            (11.17170193214362, 9.384897014549527, 21.443401053306854)

[INFO] [1684485152.593294259] [what_is_the_point_service_client]:
We have thus received the point of our quote.

            (16.58535176162403, 0.6180505400411676, 24.796597698334804)
```

CHAPTER TWENTYONE

INSPECTING SERVICES (ROS2 SERVICE)

ROS2 has a tool to help us inspect services. It is just as helpful as the tools for topics.

```
ros2 service -h
```

which outputs the detailed information of the tool, as shown below. In particular, the highlighted fields are used quite frequently in practice.

```
usage: ros2 service [-h] [--include-hidden-services]
                    Call `ros2 service <command> -h` for more
                    detailed usage. ...

Various service related sub-commands

options:
  -h, --help            show this help message and exit
  --include-hidden-services
                        Consider hidden services as well

Commands:
  call  Call a service
  find  Output a list of available services of a given type
  list  Output a list of available services
  type  Output a service's type

  Call `ros2 service <command> -h` for more detailed usage.
```

21.1 Start a service server

Similar to the discussion about topics, it is good to be able to test service servers without having to develop a complete service client. Let's start by running the service server we created just now.

> **Warning:** Be sure to terminate the Nodes we used in the past section before proceeding (e.g. with CTRL+C), otherwise, the output will look different from what is described here.

```
ros2 run python_package_that_uses_the_services what_is_the_point_service_server_node
```

21.2 Getting all services with `ros2 service list`

To see all currently active services, we run

```
ros2 service list
```

which, in this case, outputs

```
/what_is_the_point
/what_is_the_point_service_server/describe_parameters
/what_is_the_point_service_server/get_parameter_types
/what_is_the_point_service_server/get_parameters
/what_is_the_point_service_server/list_parameters
/what_is_the_point_service_server/set_parameters
/what_is_the_point_service_server/set_parameters_atomically
```

To everyone's surprise, there are a lot of services beyond the one we created. We can address those when we talk about ROS2 parameters, for now, we ignore them.

21.3 Testing your service servers with ros2 service call

Like the discussion about topics, ROS2 has a tool to call a service from the terminal, called **`ros2 service call`**. The service must be specified and an instance of its request must be written using YAML. Back to our example, we can do

```
ros2 service call /what_is_the_point \
package_with_interfaces/srv/WhatIsThePoint \
'{
quote: {
     id: 1994,
     quote: So you're telling me there's a chance,
     philosopher_name: Lloyd
     }
}'
```

which results in

```
waiting for service to become available...
requester: making request: package_with_interfaces.srv.WhatIsThePoint_
↪Request(quote=package_with_interfaces.msg.AmazingQuote(id=1994, quote='So you're␣
↪telling me there's a chance', philosopher_name='Lloyd'))

response:
package_with_interfaces.srv.WhatIsThePoint_Response(point=geometry_msgs.msg.Point(x=8.
↪327048266159165, y=95.97987946924988, z=67.03878311627777))
```

21.4 Testing your service clients???

To the best of my knowledge, there is no tool inside `ros2 service` to allow us to experiment with the service clients. For service clients, apparently, the only way to test them is to make a minimal service server to interact with them. We've already done that, so this topic ends here.

> **Warning:** This topic is under heavy construction. Don't forget your PPE (Personal Protective Equipment) if you're venturing forward.

CHAPTER

TWENTYTWO

PARAMETERS: CREATING CONFIGURABLE NODES

The Nodes we have made in the past few sections are interesting because they take advantage of the interprocess communication provided by ROS2.

Other capabilities of ROS2 that we must take advantage of are ROS2 parameters and ROS2 launch files. We can use them to modify the behavior of Nodes without having to modify their source code.

For Python users, that might sound less appealing than for users of compiled languages. However, users of your package might not want nor be able to modify the source code directly, if the package is installable or part of a larger system with multiple users.

22.1 Create the package

First, let us create an **ament_python** package that depends on our `packages_with_interfaces` and build from there.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create python_package_that_uses_parameters_and_launch_files \
--build-type ament_python \
--dependencies rclpy package_with_interfaces
```

22.2 Overview

Before we start exploring the elements of the package, let us

1. Create the Node with a configurable publisher using parameters, mostly as we saw in *Create the Node with a publisher*.
2. Create a launch file to configure the Node without modifying its source code.

22.3 Create the Node using parameters

TL;DR Using parameters in a Node

1. Declare the parameter with `Node.declare_parameter()`, usually in the class's `__init__`.
2. Get the parameter with `Node.get_parameter()` either once or continuously.

In this step, we'll work on this.

```
src/python_package_that_uses_parameters_and_launch_files
  └── python_package_that_uses_parameters_and_launch_files/
        └── __init__.py
        └── amazing_quote_configurable_publisher_node.py
```

For the sake of the example, let us suppose that we want to make an `AmazingQuote` publisher that is, now, configurable.

Let's start by creating an `amazing_quote_configurable_publisher_node.py` in `python_package_that_uses_parameters_and_launch_files/python_package_that_uses_parameters_and_launch_fil` with the following contents

`amazing_quote_configurable_publisher_node.py`

```python
import rclpy
from rclpy.node import Node
from package_with_interfaces.msg import AmazingQuote


class AmazingQuoteConfigurablePublisherNode(Node):
    """A configurable ROS2 Node that publishes a configurable amazing quote."""

    def __init__(self):
        super().__init__('amazing_quote_configurable_publisher_node')

        # Periodically-obtained parameters
        self.declare_parameter('quote', 'Use the force, Pikachu!')
        self.declare_parameter('philosopher_name', 'Uncle Ben')

        # One-off parameters
        self.declare_parameter('topic_name', 'amazing_quote')
        topic_name: str = self.get_parameter('topic_name').get_parameter_value().string_
↪value
        self.declare_parameter('period', 0.5)
        timer_period: float = self.get_parameter('period').get_parameter_value().double_
↪value

        self.configurable_amazing_quote_publisher = self.create_publisher(
            msg_type=AmazingQuote,
            topic=topic_name,
            qos_profile=1)

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

(continues on next page)

(continued from previous page)

```python

        self.incremental_id: int = 0

    def timer_callback(self):
        """Method that is periodically called by the timer."""

        quote: str = self.get_parameter('quote').get_parameter_value().string_value
        philosopher_name: str = self.get_parameter('philosopher_name').get_parameter_
↪value().string_value

        amazing_quote = AmazingQuote()
        amazing_quote.id = self.incremental_id
        amazing_quote.quote = quote
        amazing_quote.philosopher_name = philosopher_name

        self.configurable_amazing_quote_publisher.publish(amazing_quote)

        self.incremental_id = self.incremental_id + 1


def main(args=None):
    """
    The main function.
    :param args: Not used directly by the user, but used by ROS2 to configure
    certain aspects of the Node.
    """
    try:
        rclpy.init(args=args)

        amazing_quote_configurable_publisher_node =␣
↪AmazingQuoteConfigurablePublisherNode()

        rclpy.spin(amazing_quote_configurable_publisher_node)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(e)


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

22.4 Don't forget to declare the parameter!

Note: According to the official documentation, it is possible to work with undeclared parameters, but I recommend against for basic usage.

It's easy to forget it, but `Node.get_parameter()` will not work if the parameter was not first declared with `Node.declare_parameter()`. Don't forget it!

