

# ROS 2 Humble

## Definition for ROS:

- ROS (Robot Operating System) is a framework to build and control robots.
- It helps robots sense, think, communicate, and move using code.
- Provides tools to connect sensors, control motors, and run logic.

## Definition For Ubuntu:

- Ubuntu is a popular Linux-based operating system.
- It's free, stable, and used in robotics for running ROS and other software.

## Relation:

- ROS needs Ubuntu to run.
- Ubuntu acts as the platform (like Windows), and ROS is the software to build robots.

## Definition for Ros2 Humble:

- A stable, long-term support (LTS) version of ROS 2.
- Released in **May 2022**, supported till **May 2027**.
- Works with **Ubuntu 22.04**.

## Why Ros2 Humble

- It is LTS, so it gets bug fixes and security updates for 5 years.
- Stable and reliable for real-world projects.

- More tutorials and packages available.
- Compatible with common tools like RViz, Gazebo.

### **Advantages:**

1. Long-term support (2022–2027).
2. Highly stable and tested.
3. Large community and resources.
4. Compatible with Ubuntu 22.04 (common OS).
5. Good for both beginners and professionals.

### **Disadvantages:**

1. Not the latest version — lacks some new features.
2. New hardware support may be limited.
3. Slightly older APIs than Jazzy.

### **Why not Jazzy instead?**

- Jazzy is newer (2024), but not LTS — supported only till 2025.
- May have bugs or unstable changes.
- Humble is better if you want stability and long-term use.

### **Applications of ROS 2 Humble:**

1. **Autonomous Mobile Robots (AMRs)**

- For navigation, mapping (SLAM), obstacle avoidance, and path planning in delivery bots, warehouse robots, etc.

## **2. Robot Arms and Manipulators**

- Used in pick-and-place operations, industrial automation, and collaborative robots (cobots).

## **3. Drones and UAVs**

- Enables flight control, localization, and camera-based navigation in aerial robotics.

## **4. Self-Driving Vehicles (Prototypes)**

- Used in research for LiDAR, camera fusion, lane detection, and behavior planning.

## **5. Multi-Robot Systems**

- Supports coordination and communication between multiple robots in a shared environment.

## **6. Human-Robot Interaction (HRI)**

- Voice control, gesture recognition, and service robots that interact with humans.

## **7. Service & Humanoid Robots**

- Used in home assistance, healthcare robots, or robots in public environments (e.g., guides).

## **8. Simulation and Testing**

- ROS 2 + Gazebo used to simulate robot behavior before deploying on real hardware.

## 9. Agricultural Robots

- Used in autonomous tractors, crop monitoring, and smart irrigation systems.

### Installation of Ubuntu 22.04:

Ubuntu 22.04 can be installed in 3 methods:

#### Method 1: Install Ubuntu 22.04 as Dual Boot with Windows:

##### Step 1: Download Ubuntu 22.04 ISO

1. Go to: <https://ubuntu.com/download/desktop>
2. Click on “Download Ubuntu 22.04 LTS”
3. Save the .iso file (~3.5 GB)

This file is the installer image used to create a bootable USB.

##### Step 2: Create a Bootable USB using Rufus

###### Download Rufus:

- Go to: <https://rufus.ie>
- Download and open the Rufus executable (no installation needed)

## **Prepare the USB:**

1. Insert a USB drive (minimum 8 GB, will be formatted).
2. In Rufus:
  - Device: Choose your USB drive.
  - Boot selection: Click “Select” and choose the downloaded Ubuntu ISO.
  - Partition scheme: Choose GPT
  - File System: Leave as FAT32
3. Click Start
4. When asked to write in ISO mode, click Yes/OK.

This will erase everything on the USB.

Now the USB is ready to boot Ubuntu.

## **Step 3: Shrink Windows Partition (To Make Space)**

1. Press Windows + R, type `diskmgmt.msc` → Enter.
2. Right-click on the main partition (usually C:), choose Shrink Volume.
3. Enter how much to shrink (e.g., 30,000 MB = 30 GB) → Click Shrink

This creates unallocated space for Ubuntu to install into.

## **Step 4: Boot into Ubuntu Installer (via USB)**

1. Restart your computer.
2. Press the boot menu key (depends on your system):
  - Dell - F12
  - HP - ESC or F9
  - Lenovo - F10 or F12

- Acer - F12
- ASUS - ESC or F8

3. Choose your USB drive from the list.

This boots the Ubuntu installer.

### **Step 5: Install Ubuntu 22.04**

1. When Ubuntu loads, click "Install Ubuntu".
2. Choose language, keyboard layout → click Continue.
3. Choose Normal Installation → check Install third-party software → Continue.
4. Important Step – Installation Type:
  - Choose "Install Ubuntu alongside Windows Boot Manager"
  - You'll see a slider to adjust space for Ubuntu and Windows. Allocate space as you want (e.g., 30 GB for Ubuntu).
5. Click Install Now, then Continue to confirm partition changes.
6. Choose your location (India, if you're here).
7. Set up:
  - Name
  - Username
  - Password (you'll use this often)
8. Installation will begin. Wait for it to finish (10–30 mins).
9. Once done, click Restart Now.
  - Remove USB when prompted.

You'll now see a GRUB menu at startup to choose between Ubuntu or Windows!

## Method 2: Install Ubuntu 22.04 in VirtualBox

### Definition:

VirtualBox is free software that lets you run another operating system (like Ubuntu) inside your current one (like Windows) as a virtual machine. It's like creating a separate computer within your computer. Ideal for safely testing Linux, ROS, or any other OS without affecting your main system.

### Step 1: Download and Install VirtualBox

1. Visit: <https://www.virtualbox.org>
2. Click on "Downloads" → Choose Windows hosts
3. Run the downloaded .exe file.
4. Click Next through all the steps and Install

This installs VirtualBox on your Windows system.

### Step 2: Download Ubuntu 22.04 ISO

1. Visit: <https://ubuntu.com/download/desktop>
2. Click "Download Ubuntu 22.04 LTS"
3. Save the .iso file (~3.5 GB) to your computer.

This file is used as a virtual CD to install Ubuntu.

### Step 3: Create a New Virtual Machine in VirtualBox

1. Open VirtualBox (you'll see a clean interface).
2. Click New (top-left corner).

Fill these details:

- Name: Ubuntu 22.04
- Machine Folder: Leave as default
- Type: Linux
- Version: Ubuntu (64-bit)
- Click Next

Memory Size:

- Set at least 4096 MB (4 GB) (more if your system allows, like 6144 MB or 8192 MB)
- Click Next

Hard Disk:

- Choose: Create a virtual hard disk now → Click Create

#### **Step 4: Configure the Virtual Hard Disk**

1. Hard disk file type: Leave as VDI (VirtualBox Disk Image) → Click Next
2. Storage on physical hard disk: Choose Dynamically allocated → Click Next
3. File location and size:
  - Location: Leave default
  - Size: Choose 25–50 GB (minimum 25 GB)
  - Click Create



Now your virtual machine is created!

### **Step 5: Load Ubuntu ISO and Start VM**

1. In VirtualBox, select Ubuntu 22.04 → Click Start
2. A window appears asking for a startup disk:
  - Click the folder icon → Browse to the Ubuntu ISO you downloaded
  - Select it and click Start

Ubuntu installation will begin inside a virtual machine.

### **Step 6: Install Ubuntu 22.04**

Now you're inside the Ubuntu installer, follow these steps:

1. Choose Language:
  - Select your preferred language → Click Continue
2. Keyboard Layout:
  - Default is usually English (US) → Click Continue
3. Updates and Other Software:
  - Choose: Normal installation
  - Check Install third-party software → Click Continue
4. Installation Type:

You will see:

“Erase disk and install Ubuntu”. This erases the virtual disk only, NOT your Windows system.

- Select it → Click Install Now

- Confirm partition changes → Click Continue

#### 5. Time Zone:

- Choose your time zone (e.g., Asia/Kolkata)

#### 6. User Account Setup:

- Your Name
- Username: auto-filled
- Password: Choose a secure one
- Click Continue

### **Step 7: Restart the Virtual Machine**

- Once installation is complete, click Restart Now
- If prompted, remove the ISO:
  - VirtualBox usually ejects it automatically
  - If not: Go to Devices → Optical Drives → Remove disk from virtual drive

Ubuntu will now boot inside VirtualBox! You can log in with the password you created.

## Method 3:Installing Ubuntu 22.04 via WSL on Windows

### Step 1: Enable WSL and Virtual Machine Platform

Open PowerShell as Administrator and run:

bash

```
wsl --install
```

#### What it does:

- Installs WSL 2 (if not already).
- Enables the **Virtual Machine Platform** and WSL features.
- Install **Ubuntu** by default (The default version will be the currently released version,you can change the version later).

#### Errors faced:

By running this command the default Ubuntu version is installed which is not my requirement.To rectify this,we need to delete the installed Ubuntu version and then install as per our requirement .For that run,

### Step 1: Check Installed WSL Distros

Open PowerShell or Command Prompt, then run:

bash

```
wsl --list --verbose
```

#### Explanation:

- wsl → Command-line tool to manage Windows Subsystem for Linux
- --list → Lists all installed Linux distributions
- --verbose → Shows extra details like running status and WSL version

**Purpose:** To see what WSL Ubuntu versions are currently installed.

### Example output:

NAME	STATE	VERSION
* Ubuntu	Running	2
Ubuntu-20.04	Stopped	2

## Step 2: Uninstall the Wrong Ubuntu Version

To uninstall a specific distro (e.g., just named Ubuntu):

**bash**

```
wsl --unregister Ubuntu
```

If it's Ubuntu-20.04, then:

**bash**

```
wsl --unregister Ubuntu-20.04
```

This will delete all data inside that WSL instance, so back up anything important first!

Explanation:

- wsl → WSL command-line tool
- --unregister → Completely **removes the specified distro**
- Ubuntu → The **name** of the Linux distribution to remove (can also be Ubuntu-20.04, etc.)

## Step 3: Install the Correct Ubuntu Version (e.g., 22.04)

Use this command to install Ubuntu 22.04 directly:

**bash**

```
wsl --install -d Ubuntu-22.04
```

**Explanation:**

- wsl → WSL tool
- --install → Tells WSL to install a new Linux distro
- -d → Means "distribution"
- Ubuntu-22.04 → Name of the exact distro version to install

**Purpose:** Installs Ubuntu 22.04, which is required for ROS 2 Humble.

**Step 4: Launch It**

After installation:

**bash**

```
wsl -d Ubuntu-22.04
```

**Explanation:**

- wsl → Run WSL
- -d → Choose specific distro
- Ubuntu-22.04 → Open this version

**Purpose:** Launches the freshly installed Ubuntu 22.04 to ask for a username and password.

By doing this you can remove the default version and install Ubuntu as per your requirements, else You can directly install your required version by the following steps.

