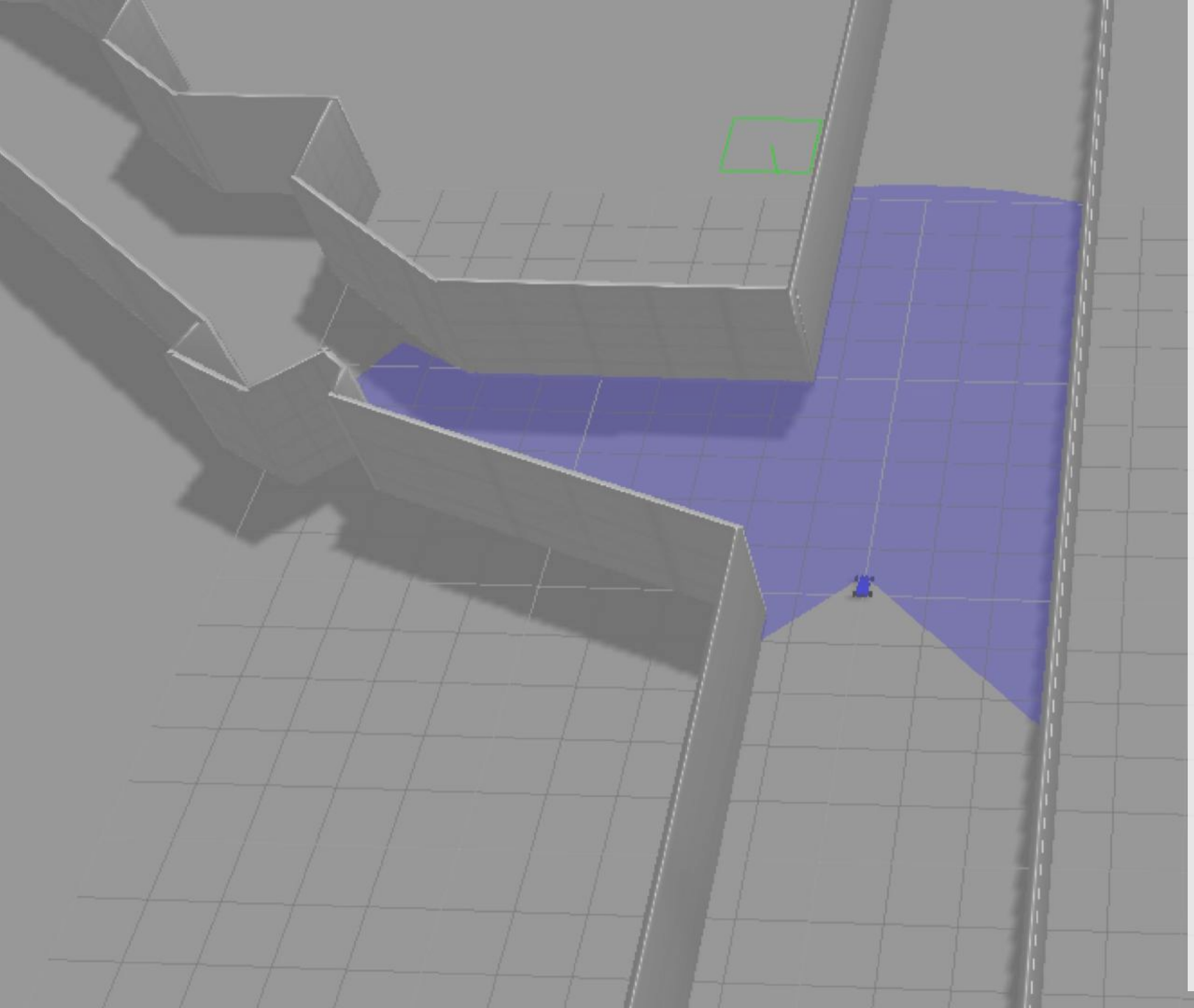

F1/10
Autonomous Racing

rospy
Publishers/Subscribers

Madhur Behl

CS 4501/SYS 4582
Spring 2019
Rice Hall 120

catkin Build System

The catkin workspace contains the following spaces

Work here

src

The *source space* contains the source code. This is where you can clone, create, and edit source code for the packages you want to build.

Don't touch

build

The *build space* is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

Don't touch

devel

The *development (devel) space* is where built targets are placed (prior to being installed).