22.5 One-off parameters

For one-off parameters, we just get them once after declaring them. Because we're using those attributes directly in the `__init__` method, they are not made attributes of the class, but they could be.

```
        # One-off parameters
        self.declare_parameter('topic_name', 'amazing_quote')
        topic_name: str = self.get_parameter('topic_name').get_parameter_value().string_
↪value
        self.declare_parameter('period', 0.5)
        timer_period: float = self.get_parameter('period').get_parameter_value().double_
↪value

        self.configurable_amazing_quote_publisher = self.create_publisher(
            msg_type=AmazingQuote,
            topic=topic_name,
            qos_profile=1)

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

In this case, we're making the topic name and publication periodicity as one-off configurable parameters.

22.6 Continuously-obtained parameters

Note: According to the official documentation, it is possible to assign callbacks to manage changes in parameters. It is not the best-documented feature and has some caveats, so we will skip that for now.

For parameters that we obtain continuously through the lifetime of the Node, we can, for example, declare them in the `__init__` method, like so

```
        # Periodically-obtained parameters
        self.declare_parameter('quote', 'Use the force, Pikachu!')
        self.declare_parameter('philosopher_name', 'Uncle Ben')
```

then obtain them in another method, like so

```
    def timer_callback(self):
        """Method that is periodically called by the timer."""
```

(continues on next page)

(continued from previous page)

```
        quote: str = self.get_parameter('quote').get_parameter_value().string_value
        philosopher_name: str = self.get_parameter('philosopher_name').get_parameter_
↪value().string_value

        amazing_quote = AmazingQuote()
        amazing_quote.id = self.incremental_id
        amazing_quote.quote = quote
        amazing_quote.philosopher_name = philosopher_name

        self.configurable_amazing_quote_publisher.publish(amazing_quote)

        self.incremental_id = self.incremental_id + 1
```

In this example, we are making the `quote` and the `philosopher_name` as configurable parameters that can be changed continuously, during the lifetime of the Node. After they are changed, the node will publish a message with different contents.

22.7 Truly configurable: using `_launch.py` files

TL;DR Using launch files

1. (Once) Create a `launch` folder in the project.
2. Create the launch file named as `launch/<something>_launch.py`.
3. (Once) modify the `setup.py` to correctly install launch files.

Differently from ROS1, in ROS2 we can use Python launch files. They are quite powerful, well documented, and mentioned first in the official documentation, so we will use them instead of XML or YAML files.

22.8 (Once) create the `launch` folder

In this step, we'll work on this.

```
src/python_package_that_uses_parameters_and_launch_files
  └── python_package_that_uses_parameters_and_launch_files/
        └── __init__.py
        └── amazing_quote_configurable_publisher_node.py
  └── launch
```

Well, without further ado

```
cd ~/ros2_tutorial_workspace/src/python_package_that_uses_parameters_and_launch_files
mkdir launch
```

22.9 Create the `launch` file

In this step, we'll work on this.

```
src/python_package_that_uses_parameters_and_launch_files
  └── python_package_that_uses_parameters_and_launch_files/
        └── __init__.py
        └── amazing_quote_configurable_publisher_node.py
  └── launch
        └── peanut_butter_falcon_quote_publisher_launch.py
```

Suppose that we are tired of all the meme quotes and want to make our Node publish a truly inspirational quote. We start by making the launch file named `peanut_butter_falcon_quote_publisher_launch.py` within the `launch` folder we just created, with the following contents

`peanut_butter_falcon_quote_publisher_launch.py`

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


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='python_package_that_uses_parameters_and_launch_files',
            executable='amazing_quote_configurable_publisher_node',
            name='peanut_butter_falcon_quote_publisher_node',
            parameters=[{
                "topic_name": "truly_inspirational_quote",
                "period": 0.25,
                "quote": "Yeah, you're gonna die, it's a matter of time. That ain't the question. The question's, "
                         "whether they're gonna have a good story to tell about you when you're gone",
                "philosopher_name": "Tyler",
            }]
        )
    ])
```

We're relying on the `LaunchDescription`, which expects a list of `launch_ros.actions`.

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

When using a `launch_ros.actions.Node`, we need to define which `package` it belongs to and the `executable` which must match the name we set for the executable in the `setup.py`

```python
            package='python_package_that_uses_parameters_and_launch_files',
            executable='amazing_quote_configurable_publisher_node',
```

Besides the parameters, we can configure the name of the Node, such that each is unique

```
        name='peanut_butter_falcon_quote_publisher_node',
```

Finally, our parameters are defined using a dictionary within a list, namely

```
                "topic_name": "truly_inspirational_quote",
                "period": 0.25,
                "quote": "Yeah, you're gonna die, it's a matter of time. That ain't the␣
↪question. The question's, "
                         "whether they're gonna have a good story to tell about you when␣
↪you're gone",
                "philosopher_name": "Tyler",
```

22.10 The `setup.py`

In this step, we'll work on this.

```
src/python_package_that_uses_parameters_and_launch_files
  └── python_package_that_uses_parameters_and_launch_files/
        └── __init__.py
        └── amazing_quote_configurable_publisher_node.py
  └── launch
        └── peanut_butter_falcon_quote_publisher_launch.py
  setup.py
```

Modify the `setup.py` to look like this

`setup.py`

```python
import os
from glob import glob
from setuptools import setup

package_name = 'python_package_that_uses_parameters_and_launch_files'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
         ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch',
↪'*launch.[pxy][yma]*'))),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='murilo',
    maintainer_email='murilomarinho@ieee.org',
    description='TODO: Package description',
```

(continues on next page)

(continued from previous page)

```
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'amazing_quote_configurable_publisher_node = '
            'python_package_that_uses_parameters_and_launch_files.amazing_quote_configurable_publisher_node:main',
        ],
    },
)
```

We have already seen a `setup.py` so many times we're almost calling it Wilson. The only difference is emphasized above inside the `data_files`, which is the line that will specify that launch files will be installed as well. Notice that the `setup.py` looks for files with a specific pattern in the folder `launch`, so be sure that your launch files have the correct name otherwise they might not be installed as expected.

22.11 Build and source

Before we proceed, let us build and source once.

```
cd ~/ros2_tutorial_workspace
colcon build
source install/setup.bash
```

Note: If you don't remember why we're building with these commands, see *Always source after you build*.

Warning: This topic is under heavy construction. Don't forget your PPE if you're venturing forward.

CHAPTER TWENTYTHREE

LAUNCH CONFIGURABLE NODES (ROS2 LAUNCH)

ROS2 has a tool to interact with launch files called **ros2 launch**.

We can obtain more information on it with

```
ros2 launch -h
```

which returns

```
usage: ros2 launch [-h] [-n] [-d] [-p | -s] [-a]
                   [--launch-prefix LAUNCH_PREFIX]
                   [--launch-prefix-filter LAUNCH_PREFIX_FILTER]
                   package_name [launch_file_name] [launch_arguments ...]

Run a launch file

positional arguments:
  package_name          Name of the ROS package which contains the launch
                        file
  launch_file_name      Name of the launch file
  launch_arguments      Arguments to the launch file; '<name>:=<value>' (for
                        duplicates, last one wins)

options:
  -h, --help            show this help message and exit
  -n, --noninteractive  Run the launch system non-interactively, with no
                        terminal associated
  -d, --debug           Put the launch system in debug mode, provides more
                        verbose output.
  -p, --print, --print-description
                        Print the launch description to the console without
                        launching it.
  -s, --show-args, --show-arguments
                        Show arguments that may be given to the launch file.
  -a, --show-all-subprocesses-output
                        Show all launched subprocesses' output by overriding
                        their output configuration using the
                        OVERRIDE_LAUNCH_PROCESS_OUTPUT envvar.
  --launch-prefix LAUNCH_PREFIX
                        Prefix command, which should go before all
                        executables. Command must be wrapped in quotes if it
                        contains spaces (e.g. --launch-prefix 'xterm -e gdb
```

(continues on next page)

(continued from previous page)

```
                        -ex run --args').
  --launch-prefix-filter LAUNCH_PREFIX_FILTER
                        Regex pattern for filtering which executables the
                        --launch-prefix is applied to by matching the
                        executable name.
```

Despite the large number of possible options, there are no notable examples of options that are of particular use to us right now.