**If you want to specifically install a Ubuntu version run:**

bash:

**wsl --install -d Ubuntu-version**

Eg:For installing Ubuntu 22.04,

bash:

**wsl --install -d Ubuntu-22.04**

-d stands for distribution, and Ubuntu-22.04 specifies the version.

## **Step 2: Restart Your PC**

## **Step 3: Launch Ubuntu 22.04**

After reboot, open "Ubuntu 22.04" from your Start Menu.

- It will ask for a username and password (Linux user account).
- After setup, you'll see the Ubuntu terminal — now you're inside Ubuntu 22.04

**If u want to create a new user(user 2),the following steps are to be followed,**

## **Step 1:Adding username**

Go to Ubuntu(WSL) terminal and run

bash

**sudo adduser yourusername**

### **Explanation:**

- sudo → Run the command with superuser (admin) privileges.
- adduser → Linux command to create a new user account.
- yourusername → The name you want for the new user (

After this, it will ask:

- Enter new UNIX password: → Type the new password (invisible).
- Retype new UNIX password: → Type it again to confirm.
- Then it may ask for full name, room number, etc. — you can press Enter to skip.

This creates the new user with its own password and home folder.

## Step 2: Give the New User Admin (sudo) Rights

bash

**sudo usermod -aG sudo new\_username**

### Explanation:

- sudo → Run as administrator.
- usermod → Modify a user account.
- -aG → Add the user to a group (without removing them from other groups).
- sudo → The name of the group we're adding them to (gives admin rights).
- newusername → The new username you're giving sudo access to.

Now, the new user can use sudo to run admin-level commands.

If you make to make new user as default user ,run

## Step 4: Set the New User as Default (So It Auto-Logs In)

Open **PowerShell (Windows)** and run:

powershell

**wsl --set-default-user new\_username**

### Explanation:

- wsl → The Windows Subsystem for Linux command-line tool.

- `--set-default-user` → Tells WSL to change the default login user.
- `newusername` → The new user you created and want to set as default.

Now, when you open Ubuntu, it logs in automatically with the new username.

### **Challenge:**

If u have forgotten your password,

For security reasons, there is no way to see or recover the old (forgotten) password of a Linux (WSL) user.

### **Why you can't retrieve the old password:**

- Linux uses hashed passwords, not plain text.
- These hashes are one-way encrypted, so the original password cannot be decoded.

### **Alternative:**

There is a way to rectify this, you can reset the password if you have access to another admin user in the same WSL. For that, run from the wsl terminal of the person who is the admin user.

bash

`sudo passwd username1`

You are using User 2 (who has sudo access) to reset the password for nithya.

### **Explanation:**

- `sudo` → Run the command with superuser (admin) privileges — needed because you're changing another user's password.
- `passwd` → Linux command to change the password for any user.
- `username1` → The username of the person whose password you want to change (in this case, the old user whose password is forgotten).



What happens next:

The terminal will not ask for the old password of user1. It will directly prompt you to:

**Enter a new password:**

**Re-type the new password to confirm:**

Hence, You can set a brand new password for the old user (user1), without knowing their old password — as long as you're using a user(user2) who has sudo access.

## **Ros 2 Humble Installation:**

### **Step 1: Set Locale:**

bash  
**Locale**

#### **What it means:**

- `locale`: This command displays your system's language and regional settings.
- It shows environment variables like `LANG`, `LC_ALL`, `LC_TIME`, etc.
- You're checking if your system is already using a UTF-8 encoding (which is required by ROS and many tools for correct character support).

You want to see values like: `en_US.UTF-8`

If Output is like this,

**LANG=C**  
**LC\_ALL=**  
**LC\_MESSAGES=C**  
or no `en_US.UTF-8`, then you need to fix it.

bash

**sudo apt update && sudo apt install locales**

What it means:

- sudo: Run the command as superuser (admin privileges).
- apt: Ubuntu's package manager (used to install software).
- update: Tells apt to refresh the list of available packages from online sources.
- &&: Run the next command only if the previous one succeeds.
- install locales: Installs the "locales" package, which contains language and region settings for the system.

This ensures your system has the tools to configure language settings.

bash

**sudo locale-gen en\_US en\_US.UTF-8**

What it means:

- locale-gen: A command to generate the specified locale definitions.
- en\_US: The basic English (US) locale (without encoding).
- en\_US.UTF-8: The UTF-8 encoded English (US) locale (UTF-8 supports all characters and is standard for ROS).

This creates and enables the en\_US.UTF-8 locale if it's not already available.

bash

**sudo update-locale LC\_ALL=en\_US.UTF-8 LANG=en\_US.UTF-8**

What it means:

- update-locale: Updates the system's default locale settings.
- LC\_ALL=en\_US.UTF-8: Sets all locale categories (like time, messages, currency) to en\_US.UTF-8.
- LANG=en\_US.UTF-8: Sets the default system language to en\_US.UTF-8.

This tells Ubuntu to use UTF-8 English as the main system language, which prevents character issues in tools like ROS.

bash

**export LANG=en\_US.UTF-8**

What it means:

- export: Temporarily sets an environment variable in the current terminal session.
- LANG: This variable controls the default language and encoding for programs.
- =en\_US.UTF-8: Sets it to US English with UTF-8 encoding.

This ensures the terminal session uses the UTF-8 language setting immediately (before reboot or re login).

bash

**locale**

What it means:

Run the locale command again to verify that:

- LANG=en\_US.UTF-8
- LC\_ALL=en\_US.UTF-8

All locale variables show UTF-8 encoding

This confirms that your changes were successfully applied.

**The above process should be done only when the locale is not UTF-8.**

## Step 2: Setup Sources

You will need to add the ROS 2 apt repository to your system. First ensure that the Ubuntu Universe repository is enabled. For that run,

bash

**sudo apt install software-properties-common**

What each part means:

1. `sudo`: Runs the command with admin (superuser) privileges. Required to install system software.
2. `apt`: The package manager used in Ubuntu to install, update, or remove software.
3. `install`: Tells `apt` that you want to install a package.
4. `software-properties-common`: This is a helper package that provides tools like `add-apt-repository`, which is used to manage software sources (like Universe, Multiverse, etc.).

**Purpose:**

To install the tools necessary for adding additional repositories to your Ubuntu system.

bash

**`sudo add-apt-repository universe`**

What each part means:

1. `sudo`: Again, you need admin access to modify system software sources.
2. `add-apt-repository`: A command used to add new software repositories to your system (APT can then use them to find and install packages).
3. `universe`: This is the name of the Ubuntu repository you're adding.
  - The Universe repository contains community-maintained open-source software, including some packages required by ROS 2.

**Purpose:**

To enable the Universe repository, which is essential for installing ROS 2 Humble and its dependencies.

**Step 3: Adding Ros2 repository and GPG key**

Goal:

Install the ROS 2 APT source package which:

- Adds the official ROS 2 repository to your system
- Adds the GPG key (for secure downloads)
- Makes future ROS 2 updates easier

bash

```
sudo apt update && sudo apt install curl -y
```

Explanation:

1. `sudo`: Run as admin
2. `apt update`: Updates the package list (so you get the latest versions).
3. `&&`: Only run the next command if the first one succeeds
4. `apt install curl -y`:
  - `curl`: A command-line tool to download files or data from the internet
  - `-y`: Automatically answers "yes" to the installation prompt

This ensures `curl` is installed, which you'll use to download the ROS APT source package.

bash

```
export ROS_APT_SOURCE_VERSION=$(curl -s  
https://api.github.com/repos/ros-infrastructure/ros-apt-source/releases/latest |  
grep -F "tag_name" | awk -F\" '{print $4}')
```

Explanation:

1. `export`: Creates an environment variable for the current terminal session.
2. `ROS_APT_SOURCE_VERSION=...`: Assigns the latest version of the `ros-apt-source` package to this variable.
3. `curl -s ...`: Downloads the latest release info of `ros-apt-source` from GitHub API.
  - `-s`: Silent mode (no progress bar).
4. `grep -F "tag_name"`: Filters the line containing the version tag name (e.g., `v1.1.0`).
5. `awk -F\" '{print $4}'`: Extracts just the version number from that line.

After this command, your terminal stores the latest version in the variable `$ROS_APT_SOURCE_VERSION`.

`bash`

```
curl -L -o /tmp/ros2-apt-source.deb  
"https://github.com/ros-infrastructure/ros-apt-source/releases/download/${ROS_  
APT_SOURCE_VERSION}/ros2-apt-source_${ROS_APT_SOURCE_VERSION}.${  
/etc/os-release && echo $VERSION_CODENAME)_all.deb"
```

### Explanation:

1. `curl` → A command-line tool to download data from URLs.
2. `-L` → Tell `curl` to follow HTTP redirects (like GitHub's download links).
3. `-o` → Specifies the name and path for the downloaded output file.
4. `/tmp/ros2-apt-source.deb` → The full path and filename to save the file in the temporary folder.
5. `"https://github.com/ros-infrastructure/ros-apt-source/releases/download/"` → The base URL where the ROS APT package is hosted on GitHub.
6. `${ROS_APT_SOURCE_VERSION}` → A variable holding the latest version of the ROS APT source package (like `1.1.0`).

7. `/ros2-apt-source_` → Start of the actual file name being downloaded.
8. `${ROS_APT_SOURCE_VERSION}` → Again inserts the version number into the filename.
9. `.` → Adds a dot to separate the version from the Ubuntu codename.
10. `$(...)` → Shell syntax to run a command inside and insert its output.
11. `/etc/os-release` → Loads Ubuntu's OS information into the current shell.
12. `echo $VERSION_CODENAME` → Prints the Ubuntu codename (e.g., jammy for 22.04).
13. `_all.deb` → Final part of the file name indicating it's a .deb file for all architectures.

bash

```
sudo dpkg -i /tmp/ros2-apt-source.deb
```

Explanation:

1. `sudo`: Admin access
2. `dpkg -i`: Install a .deb file manually
3. `/tmp/ros2-apt-source.deb`: Path to the downloaded ROS 2 apt source package

This installs the ROS 2 APT source, so your system knows where to get ROS packages from.

## Step 4: Install ROS 2 packages

bash

```
sudo apt update
```

Explanation:

1. `sudo` → Run the command with superuser (admin) privileges.
2. `apt` → Ubuntu's package manager used to manage software.
3. `update` → Refreshes the local list of available packages from online repositories (no installation yet).

