ROS Tutorials Summary (Beginner Level) (Theory Only)

ROS:

ROS, short for Robot Operating System, is a powerful open-source framework (predefined, reusable structure that provides a foundation for building software applications) used to develop robotic applications. It's not a true operating system, but rather a set of software libraries, tools and conventions. ROS simplifies the process of building robots by providing tools for connecting different robot components, such as sensors, actuators, and control systems, through a distributed architecture (software system designed to run across multiple interconnected computational nodes, rather than on a single central server).

ROS Nodes:

An executable that performs a computation. Typically, a single node does one specific task (motor control, sensor processing). It is written in Python or C++. Ros nodes use a ROS client library (allows nodes written in different programming languages to communicate (rospy, roscpp)) to communicate with other nodes. Nodes can publish or subscribe to a Topic (see later). Nodes can also provide or use a Service (see later).
Packages can be considered as the "Container" for nodes. So, we can say that a ROS Package is a folder with a standardized structure that organizes Nodes, Libraries, configuration files, launch files, and messages.

ROS PACKAGE:

Smallest independent unit of software in ROS. It provides a way to bundle together files related to a specific piece of functionality or a project. ROS Packages are used due to their modularity (breaking projects into manageable pieces such as one package for sensors, another for navigation etc), Reusability (sharing packages across projects/robots) and Dependency Management (ROS tools such as rosdep, catkin automatically handle dependencies).
A typical ROS package contains the following folders:

- CmakeLists.txt: Defines how to build the package (compiling, dependencies, installation).
- package.xml: Metadata (data that provides information about other data) (package name, version name, author, dependencies)
- src/: Contains compiled language nodes (C++ nodes) (contains source files that must be built into executables using catkin_make) (files in src/ must be declared in CMakeLists.txt to be compiled)
- scripts/: For interpreted language nodes (python), run without compilation. They must be made executable using "chmod +x my_python_node.py".
- launch/: Stores launch files (scripts that automate the startup of multiple ROS nodes, parameters, and configurations with a single command). (Defines how nodes run together). Usually in .py or .xml.
- msg/ and srv/: Custom ROS messages and services type for communication between nodes.
- config/: stores static or dynamic configurations in YAML, JSON or other formats. It is used for ROS parameters (control gains, thresholds), RVIZ configurations, URDF robot configurations.

So, nodes are hosted inside packages and can be written in:

- C++ (built into executables, stored in src/)
- Python (run directly, stored in scripts/)

Key Rules:

- One package -> Many nodes: A single pckage can contain multiple nodes. For example, a “sensors” package might contain a “camera_node” and a “lidar_node”.
- Nodes depend on Packages: Nodes use ROS libraries (roscpp, rospy, ...) declared in package.xml. Nodes can also communicate with other nodes in other packages.

ROS Workspace/Catkin Workspace:

Directory (folder) where you develop, build, and manage ROS packages. It follows a standardized structure that ROS tools (like catkin_make) recognize. It typically contains the following folders:

- src/: Contains the ROS packages (our code), and the CMakeLists.txt file that defines how to build the package, manage dependencies and the installation rules.
- build/ and devel/: Automatically generated files by the CMake (ROS build system). Also contains the devel/setup.bash that must be sourced to use the built packages.