We can call our Node, configured with our launch file, with

```
ros2 launch python_package_that_uses_parameters_and_launch_files peanut_butter_falcon_
↪quote_publisher_launch.py
```

which returns

```
[INFO] [launch]: All log files can be found below /home/murilo/.ros/log/2023-06-30-17-00-
↪07-522194-murilos-toaster-2963
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [amazing_quote_configurable_publisher_node-1]: process started with pid [2964]
```

showing that the launch was successful.

IN ANOTHER TERMINAL we run

```
ros2 topic echo /truly_inspirational_quote
```

resulting in something similar to

```
id: 301
quote: Yeah, you're gonna die, it's a matter of time. That ain't the question. The␣
↪question's, whether they're gonna have a good story ...
philosopher_name: Tyler
---
id: 302
quote: Yeah, you're gonna die, it's a matter of time. That ain't the question. The␣
↪question's, whether they're gonna have a good story ...
philosopher_name: Tyler
---
id: 303
quote: Yeah, you're gonna die, it's a matter of time. That ain't the question. The␣
↪question's, whether they're gonna have a good story ...
philosopher_name: Tyler
---
```

And there you have it. Feeling inspired yet?

> **Warning:** This topic is under heavy construction. Don't forget your PPE if you're venturing forward.

CHAPTER TWENTYFOUR

INSPECTING PARAMETERS (ROS2 PARAM)

ROS2 has a tool to interact with launch files called **ros2 param**.

We can obtain more information on it with

```
ros2 param -h
```

which returns

```
usage: ros2 param [-h] Call `ros2 param <command> -h` for more detailed usage. ...

Various param related sub-commands

options:
  -h, --help            show this help message and exit

Commands:
  delete    Delete parameter
  describe  Show descriptive information about declared parameters
  dump      Dump the parameters of a node to a yaml file
  get       Get parameter
  list      Output a list of available parameters
  load      Load parameter file for a node
  set       Set parameter

  Call `ros2 param <command> -h` for more detailed usage.
```

Note: By the time you try this out, the documentation of **ros2 param dump** might have changed. See ros2/ros2cli/#835.

As shown in the emphasized lines above, the **ros2 param** tool has a large number of useful commands to interact with parameters.

24.1 Launching the Node with parameters

Hint: If you left the Node running from the last section, just keep it that way and skip this.

```
ros2 launch \
python_package_that_uses_parameters_and_launch_files \
peanut_butter_falcon_quote_publisher_launch.py
```

24.2 List-up parameters with `ros2 param list`

Hint: Remember that *grep is your new best friend*.

Similar to other ROS2 commands, we can get a list of currently loaded parameters with

```
ros2 param list
```

which returns a well organized list showing the parameters of each active Node

```
/peanut_butter_falcon_quote_publisher_node:
  period
  philosopher_name
  quote
  topic_name
  use_sim_time
```

24.3 Obtain parameters with `ros2 param get`

To obtain the value of a parameter, we can do as follows

```
ros2 param get \
/peanut_butter_falcon_quote_publisher_node \
quote
```

which will return the current value of the parameter, in this case, the initial value we set in the launch file

```
String value is: Yeah, you're gonna die, it's a matter of time. That ain't the question.␣
↪The question's, whether they're gonna have a good story to tell about you when you're␣
↪gone
```

24.4 Let's check the output of the Node

Hint: If you left **ros2 topic echo** running from the last section, just keep it that way and skip this.

Before the next step, as we did in the past section, we do, **IN ANOTHER TERMINAL WINDOW**

```
ros2 topic echo /truly_inspirational_quote
```

24.5 Assign values to parameters with ros2 param set

For testing and regular usage, setting parameters from the command line is extremely helpful. Similar to how we are able to publish messages to topics using a ROS2 tool, we can set a parameter with the following syntax

```
ros2 param set \
/peanut_butter_falcon_quote_publisher_node \
quote \
"You got a good-guy heart. You can't do shit about it, that's just who you are. You're a␣
↪hero."
```

If everything is correct, we'll get

```
Set parameter successful
```

Changing parameters is not instantaneous and, after the change becomes visible in the Node, our Node might have to loop once before it updates itself. We will be able to see that change as follows in the terminal window running **ros2 topic echo**

```
id: 2220
quote: Yeah, you're gonna die, it's a matter of time. That ain't the question. The␣
↪question's, whether they're gonna have a good story ...
philosopher_name: Tyler
---
id: 2221
quote: You got a good-guy heart. You can't do shit about it, that's just who you are. You
↪'re a hero.
philosopher_name: Tyler
---
id: 2222
quote: You got a good-guy heart. You can't do shit about it, that's just who you are. You
↪'re a hero.
philosopher_name: Tyler
---
id: 2223
quote: You got a good-guy heart. You can't do shit about it, that's just who you are. You
↪'re a hero.
philosopher_name: Tyler
---
id: 2224
quote: You got a good-guy heart. You can't do shit about it, that's just who you are. You
```

(continues on next page)

(continued from previous page)

```
↪'re a hero.
philosopher_name: Tyler
```

24.6 Save parameters to a file with `ros2 param dump`

> **Warning:** At the time I was writing this part of the tutorial, the description of **`ros2 param dump`** was outdated. By the time you try this out, it might have been corrected. See ros2/ros2cli/#836 for more info.

Words are sometimes little happy accidents. This usage of the word dump has no relation whatsoever to, for example, Peter got dumped by Sarah and went to Hawaii. Dump files are usually related to crashes and unresponsive programs, so this name puzzles me since ROS: the first.

While we wait for someone to come and correct me on my claims above, just think about this as a weird name for **`ros2 param print_to_screen_as_yaml`**. It prints the parameters in the terminal with a YAML file format. It is nice because it gives a bit more info than **`ros2 param list`**, but not so useful as-is. The trick is that we can put all that nicely formatted content into a file with

```
cd ~/ros2_tutorial_workspace/src
ros2 param dump \
/peanut_butter_falcon_quote_publisher_node \
> peanut_butter_falcon_quote_publisher_node.yaml
```

where we are using the > (see *bash redirections*) to overwrite the contents of the `peanut_butter_falcon_quote_publisher_node.yaml` file with the output of **`ros2 param dump`**, so be careful not to overwrite your precious files by mistake.

We can inspect the contents of the file with

```
cat peanut_butter_falcon_quote_publisher_node.yaml
```

which outputs

```
/peanut_butter_falcon_quote_publisher_node:
  ros__parameters:
    period: 0.25
    philosopher_name: Tyler
    quote: Yeah, you're gonna die, it's a matter of time. That ain't the question.
      The question's, whether they're gonna have a good story to tell about you when
      you're gone
    topic_name: truly_inspirational_quote
    use_sim_time: false
```

24.7 Load parameters from a file with `ros2 param load`

Warning: To proceed, end the `peanut_butter_falcon_quote_publisher_node` Node with CTRL+C.

As in the prior step, suppose that we have a file `peanut_butter_falcon_quote_publisher_node.yaml` with the parameters we love the most. What we can do with **`ros2 param load`** is load that file. Nicely predictable and understandable naming convention.