This command checks for the latest versions of all packages available from Ubuntu and ROS sources.

bash

**sudo apt upgrade**

**Explanation:**

1. sudo → Again, run as admin (to make system-level changes).
2. apt → Package manager for Ubuntu.
3. upgrade → Installs the newest versions of currently installed packages (but doesn't remove or add any new ones).

This makes sure all your existing system packages are fully updated to avoid conflicts during ROS 2 installation.

## **Step 5:Humble installation**

bash

**sudo apt install ros-humble-desktop**

**Explanation:**

1. sudo → Run with admin privileges.
2. apt install → Installs a software package.
3. ros-humble-desktop → Installs the full version of ROS 2 with:
  - ROS core libraries
  - GUI tools like RViz and Gazebo
  - Tutorials and example packages

Best for learning and using visualization tools.

If u want to install only ros without any GUI tools,run

bash

**sudo apt install ros-humble-ros-base**

Installs only the core ROS 2 tools, communication libraries, and message packages. No GUI tools like RViz or simulation environments. Best for headless or performance-focused systems.



bash

**sudo apt install ros-dev-tools**

Adds tools like:

CMake, build-essential, colcon, etc.

Required for compiling your own ROS packages.

## Step 6:Environment Setup

bash

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

Explanation:

- source → Runs a script in the current terminal session.
- /opt/ros/humble/setup.bash → This script sets ROS 2 environment variables, so ROS commands like ros2 run and ros2 topic work.

This step is required every time you open a new terminal unless you add it to .bashrc.

If u want to automatically source it,run,

bash

**echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc**

### Explanation

1. echo → Prints a string.
2. "source /opt/ros/humble/setup.bash" → This is the command that sets up the ROS 2 environment.
3. >> ~/.bashrc → Appends it to your .bashrc file (your terminal's startup script in your home directory).

Then apply the change immediately by running:

bash

`source ~/.bashrc`

Now, every new terminal session will automatically have ROS 2 sourced

Now Ros2 Humble is successfully installed.

Below are the **terminal commands** to install **RViz** and **Gazebo** in **ROS 2 Humble** (on Ubuntu 22.04), along with a short explanation for each.

IF u have installed the ros2 desktop version, it already includes the GUI tools like gazebo and rviz, but if you have installed the ros 2 base version which does not include the GUI tools, you can use the following commands to install it.

## Install RViz (ROS Visualization Tool)

bash

`sudo apt update`

`sudo apt install ros-humble-rviz2`

### Explanation:

- `sudo` → Run as superuser (admin rights).
- `apt update` → Refreshes the list of available packages.
- `apt install` → Installs software.
- `ros-humble-rviz2` → Installs RViz2, the visualization tool for ROS 2 Humble.

## Install Gazebo (Simulation Environment)

bash

`sudo apt update`

`sudo apt install ros-humble-gazebo-ros-pkgs`

**Explanation:**

- `ros-humble-gazebo-ros-pkgs` → Installs the Gazebo ROS integration packages for ROS 2 Humble (so you can launch and control simulations).

**Code Editors used in Ros2 Humble:**

Generally in Ros 2 humble we can use 3 different code editors

1. VS Code (Recommended for beginners and projects.)
2. nano (Terminal-based, always available)
3. gedit (GUI-based, lightweight graphical editor)

**Method 1: Using VS Code**

Step 1: Install VS Code (if not installed)

bash

`sudo snap install code --classic`

**Explanation:**

- `sudo` → Run as administrator
- `snap install` → Install using snap package system
- `code` → Name of the package (VS Code)
- `--classic` → Required flag for full access to system paths

Installs Visual Studio Code.

## Step 2: Open a file in VS Code

bash

`code filename.py`

Example:

bash

`code simple_publisher.py`

Or to open the whole folder (recommended):

bash

`code .`

### Explanation:

- `.` → Current directory

Open the entire folder in VS Code.

## Method 2: Using nano (built-in terminal editor)

To open a file:

bash

`nano simple_publisher.py`

### Explanation:

- `nano` → Text editor in the terminal
- `simple_publisher.py` → File to open/edit

Use:

- Arrow keys to move
- Ctrl + O to save
- Ctrl + X to exit

### Method 3: Using gedit (GUI text editor)

To install gedit:

bash

**sudo apt install gedit**

To open a file:

bash

**gedit simple\_publisher.py**

Opens a graphical window like Notepad.

### Basic Ubuntu Terminal Commands (Used in ROS 2 Work)

Terminal commands	Meaning
<b>ls</b>	Lists all files and folders in the current directory.
<b>cd folder_name</b>	Changes directory into the specified folder.
<b>cd ..</b>	Moves two levels up in the directory tree.

<b>cd ../..</b>	Moves two levels up in the directory tree.
<b>pwd</b>	Prints the current working directory path.
<b>mkdir foldername</b>	Creates a new folder.
<b>rm filename</b>	Deletes a file.
<b>rm -r folder name</b>	Deletes a folder and everything inside it (recursive delete).
<b>touch filename</b>	Creates a new empty file.
<b>code .</b>	Open the current folder in VS Code (if VS Code is installed and code is configured).
<b>clear</b>	Clears the terminal screen.
<b>sudo command</b>	Runs a command with superuser (admin) privileges.

<b>echo "text"</b>	Prints text or variables to the terminal.
<b>source file.sh</b>	Executes the commands in a file in the current terminal session (commonly used for ROS setup).
<b>colcon build</b>	Build all ROS 2 packages inside your workspace.
<b>ros2 run pkg_name node_name</b>	Runs a specific ROS 2 node.
<b>ros2 topic echo /topic_name</b>	Shows real-time messages published to a topic.

## ROS 2 Humble Basic Commands:

Terminal commands	Meaning
<b>ros2 --version</b>	Shows the installed version of ROS 2 (e.g., humble).
<b>ros2 pkg list</b>	Lists all installed ROS 2 packages on your system.
<b>ros2 pkg prefix &lt;package_name&gt;</b>	Displays the install path of a specific package.

<b>ros2 run &lt;package_name&gt; &lt;executable_name&gt;</b>	Starts a specific ROS 2 node (binary or script) from the given package.
<b>ros2 topic list</b>	Shows all the topics currently being published or subscribed.
<b>ros2 topic echo &lt;topic_name&gt;</b>	Prints messages published to a topic in real time.
<b>ros2 topic info &lt;topic_name&gt;</b>	Shows publisher/subscriber details for a topic.
<b>ros2 topic pub &lt;topic_name&gt; &lt;msg_type&gt; "data"</b>	Publishes a message to a topic manually.
<b>ros2 node list</b>	Shows all running nodes in the current ROS 2 system.
<b>ros2 node info &lt;node_name&gt;</b>	Shows detailed information about the node's topics, services, etc
<b>ros2 msg list</b>	Lists all available ROS 2 message types.
<b>ros2 msg show &lt;msg_type&gt;</b>	Shows the fields inside a message type.
<b>ros2 service list</b>	Lists all the active services currently available.
<b>ros2 service call &lt;service_name&gt; &lt;srv_type&gt; &lt;arguments&gt;</b>	Calls a service with specific inputs.



<b>source /opt/ros/humble/setup.bash</b>	Enables ROS 2 environment in your current terminal session.
--	---

## Turtlesim:

Turtlesim is a beginner-friendly ROS 2 simulation tool that shows a small turtle moving on the screen. It's used to:

- Learn how ROS topics, nodes, and services work
- Practice publishing and subscribing to messages
- Test basic movement and control concepts

## Steps to run turtlesim:

### Step 1: Install Turtlesim

bash

**sudo apt update**

#### Explanation:

- sudo → Run the command as an administrator.
- apt → Ubuntu's package manager.
- update → Refreshes the list of available packages.

bash

**sudo apt install ros-humble-turtlesim**

#### Explanation:

- install → Installs the given package.

- `ros-humble-turtlesim` → The Turtlesim package for ROS 2 Humble version.

## Step 2: Source ROS 2 Setup File

`bash`

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

### Explanation:

- `source` → Loads environment settings into the current terminal session.
- `/opt/ros/humble/setup.bash` → This script sets up ROS 2 paths, so `ros2` commands will work.

## Step 3: Launch the Turtlesim GUI

`bash`

`ros2 run turtlesim turtlesim_node`

### Explanation:

- `ros2` → ROS 2 command-line tool.
- `run` → Run a node from a package.
- `turtlesim` → The name of the package.
- `turtlesim_node` → The executable node that opens the turtle GUI window.

This shows a window with a turtle inside a blue square.

## Step 4: Control Turtle with Keyboard

Open a new terminal, then again run:

bash

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

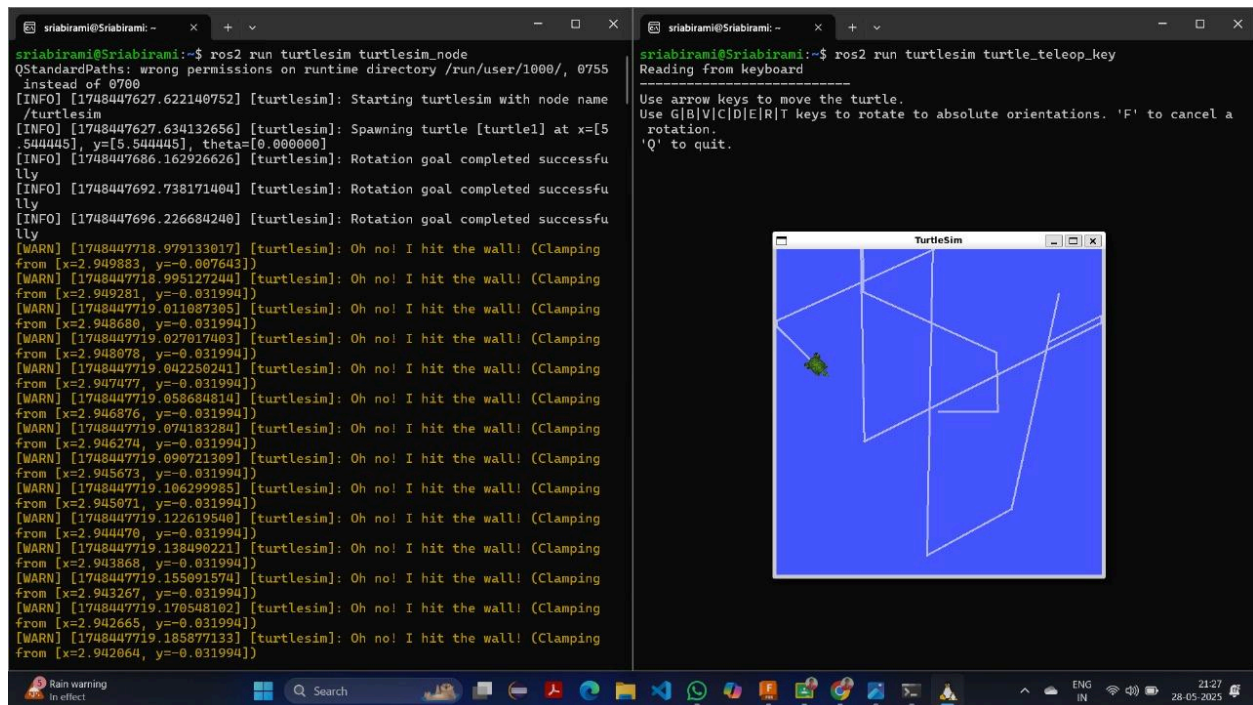
Now run:

bash

`ros2 run turtlesim turtle_teleop_key`

`turtle_teleop_key` → A node that allows controlling the turtle using keyboard arrows.

