Summer School in Robotics
Introduction to Python, Linux & ROS

Andreas Persson, Pedro Zuidberg Dos Martires
andreas.persson@oru.se, pedro.zuidberg-dos-martires@oru.se

June 14, 2022

1 Remote Development

Throughout this summer school, you will get acquainted with programming with the Robot Operating System (ROS) in a Linux environment. This is the de facto standard development environment for roboticists around the world today. For this summer school, we have already prepared a number of *Raspberry Pi* (RPi) running the *Raspbian Linux* operating systems and with ROS pre-installed. Those RPis have each a unique id, e.g. *rp1*, which is also used as the hostname by which each RPi is identified in the local network. Similarly, each RPi has a static IP address beginning with `192.168.1.1xx`, where `xx` is then replaced by the number unique for each RPi, e.g. `192.168.1.101` for *rp1*. The same RPis will later be used as the "brain" of the robots that we will build and develop within this summer school.

Each RPi has also Visual Studio Code (VSCode) together with code-server pre-installed, which allows for remote development and accessing VSCode through a regular web browser. To remotely access the VSCode through code-server, start by connecting to the *wireless network* (login credentials found on the whiteboard). Next, we will be using port forwarding via SSH to access code-server. Forward local port `8080` to `127.0.0.1:8080` on the remote instance by running the following command on your local machine, e.g. through *Windows Terminal*:

```
ssh -N -L 8080:127.0.0.1:8080 pi@<ip-address>
```

For example, for Raspberry Pi labeled *rp1*:

```
ssh -N -L 8080:127.0.0.1:8080 pi@192.168.1.101
```

If asked for a password, the password is the default `raspberry`. At this point, you can access code-server by pointing your *web browser* to:

```
http://127.0.0.1:8080
```

1.1 Setting up VSCode for Python development

Once you have connected to an RPi via your browse, you will also want to set up VSCode for Python development. To this end go to the bottom left and click on the "manage" widget (the "cogwheel" icon). Then navigate to "Extensions", type in `Python` and install the extension.

When tackling a coding project in a team it is usually a good idea to use automatic code formatting. This ensures that everyone in the team uses the same code format, which increases code readability for everyone. One of the strictest automatic formatters for Python is the `black` formatter. Install it through the *Terminal* (found in a tab in the bottom panel of VSCode) and with the use of the `pip` Python package manager:

```
pip install black
```

To finish the setup, follow the steps in this guide.

2 Introduction to Python: Hello World!

The first program people learn to write when learning a new programming language is usually a simple Hello World program, which just prints the text "Hello World!". So, here is the Python version:

```
1  #!/usr/bin/env python
2  '''
3  hello.py
4
5  A simple Hello World program.
6  '''
7
8  if __name__ == '__main__':
9      print("Hello World!")
```

The program above simply passes the string "Hello World!" to the `print` output function. The code above makes up the source file for the program. To write this source file in VSCode, create a new file "File -> New File" (or "Ctrl+N"), and start typing. Once done typing the program, save the file "File -> Save" (or "Ctrl+S") with an appropriate name, e.g., `hello.py`. Assuming that the file has been saved with the `.py` extension, you can now run the program by clicking on the "play/run" icon in the upper right corner.

Though the program above is very basic, it nevertheless contains a few examples of Python-specific syntax. For example, triple-quoted (`'''`) multi-line comments (or docstrings). A single-line comment in Python, on the other hand, begins with a simple hashtag (#). Most notable about the Python syntax is, however, the use of *indentation*, which in Python is used to indicate *which statement belongs to which block of code*, e.g, the indented `print` function (line 9) means that this line of code belongs to the block of code of the `if` statement (line 8). *For more examples of Python-specific statements, control structures, definitions, etc., see lecture slides for Lecture 1.*

2.1 Python interpreter

In order to run a Python program, the source file must be interpreted and executed by a Python interpreter. Even when running the program by clicking on the "play/run" icon, the program is actually interpreted and executed by a Python interpreter in the Terminal (further introduced in Section 3). When the Python interpreter reads a source file, it first defines a few special variables. One such variable is the `__name__` variable. Following defining the special variables, the interpreter reads the source file line by line. Hence, the entry-point for the program above is through the `if` statement (line 8), which ensure that the `__name__` variable has been set to `'__main__'`, i.e., the case when the source file has been directly invoked by the Python interpreter.

3 Basic Linux in a Terminal

Through the *Terminal* command-line interface (found in a tab in the bottom panel of VSCode), you can manually invoke Linux commands, e.g., for running Python programs. To interpret and execute the program from previous section, go to the Terminal and type the following:

```
$ python3 hello.py
```

In order to make the program directly executable (without invoking the Python interpreter), we can simply change the access permissions of the file and give execution permissions:

```
$ chmod +x hello.py
```

The program can then be executed as a regular Linux program (it should, however, be noted that it is then the `#!/usr/bin/env python` instruction, line 1, that determines how the program is executed):

```
$ ./hello.py
```

Through the Terminal command-line interface, there is a plethora of Linux commands that conveniently can be used for many tasks; especially when developing software. For those of you with no previous experience in Linux, here is a small list of useful commands:

- `ls` : Gets a directory listing. Add the flags `-lh` and you get a list with human readable file sizes.
- `mkdir <directory name>` : Create a new directory.
- `pwd` : Prints the current directory.
- `cd <directory name>` : Change current directory (to a directoy with provided directory name).
- `cd ..` : Change current directory to the parent directory. (Note that there must be a space between `cd` and the two dots.)

