# Robotics and Intelligent Systems Lab

## Week 4

Francesco Maurelli

Fall 2022

# Recap

- `rostopic`
  - `list, info, echo, type, pub, hz`

# Recap

- rostopic
  - list, info, echo, type, pub, hz
- rosmsg
  - show(info), list, package

# Recap

- `rostopic`
  - `list, info, echo, type, pub, hz`
- `rosmsg`
  - `show(info), list, package`
- `rqt_graph`
  - graphical representation of nodes and topics

# Recap

- `rostopic`
  - `list`, `info`, `echo`, `type`, `pub`, `hz`
- `rosmsg`
  - `show(info)`, `list`, `package`
- `rqt_graph`
  - graphical representation of nodes and topics
- `rqt_plot`
  - continuous stream plotting of data published on topics

# Recap

- `rostopic`
  - `list, info, echo, type, pub, hz`
- `rosmsg`
  - `show(info), list, package`
- `rqt_graph`
  - graphical representation of nodes and topics
- `rqt_plot`
  - continuous stream plotting of data published on topics
- `rossrv`
  - `list, show(info), type, call`

# Recap

- `rostopic`
  - `list, info, echo, type, pub, hz`
- `rosmsg`
  - `show(info), list, package`
- `rqt_graph`
  - graphical representation of nodes and topics
- `rqt_plot`
  - continuous stream plotting of data published on topics
- `rossrv`
  - `list, show(info), type, call`
- `rosparam`
  - `set, get, dump, load`

# Launch files

# Launchfiles

- For non-trivial programs, manually running nodes using rosrun becomes tedious and hard to reproduce.
- Launchfiles are used to start groups of nodes and pass parameters to them.
- Enables full configuration of running nodes
  - Which nodes are run
  - What parameters they use
  - What topics are named and more!
- XML-based .launch file
- Placed in `<package_folder>/launch`

# Writing a Launchfile I

- The Launchfile format relies on XML, where some of the basic elements are
  - `<launch>`: the root element, every launchfile starts with this
  - `<node>`: Used to run a node.
  - `<include>`: Lets you include a launchfile in another launchfile
  - `<remap>`: Used for Remapping arguments
  - `<param>`: Sets an individual parameter on the parameter server
  - `<rosparam>`: Controls groups of parameters using .yaml files.
  - `<group>`: Groups nodes together and allows you to easily apply common settings
  - `<arg>`: Used to specify arguments which can be used to change the behavior of the launchfile from the command-line

# Writing a Launchfile II

- `<launch>`: Root element of the launch file
  - every launchfile starts with `<launch>` and ends with `</launch>`
- `<node>`: Used to run a node.

```
<node name=".." pkg=".." type=".." output=".."/>
```

  - `name`: Name of the node
  - `pkg`: Package containing the node
  - `type`: Type oy the node, there must be a corresponding executable with the same name
  - `output`: where to output log messages (`screen`:console or `log`:log file)

# Writing a Launchfile II

- `<param>`: Used to run a node.

```
<param name=".." value=".." type=".."/>
```

  - `name`: Name of the parameter
  - `value`: value of the parameter
  - `type`: Type of the parameter (int, str, etc.)

# Writing a Launchfile III

- Write a launch file in the package beginner_tutorial to start the nodes `turtlesim_node` and `turtle_teleop_key`, and changes the value of parameter `/sim/background_r`

# Writing a Launchfile III

- Write a launch file in the package beginner_tutorial to start the nodes `turtlesim_node` and `turtle_teleop_key`, and changes the value of parameter `/sim/background_r`

```
<launch>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>

  <!-- Teleop node-->
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <param name="/sim/background_r" value="255" type="int"/>
</launch>
```

# Writing a Launchfile III

- Write a launch file in the package beginner_tutorial to start the nodes `turtlesim_node` and `turtle_teleop_key`, and changes the value of parameter `/sim/background_r`

```
<launch>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>

  <!-- Teleop node-->
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <param name="/sim/background_r" value="255" type="int"/>
</launch>
```

- launch it!

```
$ roslaunch <package_name> <launchfile>
```

# Writing a Launchfile IV

- `<arg>`: creates a reusable launch file with a configurable tag

```
<arg name="arg_name" default="default_value"/>
```

  - Use argument within Launch file `$(arg arg_name)`
  - Set when launching file

```
$ roslaunch launch_file.launch arg_name:=value
```

# Writing a Launchfile III

- Modify the launch file to set the value `/sim/background_r` as an argument

# Writing a Launchfile III

- Modify the launch file to set the value `/sim/background_r` as an argument

```
<launch>
  <arg name="red_color" default="255"/>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>

  <!-- Teleop node-->
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <param name="/sim/background_r" value="$(arg red_color)" type="int"/>
</launch>
```

# Writing a Launchfile III

- Modify the launch file to set the value `/sim/background_r` as an argument

```
<launch>
  <arg name="red_color" default="255"/>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>

  <!-- Teleop node-->
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <param name="/sim/background_r" value="$(arg red_color)" type="int"/>
</launch>
```

- launch it!