Use arrow keys to move the turtle around!



You can even control the turtle manually by publishing to the topic:

bash

`ros2 topic pub /turtle1/cmd_vel geometry_msgs/Twist "{linear: {x: 2.0}, angular: {z: 1.0}}"`

## Explanation:

- `ros2 topic pub` → Publish a message to a topic.
- `/turtle1/cmd_vel` → The topic that controls turtle motion.
- `geometry_msgs/Twist` → Message type used for velocity.
- The JSON-like data sets:
  1. `linear.x` → forward speed (2.0)
  2. `angular.z` → turning speed (1.0)

This command tells the turtle to:

- Move forward at speed 2.0 (`linear.x = 2.0`)
- Rotate (turn) to the left at speed 1.0 (`angular.z = 1.0`)

So the turtle will:

Move in a circular path (forward + turning at the same time)

## Getting into Ros2:

## 1.Creating a Workspace:

A workspace in ROS 2 is a directory where you keep your own packages, build them, and run them.

### Why to Create a Workspace:

- To organize your code (custom packages, scripts, nodes).
- To build your packages using colcon build.
- To test, run, and modify ROS 2 code easily.
- ROS 2 does not allow editing system packages, so we create our own workspace.

## Steps to Create a ROS 2 Workspace

I would like to create a workspace named ros2\_ws (a common name).

### Terminal Commands:

```
bash
```

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

### Explanation:

- mkdir → Make directory (create a folder).
- -p → Create parent directories if they don't exist (no error if they do).
- ~ → Your home directory (e.g., /home/nithya).
- ros2\_ws → The name of your ROS 2 workspace.
- src → The source folder where you will place packages.

```
bash
```

`cd ~/ros2_ws`

**Explanation:**

- `cd` → Change directory.
- `~/ros2_ws` → Move into your workspace directory.

bash

`colcon build`

**Explanation:**

- `colcon` → The ROS 2 build tool (like catkin in ROS 1).
- `build` → Builds all packages inside the `src/` folder.

bash

`source install/setup.bash`

**Explanation:**

- `source` → Loads the environment variables into the current terminal.
- `install/setup.bash` → Sets up the workspace's paths so ROS 2 can find your packages.

## **2.Creating a Package:**

This is where you'll write your custom nodes (like Python or C++ publisher/subscriber code).

A package is a folder that contains:

- Your code (scripts or nodes)

- A package.xml file (metadata)
- A setup.py (for Python) or CMakeLists.txt (for C++) file (build instructions)

### **Steps to Create a ROS 2 Package:**

#### **Step 1: Make sure you're inside your workspace's src/ folder:**

bash

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

#### **Step 2: Now create your package:**

##### **For python:**

bash

```
ros2 pkg create --build-type ament_python my_py_pkg
```

##### **Explanation:**

- ros2 → ROS 2 command-line interface
- pkg → Refers to package commands
- create → Command to create a new package
- --build-type ament\_python → Tell ROS that this is a Python-based package using the ament build system
- my\_py\_pkg → Name of your new package (you can name it anything)

#### **This will create the following folder:**

bash

```
~/ros2_ws/src/my_py_pkg/
```

With these files inside (For python)

- setup.py → Python build script

- package.xml → Metadata about the package
- my\_py\_pkg/\_\_init\_\_.py → Marks it as a Python module
- resource/ → Resource info
- test/ → Folder for tests

### **Step 3:After Creating the Package**

1. Go back to your workspace root:

bash

`cd ~/ros2_ws`

2. Build the package:

bash

`colcon build`

If build is successful, you'll see:

output:Summary: 1 package finished

3. Source the environment:

bash

`source install/setup.bash`



## ROS 2 Core Concepts

Example Robot: A Smart Vacuum Cleaner

Let's assume we are building a robot vacuum cleaner. It will:

- Move around
- Sense obstacles
- Report battery level
- Go to the charging station when needed

Using this, the following concepts will be explained

### 1. Node

A Node is a single executable or script that performs a specific function in the robot.

Example:

- `sensor_node`: reads data from obstacle sensors
- `motor_node`: controls wheel movement
- `battery_node`: checks battery status

Each of these is a separate node.

#### Real Use:

You write a Python/C++ file (e.g., `motor_node.py`) that contains the logic to control the motor.

### Steps to Create a Node in ROS 2 (Python)(Example)

#### Step 1: Navigate to your package folder

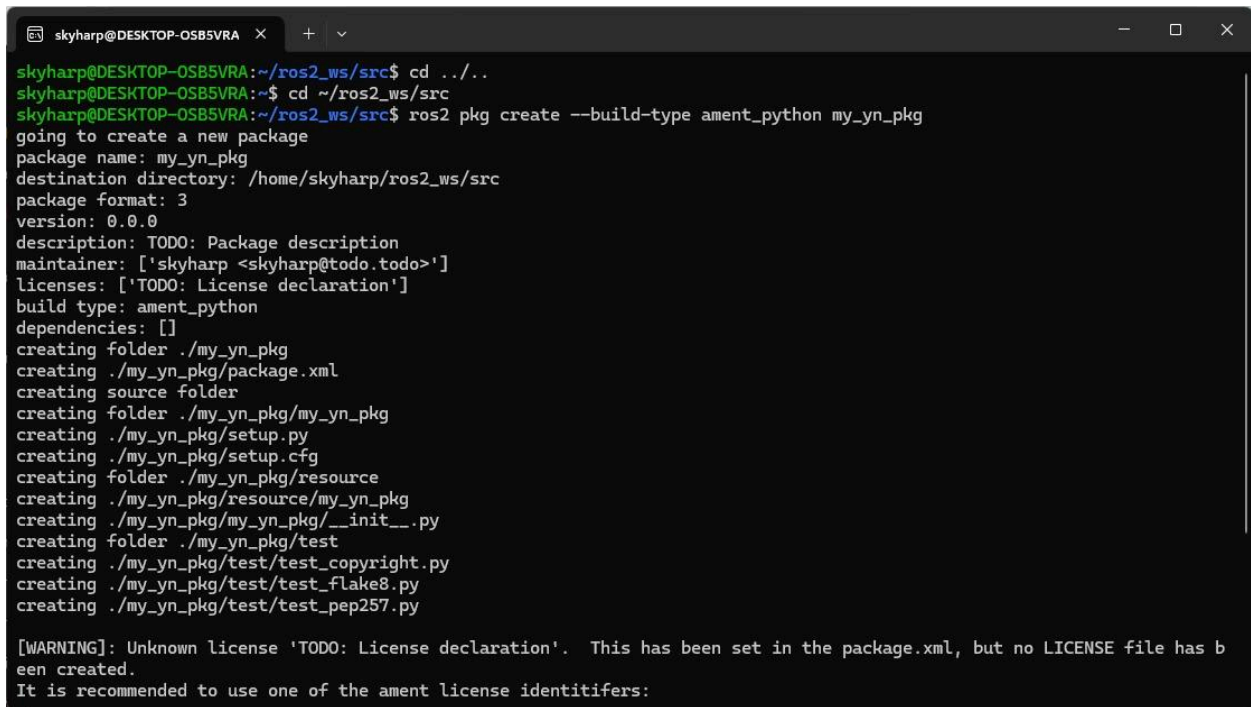
I have already created a package named `my_yn_pkg` using the above steps

bash

`cd ~/ros2_ws/src/my_py_pkg/my_yn_pkg`

### Explanation:

- `cd` → change directory
- `~/ros2_ws` → your workspace
- `src/my_py_pkg/my_yn_pkg` → folder where node scripts live
  1. `my_py_pkg`-This is the ROS 2 package directory (outer)
  2. `my_yn_pkg`-This is the Python module/package directory (inner)



```
skyharp@DESKTOP-OSB5VRA:~/ros2_ws/src$ cd ../../
skyharp@DESKTOP-OSB5VRA:~$ cd ~/ros2_ws/src
skyharp@DESKTOP-OSB5VRA:~/ros2_ws/src$ ros2 pkg create --build-type ament_python my_yn_pkg
going to create a new package
package name: my_yn_pkg
destination directory: /home/skyharp/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['skyharp <skyharp@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
creating folder ./my_yn_pkg
creating ./my_yn_pkg/package.xml
creating source folder
creating folder ./my_yn_pkg/my_yn_pkg
creating ./my_yn_pkg/setup.py
creating ./my_yn_pkg/setup.cfg
creating folder ./my_yn_pkg/resource
creating ./my_yn_pkg/resource/my_yn_pkg
creating ./my_yn_pkg/my_yn_pkg/__init__.py
creating folder ./my_yn_pkg/test
creating ./my_yn_pkg/test/test_copyright.py
creating ./my_yn_pkg/test/test_flake8.py
creating ./my_yn_pkg/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 identifiers:
```

### Step 2: Create your node file

Before creating a node ,ensure you are inside the correct package

bash

`cd ~/ros2_ws/src/my_py_package/my_yn_package`

bash

`touch yn.py`

### Explanation:

- touch → creates an empty file
- yn.py → name of your node file

### Step 3: Make it executable

bash

`chmod +x yn.py`

### Explanation:

- chmod → change file permission
- +x → make the file executable
- yn.py → your node file

### Step 4: Open VS code or nano to paste the python code

bash(For vscode)

`code .`

bash(For nano)

`nano yn.py`

### Step 5: Paste the python code in the created node and save it.

#### Code

`# Import the rclpy library (ROS 2 client library for Python)`

`import rclpy`

`# Import the Node class from rclpy.node module`

`from rclpy.node import Node`

# Define a custom node class named MyNode, which inherits from rclpy's Node

```
class MyNode(Node):
```

```
    def __init__(self):
```

```
        # Initialize the node with the name 'my_node'
```

```
        super().__init__('my_node')
```

```
        # Print a log message to the console
```

```
        self.get_logger().info('Hello, ROS 2!')
```

