AI & Robotics

Logging in Python and with rospy

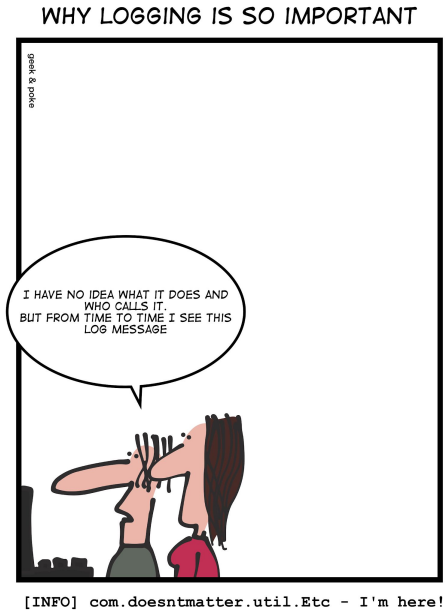
PXL AI & ROBOTICS LAB

Goals

The **junior-colleague**

- can log information using the correct level with the standard logging module
- can log information using the correct level with the ROS logging system

Logging



- Useful in all (non-trivial) applications
- At a minimum: all errors
- Major function points across the application
- It leads to an understanding of the application flow
- Include enough information to point to the line of code triggering the error, warning, . . .

Python Logging

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

[**SOURCE**]
https://docs.python.org/2/howto/logging.html

Python Logging

If your program consists of multiple modules, here's an example of how you could organize logging in it:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

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

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

[SOURCE]
https://docs.python.org/2/howto/logging.html

Python Logging

Python » English 2.7.16 Documentation » Python HOWTOs » previous | next | modules | inde

able of Contents

Logging Cookbook

Author: Vinay Sajip <vinay_sajip at red-dove dot com>

This page contains a number of recipes related to logging, which have been found useful in the past.

Using logging in multiple modules

Multiple calls to `logging.getLogger('someLogger')` return a reference to the same logger object. This is true not only within the same module, but also across modules as long as it is in the same Python interpreter process. It is true for references to the same object; additionally, application code can define and configure a parent logger in one module and create (but not configure) a child logger in a separate module, and all logger calls to the child will pass up to the parent. Here is a main module:

```
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

[**SOURCE**]
https://docs.python.org/2/howto/logging-cookbook.html

Logging: PRO-TIPs

1. Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console. **Another common way is to write them to a disk file.**

   → [M] Google its benefits ;-)

2. `tail -f filename`

ROS Verbosity Levels

DEBUG

Information that you never need to see if the system is working properly.

Examples:

"Received a message on topic X from caller Y"

"Sent 20 bytes on socket 9".

INFO

Small amounts of information that may be useful to a user.

Examples:

"Node initialized"

"Advertised on topic X with message type Y"

"New subscriber to topic X: Y"

[**SOURCE**]
http://wiki.ros.org/Verbosity%20Levels

ROS Verbosity Levels

WARN

Information that the user may find alarming, and may affect the output of the application, but is part of the expected working of the system.

Examples:

"Could not load configuration file from <path>. Using defaults."

ERROR

Something serious (but recoverable) has gone wrong.

Examples:

"Haven't received an update on topic X for 10 seconds.
Stopping robot until X continues broadcasting."

"Received unexpected NaN value in transform X. Skipping..."

[**SOURCE**]
http://wiki.ros.org/Verbosity%20Levels

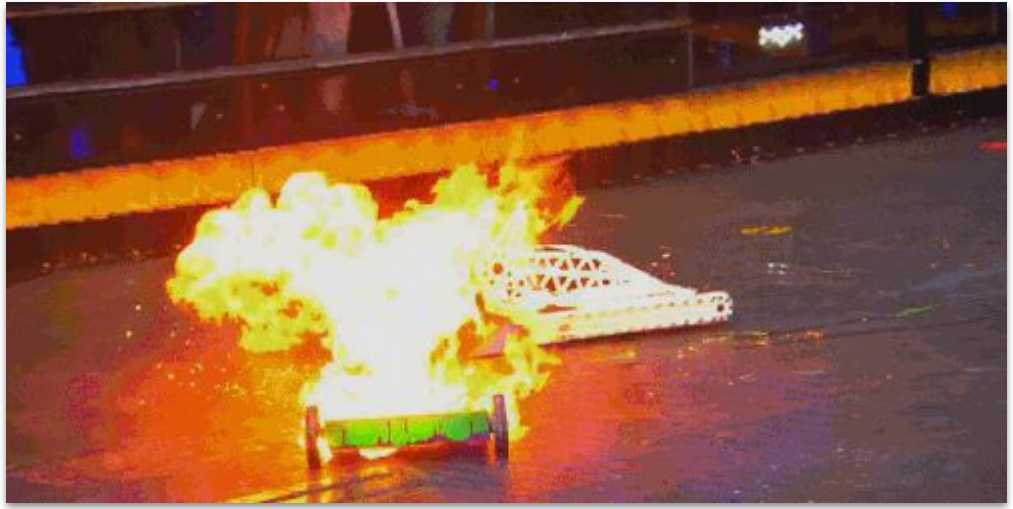
ROS Verbosity Levels

FATAL

Something unrecoverable has happened.

Examples:

"Motors have caught fire!"



[SOURCE]
http://wiki.ros.org/Verbosity%20Levels

Logging with rospy

rospy has several methods for writing log messages, all starting with "log":