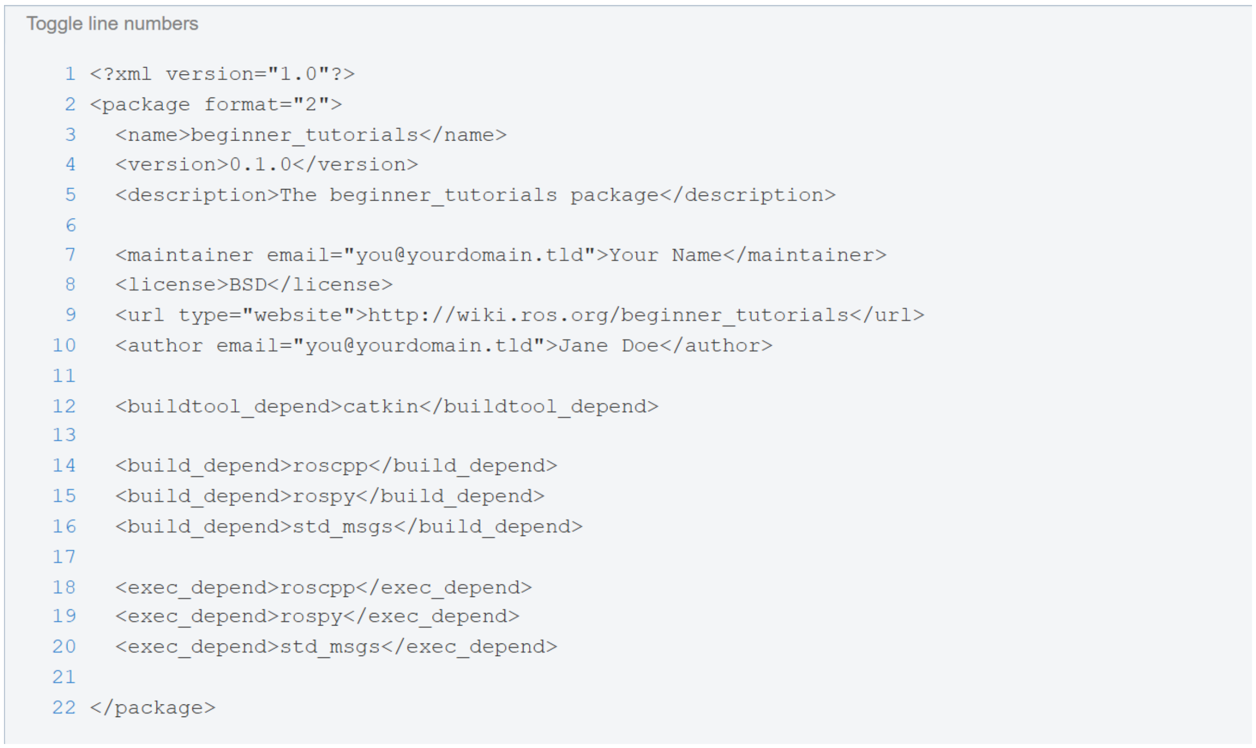
Building Process:

As previously mentioned, every ROS package has its own CMakeLists.txt file, in top level (Workspace level) there is also a single CMakeLists.txt file that is automatically generated when you run catkin_make and is not manually maintained. When you run catkin_make in your catkin_ws following steps happen under the hood:

1. Initial Setup:
   You run catkin_make from the root of the workspace (catkin_ws/**)**. Catkin looks for:
   - A src/ directory (containing ROS packages).
   - A top-level CMakeLists.txt (auto generated if missing).
2. Top-level CMakeLists.txt (Workspace Level):
   This file sets up the catkin environment and discovers all the packages inside src/ (recursively searches src/ for packages by looking for package.xml + CMakeLists.txt)
3. Processing individual packages:
   Reads package.xml (extracts dependencies such as roscpp, std_messages). And then reads the package’s CMakeLists.txt which says how the package is to be built.
4. CMake Configuration and Final Output:
   Processes all the package’s CMakeLists.txt and makes the Makefiles in (catkin_ws/build/). All packages are built and usable in the devel/ folder and generate the environment set-up files.

So always after using building the workspace using catkin_make, go to the devel/ folder and source the setup.bash file to overlay this workspace on top of the environment (so ROS checks your workspace for package before the default ROS path).

The following is a typical package.xml file without comments:



```xml
<?xml version="1.0"?>
<package format="2">
  <name>beginner_tutorials</name>
  <version>0.1.0</version>
  <description>The beginner_tutorials package</description>

  <maintainer email="you@yourdomain.tld">Your Name</maintainer>
  <license>BSD</license>
  <url type="website">http://wiki.ros.org/beginner_tutorials</url>
  <author email="you@yourdomain.tld">Jane Doe</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>

</package>
```

Points and Tips:

- ROS Tools (roscd, etc) only find ROS packages that are within the directories listed in ROS_PACKAGE_PATH.
- Each package must have its own folder (no nested packages nor multiple packages sharing the same directory).
- Packages must be created and stored in the source space directory of the catkin workspace. Creating a catkin Package (Must be done inside the src/ directory). This will create a package_name folder which contains a package.xml and a CMakeLists.txt which have been partially filled out with the information you gave to catkin_create_package. The given dependencies are stored automatically in the package.xml file.
- After you have created a new package, you need to build the package in the catkin workspace using catkin_make.
- The generated package.xml file that is created when you create a package can be customized by changing the description tag, maintainer tag, license tags, dependencies tag (can be build_depend (available at compile time), buildtool_depend (build tool package), exec_depend (packages needed at run time). Usually, we must manually add exec_depend packages in the package.xml, as for the packages they are same as the build_depend packages that are automatically added in the package.xml.
- Remember that once a package has been created it needs to be built to compile the source code, resolve dependencies and generate the necessary files so that the package can be run, tested and integrated into the system. The package can be built if sall of the needed system dependencies are installed.
- Once a package is built, the build folder of the catkin workspace is the location of the build space (where cmake and make are called to configure and build your packages). The devel

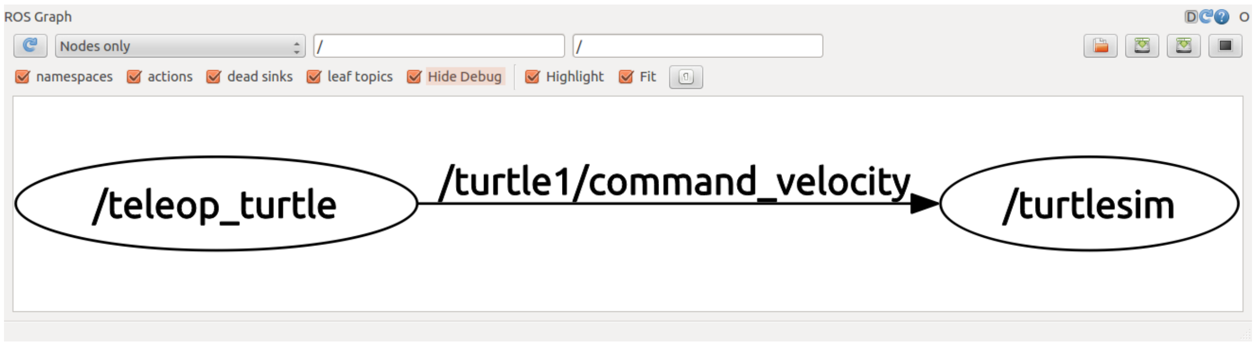
folder is the default location of the devel space, which is where your executables and libraries go before you install your packages.
- There exists a fundamental difference between installing a package and running a package. Installing a package involves downloading the package (from source), building it and registering it in your ROS Workspace. By installing a package, it gets added in your ROS_PACKAGE_PATH and its executables (nodes), libraries and messages become available which means you can reference it in other packages or launch files. By running a package, we are executing one or more nodes from the installed package. By doing this, the node(s) start and begin publishing/subscribing to topics, providing services, etc.

Quick overview of Graph Concepts (network of interconnected nodes and their communication pathways):
- Nodes: Executable that uses ROS to communicate with other nodes
- Messages: ROS data type used by nodes when subscribing or publishing to a topic
- Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages
- Master: Name service for ROS (helps nodes find and communicate with each other by maintaining a registry of all active nodes, topics and services in a ROS network)
- rosout: Ros equivalent of stdout/stderr (collects and displays log messages from nodes)
- roscore: Master + rosout +parameter server (essential command in ROS that starts up the core components required for a ROS system to function. It acts as the centralized hub for communication between ROS nodes by initializing the necessary services and processes.

So roscore is the first thing that you should run when using ROS and need nodes to communicate. And then in a new terminal using rosnode you can see the currently actively running nodes.

ROS Topics:

Topic is a communication channel used by nodes to exchange messages (information) in a publish and subscribe mode. It allows different nodes in a distributed ROS system to share information with each other without requiring direct connections between them. Topics are essential to enable nodes to interact by sending and receiving messages "asynchronously".
rqt_graph creates a dynamic graph of what's going on in the system.