# Define the main function - the entry point of the ROS 2 Python program

```
def main(args=None):
```

```
    # Initialize the ROS 2 communication system
```

```
    rclpy.init(args=args)
```

```
    # Create an instance of the custom node (MyNode)
```

```
    node = MyNode()
```

```
    try:
```

```
        # Keep the node alive so it can process callbacks (if any)
```

```
        rclpy.spin(node)
```

```
    except KeyboardInterrupt:
```

```
        # Handle the case when the program is stopped with Ctrl+C
```

```
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
```

```
    finally:
```

# Cleanly destroy the node

```
node.destroy_node()
```

# Shut down the ROS 2 communication system

```
rcipy.shutdown()
```

# Ensure that the main function runs when the script is executed directly

```
if __name__ == '__main__':
```

```
    main()
```

The screenshot shows a ROS 2 development environment. On the left, a terminal window displays the following commands and output:

```
skyharp@DESKTOP-OSB5VRA:~/ros2_ws/src$ cd ~/ros2_ws/src
skyharp@DESKTOP-OSB5VRA:~/ros2_ws/src$ cd my_yn_pkg
skyharp@DESKTOP-OSB5VRA:~/ros2_ws/src/my_yn_pkg$ touch yn.py
skyharp@DESKTOP-OSB5VRA:~/ros2_ws/src/my_yn_pkg$ code .
skyharp@DESKTOP-OSB5VRA:~/ros2_ws/src/my_yn_pkg$
```

On the right, a code editor shows the contents of `yn.py`:

```
1 import rcipy
2 from rcipy.node import Node
3
4 class MyNode(Node):
5     def __init__(self):
6         super().__init__('my_node')
7         self.get_logger().info('Hello, ROS 2!')
8
9 def main(args=None):
10     rcipy.init(args=args)
11     node = MyNode()
12     try:
13         rcipy.spin(node)
14     except KeyboardInterrupt:
15         node.get_logger().info('Keyboard Interrupt (SIGINT)')
16     finally:
17         node.destroy_node()
18         rcipy.shutdown()
19
20 if __name__ == '__main__':
21     main()
```

## Step 6:Edit setup.py:

Open `~/ros2_ws/src/my_yn_pkg/setup.py` and make sure it looks like this:

Add the line in entry-point:

```
entry_points={
```

```
    'console_scripts': [  
        'yn = my_yn_pkg.yn:main',  
    ],  
},
```

## Step 7: Build the Package

Go back to workspace root and build:

bash

```
cd ~/ros2_ws
```

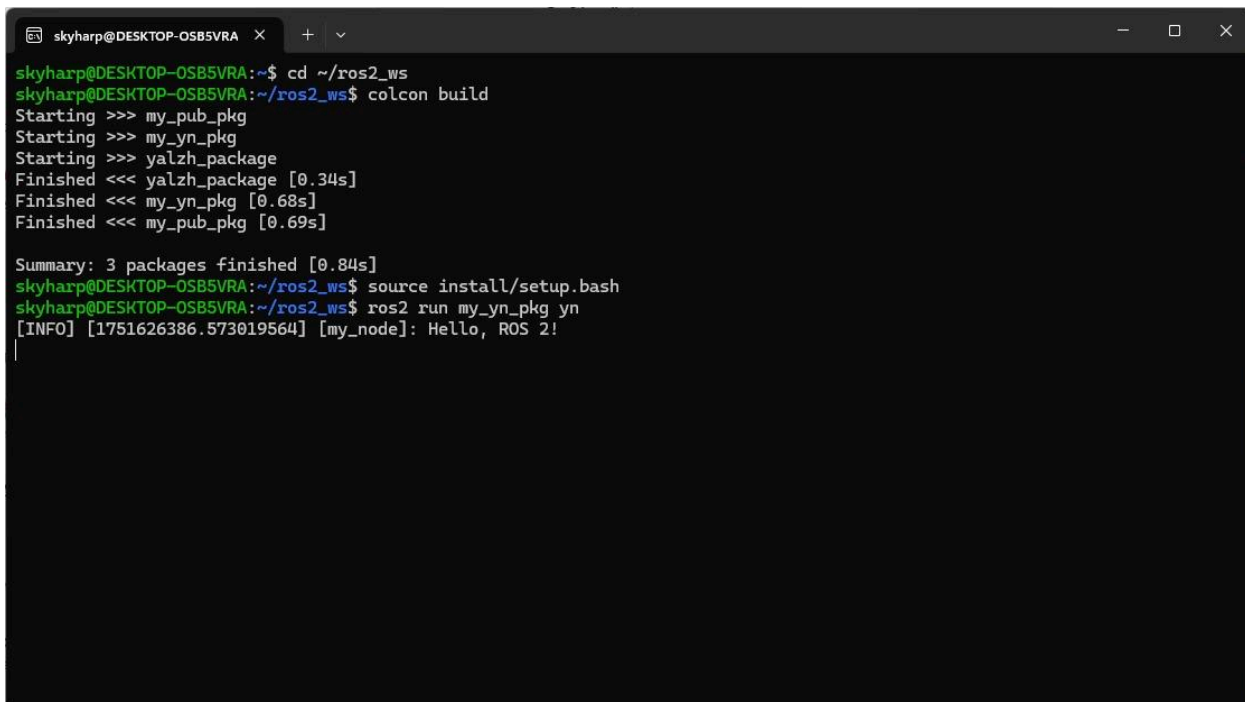
```
colcon build
```

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

Now run the node using:

bash

```
ros2 run my_yn_pkg yn
```



```
skyharp@DESKTOP-OSB5VRA x + v  
skyharp@DESKTOP-OSB5VRA:~$ cd ~/ros2_ws  
skyharp@DESKTOP-OSB5VRA:~/ros2_ws$ colcon build  
Starting >>> my_pub_pkg  
Starting >>> my_yn_pkg  
Starting >>> yalzh_package  
Finished <<< yalzh_package [0.34s]  
Finished <<< my_yn_pkg [0.68s]  
Finished <<< my_pub_pkg [0.69s]  
  
Summary: 3 packages finished [0.84s]  
skyharp@DESKTOP-OSB5VRA:~/ros2_ws$ source install/setup.bash  
skyharp@DESKTOP-OSB5VRA:~/ros2_ws$ ros2 run my_yn_pkg yn  
[INFO] [1751626386.573019564] [my_node]: Hello, ROS 2!
```

During this process,I have faced an error ,in which my node is outside the package(inner package which holds the python module).To rectify this,I moved my node inside my package and then built the workspace to get the output.

### **Step 1: Make Sure Directory Structure is Correct**

Check the following folder exists:

bash

```
ls ~/ros2_ws/src/my_yn_pkg/my_yn_pkg/
```

You should see yn.py there. If not, move it:

bash

```
mv ~/ros2_ws/src/my_yn_pkg/yn.py ~/ros2_ws/src/my_yn_pkg/my_yn_pkg/
```

### **Step 2: Edit setup.py**

bash

```
code ~/ros2_ws/src/my_yn_pkg/setup.py
```

Make sure it contains this in the setup() function:

```
entry_points={  
    'console_scripts': [  
        'yn = my_yn_pkg.yn:main',  
    ],  
},
```

This makes yn the executable name.

### **Step 3: Rebuild and Run**

bash

`cd ~/ros2_ws`

`colcon build`

`source install/setup.bash`

`ros2 run my_yn_pkg yn`

```
skyharp@DESKTOP-0S85VRA: ~$ cd ~/ros2_ws
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ source install/setup.bash
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ cd ~/ros2_ws/src/my_py_pkg/my_yn_pkg
-bash: cd: /home/skyharp/ros2_ws/src/my_py_pkg/my_yn_pkg: No such file or directory
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ cd ~/ros2_ws/src/my_yn_pkg
skyharp@DESKTOP-0S85VRA: ~/ros2_ws/src/my_yn_pkg$ touch yn.py
skyharp@DESKTOP-0S85VRA: ~/ros2_ws/src/my_yn_pkg$ code .
skyharp@DESKTOP-0S85VRA: ~/ros2_ws/src/my_yn_pkg$ chmod +x yn.py
skyharp@DESKTOP-0S85VRA: ~/ros2_ws/src/my_yn_pkg$ cd ../..
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ ros2 run MY_YN_PKG yn.py
Package 'MY_YN_PKG' not found
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ ros2 run MY_YN_PKG yn.py
Package 'MY_YN_PKG' not found
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ ros2 run my_yn_pkg yn.py
No executable found
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ ls ~/ros2_ws/src/MY_YN_PKG/my_yn_pkg
ls: cannot access '/home/skyharp/ros2_ws/src/MY_YN_PKG/my_yn_pkg': No such file or directory
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ ls ~/ros2_ws/src/my_yn_pkg/my_yn_pkg/
__init__.py
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ mv ~/ros2_ws/src/my_yn_pkg/yn.py ~/ros2_ws/src/my_yn_pkg/my_yn_pkg/
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ cd ~/ros2_ws
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ colcon build
Starting >>> my_pub_pkg
Starting >>> my_yn_pkg
Starting >>> yalzh_package
Finished <<< yalzh_package [0.34s]
Finished <<< my_pub_pkg [0.78s]
Finished <<< my_yn_pkg [0.70s]

Summary: 3 packages finished [0.85s]
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ source install/setup.bash
skyharp@DESKTOP-0S85VRA: ~/ros2_ws$ ros2 run my_yn_pkg yn
[INFO] [1751625311.448677562] [my_node]: Hello, ROS 2!
```

## 2. Topic

A Topic is a named channel where nodes send or receive messages. It is used for continuous data sharing.

### Example:

- sensor\_node publishes laser scan data to /scan
- navigation\_node subscribes to /scan to avoid obstacles
- battery\_node publishes to /battery\_status



- ui\_node subscribes to /battery\_status to show charge level

It's a one-way flow of data: Publish → Subscribe.

### **Real Use:**

1.To see active topics:

bash

**ros2 topic list**

### **Explanation:**

- ros2 topic → Work with topics
- list → Show all active topics

2.To see battery status:

bash

**ros2 topic echo /battery\_status**

### **Explanation:**

- echo → View live data from topic
- /battery\_status → Topic name (sent by battery node)

To publish manually to a topic:

bash

**ros2 topic pub /cmd\_vel geometry\_msgs/Twist "{linear: {x: 0.5}, angular: {z: 0.0}}"**

### **Explanation:**

- pub → Publish data to a topic
- /cmd\_vel → Velocity command topic

- geometry\_msgs/Twist → Message type (linear and angular velocity)
- "{ linear: {x: 0.5}, angular: {z: 0.0} }" → Move forward at 0.5 m/s, no turning

This sends a motion command to the robot.

