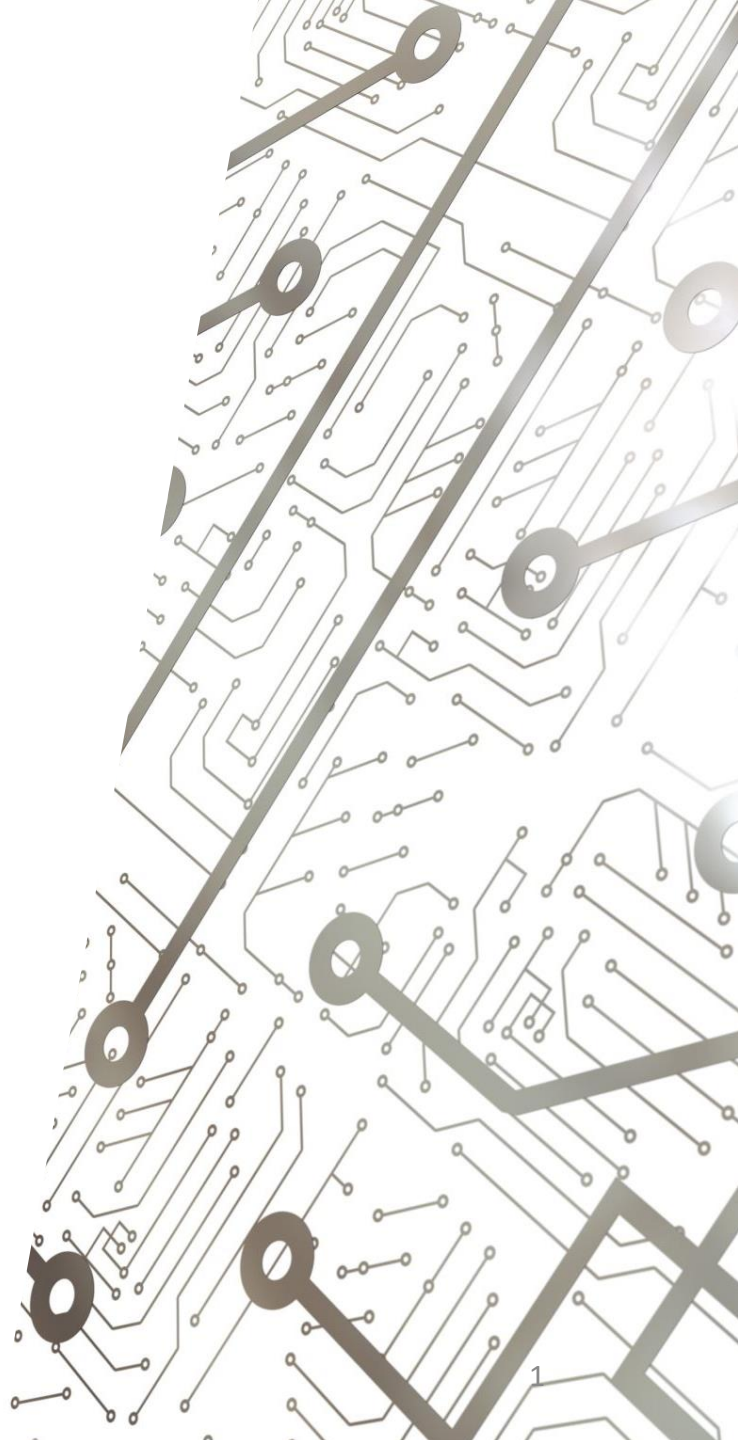
Software Lecture 3:

Nodes in Python

Jacques Cloete



Time to Create Our First Program!

- In the last lesson we created our own workspace, tutorial_ws, and created a package, tutorial_scripts, inside that workspace
- Time to create our first program! We will create a simple publisher-subscriber pair in Python
- Navigate into the source (src) folder inside our tutorial_scripts package and create a new Python script using Terminal: touch Example_Publisher.py

Creating Our First Publisher

- Once the file has been created, double-click on it to open up the editor (I actually recommend this approach instead of writing the code in Terminal – nicer interface!)