ROS Messages:

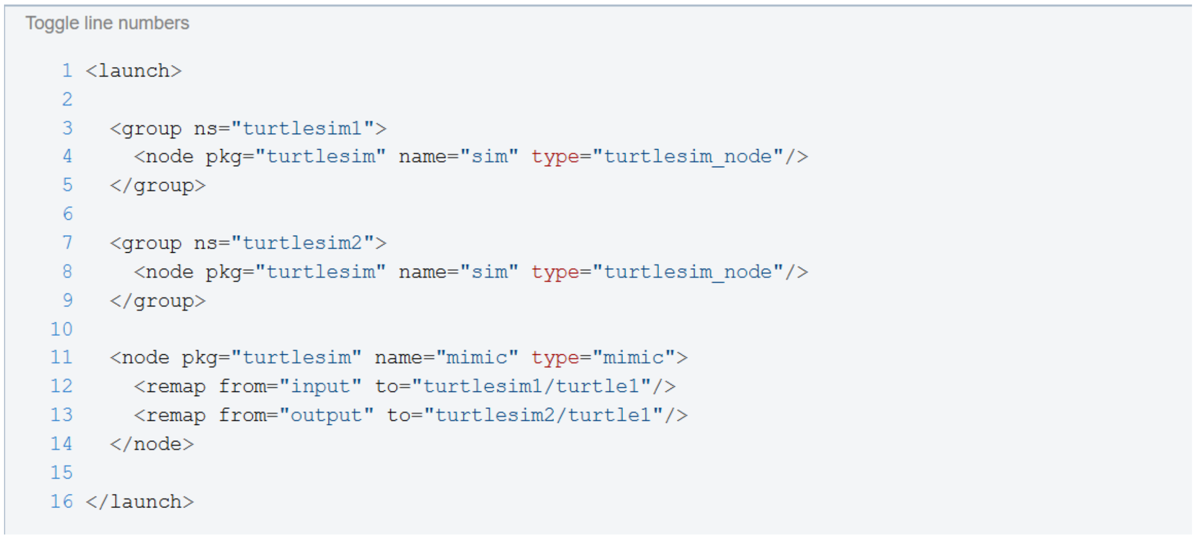
Communication on topics happens by sending ROS messages between nodes. For the publisher and subscriber to communicate, the publisher and subscriber must send and receive the same "type" of message. This means that a topic "type" is defined by the message "type" published in it.

ROS Services and Parameters:

Services and Parameters are additional mechanisms for communication and configuration, complementing the topic-based publish and subscribe system. ROS Services provide synchronous, request-response communication model. Unlike the asynchronous publish/subscribe model of the topics. Services are used when you need to perform a specific task and get an immediate response. A service consists of 2 parts: a request and a response. One node sends a request to perform a task, and another node responds after executing the task. We have a "service provider (server)" and a "client". "rosservice" has many commands that can be used on service.
Parameters are a way to store and retrieve configuration settings within a ROS system. The "Parameter Server" stores these parameters, which can be accessed and modified by any node. So, parameters are for storing static or semi-static configuration data that can be accessed or modified. "rosparam" allows you to store and manipulate data on the ROS Parameter Server (uses YAML markup language for syntax). The Parameter Server can store integers, floats, boolean, dictionaries, and lists.

rqt_console and roslaunch:

rqt_console is an important tool in ROS used for debugging and roslaunch is another important tool that is used to start many nodes at once.
rqt_console attaches to ROS's logging framework to display log outputs from nodes. It is a GUI tool in ROS used to display & filter log messages generated by nodes. It provides a way to monitor and analyze the logs that nodes generate while they run, making it an essential tool for debugging. rqt_logger_level allows us to change the verbosity level (FATAL, ERROR, WARN, INFO, DEBUG) of nodes as they run. Fatal has the highest priority and Debug has the lowest. By setting the logger level, you will get all the logging messages of that priority level or higher.
roslaunch starts nodes as defined in a launch file. The following is a sample launch file that starts a mimic turtle node:



```xml
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

Creating a ROS msg and srv:

“msg” files define the structure of messages exchanged between nodes. They specify the data types and fields of a message. For example, a pose.msg might define fields like position (x,y,z) and orientation (quaternion). msgs are just simple text files with a field type and field name per line. The field types are:

- int8, int16, int32, int64 (plus uint*)
- float32, float64
- string
- time, duration
- other msg files
- variable-length array [] and fixed-length array [C]
- special type “Header”. It contains a timestamp and coordinate frame information

Sample msg file:

```
string first_name
string last_name
uint8 age
uint32 score
```

“srv” files define service requests and responses. They are used for synchronous communication between nodes. Each .srv file has 2 parts: a request and a response. srv files are just like msg files, except they contain 2 parts: a request and a response separated by a “---” line.
Sample srv file:

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

msg files are stored in the msg directory of a package, and srv files are stored in the srv directory. After creating the message files, you need to make sure that the msg files are turned into source code for C++, python and other languages by updating the package.xml file and CMakeLists.txt. Same also holds for srv files.

Note: catkin_make is the traditional ROS build tool that compiles all packages together in a rigid workspace structure (src, build, devel), while catkin build (from catkin_tools) offers modern improvements like isolated parallel builds, per-package dependency handling, and incremental compilation for faster, more flexible development. Use catkin_make for simple projects or legacy compatibility, and catkin build for larger workspaces or optimized workflows.

Writing a simple Publisher and Subscriber node (Python):

Python executables need to be made executable by running the appropriate command and the CMakeLists.txt must be modified as well to make sure the python script gets installed properly and that it uses the right python interpreter.
The following is a simple publisher node written in python:

Toggle line numbers

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

The following is a simple subscribe node written in python:

Toggle line numbers

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

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

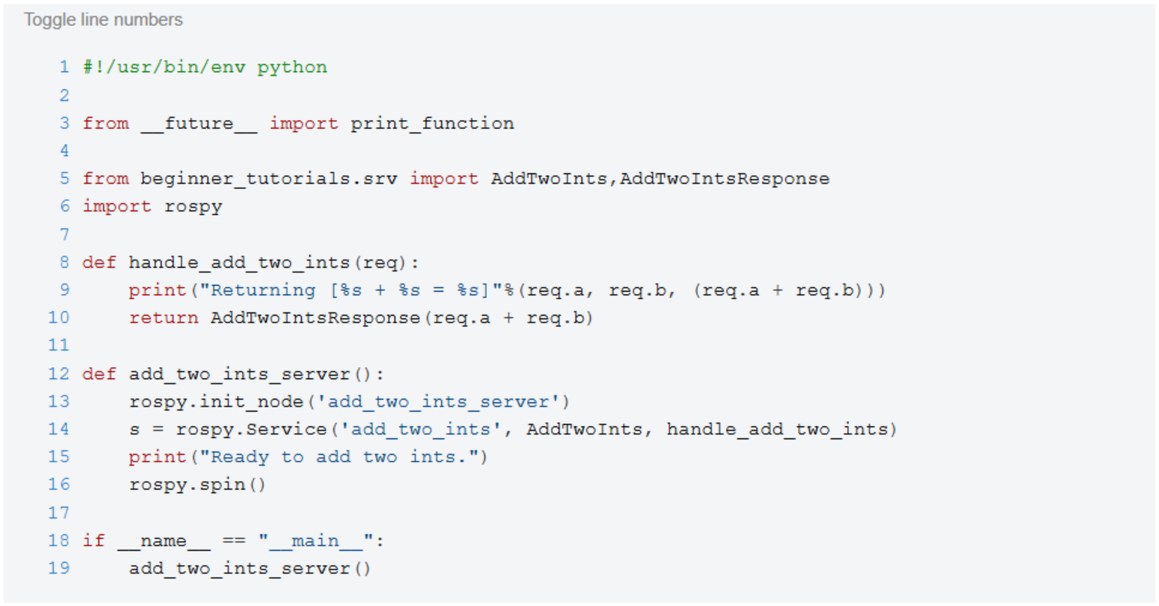
Note: Also, the subscriber node must be made executable and the CMakeLists.txt must be modified as well. Once both are created, the workspace must be built.

Note: If you are using catkin, make sure you have sourced your workspace setup.sh file (in the devel directory) after calling catkin_make but before running your nodes (applications).

Writing a simple Service and Client (Python):

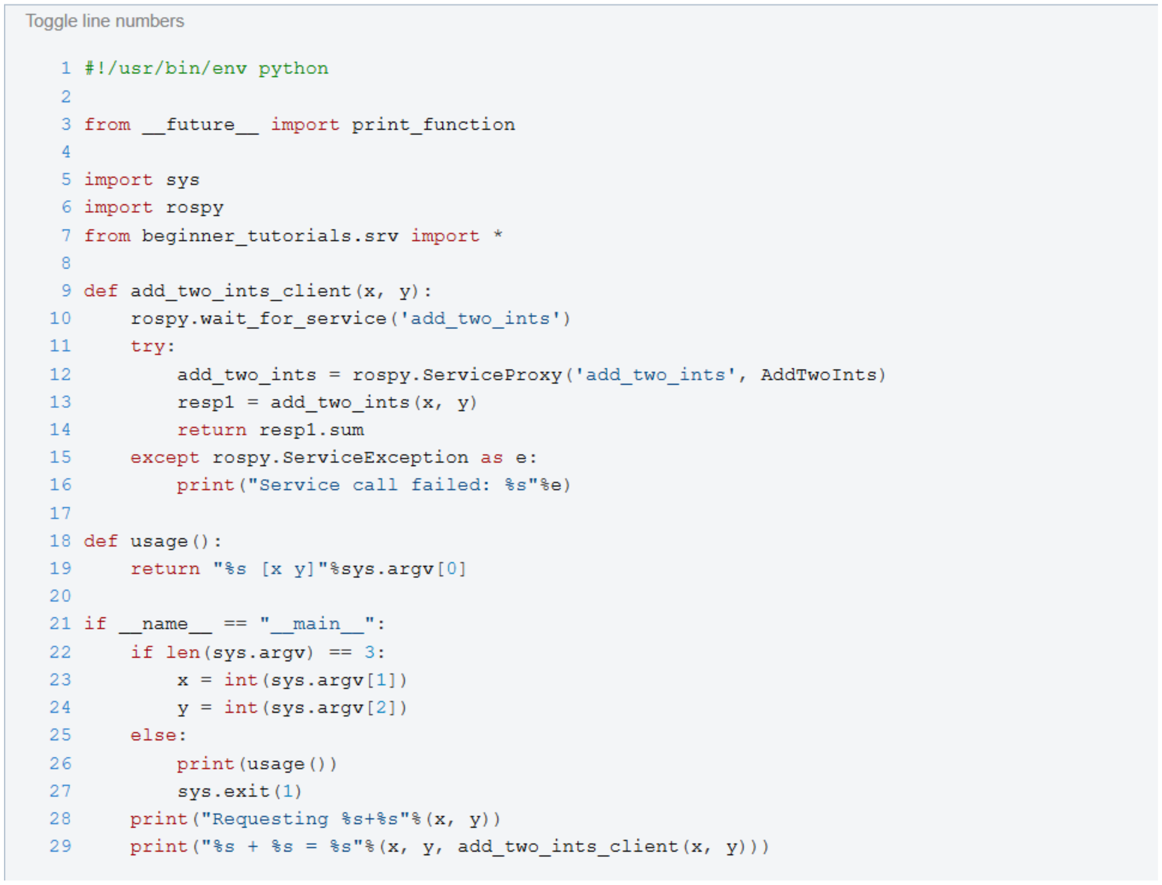
The Service and the Client written in Python, need to be created again in the scripts/ directory. And as the procedure for the Publisher and Subscriber we need to make the files executable and subsequently modify the CMakeLists.txt and also building of the catkin workspace.

The following is a simple Service:



```python
#!/usr/bin/env python

from __future__ import print_function

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

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

The following is a simple Client:



```python
#!/usr/bin/env python

from __future__ import print_function

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print(usage())
        sys.exit(1)
    print("Requesting %s+%s"%(x, y))
    print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

Recording and playing back data:

A .bag file is ROS's native data recording format—essentially a log file that stores ROS topic messages (like sensor data, robot commands, or system states) exchanged during a session. Think of it as a "black box" for robots or a "DVR for ROS systems." It captures all message traffic (topic data) in a system during a specific time period, allowing you to:

- Replay sensor data or commands exactly as they occurred
- Debug systems without needing live hardware
- Share datasets for testing or development

The first step of recording and playing back data is to record the topic data from a running ROS system. This is done by seeing the list of available topics and the list of published topics are the only message types that could potentially be recorded in the data log files, as only published messages are recorded. Then we have the option to record all the published topics or select only particular topics. The recorded bag file is created once the recording has stopped.

Once we have a bag file, we can examine it and play it back using appropriate commands.

"rosbag" commands are limited in their ability to exactly duplicate the behavior of a running system in terms of when messages are recorded and processed by rosbag record, and when messages are produced and processed when using rosbag play.

Reading messages from a bag file:

There are 2 main methods:

1. Playing back the bag file and subscribing to the required topics and looking at them in multiple terminals.
2. Using the "ros_readfbagfile script to easily extract the topics of interest (you need to download the ros_readfbagfile.py script).

"roswtf" is a tool that examines packages and the system in general to try and find problems.