We can start the Node with the launch file

```
ros2 launch python_package_that_uses_parameters_and_launch_files \
peanut_butter_falcon_quote_publisher_launch.py
```

which, at the beginning, will have the parameters set in the `_launch.py`. We can then

```
cd ~/ros2_tutorial_workspace/src
ros2 param load \
/peanut_butter_falcon_quote_publisher_node \
peanut_butter_falcon_quote_publisher_node.yaml
```

which will return

```
Set parameter period successful
Set parameter philosopher_name successful
Set parameter quote successful
Set parameter topic_name successful
Set parameter use_sim_time successful
```

indicating that all parameters defined in the YAML were successfully loaded.

CHAPTER

TWENTYFIVE

FORBIDDEN TOPICS

Warning: Scary things are out there.

25.1 Doing all that C++ stuff with `ament_cmake`

25.1.1 Using this section

For the Python version of this tutorial, we held hands and walked slowly into the sunset while sipping some affordable but tasty wine.

For the **`ament_cmake`** version of this tutorial, I'll suppose you know all that and throw in extra info that I suppose is useful. No hand-holding anymore.

Creating C++ Nodes (for `ament_cmake`)

The C++ binary block for `ament_cmake`

TL;DR

When adding a new Node in an existing `CMakeLists.txt`, you might benefit from using the following template.

Remember to:

1. Add **ALL** dependencies (including ROS2 ones) with `find_package`, if applicable.

   ```
   # find dependencies
   find_package(ament_cmake REQUIRED)
   find_package(rclcpp REQUIRED)
   ```

2. Change `print_forever_node` to the name of your Node.
3. Add all source files to `add_executable`.
4. Add all ROS2 dependencies of this binary to `ament_target_dependencies`.
5. Add any other (**NOT ROS2**) libraries to `target_link_libraries`.

```
############################
# CPP Binary Block [BEGIN] #
# vvvvvvvvvvvvvvvvvvvvvvvv #
# https://ros2-tutorial.readthedocs.io/en/latest/
# While we cant use blocks https://cmake.org/cmake/help/latest/command/block.html
↪#command:block
# we use set--unset
set(RCLCPP_LOCAL_BINARY_NAME print_forever_node)

add_executable(${RCLCPP_LOCAL_BINARY_NAME}
    src/print_forever_node_main.cpp
    src/print_forever_node.cpp

    )

ament_target_dependencies(${RCLCPP_LOCAL_BINARY_NAME}
    rclcpp

    )

target_link_libraries(${RCLCPP_LOCAL_BINARY_NAME}

    )

target_include_directories(${RCLCPP_LOCAL_BINARY_NAME} PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>)

target_compile_features(${RCLCPP_LOCAL_BINARY_NAME} PUBLIC c_std_99 cxx_std_17)

install(TARGETS ${RCLCPP_LOCAL_BINARY_NAME}
    DESTINATION lib/${PROJECT_NAME})

unset(RCLCPP_LOCAL_BINARY_NAME)
# ^^^^^^^^^^^^^^^^^^^^^^ #
# CPP Binary Block [END] #
##########################
```

Create the package

Warning: We'll skip using the `--node-name` option to create the Node template, because, currently, it generates a Node and a `CMakeLists.txt` different from my advice.

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create cpp_package_with_a_node \
--build-type ament_cmake \
--dependencies rclcpp
```

which outputs

ros2 pkg create output

```
going to create a new package
package name: cpp_package_with_a_node
destination directory: /home/murilo/ROS2_Tutorial/ros2_tutorial_workspace/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['murilo <murilomarinho@ieee.org>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: ['rclcpp']
creating folder ./cpp_package_with_a_node
creating ./cpp_package_with_a_node/package.xml
creating source and include folder
creating folder ./cpp_package_with_a_node/src
creating folder ./cpp_package_with_a_node/include/cpp_package_with_a_node
creating ./cpp_package_with_a_node/CMakeLists.txt

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the␣
↪package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

Package-related sources

In this step, we'll work on these.

```
cpp_package_with_a_node
├── CMakeLists.txt
├── include
│   └── cpp_package_with_a_node
│       └── .placeholder
├── package.xml
└── src
    ├── print_forever_node.cpp
    ├── print_forever_node.hpp
    └── print_forever_node_main.cpp
```

The files already exist, we just need to modify them as follows

package.xml

The package.xml works the same way as in **ament_python**, with the exception of the two lines about **ament_cmake** shown below.

package.xml

```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>cpp_package_with_a_node</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="murilomarinho@ieee.org">murilo</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

CMakeLists.txt

A *one-size-fits-most* solution is shown below. For each new Node we add a block to the CMakeLists.txt with the following format.

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.8)
project(cpp_package_with_a_node)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)

############################
# CPP Binary Block [BEGIN] #
# vvvvvvvvvvvvvvvvvvvvvvvv #
# https://ros2-tutorial.readthedocs.io/en/latest/
# While we cant use blocks https://cmake.org/cmake/help/latest/command/block.html#command:block
```

(continues on next page)

(continued from previous page)

```
# we use set--unset
set(RCLCPP_LOCAL_BINARY_NAME print_forever_node)

add_executable(${RCLCPP_LOCAL_BINARY_NAME}
    src/print_forever_node_main.cpp
    src/print_forever_node.cpp

    )

ament_target_dependencies(${RCLCPP_LOCAL_BINARY_NAME}
    rclcpp

    )

target_link_libraries(${RCLCPP_LOCAL_BINARY_NAME}

    )

target_include_directories(${RCLCPP_LOCAL_BINARY_NAME} PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>)

target_compile_features(${RCLCPP_LOCAL_BINARY_NAME} PUBLIC c_std_99 cxx_std_17)

install(TARGETS ${RCLCPP_LOCAL_BINARY_NAME}
    DESTINATION lib/${PROJECT_NAME})

unset(RCLCPP_LOCAL_BINARY_NAME)
# ^^^^^^^^^^^^^^^^^^^^^^ #
# CPP Binary Block [END] #
##########################

if(BUILD_TESTING)
    find_package(ament_lint_auto REQUIRED)
    # the following line skips the linter which checks for copyrights
    # comment the line when a copyright and license is added to all source files
    set(ament_cmake_copyright_FOUND TRUE)
    # the following line skips cpplint (only works in a git repo)
    # comment the line when this package is in a git repo and when
    # a copyright and license is added to all source files
    set(ament_cmake_cpplint_FOUND TRUE)
    ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Making C++ ROS2 Nodes

(Murilo's) rclcpp best practices

For each new C++ Node, we make three files following the style below.

For a Node called `print_forever_node` we have

1. `src/print_forever_node.hpp` with the Node's class definition. In general, this is not exported to other packages, so it should not be in the package's `include` folder.
2. `src/print_forever_node.cpp` with the Node's class implementation.
3. `src/print_forever_node_main.cpp` with the Node's main function implementation.

In this step, we'll work on these.