- Add the following code to the top of the file:

```
#!/usr/bin/env python3

import rospy    # Imports rospy, which provides all the basic ROS-based Python tools we need
import std_msgs.msg     # Imports the standard ROS message types
```

- The first line establishes the file as Python 3 code, the two next ones import important ROS-based tools

Creating Our First Publisher

- Now let's define a publisher:

```
# Define a publisher to a new topic 'Example_Topic', which can take Strings as messages:
pub = rospy.Publisher('Example_Topic', std_msgs.msg.String, queue_size=10)

# Note: queue_size determines the max. number of messages that can wait in the 'queue' to be published before any
further additions are discarded (until space is made in the queue)
```

- Next thing to do is set up the code as a ROS node:

```
# Set up a new ROS node to act as the publisher:
rospy.init_node('Example_Publisher')
```

- Then define the rate at which it will publish messages:

```
# Define the rate at which the publisher publishes messages:
r = rospy.Rate(1)        # 1 Hz
```

Creating Our First Publisher

- Finally, let's add the loop that will run while the node is active
- The loop will get the node to publish a new message every second:

```
while not rospy.is_shutdown():   # While this node has not been terminated:

    # Publish the message "Example_Message" to Example_Topic
    pub.publish(std_msgs.msg.String("Example_Message"))

    # Sleep for an amount of time determined by the publishing frequency (1 second here)
    r.sleep()
```

Creating Our First Publisher

- The full Python script:

```
#!/usr/bin/env python3

import rospy    # Imports rospy, which provides all the basic ROS-based Python tools we need
import std_msgs.msg     # Imports the standard ROS message types

# Define a publisher to a new topic 'Example_Topic', which can take Strings as messages:
pub = rospy.Publisher('Example_Topic', std_msgs.msg.String, queue_size=10)

# Note: queue_size determines the max. number of messages that can wait in the 'queue' to be published before any
further additions are discarded (until space is made in the queue)

# Set up a new ROS node to act as the publisher:
rospy.init_node('Example_Publisher')

# Define the rate at which the publisher publishes messages:
r = rospy.Rate(1)       # 1 Hz


while not rospy.is_shutdown():   # While this node has not been terminated:

        # Publish the message "Example_Message" to Example_Topic
        pub.publish(std_msgs.msg.String("Example_Message"))

        # Sleep for an amount of time determined by the publishing frequency (1 second here)
        r.sleep()
```

Creating Our First Subscriber

- Now that we've created our first publisher, we need to make a subscriber to go with it!
- Create another Python script using Terminal:
  touch Example_Publisher.py
- Again, add the necessary imports to the top of the script:

```
#!/usr/bin/env python3

import rospy    # Imports rospy, which provides all the basic ROS-based Python tools we need
import std_msgs.msg     # Imports the standard ROS message types
```

Creating Our First Subscriber

- When the subscriber receives a message, it calls a function to handle the message received – this is the 'callback' function
- Let's create a callback function to print the message received in Terminal:

```
# Define function 'callback' that is called when the subscriber receives a message:
def callback(message):

        # For this simple case, just print the message received in Terminal
        rospy.loginfo("Message Received: %s",message.data)
```

Creating Our First Subscriber

- When the subscriber receives a message, it calls a function to handle the message received – this is the 'callback' function
- Let's create a callback function to print the message received in Terminal:

```
# Define function 'callback' that is called when the subscriber receives a message:
def callback(message):

        # For this simple case, just print the message received in Terminal
        rospy.loginfo("Message Received: %s",message.data)
```

Creating Our First Subscriber

- Finally, add the 'main' code (this is acts as the point of execution for the program)
- It sets up the code as a ROS node, then more specifically a subscriber, then gets it to keep running until you shut it down

```
if __name__ == '__main__':      # Write your main code in here!

    # Set up a new ROS node to act as the subscriber:
    rospy.init_node('Example_Subscriber')

    # Define the node to act as a subscriber to 'Example_Topic', calling 'callback' when a message is received:
    rospy.Subscriber('Example_Topic', std_msgs.msg.String, callback)

    rospy.spin()    # Simply prevents Python from exiting until this node is terminated
```

Creating Our First Subscriber

- The full Python script:

```
#!/usr/bin/env python3

import rospy    # Imports rospy, which provides all the basic ROS-based Python tools we need
import std_msgs.msg     # Imports the standard ROS message types

# Define function 'callback' that is called when the subscriber receives a message:
def callback(message):

        # For this simple case, just print the message received in Terminal
        rospy.loginfo("Message Received: %s",message.data)

if __name__ == '__main__':      # Write your main code in here!

        # Set up a new ROS node to act as the subscriber:
        rospy.init_node('Example_Subscriber')

        # Define the node to act as a subscriber to 'Example_Topic', calling 'callback' when a message is received:
        rospy.Subscriber('Example_Topic', std_msgs.msg.String, callback)

        rospy.spin()    # Simply prevents Python from exiting until this node is terminated
```

Testing Our Code

- Before we can run our code, we need to give permissions to make it executable

- In Terminal, run the following to give permissions:
  sudo chmod a+x Example_Publisher.py Example_Subscriber.py

- Note: You will have to do this for every new script you make!