- arrow keys : Press the up and down arrow keys to traverse the history of entered commands (to quickly redo a command that you typed in previously).
- tab key : Auto-completion of commands and file paths. For example, if there is a sub-directory called "workspace", you can type `cd wo[tab]` instead of typing out the full name of the directory.

A more comprehensive list of Linux commands can be found by link: https://wiki.debian.org/ShellCommands.

4 Introduction to ROS

The *Robotic Operating System* (ROS) is a networked system that aims at providing OS-like capabilities to control robots. These capabilities include, among others, hardware abstraction, interprocess communication, and package management. ROS comes with a set of tools and libraries to write, test, and deploy robotic applications. In ROS, executable programs are called *nodes*, which can communicate with each other through a *publish/subscribe* model of communications in which nodes exchange *messages* through named *topics*. For instance, a node controlling a distance sensor can read and publish distance measurement messages that other nodes can subscribe to. Nodes can live on the same computer or they can be distributed over a network of computers. In this initial practical tutorial, we will write two nodes - a *publisher* and a *subscriber* that will exchange simple text messages.

4.1 Creating a workspace and a ROS package

In ROS, nodes are structured in *packages*. ROS packages are further structured through *workspaces*. The initial first step is, therefore, to create a workspace. Through the use of `catkin`, the official build system of ROS, create a workspace by executing the following sequence of commands in the Terminal:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws
$ catkin_make
```

Within the newly created workspace, use the `catkin` build system to also create a package, here called `tutorial`, which depends upon the both existing packages `rospy` and `std_msgs`:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg tutorial rospy std_msgs
```

Also, let's create a `scripts` folder for Python source files:

```
$ cd ~/catkin_ws/src/tutorial
$ mkdir scripts
$ cd scripts
```

4.2 Writing and compiling ROS nodes

Within the `scripts` folder, create a source file, called `talker.py`, and type the source code for the *publisher* that will continually broadcast text messages on a named topic, called `chatter`:

```python
#!/usr/bin/env python
import rospy                      # Import the ROS Python library
from std_msgs.msg import String  # Import String message type from standard messages

# Main function
if __name__ == '__main__':
    try:

        # Init the connection with the ROS system
        rospy.init_node('talker', anonymous=True)

        # Create a publisher that will publish messages on topic named 'chatter'
        pub = rospy.Publisher('chatter', String, queue_size=10)

        # Start the ROS main loop, running with a frequency of 10Hz
        rate = rospy.Rate(10)
        while not rospy.is_shutdown():

            # Create and publish a String message
            str = "hello world %s" % rospy.get_time()
            rospy.loginfo(str)
            pub.publish(str)

            # Call sleep to maintain the desired rate
            rate.sleep()

    except rospy.ROSInterruptException:
        pass
```

Create another source file, called `listener.py`, and type the source code for the *subscriber* that will be receiving the text messages broadcasted by the publisher (on topic `chatter`):

```python
#!/usr/bin/env python
import rospy                      # Import the ROS Python library
from std_msgs.msg import String  # Import String message type from standard messages

# A callback function that is called every time there is new a message
# available on the topic of interest, i.e., the 'chatter' topic in this case.
def callback(msg):
   rospy.loginfo(rospy.get_caller_id() + "I heard %s", msg.data)

# Main function
if __name__ == '__main__':

    # Init the connection with the ROS system
    rospy.init_node('listener', anonymous=True)

    # Initialize a subscriber that will receive and handle message
    # though a given callback function
    rospy.Subscriber("chatter", String, callback)

    # Keeps the node spinning until the node is stopped
    rospy.spin()
```

In order to run the nodes in the ROS system, we also need to make both source files directly executable (without the need to explicitly invoking the Python interpreter):

```
$ chmod +x talker.py listener.py
```

To make sure the Python script gets installed properly, and uses the right Python interpreter, we also need to edit a file called `CMakeLists.txt`, which is a file that is generated when the package is created and that can be found in the root folder of the package (i.e, `~/catkin_ws/src/tutorial`). Add the following lines to the file `CMakeLists.txt`:

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

Finally, we are now ready to compile both the workspace and the `tutorial` package within the workspace (through the use of the `catkin` build system):

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

4.3 Running and inspecting the ROS communication

We are now almost ready to run the nodes of this `tutorial`. However, before running the nodes, we need to *source* the workspace's `setup.bash` file. This tells ROS which workspace to use and which packages that are available through the workspace:

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

Running several nodes, together with the ROS core system, requires several command-line Terminals (you can add more Terminals by the "plus" icon on the bottom panel of VSCode). In the first Terminal, start the ROS core system by command:

```
$ roscore
```

This command launch the `roscore` process which itself will start a `ROS Master` and a `rosout` logging node. The `roscore` program is necessary for the different nodes to communicate with each other. With the `roscore` running, we can now use the `rosrun` command to start individual ROS nodes. This `rosrun` command has two arguments that specify the package and the node, respectively. In a second Terminal, use the `rosrun` command to run the `talker.py` node of the `tutorial` package:

```
$ rosrun tutorial talker.py
```

In a third Terminal, use the same `rosrun` command to also run the `listener.py` node of the `tutorial` package:

```
$ rosrun tutorial listener.py
```

In the different Terminals, you should now be able to see the `loginfo` print output from the communication between both nodes. In addition, you can also check which topics that are available in the running ROS system by using yet another Terminal and invoking the command:

```
$ rostopic list
```

This command list all available topics in the running ROS system. To get more information about a specific topic, you can also specify further arguments together with the name of the topic, e.g.:

```
$ rostopic info chatter
```

Once you are done, you can shut down running nodes and programs (including `roscore`) with the "Ctrl+C" command and close Terminals with the `exit` command. *That's it for this tutorial!*