# Writing a Launchfile III

- Modify the launch file to set the value `/sim/background_r` as an argument

```
<launch>
  <arg name="red_color" default="255"/>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>

  <!-- Teleop node-->
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <param name="/sim/background_r" value="$(arg red_color)" type="int"/>
</launch>
```

- launch it!

```
$roslaunch beginner_tutorials start_turtlesim_arg.launch red_color:=100
```

# Writing a Launchfile IV

- `<group>`: Groups nodes in one namespace

```
<group ns=".."> ... </group>
```

  - `ns`: Namespace for a group

- `<remap>`: remapping names of arguments

```
<remap from=".." to="..">
```

  - `from`: original name of an argument
  - `to`: remapped name

# Writing a Launchfile V

- run the node `mimic` from the `trurtlesim` package and inspect the topics it publishes and subscribes to.

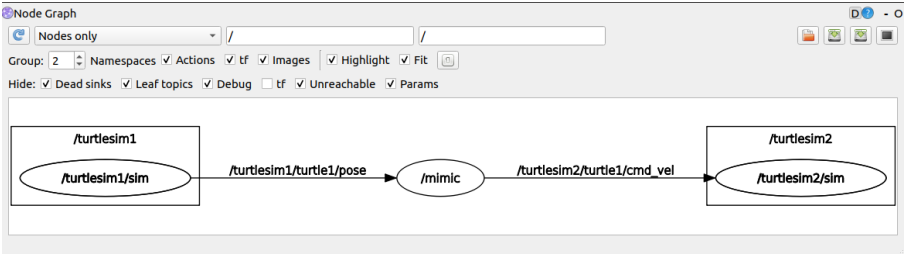
# Writing a Launchfile V

- run the node `mimic` from the `trurtlesim` package and inspect the topics it publishes and subscribes to.
- Write a launch file that
  - starts two instances of turtlesim each in a separate group.
  - starts the node `mimic`
  - remaps the output of turtle1 to input of mimic
  - remaps the output of mimic to input of turtle2

```
<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>
```

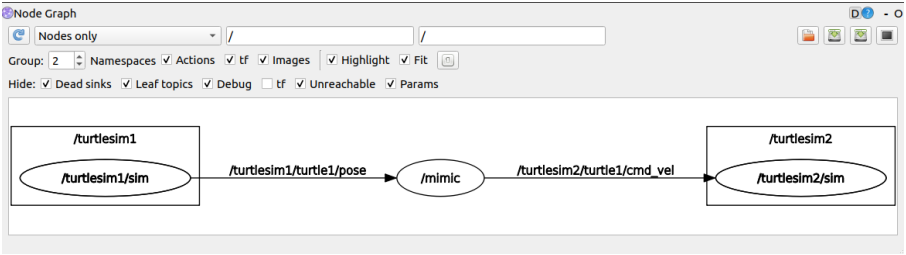
# Writing a Launchfile V

- launch it then run rqt_graph



# Writing a Launchfile V

- launch it then run rqt_graph

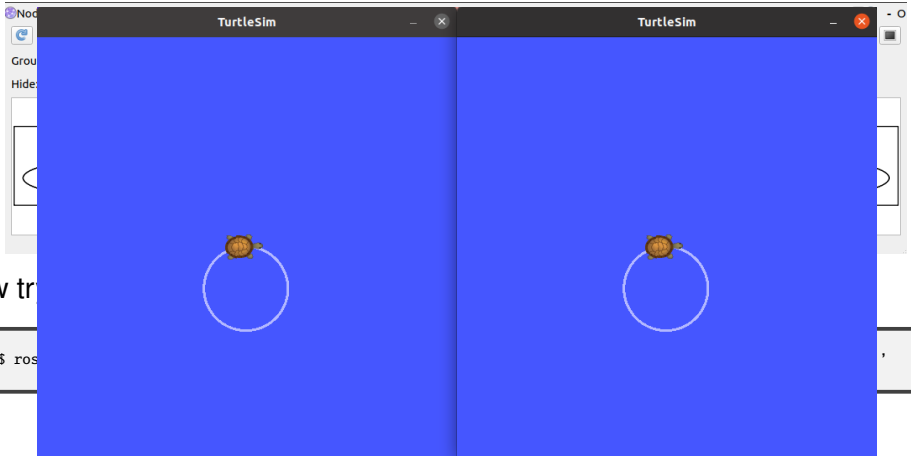



- now try publishing to turtlesim1

```
$ rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

# Writing a Launchfile V

- launch it then run rqt_graph




- now tr