- Now, in three separate Terminals, run the following:
  1. roscore
  2. rosrun tutorials_ws Example_Publisher.py
  3. rosrun tutorials_ws Example_Subscriber.py
- Note: rosrun <package> <file> to start up a ROS program

Testing Our Code

- You should see something like this:

```
jacques@JC-Workstation:~$ rosrun tutorial_scripts Example_Subscriber.py
[INFO] [1652564980.451556]: Message Received: Example_Message
[INFO] [1652564981.451324]: Message Received: Example_Message
[INFO] [1652564982.451504]: Message Received: Example_Message
[INFO] [1652564983.451813]: Message Received: Example_Message
[INFO] [1652564984.451400]: Message Received: Example_Message
[INFO] [1652564985.451348]: Message Received: Example_Message
[INFO] [1652564986.451307]: Message Received: Example_Message
[INFO] [1652564987.451324]: Message Received: Example_Message
[INFO] [1652564988.451280]: Message Received: Example_Message
[INFO] [1652564989.451654]: Message Received: Example_Message
[INFO] [1652564990.451457]: Message Received: Example_Message
[INFO] [1652564991.451319]: Message Received: Example_Message
[INFO] [1652564992.451706]: Message Received: Example_Message
[INFO] [1652564993.451602]: Message Received: Example_Message
[INFO] [1652564994.451615]: Message Received: Example_Message
```

Testing Our Code

- If you run rostopic list, you will see \Example_Topic there!
- Running rostopic echo \Example_Topic in a 4th Terminal:

```
^Cjacques@JC-Workstation:~$ rostopic echo \Example_Topic
data: "Example_Message"
---
data: "Example_Message"
---
data: "Example_Message"
---
data: "Example_Message"
---
data: "Example_Message"
---
data: "Example_Message"
---
data: "Example_Message"
---
```

Publishers and Subscibers in Action

- The publisher node 'Example_Publisher' is repeatedly publishing message 'Example_Message' onto the topic 'Example_Topic'
- The subscriber node 'Example_Subscriber' listens into the topic and receives the messages, printing them in Terminal

Application: Square Number Factory

- Time to put all our knowledge to use with an application!
- Consider a system where a 'customer' generates a stream of numbers, requesting for a service to square them
- The numbers are squared by a separate 'factory'
- A third member, the 'receiver', collects the square numbers from the factory and prints them out

Application: Square Number Factory

- I will provide less guidance for the basics now; all the tools you need can be found in these lecture slides!
- Create three new Python scripts:

1. Square_Number_Customer.py
2. Square_Number_Factory.py
3. Square_Number_Receiver.py

Square Number Customer

- This node is just acting as a publisher
- Note: It publishes random integers between 1 and 10!

```
#!/usr/bin/env python3

import rospy
import std_msgs.msg
import random   # Used for random number generation

pub = rospy.Publisher('Square_Number_Order', std_msgs.msg.Int16, queue_size=10)

rospy.init_node('Square_Number_Customer')

r = rospy.Rate(2)       # 2 Hz


while not rospy.is_shutdown():

        order = random.randint(1,10)     # Randomly generate a new order

        rospy.loginfo("Number to be Squared: %d",order)

        pub.publish(std_msgs.msg.Int16(order))   # Publish order

        r.sleep()
```

Square Number Factory

- This node is acting as a publisher AND subscriber
- Note: It publishes its messages from INSIDE its callback! Whenever is receives a message, it publishes one in response

```
#!/usr/bin/env python3

import rospy
import std_msgs.msg

pub = rospy.Publisher('Square_Number_Processed', std_msgs.msg.Int16, queue_size=10)

def callback(message):

        rospy.loginfo("Order Received: %d",message.data)

        processed_order = message.data*message.data      # Process order (i.e. square it)

        rospy.loginfo("Processed Order: %d",processed_order)

        pub.publish(std_msgs.msg.Int16(processed_order)) # Publish processed order


if __name__ == '__main__':

        rospy.init_node('Square_Number_Factory')

        rospy.Subscriber('Square_Number_Order', std_msgs.msg.Int16, callback)     # Listen for orders

        rospy.spin()
```