```
cpp_package_with_a_node
├── CMakeLists.txt
├── include
│   └── cpp_package_with_a_node
│       └── .placeholder
├── package.xml
└── src
    ├── print_forever_node.cpp
    ├── print_forever_node.hpp
    └── print_forever_node_main.cpp
```

These files do not exists, so we'll create them.

folder

```
cd ~/ros2_tutorial_workspace/src/cpp_package_with_a_node
mkdir src
```

src/..._node.hpp

Similar to what we did in Python, we inherit from `rclcpp::Node`. Whatever is different is owing to differences in languages.

`print_forever_node.hpp`

```cpp
#pragma once

#include <memory>
#include <rclcpp/rclcpp.hpp>

/**
 * @brief A ROS2 Node that prints to the console periodically, but in C++.
 */
class PrintForeverNode: public rclcpp::Node
```

(continues on next page)

(continued from previous page)

```cpp
{
private:
    double timer_period_;
    int print_count_;
    //also equivalent to rclcpp::TimerBase::SharedPtr
    std::shared_ptr<rclcpp::TimerBase> timer_;

    void _timer_callback();
public:
    PrintForeverNode();

};
```

src/…_node.cpp

The implementation has nothing special, just don't forget to initialize the parent class, `rclcpp::Node`, with the name of the node.

print_forever_node.cpp

```cpp
#include "print_forever_node.hpp"

/**
 * @brief PrintForeverNode::PrintForeverNode Default constructor.
 */
PrintForeverNode::PrintForeverNode():
    rclcpp::Node("print_forever_cpp"),
    timer_period_(0.5),
    print_count_(0)
{
    //(Smart) pointers at the one thing that it doesn't matter much if they are not initialized in the member initializer list
    //and this is a bit more readable.
    timer_ = create_wall_timer(
                std::chrono::milliseconds(long(timer_period_*1e3)),
                std::bind(&PrintForeverNode::_timer_callback, this) //Note here the use of std::bind to build a single argument
                );
}

/**
 * @brief PrintForeverNode::_timer_callback periodically prints class info using RCLCPP_INFO.
 */
void PrintForeverNode::_timer_callback()
{
    RCLCPP_INFO_STREAM(get_logger(),
                       std::string("Printed ") +
                       std::to_string(print_count_) +
                       std::string(" times.")
                       );
```

(continues on next page)

(continued from previous page)

```
    print_count_++;
}
```

src/…_main.cpp

Given that we are using rclcpp::spin(), there is nothing special here either. Just remember to not mess up the std::make_shared and always use perfect forwarding. See Perfect forwarding. The rclcpp::spin() handles the SIGINT when we, for example, press CTRL+C on the terminal. It is not perfect, but it does the trick for simple nodes like this one.

print_forever_node_main.cpp

```cpp
#include <rclcpp/rclcpp.hpp>

#include "print_forever_node.hpp"

int main(int argc, char** argv)
{
    rclcpp::init(argc,argv);

    try
    {
        auto node = std::make_shared<PrintForeverNode>();

        rclcpp::spin(node);
    }
    catch (const std::exception& e)
    {
        std::cerr << std::string("::Exception::") << e.what();
    }

    return 0;
}
```

Add a .placeholder if your include/<PACKAGE_NAME> is empty

> **Warning:** If you don't do this and add this package as a git repository without any files on the include/, **CMake** might return with an error when trying to compile your package.

```
cpp_package_with_a_node
├── CMakeLists.txt
├── include
│   └── cpp_package_with_a_node
│       └── .placeholder
├── package.xml
└── src
    ├── print_forever_node.cpp
```

(continues on next page)

(continued from previous page)

```
    ├── print_forever_node.hpp
    └── print_forever_node_main.cpp
```

Empty directories will not be tracked by git. A file has to be added to the index. We can create an empty file in the `include` folder as follows

```
cd ~/ros2_tutorial_workspace/src/cpp_package_with_a_node/src
touch include/cpp_package_with_a_node/.placeholder
```

Running a C++ Node

As simple as it has always been, see *Running a node (ros2 run)*.

```
ros2 run cpp_package_with_a_node print_forever_node
```

which returns

```
[INFO] [1688620414.406930812] [print_forever_node]: Printed 0 times.
[INFO] [1688620414.906890884] [print_forever_node]: Printed 1 times.
[INFO] [1688620415.406907619] [print_forever_node]: Printed 2 times.
[INFO] [1688620415.906881003] [print_forever_node]: Printed 3 times.
[INFO] [1688620416.406900108] [print_forever_node]: Printed 4 times.
[INFO] [1688620416.906886691] [print_forever_node]: Printed 5 times.
[INFO] [1688620417.406881803] [print_forever_node]: Printed 6 times.
[INFO] [1688620417.906858551] [print_forever_node]: Printed 7 times.
[INFO] [1688620418.406894922] [print_forever_node]: Printed 8 times.
```

and we'll use `CTRL+C` to stop the node, resulting in

```
[INFO] [1688620418.725674401] [rclcpp]: signal_handler(signum=2)
```

Creating C++ Libraries (for `ament_cmake`)

The C++ library block for `ament_cmake`

TL;DR

When your project exports a library, you might benefit from using the following template. Note that there is, in general, no reason to define multiple libraries. A single shared library can hold all the content that you want to export from a package, hence the library named `${PROJECT_NAME}`.

Remember to

1. Add all exported headers to `include/<PACKAGE_NAME>` otherwise other packages cannot see it.
2. Add all source files of the library to `add_library`.
3. Add all ROS2 dependencies of the library to `ament_target_dependencies`.
4. Add **ALL** dependencies for which you used `find_package` to `ament_export_dependencies`, otherwise dependencies might become complex for projects that use your library.

5. Add any other (**NOT ROS2**) libraries to `target_link_libraries`.

```
####################################
# CPP Shared Library Block [BEGIN] #
# vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv #
# https://ros2-tutorial.readthedocs.io/en/latest/
# The most common use case is to merge everything you need to export
# into the same shared library called ${PROJECT_NAME}.
add_library(${PROJECT_NAME} SHARED
    src/sample_class.cpp

    )

ament_target_dependencies(${PROJECT_NAME}
    rclcpp

    )

target_include_directories(${PROJECT_NAME}
    PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>)

ament_export_targets(export_${PROJECT_NAME} HAS_LIBRARY_TARGET)
ament_export_dependencies(
    rclcpp
    Eigen3
    Qt5Core

    )

target_link_libraries(${PROJECT_NAME}
    Qt5::Core

    )

install(
    DIRECTORY include/
    DESTINATION include
    )

install(
    TARGETS ${PROJECT_NAME}
    EXPORT export_${PROJECT_NAME}
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    RUNTIME DESTINATION bin
    INCLUDES DESTINATION include
    )
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ #
# CPP Shared Library Block [END] #
##################################
```

The base package can be created with

```
cd ~/ros2_tutorial_workspace/src
ros2 pkg create cpp_package_with_a_library \
--build-type ament_cmake \
--dependencies rclcpp
```

resulting in the following output

ros2 pkg create output

```
ros2 pkg create cpp_package_with_a_library \
--build-type ament_cmake \
--dependencies rclcpp
going to create a new package
package name: cpp_package_with_a_library
destination directory: /home/murilo/ROS2_Tutorial/ros2_tutorial_workspace/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['murilo <murilomarinho@ieee.org>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: ['rclcpp']
creating folder ./cpp_package_with_a_library
creating ./cpp_package_with_a_library/package.xml
creating source and include folder
creating folder ./cpp_package_with_a_library/src
creating folder ./cpp_package_with_a_library/include/cpp_package_with_a_library
creating ./cpp_package_with_a_library/CMakeLists.txt

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the␣
↪package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

Package-related sources

In this step, we'll work on these.

```
cpp_package_with_a_library
├── CMakeLists.txt
├── include
│   └── cpp_package_with_a_library
│       └── sample_class.hpp
├── package.xml
└── src
    ├── sample_class.cpp
    ├── sample_class_local_node.cpp
    ├── sample_class_local_node.hpp
    └── sample_class_local_node_main.cpp
```

The files already exist, we just need to modify them as follows

package.xml

Nothing new here.