```
$ ros
```

# Names in ROS

- Every item in the ROS Computation Graph has a Graph Resource Name
- You manipulate names in ROS using Remapping, which enables you to re-wire the computation graph and connect otherwise unrelated components
- ROS's name system is the source of much of its flexibility, and you will use it in every ROS program you write.
- More info:
  - https://wiki.ros.org/Names
  - https://wiki.ros.org/Remapping%20Arguments

# Creating Messages and Services

# msg and srv

## msg

- msg files are simple text files that describe the fields of a ROS message.
- They are used to generate source code for messages in different languages

## srv

- an srv file describes a service. It is composed of two parts: a request and a response.

# ROS msg I

- msg files are stored in the msg directory of a package, and srv files are stored in the srv directory.

# ROS msg I

- msg files are stored in the msg directory of a package, and srv files are stored in the srv directory.
- msgs are just simple text files with a field type and field name per line. The field types you can use are:
  - int8, int16, int32, int64 (plus uint*) float32, float64
  - string
  - time, duration
  - other msg files
  - variable-length array[] and fixed-length array[C]

# ROS msg I

- msg files are stored in the msg directory of a package, and srv files are stored in the srv directory.
- msgs are just simple text files with a field type and field name per line. The field types you can use are:
  - int8, int16, int32, int64 (plus uint*) float32, float64
  - string
  - time, duration
  - other msg files
  - variable-length array[] and fixed-length array[C]
- There is also a special type in ROS: Header, the header contains a timestamp and coordinate frame information that are commonly used in ROS.

# ROS msg II

- Inspect the `Header` msg using

```
rosmsg show Header
```

# ROS msg II

- Inspect the `Header` msg using

```
rosmsg show Header
```

```
▶ rosmsg show Header
[std_msgs/Header]:
uint32 seq
time stamp
string frame_id
```

# ROS msg II

- Inspect the `Header` msg using

```
rosmsg show Header
```

```
▶ rosmsg show Header
[std_msgs/Header]:
uint32 seq
time stamp
string frame_id
```

- Here is an example of a msg that uses a Header, a string primitive, and two other msgs

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

# ROS msg III

- Define a new msg in the package `beginner_tutorials` that was created previously

```
$ roscd beginner_tutorials
$ mkdir msg
$ touch msg/Student.msg
```

# ROS msg III

- Define a new msg in the package `beginner_tutorials` that was created previously

```
$ roscd beginner_tutorials
$ mkdir msg
$ touch msg/Student.msg
```

- Now open the created msg file and add elements in (one per line)

# ROS msg III

- Define a new msg in the package `beginner_tutorials` that was created previously

```
$ roscd beginner_tutorials
$ mkdir msg
$ touch msg/Student.msg
```

- Now open the created msg file and add elements in (one per line)

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

# ROS msg IV

- To make sure that the text file we created is turned into source code for C++ or Python, we need to do the following:
- Open `package.xml`, and make sure these two lines are in it and uncommented:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

# ROS msg IV

- To make sure that the text file we created is turned into source code for C++ or Python, we need to do the following:
- Open `package.xml`, and make sure these two lines are in it and uncommented:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

- Now open the `CMakeLists.txt` and add the `message_generation` dependency to the `find_package` so that you can generate messages

# ROS msg IV

- To make sure that the text file we created is turned into source code for C++ or Python, we need to do the following:
- Open `package.xml`, and make sure these two lines are in it and uncommented:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

- Now open the `CMakeLists.txt` and add the `message_generation` dependency to the `find_package` so that you can generate messages

```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

# ROS msg V

- Make sure you export the message runtime dependency.

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES beginner_tutorials
  CATKIN_DEPENDS message_runtime
#  DEPENDS system_lib
)
```

# ROS msg V

- Make sure you export the message runtime dependency.

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES beginner_tutorials
  CATKIN_DEPENDS message_runtime
#  DEPENDS system_lib
)
```

- Next add the msg files you created under `add_message_files`

```
## Generate messages in the 'msg' folder
add_message_files(
   FILES
   Student.msg
)
```

# ROS msg V

- Make sure you export the message runtime dependency.