Square Number Receiver

- This node is just acting as a subscriber

```
#!/usr/bin/env python3

import rospy
import std_msgs.msg

def callback(message):

        rospy.loginfo("Square Number Received: %d",message.data) # Display received square number


if __name__ == '__main__':

        rospy.init_node('Square_Number_Receiver')

        rospy.Subscriber('Square_Number_Processed', std_msgs.msg.Int16, callback) # Listen for processed orders

        rospy.spin()
```

Testing Our Program

- Try running the nodes! Remember roscore, and to give permissions to each new script!

- You should see something like this:

```
[1652567104.009430]: Number to be Squared: 7
[1652567104.509432]: Number to be Squared: 4
[1652567105.009445]: Number to be Squared: 6
[1652567105.509617]: Number to be Squared: 8
[1652567106.009449]: Number to be Squared: 2
[1652567106.509397]: Number to be Squared: 5
[1652567107.009592]: Number to be Squared: 6
[1652567107.509456]: Number to be Squared: 7
[1652567108.009485]: Number to be Squared: 8
[1652567108.509408]: Number to be Squared: 6
[1652567109.009387]: Number to be Squared: 6
[1652567109.509302]: Number to be Squared: 6
[1652567110.009583]: Number to be Squared: 10
[1652567110.509373]: Number to be Squared: 3
[1652567111.009404]: Number to be Squared: 2
[1652567111.509265]: Number to be Squared: 3
[1652567112.009409]: Number to be Squared: 10
[1652567112.509387]: Number to be Squared: 3
[1652567113.009526]: Number to be Squared: 2
[1652567113.509718]: Number to be Squared: 10
[1652567114.009459]: Number to be Squared: 7
```

```
[1652567164.018283]: Processed Order: 9
[1652567164.513888]: Order Received: 2
[1652567164.517609]: Processed Order: 4
[1652567165.015033]: Order Received: 9
[1652567165.018699]: Processed Order: 81
[1652567165.514714]: Order Received: 5
[1652567165.519235]: Processed Order: 25
[1652567166.014596]: Order Received: 10
[1652567166.018242]: Processed Order: 100
[1652567166.520172]: Order Received: 7
[1652567166.523923]: Processed Order: 49
[1652567167.014444]: Order Received: 3
[1652567167.018492]: Processed Order: 9
[1652567167.513529]: Order Received: 4
[1652567167.515694]: Processed Order: 16
[1652567168.013505]: Order Received: 2
[1652567168.016269]: Processed Order: 4
[1652567168.513361]: Order Received: 6
```

```
[1652567167.021391]: Square Number Received: 9
[1652567167.519212]: Square Number Received: 16
[1652567168.020201]: Square Number Received: 4
[1652567168.521908]: Square Number Received: 36
[1652567169.016939]: Square Number Received: 100
[1652567169.518016]: Square Number Received: 1
[1652567170.019966]: Square Number Received: 81
[1652567170.520799]: Square Number Received: 64
[1652567171.019966]: Square Number Received: 100
[1652567171.520092]: Square Number Received: 81
[1652567172.020525]: Square Number Received: 81
[1652567172.520951]: Square Number Received: 16
[1652567173.021216]: Square Number Received: 49
[1652567173.519859]: Square Number Received: 25
[1652567174.019742]: Square Number Received: 25
[1652567174.523136]: Square Number Received: 64
[1652567175.023552]: Square Number Received: 9
[1652567175.521141]: Square Number Received: 36
```

Closing Thoughts

- Note that our new program uses 3 nodes and 2 topics
- You may be wondering; why couldn't the customer and receiver be the same node?
- Well, as great as ROS messages are, they have their limits
- To repeatedly publish messages with a time-delay between means that we can't be continually listening for new messages at the same time; this is just how the while-loop works, unfortunately!

A Better Solution?

- Thankfully, ROS has us covered with ROS actions!
- These allow you to set up a client and a server; the client sends out goals while the server uses to complete a task
- Once it's done, the server sends the output back to the client! It even continually sends feedback as the task is underway!
- This way, the 'customer' will also be the 'receiver' of the output once the task is complete
- More on this next time!

Summary

We covered:

- Creating and running Python code in Ubuntu
- Creating ROS nodes with Python
- Creating our first ROS application!

Next time, we will learn about ROS Actions!