package.xml

```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>cpp_package_with_a_library</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="murilomarinho@ieee.org">murilo</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

CMakeLists.txt

A *one-size-fits-most* solution is shown below. We don't need to add multiple libraries, so a single library can hold all the content you might want to export. The user of the library will see it nicely split by your header files, so it will be as neat as you make them.

Note that, because the local Node depends on the library being exported by this project, it needs to explicitly link to it.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(cpp_package_with_a_library)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(Eigen3 REQUIRED)
find_package(Qt5Core REQUIRED)

####################################
# CPP Shared Library Block [BEGIN] #
# vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv #
# https://ros2-tutorial.readthedocs.io/en/latest/
# The most common use case is to merge everything you need to export
# into the same shared library called ${PROJECT_NAME}.
add_library(${PROJECT_NAME} SHARED
    src/sample_class.cpp

    )

ament_target_dependencies(${PROJECT_NAME}
    rclcpp

    )

target_include_directories(${PROJECT_NAME}
    PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>)

ament_export_targets(export_${PROJECT_NAME} HAS_LIBRARY_TARGET)
ament_export_dependencies(
    rclcpp
    Eigen3
    Qt5Core

    )

target_link_libraries(${PROJECT_NAME}
    Qt5::Core
```

(continues on next page)

(continued from previous page)

```
    )

install(
    DIRECTORY include/
    DESTINATION include
    )

install(
    TARGETS ${PROJECT_NAME}
    EXPORT export_${PROJECT_NAME}
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    RUNTIME DESTINATION bin
    INCLUDES DESTINATION include
    )
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ #
# CPP Shared Library Block [END] #
##################################

############################
# CPP Binary Block [BEGIN] #
# vvvvvvvvvvvvvvvvvvvvvvvv #
# https://ros2-tutorial.readthedocs.io/en/latest/
# While we cant use blocks https://cmake.org/cmake/help/latest/command/block.html
↪#command:block
# we use set--unset
set(RCLCPP_LOCAL_BINARY_NAME sample_class_local_node)

add_executable(${RCLCPP_LOCAL_BINARY_NAME}
    src/sample_class_local_node_main.cpp
    src/sample_class_local_node.cpp
    src/sample_class.cpp

    )

ament_target_dependencies(${RCLCPP_LOCAL_BINARY_NAME}
    rclcpp

    )

target_link_libraries(${RCLCPP_LOCAL_BINARY_NAME}
    ${PROJECT_NAME}

    )

target_include_directories(${RCLCPP_LOCAL_BINARY_NAME} PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>)

target_compile_features(${RCLCPP_LOCAL_BINARY_NAME} PUBLIC c_std_99 cxx_std_17)

```

(continues on next page)

(continued from previous page)

```
install(TARGETS ${RCLCPP_LOCAL_BINARY_NAME}
    DESTINATION lib/${PROJECT_NAME})

unset(RCLCPP_LOCAL_BINARY_NAME)
# ^^^^^^^^^^^^^^^^^^^^^^ #
# CPP Binary Block [END] #
##########################

if(BUILD_TESTING)
    find_package(ament_lint_auto REQUIRED)
    # the following line skips the linter which checks for copyrights
    # comment the line when a copyright and license is added to all source files
    set(ament_cmake_copyright_FOUND TRUE)
    # the following line skips cpplint (only works in a git repo)
    # comment the line when this package is in a git repo and when
    # a copyright and license is added to all source files
    set(ament_cmake_cpplint_FOUND TRUE)
    ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Library sources

In this step, we'll work on these.

```
cpp_package_with_a_library
├── CMakeLists.txt
├── include
│   └── cpp_package_with_a_library
│       └── sample_class.hpp
├── package.xml
└── src
    ├── sample_class.cpp
    ├── sample_class_local_node.cpp
    ├── sample_class_local_node.hpp
    └── sample_class_local_node_main.cpp
```

sample_class.hpp

A class that does a bunch of nothing, but that depends on Eigen3 and Qt, as an example.

`sample_class.hpp`

```
#pragma once

#include <ostream>

```

(continues on next page)

(continued from previous page)

```cpp
#include <QString>
#include <eigen3/Eigen/Dense>

class SampleClass
{
private:
    int a_private_member_;
    const QString a_private_qt_string_;
    const Eigen::MatrixXd a_private_eigen3_matrix_;

public:
    SampleClass();

    int get_a_private_member() const;
    void set_a_private_member(int value);
    std::string to_string() const;

    static double sum_of_squares(const double&a, const double& b);
};

std::ostream& operator<<(std::ostream& os, const SampleClass& sc);
```

sample_class.cpp

sample_class.cpp

```cpp
#include <cpp_package_with_a_library/sample_class.hpp>



/**
 * @brief SampleClass::SampleClass the default constructor.
 */
SampleClass::SampleClass():
    a_private_qt_string_("I am a QString"),
    a_private_eigen3_matrix_((Eigen::Matrix2d() << 1,2,3,4).finished())
{

}

/**
 * @brief SampleClass::get_a_private_member.
 * @return an int with the value of a_private_member_.
 */
int SampleClass::get_a_private_member() const
{
    return a_private_member_;
}

/**
 * @brief SampleClass::set_a_private_member.
```

(continues on next page)

(continued from previous page)

```cpp
 * @param value The new value for a_private_member_.
 */
void SampleClass::set_a_private_member(int value)
{
    a_private_member_ = value;
}

/**
 * @brief SampleClass::sum_of_squares.
 * @param a The first number.
 * @param b The second number.
 * @return a*a + 2*a*b + b*b.
 */
double SampleClass::sum_of_squares(const double &a, const double &b)
{
    return a*a + 2*a*b + b*b;
}

/**
 * @brief SampleClass::to_string converts a SampleClass to a std::string representation.
 * @return a pretty(-ish) std::string representation of the object.
 */
std::string SampleClass::to_string() const
{
    std::stringstream ss;
    ss << "Sample_Class:: " << std::endl <<
          "a_private_member_ = "        << std::to_string(a_private_member_) << std::endl <<
          "a_private_qt_string_ = "     << a_private_qt_string_.toStdString() << std::endl <<
          "a_private_eigen3_matrix_ = " << a_private_eigen3_matrix_ << std::endl;
    return ss.str();
}

/**
 * @brief operator << the stream operator for SampleClass objects.
 * @param [in/out] the std::ostream to be modified.
 * @param [in] sc the SampleClass whose representation is to be streamed.
 * @return the modified os with the added SampleClass string representation.
 * @see SampleClass::to_string().
 */
std::ostream &operator<<(std::ostream &os, const SampleClass &sc)
{
    return os << sc.to_string();
}
```

Sources for a local node that uses the library

In this step, we'll work on these.

```
cpp_package_with_a_library
├── CMakeLists.txt
├── include
│   └── cpp_package_with_a_library
│       └── sample_class.hpp
├── package.xml
└── src
    ├── sample_class.cpp
    ├── sample_class_local_node.cpp
    ├── sample_class_local_node.hpp
    └── sample_class_local_node_main.cpp
```

Just in case you need to have a node, in the same package, that also uses the library exported by this package. Nothing too far from what we have already done.

sample_class_local_node.cpp

sample_class.cpp

```cpp
#include "sample_class_local_node.hpp"

/**
 * @brief SampleClassLocalNode::SampleClassLocalNode Default constructor.
 */
SampleClassLocalNode::SampleClassLocalNode():
    rclcpp::Node("sample_class_local_node"),
    timer_period_(0.5),
    print_count_(0)
{
    timer_ = create_wall_timer(
                std::chrono::milliseconds(long(timer_period_*1e3)),
                std::bind(&SampleClassLocalNode::_timer_callback, this)
                );
}

/**
 * @brief SampleClassLocalNode::_timer_callback periodically prints class info using RCLCPP_INFO.
 */
void SampleClassLocalNode::_timer_callback()
{
    RCLCPP_INFO_STREAM(get_logger(),
                       std::string("sum_of_squares = ") +
                       std::to_string(SampleClass::sum_of_squares(print_count_,print_count_-5))
                       );

```

(continues on next page)

(continued from previous page)

