Create a Workspace :

**mkdir -p (workspace\_name)/src**

**cd (workspace\_name)/**

**catkin\_init\_workspace**

**catkin\_make**

**source devel/setup.bash**

**echo $ROS\_PACKAGE\_PATH**

Create a package :

In the src folder :

**catkin\_create\_pkg (package\_name) roscpp std\_msgs (other dependencies)**

Grom Git :

In src folder :

**mkdir (package\_name)**

**cd (package\_name)**

**git init**

**git pull (link og the package to be pulles)**

Terminal Commands :

Publish a topic :

**rostopic pub /(topic\_name) std\_msgs/(data\_type) (data\_value)**

View a topic :

**rostopic echo (topic\_name)**

View a topic :

**rosnode info /(ros\_node\_name)**

View active topics and nodes :

**rosnode list (-v)**

**rostopic list (-v)**

**Recording the Published Data :** [(Tutorial)](http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data)

**mkdir ~/bagfiles**

**cd ~/bagfiles**

**rosbag record -a**

In the window running rosbag record exit with a Ctrl-C. Now examine the contents of the directory ~/bagfiles. You should see a file with a name that begins with the year, date, and time and the suffix .bag. This is the bag file that contains all topics published by any node in the time that **rosbag record** was running.

**rosbag info <your bagfile>**

To view the the .bag file without playing it back.

**rosbag play -b (Initial wait time) -s (start point time) -r (rate of replay) <your bagfile>**

Example: rosbag play -b 0.5 -s 5 -r 2 2018-10-06-05-04-57.bag

This command plays the recorded data in the bagfile. The file will wait for 0.5 sec before publishing the recorded data and will start from 5th second of the recording i.e data recorded recorded in first seconds is skipped while publishing and the data is published at 2 times the original rate.

**rosbag record -O (.bag file name) /(topic1) /(topic2)**

The previous command records data of the entire topics. This records the data of only the selected topics

**Starting with ROSWTF :** [(Tutorial)](http://wiki.ros.org/ROS/Tutorials/Getting%20started%20with%20roswtf)

**roscd rosmaster**

**roswtf**

To check the installation and error present if any. The ros master should not be running while checking

**ps -ef | grep -i rosmaster**

To check if the master (roscore) is running or inactive

**roscore**

**roscd rosmaster**

**roswtf**

Run the master and repeat the steps. It will now check for any errors in the active nodes and topics. It also displays unsubscribed topics and empty subscribed topics.

There are many other types of problems that roswtf can find. If you find yourself stumped by a build or communication issue, try running it and seeing if it can point you in the right direction.

**Creating ROS Packages Manually :** ([Tutorial](http://wiki.ros.org/ROS/Tutorials/Creating%20a%20Package%20by%20Hand))

Create A normal directory with mkdir with the required package name and navigate to that folder. We then create CMakeLists.txt and package.xml files manually.

**CMakeLists.txt :**

Basic (Minimum) Structure :

cmake\_minimum\_required(VERSION 2.8.3)

project(package name)

find\_package(catkin REQUIRED roscpp std\_msgs)

catkin\_package()

For additional features : ([link](http://wiki.ros.org/catkin/CMakeLists.txt))

**Package.xml :**

Basic (Minimum) Structure :

<package format="2">

<name>package\_name</name>

<version>1.2.4</version>

<description>

This package provides foo capability.

</description>

<maintainer email=”maintainer\_email”>Maintainer\_Name</maintainer>

<license>BSD</license>

<buildtool\_depend>catkin</buildtool\_depend>

<build\_depend>roscpp</build\_depend>

<build\_depend>std\_msgs</build\_depend>

<exec\_depend>roscpp</exec\_depend>

<exec\_depend>std\_msgs</exec\_depend>

</package>

For additional features : ([link](http://wiki.ros.org/catkin/Tutorials/CreatingPackage#ROS.2BAC8-Tutorials.2BAC8-catkin.2BAC8-CreatingPackage.Customizing_the_package.xml))

**rospack find fpackage\_name**

To find and get the exact location of an ros package. We can see that now ros recognizes this folder as a package

## **System Dependencies :** ([Tutorial](http://wiki.ros.org/ROS/Tutorials/rosdep))

**sudo rosdep init**

**rosdep update**

To initialize rosdep before using it to install dependencies

**rosdep install [package]**

To install the system dependencies using rosdep

**rosdep update**

**rosdep resolve my\_dependency\_name**

It sends pull request to rosdistro github repository.