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES beginner_tutorials
  CATKIN_DEPENDS message_runtime
#  DEPENDS system_lib
)
```

- Next add the msg files you created under `add_message_files`

```
## Generate messages in the 'msg' folder
add_message_files(
   FILES
   Student.msg
)
```

- Finally ensure the `generate_messages()` function is called

```
## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

# ROS msg VI

- To use the msg you created, build the package using `catkin build`

# ROS msg VI

- To use the msg you created, build the package using `catkin build`
- You can now inspect the generated msg using

```
$ rosmsg show beginner_tutorials/Student
string first_name
string last_name
uint8 age
uint32 score
```

# ROS msg VI

- To use the msg you created, build the package using `catkin build`
- You can now inspect the generated msg using

```
$ rosmsg show beginner_tutorials/Student
string first_name
string last_name
uint8 age
uint32 score
```

- If you forgot which package defines a certain message

```
$ rosmsg show Student
[beginner_tutorials/Student]:
string first_name
...
```

# ROS srv I

- srv files are just like msg files, except they contain two parts: a request and a response. The two parts are separated by a '---' line.
- srv files are stored in a directory called *srv/*
- Here is an example of a srv file:

```
int64 A
int64 B
---
int64 Sum
```

- A and B are the request, and Sum is the response

# ROS srv I

- srv files are just like msg files, except they contain two parts: a request and a response. The two parts are separated by a '---' line.
- srv files are stored in a directory called *srv/*
- Here is an example of a srv file:

```
int64 A
int64 B
---
int64 Sum
```

- A and B are the request, and Sum is the response
- Create a srv form the example shown above and store in in a file called `AddTwoInts.srv`

# ROS srv II

- Create the srv files using

```
$ roscd beginner_tutorials
$ mkdir srv $ touch srv/AddTwoInts.msg
```

- Populate your srv file with the desired elements
- You need the same changes to package.xml for services as for messages
- Same changes as in messages need to be done in CMakelists.txt except for `add_message_files`
- Modify the lines at `add_service_files` as

```
add_service_files(
   FILES
   AddTwoInts.srv
)
```

# ROS msg VI

- Run `catkin build` to generate the created srv
- You can now inspect the generated srv using

```
$ rossrv show beginner_tutorials/AddTwoInts
int64 a int64 b --- int64 sum
```

# ROS msg VI

- Run `catkin build` to generate the created srv
- You can now inspect the generated srv using

```
$ rossrv show beginner_tutorials/AddTwoInts
int64 a int64 b --- int64 sum
```

- Similar to rosmsg, you can find service without specifying package name

```
$ rosmsg show AddTwoInts
[beginner_tutorials/AddTwoInts]:
int64 a
...

[rospy_tutorials/AddTwoInts]:
int64 a
...
```

# ROS msg VI

- Run `catkin build` to generate the created srv
- You can now inspect the generated srv using

```
$ rossrv show beginner_tutorials/AddTwoInts
int64 a int64 b --- int64 sum
```

- Similar to rosmsg, you can find service without specifying package name

```
$ rosmsg show AddTwoInts
[beginner_tutorials/AddTwoInts]:
int64 a
...

[rospy_tutorials/AddTwoInts]:
int64 a
...
```

- Notice that a second one exists from the rospy_tutorials package

# rospy

# rospy Introduction

- rospy is the ROS python client library which allows your python programs to interact with other ROS processes running on your system.
- rospy is written in pure python
- The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters
- rospy favors implementation speed (i.e. developer time) over runtime performance so that algorithms can be quickly prototyped and tested within ROS
- Many of the ROS tools, such as rostopic and rosservice, are built on top of rospy

# Writing the Publisher Node using rospy I

- let's create a ROS node in the pakcage `beginner_tutorials` which will continuously publish a message

```
$ roscd beginner_tutorials
$ mkdir scripts
$ touch scripts/talker.py
```

- Make the created file an executable

```
$ beginner_tutorials
$ cd scripts
$ chmod +x talker.py
```

# Writing the Publisher Node using rospy II

- The first line in a ROS node using rospy should start with

```
1 #!/usr/bin/env python
2
```

- This line makes sure your script is executed as a Python script.