```
    RCLCPP_INFO_STREAM(get_logger(),
                       sample_class_.to_string() +
                       std::to_string(print_count_) +
                       std::string(" times.")
                       );
    print_count_++;
}
```

sample_class_local_node.hpp

sample_class_local_node.cpp

```
#pragma once

#include <rclcpp/rclcpp.hpp>
#include <cpp_package_with_a_library/sample_class.hpp>

/**
 * @brief A ROS2 Node that uses the SampleClass within the same package.
 */
class SampleClassLocalNode: public rclcpp::Node
{
private:
    SampleClass sample_class_;

    double timer_period_;
    int print_count_;
    rclcpp::TimerBase::SharedPtr timer_;

    void _timer_callback();
public:
    SampleClassLocalNode();

};
```

sample_class_local_node_main.cpp

sample_class.cpp

```
#include <rclcpp/rclcpp.hpp>

#include "sample_class_local_node.hpp"

int main(int argc, char** argv)
{
    rclcpp::init(argc,argv);

    try
    {
        auto node = std::make_shared<SampleClassLocalNode>();
```

(continues on next page)

(continued from previous page)

```

        rclcpp::spin(node);
    }
    catch (const std::exception& e)
    {
        std::cerr << std::string("::Exception::") << e.what();
    }

    return 0;
}
```

#vent Demystifying C++

> **Warning:** Anything below this point is just me venting about topics that frequently come up when C++ is mentioned.

But, C++ is difficult

I think C++ organically follows Bushnell's Law, adjusted for the topic

> All the best [programming languages] are easy to learn and difficult to master. They should reward the first quarter and the hundredth.

Beauty is in the eye of the beholder, but soon enough, if you're doing anything state-of-the-art, you'll hit performance bottlenecks with Python (and friends) that will naturally pull you towards C++.

But with Python, we don't need C++

This makes me feel like breaking the news to someone that Santa isn't real, but just as an example, see numpy and PyTorch.

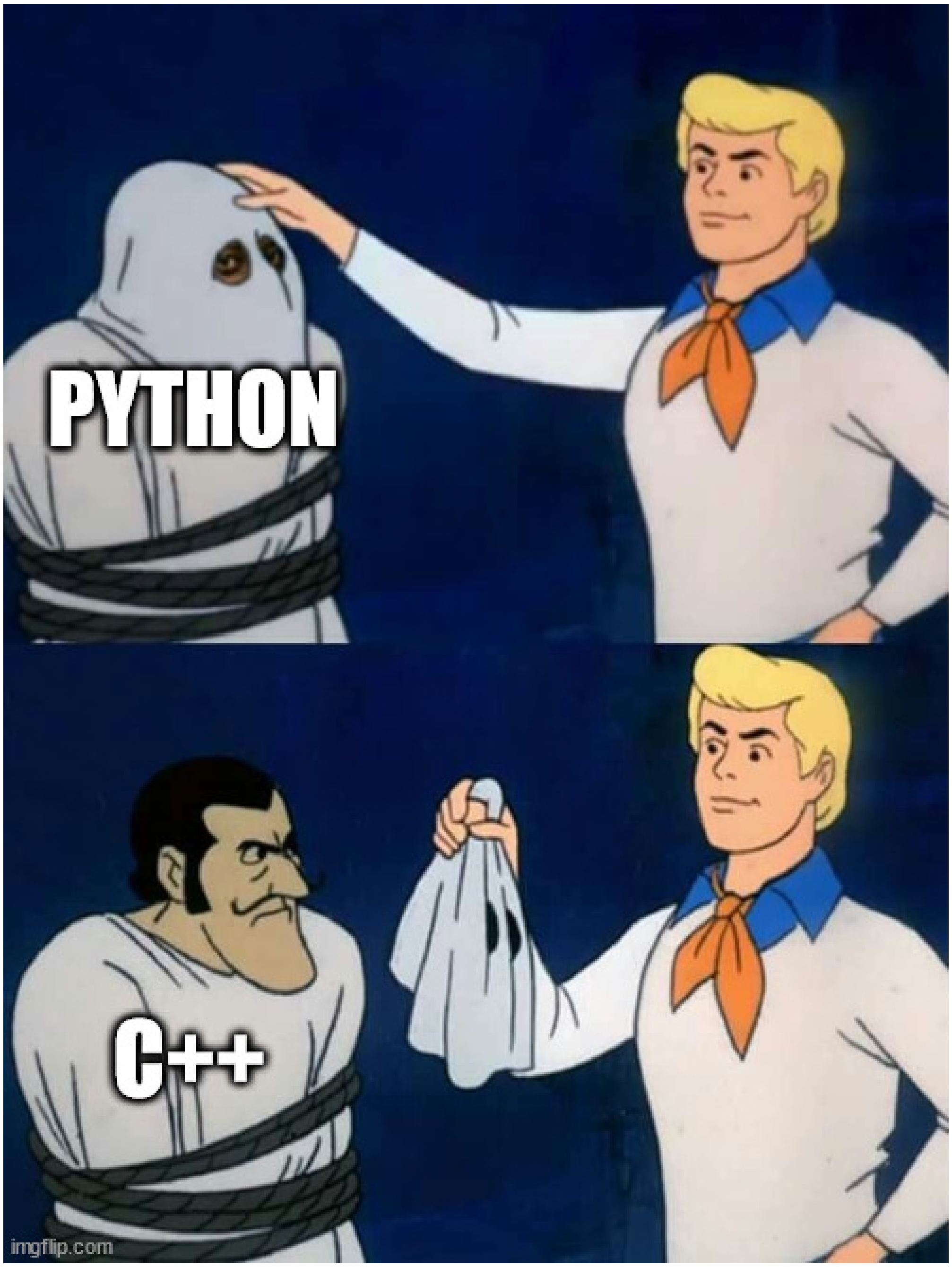
Why is NumPy Fast?

> [. . .] these things are taking place, of course, just "behind the scenes" in optimized, pre-compiled C code [. . .]

Using the PyTorch C++ Frontend

> [. . .] While the primary interface to PyTorch naturally is Python, this Python API sits atop a substantial C++ codebase providing foundational data structures and functionality such as tensors and automatic differentiation. [. . .]

The memefied version of this discussion is



Why use C++ if it sucks??

There's much folklore around C++. "C is faster than C++." "C++ is unsafe" (I'm looking at you, Rust). Anyhow, we'd all benefit if people stopped spreading weird fallacies about the C++ language when the problems they have can usually be attributed instead to a skill issue. Some quick info from Stroustrup's FAQ, also known as the person who designed and implemented the C++ programming language.

`<begin Stroustrup's FAQ quote>`

What is the difference between C and C++?

C++ is a direct descendant of C that retains almost all of C as a subset. C++ provides stronger type checking than C and directly supports a wider range of programming styles than C. C++ is "a better C" in the sense that it supports the styles of programming done using C with better type checking and more notational support (without loss of efficiency). In the same sense, ANSI C is a better C than K&R C. In addition, C++ supports data abstraction, object-oriented programming, and generic programming (see my books). I have never seen a program that could be expressed better in C than in C++ (and I don't think such a program could exist - every construct in C has an obvious C++ equivalent). [. . .]

C++ is low-level?
No. C++ offers both low-level and high-level features. C++ has low-level parts, such as pointers, arrays, and casts. These facilities are (almost identical to what C offers) are essential (in some form or other) for close-to-the-hardware work. So, if you want low-level language facilities, yes C++ provides a well-tried set of facilities for you. However, when you don't want to use low-level features, you don't need to use the C++ facilities (directly). Instead, you can rely on higher-level facilities, including libraries. For example, if you don't want to use arrays and pointers, standard library strings and containers are (better) alternatives in many cases. If you use only low-level facilities, you are almost certainly wasting time and complicating maintenance without performance advantages (see Learning Standard C++ as a New Language). You may also be laying your systems open to attacks (e.g. buffer overflows).

C++ too slow for low-level work?
No. If you can afford to use C, you can afford to use C++, even the higher-level facilities of C++ where you need their functionality. See Abstraction and the C++ machine model and the ISO C++ standards committee's Technical Report on Performance.

C++ is useful only if you write truly object-oriented code?
No. That is, "no" for just about any reasonable definition of "object-oriented". C++ provides support for a wide variety of needs, not just for one style or for one kind of application. In fact, compared to C, C++ provides more support for very simple programming tasks. For example, the standard library and other libraries radically simplifies many otherwise tedious and error-prone tasks. C++ is widely used for huge applications but it also provides benefits for even tiny programming tasks.

`<end Stroustrup's FAQ quote>`

But I hate pointers, and pointers hate me: The ballad of `segmentation fault (core dumped)`

In things entirely written in modern C++ (loosely C++11 and above, but C++14 and above for what I want to say here), you shouldn't see any `new` or any loose raw pointer modifiers `*`.

Use smart pointers. In general, std::shared_ptr and, if needed, std::unique_ptr.

If only using smart pointers you still manage to get a segmentation fault, then hats off to you.

But I can get segfaults with `std::vector`

As a successor of C, the standard library in C++ kept some of its predecessor's behavior of not generating exceptions.

For example, with trigonometric functions in C++, the error handling is C-like

For instance getting the `acos` of 1.1, which is invalid, will fail silently in C++. We must check if the output is `NaN`, e.g. with

