Lecture 06.02 ROS packages

packages ROS code is arranged into *packages*. Before we can describe packages, though, we need two apparatus: the ROS build system `catkin` and the ROS *workspace*.
catkin
workspaces

06.02.1 The ROS build system `catkin`

source code Most software is written by programmers as *source code* in some programming language. In this text, we write source code in Python. When the software is ready to be used, it is converted from source code into (binary) machine code and packaged up for distribution. A piece of software that
build automation utility controls this process is called a *build automation utility*. Examples include Make, Qbs, and Cabal.

catkin ROS has its own build system `catkin` built atop CMake, which is itself built on Make. It shares a name with the cluster of flowers such as that of the willow, pictured in Figure 06.3. Because we are developing in Python, we will use only a few of `catkin`'s features, some of which are introduced in the following sections.

Figure 06.3: the catkin of a willow (Didier Descouens).

06.02.2 ROS workspaces

workspaces

Workspaces are directories in which you can develop ROS code. Each project should have its own workspace, and workspaces cannot interact.

06.02.2.1 Setting up a workspace

We will now set up a workspace. Open a bash terminal.[2] Change (`cd`) to a convenient directly like your user home directory `~`. Make a new directory for your code like `code` as follows.

```
cd ~ # change directory to user home
mkdir -p code # -p creates dir only if it doesn't exist
cd code # change directory into code
```

Now make a directory `ros_ws_01` for your new workspace.

```
mkdir -p ros_ws_01
cd ros_ws_01
```

Every workspace needs a source directory `src`.

```
mkdir -p src
cd src
```

Let's inspect the tree we've made.

```
pwd # print current directory
```

```
/home/picone/code/ros_ws_01/src
```

06.02.2.2 Initializing the workspace

Now that we're in the `src` directory, we can initialize a workspace.

```
catkin_init_workspace
```

[2] We assume you have sourced the ROS distribution `setup.bash` in your `.bashrc` file so it will load when you open a new bash terminal.

```
Copying file from
↪  "/opt/ros/melodic/share/catkin/cmake/toplevel.cmake" to
↪  "/home/picone/code/ros_ws_01/src/CMakeLists.txt"
```

As we can see, this created a file `CMakeLists.txt`.

```
ls # list files and folders in current dir
```

```
CMakeLists.txt
```

Now we can finalize our new workspace using the `catkin_make` command from the workspace root.

```
cd .. # up a level to ros_ws
catkin_make
```

We have made a workspace!

06.02.2.3 Sourcing the workspace

Let's investigate the new directories in our workspace.

```
ls
```

```
build  devel  src
```

So `build` and `devel` are new! We will not make much use of the former, but the latter will include the `setup.bash` file, which we will source in order to make available to our shell the new workspace.

```
source devel/setup.bash
```

Note that this must be sourced whenever a new terminal (bash shell) is opened. Of course, you can make this automatically be sourced in your `~/.bashrc` file, but this assumes you will only be using this workspace.

06.02.3 ROS packages

packages

ROS *packages* are code directories containing certain files and organized in a certain way. Packages are usually written for specific applications, but could be applied to many others. The ROS community tends to share packages and develop them cooperatively, but there are privately held packages as well (the ROS license permits this).

06.02.3.1 Creating a new package

In this section, we will create a new package. Packages are developed in a workspace's `src` directory. Let's `cd` to that of the workspace created in the preceding section.

```
cd ~/code/ros_ws_01/src
```

We can create a new package as follows.

```
catkin_create_pkg sweet_package rospy
```

```
Created file sweet_package/package.xml
Created file sweet_package/CMakeLists.txt
Created folder sweet_package/src
Successfully created files in
↪  /home/picone/code/ros_ws_01/src/sweet_package. Please adjust
↪  the values in package.xml.
```

This created the directory `sweet_package` and populated it with `CMakeLists.txt`, `package.xml`, and the directory `src`.

```
cd sweet_package
ls
```

```
CMakeLists.txt  package.xml  src
```

The first of these has information for `catkin` and the directory `src` is initially empty – it will contain the package source code we will write. The `package.xml` file contains package metadata and should be edited.

```
cat package.xml
```

The following is an abbreviated version of the `package.xml` file contents with some editing.[3]

[3] A built-in text editor `gedit` can be used (e.g. `gedit package.xml`). However, consider installing the friendlier app Sublime Text via the Ubuntu Software app store. It will give you the command `subl` (e.g. `subl package.xml`) which you can use to easily edit many text-based files such as `xml` files.

```
<?xml version="1.0"?>
<package format="2">
<name>sweet_package</name>
<version>0.0.0</version>
<description>The sweet_package package</description>

<!-- One maintainer tag required, one per tag -->
<maintainer email="rpicone@stmartin.edu">Rico Picone</maintainer>

<!-- One license tag required, multiple allowed, one per tag -->
<license>BSD</license>

<!-- Url tags are optional, multiple allowed, one per tag -->
<url type="website">http://wiki.ros.org/sweet_package</url>

<!-- Author tags are optional, multiple allowed, one per tag -->
<author email="rpicone@stmartin.edu">Rico Picone</author>

<!-- The *depend tags are used to specify dependencies -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>rospy</build_depend>
<build_export_depend>rospy</build_export_depend>
<exec_depend>rospy</exec_depend>

<!-- The export tag contains other, unspecified, tags -->
<export>
</export>
</package>
```

depend tags

I have filled in some of this information as an example. Of special importance are the *depend tags*. When we called `catkin_create_package`, the first argument was the name of our new package `sweet_package` and the second argument was a dependency `rospy`, a ROS package which is the dependency that is required for writing nodes in Python. Note that this dependency appears in `package.xml` under multiple types of depend tags; the differences among these tags will be discussed, later. For now, note that we could have added more dependencies when we created the package by listing them after `rospy`. But we can always add more dependencies later by directly editing `package.xml`.

Now that we have a package, we can add Python code files (`.py`) that will become ROS graph nodes to the `sweet_package/src` directory. Before we do this for our own package, however, let's first learn how to run some nodes that come from pre-existing packages.