# Writing the Publisher Node using rospy II

- The first line in a ROS node using rospy should start with

```
1 #!/usr/bin/env python
2
```

- This line makes sure your script is executed as a Python script.
- Next we need to import rospy, as well as the String message from std_msgs

```
3 import rospy
4 from std_msgs.msg import String
5
```

- This will allow us to use `std_msgs/String` message type for publishing

# Writing the Publisher Node using rospy III

create a publisher with the topic 'chatter' and msg type String

```
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()
```

# Writing the Publisher Node using rospy III

```
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()
```

create a publisher with the topic 'chatter' and msg type String

create a node called 'talker' and create a Rate object with a value of 10 Hz

# Writing the Publisher Node using rospy III

```
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()
```

create a publisher with the topic 'chatter' and msg type String

create a node called 'talker' and create a Rate object with a value of 10 Hz

start a loop by checking the tag is_shutdown and define a string

# Writing the Publisher Node using rospy III

```
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()
```

create a publisher with the topic 'chatter' and msg type String

create a node called 'talker' and create a Rate object with a value of 10 Hz

start a loop by checking the tag is_shutdown and define a string

print message to screen, and write it Node's log file. Publish the msg onto the topic

# Writing the Publisher Node using rospy III

```
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()
```

create a publisher with the topic 'chatter' and msg type String

create a node called 'talker' and create a Rate object with a value of 10 Hz

start a loop by checking the tag is_shutdown and define a string

print message to screen, and write it Node's log file. Publish the msg onto the topic

sleep long enough to maintain the desired rate through the loop

# Writing the Publisher Node using rospy IV

```
1 #!/usr/bin/env python
2
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
```

# Writing the Subscriber Node using rospy I

- let's create a ROS node in the pakcage `beginner_tutorials` which will continuously listens to a message

```
$ roscd beginner_tutorials
$ mkdir scripts
$ touch scripts/listener.py
```

- Make the created file an executable

```
$ beginner_tutorials
$ cd scripts
$ chmod +x listener.py
```

# Writing the Subscriber Node using rospy II

```
1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import String
4
5 def callback(data):
6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
7
8 def listener():
9
10    # In ROS, nodes are uniquely named. If two nodes with the same
11    # name are launched, the previous one is kicked off. The
12    # anonymous=True flag means that rospy will choose a unique
13    # name for our 'listener' node so that multiple listeners can
14    # run simultaneously.
15    rospy.init_node('listener', anonymous=True)
16
17    rospy.Subscriber("chatter", String, callback)
18
19    # spin() simply keeps python from exiting until this node is stopped
20    rospy.spin()
21
22 if __name__ == '__main__':
23    listener()
```

# Writing the Subscriber Node using rospy II

```
1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import String
4
5 def callback(data):
6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
7
8 def listener():
9
10    # In ROS, nodes are uniquely named. If two nodes with the same
11    # name are launched, the previous one is kicked off. The
12    # anonymous=True flag means that rospy will choose a unique
13    # name for our 'listener' node so that multiple listeners can
14    # run simultaneously.
15    rospy.init_node('listener', anonymous=True)
16
17    rospy.Subscriber("chatter", String, callback)
18
19    # spin() simply keeps python from exiting until this node is stopped
20    rospy.spin()
21
22 if __name__ == '__main__':
23    listener()
```

The standard declartion of a node name

# Writing the Subscriber Node using rospy II

```
1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import String
4
5 def callback(data):
6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
7
8 def listener():
9
10    # In ROS, nodes are uniquely named. If two nodes with the same
11    # name are launched, the previous one is kicked off. The
12    # anonymous=True flag means that rospy will choose a unique
13    # name for our 'listener' node so that multiple listeners can
14    # run simultaneously.
15    rospy.init_node('listener', anonymous=True)
16
17    rospy.Subscriber("chatter", String, callback)
18
19    # spin() simply keeps python from exiting until this node is stopped
20    rospy.spin()
21
22 if __name__ == '__main__':
23    listener()
```

The standard declartion of a node name

Subscribes to the chatter topic which is of type std_msgs.msgs.String. When new messages are received, callback is invoked with the message as the first argument

# Writing the Subscriber Node using rospy II

```
1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import String
4
5 def callback(data):
6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
7
8 def listener():
9
10    # In ROS, nodes are uniquely named. If two nodes with the same
11    # name are launched, the previous one is kicked off. The
12    # anonymous=True flag means that rospy will choose a unique
13    # name for our 'listener' node so that multiple listeners can
14    # run simultaneously.
15    rospy.init_node('listener', anonymous=True)
16
17    rospy.Subscriber("chatter", String, callback)
18
19    # spin() simply keeps python from exiting until this node is stopped
20    rospy.spin()
21
22 if __name__ == '__main__':
23    listener()
```