```
#include <cmath>
#include <iostream>

int main()
{
```

(continues on next page)

(continued from previous page)

```
  auto a = std::acos(1.1);
  std::cout << std::isnan(a) ? "the output was invalid but no exception was thrown " : a
↪<< std::endl;
}
```

the same applies if we try to access beyond a vector's limits with the good and old `operator[]`. Instead of doing that, use the method `.at()`, which checks the bounds.

```
#include <iostream>
#include <vector>
#include <exception>

int main()
{
  auto v = {1.0,2.0,3.0,4.0};
  try
  {
    std::cout << v.at(22) << std::endl;
  }
  catch (const std::out_of_range& e)
  {
    std::cout << e.what() << std::endl;
  }
}
```

As a conclusion, find the correct function/method or throw an exception yourself.

But C++ makes too many copies of objects: The sonata of "I don't know perfect forwarding"

I see this claim all the time and it has many skill-issue-related causes, but basically, it shows up more frequently in the constructors of `std::vector` and `std::shared_ptr`.

Let's suppose that we have a class

```
class Potato{
  private:
    double size_;
  public:
    Potato(const double& size):
    size_(size)
    {};
};
```

for which we want to get a `std::shared_ptr`. Do not do this

```
auto potato_ptr = std::make_shared<Potato>(Potato(20.0));
```

> **Warning:** This is not the only issue you can have by doing this. It can generate all sorts of issues, in particular with classes that are not copyable.

because that will create one instance of `Potato(20.0)`, just to copy it when creating the `std::shared_ptr`. Do this, instead

```
auto potato_ptr = std::make_shared<Potato>(20.0);
```

by forwarding the argument to the constructor instead of calling it explicitly.

For everything else that you don't want to copy, use `std::move()`, but you don't see it that much unless you're designing a library.

CHAPTER

TWENTYSIX

FREQUENTLY ASKED QUESTIONS (FAQ)

Note: Also known as, frequently made comments, things I'd like to mention, etc.

26.1 You got the name wrong, it's ROS 2 not ROS2

Besides the humorous nature of the meme below and my love for the 1993's blockbuster, this is an inconspicuous way of showing, in every single section, that these tutorials are not official.

26.2 It's not Linux, it's GNU/Linux: Keep all grievances in `#vent`

The wording on these tutorials is precise as possible. Note that some terms are commonly used with loose meanings, but I hope that the message is still conveyed. This applies to the whole tutorial, given that even official sources are not uniform in their terminology.

So, to end any deep discussions that might distract you from the point of these tutorials before they even start, I'll let you with the world-renowned Linux copypasta edited with what was actually said

> *I'd just like to interject for a moment. What you're referring to as Linux, is in fact, GNU/Linux, or as I've recently taken to calling it, GNU plus Linux. Linux is not an operating system [. . .]. Many computer users run a modified version of the GNU system every day, without realizing it. Through a peculiar turn of events, the version of GNU which is widely used today is often called "Linux," and many of its users are not aware that it is basically the GNU system, developed by the GNU Project. There really is a Linux, and these people are using it, but it is just a part of the system they use.*
>
> *Linux is the kernel: the program in the system that allocates the machine's resources to the other programs that you run. The kernel is an essential part of an operating system, but useless by itself; it can only function in the context of a complete operating system. Linux is normally used in combination with the GNU operating system: the whole system is basically GNU with Linux added, or GNU/Linux. All the so-called "Linux" distributions are really distributions of GNU/Linux.*

26.3 The difference between Python *scripts* and *modules*

According to The Python Tutorial on Modules, the definition of *script* and *module* is not disjoint, in fact, it is said that

> *[. . .] you can make the file usable as a script as well as an importable module [. . .]*

In the official documentation, a Python script is defined as

> *[. . .] a [script is a] somewhat longer program, [for when] you are better off using a text editor to prepare the input for the interpreter and running it with [a script] as input instead [of using an interactive instance of the interpreter].*

and a module is defined as

> *[A module is a file] to put definitions [. . .] and use them in a script or in an interactive instance of the interpreter.*

There are more profound differences in how the Python interpreter handles *scripts* and *modules*, but in the wild the the difference is usually as I described in *Terminology*.

26.4 The difference between Python *modules* and *packages*

According to the Holy Book of Modules, a definition of packages is given *en passant* as follows

> *Suppose you want to design a collection of modules (a "package") [. . .]*

In practice, the line between modules and packages tends to be somewhat blurred. It could be a single folder with many modules but at the same time they come up with namings such as submodule

> *Packages are a way of structuring Python's module namespace [. . .]. For example, the module name A.B designates a submodule named B in a package named A.*

What most people want to say when they mention a package is, usually, either a folder with a `__init__.py` or a folder with a `setup.py` that can be built into a `wheel` or something similar.

CHAPTER

TWENTYSEVEN

WARNINGS

Warning: If you're using macOS or Windows, this is **NOT** the guide for you. There might be a lot of overlap, but none of the code shown here has been tested on those operating systems.

Warning: This project is under active development and is currently a draft.

CHAPTER

TWENTYEIGHT

DISCLAIMERS

By reading and/or using this tutorial in total or in part, you agree to these terms.

Disclaimer

ANYTHING ON THIS TUTORIAL–EVEN THINGS THAT ACTUALLY WORK–IS ENTIRELY FICTIONAL. SOME MEMES ARE ATTEMPTED. . . .POORLY. THE TUTORIAL CONTAINS MISPLACED MOVIE REFERENCES AND DUE TO ITS LOW-HANGING FRUIT HUMOUR, IT SHOULD NOT BE READ BY ANYONE.

Disclaimer

All advice, comments, and terrible memes in this tutorial are my own and not endorsed by anyone or anything else mentioned herein. It's not even endorsed by me.

Disclaimer

THIS TUTORIAL AND RELATED SOFTWARE ARE PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE AND/OR TUTORIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