## **Publisher Example:**

### **1. Create the Package**

bash

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

```
ros2 pkg create --build-type ament_python my_robot_controller
```

### **2.Go Into the Package Folder**

bash

```
cd my_robot_controller
```

### **3. Create Node Files**

**Before creating a node ,ensure you are inside the correct package**

bash

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

bash

```
touch draw_circle.py
```

Creates Python script for publishing the data.

### **4.Make the Python Files Executable**

bash

```
chmod +x draw_circle.py
```

## 5. Open Vs code and paste the following code

bash

code .

Code:

```
    rclpy.shutdown() # Shut down the ROS 2 client library when done
#!/usr/bin/env python3 # Shebang line indicating the script should be
run with Python 3

import rclpy # Import the ROS 2 Python client library

from rclpy.node import Node # Import the Node class from rclpy to create
ROS nodes

from geometry_msgs.msg import Twist # Import the Twist message type used
to control robot velocity

# Define a class called DrawCircle that inherits from Node

class DrawCircle(Node):

    def __init__(self):

        super().__init__("draw_circle")

        # Initialize the node with the name "draw_circle"

        self.cmd_vel_pub_ = self.create_publisher(Twist,
"/turtle1/cmd_vel",10)

# Create a publisher that will publish Twist messages to the topic
/turtle1/cmd_vel. The queue size is set to 10 (buffer size if subscriber
is slow)

        self.timer_ = self.create_timer(0.5,self.send_velocity_command)

# Create a timer that calls the send_velocity_command method every 0.5
seconds
```

```

        self.get_logger().info("draw circle node started")

# Log a message to indicate that the node has started

    def send_velocity_command(self): # Define the function that will send
velocity commands to make the turtle move in a circle

        msg=Twist()      # Create a new Twist message

        msg.linear.x = 2.0      # Set linear velocity in the x-direction
(forward speed)

        msg.angular.z = 1.0      # Set angular velocity around the z-axis
(rotation)

        self.cmd_vel_pub_.publish(msg)      # Publish the message to the
/turtle1/cmd_vel topic

# Define the main function that will be executed when the script runs
def main(args=None):

    rclpy.init(args=args)      # Initialize the ROS 2 Python client library

    node=DrawCircle()          # Create an instance of the DrawCircle node

    rclpy.spin(node)           # Keep the node alive and responsive to callbacks
(like the timer)

```

## 5. Edit setup.py to Add Executables

bash

[code ~/ros2\\_ws/src/my\\_yn\\_pkg/setup.py](#)

**Modify setup.py:**

```
entry_points={  
    'console_scripts': [  
        "draw_circle = my_robot_controller.draw_circle:main"  
    ],  
}
```

## 6. Build the Package\*

Go to the root workspace:

```
bash
```

```
cd ~/ros2_ws
```

```
colcon build
```

## 7. Source the Workspace

```
bash
```

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

Loads the package environment so ROS 2 can find your new nodes.

## 8. Run the Publisher

```
bash
```

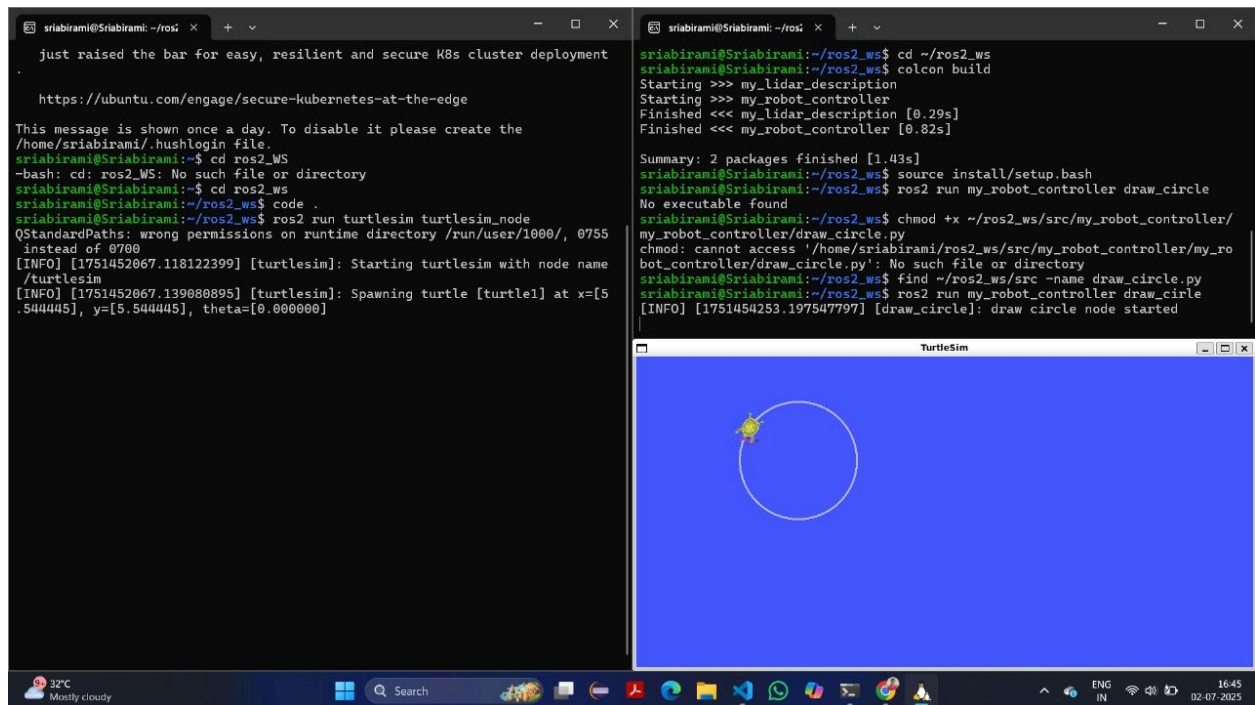
```
ros2 run my_robot_controller draw_circle.py
```

This starts the publisher node and begins sending messages on topic chatter.

```
bash
```

```
ros2 run turtlesim turtlesim_node
```

## Output:



```
sriabirami@sriabirami: ~/ros1 x + v
just raised the bar for easy, resilient and secure K8s cluster deployment

https://ubuntu.com/engage/secure-kubernetes-at-the-edge

This message is shown once a day. To disable it please create the
/home/sriabirami/.hushlogin file.
sriabirami@sriabirami:~$ cd ros2_ws
-bash: cd: ros2_ws: No such file or directory
sriabirami@sriabirami:~$ cd ros2_ws
sriabirami@sriabirami:~/ros2_ws$ code .
sriabirami@sriabirami:~/ros2_ws$ ros2 run turtlesim turtlesim_node
QStandardPaths: wrong permissions on runtime directory /run/user/1000/, 0755
instead of 0700
[INFO] [1751452067.118122399] [turtlesim]: Starting turtlesim with node name
/turtlesim
[INFO] [1751452067.139080895] [turtlesim]: Spawning turtle [turtle1] at x=[5
.5444445], y=[5.5444445], theta=[0.000000]

sriabirami@sriabirami:~/ros2_ws$ cd ~/ros2_ws
sriabirami@sriabirami:~/ros2_ws$ colcon build
Starting >>> my_lidar_description
Starting >>> my_robot_controller
Finished <<< my_lidar_description [0.29s]
Finished <<< my_robot_controller [0.82s]

Summary: 2 packages finished [1.43s]
sriabirami@sriabirami:~/ros2_ws$ source install/setup.bash
sriabirami@sriabirami:~/ros2_ws$ ros2 run my_robot_controller draw_circle
No executable found
sriabirami@sriabirami:~/ros2_ws$ chmod +x ~/ros2_ws/src/my_robot_controller/
my_robot_controller/draw_circle.py
chmod: cannot access '/home/sriabirami/ros2_ws/src/my_robot_controller/my_ro
bot_controller/draw_circle.py': No such file or directory
sriabirami@sriabirami:~/ros2_ws$ find ~/ros2_ws/src -name draw_circle.py
sriabirami@sriabirami:~/ros2_ws$ ros2 run my_robot_controller draw_circle
[INFO] [1751454253.197547797] [draw_circle]: draw circle node started
```

TurtleSim

The TurtleSim window shows a blue background with a white circle. A small yellow turtle icon is positioned at the center of the circle, having just completed a full rotation.

## Output of the Code:

When you run it, it will:

- Every 0.5 seconds, publish a Twist message to the topic /turtle1/cmd\_vel with:
- linear.x = 2.0 → move forward at speed 2
- angular.z = 1.0 → rotate at angular speed 1

This makes the turtle in the Turtlesim simulator move in a circular path.

### 1. Publisher in the Code:

```
self.cmd_vel_pub_ = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)
```

This is the publisher object.

- It publishes messages of type Twist
- On the topic: /turtle1/cmd\_vel
- With a queue size of 10 (helps in buffering if subscriber is slow)

This publisher is used in the `send_velocity_command()` function.

## 2. Topic Used: `/turtle1/cmd_vel`

- This topic is a standard topic used by Turtlesim
- It receives messages of type `geometry_msgs/msg/Twist`
- Twist message has:
  - linear (x, y, z) — for moving forward/backward
  - angular (x, y, z) — for rotation around axes

So, the turtle moves forward and turns at the same time → draws a circle

## 3. Subscriber in this Case:

The Turtlesim node (the GUI simulator) acts as the subscriber here.

Turtlesim internally subscribes to this topic to move the turtle.

So:

- Your node is the publisher
- Turtlesim is the subscriber

## Subscriber Example:

### 1. Create the Package

bash

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

```
ros2 pkg create --build-type ament_python my_robot_controller
```

## 2.Go Into the Package Folder

bash

```
cd my_robot_controller
```

## 3. Create Node Files

Before creating a node ,ensure you are inside the correct package

bash

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

bash

```
touch pose_subscriber.py
```

Creates Python script for Subscribing the data.

## 4.Make the Python Files Executable

bash

```
chmod +x pose_subscriber.py
```

## 5.Open Vs code and paste the following code

bash

code .