```
rospy.logdebug(msg, *args, **kwargs)
rospy.loginfo(msg, *args, **kwargs)
rospy.logwarn(msg, *args, **kwargs)
rospy.logerr(msg, *args, **kwargs)
rospy.logfatal(msg, *args, **kwargs)
```

These levels have a one-to-one correspondence to ROS' logging verbosity levels.

Each `rospy.log*()` method can take in a string `msg`. If `msg` is a format string, you can pass in the arguments for the string separately, e.g.

```
rospy.logerr("%s returned the invalid value %s", other_name, other_value)
```

[SOURCE]
http://wiki.ros.org/rospy/Overview/Logging

Logging with rospy

ROS.org

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

Search: Submit

Documentation Browse Software News Download

rospy/ Overview/ Logging

rospy overview: *Initialization and Shutdown | Messages | Publishers and Subscribers | Services | Parameter Server | Logging | Names and Node Information | Time | Exceptions | tf/Overview | tf/Tutorials | Python Style Guide*

Contents

ROS has its own topic-based mechanism, called rosout for recording log messages from nodes. These log messages are human-readable string messages that convey the status of a node.

rospy has several methods for writing log messages, all starting with "log":

```
rospy.logdebug(msg, *args, **kwargs)
rospy.loginfo(msg, *args, **kwargs)
rospy.logwarn(msg, *args, **kwargs)
rospy.logerr(msg, *args, **kwargs)
rospy.logfatal(msg, *args, **kwargs)
```

These levels have a one-to-one correspondence to ROS' logging verbosity levels.

Each `rospy.log*()` method can take in a string `msg`. If `msg` is a format string, you can pass in the arguments for the string separately, e.g.