If necessary, clean the entire build and devel space with

```
> catkin clean
```

More info
http://wiki.ros.org/catkin/workspaces

ROS Packages

- ROS software is organized into *packages*, which can contain source code, launch files, configuration files, message definitions, data, and documentation
- A package that builds up on/requires other packages (e.g. message definitions), declares these as *dependencies*

To create a new package, use

```
> catkin_create_pkg package_name
    {dependencies}
```

Separate message definition packages from other packages!

package_name

- **`config`** — Parameter files (YAML)
- **`include/`*`package_name`*** — C++ include headers
- **`launch`** — *.launch files
- **`src`** — Source files
- **`test`** — Unit/ROS tests
- **`CMakeLists.txt`** — CMake build file
- **`package.xml`** — Package information

package_name_msgs

- **`action`** — Action definitions
- **`msg`** — Message definitions
- **`srv`** — Service definitions
- **`CMakeLists.txt`** — Cmake build file
- **`package.xml`** — Package information

More info
http://wiki.ros.org/Packages

ROS Packages
package.xml

Must be included with any catkin-compliant package's root folder.

- The `package.xml` file defines the properties of the package
  - Package name
  - Version number
  - Authors
  - **Dependencies on other packages**
  - …

package.xml

```xml
<?xml version="1.0"?>
<package format="2">
  <name>ros_package_template</name>
  <version>0.1.0</version>
  <description>A template for ROS packages.</description>
  <maintainer email="pfankhauser@e…">Peter Fankhauser</maintainer>
  <license>BSD</license>
  <url type="website">https://github.com/ethz-asl/ros_best_pr…</url>
  <author email="pfankhauser@ethz.ch">Peter Fankhauser</author>

  <buildtool_depend>catkin</buildtool_depend>

  <depend>roscpp</depend>
  <depend>sensor_msgs</depend>
</package>
```

More info
http://wiki.ros.org/catkin/package.xml

ROS Packages
CMakeLists.xml

The `CMakeLists.txt` is the input to the CMakebuild system

1. Required CMake Version (`cmake_minimum_required`)
2. Package Name (`project()`)
3. Find other CMake/Catkin packages needed for build (`find_package()`)
4. Message/Service/Action Generators (`add_message_files()`, `add_service_files()`, `add_action_files()`)
5. Invoke message/service/action generation (`generate_messages()`)
6. Specify package build info export (`catkin_package()`)
7. Libraries/Executables to build (`add_library()/add_executable()/target_link_libraries()`)
8. Tests to build (`catkin_add_gtest()`)
9. Install rules (`install()`)

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_package_template)

## Use C++11
add_definitions(--std=c++11)

## Find catkin macros and libraries
find_package(catkin REQUIRED
  COMPONENTS
    roscpp
    sensor_msgs
)

…
```

More info
http://wiki.ros.org/catkin/CMakeLists.txt

ROS Client Libraries

Client Library	Language	Comments
roscpp	C++	Most widely used, high performance
rospy	Python	Good for rapid-prototyping and non-critical-path code
roslisp	LISP	Used for planning libraries
rosjava	Java	Android support
roslua	Lua	Light-weight scripting
roscs	Mono/.Net	Any Mono/.Net language
roseus	EusLisp	
PhaROS	Pharo Smalltalk	
rosR	R	Statistical programming

Experimental

Client API Commonly Used Features

Object / Feature	Description	roscpp	rospy
API root	Objects and methods for interacting with ROS	ros::NodeHandle	rospy
Parameter server client	Query and set parameter server dictionary entries	.getParam .param .searchParam .setParam	.get_param .search_param .set_param
Subscriber	Receive messages from a topic	.subscribe	.Subscriber
Publisher	Send messages to a topic	.advertise	.Publisher
Service	Serve and call remote procedures	.advertiseService .serviceClient	.Service .ServiceProxy
Timer	Periodic interrupt	.createTimer	.Timer
Logging	Output strings to rosconsole	ROS_DEBUG, ROS_INFO, ROS_WARN, etc.	.logdebug, .loginfo, .logwarn, .logerr, .logfatal
Initialization & Event Loop	Set node name, contact Master, enter main event loop	ros::init .spin	.init_node .spin
Messages	Create and extract data from ROS messages	Specifics depends on message	
		std_msgs::String	std_msgs.msg.String

rospy client library: Example

```
1 import rospy
2 from std_msgs.msg import String
3
4 pub = rospy.Publisher('topic_name', String, queue_size=10)
5 rospy.init_node('node_name')
6 r = rospy.Rate(10) # 10hz
7 while not rospy.is_shutdown():
8     pub.publish("hello world")
9     r.sleep()
```

rospy client library: Initializing your ROS Node

```
rospy.init_node('my_node_name')
```

and

```
rospy.init_node('my_node_name', anonymous=True)
```

You can only have one node in a rospy process,

so you can only call rospy.init_node()once.

Names have important properties in ROS.
Most importantly, they must be **unique**.
In cases where you don't care about unique names for a particular node, you may wish to initialize the node with an *anonymous* name.

rospy client library: Testing for shutdown

```
while not rospy.is_shutdown():
    do some work
```

and

```
... setup callbacks
rospy.spin()
```

The spin() code simply sleeps until the is_shutdown() flag is True.

There are multiple ways in which a node can receive a shutdown request, so it is important that you use one of the two methods above for ensuring your program terminates properly.

rospy client library: Registering shutdown hooks

```
rospy.on_shutdown(h)
```

Register handler to be called when rospy process begins shutdown. h is a function that takes no arguments.