**Code:**

```
#!/usr/bin/env python3 # Shebang line tells the system to use Python 3
to run this script
import rclpy           # Import the ROS 2 Python client library
from rclpy.node import Node #Import the base class for creating ROS nodes
from turtlesim.msg import Pose #Import the Pose message type used to get
the turtle's position
```



```

class PoseSubscriberNode(Node): # Define a class that inherits from Node
    to create a custom ROS2 node

    def __init__(self):
        super().__init__("pose_subscriber")
#Initialize the node with the name "pose_subscriber"

        self.pose_subscriber_ = self.create_subscription(Pose,
"/turtle1/pose",self.pose_callback,10)
        # Create a subscriber to the "/turtle1/pose" topic, which sends
Pose messages.When a message is received, it will call the pose_callback
function

        def pose_callback(self,msg: Pose): #Callback function that is
automatically called whenever a new Pose message is received
            self.get_logger().info("(" +str(msg.x)+", "+str(msg.y)+") ")
            #Log the current x and y position of the turtle

def main(args=None): #Main function to initialize, run, and shut down
the node
    rclpy.init(args=args) #Initialize ROS 2 communication
    node=PoseSubscriberNode()#Create an instance of the subscriber node
    rclpy.spin(node) #Keep the node alive and responsive to messages
    rclpy.shutdown() #Shutdown the ROS 2 system once the node is stopped

```

## 5. Edit setup.py to Add Executables

bash

code ~/ros2\_ws/src/my\_yn\_pkg/setup.py

Modify setup.py:

entry\_points={

    'console\_scripts': [

        "pose\_subscriber = my\_robot\_controller.pose\_subscriber:main",

```
],
```

```
},
```

## 6. Build the Package\*

Go to the root workspace:

```
bash
```

```
cd ~/ros2_ws
```

```
colcon build
```

## 7. Source the Workspace

```
bash
```

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

Loads the package environment so ROS 2 can find your new nodes.

## 8. Run the Subscriber:

```
bash
```

```
ros2 run turtlesim turtlesim_node
```

```
bash
```

```
ros2 run my_robot_controller pose_subscriber.py
```

This starts the subscriber node and begins sending messages on topic chatter.

Output:  subscriber.mp4

## Output – How It Works

1. Turtlesim is running in the background, and it keeps track of the turtle's position in real time.
2. Turtlesim publishes the current position of the turtle (x, y, orientation, etc.) to the `/turtle1/pose` topic using the Pose message format.
3. The ROS 2 node `PoseSubscriberNode` is running and subscribes to this topic.
4. Every time a new pose message is published, your node automatically receives it through the callback function `pose_callback`.
5. The callback function extracts the x and y values from the Pose message and prints them to the terminal.

## 1. Publisher:

The publisher in this setup is Turtlesim's internal node, not the code you wrote.

Details:

- Turtlesim internally has a node that tracks and updates the turtle's position.
- This internal node publishes data to the topic `/turtle1/pose`
- The message type it uses is: `turtlesim/msg/Pose`

Message Fields include:

- `x`: horizontal position
- `y`: vertical position
- `theta`: orientation angle
- `linear_velocity`: speed forward
- `angular_velocity`: turning rate

So basically, Turtlesim continuously publishes the turtle's live position on `/turtle1/pose`.

## 2. Topic: /turtle1/pose

- /turtle1/pose is the topic that carries the Pose of the turtle
- The message type is Pose (from turtlesim/msg/Pose)
- This topic is used to monitor the real-time position and orientation of the turtle

## 3. Subscriber:

Your Python class PoseSubscriberNode creates a ROS 2 subscriber.

What it does:

- It subscribes to the /turtle1/pose topic
- Waits for any new Pose message to arrive
- When a message arrives, it calls pose\_callback().

We can also have publisher and subscriber in the same node. Let us see the steps to create it and how it works.

## Publisher and Subscriber in the same node:

In this, a node is created in the same format as how a normal node is created and the subscriber and publisher are modified only inside the node

### 1. Create the Package

bash

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

```
ros2 pkg create --build-type ament_python my_robot_controller
```

## 2.Go Into the Package Folder

bash

```
cd my_robot_controller
```

## 3. Create Node Files

Before creating a node ,ensure you are inside the correct package

bash

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

bash

```
touch turtle_controller.py
```

Creates Python script for publishing the data.

## 4.Make the Python Files Executable

bash

```
chmod +x turtle_controller.py
```

## 5.Open Vs code and paste the following code

bash

code .

Code:

```
#!/usr/bin/env python3          # Shebang: Run with Python 3
import rclpy                    # ROS 2 Python client library
from rclpy.node import Node     # Base class for creating ROS nodes
from turtlesim.msg import Pose  # Pose message: contains x, y, theta of
turtle                         #
from geometry_msgs.msg import Twist #Twist message: for velocity commands
```

```

from turtlesim.srv import SetPen # Service to change pen color, width,
etc.

from functools import partial    #Used to pass extra arguments to callbacks

class MyturtleControllerNode(Node):# Define a custom Node class to control
the turtle
    def __init__(self):
        super().__init__("turtle_controller") # Initialize the node with
a name

        self.previous_x_=0 # Store the last x-position to detect crossing
midpoint (5.5)

        self.cmd_vel_publisher_ = self.create_publisher(Twist,
"/turtle1/cmd_vel",10)    # Publisher to send velocity commands to the
turtle

        self.pose_subscriber_ = self.create_subscription(Pose,
"/turtle1/pose",self.pose_callback,10)    # Subscriber to listen to
turtle's position updates

        self.get_logger().info("Turtle controller has been started")
                                # Log that node is active

    def pose_callback(self,pose: Pose): #Called whenever a new Pose
message is received
        cmd=Twist() # Create a velocity command message

        #If the turtle is near the boundary, turn in a circle
        if pose.x>9.0 or pose.x < 2.0 or pose.y>9.0 or pose.y<2.0:
            cmd.linear.x = 1.0
            cmd.angular.z=0.9

        # Move straight at higher speed in the center
        else:
            cmd.linear.x = 5.0
            cmd.angular.z=0.0

        self.cmd_vel_publisher_.publish(cmd) # Publish velocity command

    # Check if turtle crosses x = 5.5 from left to right
    if pose.x>5.5 and self.previous_x_<=5.5:
        self.previous_x_=pose.x
        self.get_logger().info("Set colour to red")

```

```

        self.call_set_pen_service(255,0,0,10,0) #Change pen to red

#Check if turtle crosses x = 5.5 from right to left
    elif pose.x<=5.5 and self.previous_x>5.5:
        self.previous_x=pose.x
        self.get_logger().info("Set colour to green")
        self.call_set_pen_service(0,255,0,3,0) #Change pen to green

#Call the /turtle1/set_pen service to change the pen properties
    def call_set_pen_service(self,r,g,b,width,off):
        client=self.create_client(SetPen,"/turtle1/set_pen")# Create a
service client
# Wait until the service is available
        while not client.wait_for_service(1.0):
            self.get_logger().warn("waiting for service...")

#Prepare the request with desired color and pen settings
        request= SetPen.Request()
        request.r=r
        request.g=g
        request.b=b
        request.width=width
        request.off=off
        #Send the request asynchronously
        future=client.call_async(request)
        future.add_done_callback(partial(self.callback_set_pen))
                                # Call function after response

#Callback function to handle the response of set_pen service
    def callback_set_pen(self,future):
        try:
            response=future.result() # Get result from future (no return
value in this case)
        except:
            self.get_logger().error("service call failed: %r"%(e,))
                                #Handle exceptions

# Main entry point of the program
def main(args=None):
    rclpy.init(args=args) # Initialize the ROS 2 communication

```

```
node=MyturtleControllerNode()    # Create node instance
rclpy.spin(node)                  #Keep node alive to process callbacks
rclpy.shutdown()                  # Shutdown when done
```

## 5. Edit setup.py to Add Executables

bash

`code ~/ros2_ws/src/my_yn_pkg/setup.py`

Modify setup.py:

```
entry_points={
    'console_scripts': [
        "turtle_controller = my_robot_controller.turtle_controller:main",
    ],
},
```

## 6. Build the Package\*

Go to the root workspace:

bash

`cd ~/ros2_ws`

`colcon build`

## 7. Source the Workspace

bash

`source install/setup.bash`

Loads the package environment so ROS 2 can find your new nodes.