Used when a dependency is listed that doesn't match the name of a ROS package built on the buildfarm. Then rosdep checks if there exists a rule to resolve it for the proper platform and package manager you are using.

We can check after update with resolve command.

**ROSLAUNCH :** ([Tutorial](http://wiki.ros.org/ROS/Tutorials/Roslaunch%20tips%20for%20larger%20projects))

.launch files are used to launch more than 1 node at a time and can alsp control the machine on which to launch the particular node and connections between nodes on different machines.

Machine Tags :

**<launch>**

**<machine name="c1" address="pre1" ros-root="$(env ROS\_ROOT)" ros-package-path="$(env ROS\_PACKAGE\_PATH)" default="true" />**

**<machine name="c2" address="pre2" ros-root="$(env ROS\_ROOT)" ros-package-path="$(env ROS\_PACKAGE\_PATH)" />**

**</launch>**

This code is present in .machine files which are launched using the .launch files.This file sets up a mapping between logical machine names, "c1" and "c2" in this case, and actual host names, such as "pre2". It even allows controlling the user you log in as (assuming you have the appropriate ssh credentials).

**<node pkg="amcl" type="amcl" name="amcl" machine="c1">**

Now this line of code in .xml files will launch amcl node on machine c1

NOTE : Instead of predefining the machine file, we can use an env variable and set the variable before launching the files.

## Parameters, namespaces, and yaml files:

In the .xml file of any node :

**<remap from="odom" to="pr2\_base\_odometry/odom" />**

If the required topics are being published elsewhere in any situation, we can use this remap tag to use that new topic

**<param name="controller\_frequency" value="10.0" />**

This param tag is used to set the variable values. This tag is to be used inside node tag and therefore refers to the local variable sof the node

**<rosparam file="$(find 2dnav\_pr2)/config/costmap\_common\_params.yaml" command="load" ns="global\_costmap" />**

This rosparam tag is used to load parameters and their values (if preassigned) from .yaml files. If no ns attribute is present , then by default it belongs to the parent node i.e it will be local\_costmap of the node.

raytrace\_range: 3.0

footprint: [[-0.325, -0.325], [-0.325, 0.325], [0.325, 0.325], [0.46, 0.0], [0.325, -0.325]]

inflation\_radius: 0.55

# BEGIN VOXEL STUFF

observation\_sources: base\_scan\_marking base\_scan tilt\_scan ground\_object\_cloud

The above code is an example of .yaml file.

Reusing the code :

In the . launch file if we replace :

<include file="$(find 2dnav\_pr2)/config/map\_server.xml" />

By

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(find gazebo\_worlds)/Media/materials/textures/map3.png 0.1" respawn="true" machine="c1" />

We can observe we are launching the same node but by changing the env var (gazebo\_worlds) we can load different maps.

Params Overriding :

To override a parameters value , create a new launch file and then inthe code first include the previous launch file and then use param tag to override the required parameter.Eg :

**<launch>**

**<include file="$(find 2dnav\_pr2)/move\_base/2dnav\_pr2.launch" />**

**<param name="move\_base/local\_costmap/resolution" value="0.5"/>**

**</launch>**

**Importing Custom Messages :** ([Tutorial](http://wiki.ros.org/ROS/Tutorials/DefiningCustomMessages))

We already saw how to write a .msg file including custom messages in beginner tutorials. Here we ll see how to import them :

In C++ :

#include <(.msg file name)/(the message to be imported).h>

Example :