Callback function that takes a message as input. Here we only print message to screen, and write it Node's log file

The standard declartion of a node name

Subscribes to the chatter topic which is of type std_msgs.msgs.String. When new messages are received, callback is invoked with the message as the first argument

# Writing the Subscriber Node using rospy II

```
1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import String
4
5 def callback(data):
6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
7
8 def listener():
9
10    # In ROS, nodes are uniquely named. If two nodes with the same
11    # name are launched, the previous one is kicked off. The
12    # anonymous=True flag means that rospy will choose a unique
13    # name for our 'listener' node so that multiple listeners can
14    # run simultaneously.
15    rospy.init_node('listener', anonymous=True)
16
17    rospy.Subscriber("chatter", String, callback)
18
19    # spin() simply keeps python from exiting until this node is stopped
20    rospy.spin()
21
22 if __name__ == '__main__':
23    listener()
```

Callback function that takes a message as input. Here we only print message to screen, and write it Node's log file

The standard declartion of a node name

Subscribes to the chatter topic which is of type std_msgs.msgs.String. When new messages are received, callback is invoked with the message as the first argument

# Building Python Nodes

- open your CMakeLists.txt file and add your files under

```
catkin_install_python(PROGRAMS scripts/talker.py scripts/listener.py
   DESTINATION $CATKIN_PACKAGE_BIN_DESTINATION
)
```

- Finally go to your catkin workspace and build your pakcage

```
$ cd ~/catkin_ws
$ catkin build
```

# Inspecting Publisher & Subscriber

- Let's run both node to inspect what they are doing
- Make sure roscore is up and running
- Source your workspace environment and run the talker

```
$ rosrun beginner_tutorials talker.py
```

```
> rosrun beginner_tutorials talker.py
[INFO] [1632142191.934912]: hello world 1632142191.9346843
[INFO] [1632142192.035425]: hello world 1632142192.035042
[INFO] [1632142192.135123]: hello world 1632142192.1349468
[INFO] [1632142192.235051]: hello world 1632142192.2348762
[INFO] [1632142192.335307]: hello world 1632142192.3349996
[INFO] [1632142192.435291]: hello world 1632142192.434974
[INFO] [1632142192.535462]: hello world 1632142192.535004
[INFO] [1632142192.635211]: hello world 1632142192.6349683
[INFO] [1632142192.735102]: hello world 1632142192.734927
[INFO] [1632142192.835207]: hello world 1632142192.8349004
```

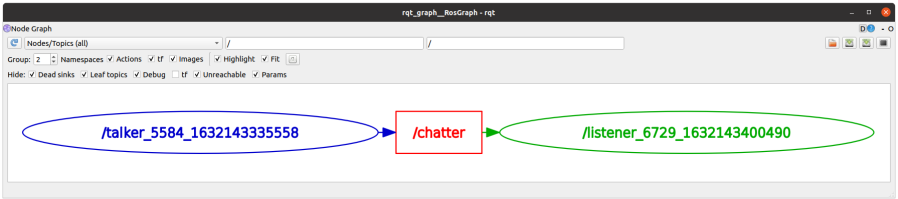
# Inspecting Publisher & Subscriber

- Now run the listener node using

```
$ rosrun beginner_tutorials listener.py
```

```
▶ rosrun beginner_tutorials listener.py
[INFO] [1632143352.117635]: /listener_5734_16321433518361 heard hello world 1632143352.1152816
[INFO] [1632143352.217623]: /listener_5734_16321433518361 heard hello world 1632143352.2151952
[INFO] [1632143352.320448]: /listener_5734_16321433518361 heard hello world 1632143352.3155568
[INFO] [1632143352.420459]: /listener_5734_16321433518361 heard hello world 1632143352.4155786
[INFO] [1632143352.520290]: /listener_5734_16321433518361 heard hello world 1632143352.515331
[INFO] [1632143352.617620]: /listener_5734_16321433518361 heard hello world 1632143352.615304
[INFO] [1632143352.717577]: /listener_5734_16321433518361 heard hello world 1632143352.7151933
```

- Inspect the computation graph using rqt_graph



# Exercise

- Create a message WaveParams.msg that contains 3 elements:
  - period
  - magnitude
  - phase
- create a node that imports the created type and publishes a topic with the same message type created
- create another node that subscribes to the published topic and generates a sinusoidal signal and prints it on the screen.