## 8. Run the Node:

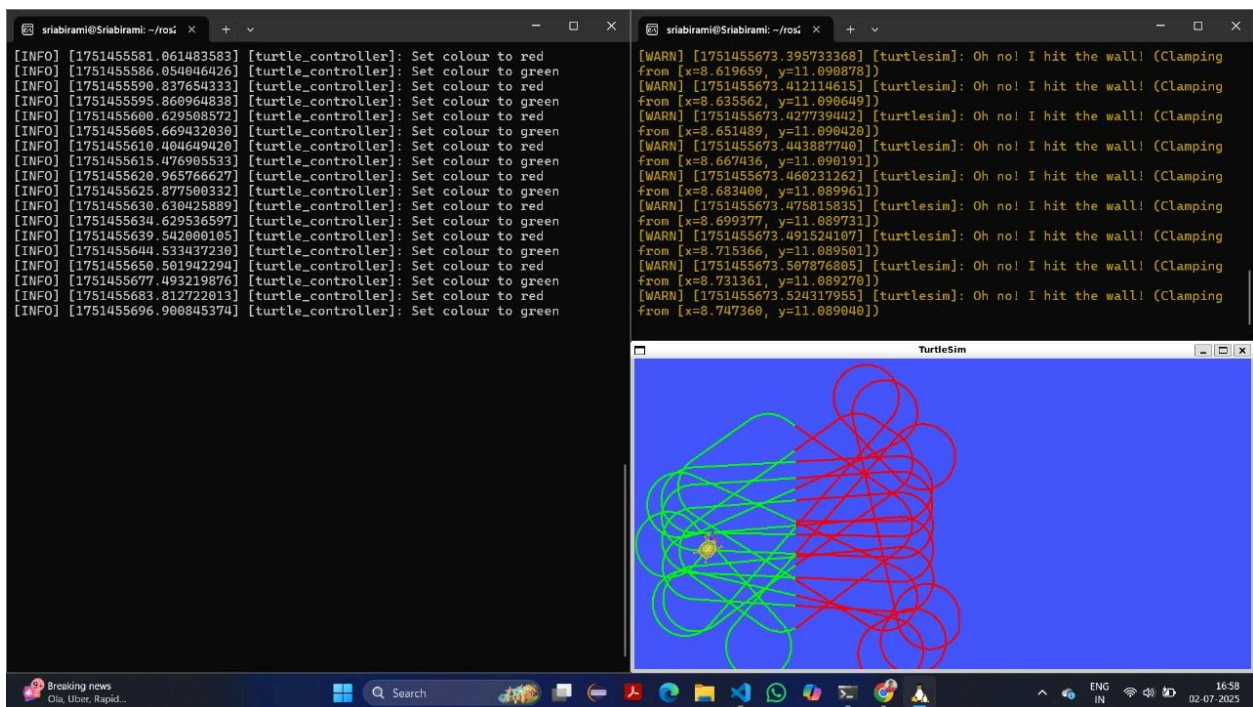
bash

ros2 run turtlesim turtlesim\_node

bash

ros2 run my\_robot\_controller turtle\_controller.py

## Output:



```
[INFO] [1751455581.061483583] [turtle_controller]: Set colour to red
[INFO] [1751455586.054046426] [turtle_controller]: Set colour to green
[INFO] [1751455590.837654333] [turtle_controller]: Set colour to red
[INFO] [1751455595.860964838] [turtle_controller]: Set colour to green
[INFO] [1751455600.629508572] [turtle_controller]: Set colour to red
[INFO] [1751455605.669432030] [turtle_controller]: Set colour to green
[INFO] [1751455610.404649420] [turtle_controller]: Set colour to red
[INFO] [1751455615.476905533] [turtle_controller]: Set colour to green
[INFO] [1751455620.965766627] [turtle_controller]: Set colour to red
[INFO] [1751455625.877509332] [turtle_controller]: Set colour to green
[INFO] [1751455630.630492889] [turtle_controller]: Set colour to red
[INFO] [1751455634.629536897] [turtle_controller]: Set colour to green
[INFO] [1751455639.542000105] [turtle_controller]: Set colour to red
[INFO] [1751455644.533437230] [turtle_controller]: Set colour to green
[INFO] [1751455650.501942294] [turtle_controller]: Set colour to red
[INFO] [1751455677.493219876] [turtle_controller]: Set colour to green
[INFO] [1751455683.812722013] [turtle_controller]: Set colour to red
[INFO] [1751455696.908845374] [turtle_controller]: Set colour to green

[WARN] [1751455673.395733368] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.619659, y=11.090878])
[WARN] [1751455673.412114615] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.635562, y=11.090649])
[WARN] [1751455673.427739442] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.651489, y=11.090426])
[WARN] [1751455673.443887740] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.667436, y=11.090191])
[WARN] [1751455673.460231262] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.683400, y=11.089961])
[WARN] [1751455673.475815935] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.699377, y=11.089731])
[WARN] [1751455673.491524107] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.715366, y=11.089501])
[WARN] [1751455673.507876805] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.731361, y=11.089270])
[WARN] [1751455673.524317955] [turtlesim]: Oh no! I hit the wall! (Clamping
from [x=8.747360, y=11.089040])
```

## Output – How It Works

The node turtle\_controller starts and logs:

1. As the turtle moves, the node receives its position through the /turtle1/pose topic (handled by the subscriber).
2. Every time a new position (Pose) is received:

- If the turtle is near the edge of the screen ( $x < 2$  or  $x > 9$ ,  $y < 2$  or  $y > 9$ ), the node sends a circular movement command to make the turtle turn.
  - If the turtle is in the center, it sends a straight fast movement command.
3. When the turtle crosses  $x = 5.5$ :
- From left to right, the pen color is set to red, and this is logged.
  - From right to left, the pen color is set to green, and this is logged:
4. The turtle moves continuously, drawing with different pen colors based on its direction.

## 1. Publisher

```
self.cmd_vel_publisher_ = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)
```

Details:

- Publishes messages of type Twist.
- Topic: /turtle1/cmd\_vel
- Controls the turtle's linear and angular velocity.
- Used in pose\_callback() to move the turtle based on its position.

Purpose:

Sends movement commands to the turtle:

- Fast forward when in the center.
- Rotate near boundaries to stay inside.

## 2. Subscriber

```
self.pose_subscriber_ = self.create_subscription(Pose, "/turtle1/pose",  
self.pose_callback, 10)
```

Details:

- Subscribes to /turtle1/pose
- Message type: Pose (includes x, y, theta, velocity)
- Callback: pose\_callback()

Purpose:

- Continuously receives the current position of the turtle.
- Triggers decisions: move straight, turn, or change pen color.

## 3. Topics Used

1. /turtle1/cmd\_vel

- Type: geometry\_msgs/msg/Twist
- Used to control the velocity (linear + angular) of the turtle.
- Published by your node.

2. /turtle1/pose

- Type: turtlesim/msg/Pose

- Provides real-time position of the turtle.
- Published by Turtlesim, subscribed by your node.

### 3. /turtle1/set\_pen

- Type: turtlesim/srv/SetPen
- A service, not a topic.
- Used to change pen color, width, or turn it off.
- Called by your node when crossing  $x = 5.5$ .

## 3. Service

A Service is used when one node wants to ask another node to do something once, and get a response back.

### Example:

- User wants to reset the robot's map
- ui\_node sends a request: "Reset map?"
- map\_node replies: "Map reset successful."

It's a two-way interaction: Request → Response.

### ROS2 Service with AddTwoInts Example:

#### Step 1: Open Your Package in VS Code

From your workspace:

```
bash
```

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

```
code .
```

## Step 2: Manually Create Service Scripts

Inside my\_robot\_controller/my\_robot\_controller/, create two files:

**Before creating a node ,ensure you are inside the correct package**

```
bash
```

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

```
bash
```

```
touch add_two_ints_server.py
```

```
bash
```

```
code .
```

**Code:**

```
import rclpy # ROS 2 Python client library
```

```
from rclpy.node import Node # Base class for creating ROS 2 nodes
```

```
from example_interfaces.srv import AddTwoInts # Built-in service type with request: a,  
b; response: sum
```

```
class AddTwoIntsServer(Node): # Defines a new node class for the service server
```

```
    def __init__(self):
```

```

    super().__init__('add_two_ints_server') # Initialize node with name
'add_two_ints_server'

    self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_callback)

    # Create service: type=AddTwoInts, name=/add_two_ints, callback=add_callback

    self.get_logger().info("Service server ready on 'add_two_ints'")

# Print log when server is ready

```

```

def add_callback(self, request, response): # Callback called when client sends
request

    response.sum = request.a + request.b # Add the two numbers and store in
response

    self.get_logger().info(f"Request: {request.a} + {request.b} = {response.sum}")

    # Log the request and result

    return response # Return the response to the client

```

```

def main(args=None):

    rclpy.init(args=args) # Initialize rclpy

    node = AddTwoIntsServer() # Create node instance

    rclpy.spin(node) # Keep node alive and listen for service requests

    rclpy.shutdown() # Shutdown cleanly when done

```

bash

**touch add\_two\_ints\_client.py**

bash

**code .**

## Code

Paste this code in the node created.

```
import rclpy # ROS 2 Python client library

from rclpy.node import Node # Base class for creating ROS 2 nodes

from example_interfaces.srv import AddTwoInts # Built-in AddTwoInts service type


class AddTwoIntsClient(Node): # Client node that calls the add_two_ints service

    def __init__(self):

        super().__init__('add_two_ints_client') # Node name: add_two_ints_client

        self.client = self.create_client(AddTwoInts, 'add_two_ints')

        # Create client for service 'add_two_ints' of type AddTwoInts

        while not self.client.wait_for_service(timeout_sec=1.0):

            # Wait until server is available

            self.get_logger().info("Waiting for service server...") # Log every second while
waiting

        self.req = AddTwoInts.Request() # Prepare a request object to send data


    def send_request(self, a, b):

        self.req.a = a # Set first number in request

        self.req.b = b # Set second number in request
```

```

future = self.client.call_async(self.req) # Send request asynchronously

rcipy.spin_until_future_complete(self, future) # Wait for the result

return future.result() # Return the response from the server

```

```
def main(args=None):
```

```

    rcipy.init(args=args) # Initialize rcipy

    node = AddTwoIntsClient() # Create client node

    response = node.send_request(5, 6) # Send 5 and 6 to the server

    node.get_logger().info(f"Sum: {response.sum}")

    # Log the result (should be 11)

    rcipy.shutdown() # Clean shutdown

```

### Step 3: Make Scripts Executable

```
bash
```

```
chmod +x add_two_ints_server.py
```

```
chmod +x add_two_ints_client.py
```

### Step 4: Update setup.py

Open my\_robot\_controller/setup.py

Add your new scripts inside entry\_points > console\_scripts

```
entry_points={
```

```
    'console_scripts': [    # Add your service scripts
```

```
        'add_two_ints_server = my_robot_controller.add_two_ints_server:main',
```



```
'add_two_ints_client = my_robot_controller.add_two_ints_client:main',  
  
],  
  
},
```

## Step 5: Update package.xml

Open my\_robot\_controller/package.xml

Add these if not already there:

```
<exec_depend>rclpy</exec_depend>
```

```
<exec_depend>example_interfaces</exec_depend>
```

These provide a Python client library and the service message type.

## Step 6: Build the Package

bash

```
cd ~/ros2_ws
```

```
colcon build
```

Then source the overlay:

bash

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

## Step 7: Run and Test

**Terminal 1 – Start the server:**

bash

```
ros2 run my_robot_controller add_two_ints_server
```

**Terminal 2 – Start the client:**

bash

`ros2 run my_robot_controller add_two_ints_client`

```
sriabirami@Sriabirami: ~/ros2  × + ▾  
sriabirami@Sriabirami:~$ cd ~/ros2_ws  
sriabirami@Sriabirami:~/ros2_ws$ ros2 run my_robot_controller add_two_ints_server  
[INFO] [1751626444.951837196] [add_two_ints_server]: Service server ready on 'add_two_ints'  
[INFO] [1751626471.470092091] [add_two_ints_server]: Request: 6 + 4 = 10
```

```
sriabirami@Sriabirami: ~/ros2  × + ▾  
sriabirami@Sriabirami:~$ cd ~/ros2_ws  
sriabirami@Sriabirami:~/ros2_ws$ ros2 run my_robot_controller add_two_ints_client  
[INFO] [1751626471.478408216] [add_two_ints_client]: Sum: 10  
sriabirami@Sriabirami:~/ros2_ws$ |
```

## ROS 2 Service Communication Flow:

### Components:

#### 1 . **Service Server Node** – AddTwoIntsServer

- ↪ Offers the service called /add\_two\_ints
- ↪ Waits for incoming requests and responds with the sum.

#### 2 . **Service Client Node** – AddTwoIntsClient

- ↪ Sends a request to /add\_two\_ints with two integers (e.g., 5 and 6)
- ↪ Waits for the server to respond and prints the result.

### Output-Step-by-Step Flow :

1.Start the **server node** first:

bash

**ros2 run my\_package add\_two\_ints\_server**

You will see:

**[INFO] [add\_two\_ints\_server]: Service server ready on 'add\_two\_ints'**

2.Now run the **client node** in another terminal:

bash

**ros2 run my\_package add\_two\_ints\_client**

Internally, this happens:

- The client waits until the server is available.
- Then it sends a request with  $a = 5$  and  $b = 6$ .

The server receives the request and logs:

**[INFO] [add\_two\_ints\_server]: Request:  $5 + 6 = 11$**

The client receives the response and logs:

**[INFO] [add\_two\_ints\_client]: Sum: 11**

**Youtube Video :**

 **ROS2 Tutorial - ROS2 Humble 2H50 [Crash Course]**