#include <std\_msgs/String.h>

In Python :

from (.msg file name).msg import (Message to be imported)

Example :

From std\_msgs.msg import String

Also add the following to package.xml file :

<build\_depend>name\_of\_package\_containing\_custom\_msg</build\_depend>

<exec\_depend>name\_of\_package\_containing\_custom\_msg</exec\_depend>

If using C++(for building nodes) also add :

<depend>std\_msgs</depend>

<depend>sensor\_msgs</depend>

(similarly whatever packages are required)

Also add the following to CMakeLists file :

add\_dependencies(your\_program ${catkin\_EXPORTED\_TARGETS})

If using C++(for building nodes) also add :

find\_package(catkin REQUIRED COMPONENTS std\_msgs sensor\_msgs)

include\_directories(include ${catkin\_INCLUDE\_DIRS})

add\_dependencies(your\_program ${catkin\_EXPORTED\_TARGETS})

add\_dependencies(your\_library ${catkin\_EXPORTED\_TARGETS})

# **Packaging your ROS project as a snap** ([Tutorial](http://wiki.ros.org/ROS/Tutorials/Packaging%20your%20ROS%20project%20as%20a%20snap))

Snaps are packages that bundle an application and its dependencies. They offer a number of features that address important concerns as one gets closer to shipping a robotic platform

**sudo snap install --classic snapcraft**

To install snap from terminal

Must include proper installation rules in CMakeLists.txt for the snap to function without any errors.

In Python :

## Install scripts

install(PROGRAMS scripts/talker.py scripts/listener.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

In C++ :

## Install talker and listener

install(TARGETS talker listener

RUNTIME DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

Now move to workspace parent directory and in terminal-

**snapcraft init**

This creates a snapcraft.yaml file , which we can edit according to the requirements.Refer the above link to know how to write .yaml file

Once done , run **snapcraft** from parent directory to create the .snap file that can be shared and contains the workspace data

**sudo snap install --dangerous (.snap file\_name).snap**

This snap is completely standalone: you could email it to someone and they'd be able to install it and run your ROS system, even if they didn't have ROS installed themselves.

For further reference on SNAP for ROS , refer : [link](https://docs.snapcraft.io/ros-applications)

# **Using a C++ class in Python** ([Tutorial](http://wiki.ros.org/ROS/Tutorials/Using%20a%20C%2B%2B%20class%20in%20Python))

The binding occurs through two wrapper classes, one in C++ and one in Python. The C++ wrapper translates input from serialized content to C++ message instances and output from C++ message instances into serialized content.

The Python wrapper translates input from Python message instances into serialized content and output from serialized content to Python message instances. The translation from Python serialized content (str) into C++ serialized content (std::string) is built in the Boost Python library.

For Wrapper Codes example , refer to above link.

In order to be able to import the class as python\_bindings\_tutorial.AddTwoInts, we import the symbols in \_\_init\_\_.py.

from python\_bindings\_tutorial.\_add\_two\_ints\_wrapper\_py import AddTwoInts (Code in \_\_init\_\_.py)

To see how to modify CmakeLists.txt , refer the above link.

Create setup.py file in the package parent directory containing :

from distutils.core import setup

from catkin\_pkg.python\_setup import generate\_distutils\_setup

# fetch values from package.xml

setup\_args = generate\_distutils\_setup(

packages=['python\_bindings\_tutorial'],

package\_dir={'': 'src'})

setup(\*\*setup\_args)

CATKIN is used to invoke this setup/py file instead of manually doing it.

## Class with NodeHandle :

A Python call to rospy.init\_node does not initialize roscpp. If roscpp is not initialized, instantiating ros::NodeHandle will lead to a fatal error. So add following lines to python script in any wrapper class

from moveit\_ros\_planning\_interface.\_moveit\_roscpp\_initializer import roscpp\_init

roscpp\_init('node\_name', [])

## Class with ROS Message :

Refer the above link to see what lines to add to wrapper class if there are ROS messages with std::vector<M> as return type.