```
rospy.logerr("%s returned the invalid value %s", other_name, other_value)
```

1. Reading log messages

There are four potential places a log message may end up depending on the verbosity level:

stdout

[SOURCE]
http://wiki.ros.org/rospy/Overview/Logging

Logging with rospy

	Debug	Info	Warn	Error	Fatal
stdout		X			
stderr			X	X	X
log file	X	X	X	X	X
/rosout	o	X	X	X	X

Also note that this table is different for roscpp.

[**SOURCE**]
http://wiki.ros.org/rospy/Overview/Logging

Logging with rospy

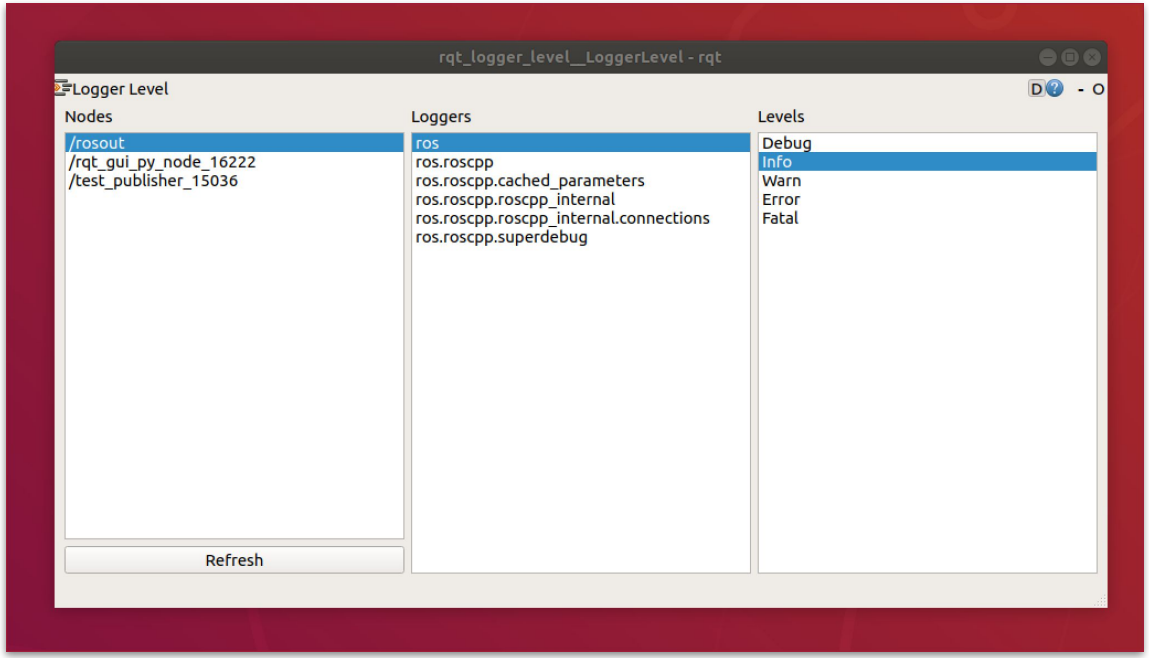
Here's a quick example with a talker Node:

Toggle line numbers

```
1    topic = 'chatter'
2    pub = rospy.Publisher(topic, String)
3    rospy.init_node('talker', anonymous=True)
4    rospy.loginfo("I will publish to the topic %s", topic)
5    while not rospy.is_shutdown():
6        str = "hello world %s"%rospy.get_time()
7        rospy.loginfo(str)
8        pub.publish(str)
9        rospy.sleep(0.1)
```

[SOURCE]
http://wiki.ros.org/rospy/Overview/Logging

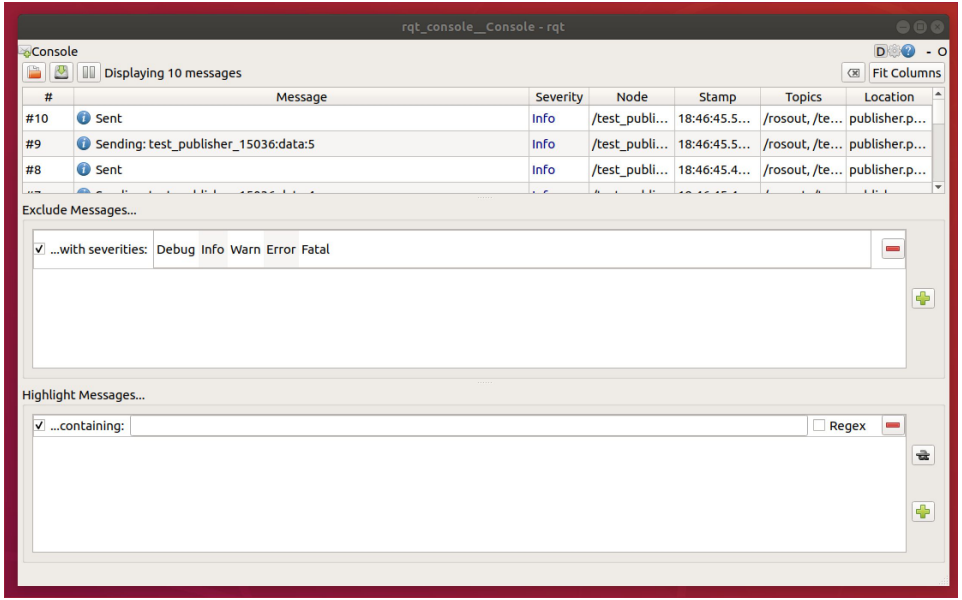
ROS Logger Level GUI




```
$ rosrun rqt_logger_level rqt_logger_level
```

[SOURCE]
http://wiki.ros.org/rqt_logger_level

ROS rqt_console




```
$ rosrun rqt_console rqt_console
```

[SOURCE]
http://wiki.ros.org/rqt_console