```
def myhook():
  print "shutdown time!"

rospy.on_shutdown(myhook)
```

You can request a callback using rospy.on_shutdown() when your node is about to begin shutdown. This will be invoked before actual shutdown occurs, so you can perform service and parameter server calls safely.
Messages are not guaranteed to be published.

rospy client library: Message generation

package_name/msg/Foo.msg → package_name.msg.Foo

rospy takes msg files and generates Python source code for them.

To use the std_msgs/String message in your code you would use one of the following import statements:

```
import std_msgs.msg
msg = std_msgs.msg.String()
```

or

```
from std_msgs.msg import String
msg = String()
```

rospy client library: std_msgs

http://wiki.ros.org/std_msgs

ROS.org

About | Support | Discussion Forum | Service Status | Q&A answers.ros.org

Search: Submit

Documentation Browse Software News Download

std_msgs

electric | fuerte | groovy | hydro | indigo | jade | kinetic | lunar | melodic

Documentation Status

Package Summary

✔ Released ✔ Continuous integration ✔ Documented

Standard ROS Messages including common message types representing primitive data types and other basic message constructs, such as multiarrays. For common, generic robot-specific message types, please see common_msgs.

- Maintainer status: maintained
- Maintainer: Tully Foote <tfoote AT osrfoundation DOT org>
- Author: Morgan Quigley <mquigley AT cs.stanford DOT edu>, Ken Conley <kwc AT willowgarage DOT com>, Jeremy Leibs <leibs AT willowgarage DOT com>
- License: BSD
- Bug / feature tracker: https://github.com/ros/std_msgs/issues
- Source: git https://github.com/ros/std_msgs.git (branch: groovy-devel)

Package Links

Code API
Msg API
FAQ
Change List
Reviews

Dependencies (3)
Used by (14)
Jenkins jobs (10)

Wiki

Distributions
ROS/Installation
ROS/Tutorials
RecentChanges
std_msgs

Page

Immutable Page
Info
Attachments
More Actions:

User

Login

rospy client library: std_msgs

std_msgs/String Message

File: std_msgs/String.msg

Raw Message Definition

```
string data
```

Compact Message Definition

```
string data
```

2. ROS Message Types

ROS Message Types

- Bool
- Byte
- ByteMultiArray
- Char
- ColorRGBA
- Duration
- Empty
- Float32
- Float32MultiArray
- Float64
- Float64MultiArray
- Header
- Int16
- Int16MultiArray
- Int32
- Int32MultiArray
- Int64
- Int64MultiArray
- Int8
- Int8MultiArray
- MultiArrayDimension
- MultiArrayLayout
- String
- Time
- UInt16
- UInt16MultiArray
- UInt32
- UInt32MultiArray
- UInt64
- UInt64MultiArray
- UInt8
- UInt8MultiArray

geometry_msgs/Twist Message

File: geometry_msgs/Twist.msg

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
Vector3  angular
```

Compact Message Definition

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

rospy client library: Message initialization

No arguments

```
msg = std_msgs.msg.String()
msg.data = "hello world"
```

In the no-arguments style you instantiate an empty Message and populate the fields you wish to initialize.

rospy client library: Message initialization

In-order arguments (*args):

std_msgs.msg.String only has a single string field

```
msg = std_msgs.msg.String("hello world")
```

`std_msgs.msg.ColorRGBA` has four fields (r, g, b, a), so we could call:

```
msg = std_msgs.msg.ColorRGBA(255.0, 255.0, 255.0, 128.0)
```

A new Message instance will be created with the arguments provided, in order. The argument order is the same as the order of the fields in the Message, and you must provide a value for all of the fields.

rospy client library: Message initialization

Keyword arguments (kwds)**

```
msg = std_msgs.msg.String(data="hello world")
```

std_msgs.msg.ColorRGBA has four fields (r, g, b, a), so we could call:

```
msg = std_msgs.msg.ColorRGBA(b=255)
```

b=255 and the rest of the fields set to 0.0.

You only initialize the fields that you wish to provide values for.
The rest receive default values

rospy client library: Message initializations

Keyword arguments (kwds)**

```
msg = std_msgs.msg.String(data="hello world")
```

- Resilient to many types of msg changes
- Concise

In-order arguments (*args):

```
msg = std_msgs.msg.String("hello world")
```

- See entire message in code
- Track changes

No arguments

```
msg = std_msgs.msg.String()
msg.data = "hello world"
```

- More lines of code
- Useful when message default values are embedded.

rospy client library: Publishing to a topic

Create a handle to publish messages to a topic using the rospy.Publisher class

```
pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
pub.publish(std_msgs.msg.String("foo"))
```

You can then call publish() on that handle to publish a message

rospy client library: rospy.Publisher initilization

rospy.Publisher(topic_name, msg_class, queue_size)

```
pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
```

`subscriber_listener=rospy.SubscribeListener`

Receive callbacks via a rospy.SubscribeListener instance when new subscribers connect and disconnect.

`latch=False`

When a connection is latched, the last message published is saved and sent to any future subscribers that connect.

rospy client library: Publisher.publish()

Explicit style

You create your own Message instance and pass it to publish

```
pub.publish(std_msgs.msg.String("hello world"))
```

rospy client library: Publisher.publish()

Implicit style with in-order arguments

A new Message instance will be created with the arguments provided, in order. The argument order is the same as the order of the fields in the Message.
You must provide a value for all of the fields.

example, `std_msgs.msg.String` only has a single `string` field, so you can call:

```
pub.publish("hello world")
```

`std_msgs.msg.ColorRGBA` has four fields (`r`, `g`, `b`, `a`), so we could call:

```
pub.publish(255.0, 255.0, 255.0, 128.0)
```

which would create a `ColorRGBA` instance with `r`, `g`, and `b` set to 255.0 and `a` set to 128.0.

rospy client library: Publisher.publish()

Implicit style with keyword arguments

You only initialize the fields that you wish to provide values for. The rest receive default values

```
pub.publish(data="hello world")
```

`std_msgs.msg.ColorRGBA` has four fields (`r`, `g`, `b`, `a`), so we could call:

```
pub.publish(b=255)
```

which would publish a `ColorRGBA` instance with `b=255` and the rest of the fields set to 0.0.

rospy client library: Choosing a good queue_size

If you're just sending one message at a fixed rate it is fine to use a queue size as small as the frequency of the publishing.

If you are sending multiple messages in a burst you should make sure that the queue size is big enough to contain all those messages. Otherwise it is likely to lose messages.

Toggle line numbers

```
1 #!/usr/bin/env python
2 # license removed for brevity
3 import rospy
4 from std_msgs.msg import String
5
6 def talker():
7     pub = rospy.Publisher('chatter', String, queue_size=10)
8     rospy.init_node('talker', anonymous=True)
9     rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" % rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```

makes sure your script is executed as a Python script.

The std_msgs.msg import is so that we can reuse the std_msgs/String message type

publishing to the chatter topic using the message type String

tells rospy the name of your node

creates a Rate object rate.

checking the rospy.is_shutdown() flag

Create the message

the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout

publishes a string to our chatter topic

sleeps just long enough to maintain the desired rate through the loop.

Python __main__ check

This catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed

rospy client library: Subscribing to a topic

```
rospy.Subscriber(topic_name, msg_class, callback_function)
```

```
1  import rospy
2  from std_msgs.msg import String
3
4  def callback(data):
5      rospy.loginfo("I heard %s",data.data)
6
7  def listener():
8      rospy.init_node('node_name')
9      rospy.Subscriber("chatter", String, callback)
10     # spin() simply keeps python from exiting until this node is stopped
11     rospy.spin()
```

```python
#!/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()
```

Same as before (publisher)

callback is invoked with the message as the first argument.

tells rospy the name of your node

declares that your node subscribes to the chatter topic which is of type std_msgs.msgs.String

rospy.spin() simply keeps your node from exiting until the node has been shutdown

- To create a new package, navigate to your workspace and then use the **catkin_create_pkg** utility.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg beginner_tutorials std_msgs
rospy roscpp
```

- To build the new created package do:

```
$ cd ~/catkin_ws
$ catkin_make
$ . ~/catkin_ws/devel/setup.bash
```

Writing a publisher node

- Create a **src** folder inside your *beginner_tutorials* package. And create a file named **talker.py** in it.

```
$ cd ~/catkin_ws/src/beginner_tutorials
$ mkdir src // if not exists
$ cd src
$ touch talker.py
$ chmod +x talker.py
```

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

Writing a subscriber node

- Create a file named **listener.py** in the *beginner_tutorials/src* folder.

```
$ cd ~/catkin_ws/src/beginner_tutorials/src
$ touch listener.py
$ chmod +x listener.py
```

```python
#!/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():
    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()
```

Build and run

- To build your code do:

```
$ cd ~/catkin_ws
$ catkin_make
```

- To run it execute these three lines in three different terminals:

```
$ roscore
-----------------------------------------------
$ rosrun beginner_tutorials talker.py
-----------------------------------------------
$ rosrun beginner_tutorials listener.py
```

- To see the publis/subscriber connection do:

```
$ rosru rqt_graph rqt_graph
```